Bernhard Baltes-Götz & Paul Frischknecht

Einführung in das Programmieren mit C# 11

2023 (Rev. 240207)

Vorwort

Dieses Manuskript ist aus C# - Einführungskursen an der Universität Trier hervorgegangen, wurde aber mittlerweile aktualisiert und erheblich erweitert.

Lerninhalte und -ziele

C# ist eine von der Firma Microsoft für die .NET - Technologie entwickelte und von der internationalen IT-Normungsorganisation ECMA[1] standardisierte Programmiersprache, die auf den Vorbildern C++ und Java (siehe z. B. Baltes-Götz & Götz 2023) aufbaut, aber auch viele eigenständige Lösungen bietet.

Die .NET - Technologie hat sich als Standard für die Softwareentwicklung unter Windows etabliert und ist mittlerweile auf andere Betriebssysteme (Linux, macOS, Android, iOS) portiert worden. Im Herbst 2020 koexistierten die folgenden .NET – Implementationen:

- das nur für Windows verfügbare **.NET Framework**
- das für Linux, macOS und Windows verfügbare Open Source – Projekt **Mono**[2]
- das aus Mono entstandene Angebot der (im Jahr 2016 von Microsoft übernommenen) Firma **Xamarin** für mobile Anwendungen unter Android und iOS[3]
- das von Microsoft geförderte, für Linux, macOS und Windows verfügbare Open Source – Projekt **.NET Core**
- die von Microsoft zur Unterstützung diverser Zielgeräte (inkl. PCs mit Windows 10 und 11 und der Spielekonsole XBox) durch Store-Apps entwickelte **UWP** (*Universal Windows Platform*)

Im November 2020 ist das plattformübergreifende .NET 5 zusammen mit C# 9 erschienen, um alle bisherigen .NET - Implementationen allmählich abzulösen. Man kann .NET 5 als Weiterentwicklung von .NET Core 3.1 auffassen, wobei die Versionsnummer 4 wegen der Verwechslungsgefahr mit dem .NET Framework 4.x übersprungen wurde.[4] Unter Windows unterstützen .NET 5 und seine Nachfolger auch Anwendungen mit den etablierten GUI-Techniken (*Graphical User Interface*) WPF (*Windows Presentation Foundation*) und WinForms.

Zusammen mit der Version 6 der konsolidierten .NET - Technologie wurde im Jahr 2022 das **Multi-platform App UI** (**MAUI**) eingeführt zur Entwicklung von Programmen, die unter Android, iOS, macOS und Windows laufen, wobei die Vorarbeit von Xamarin eingeflossen ist.[5] Damit ist in .NET eine plattformübergreifende grafische Bedienoberfläche (leider ohne Linux-Unterstützung) realisierbar.

Die zu .NET gehörende Webtechnologie **Blazor WebAssembly** (**Blazor WASM**) macht es möglich, eine im Web-Browser laufende Anwendung (eine sogenannte *Single Page App, SPA*) mit C# zu erstellen.[6]

---

1 Ursprünglich stand ECMA 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/

4 https://docs.microsoft.com/en-us/dotnet/core/dotnet-five

5 https://www.heise.de/developer/meldung/Build-2020-Aus-Xamarin-Forms-wird-MAUI-4724947.html
https://devblogs.microsoft.com/dotnet/introducing-net-multi-platform-app-ui/

6 https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps

Die konsolidierte .NET - Technologie besitzt wegen der unterschiedlichen technischen Voraussetzungen auf den zu versorgenden IT-Systemen (z. B. Arbeitsplatzrechner, Server, mobile Geräte, Browser als Ausführungsumgebung) immer noch einige Variabilität. Allen Varianten gemeinsam ist eine Basisklassenbibliothek, die aber durch anwendungsspezifische Bibliotheken (z. B. für den Windows-Desktop, für Web- oder MAUI-Anwendungen) ergänzt wird. Zur Ausführung von .NET - Anwendungen (Assemblies) kommen zwei verschiedene Laufzeitumgebungen zum Einsatz:

- **CoreCLR** (*Core Common Language Runtime*) auf Arbeitsplatzrechnern und Servern unter Linux, macOS und Windows
- **Mono-Runtime** auf mobilen Geräten unter Android und iOS sowie für Blazor WASM

Im Manuskript werden C# 11 und .NET 7 meist im Rahmen von Konsolenanwendungen vorgestellt, die unter Linux, macOS und Windows laufen und die CoreCLR verwenden. Ein Kapitel ist der nur unter Windows verfügbaren GUI-Technik WPF gewidmet, die gelegentlich auch in anderen Kapiteln für Abwechslung sorgt.

Ein Vorteil der .NET - Technologie ist die freie Wahl zwischen verschiedenen Programmiersprachen, doch kann C# trotz der großen Konkurrenz als die bevorzugte .NET - Programmiersprache gelten. Schließlich wurde auch die .NET – Basisklassenbibliothek in C# entwickelt, und steht nun dank Open Source – Lizenzmodell uneingeschränkt als Vorbild und Inspirationsquelle zur Verfügung.

Der Kurs behandelt wesentliche Konzepte der objektorientierten Software-Entwicklung (z. B. Klassen, Vererbung, Polymorphie, Schnittstellen) und berücksichtigt viele Standardthemen der Programmierpraxis (z. B. Ausnahmebehandlung, Dateizugriff, grafische Bedienoberflächen, Multithreading, Netzwerk, Datenbanken).

Voraussetzungen bei den Teilnehmenden[1]

- **Programmierkenntnisse**
  Programmierkenntnisse werden *nicht* vorausgesetzt. Leser *mit* Programmiererfahrung werden sich bei den ersten Kapiteln eventuell etwas langweilen.
- **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 Erfolg (also Spaß) erleben.

Software zum Üben

Für die unverzichtbaren Übungen sollte ein Rechner mit einer aktuellen C# - Entwicklungsumgebung zur Verfügung stehen. Im Kurs wird die kostenlose Community-Variante von Microsofts Visual Studio 2022 bevorzugt. Abgesehen vom Kapitel über die GUI-Programmierung unter Windows kann aber auch das für Linux, macOS und Windows verfügbare und ebenfalls von Microsoft stammende Visual Studio Code verwendet werden. Details zum Bezug und zur Installation der Software finden Sie im Kapitel 3.

---

[1] Zur Vermeidung von sprachlichen Umständlichkeiten wird in diesem Manuskript meist die männliche Form verwendet. Die „Teilnehmenden“ sind stilistisch akzeptabel. Im nächsten Satz stünden aber die umständliche Formulierung „Leser und Leserinnen“ oder die zumindest ungewohnte Formulierung „Lesende“ zur Wahl. Trotz großer Sympathie für das Ziel einer geschlechtsneutralen Sprache scheint uns gegenwärtig die männliche Form das kleinere Übel zu sein.

Dateien zum Manuskript

Die aktuelle Version dieses Manuskripts ist zusammen mit den behandelten Beispielen und Lösungsvorschlägen zu vielen Übungsaufgaben hier zu finden:

https://bebagoe.de/csharp/

Hinweise auf Fehler und Mängel im Text werden unter der Mail-Adresse

baltes@bebagoe.de

dankbar entgegengenommen.[1]

Trier (D) und Aadorf (CH), im Dezember 2023 Bernhard Baltes-Götz & Paul Frischknecht

[1] Für aufmerksame Hinweise auf mittlerweile behobene Fehler möchten wir uns bei Soeren Borchers, Colin Christ, Marian Peters und Jens Weber herzlich bedanken.

Inhaltsverzeichnis

1 Einstieg in die objektorientierte Software-Entwicklung mit C#

In diesem Kapitel möchten wir Sie mit der Denk- und Arbeitsweise der *objektorientierten* Programmierung (in C#) vertraut machen. Wir werden als Beispiel den Kern eines Bruchrechnungs-Lernprogramms konzipieren und implementieren.

1.1 Was ist ein Computer-Programm?

Ein Computer-Programm besteht im Wesentlichen (von Medien und anderen Ressourcen einmal abgesehen) aus einer Menge von wohlgeformten und wohlgeordneten *Definitionen* und *Anweisungen* zur Bewältigung bestimmter Aufgaben. 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, CPU-Leistung) bestimmte Ausgangszustände in akzeptable Zielzustände überführen.
  Beispiel: Im Speditionsprogramm muss u. a. für jede Tour zu den (Ent-)ladestationen eine optimale Route ermittelt werden (hinsichtlich Kraftstoffverbrauch, Fahrtzeit, Mautkosten etc.).

Wir wollen präzisere und komplettere Definitionen zum komplexen Begriff eines Computer-Programms den Informatik-Lehrbüchern überlassen (siehe z. B. Goll & Heinisch 2016) 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 angenehm, aber für das Programmieren mit C# nicht besonders repräsentativ. Z. B. ist von der Objektorientierung außer einem gewissen Formalismus nichts vorhanden.
- Repräsentative C# - Programme eignen sich in der Regel wegen ihrer Länge und Komplexität (aus der Sicht eines Anfängers) nicht für den Einstieg. Beispielsweise können wir das eben zur Illustration einer realen Aufgabenstellung verwendete, aber sehr aufwändige, Speditionsverwaltungsprogramm jetzt nicht vorstellen.

Wir betrachten 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. Das Beispiel wird in sukzessive ausgebauter Form im Kurs noch oft verwendet.

1.2 Objektorientierte Analyse und Modellierung

Einer objektorientierten Programmierung geht die **objektorientierte Analyse** der Aufgabenstellung voraus mit dem Ziel einer Modellierung durch kooperierende **Klassen**. Man identifiziert per **Abstraktion** die beteiligten Kategorien von Individuen bzw. Objekten und definiert für sie jeweils eine **Klasse**. Eine solche Klasse ist gekennzeichnet durch:

- **Merkmale** (**Instanz- bzw. Klassenvariablen, Felder**)
  Die *Objekte* bzw. *Instanzen* der Klasse und auch die Klasse selbst besitzen jeweils einen **Zustand**, der durch Merkmale gekennzeichnet ist. Im Beispiel der Klasse `Bruch` ...
  - besitzt ein Objekt die Merkmale *Zähler* und *Nenner*,
  - gehört zu den Merkmalen der Klasse z. B. die Anzahl der bei einem Programmeinsatz bereits erzeugten Brüche.

  In einem aufgrund der objektorientierten Analyse entstehenden Computer-Programm landet jede Merkmalsausprägung in einer sogenannten **Variablen**. Dies ist ein benannter Speicherplatz, der Werte eines bestimmten Typs (z. B. Zahlen, Zeichenfolgen) aufnehmen kann. Variablen zum Speichern 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 oder der Klasse selbst zugeordnet. Im Beispiel der Klasse `Bruch` ...
  - hat ein Objekt z. B. die Fähigkeit zum Kürzen von Zähler und Nenner,
  - kann die Klasse z. B. über die Anzahl der bereits erzeugten Brüche informieren.

  In einem aufgrund der objektorientierten Analyse entstehenden Computer-Programm sind die Handlungskompetenzen durch sogenannte **Methoden** repräsentiert. Diese ausführbaren Programmbestandteile realisieren die oben angesprochenen Algorithmen. Die Kommunikation zwischen Klassen und Objekten besteht darin, ein Objekt oder eine 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).

Bei einer definitiv nur *einfach* zu besetzenden Rolle kann eine Klasse zum Einsatz kommen, die ausnahmsweise *nicht* zum Instanziieren (Erzeugen von Objekten) gedacht ist, aber als Akteur mitwirkt.

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 Felder zum Speichern von ganzen Zahlen (C# - Datentyp **int**) mit den folgenden Namen realisiert:

- `zaehler`
- `nenner`

Eine wichtige, auf den ersten Blick leicht zu übersehende Entscheidung der Modellierungsphase besteht darin, beim Zähler und beim Nenner eines Bruchs auch negative ganze Zahlen zu erlauben. Alternativ könnte man ...

- beim Nenner negative Werte verbieten, um folgende Beispiele auszuschließen:
$$\frac{2}{-3}, \frac{-2}{-3}$$
- beim Zähler und beim Nenner negative Werte verbieten, weil ein Bruch als Anteil aufgefasst und daher stets größer oder gleich null sein sollte.

Indem beim Zähler und beim Nenner auch negative ganze Zahlen zugelassen werden, findet sich unter den Objekten der resultierenden Klasse z. B. eine einfache Lösung für die folgende Gleichung, was irgendwann vorteilhaft sein könnte:

$$-3x = 2 \Leftrightarrow x = \frac{2}{-3}$$

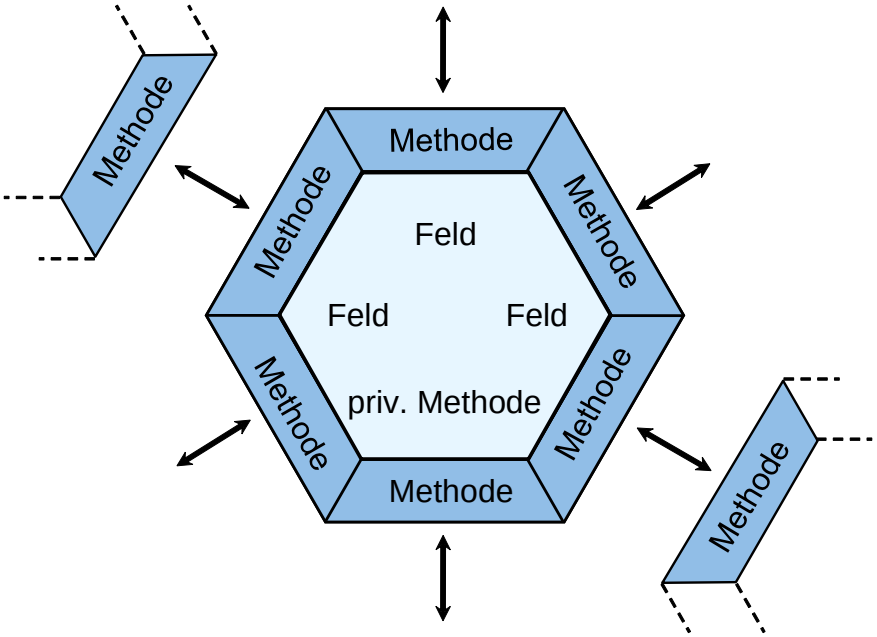
Auf das oben angedeutete *klassenbezogene* Merkmal mit der Anzahl bereits erzeugter Brüche wird vorläufig verzichtet.

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

Demgegenüber sind die **Handlungskompetenzen** (Methoden) einer Klasse in der Regel von anderen Klassen 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. Man spricht auch vom **API** (*Application Programming Interface*) einer Klasse.[1]

Die folgende, an Goll & Heinisch (2016) angelehnte Abbildung zeigt für eine Klasse ...

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



Für die Objekte einer Klasse und die Klasse selbst wird in der objektorientierten Analyse und Modellierung die Befähigung eingeplant, auf eine Reihe von **Nachrichten** mit einem bestimmten Verhalten zu reagieren. In unserem Beispiel sollten die Objekte der Klasse `Bruch` z. B. eine Methode zum Kürzen besitzen. Dann kann einem konkreten `Bruch`-Objekt durch Aufrufen dieser Methode die Nachricht zugestellt werden, Zähler und Nenner zu kürzen.

---

[1] Im Kapitel 9 wird mit dem Begriff *Interface* (alias: *Schnittstelle*) eine Liste von öffentlichen Methoden bezeichnet, wobei von einer Methode lediglich die zur Kommunikation erforderlichen Informationen vorhanden sind, aber keine Implementation. Wenn sich eine Klasse zu einem solchen Interface (verstanden als Verpflichtungserklärung) bekennt, dann muss sie alle Interface-Methoden implementieren. Der Begriff *Interface* (alias: *Schnittstelle*) wird also in zwei unterscheidbaren, aber stark verwandten Bedeutungen verwendet.

Sich unter einem Bruch ein Objekt vorzustellen, das Nachrichten empfängt und mit einem passenden Verhalten darauf reagiert, 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). Die objektorientierte Modellierung eines Anwendungsbereichs ist nicht unbedingt eine direkte Abbildung, sondern eher eine *Rekonstruktion*. Einerseits soll der Anwendungsbereich im Modell gut repräsentiert sein, andererseits soll eine möglichst stabile, gut erweiterbare und wiederverwendbare Software entstehen.

Um (Objekten aus) 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 gekapselten Merkmal ist auch der direkte *Lese*zugriff ausgeschlossen, sodass 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 gravierende Vorteile realisiert:

- **Stabilität**
  Die Merkmalsausprägungen sind vor unsinnigen und gefährlichen Zugriffen geschützt, wenn Veränderungen nur über die vom Klassendesigner sorgfältig entworfenen Methoden möglich sind. Treten trotzdem Fehler auf, dann sind diese leicht zu identifizieren, weil nur wenige Methoden verantwortlich sein können.
- **Produktivität**
  Durch Datenkapselung wird die **Modularisierung** unterstützt, sodass große Software-Systeme beherrschbar werden und zahlreiche Programmierer möglichst 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 (*Base Class Library*, BCL) (siehe Abschnitt 2.6), von denen alle C# - Programmierer regen Gebrauch machen. Auch die Klasse `Bruch` aus dem Beispielprojekt besitzt einiges Potenzial zur Wiederverwendung.

Im Vergleich zu anderen objektorientierten Programmiersprachen wie z. B. Java und C++ bietet C# mit den sogenannten **Eigenschaften** (engl.: **properties)** eine Möglichkeit, den Aufwand mit den Methoden zum Lesen oder Verändern von Merkmalsausprägungen für den Designer und vor allem für den Nutzer einer Klasse zu reduzieren. In der Klasse `Bruch` werden wir z. B. zum Feld `nenner` die *Eigenschaft* `Nenner` (großer Anfangsbuchstabe!) definieren, welche dem Nutzer der Klasse `Bruch` *Methoden* zum Lesen und Setzen des Nenners bietet, wobei dieselbe Syntax wie beim direkten Zugriff auf das Feld verwendet werden kann. Um diese Aussage zu illustrieren, greifen wir der Beschäftigung mit elementaren C# - Sprachelementen vor. In der folgenden Anweisung wird der `nenner` eines `Bruch`-Objekts mit dem Namen `b1` auf den Wert 4 gesetzt:

```
b1.Nenner = 4;
```

Während der Entwickler der Klasse `Bruch` Zugriffs*methoden* bereitzustellen hat (siehe unten), sehen die Nutzer (das sind die Entwickler *anderer* Klassen) eine öffentliche *Eigenschaft*. 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 durch die

C# - Eigenschaften unscharf zu werden scheint. Letztlich erspart eine C# - Eigenschaft wie `Nenner` dem Nutzer lediglich die Verwendung von Zugriffs*methoden*, z. B.

```
b1.SetzeNenner(4);
```

Damit kann man die C# - Eigenschaften in die Kategorie *syntactic sugar* (Mössenböck 2019, S. 3) einordnen, doch ist dieser Zucker nicht nur angenehm, sondern auch nützlich ohne jede schädliche Nebenwirkung. Wegen ihrer intensiven Nutzung in C# - Programmen ist ein Auftritt der Eigenschaften im ersten Kursbeispiel trotz des angesprochenen didaktischen Problems gerechtfertigt.

In realen (komplexeren) Programmen wird keinesfalls *jedes* gekapselte Feld über eine Eigenschaft zum Lesen und geschützten Schreiben für die Außenwelt zugänglich gemacht. Es sind auch Felder möglich, die von anderen Klassen nur gelesen oder nur verändert werden dürfen.

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

- `Nenner` (Eigenschaft zum Feld `nenner`)
  Das Objekt wird beauftragt, seinen `nenner`-Zustand mitzuteilen bzw. zu verändern. Ein direkter Zugriff auf das Feld soll fremden Klassen *nicht* erlaubt sein (Datenkapselung). Bei dieser Vorgehensweise kann ein `Bruch`-Objekt z. B. verhindern, dass sein `nenner` auf null 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 im Vergleich zu einem für andere Klassen direkt zugänglichen Feld. Sie ist aber der Einheitlichkeit und damit der Einfachheit halber angebracht und hält die Möglichkeit offen, das Merkmal `zaehler` einmal anders zu realisieren.
- `Kuerze()`
  Durch diese Methode wird das angesprochene Objekt beauftragt, `zaehler` und `nenner` zu kürzen. Welcher Algorithmus dazu benutzt wird, bleibt dem Klassendesigner überlassen. Wie das Beispiel zeigt, wird dem Namen einer Methode eine in runden Klammern eingeschlossene, eventuell leere *Parameterliste* angehängt.
- `Addiere(Bruch b)`
  Die Parameterliste dieser Methode ist nicht leer. Methodenparameter, mit denen wir uns noch ausführlich beschäftigen werden, haben einen Namen (im Beispiel: `b`) und einen Datentyp (im Beispiel: `Bruch`). Durch diese Methode wird das angesprochene Objekt beauftragt, den als Parameter übergebenen Bruch zum eigenen Wert zu addieren.
- `Frage()`
  Durch diese Methode wird das angesprochene Objekt beauftragt, `zaehler` und `nenner` beim Anwender via Konsole (Eingabeaufforderung) zu erfragen.
- `Zeige()`
  Durch diese Methode wird das angesprochene Objekt 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 Klasse `Bruch` enthält die folgenden Mitglieder:

- Felder
  `zaehler`, `nenner`
- Eigenschaften
  `Zaehler`, `Nenner`
  Bei diesen Eigenschaften handelt es sich jeweils um ein Paar von Methoden für den lesenden und den schreibenden Zugriff.
- 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, Spontanität, Kreativität und auch Emotionen besitzen. 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.

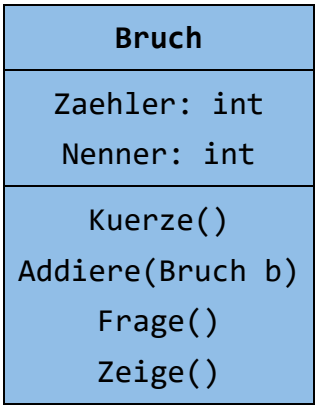
Um die durch objektorientierte Analyse gewonnene Modellierung eines Anwendungsbereichs standardisiert und übersichtlich zu beschreiben, wurde die **Unified Modeling Language** (UML) entwickelt, die bevorzugt mit Diagrammen arbeitet.[1] Eine Klasse wird durch ein Rechteck mit drei Bereichen dargestellt:

- 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) Felder angesprochen werden. Es hängt vom Adressaten eines Klassendiagramms (z. B. Entwickler-Teamkollege oder Anwender der Klasse) ab, ob man als Merkmale die Felder, die Eigenschaften oder beides angibt. Stellt man Felder und Eigenschaften getrennt dar, dann resultiert ein UML-Klassendiagramm mit *vier* Bereichen (siehe unten).
- 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 Parameter eines Methodenaufrufs (mit Spezifikationen zur gewünschten Ausführungsart bzw. mit Details der gesendeten Nachricht) den Datentyp an.

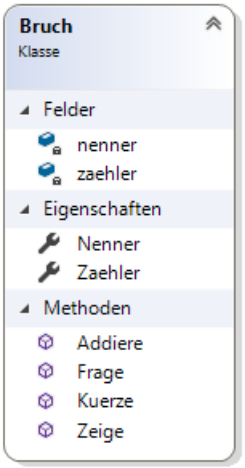
Für die Klasse `Bruch` resultiert die folgende Darstellung, wenn die Eigenschaften aus der Anwenderperspektive betrachtet werden (als Merkmale und nicht als Methodenpaare):

---

[1] Während die UML im akademischen Bereich nachdrücklich empfohlen wird, ist ihre Verwendung in der Software-Branche noch entwicklungsfähig, wie empirische Studien gezeigt haben (siehe z. B. Baltes & Diehl 2014, Petre 2013).



Die im Kurs bevorzugte Entwicklungsumgebung Visual Studio Community 2022 erstellt auf Wunsch zur Klasse `Bruch` das folgende Diagramm, wobei Felder, Eigenschaften und Methoden als Member zu sehen sind:[1]

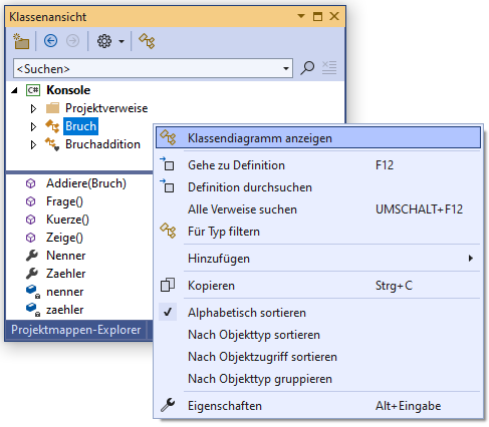


Sind bei einer Anwendung *mehrere* Klassen beteiligt, dann sind auch die *Beziehungen* zwischen den Klassen wesentliche Bestandteile des Modells. In einem UML-Klassendiagramm können u. a.

---

[1] Obwohl die Installation der Entwicklungsumgebung noch aussteht (siehe Abschnitt 3.3), beschreiben wir hier das Erstellen eines Klassendiagramms:

- Der **Klassen-Designer** muss ergänzend installiert werden (siehe Abschnitt 3.3.6.2).
- Im **Projektmappen-Explorer** muss über den Menübefehl
  Ansicht > Klassenansicht
  die **Klassenansicht** aktiviert werden.
- In der **Klassenansicht** wählt man aus dem Kontextmenü zur betroffenen Klasse das Item **Klassendiagramm anzeigen**:



die folgenden Beziehungen zwischen Klassen (bzw. zwischen den Objekten von Klassen) dargestellt werden:

- Spezialisierung bzw. Vererbung („Ist-ein - Beziehung“)
  Beispiel: Ein Lieferwagen ist ein spezielles Auto.
- Komposition („Hat - Beziehung“)
  Beispiel: Ein Auto hat einen Motor.
  Mit der Komposition werden wir uns im Abschnitt 5.9 näher beschäftigen.
- Assoziation („Kennt - Beziehung“)
  Beispiel: Ein (intelligentes, autonomes) Auto kennt eine Liste von Parkplätzen.

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

1.3 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 sogenannte **Quellcode**, der am besten in einer Textdatei namens **Bruch.cs** untergebracht wird.

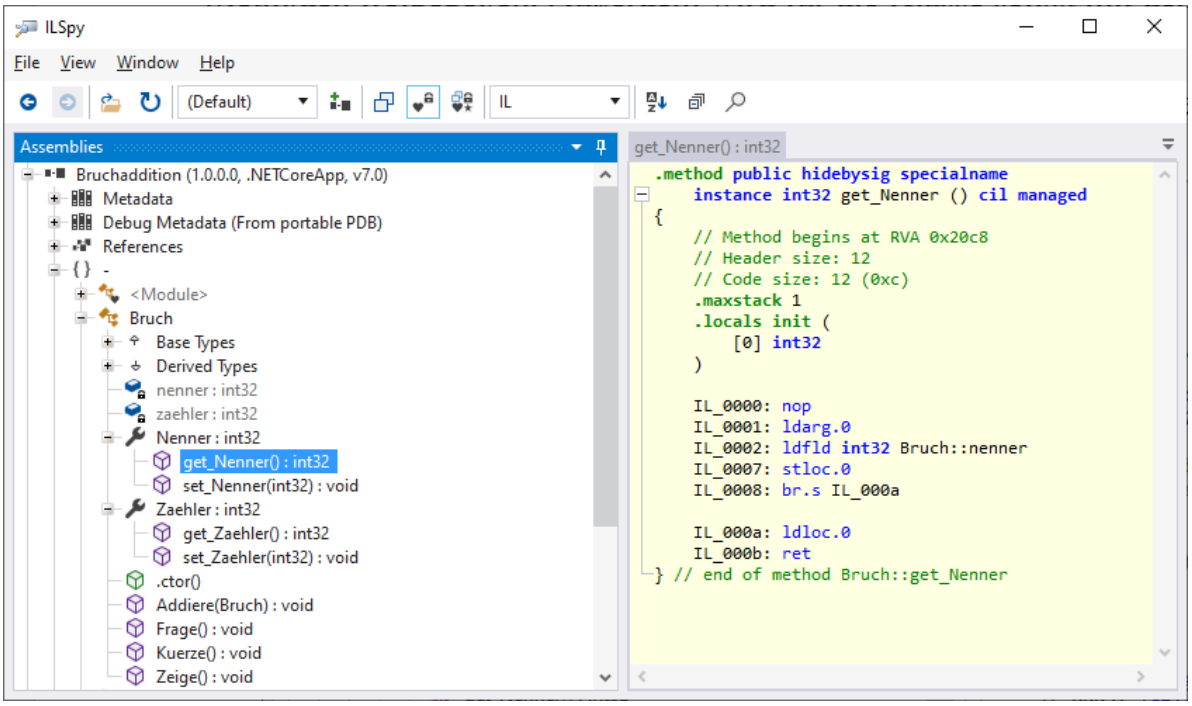
Zwar sind Ihnen die meisten Details der folgenden Klassendefinition selbstverständlich jetzt noch fremd, doch sind die Variablendeklarationen sowie die Eigenschafts- und Methodenimplementationen als zentrale Bestandteile leicht zu erkennen:

```
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($"   {zaehler}\n  -----\n   {nenner}\n");
   }
```

```
    public void Kuerze() {
       // größten gemeinsamen Teiler mit dem Euklidischen Algorithmus bestimmen
       if (zaehler != 0) {
          int az = Math.Abs(zaehler);
          int an = Math.Abs(nenner);
          while (az != an)
             if (az > an)
                az = az - an;
             else
                an = an - az;
          zaehler = zaehler / az;
          nenner  = nenner / az;
       } else
          nenner = 1;
    }

    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());
    }
}
```

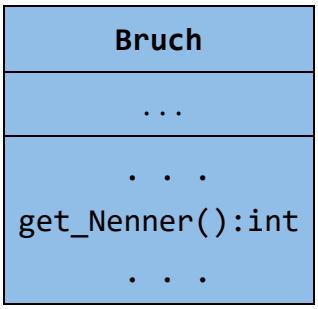
Weil für die beiden Felder (`zaehler`, `nenner`) die voreingestellte **private**-Deklaration unverändert gilt, ist im Beispielprogramm das Prinzip der Datenkapselung realisiert. Eigenschaften und Methoden werden durch die Verwendung des Modifikators **public** für die Verwendung in klassenfremden Methoden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

Damit die Klasse `Bruch` in einem Programm verwendet werden kann, muss ein sogenannter Compiler aus dem Quellcode den IL-Code (*Intermediate Language*) erstellen (siehe z. B. Abschnitt 2.3). Im aktuellen Kapitel wollten wir uns eigentlich auf den Quellcode beschränken, doch eine Inspektion der übersetzten Klasse mit dem „Spionageprogramm“ ILSpy (vgl. Abschnitt 3.4.1) liefert wichtige Einblicke:



Es bestätigt sich die Aussage von Abschnitt 1.2, dass hinter den C# - Eigenschaften letztlich Methoden für Lese- und Schreibzugriffe stehen (siehe z. B. `get_Nenner()`, `set_Nenner()`).

Im IL-Code der Klasse `Bruch` befindet sich u. a. die Methode `get_Nenner()`, die ihrem Aufrufer einen Wert vom Datentyp **int** liefert (Ganzzahl mit 32 Bit Speicherbedarf). Von den C# - Methoden der Klasse `Bruch` liefert zufälligerweise keine einzige einen Rückgabewert, sodass im UML-Diagramm zur Klasse `Bruch` der wichtige Fall einer Methode mit Rückgabe nicht auftaucht (siehe Abschnitt 1.2). Würde man die Methode `get_Nenner()` trotz der nicht konventionskonformen Benennung im C# - Quellcode definieren, dann würde sie im Klassendiagramm so aussehen:



Wie Sie bei späteren Beispielen erfahren werden, dienen in einem objektorientierten Programm 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).

1.4 Algorithmen

Zu Beginn von Kapitel 1 wurden mit der *Modellierung des Anwendungsbereichs* und der *Realisierung von Algorithmen* zwei wichtige Aufgaben der Software-Entwicklung genannt, von denen die letztgenannte bisher kaum zur Sprache kam. Auch im weiteren Verlauf des Manuskripts wird die explizite Diskussion von Algorithmen (z. B. hinsichtlich 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. Damit die Beschäftigung mit Algorithmen im Manuskript nicht ganz fehlt, werden wir im Rahmen des Bruchrechnungsbeispiels alternative Verfahren zum Kürzen von Brüchen betrachten.

In der initialen Variante der Klasse `Bruch` (siehe Abschnitt 1.3) verwendet die Methode `Kuerze()` den bekannten und nicht gänzlich trivialen **euklidischen Algorithmus**, um den größten gemeinsamen Teiler (GGT) der Beträge 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:

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

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:

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

Weil die Paare (u, v) und (v, u - v) dieselben Mengen gemeinsamer Teiler besitzen, sind auch die größten gemeinsamen Teiler identisch.

Beim Übergang von

$$(u, v) \quad \text{mit } u > v > 0$$

zu

$$(v, u - v) \quad \text{mit } v > 0 \text{ und } u - v > 0$$

wird die größere von den beiden Zahlen durch eine echt kleinere Zahl ersetzt, während der GGT identisch bleibt.

Wenn v und (u - v) in einem Prozessschritt identisch werden, dann ist der GGT gefunden (= v). Das muss nach endlich vielen Schritten passieren, denn:

- Solange die beiden Zahlen im aktuellen Schritt k noch *verschieden* sind, resultieren im nächsten Schritt k+1 zwei neue Zahlen mit einem echt kleineren Maximum.
- Alle Zahlen bleiben > 0.
- Das Verfahren endet in endlich vielen Schritten, eventuell mit $v = 1$.

Weil die Zahl 1 als trivialer Teiler zugelassen ist, existiert zu zwei natürlichen Zahlen immer ein größter gemeinsamer Teiler, der eventuell gleich 1 ist.

Diese Ergebnisse werden 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 vereinfachten 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 zum Abschnitt 4.7 sollen Sie eine effizientere Variante erstellen.

1.5 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 `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 **Startklasse** geben muss, die eine Methode mit dem Namen **Main()** in ihrem klassenbezogenen Handlungsrepertoire besitzt.[1] Beim Start eines Programms wird die Startklasse aufgefordert, ihre **Main()** - Methode auszuführen. Diese Methode kann als *Einsprungpunkt* (engl. *Entrypoint*) für das Programm bezeichnet werden.

Es bietet sich an, die oben angedachte Handlungssequenz des Bruchadditionsprogramms in der obligatorischen **Main()** - Methode der Startklasse unterzubringen.

Die auf Wiederverwendbarkeit hin konzipierte Klasse `Bruch` sollte nicht mit der Startmethode für eine spezielle Anwendung belastet werden. 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:[2]

---

[1] Genau genommen kann statt einer Klasse auch eine sogenannte *Struktur* die Rolle des Starters übernehmen und die **Main()** - Methode realisieren, siehe:
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/main-and-command-args/
Mit den Strukturen, die als leichtgewichtige Klassen aufgefasst werden können, beschäftigen wir uns im Abschnitt 6.1.

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

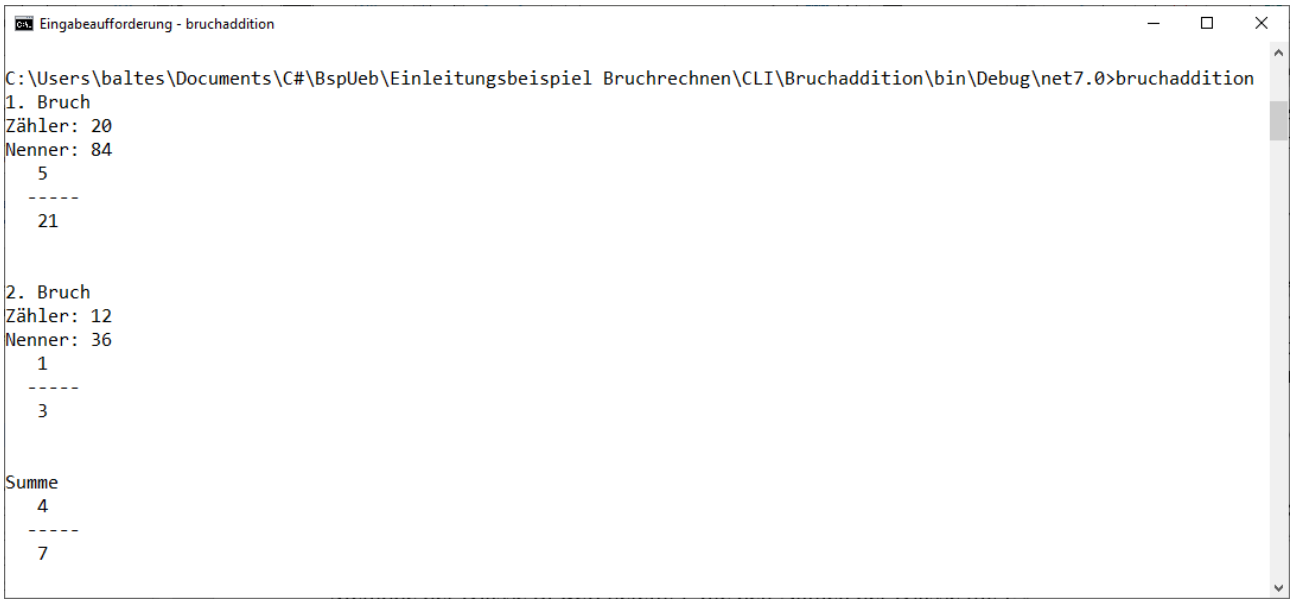
Quellcode in **Bruchaddition.cs**	Ausgabe (Eingaben **fett**)

`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 unter Windows startet man aus dem Ordner

...\BspUeb\Einleitungsbeispiel Bruchrechnen\CLI\Bruchaddition\bin\Debug\net7.0

(zu finden an der im Vorwort vereinbarten Stelle) das Programm **Bruchaddition.exe** (z. B. per Doppelklick):[1]



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 .NET - Basisklassenbibliothek (Base Class Library, BCL), indem wir deren Klassenmethoden **WriteLine()** und **ReadLine()** aufrufen. Im Manuskript müssen wir nicht nur die Programmiersprache C# behandeln, sondern auch die .NET – Basisklassenbibliothek, die eine enorme Anzahl von Klassen enthält und bei der

[1] Damit sich **Bruchaddition.exe** (z. B. per Doppelklick) starten lässt, muss im selben Ordner auch die Datei **Bruchaddition.runtimeconfig.json** vorhanden sein. Diese Konfigurationsdatei entsteht automatisch zusammen mit der ausführbaren Datei (siehe z. B. Abschnitt 3.1.5).

Programmentwicklung unverzichtbar ist. Die Funktionalität unserer Programme resultiert wesentlich aus der Nutzung von BCL-Klassen (durch den Aufruf von Klassen- und/oder Instanzmethoden). In unseren eigenen Klassendefinitionen nutzen wir vorhandene Klassen aus der BCL und aus anderen Bibliotheken, damit wir nicht das Rad und ähnliche Dinge selbst erfinden müssen.

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 `Bruch` kann in verschiedenen Programmen eingesetzt werden (Wiederverwendbarkeit). Dies fällt leicht, weil die Objekte Handlungskompetenzen (Methoden) besitzen **und** ihren Zustand mit den erforderlichen Instanzvariablen verwalten.
  - Beim Umgang mit den `Bruch`-Objekten sind wenige Probleme zu erwarten, weil nur klasseneigene Methoden direkten 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) sichtbar, was im Manuskript noch ausführlich zu behandeln ist.[1]
- 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 allerdings Quellcodedateien mit mehreren Klassen und einem beliebigen Dateinamen erlaubt.

Während bei den Namen von C# - Klassen die Groß-/Kleinschreibung signifikant ist, spielt sie bei Dateinamen unter Windows bekanntlich keine Rolle. Wenn eine C# - Quellcodedatei der üblichen Praxis folgend genau *eine* Klassendefinition enthält, sollte der Dateiname *inkl. Groß-/Kleinschreibung* von der Klasse übernommen werden. Diese Konvention wird auch im Quellcode der .NET - Basisklassenbibliothek (Base Class Library, BCL) beachtet.[2]

Im Quellcode des Beispielprogramms blieben aus didaktischen Gründen zwei Kürzungsmöglichkeiten ungenutzt:[3]

- Seit C# 10 kann man dank *impliziter* **using**-Direktiven zum Importieren von Namensräumen in vielen Fällen auf explizite **using**-Direktiven verzichten (siehe Abschnitt 2.6.3).
- Seit C# 9 darf man in der Startklasse eines Konsolenprogramms die Definitionsköpfe der Klasse und der **Main()** – Methode weglassen. Die in diesem Kontext als *Anweisungen auf oberster Ebene* (engl.: *top-level statements*) bezeichneten Anweisungen der **Main()** – Methode schreibt man an den Anfang der Quellcodedatei (siehe Abschnitt 4.1.2).

---

[1] Wenn sich die Klassen `Bruchaddition` und `Bruch` in derselben Übersetzungseinheit (*Assembly* genannt) befinden, dann ist der Zugriffsmodifikator **public** für die Klasse `Bruch` nicht erforderlich. Im Abschnitt 1.6 mit einem Ausblick auf Anwendungen mit grafischer Bedienoberfläche wird jedoch ein Programm vorgestellt, das ein eigenständiges, „externes“ Assembly mit der Klasse `Bruch` verwendet. Das ist der typische Fall, weil die Klasse `Bruch` nicht neu übersetzt werden muss.

[2] Wie man den BCL-Quellcode einsehen und herunterladen kann, erläutert der Abschnitt 2.6.4.

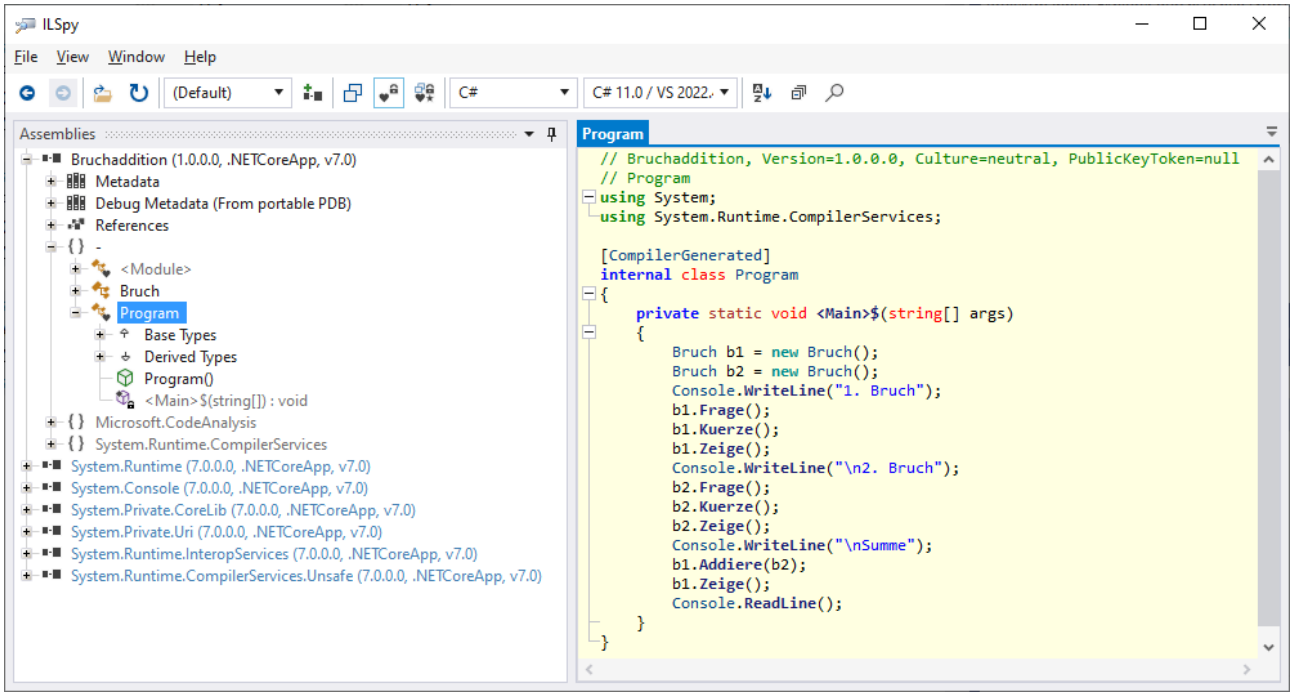
[3] https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/top-level-statements

Mit dem folgenden Quellcode in der Datei **Bruchaddition.cs**

```
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();
```

resultiert nach der Übersetzung in die Intermediate Language ein Programm mit unverändertem Verhalten.

Eine Inspektion des übersetzten Quellcodes mit dem Diagnoseprogramm ILSpy



zeigt, dass der Compiler die weggelassenen Bestandteile in leicht geänderter Form eingefügt hat:

- Er hat den BCL-Namensraum **System** durch eine **using**-Direktive importiert.
- Er hat eine Startklasse mit dem Namen **Program** definiert und durch das Attribut `[CompilerGenerated]` dekoriert.[1]
- Er hat in der Klasse **Program** eine statische Methode mit dem ungewöhnlichen Namen **<Main>$()** definiert, der gegen die Benennungsregeln verstößt und folglich von uns nicht vergeben werden dürfte.[2]

---

[1] Mit Attributen werden wir uns im Kapitel 14 beschäftigen.

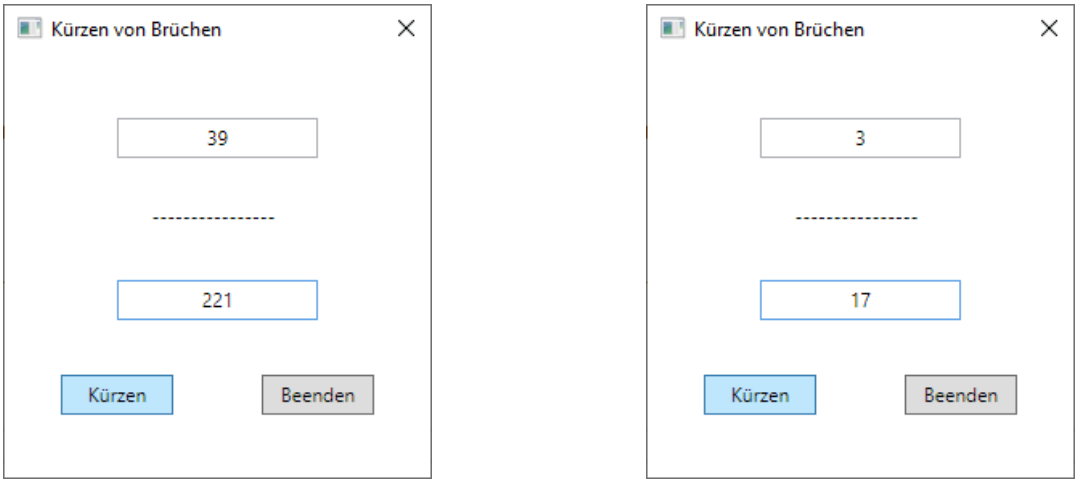
[2] Die Verwendung eines **string[]** -Parameters, der für eine Serie von Zeichenfolgen steht, ist die bei **Main()** – Methoden generell erlaubte Alternative zur leeren Parameterliste.

Es spricht nichts dagegen, bei Konsolenprogrammen die impliziten **using**-Direktiven und die Anweisungen auf oberster Ebenen zu nutzen. Allerdings sollte man verstanden habe, dass …

- sich am Aufbau des Programms nichts ändert,
- der Compiler die vom Programmierer eingesparten Zeilen übernimmt.

1.6 Ausblick auf Anwendungen mit grafischer Bedienoberfläche

Das im Kapitel 1 entwickelte Beispielprogramm arbeitet der Einfachheit halber mit einer konsolenorientierten Ein- und Ausgabe. Nachdem wir im Manuskript in dieser übersichtlichen Umgebung grundlegende C# - Sprachelemente kennengelernt haben, werden wir uns selbstverständlich auch mit der Programmierung von grafischen *Bedienoberflächen* beschäftigen. Im folgenden Programm zum Kürzen von Brüchen wird die oben definierte Klasse `Bruch` verwendet. An Stelle ihrer Methoden `Frage()` und `Zeige()` kommen jedoch grafikorientierte Techniken zum Einsatz:



Zum Starten des Programms (nur unter Windows möglich) genügt ein Doppelklick auf die Datei **BruchKürzenGui.exe** im Ordner:[1]

…\BspUeb\Einleitungsbeispiel Bruchrechnen\GUI\bin\Debug\net6.0-windows

Mit dem Quellcode zur Gestaltung der grafischen Oberfläche könnten Sie im Moment noch nicht allzu viel anfangen. Wir werden das Programm im Abschnitt 5.13 erstellen.

1.7 Zusammenfassung zum Kapitel 1

Zweck des ersten Kapitels war, Sie mit der Denk- und Arbeitsweise der objektorientierten Programmierung vertraut zu machen. Alle dabei erwähnten Konzepte und die technische 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 aus dem ersten Kapitel 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.

---

[1] Während die.NET – Basisbibliothek (die BCL) plattformunabhängig ist, steht die im Programm verwendete GUI-Bibliothek WPF nur unter Windows zur Verfügung (siehe Kapitel 2).

- Eine Klasse besitzt einen **Zustand** (beschrieben durch **Merkmale**) und **Handlungskompetenzen** (realisiert durch **Methoden**).
- Ein Merkmal bzw. eine Methode wird entweder den Objekten einer Klasse oder der Klasse selbst zugeordnet.
- Bei einer technischen Betrachtungsweise besitzt eine Klasse Felder, Eigenschaften und Methoden als Member.
- Die Felder sind in der Regel vor dem direkten Zugriff durch andere Klassen geschützt. Die Methoden sind hingegen oft auch für andere Klassen nutzbar und ermöglichen somit eine Kommunikation.
- In C# kann fremden Klassen für ein Feld durch eine sogenannte *Eigenschaft* ein kontrollierter Zugang verschafft werden, wobei es sich letztlich um ein Paar von Methoden für den lesenden und den schreibenden Zugriff handelt.
- Eine Klasse dient in der Regel als **Bauplan für Objekte**, kann aber auch selbst aktiv werden (Methoden ausführen und aufrufen).
- In den Methodendefinitionen werden Algorithmen mittels Befehlen zur Programmablaufsteuerung realisiert. Dabei kommen vordefinierte Klassen aus diversen Bibliotheken zum Einsatz, aber auch selbst erstellte Klassen. Von den einbezogenen Bibliotheken spielt die .NET - Basisklassenbibliothek (*Base Class Library*, BCL) eine herausragende Rolle.
- 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“.

1.8 Übungsaufgaben zum Kapitel 1

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Die Merkmale einer Klasse sind meist gekapselt, also vor direkten Zugriffen durch fremde Klassen geschützt.
2. Die Methoden einer Klasse sind grundsätzlich öffentlich.
3. Bei den Eigenschaften von C# handelt es sich in der Regel um Paare von Zugriffs*methoden*, die aber syntaktisch wie (öffentlich verfügbare) Felder verwendet werden.
4. In C# besitzt jede Klasse eine Methode namens **Main()** in ihrem klassenbezogenen Handlungsrepertoire.
5. Der Zustand eines Objekts darf nur durch Methoden der eigenen Klasse verändert werden.

2) Warum steigt die Produktivität der Software-Entwicklung durch objektorientiertes Programmieren?

2 Grundzüge der .NET – Technologie

Im Kapitel 1 haben Sie C# als eine Programmiersprache kennengelernt, die Ausdrucksmittel zur Modellierung des Anwendungsbereichs 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 (2023) die CPU eines handelsüblichen Arbeitsplatzrechners (mit einer Taktfrequenz von ca. 3-5 GHz und zahlreichen Kernen/Threads) mehrere hundert Milliarden Befehle pro Sekunde (*Instructions Per Second*, *IPS*) schafft.[1]

Ein C# - Quellcode muss also erst in Maschinencode übersetzt werden, damit das Programm von einem Rechner ausgeführt werden kann. Dabei findet eine *zweischrittige* Übersetzung statt, die stark an die Java – Software-Technik erinnert:

- Der C# - Quellcode wird vom Compiler in die sogenannte *Intermediate Language* (*IL*) übersetzt. Man kann sich darunter (wie beim Java-Bytecode) den Maschinencode einer *virtuellen* Maschine vorstellen, die sich relativ leicht auf reale Hardware-Architekturen abbilden lässt.
- Auf jeder realen Maschine muss sich eine .NET – Laufzeitumgebung befinden, die den IL-Code in die jeweilige Maschinensprache übersetzt. Weil der Benutzer bereits auf den Start der Software wartet, kommt ein flink agierender *JIT*-Compiler zum Einsatz (*Just In Time*), der nur den gerade auszuführenden Code übersetzt.

Im aktuellen Kapitel werden elementare Eigenschaften der .NET – Software-Technik behandelt:

- C# - Compiler und IL (Intermediate Language)
- Assembly (Übersetzungseinheit) mit IL-Code und Metadaten
- CLR (Common Language Runtime) mit JIT-Compiler
- BCL (Base Class Library)
- Namensräume

Dabei werden auch einige, früher oder später relevante Details erwähnt, die man beim Einstieg in die Programmiersprache C# noch nicht unbedingt kennen muss. Im Text befinden sich Hinweise auf Abschnitte, die beim ersten Lesen übersprungen werden dürfen.

2.1 Dokumentationswebseiten von Microsoft

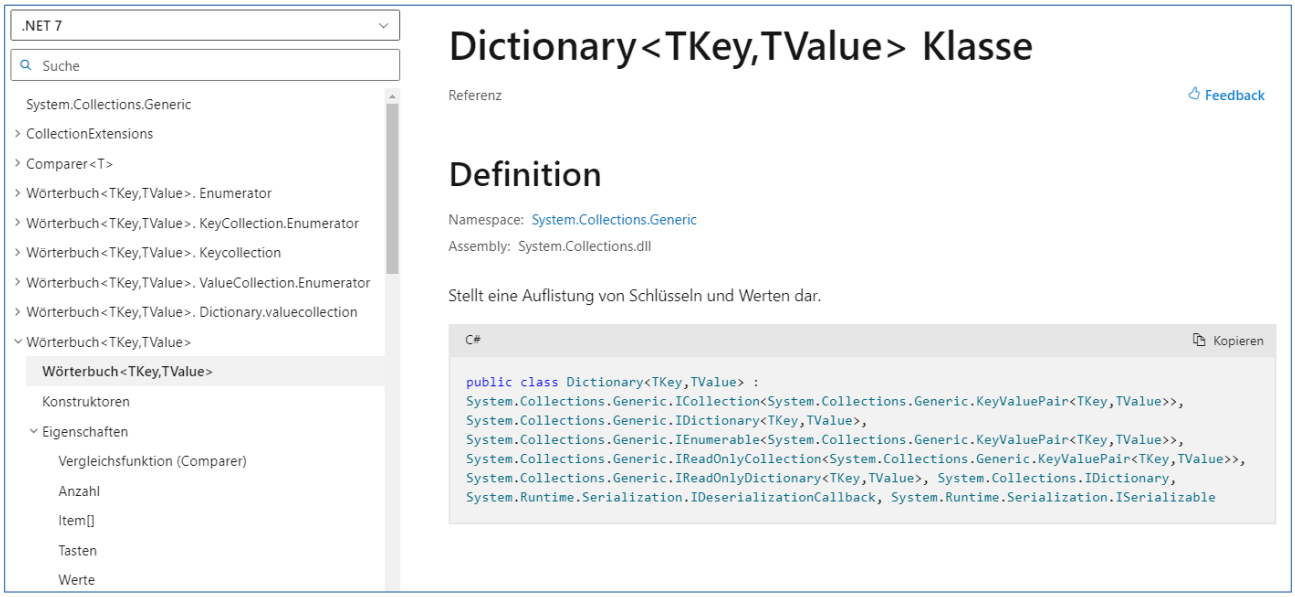
Im Manuskript werden zahlreiche Dokumentationswebseiten von Microsoft zitiert, wobei die US-Version (in Englisch) meist den Vorzug erhält, weil sich in die deutsche Übersetzung immer wieder sinnentstellende Fehler einschleichen. So ist z. B. auf der deutschsprachigen Webseite

https://learn.microsoft.com/de-de/dotnet/api/system.collections.generic.dictionary-2

über die generische Klasse **Dictionary<TKey, TValue>** (zu generischen Klassen siehe Kapitel 8) in der Navigationszone am linken Rand die *nicht-existente* Klasse **Wörterbuch** zu sehen:

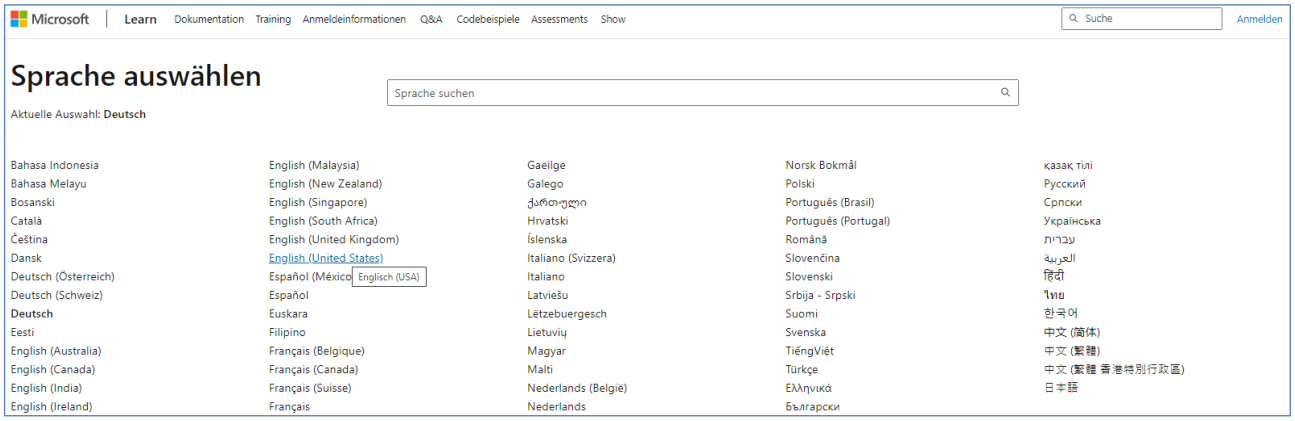
---

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

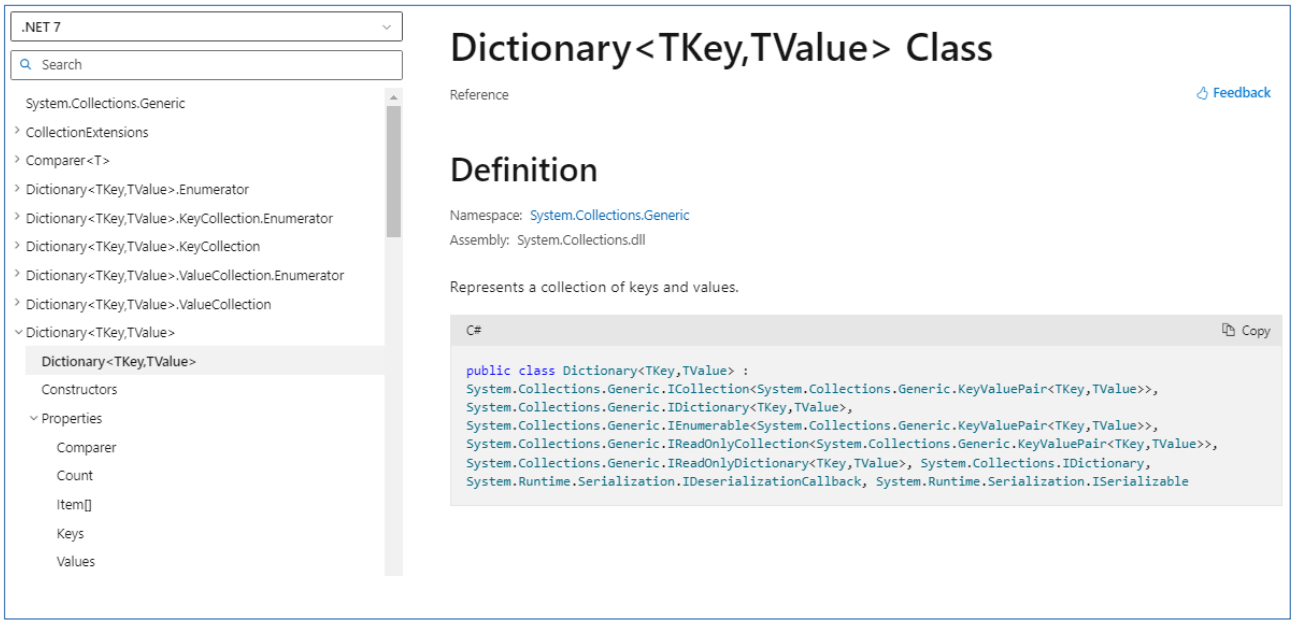


Außerdem wird die Eigenschaft **Keys** der Klasse **Dictionary<TKey, TValue>** überflüssigerweise und falsch in *Tasten* übersetzt.

Am unteren Rand einer Dokumentationswebseite kann man die Sprache wählen,



und im Beispiel verschwinden die irreführenden Bezeichnungen nach dem Wechsel zur US-Variante:



2.2 .NET - Implementationen

Wie schon im Vorwort erwähnt, bemüht sich Microsoft seit 2020 darum, das .NET - Ökosystem nach einer Phase der Diversifizierung wieder zu konsolidieren, um plattformübergreifende Lösungen mit einer möglichst einheitlichen Technologie zu realisieren. Eine Orientierung über die .NET – Implementationen hilft z. B. bei der Unterscheidung zwischen veralteten und zukunftssicheren Segmenten im .NET - Ökosystem. Wer den kürzesten Weg zur Beschäftigung mit der Programmiersprache C# sucht, kann aber den Abschnitt 2.2 vorerst auslassen.

Aktuell (2023) unterstützt Microsoft die folgenden vier .NET-Implementationen:[1]

Implementation	**Unterstützte Betriebssysteme**	**Version im Juni 2023**	**Version von .NET Standard/C#**
.NET (inkl. **.NET Core**)	Linux, macOS, Windows, Android, iOS	7.0	2.1/11
.NET Framework	Windows	4.8	2.0/7.3
Mono[2]	Linux, macOS, Windows	6.12	2.1/8.0
UWP (Universal Windows Platform)[3]	Windows 10/11	Identisch mit der Windows-Version	2.0/7.3

Von C# unterstützen die .NET - Implementationen unterschiedliche Sprachversionen. Nur die sparsam mit *.NET* bezeichnete Implementation unterstützt die aktuelle C# - Version 11.

In der Diversifizierungsphase hatte Microsoft eine Sammlung von Spezifikationen namens **.NET Standard** definiert, um die Kooperation von .NET - Implementationen zu ermöglichen.[4] Durch die seit .NET 5 gestartete Konsolidierung hat .NET Standard an Bedeutung verloren. Gelegentlich ist aber die Version 2.0 noch von Interesse, weil sie von allen aktuell einsetzbaren (mit Sicherheits-Updates versorgten) .NET – Implementationen unterstützt wird:

- .NET ab 5.0, .NET Core ab 3.0
  Seit der Version 6 gehören zu .NET auch die unter Android, iOS, macOS und Windows laufenden Anwendungen mit *Multi-platform App UI* (*MAUI*).
- .NET Framework ab 4.6.1
- Mono ab 5.4
- UWP ab 10.0.16299

Damit eine neu entwickelte .NET – Bibliothek (eine DLL-Datei mit dem IL-Code von Klassen und anderen Typen, ohne Startklasse) in *mehreren* .NET – Implementationen genutzt werden kann (z. B. in .NET 7 und .NET Framework 4.8), sollte in der **csproj**-Projektdatei (siehe z. B. Abschnitt 3.1.2) **netstandard2.0** als **TargetFramework** angegeben werden, z. B.:

---

1 https://learn.microsoft.com/en-us/dotnet/fundamentals/implementations

2 http://www.mono-project.com/
https://de.wikipedia.org/wiki/Mono_(Software)

3 https://learn.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide

4 https://learn.microsoft.com/en-us/dotnet/standard/net-standard
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Library</OutputType>
    <TargetFramework>netstandard2.0</TargetFramework>
     . . .
  </PropertyGroup>
   . . .
</Project>
```

Die konsolidierte .NET - Technologie besitzt wegen der unterschiedlichen technischen Voraussetzungen auf den zu versorgenden IT-Systemen (z. B. Arbeitsplatzrechner, Server, mobile Geräte, Browser als Ausführungsumgebung) immer noch einige Variabilität. Allen Varianten gemeinsam ist eine Basisklassenbibliothek, die durch anwendungsspezifische Bibliotheken (z. B. für den Windows-Desktop, für Web- oder MAUI-Anwendungen) ergänzt wird. Zur Ausführung von .NET - Anwendungen (Assemblies) kommen zwei verschiedene Laufzeitumgebungen zum Einsatz:

- **CoreCLR** (*Core Common Language Runtime*) auf Arbeitsplatzrechnern und Servern unter Linux, macOS und Windows
- **Mono-Runtime** auf mobilen Geräten unter Android und iOS sowie für Blazor WASM

Im Manuskript …

- werden C# 11 und .NET 7 vorgestellt
- und meist im Rahmen von Konsolenanwendungen behandelt, die unter Linux, macOS und Windows laufen und folglich die CoreCLR verwenden.

Ein Kapitel ist der nur unter Windows verfügbaren GUI-Technik WPF gewidmet, die gelegentlich auch in anderen Kapiteln für Abwechslung sorgt.

Das .NET – Framework wird als Auslaufmodell im Unterschied zur vorherigen Version des Manuskripts (Baltes-Götz 2021) nicht mehr berücksichtigt, und für .NET MAUI fehlte leider die Zeit. Zu den beiden ausgegrenzten .NET - Optionen folgen aber gleich einige Erläuterungen.

Die von Microsoft im Internet angebotenen Informationen zur .NET - Technologie sind zwar detailliert und nützlich, lassen aber Einsteiger oft im Unklaren darüber, für welche .NET – Implementation sie jeweils gelten.

2.2.1 .NET Framework

Diese 2002 eingeführte und somit älteste .NET – Implementation hat die Software-Entwicklung für das Betriebssystem Windows revolutioniert. Obwohl die Verwendung einer virtuellen Maschinensprache als Zwischencode das Designziel der Plattformunabhängigkeit erkennen lässt, hat Microsoft bis zum Jahr 2016 (Erscheinen von .NET Core) die Entwicklung von .NET – Software nur unter Windows unterstützt, während andere Firmen (Novell und Xamarin) schon seit 2004 an der freien, für mehrere Betriebssysteme verfügbaren .NET – Implementation Mono gearbeitet haben.

Weil das .NET Framework Bestandteil aller aktuellen Windows-Betriebssysteme ist, können Programme für diese .NET - Implementation auf allen Windows-Rechnern ohne die vorherige Installation einer Laufzeitumgebung ausgeführt werden. Das .NET Framework hat seit 2002 zahlreiche Aktualisierungen erfahren und ist 2019 bei der finalen Version 4.8 angekommen, die nicht mehr weiterentwickelt wird. Weil C# parallel zur .NET - Technologie modernisiert wird, kommt das .NET Framework nicht über die Version 7.3 der Programmiersprache hinaus.

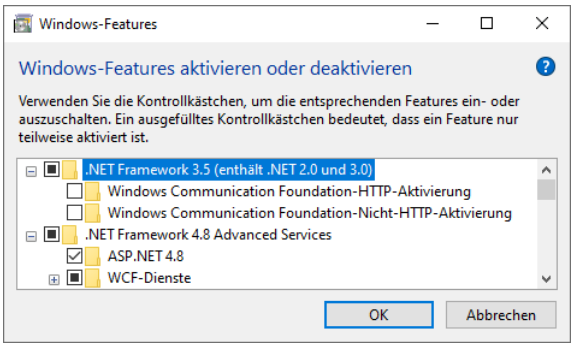
Seit 2020 ist mit dem aus .NET Core 3.1 hervorgegangenen .NET 5 die Plattformunabhängigkeit zu einem vorrangigen Merkmal geworden, wobei aber unter Windows die klassischen GUI-Bibliotheken für Desktop-Anwendungen (WinForms und WPF) unterstützt werden. Damit gibt es keinen Grund mehr für eine Software-Erstellung mit dem .NET Framework, das von den

Weiterentwicklungen der Standardklassenbibliothek (BCL) und der Programmiersprachen abgeschnitten ist. Eine zur Ausführung von modernen .NET – Anwendungen geeignete Laufzeitumgebung lässt sich unter Windows schnell nachrüsten. Außerdem besteht die Möglichkeit, die Laufzeitumgebung inklusive der erforderlichen Bibliotheksbestandteile zusammen mit einer .NET – Anwendung auszuliefern.

Während es kaum einen Grund gibt, neue Software für das .NET Framework zu erstellen, spricht nichts gegen die Verwendung eines vorhandenen Programms. In allen aktuell durch Sicherheitsupdates versorgten Versionen von Windows 10 und 11 ist das .NET Framework in der Version 4.8 enthalten. Es kann erforderlich werden, eine ältere Framework-Version via Windows-Systemsteuerung (zu starten durch Eingabe von *Systemsteuerung* oder *Control* im Taskleisten-Suchfeld von Windows) über

Programme > Windows-Features aktivieren oder deaktivieren

freizuschalten, weil .NET Framework - Programme zur Vermeidung von Versionskonflikten 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.

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

.NET Framework - 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 laufen, sind dort *zwei* .NET Framework - Laufzeitumgebungen (mit 32 bzw. 64 Bit) vorhanden.

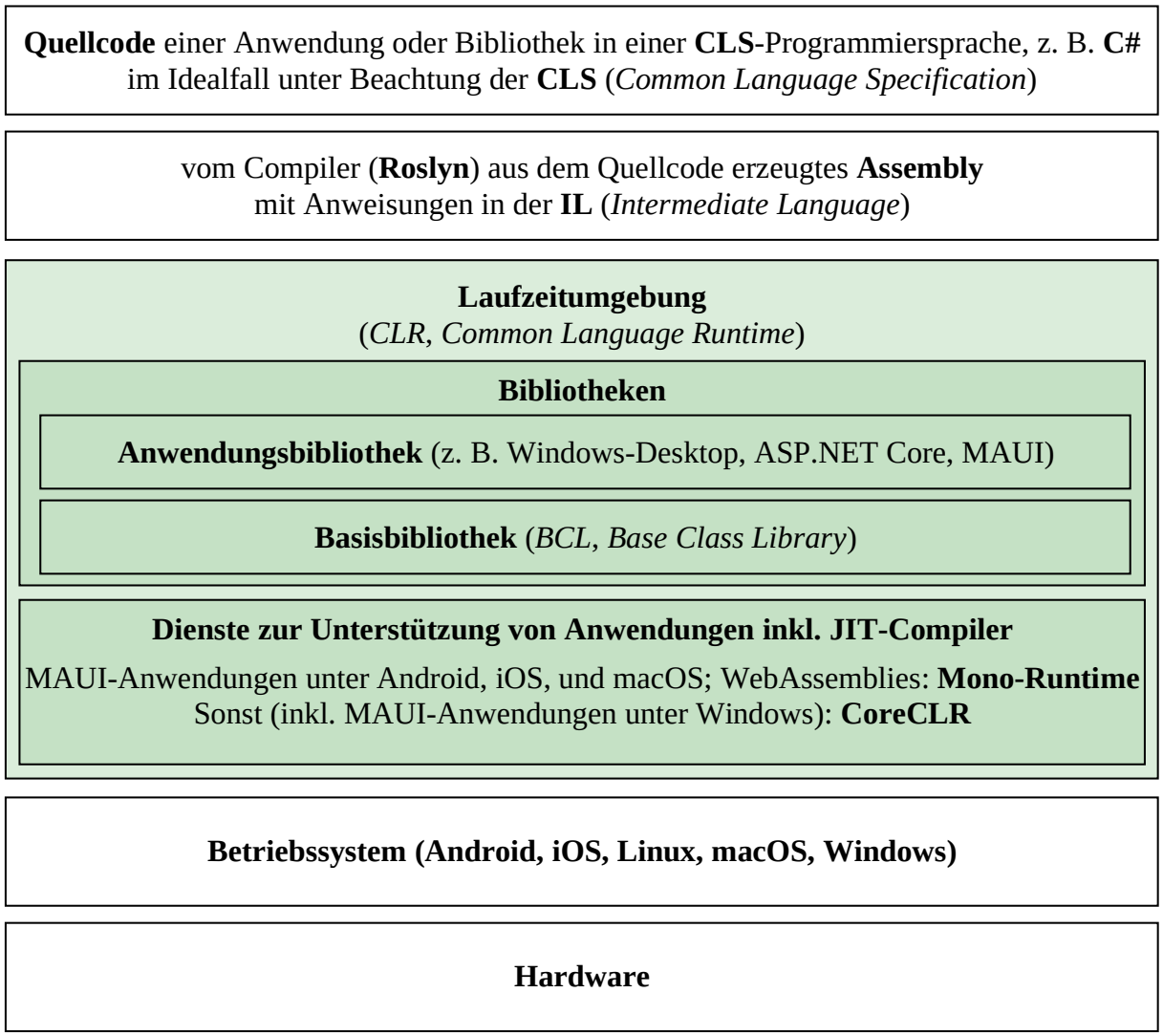
Das .NET Framework ist (zusammen mit .NET Standard) durch die nun mit *.NET* bezeichnete, aus .NET Core hervorgegangene, einheitliche und plattformübergreifende Implementation überflüssig geworden. Für *neue* Projekte sollte das .NET Framework auch dann gemieden werden, wenn Windows die einzige Zielplattform ist, denn:

- Für das .NET Framework endet die C# - Entwicklung mit der Version 7.3, während die Programmiersprache bereits bei der Version 11 angekommen ist.
- Das einheitliche .NET unterstützt unter Windows auch die traditionellen GUI-Bibliotheken WPF und WinForms.

2.2.2 .NET 7

Im November 2020 ist das plattformübergreifende .NET 5 zusammen mit C# 9 erschienen, um alle bisherigen .NET - Implementationen durch eine einheitliche Lösung zu ersetzen. Dabei hat Microsoft auch einige Altlasten aus der Standardbibliothek entfernt und den Zusatz *Framework* aus dem Namen gestrichen. Man kann .NET 5 als Weiterentwicklung von .NET Core 3.1 auffassen, wobei die Versionsnummer 4 wegen der Verwechslungsgefahr mit dem .NET Framework 4.x übersprungen wurde. Unter Windows unterstützen .NET 5 und seine Nachfolger auch Anwendungen mit den etablierten GUI-Techniken WPF (*Windows Presentation Foundation*) und WinForms. Im Juni 2023 ist die .NET – Version 7 aktuell, und im November 2023 ist mit .NET 8 zu rechnen.

Die folgende Abbildung zeigt, welcher Stapel von Technologien bei der Ausführung eines in C# geschriebenen Programms für .NET 7 beteiligt ist:



Wir beschränken uns im weiteren Verlauf von Kapitel 2 auf .NET 5 und seine Nachfolger.

2.2.2.1 Veröffentlichungsplan und Unterstützungsdauer für .NET - Versionen

Microsoft plant, in jedem November eine neue .NET - Hauptversion zu veröffentlichen, die alternierend mit einer einfachen oder langen Unterstützungsdauer ausgestattet ist:[1]

- STS (Standard Term Support)
  18 Monate Versorgung mit Sicherheits-Updates
- LTS (Long Term Support)
  26 Monate Versorgung mit Sicherheits-Updates

Wie die folgende Tabelle für die .NET - Versionen 5 bis 7 zeigt, hat Microsoft bisher seine Zusagen eingehalten:

Version	Erschienen	Letzte Patch-Version	Typ der Unterstützung	Ende der Unterstützung
.NET 7	8.11.2022	7.0.5	STS	10.5. 2024
.NET 6	8.11.2021	6.0.16	LTS	12.11.2024
.NET 5	10.11.2020	5.0.17	STS	10.5.2022

2.2.2.2 .NET MAUI

Seit der Version 6 sind .NET - Anwendungen mit grafischer Bedienoberfläche möglich, die mit identischem Quellcode unter Android, iOS, macOS und Windows laufen. Diese simultane Unterstützung von Betriebssystemen für mobile Geräte und Arbeitsplatzrechner wird mit einer Technik namens *.NET MAUI* realisiert, die aus dem Produkt *Forms* der Firma Xamarin hervorgegangen ist.[2] Obwohl .NET MAUI zu .NET gehört, also *nicht* als eigene .NET – Implementation betrachtet wird, bestehen doch einige Besonderheiten:

- Unter Android, iOS und macOS wird die Mono-Runtime verwendet, unter Windows hingegen die bei .NET übliche CoreCLR.
- Unter Windows wird statt der traditionellen GUI-Bibliotheken WinForms und WPF die WinUI-Bibliothek in der Version 3 verwendet. Microsoft hat diese Bibliothek zur Entwicklung innovativer Desktop-Anwendungen mit Metro-Bedienkonzept unter Windows 10 (ab Version 1809) und 11 entwickelt (siehe Abschnitt 12.1.1.3).
- Für iOS und macOS wird eine .NET MAUI – Anwendung *nicht* als IL-Code ausgeliefert, der durch einen JIT-Compiler (Just-In-Time) im Maschinencode zu übersetzen ist, sondern als nativer Assembler-Code für den ARM-Prozessor.
- Unter macOS wird die als *Catalyst* bezeichnete Technik zur Ausführung von iOS-Anwendungen durch das Desktop-Betriebssystem verwendet.

Ist eine .NET - Multi-Plattform - Anwendung mit grafischer Bedienoberfläche geplant, dann bietet .NET MAUI die Chance zur Vermeidung von Parallelentwicklungen. Soll eine Anwendung hingegen nur unter Windows laufen, dann ist wegen der besseren Funktionalität und Kompatibilität eine auf Windows spezialisierte und etablierte Lösung zu bevorzugen (z. B. WPF oder WinForms). Microsoft empfiehlt in dieser Lage das Windows App SDK, das als Nachfolger des wenig erfolgreichen UWP-Anwendungsmodells aufgefasst werden kann (siehe Abschnitt 12.1.1.3).[3]

---

[1] https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core

[2] https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui

[3] https://learn.microsoft.com/en-us/windows/apps/windows-dotnet-maui/

2.3 C# - Compiler und Intermediate Language

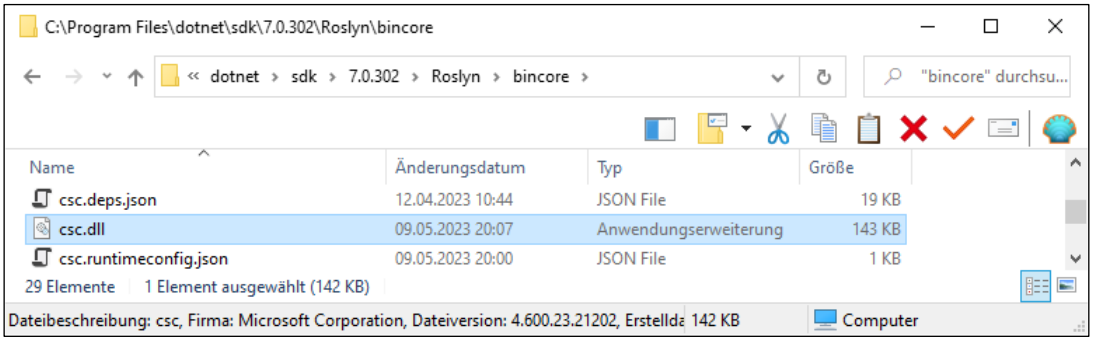
Den (z. B. mit einem beliebigen Texteditor verfassten) C# - Quellcode übersetzt ein C# - **Compiler** in die **Intermediate Language** (**IL**).[1] Wenngleich dieser Zwischencode von keiner CPU direkt ausgeführt werden kann, hat er doch bereits viele Verarbeitungsschritte auf dem Weg vom Quell- zum Maschinencode durchlaufen. Die Übersetzung des Zwischencodes in die Maschinensprache einer konkreten CPU geschieht *Just-In-Time* bei der Ausführung des Programms durch die CLR (siehe Abschnitt 2.5).

2.3.1 .NET Compiler Platform Roslyn

Für die Übersetzung von C# - Quellcode in IL-Code ist seit ca. 2014 die unter dem Namen **Roslyn** bekannte **.NET Compiler Platform** zuständig.[2] In der plattformübergreifenden .NET – Entwicklung (seit .NET Core bzw. .NET 5) arbeitet der Roslyn-Compiler im Hintergrund unter der Aufsicht des Erstellungssystems **MSBuild**.[3]

Ein Erstellungssystem ist ein Computer-Programm, das automatisiert aus Quellcode ein ausführbares Computer-Programm erzeugt. Im Fall eines C# - Programms ist dabei vor allem die Übersetzung in IL-Code zu leisten, und dazu verwendet das Erstellungssystem MSBuild den Roslyn-Compiler. Beim Erstellen eines ausführbaren Programms fallen aber noch weitere Aufgaben an, z. B. das Einbinden einer .NET – Laufzeitumgebung. Insofern ist die Software-Erstellung eine komplexere Aufgabe als die Quellcode-Übersetzung. Als Vorbild für das Erstellungssystem MSBuild hat laut Wikipedia das Programm *Apache Ant* aus der Java-Welt gedient.[4]

Bei den im Kapitel 3 beschriebenen Verfahren zur .NET – Entwicklung (z. B. mit Hilfe der Entwicklungsumgebung Visual Studio) ist immer MSBuild am Werk. Der dabei im Hintergrund tätige Roslyn-Compiler wird erst sichtbar, wenn es Fehler oder Warnungen zu melden gibt. Im aktuellen Abschnitt werden wir aus didaktischen Gründen den Compiler *direkt* ansprechen und dabei die mit dem .NET - SDK 7 installierte Version verwenden (siehe Abschnitt 3.1):



Der Compiler steckt (wie alle plattformübergreifenden .NET – Anwendungen) in einer Datei mit der Namenserweiterung **dll**. Die Datei **csc.dll** kann unter Windows mit Hilfe des .NET - SDK – Bestandteils **dotnet.exe** unter Beteiligung einer obligatorischen Konfigurationsdatei (**csc.runtimeconfig.json**) ausgeführt werden. Momentan sollen Sie noch nicht handeln, sondern den Entstehungsweg eines .NET – Programms beobachten (so ähnlich wie bei der modernen YouTube-Didaktik, aber mit mehr Kontrolle für Sie).

---

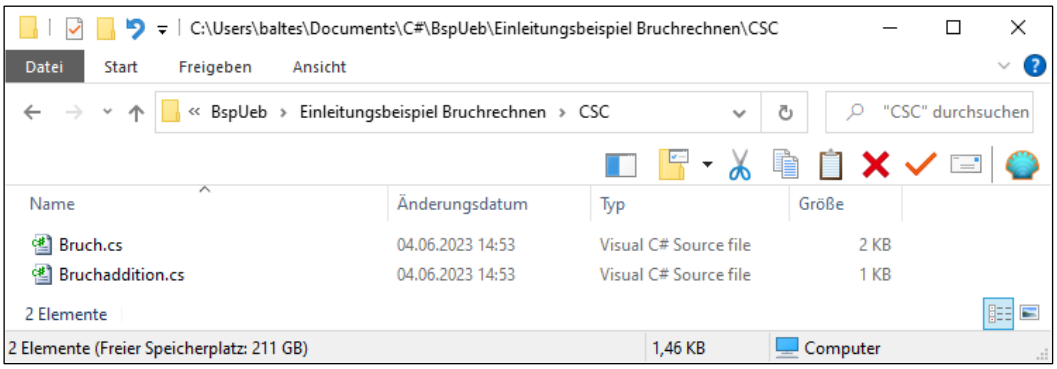
[1] Frühere Bezeichnungen:
- Die Bezeichnung *Common Intermediate Language* (*CIL*) ist noch weit verbreitet.
- Die ursprüngliche Bezeichnung *Microsoft Intermediate Language* (*MSIL*) wird nicht mehr verwendet.

[2] https://en.wikipedia.org/wiki/Roslyn_(compiler)

[3] https://learn.microsoft.com/en-us/visualstudio/msbuild/msbuild

[4] https://de.wikipedia.org/wiki/MSBuild

Befinden sich die aus dem Bruchrechnungsbeispiel von Kapitel 1 bekannten Quellcodedateien **Bruch.cs** und **Bruchaddition.cs** im aktuellen Verzeichnis



eines Konsolenfensters, dann eignet sich das folgende Kommando zur Übersetzung in die Datei **Bruchaddition.dll** mit IL-Code:

```
>dotnet "C:\Program Files\dotnet\sdk\7.0.302\Roslyn\bincore\csc.dll"
 -r:"C:\Program Files\dotnet\shared\Microsoft.NETCore.App\7.0.5\System.Runtime.dll"
 -r:"C:\Program Files\dotnet\shared\Microsoft.NETCore.App\7.0.5\System.Private.CoreLib.dll"
 -r:"C:\Program Files\dotnet\shared\Microsoft.NETCore.App\7.0.5\System.Console.dll"
 -out:Bruchaddition.dll *.cs
```

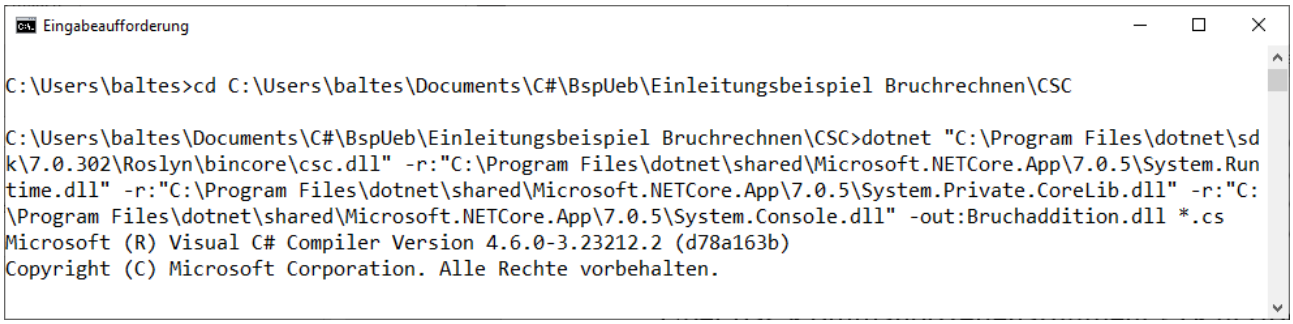
Über das Kommandozeilenargument **r** (Kurzform für **reference**) werden Bibliotheks-Assemblies benannt, die vom Compiler nach den im zu übersetzenden Quellcode benutzten Klassen und anderen Typen durchsucht werden sollen. Bei den im Kapitel 3 vorgeführten Erstellungsverfahren werden die benötigten Referenzen von MSBuild automatisch gesetzt.

Mit dem Kommandozeilenargument **out** lässt sich der Name der Ausgabedatei festlegen. Per Voreinstellung wird …

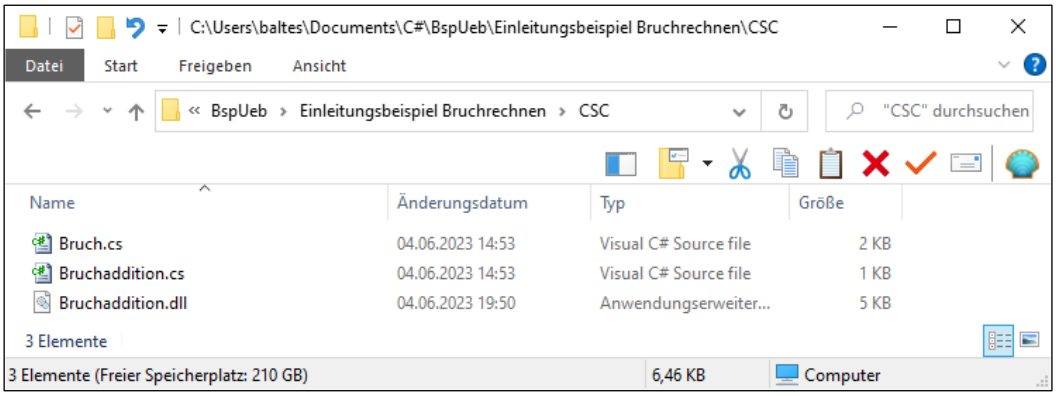
- der Name der Startklasse verwendet
- und unter Windows im Widerspruch zu der in .NET üblichen Benennung von plattformübergreifenden Anwendungen die Namenserweiterung **exe** angehängt.

Daher wird im Beispiel für eine konventionskonform benannte Ausgabedatei gesorgt.

Aus dem Aufruf



resultiert ein (unter Linux, macOS und Windows) ausführbares Programm:



Um Missverständnisse zu vermeiden, soll noch einmal wiederholt werden, dass wir im aktuellen Abschnitt aus didaktischen Gründen den etwas umständlichen direkten Aufruf des Roslyn-Compilers verwenden. Im Kapitel 3 werden Sie bequemere Verfahren kennenlernen, wobei insbesondere die Compiler-Optionen …

- entweder über die Projektdatei (mit der Namenserweiterung **csproj**)
- oder über Dialogfenster der Entwicklungsumgebung

gesetzt werden.[1]

Im Übersetzungsergebnis **Bruchaddition.dll** ist u. a. der IL-Code der beiden Klassen `Bruch` und `Bruchaddition` enthalten. Besonders kurz sind die **get**- und die **set**-Methode, die der Compiler zu einer C# - Eigenschaft erstellt (vgl. Abschnitt 1.2), z. B.:[2]

[1] Wer trotzdem den direkten Compiler-Aufruf verwenden möchte, erfährt durch das Abschicken eines Fragezeichens die Kommandozeilenargumente, z. B.:

```
dotnet "C:\Program Files\dotnet\sdk\7.0.302\Roslyn\bincore\csc.dll" -?
```

[2] Diese Ausgaben liefert das Programm ILSpy, das wir schon mehrfach verwendet haben. Seine Installation wird im Abschnitt 3.4.1 beschrieben.

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

C# - Compiler

⇒

```
get_Nenner() : int32
get_Nenner() : int32
  .method public hidebysig specialname
      instance int32 get_Nenner () cil managed
  {
      // Method begins at RVA 0x2090
      // Header size: 12
      // Code size: 12 (0xc)
      .maxstack 1
      .locals init (
          [0] int32
      )

      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
```

```
set_Nenner(int32) : void
set_Nenner(int32) : void
  .method public hidebysig specialname
      instance void set_Nenner (
          int32 'value'
      ) cil managed
  {
      // Method begins at RVA 0x20a8
      // Header size: 12
      // Code size: 17 (0x11)
      .maxstack 2
      .locals init (
          [0] bool
      )

      IL_0000: nop
      IL_0001: ldarg.1
      IL_0002: ldc.i4.0
      IL_0003: cgt.un
      IL_0005: stloc.0
      IL_0006: ldloc.0
      IL_0007: brfalse.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
```

Aus der `Bruch`-Eigenschaft `Nenner` resultiert u. a. die Methode `get_Nenner()`, deren IL-Code aus einer lokalen Variablen und sieben Anweisungen besteht.

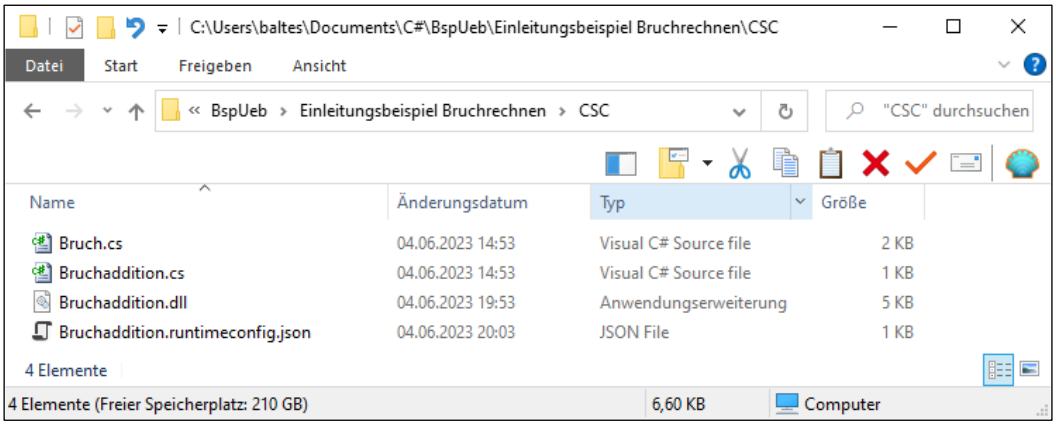
Die konzeptionelle Verwandtschaft des IL-Codes mit dem Bytecode der Java-Technologie ist offensichtlich.

2.3.2 Programm starten

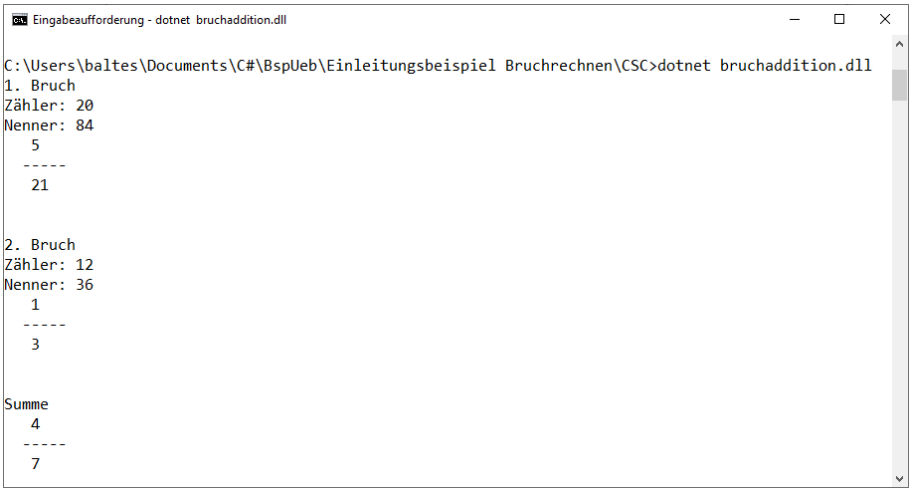
Damit das Programm ausgeführt werden kann, wird eine Konfigurationsdatei mit dem vorgeschriebenen Namen **Bruchaddition.runtimeconfig.json** und dem folgenden Inhalt benötigt:

```
{
  "runtimeOptions": {
    "tfm": "net7.0",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "7.0.0"
    }
  }
}
```

Nun



kann das Programm in der Datei **Bruchaddition.dll** per **dotnet.exe** gestartet werden:



Im Abschnitt 2.3.1 wurde aus didaktischen Gründen der C# - Compiler in der Datei **csc.dll** *direkt* verwendet. Aus dem empfohlenen Erstellungsprozess per MSBuild resultieren unter Windows zusätzlich zur **dll**-Datei mit dem IL-Code …

- die zum Starten benötigte Konfigurationsdatei (mit der Namenserweiterung **runtimeconfig.json**)
- und eine **exe**-Datei, die einen Programmstart auf windows-übliche Weise (z. B. per Doppelklick) erlaubt (siehe Abschnitt 3.1), wobei die Konfigurationsdatei unverzichtbar bleibt.

2.3.3 Common Language Specification

Mittlerweile sind für viele Programmiersprachen IL-Compiler verfügbar (z. B. für C, C++, C#, COBOL, Eiffel, F#, Java, JavaScript, Python, Ruby, Scala, Smalltalk, Visual Basic .NET).[1] Allerdings unterstützen diese Compiler in der Regel nicht den gesamten IL-Sprachumfang, sodass sich mit Compilern zu verschiedenen .NET - Sprachen durchaus Klassen produzieren lassen, die *nicht* zusammenarbeiten können. 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 von Visual Basic .NET erstellten Klassen *nicht* kooperieren.

Microsoft hat unter dem Namen **Common Language Specification** (CLS) die Voraussetzungen für die programmiersprachen-unabhängige Kooperation von Klassen definiert.[2] Beachtet man bei der

---

1 https://de.wikipedia.org/wiki/Liste_von_.NET-Sprachen

2 https://learn.microsoft.com/en-us/dotnet/standard/language-independence

Klassendefinition diese Bedingungen, dann ist die Interoperabilität mit anderen CLS-kompatiblen Klassen unabhängig von der verwendeten Programmiersprache sichergestellt.

Ein .NET - Compiler überwacht die CLS-Kompatibilität, wenn er über das Attribut **CLSCompliant** dazu aufgefordert wird, z. B.:[1]

```
[assembly: CLSCompliant(true)]

public class Prog {
    public void TuWas() {
        Console.WriteLine("TuWas");
    }

    public void tuWas() {
        Console.WriteLine("tuWas");
    }

    static void Main(string[] args) {
        Prog p = new();
        p.TuWas();
        p.tuWas();
    }
}
```

Wird die Übersetzung dieser Klasse im Visual Studio angefordert, dann erscheint in der **Fehlerliste** die folgende Warnung:

2.4 Assembly

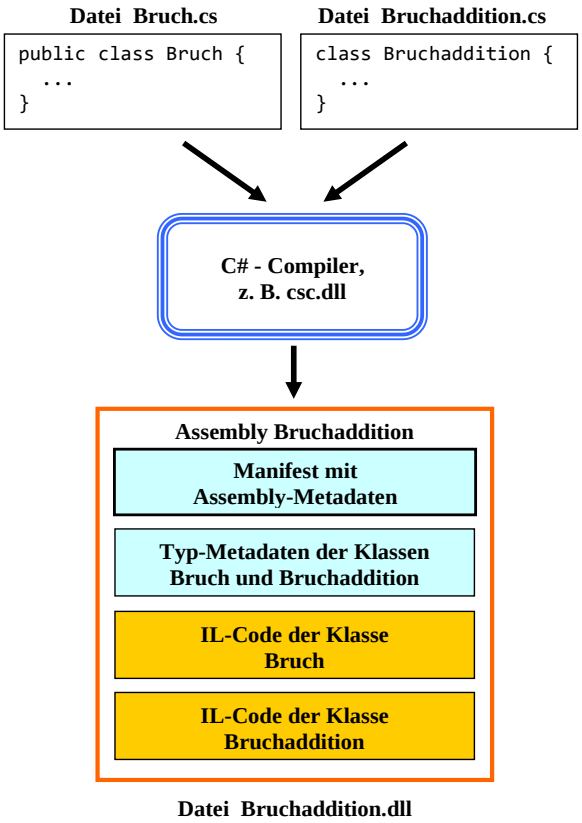
Die von einem .NET - Compiler erzeugte Binärdatei wird als **Assembly** bezeichnet und verwendet seit .NET 5 in der Regel die Namenserweiterung **dll**. Ein *ausführbares* Assembly benötigt im Unterschied zu einem *Bibliotheks*-Assembly eine Startklasse, die eine statische Methode namens **Main()** besitzen muss (siehe Abschnitt 1.5).

Man übergibt dem Compiler im Allgemeinen *mehrere* Quellcodedateien mit den Definitionen von Klassen (oder anderen, später noch ausführlich zu behandelnden Typen) und erhält als Ergebnis *ein* Assembly. Im Bruchadditionsbeispiel erzeugt der Compiler das ausführbare Assembly **Bruchaddition.dll** mit dem Zwischencode der Klassen `Bruch` und `Bruchaddition` (siehe Abschnitt 2.3).

Die folgende Abbildung (nach Mössenböck 2019, S. 7) fasst wesentliche Informationen über Quellcode, C# - Compiler, IL-Code und Assemblies anhand des Bruchadditionsbeispiels zusammen:

---

[1] Mit Attributen werden wir uns im Kapitel 14 beschäftigen. Der Zugriffsmodifikator **public** ist bei diesem Programm erforderlich, weil die Voreinstellung **internal** lediglich anderen Klassen in *derselben Übersetzungseinheit* den Zugriff auf die Klasse erlaubt, sodass kein Anlass für die Warnung besteht.



Neben dem IL-Code der enthaltenen Klassen sind als weitere wichtige Assembly-Bestandteile die anschließend zu beschreibenden *Metadaten* zu sehen. Von der Option, Ressourcen (z. B. Medien, Lokalisierungen von Zeichenfolgen) in ein Assembly aufzunehmen, wird im Beispiel kein Gebrauch gemacht.

Vermutlich ist mit dem Namen *Assembly* eine *Zusammenstellung* von …

- mehreren Typdefinitionen (basierend auf mehreren Quellcodedateien),
- Metadaten
- und Ressourcen

gemeint.

Assemblies sind …

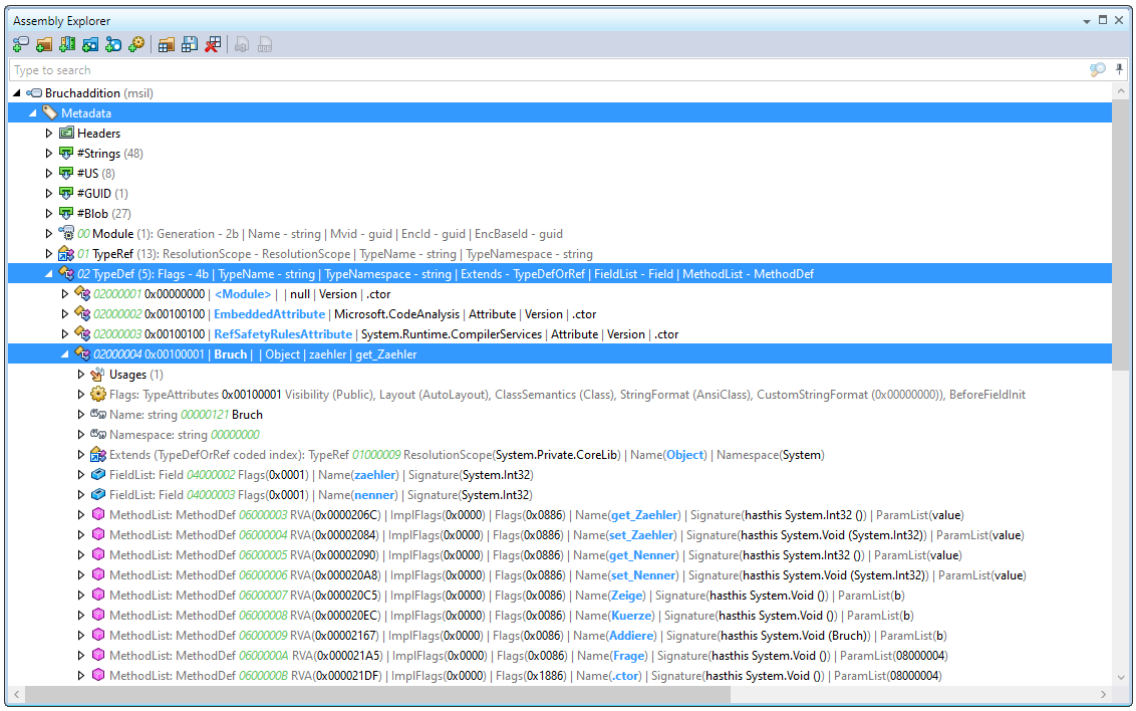
- die wesentlichen Bestandteile einer veröffentlichten Anwendung,
- die Träger von Versionsangaben,
- die Träger von Sicherheitsmerkmalen (z. B. Einschränkung der Verwendung auf Administratoren).

Wer möglichst schnell in die Programmierung einsteigen möchte, kann den Abschnitt 2.4 an dieser Stelle verlassen und die vertiefende Beschäftigung mit den Assembly-Details verschieben.

2.4.1 Typ-Metadaten

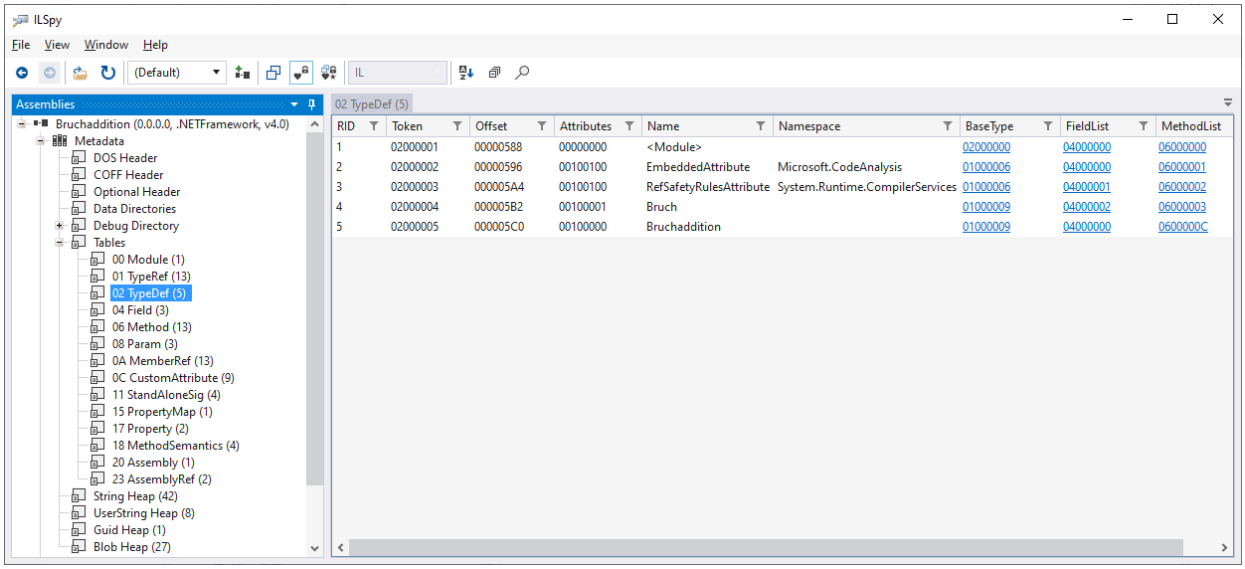
Ein Assembly enthält **Typ-Metadaten**, die alle im Assembly implementierten sowie die vom Assembly referenzierten Typen beschreiben und von der Laufzeitumgebung bei der Programmausführung für die Verwaltung der Typen genutzt werden (siehe Abschnitt 2.5).

Im Assembly **Bruchaddition.dll** sind z. B. die folgenden Metadaten zu der dort implementierten Klasse `Bruch` vorhanden:

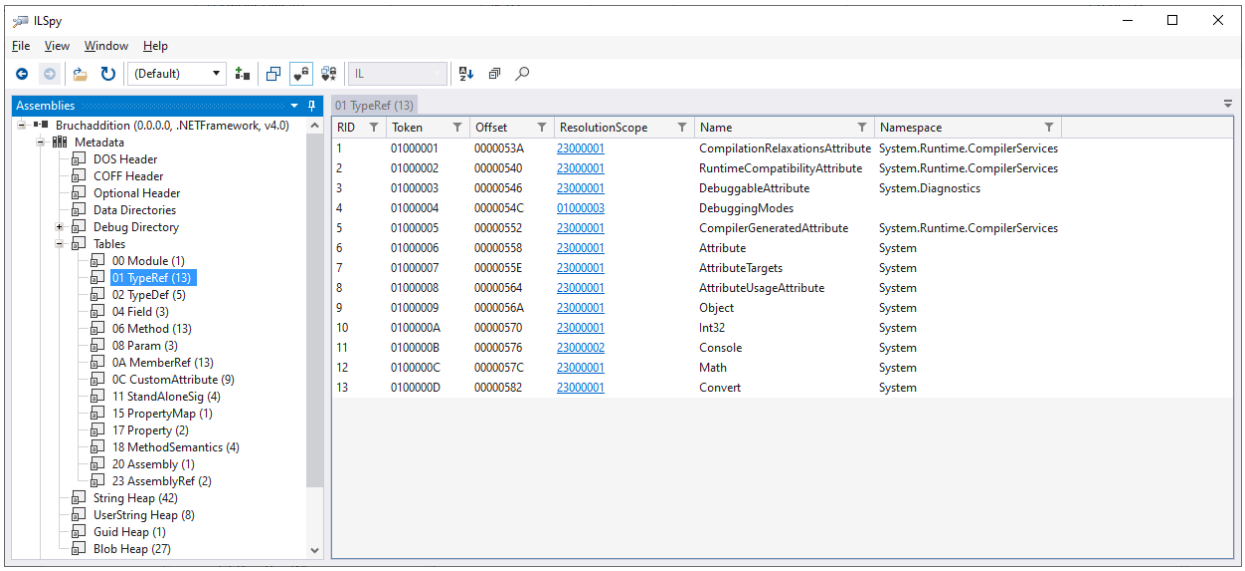


Zur Anzeige der Typ-Metadaten wird hier das Programm **dotPeek** der Firma JetBrains verwendet (siehe Abschnitt 3.4.2). Es ist ebenso kostenlos verfügbar wie das bereits mehrfach verwendete Programm ILSpy mit ähnlichen Kompetenzen zur Assembly-Inspektion (siehe Abschnitt 3.4.1).

In der IL-Darstellung eines Assemblies ist die Anordnung der Metadaten in Tabellen deutlich zu erkennen. Neben den *Definitionstabellen* mit Metadaten zu den im Assembly implementierten Klassen



enthalten die Metadaten auch *Referenztabellen* mit Informationen zu den *fremden* Klassen, die im Assembly benutzt (referenziert) werden:



2.4.2 Manifest mit Assembly-Metadaten

Neben dem IL-Code und den Typ-Metadaten enthält ein Assembly das sogenannte **Manifest** mit den **Assembly-Metadaten**. Anschließend werden wichtige Assembly-Metadaten beschrieben.

2.4.2.1 Einfacher Name

Seinen einfachen Namen übernimmt ein Assembly von der Ausgabedatei des Compilers (ohne Namenserweiterung). Wir haben den Ausgabedateinamen im Abschnitt 2.3.1 im Compiler-Aufruf angegeben. In der Regel legt MSBuild diesen Namen fest und verwendet dabei den Projektnamen als Voreinstellung (siehe Kapitel 3). Das im Abschnitt 3.4.1 beschriebene Programm ILSpy zur Analyse von Assemblies enthält in der Datei **ILSpy.dll** das Anwendungs-Assembly, das also den einfachen Namen *ILSpy* besitzt. Statt über das Assembly **Bruchaddition.dll** hätte oben über das Assembly `Bruchaddition` in der Datei **Bruchaddition.dll** geschrieben werden müssen.

Allzu kurz sollte der Name eines Assemblies nicht sein, weil damit die Gefahr von Namenskonflikten steigt. Die folgenden Assembly-Dateinamen aus der BCL von .NET 7.0 bzw. aus dem Installationsordner von ILSpy 8.0.0.7345 realisieren ungefähr die im Abschnitt 2.6.1 vorgeschlagenen Regeln für die Bezeichnung von Namensräumen:

- **System.Private.CoreLib.dll**
- **ICSharpCode.TreeView.dll**

2.4.2.2 Versionsangaben

Durch eine gründliche Versionsverwaltung werden Probleme mit Versionsunverträglichkeiten vermieden. Ein Assembly merkt sich von einem vorausgesetzten Assembly nicht nur den Namen, sondern auch die Version, und die ist sehr differenziert anzugeben:[1]

Major.Minor.Build.Revision

Die Autoren des oben als Beispiel verwendeten Assemblies ILSpy haben sich offenbar auf zwei Kriterien beschränkt und die Version 8.0.0.7345 vergeben.

---

[1] https://learn.microsoft.com/en-us/troubleshoot/developer/visualstudio/general/assembly-version-assembly-file-version

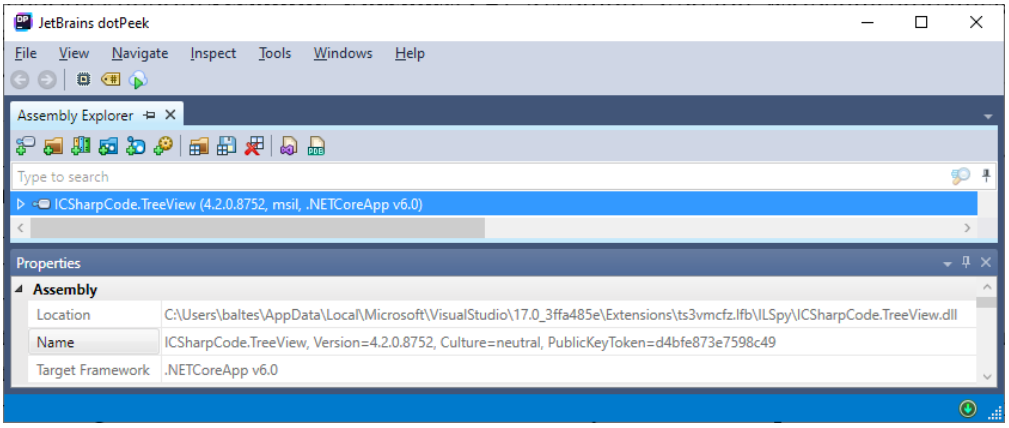
Microsoft hat sich für Assemblies gleich *drei* Versionsangaben ausgedacht:

- Die bisher diskutierte **Assembly-Version** (technisch realisiert über das **AssemblyVersionAttribute**)[1]
  Beim Laden eines referenzierten Assemblies während der Programmausführung interessiert sich die Laufzeitumgebung (CLR) ausschließlich für die Assembly-Version.
- Die **Assembly-Dateiversion** (technisch realisiert über das **AssemblyFileVersionAttribute**)
  Im .NET Framework wird bei der Suche nach einem referenzierten *signierten* Assembly (vgl. Abschnitt 2.4.2.4) eine exakte Versionsübereinstimmung verlangt, wenn nicht per Anwendungskonfigurationsdatei eine Ersatzlösung zugelassen ist. Wenn unter diesen Voraussetzungen ein Bibliotheks-Assembly …
  - von vielen anderen Assemblies genutzt wird
  - und oft eine neue Assembly-Version erhält,

  dann müssen die nutzenden Assemblies jedes Mal neu übersetzt werden, wenn keine Ausnahme explizit per Anwendungskonfigurationsdatei zugelassen ist. Diesen Aufwand kann man folgendermaßen vermeiden:
  - Solange ein verbessertes Bibliotheks-Assembly abwärts-kompatibel ist, bleibt die Assembly-Version unverändert.
  - Wenn ein renoviertes Bibliotheks-Assembly *nicht* abwärtskompatibel ist, dann muss es eine neue Assembly-Version erhalten.

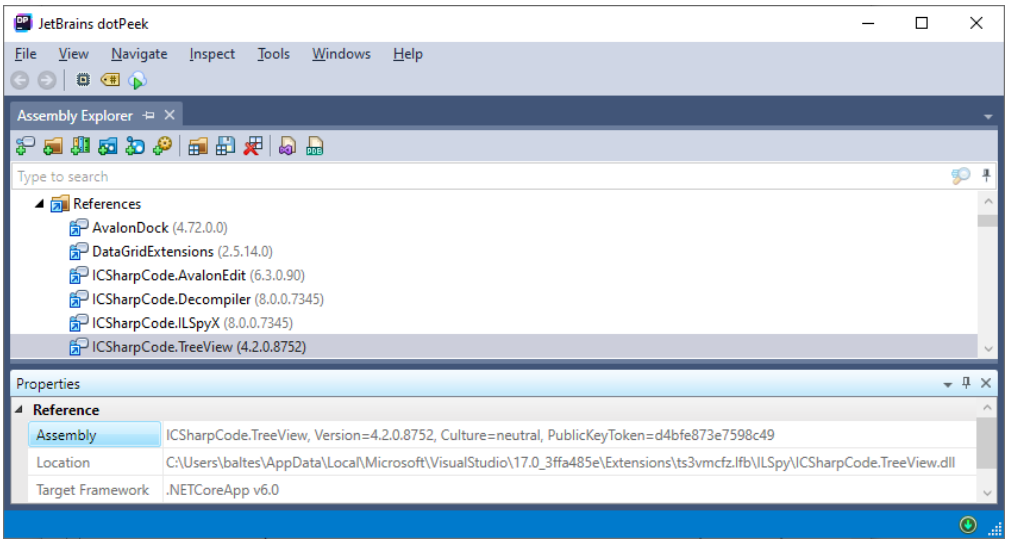
  Durch diese Praxis taugt aber die Assembly-Version z. B. nicht mehr zur Identifikation der aktuellen Variante. Für diese Aufgabe wird im .NET Framework oft die Assembly-*Datei*-version genutzt, die z. B. per Windows-Explorer ermittelt werden kann.

  Seit .NET Core bzw. .NET 5 sind die Regeln beim Laden von Assemblies gelockert, und es wird generell eine aktuellere Version akzeptiert. Damit entfällt der im .NET Framework bestehende Grund für Abweichungen zwischen der Assembly-Version und der Assembly-Dateiversion. Wie eine Inspektion mit dotPeek (vgl. Abschnitt 3.4.2) ergibt, stimmen beim Bibliotheks-Assembly **ICSharpCode.TreeView**,

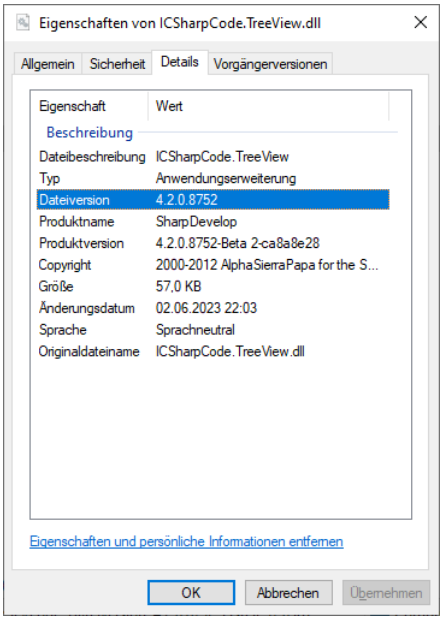


  das vom ausführbaren Assembly des Analyseprogramms **ILSpy** (vgl. Abschnitt 3.4.1) referenziert wird,

[1] Attribute sind spezielle Klassen, mit denen wir uns im Kapitel 14 beschäftigen werden. Über die Vergabe der Versionsattribute informiert der Abschnitt 14.3.



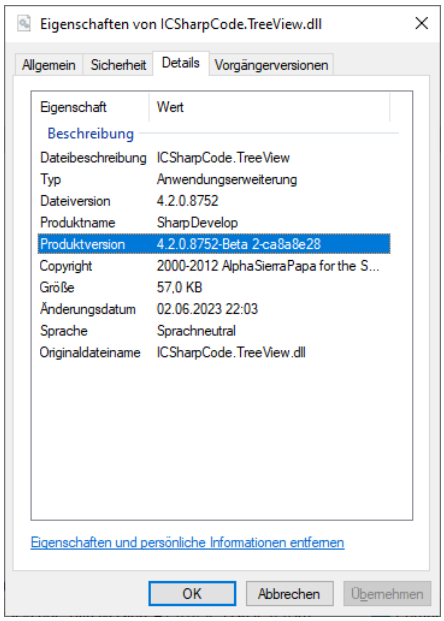
die Assembly-Version und die Assembly-Dateiversion überein:



- Die **Assembly-Produktversion** (technisch realisiert über das **AssemblyInformationalVersionAttribute**)
  Der Windows-Explorer zeigt zu einer Assembly-Datei neben der Dateiversion noch die Produktversion an. Sie dient keinen technischen Zwecken, sondern zur Kommunikation bzw. Vermarktung. Während die Assembly-Version und die Assembly-Dateiversion das Schema

  Major.Minor.Build.Revision

  einhalten müssen, kann die Assembly-Produktversion aus einem beliebigen Text bestehen, z. B.:



Die Assembly-Version ist obligatorisch und wird bei einer fehlenden Angabe auf 0.0.0.0 gesetzt. Eine fehlende Datei- oder Produktversion erhält den Wert der Assembly-Version.

2.4.2.3 Kulturinformation

Für die sprachlichen und kulturellen Lokalisierungen in Zeichenfolgen und Bildern ist die *neutrale* Ausprägung üblich, sofern es sich nicht um ein sogenanntes *Satelliten-Assembly* mit Lokalisierungen für eine spezielle Sprache mit länderspezifischen Besonderheiten handelt (z. B. de-ch).[1] Wenn kein Satelliten-Assembly mit Lokalisierungen vorhanden ist, dann wird die neutrale Variante verwendet.

2.4.2.4 Sicherheitsmerkmale

Im .NET Framework ist das Signieren eines Assemblies von Bedeutung, weil nur signierte Assemblies im sogenannten *Global Assembly Cache* (GAC) eines lokalen Rechners zur Verwendung durch diverse Anwendungen abgelegt werden dürfen. Ein signiertes Assembly enthält in seinem Manifest:[2]

- eine digitale Signatur
  Dazu wird der Hash-Wert des Assemblies durch den privaten Schlüssel des Herausgebers signiert.
- den öffentlichen Schlüssel des Herausgebers

Seit .NET Core hat das Signieren von Assemblies erheblich an Bedeutung verloren, weil es keinen GAC mehr gibt. Allerdings sorgt der öffentliche Schlüssel des Herausgebers als Bestandteil des voll qualifizierten Namens immer noch für die eindeutige Identifikation eines Assemblies (siehe Abschnitt 2.4.2.5), und die Assemblies in der .NET – Laufzeitbibliothek sind signiert.[3]

Unter Windows kann für ein Assembly eingestellt werden, dass es nur von Benutzern mit administrativen Rechten ausgeführt werden darf. Genau genommen befindet sich diese Einstellung nicht im Assembly-, sondern im Anwendungs-Manifest (Albahari 2022, S. 743).

---

1 https://learn.microsoft.com/en-us/dotnet/core/extensions/create-satellite-assemblies

2 Die Aufzählungspunkte sind mit Vorkenntnissen über Sicherheitstechnik zu verstehen und dürfen ignoriert werden.

3 https://learn.microsoft.com/en-us/dotnet/standard/assembly/sign-strong-name

2.4.2.5 Identität

Bei der Suche nach einem benötigten Assembly verwendet die CLR (die Laufzeitumgebung, siehe Abschnitt 2.5) die sogenannte *Identität* des Assemblies. Darunter ist der voll-qualifizierte Name zu verstehen, der vier Bestimmungsstücke enthält (Albahari 2022, S. 747ff):

- den einfachen Namen (z. B. ILSpy)
- die Assembly-Version (z. B. 8.0.0.7345)
- die Kultur (z. B. neutral)
- bei signierten Assemblies (siehe Abschnitt 2.4.2.4) den öffentlichen Schlüssel des Herausgebers (z. B. d4bfe873e7598c49), ansonsten den Ersatzwert **null**

Das als Beispiel betrachtete Anwendungs-Assembly des .NET – Programms ILSpy hat also den voll-qualifizierten Namen:

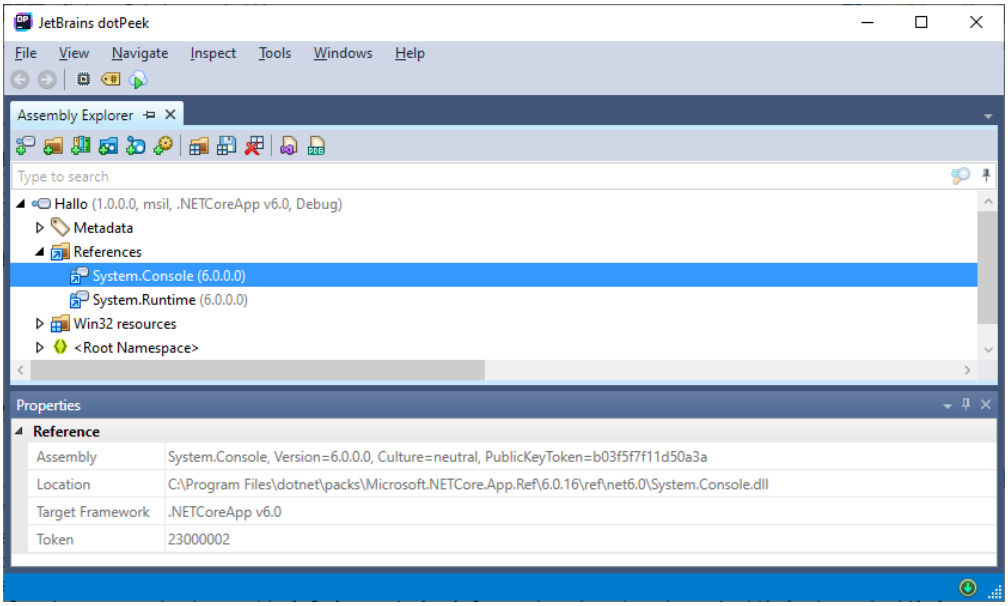
```
ILSpy, Version=8.0.0.7345, Culture=neutral, PublicKeyToken=d4bfe873e7598c49
```

Bei einem signierten Assembly spricht man von einem *starken Namen* (engl.: *strong name*).[1] Seit .NET Core bzw. .NET 5 hat das Signieren von Assemblies erheblich an Bedeutung verloren. Allerdings sorgt der öffentliche Schlüssel des Herausgebers als Bestandteil des voll qualifizierten Namens immer noch für die eindeutige Identifikation eines Assemblies.

2.4.2.6 Abhängigkeit von anderen Assemblies

Im Manifest eines Assemblies ist dokumentiert, welche anderen Assemblies referenziert (verwendet) werden, wobei wiederum exakte Versionsangaben für den störungsfreien Betrieb sorgen. Im Unterschied zum .NET Framework wird ab .NET 5 bei der Programmausführung von der Laufzeitumgebung (Common Language Runtime, CLR, siehe Abschnitt 2.5) die im Manifest eingetragene oder eine aktuellere Version akzeptiert und geladen.

Das Analyseprogramm dotPeek (vgl. Abschnitt 3.4.2) kann die von einem Assembly benötigten Referenzen anzeigen, z. B.:



Wir haben im Abschnitt 2.3.1 für das Assembly `Bruchaddition` aus didaktischen Gründen die vorausgesetzten Assemblies im Aufruf des Roslyn-Compilers angegeben. Wie man für ein selbst entwickeltes Assembly auf zeitgemäße und bequeme Art die voreingestellte Liste der Abhängigkeiten erweitert, erfahren Sie im Abschnitt 3.3.9.

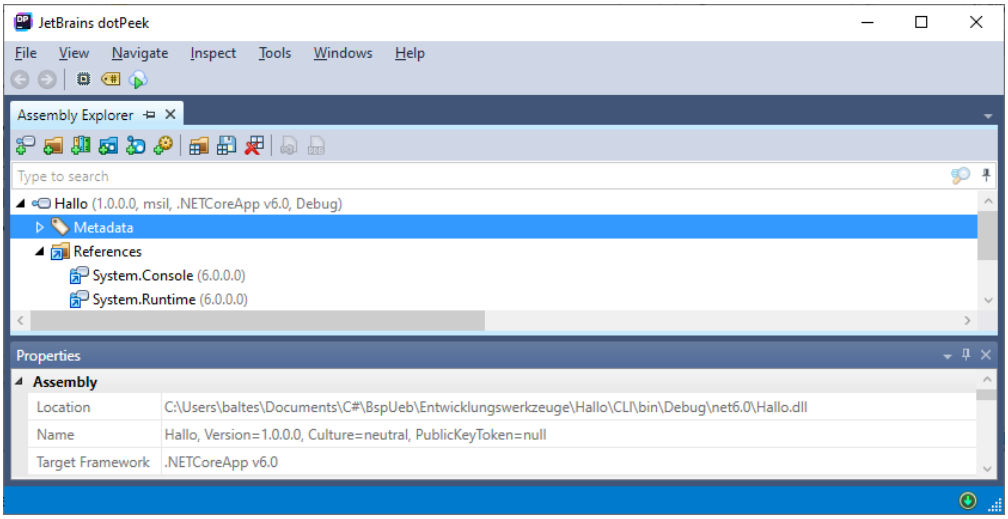
---

[1] https://learn.microsoft.com/en-us/dotnet/standard/assembly/sign-strong-name

2.4.2.7 Assembly-Eigenschaften festlegen oder ermitteln

Für ein selbst erstelltes Assembly legt man die Manifest-Attribute in der Regel mit Hilfe der Entwicklungsumgebung fest (im Visual Studio über den Menübefehl **Projekt > Eigenschaften**).

Viele Manifest-Attribute eines *fremdentwickelten* Assemblies bringt man mit einem Analyseprogramm wie dotPeek (vgl. Abschnitt 3.4.2) in Erfahrung, z. B. die Identität (bzw. den voll-qualifizierten Namen, vgl. Abschnitt 2.4.2.5):



Im Abschnitt 2.4.2.6 haben wir dotPeek zur Anzeige der Abhängigkeiten von anderen Assemblies verwendet.

2.4.3 Verweis-Assemblies

Von einem *Verweis- bzw. Referenz-Assembly* spricht man dann, wenn …[1]

- seine Typ-Metadaten nur das API enthalten, also die privaten Member der Typen verbergen,
- und keine Implementationen vorhanden sind, aber auf die implementierenden Assemblies verwiesen wird

Während ein Verweis-Assembly für die Zwecke des Compilers ausreicht, werden zur Programmausführung natürlich die referenzierten implementierenden Assemblies benötigt.[2] Ein reguläres, auch Implementationen enthaltendes Assembly wird gelegentlich als *Implementierungs-Assembly* bezeichnet.

Ein Grund für die Beschäftigung mit Verweis-Assemblies besteht in den bei vielen BCL-Klassen zu beobachtenden widersprüchlichen Angaben zum implementierenden Assembly. Die folgende Anweisung[3]

```
Console.WriteLine(typeof(Math).Assembly.Location);
```

liefert zur Klasse **Math** im Namensraum **System** als Pfad zur implementierenden Assembly-Datei:

---

1 https://learn.microsoft.com/en-us/dotnet/standard/assembly/reference-assemblies

2 Im Abschnitt 2.3.1 hat sich gezeigt, dass man beim direkten Aufruf des Compilers sowohl das Verweis-Assembly **System.Runtime.dll** als auch das Implementations-Assembly **System.Private.CoreLib.dll** angeben muss. Weil der Compiler bei der Projekteerstellung (per dotnet-CLI oder Visual Studio) stets indirekt angesprochen wird (unter Vermittlung durch MSBuild), können wir die Frage nach den dabei benötigten Referenzen ignorieren.

3 Per **typeof**-Operator wird zur Klasse **Math** im Namensraum **System** ein beschreibendes **Type**-Objekt ermittelt, das eine Eigenschaft namens **Assembly** besitzt, die auf ein Objekt der Klasse **Assembly** zeigt. Dieses Objekt repräsentiert das die Klasse **Math** implementierende Assembly und kann über seine **Location**-Eigenschaft nach dem Pfad zur Assembly-Datei befragt werden.

C:\Program Files\dotnet\shared\Microsoft.NETCore.App\7.0.9\System.Private.CoreLib.dll

In der Online-Dokumentation zur Klasse **Math** wird aber das Assembly **System.Runtime.dll** genannt:[1]

Math Class

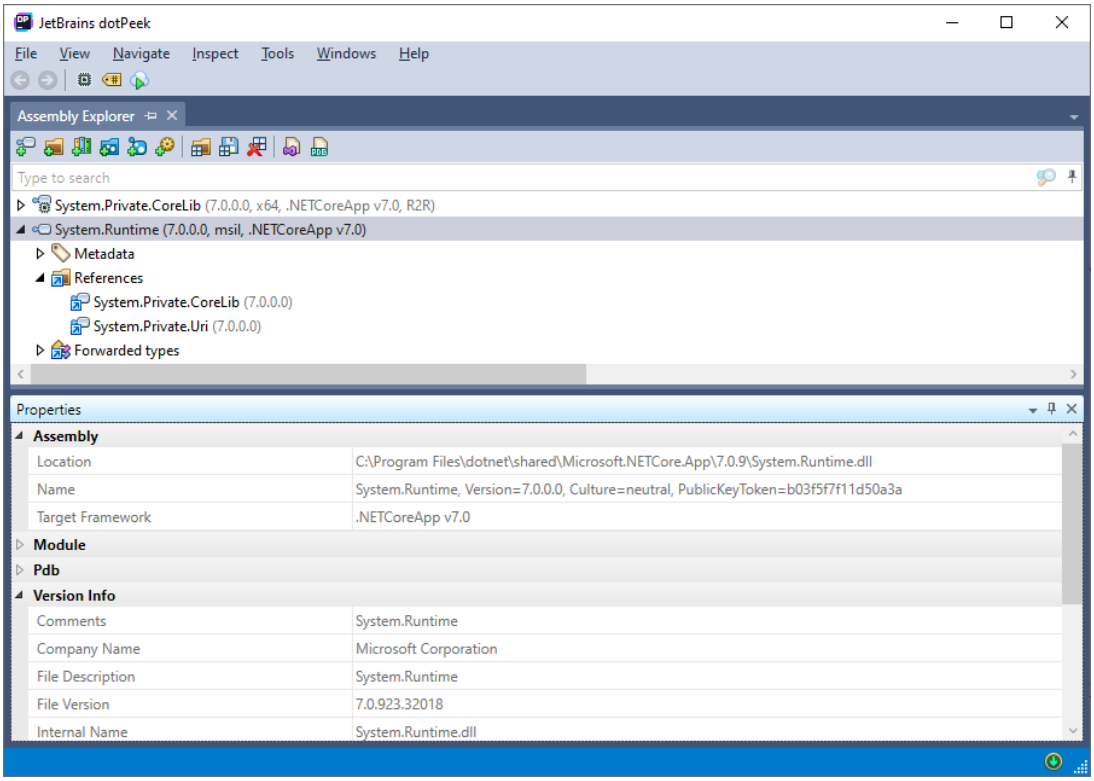
Reference Feedback

Definition

Namespace: System
Assembly: System.Runtime.dll

Provides constants and static methods for trigonometric, logarithmic, and other common mathematical functions.

Eine Inspektion mit dotPeek zeigt, dass es sich bei **System.Runtime.dll** um ein *Verweis*-Assembly handelt:

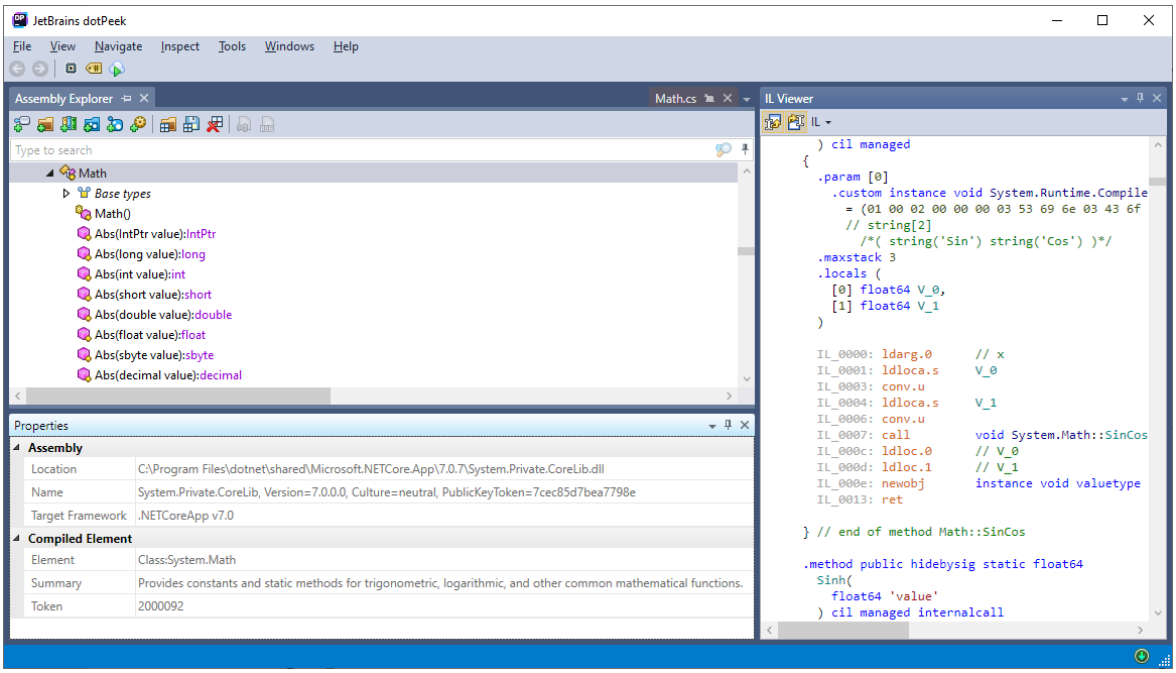


Es sind Metadaten und vor allem Referenzen auf die implementierenden Assemblies **System.Private.CoreLib** und **System.Private.Uri** vorhanden, aber kein IL-Code.

Im Assembly **System.Private.CoreLib** ist die Klasse **Math** implementiert:

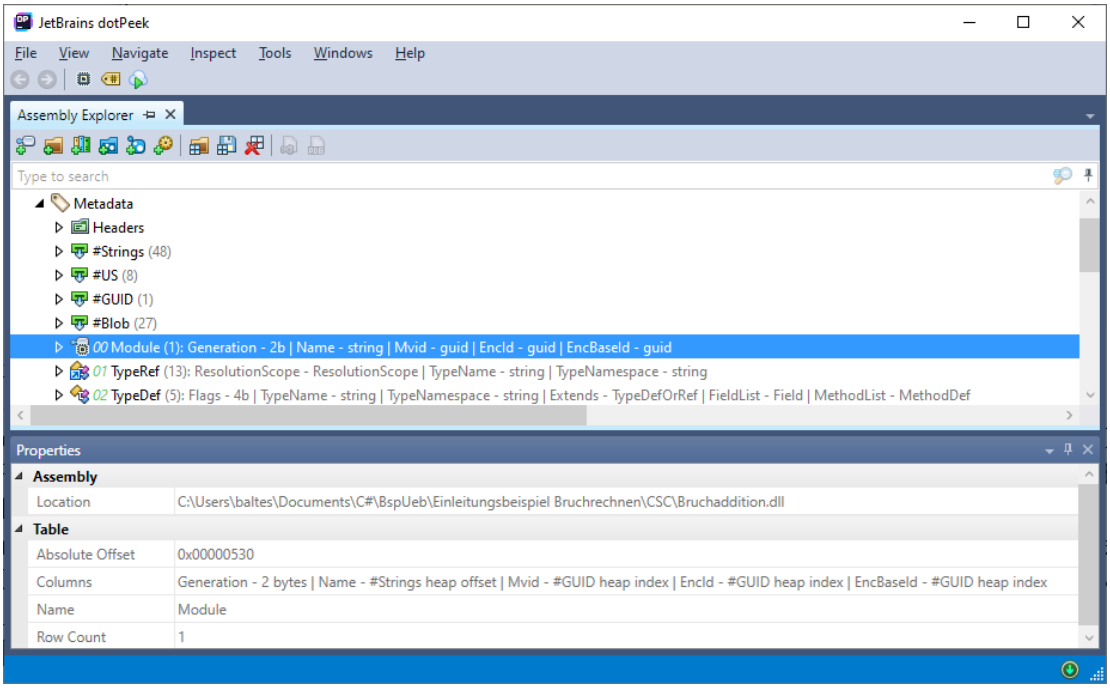
---

1 https://learn.microsoft.com/en-us/dotnet/api/system.math



2.4.4 Module

Im .NET *Framework* kann ein Assembly aus *mehreren Moduldateien* mit IL-Code und zugehörigen Typ-Metadaten bestehen, wobei das Manifest nur in *einer* Datei vorhanden ist und die Namen der weiteren zum Assembly gehörenden Dateien enthält. Mit .NET Core bzw. .NET 5 wurden die Multidatei-Assemblies abgeschafft, aber die Module sind noch vorhanden. Nun besteht jedes Assembly aus einer einzigen Datei und aus einem einzigen Modul, und dieser Modul-Container im Assembly-Container enthält den IL-Code der Typen, die Typ-Metadaten, die Assembly-Metadaten (das Manifest) sowie optionale Ressourcen. Bei der Assembly-Beschreibung (hier durch das Analyseprogramm dotPeek) taucht also regelmäßig ein Modul auf, z. B.:



2.4.5 Bestandteile von .NET - Anwendungen

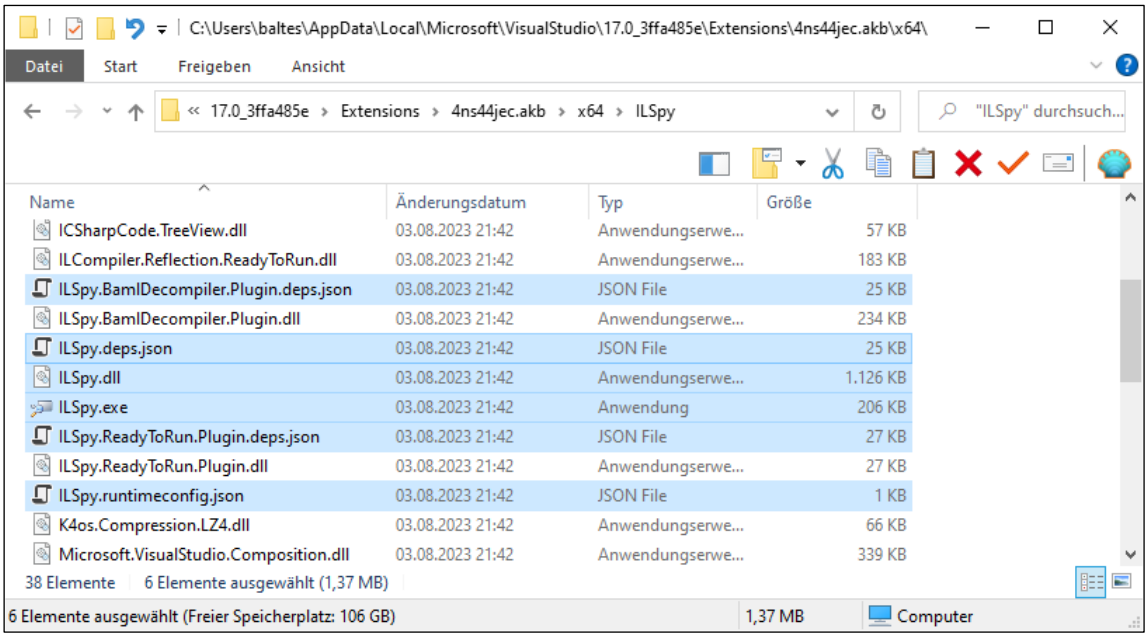
Ein einzelnes Assembly kann beliebig viele Klassen enthalten und außerdem auf die mächtige .NET – Basisbibliothek mit zahlreichen Klassen zurückgreifen, sodass durchaus nützliche Einzel-Assembly-Anwendungen möglich sind. Oft werden aber Funktionen bzw. Klassen aus weiteren, bereits bestehenden Bibliotheken benötigt. Bei der Neuentwicklung von zusammengehörigen Klassen kann die Unterbringung in einem Bibliotheks-Assembly sinnvoll sein, um die Nutzung in mehreren Anwendungen zu ermöglichen. Folglich besteht eine praxistaugliche .NET – Anwendung meist aus …

- einem Anwendungs-Assembly (mit Startklasse)
  Unter Windows ist nach einer Erstellung per MSBuild neben dem ausführbaren Assembly mit IL-Code auch ein Maschinencode-Programm in einer **exe**-Datei vorhanden. Zum Starten genügt dann z. B. ein Doppelklick auf die **exe**-Datei.
- mehreren Bibliotheks-Assemblies
  Benötigte Bibliotheks-Assemblies, die nicht zur Laufzeitumgebung gehören, befinden sich meist im Anwendungsordner.
- Laufzeit-Konfigurationsdateien
  Zum Anwendungs-Assembly **ILSpy** existiert die obligatorische Konfigurationsdatei **ILSpy.runtimeconfig.json**, der die zur Ausführung des Programms erforderliche Version der .NET – Laufzeitumgebung zu entnehmen ist. Zu jedem Assembly existiert eine Konfigurationsdatei mit Informationen zu den referenzierten Bibliotheks-Assemblies. Im Fall des Assemblies **ILSpy** lautet der Dateiname **ILSpy.deps.json**.

Das .NET – Programm **ILSpy**, das wir mehrfach zur Assembly-Inspektion eingesetzt haben, besteht z. B. aus …

- dem Anwendungs-Assembly **ILSpy.dll** mit dem Windows-Starthelfer **ILSpy.exe**,
- über 30 Bibliotheks-Assemblies
- und zwei Konfigurationsdateien.

Wir werden das Programm im Abschnitt 3.4 als Erweiterung zu der im Manuskript bevorzugten Entwicklungsumgebung Visual Studio installieren:



2.5 CLR

Wie im Abschnitt 2.2 berichtet wurde, kommen in .NET 7 zwei verschiedene Laufzeitumgebungen zum Einsatz:

- **CoreCLR** (*Core Common Language Runtime*) auf Arbeitsplatzrechnern und Servern unter Linux, macOS und Windows
- **Mono-Runtime** für MAUI-Anwendungen unter Android, iOS und macOS sowie für Blazor WASM

Im Manuskript wird ausschließlich die CoreCLR verwendet, und diese Laufzeitumgebung ist gemeint, wenn über *die CLR* geschrieben wird.

2.5.1 JIT-Übersetzung und weitere Aufgaben der CLR

Bislang haben Sie erfahren, dass aus dem C# - Quellcode durch einen Compiler (z. B. **csc.dll**) ein Assembly mit Zwischencode (IL) erzeugt wird. Beim Programmstart ist eine weitere Übersetzung in den Maschinencode der aktuellen CPU erforderlich, die von der CLR erledigt wird. Dazu besitzt die CLR einen JIT-Compiler (*Just In Time*), der Leistungseinbußen aufgrund der zweistufigen Übersetzungsprozedur minimiert, indem er …

- nur den zur Ausführung anstehenden IL-Code übersetzt
- und den erzeugten Maschinencode speichert für den Fall einer erneuten Verwendung.

Sorgen um mangelnde Performanz aufgrund der zweistufigen Übersetzung sind unbegründet. Wenn Benchmark-Ergebnisse einen Leistungsvorteil der Maschinencode-Compiler (z. B. bei den Programmiersprachen C/C++) berichten, dann geht es meist um mathematische Berechnungen (z. B. Mandelbrot-Aufgaben).[1] Diese Ergebnisse sind wenig aussagekräftig in Bezug auf Leistungskriterien, die komplexe objektorientierte Programme zu erfüllen haben (z. B. Durchsatz eines Webservers). Ein Grund für das günstige Leistungsverhalten der Zwischencode-Sprachen C# und Java ist darin zu sehen, dass die meist sehr aktuelle CLR die lokal vorhandene CPU besser kennt und z. B. deren Befehlserweiterungen besser ausnutzt als ein Maschinencode-Compiler, der ...

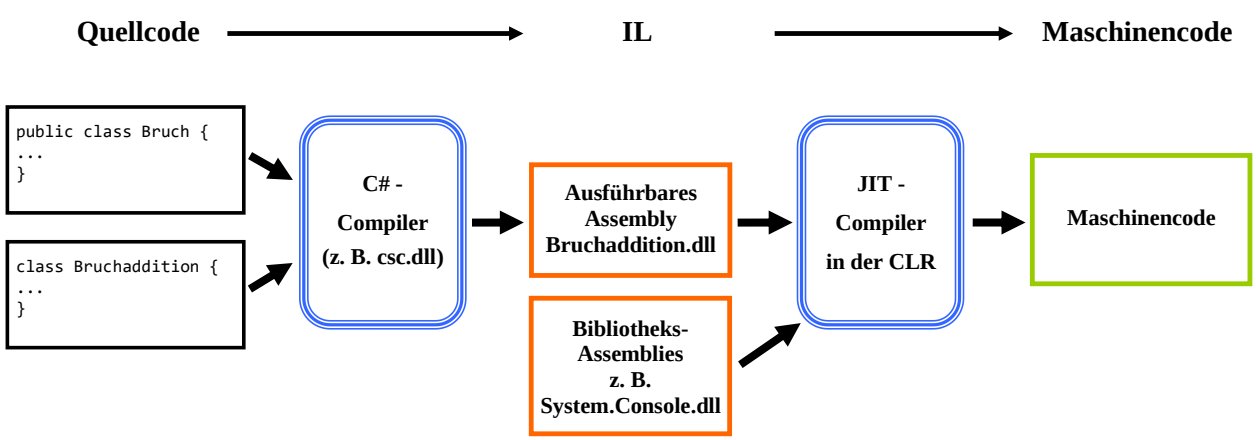
- zum Zeitpunkt der Übersetzung älter war als die ausführenden Prozessoren,
- auf maximale Kompatibilität Wert gelegt und manche CPU-Spezifika nicht unterstützt hat, damit sein Produkt auf möglichst vielen CPUs ablauffähig ist.

Dass es sich bei der .NET – Laufzeitumgebung um eine *Common Language* Runtime handelt, bedurfte eigentlich keiner terminologischen Hervorhebung, weil alle .NET – Programmiersprachen in denselben IL-Code übersetzt werden, den die CLR dann auszuführen hat.

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

---

[1] Siehe z. B. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/csharpcore-gpp.html



Die CLR hat bei der Verwaltung bzw. Unterstützung von .NET - Anwendungen neben der Übersetzung von IL-Code in Maschinencode noch weitere Aufgaben zu erfüllen, z. B.:[1]

- **Lokalisation und Laden von referenzierten Assemblies**
  Die von einem Assembly des Programms referenzierten Assemblies befinden sich …
  - entweder im Anwendungsordner,
  - oder an einem Ort, den eine Konfigurationsdatei mit der Namenserweiterung **deps.json** benennt,
  - oder in der Laufzeitumgebung.

  Über komplexere Fälle (z. B. Anwendung mit Plugin-System und benutzer-spezifischen Assemblies) informiert Albahari (2022, S. 759ff).
- **Verifikation des IL-Codes**
  Irreguläre Aktionen einer .NET - Anwendung werden vom Verifier der CLR verhindert. Das macht .NET - Anwendungen sehr stabil.
- **Unterstützung bei der Speicherverwaltung**
  Überflüssig gewordene Objekte werden vom Garbage Collector (Müllsammler) der CLR automatisch aus dem Speicher entfernt.
- **Unterstützung beim Multithreading**
  Um Mehrkern-CPUs auszulasten, muss ein Programm mehrere parallele Ausführungspfade (Threads) enthalten.
- **Behandlung von Ausnahmefehlern**
  Die Ausführung einer Methode kann durch besondere Umstände scheitern (z. B. Ausfall der Netzwerkverbindung). Bei der Behandlung solcher Fehler ist die CLR beteiligt (siehe Fruja & Börger 2005).
- **Laufzeitbibliotheken**
  Nach der Überblicksdarstellung im Abschnitt 2.2.2 sind die BCL (Base Class Library) und ihre anwendungstyp-spezifischen Ergänzungen ebenfalls der CLR zuzurechnen (siehe Abschnitt 2.6). Eine Desktop-Anwendung kann z. B. in .NET 7 auf ca. 250 Bibliotheks-Assemblies zugreifen, die insgesamt eine enorme Fülle von Klassen und anderen Typen mit ausgereiften Lösungen enthalten.

Weil die CLR den IL-Code kontrolliert und unterstützt, spricht man von *verwaltetem Code*. Der von einem klassischen Compiler (z. B. für die Programmiersprache C++) erstellte Maschinencode wird als *nicht-verwaltet* bezeichnet, weil er sich z. B. selbst um die Freigabe des Speichers von überflüssig gewordenen Objekten kümmern muss.

---

[1] In der Liste tauchen Begriffe auf, die später ausführlich behandelt werden und daher an dieser Stelle keine Kopfschmerzen auslösen sollten (Garbage Collector, Multithreading, Ausnahmefehler).

Wer möglichst schnell in die Programmierung einsteigen möchte, kann den Abschnitt 2.5 an dieser Stelle verlassen.

2.5.2 AOT-Übersetzung

Der .NET – Erstellungsprozess erlaubt auch eine sogenannte *AOT* – Übersetzung (*Ahead Of Time*) in den Maschinencode einer bestimmten Kombination aus CPU-Architektur (z. B. ARM, x64) und Betriebssystem (Microsoftbegriff: *Zielruntime* bzw. *target runtime*), was folgende Vorteile bietet:

- Der Anwendungsstart und die erste Verwendung von Code werden beschleunigt, weil die Übersetzung von IL-Code in Maschinencode entfällt oder nur noch in Ausnahmefällen erforderlich ist.
- Unter dem Betriebssystem iOS für mobile Geräte der Firma Apple ist die AOT-Übersetzung unumgänglich, weil hier kein dynamisch erzeugter Code ausgeführt werden darf.

Seit .NET 5 hat sich unter dem Namen *R2R* (*Ready To Run*) eine AOT-Technik etabliert, die eine Kombination aus IL-Code und Maschinencode verwendet, um die Vorteile beider Ansätze zu kombinieren. In .NET 7 ist unter dem Namen *NativeAOT* eine AOT-Technik dazugekommen, die komplett auf einen JIT-Compiler verzichtet. Die beiden Alternativen zu den nach wie vor dominanten, unter mehreren Betriebssystemen lauffähigen IL-Programmen werden anschließend beschrieben. Man kann die AOT-Technik als eine Option zum *Veröffentlichen* von Software auffassen, und wir werden im Kapitel 16 im Zusammenhang mit diesem Thema die Erstellung einer R2R-Anwendung demonstrieren (siehe Abschnitt 16.2).

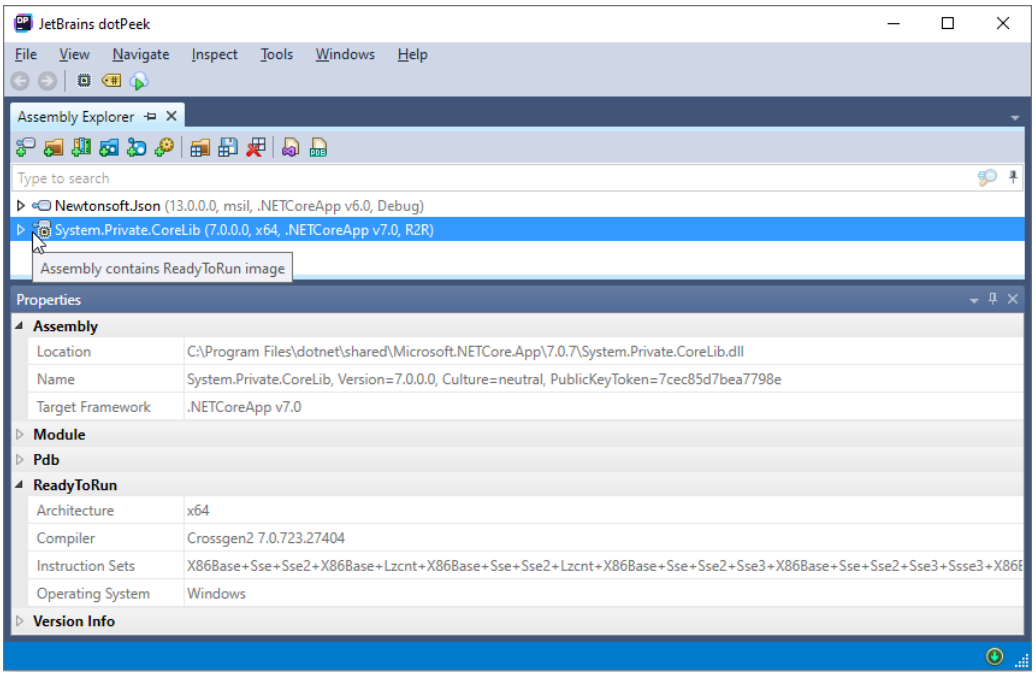
2.5.2.1 Programme mit vorbereiteter JIT-Übersetzung (ReadyToRun)

Das schon in der ersten .NET - Framework – Version enthaltene Programm **ngen.exe** (*Native Image Generator)* erstellt aus einem Assembly ein Maschinencode-Programm, das unter Windows ohne JIT-Compiler ausgeführt werden kann. Für uns relevanter ist das ab .NET 6 in der Version 2 verfügbare Werkzeug **crossgen2.exe**. Das in C# (statt C++) neu implementierte Programm liefert ein Assembly im sogenannten **R2R**-Format (**ReadyToRun**), das *zusätzlich* zum IL-Code auch Maschinencode enthält. Das steigert zwar den Platzbedarf, doch ist der JIT-Compiler weiterhin in der Lage, Code-Optimierungen für die lokal angetroffene CPU vorzunehmen. Er arbeitet per Voreinstellung *mehrstufig*, indem er zunächst den Code der AOT-Übersetzung verwendet und bei wiederholt benötigtem Code eine Optimierung vornimmt (engl.: *tiered compilation*).

Von **crossgen2.exe** wird der zweite Übersetzungsschritt (vom IL- in den Maschinencode) vorweggenommen und im Assembly gespeichert. Der JIT-Compiler in der CLR kommt aber trotzdem zum Einsatz:

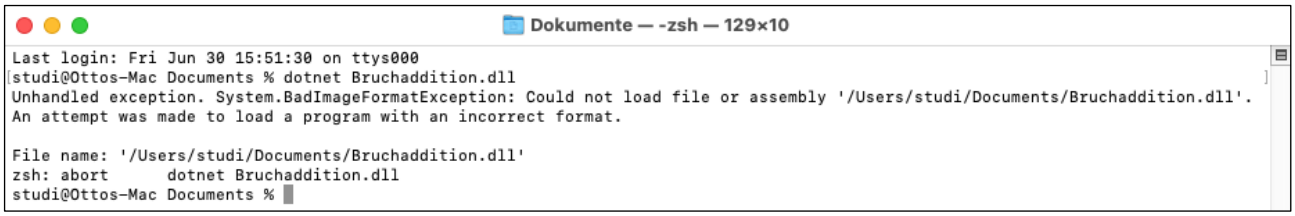
- Er nimmt für wiederholt benötigten Code Optimierungen im Rahmen der mehrstufigen Übersetzung vor.
- Er übersetzt IL-Code, der vom AOT-Compiler nicht verarbeitet werden konnte.

Naheliegenderweise kommt die R2R-Technik auch bei den Bibliotheks-Assemblies der Laufzeitumgebung zum Einsatz, was mit dem (schon im Abschnitt 2.4 verwendeten) Diagnoseprogramm dotPeek leicht nachzuweisen ist, z. B.:



Die Verwendung der R2R-Technik im CLR-Assembly **System.Private.CoreLib** ist am speziellen Symbol zu erkennen. In den Metadaten sind das Betriebssystem und die CPU-Architektur des R2R-Codes vermerkt (im Beispiel die Zielruntime **win-x64**).

Leider ist ein R2R-Assembly trotz des enthaltenen IL-Codes nur für die beim Erstellen anvisierte Zielruntime geeignet. Ein für die Zielruntime **win-x64** produziertes R2R-Konsolen-Assembly kann z. B. unter macOS *nicht* ausgeführt werden:



2.5.2.2 NativeAOT-Programme

Seit .NET 7 kann nativer Code *direkt* (ohne die nachträgliche Wandlung durch **crossgen2.exe**) erstellt werden. Man erhält ein eigenständiges Programm im Format der Zielplattform (in .NET 7 sind das nur Linux und Windows), das ohne JIT-Compiler auskommt und auf dem ausführenden Rechner keine dort installierte Laufzeitumgebung voraussetzt. Unverzichtbare CLR-Bestandteile (wie z. B. der Garbage Collector) werden in das Kompilat aufgenommen. Aus der NativeAOT-Veröffentlichung resultiert grundsätzlich eine *einzelne* Datei. Das ist aber auch bei der R2R-Veröffentlichung eine Option. Der wesentliche Unterschied zwischen den beiden AOT-Verfahren besteht darin, dass bei der NativeAOT-Veröffentlichung der native Code *nicht* durch IL-Code ergänzt wird. Dadurch wird das veröffentlichte Programm kleiner, aber nicht in jeder Hinsicht schneller.

Im Vergleich zur R2R-Technik bietet die NativeAOT-Technik u. a. die folgenden Vorteile:[1]

[1] https://devblogs.microsoft.com/dotnet/announcing-dotnet-7-preview-3/

- Kürzere Startzeit, weil kein JIT-Compiler geladen werden muss
- Kleineres Volumen im Arbeitsspeicher und auf dem Festspeicher
  Im Vergleich zu einem traditionellen IL-Assembly, das eine lokal installierte Laufzeitumgebung voraussetzt, ist das Festspeichervolumen aber erheblich größer.
- Verwendbarkeit auf Plattformen, die keinen JIT-Compiler erlauben (z. B. iOS)

Als Nachteile sind u. a. zu nennen:[1]

- In .NET 7 ist die NativeAOT-Unterstützung eingeschränkt auf:
  - Linux und Windows
  - Konsolenprogramme (mit Ausnahme von ASP.NET Core)
- Weil kein JIT-Compiler beteiligt ist, kann zur Laufzeit …
  - keine optimierende Anpassung an das lokale System vorgenommen werden,
  - kein IL-Code per Reflektion erstellt werden, um z. B. einen speziellen regulären Ausdruck beschleunigt auszuwerten.[2]
- Es können keine Assemblies dynamisch geladen werden, was z. B. bei Plugin-Lösungen erforderlich ist.

Das in C# programmierte Werkzeug **crossgen2** zum Erstellen von R2R-Assemblies (vgl. Abschnitt 2.5.2.1) ist …

- in .NET 6 als R2R-Anwendung realisiert
- in .NET 7 (jedenfalls unter Windows mit x64-CPU) als NativeAOT-Anwendung realisiert.

Die Wahl der NativeAOT-Lösung für .NET 7 wird so begründet:[3]

> Crossgen benefits heavily from Native AOT because it's a short-lived process and the startup overhead dominates the overall execution time.

Bei der im Abschnitt 16.2 beschriebenen Erstellung einer R2R-Konsolenanwendung mit **crossgen2** ergaben sich die folgenden handgestoppten Laufzeiten:

- .NET 6 (**crossgen2** mit R2R-Technik) 22 Sekunden
- .NET 7 (**crossgen2** mit NativeAOT-Technik) 18 Sekunden

2.6 BCL und Namensräume

Damit Programmierer nicht das Rad und ähnliche Dinge neu erfinden müssen, bieten die .NET – Laufzeitumgebungen eine umfangreiche Bibliothek mit fertigen Klassen (und sonstigen Typen) für nahezu alle Routineaufgaben, die als **Base Class Library** (**BCL**) bezeichnet wird.[4] Dass diese Bibliothek in *allen* .NET - Programmiersprachen zur Verfügung steht, ist für C# - Einsteiger noch wenig relevant. Bei der späteren Teamarbeit kann die sprachunabhängige .NET - Architektur jedoch bedeutsam werden, wenn Anhänger verschiedener Programmiersprachen zusammentreffen.

In der plattformübergreifenden .NET – Implementation (.NET 5 und Nachfolger), auf die wir uns im Manuskript beschränken, kommt zur Basisbibliothek für jeden Anwendungstyp (z. B. Windows-Desktop, ASP.NET Core) noch eine spezifische Bibliothek hinzu (siehe Abschnitt 2.2.2). Zur

---

1 https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/?tabs=net7

2 Um auf eine in der professionellen Programmierung relevante Einschränkung der AOT-Übersetzung hinweisen zu können, mussten in diesem Satz ambitionierte Programmiertechniken (Reflektion, reguläre Ausdrücke) erwähnt werden, über die Programmiereinsteiger noch nicht nachdenken müssen.

3 https://devblogs.microsoft.com/dotnet/announcing-dotnet-7-preview-3/#faster-lighter-apps-with-native-aot

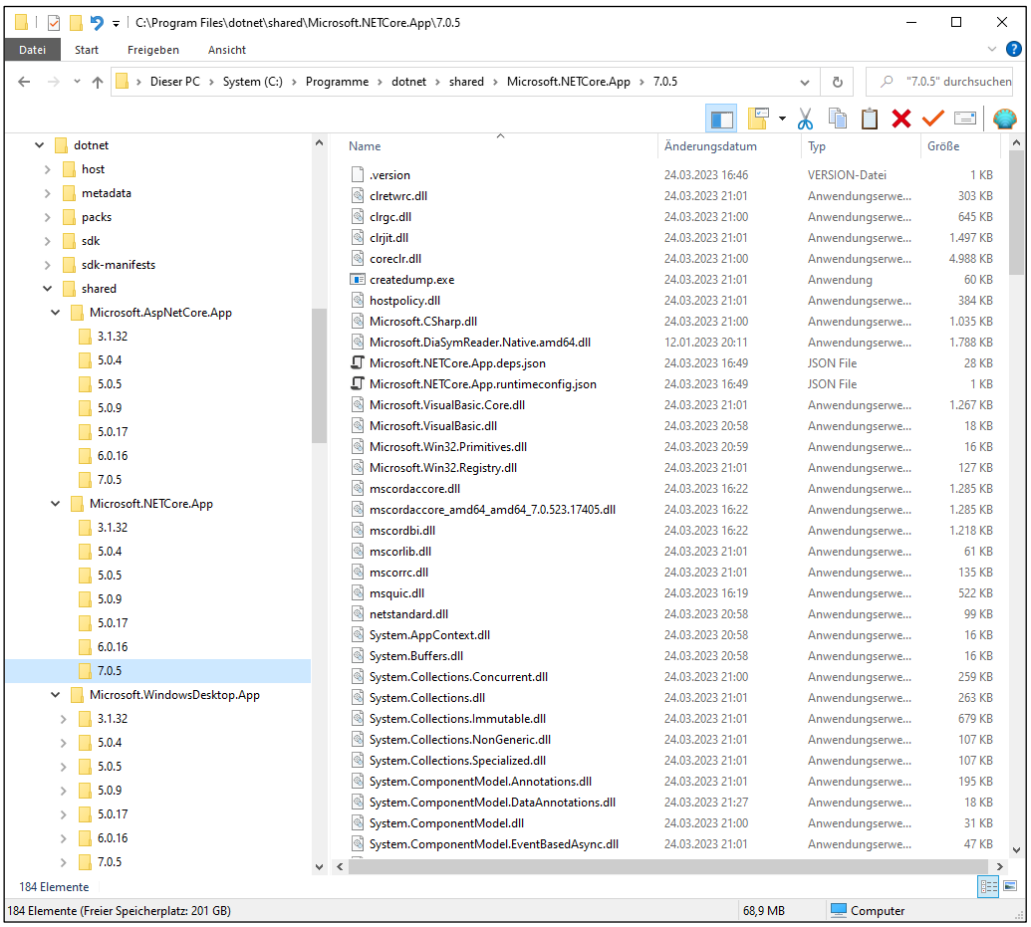
4 Statt von der *Base Class Library* wird im .NET Framework (vgl. Abschnitt 2.2.1) auch von der *Framework Class Library* (FCL) gesprochen.

Vereinfachung der Ausdrucksweise steht *BCL* ab jetzt für die Kombination aus der generellen und der anwendungsspezifischen Laufzeitbibliothek.

Unter Windows befindet sich im Ordner

C:\Program Files\dotnet\shared

für jeden installierten Anwendungstyp und jede installierte .NET – Version ein Unterordner mit Bibliotheks-Assemblies, z. B.:



Der Ordner **Microsoft.NETCore.App** enthält die generell benötigten Assemblies, und in den Ordnern der speziellen Anwendungstypen (z. B. **Microsoft.WindowsDesktop.App**) befinden sich die Ergänzungen. Eine Desktop-Anwendung kann in .NET 7 auf ca. 250 Bibliotheks-Assemblies zugreifen, die insgesamt eine enorme Fülle von Klassen und anderen Typen mit ausgereiften Lösungen implementieren.

Bevor man selbst eine Klasse oder Methode entwickelt, sollte man unbedingt die BCL auf die Existenz einer Lösung untersuchen, denn die BCL-Lösungen ...

- sind leistungsoptimiert und sorgfältig getestet,
- werden ständig weiterentwickelt.

Durch die Verwendung der BCL steigert man in der Regel die Qualität der entstehenden Software, spart viel Zeit und verbessert auch noch die Lesbarkeit des Quellcodes, weil die BCL-Lösungen vielen Entwicklern vertraut sind.

Beim Einstieg in die Programmierung mit C# ist ...

- einerseits eine **Programmiersprache** mit bestimmter Syntax und Semantik zu erlernen
- und andererseits eine umfangreiche **Basisklassenbibliothek** zu studieren, die wesentlich an der Funktionalität eines Programms beteiligt ist.

2.6.1 Namensräume und namespace-Direktive

Damit nicht alle Klassen und sonstige Datentypen in einem gemeinsamen Namensraum koexistieren müssen, was unweigerlich Namenskonflikte zur Folge hätte, wurde das Konzept der **Namensräume** eingeführt. So dürfen z. B. zwei Klassen denselben Namen tragen, wenn sie sich in verschiedenen Namensräumen befinden. Ihre **voll qualifizierten Namen** (inkl. Namensraumbezeichnung) sind dann verschieden. Die Option der Namensräume wird speziell in der BCL intensiv dazu genutzt, die enorme Zahl von Klassen und sonstigen Datentypen nach funktionaler Verwandtschaft zu gruppieren.

Namensräume sind aber keinesfalls auf die BCL beschränkt, und die von C# - Entwicklungsumgebungen angebotenen Projektvorlagen (siehe z. B. Abschnitt 3.3) definieren meist per **namespace**-Direktive einen eigenen Namensraum für jedes neue Projekt, z. B.:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

namespace MyFirstApp {
    class Program {
        static void Main(string[] args) {
        }
    }
}
```

Bei kleinen Beispielprogrammen sind Namensräume überflüssig, und wir werden meist der Übersichtlichkeit halber darauf verzichten. Als Konsequenz dieser Praxis landen Klassen und andere Typen im **globalen Namensraum**, und die Gefahr von Namenskollisionen steigt.[1] Auch im Bruchrechnungs-Einstiegsbeispiel (siehe Abschnitt 1.2) wurde der Einfachheit halber auf die Definition eines Namensraums verzichtet.

Bei professionellen Projekten sollte man unbedingt Namensräume verwenden und dabei auch auf sinnvolle Namensraumbezeichnungen achten. Microsoft empfiehlt das folgende Schema:[2]

Company.(*Product* | *Technology*)[.*Feature*][.*Subnamespace*]

Zur kompakten Beschreibung der Syntax werden hier metasprachliche Regeln verwendet:

- Platzhalter sind an kursiver Schrift zu erkennen, statische Elemente an fetter Schrift. Im Beispiel treten nur Punkte als statische Elemente auf.
- Die runden und eckigen Klammern sowie der senkrechte Strich dienen als Metazeichen, gehören also *nicht* zur Namensraumbezeichnung:
  - Eckige Klammern begrenzen optionale Elemente.
  - Aus einer durch runde Klammern begrenzten Liste von Elementen, die durch senkrechte Striche voneinander getrennt sind, muss genau *ein* Element gewählt werden.

Beispiele:

- `Microsoft.Win32`
- `IBM.Data.DB2`

---

[1] Der globale Namensraum enthält neben den Typen, die zu keinem benannten Namensraum gehören, auch die benannten Namensräume.

[2] https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-namespaces

Namensraumbezeichnungen können hierarchisch organisiert werden, was speziell bei großen Bibliotheken für Ordnung und entsprechend lange vollqualifizierte Namen mit Punkten zwischen den Unterraumbezeichnungen sorgt. Die BCL verwendet (abweichend vom obigen Schema) **System** als Wurzelnamensraum und enthält z. B. den Namensraum

System.Threading

für Klassen und andere Datentypen zur Multithreading-Programmierung (mit parallelen Ausführungspfaden).

Im Zusammenhang mit dem Importieren eines Namensraums in eine Quellcodedatei per **using**-Direktive (siehe Abschnitt 2.6.3.1) ist zu beachten, dass die Typen in Unternamensräumen dabei *nicht* einbezogen sind. Durch Namensraumbezeichnungen mit gemeinsamen Startsegmenten werden funktionale Verwandtschaften signalisiert, aber keine Zugehörigkeiten im Sinne der Mengentheorie definiert.

Namensräume und Assemblies sind zwei voneinander *unabhängige* Organisationsstrukturen:

- Die Typen eines Namensraums können in verschiedenen Assemblies implementiert werden.
- In einem Assembly können Typen aus verschiedenen Namensräumen implementiert werden.

Z. B. befindet sich die Klasse **Uri** aus dem Namensraum **System**, die zur Modellierung von Netzwerk-Ressourcen dient, im Assembly **System.Runtime.dll**. Die zum selben Namensraum gehörende Klasse **Console**, die in unseren Übungsprogrammen häufig zur Ein-/Ausgabe per Konsole verwendet wird, steckt jedoch im Assembly **System.Console.dll**.

Eine **namespace**-Direktive hat traditionell einen durch geschweifte Klammern begrenzten Anwendungsbereich, und alle dortigen Typdefinitionen werden in den Namensraum einbezogen (siehe Beispiel oben). Meist sollen aber alle in der *Quellcodedatei* befindliche Typdefinitionen in den Namensraum einbezogen werden, und die per **namespace**-Direktive verursachte zusätzliche Klammernebene ist lästig. Seit C# 10 kann sie weggelassen werden, wenn die dateiglobale Gültigkeit der **namespace**-Direktive gemeint ist, z. B.:

```
using System;
 . . .
namespace MyFirstApp;
class Program {
    static void Main(string[] args) {
    }
}
```

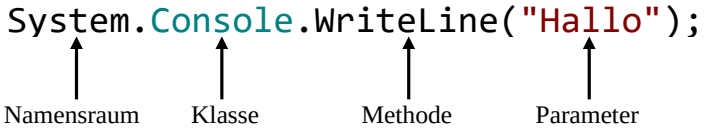
2.6.2 Leistungsumfang der BCL

Einen ersten Eindruck vom Leistungsvermögen der BCL 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 unbekannter Begriffe auslösen. Im Zweifelsfall kann die Tabelle mit reduzierter Verarbeitungstiefe überflogen werden:

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, Mengen 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	... ermöglicht es u. a., zur Laufzeit Informationen über Klassen abzufragen oder neue Methoden zu erzeugen. Dabei werden die Metadaten in den Assemblies genutzt.
System.Security	... enthält Klassen, die sich z. B. mit Verschlüsselungs-Techniken beschäftigen.
System.Threading	... unterstützt parallele Ausführungspfade.
System.Windows.Controls	... enthält Klassen für die Steuerelemente einer Windows-Anwendung mit einer grafischen Bedienoberfläche in WPF-Technik (z. B. Befehlsschalter, Textfelder, Menüs).
System.XML	... enthält Klassen für die Behandlung von XML-Dokumenten.

2.6.3 Explizite und implizite using-Direktive

Eine Klasse muss im Quellcode grundsätzlich mit ihrem voll qualifizierten Namen angesprochen werden (inkl. Namensraum), z. B.:



2.6.3.1 Explizite using-Direktive

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

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

Durch diese **using**-Direktive wird dafür gesorgt, dass die CLR jedem einfachen und im globalen Namensraum nicht auffindbaren Klassennamen das Präfix **System** voranstellt und dann die

referenzierten Assemblies (siehe Abschnitt 2.4.2.6) nach dem vervollständigten Namen durchsucht. Selbstverständlich können auch *mehrere* Namensräume importiert werden, was den Suchaufwand für die CLR erhöht, ohne spürbare Verzögerungen zu bewirken.

Bei Namenskollisionen gewinnt der lokalste Bezeichner. Im folgenden Beispiel werden der Namensraumbezeichner **System** und der Klassenname **Console** (im Namensraum **System**) durch lokale Variablennamen 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 den Bezeichner `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 den Bezeichner `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 Suche nach dem Bezeichner **System** anordnen, und die folgende Zeile führt trotz der Verdeckungen zum gewünschten Methodenaufruf:

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

Sie werden in eigenen Programmen das Verdecken von wichtigen Bezeichnern aus der BCL sicher unterlassen. Wenn komplexe Softwaresysteme unter Beteiligung vieler Programmierer entstehen, sind Namenskollisionen aber nicht auszuschließen. Der von Entwicklungsumgebungen automatisch erstellte Quellcode enthält daher häufig den **::** - Operator in Verbindung mit dem Schlüsselwort **global**, wie im Abschnitt 2.6.3.2 an einem Beispiel zu sehen ist.

Es ist zu beachten, dass durch eine **using**-Direktive für einen Namensraum die eventuell vorhandenen „untergeordneten“ Namensräume *nicht* einbezogen werden. Durch

```
using System;
```

wird also z. B. der Namensraum **System.Threading** *nicht* importiert.

2.6.3.2 Implizite using-Direktive

Seit C# 10 kann man in einer Quellcodedatei auf explizite **using**-Direktiven verzichten,

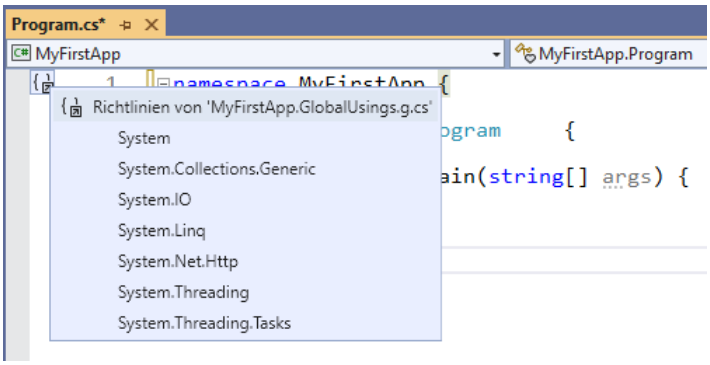
```
namespace MyFirstApp;
class Program {
    static void Main(string[] args) {
    }
}
```

wenn durch das **PropertyGroup**-Unterelement **ImplicitUsings** in der Projektdatei (siehe Abschnitt 3.1.2) die automatische Ergänzung einer voreingestellten Liste von **using**-Direktiven aktiviert ist,

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

und die Liste der impliziten **using**-Direktiven den eigenen Anforderungen genügt.

Die im Manuskript bevorzugte Entwicklungsumgebung **Visual Studio** (siehe Abschnitt 3.3) zeigt nach einem Mausklick auf das im Quellcodeeditor erscheinende Symbol die für einen Anwendungstyp eingestellte Liste von implizit importierten Namensräumen an, z. B.:



Außerdem erfährt man noch den Namen der automatisch angelegten Quellcodedatei mit den impliziten **using**-Direktiven (im Beispiel: **MyFirstApp.GlobalUsings.g.cs**):

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

Selbstverständlich sind explizite **using**-Direktiven weiterhin erlaubt, um für spezielle Quellcodedateien die Liste der impliziten Namensraumimporte zu ergänzen.

2.6.3.3 Globale using-Direktive

In den am Ende des Abschnitts 2.6.3.2 präsentierten automatisch erstellten impliziten **using**-Direktiven taucht das Schlüsselwort **global** zweimal auf, z. B.:

```
global using global::System;
```

Das *zweite* Auftreten des Schlüsselworts (zusammen mit dem **::** - Operator) sorgt dafür, dass die Suche nach dem Bezeichner **System** im globalen Namensraum beginnt, sodass **System** unabhängig von lokalen Definitionen als Namensraumbezeichner erkannt wird (siehe Abschnitt 2.6.3.1).

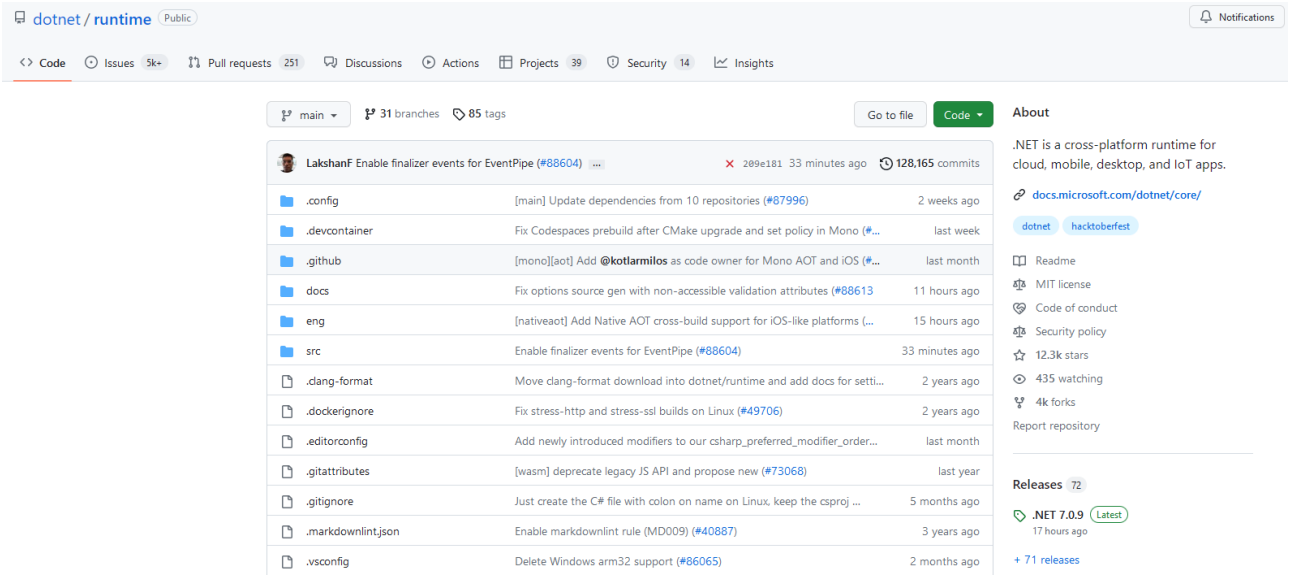
Seit C# 10 darf eine **using**-Direktive durch das *einleitende* Schlüsselwort **global** dekoriert werden, damit sie sich auf *alle* Quellcodedateien im Projekt auswirkt. Die im Abschnitt 2.6.3.2 behandelten impliziten **using**-Direktiven sind also global wirksam. Global wirksame explizite **using**-Direktiven sollten in einer herausgehobenen Quellcodedatei untergebracht werden.

2.6.4 BCL-Quellcode

Es ist oft von Interesse, den Quellcode von Typen (z. B. Klassen) aus der BCL zu inspizieren. So kann man technische Fragen klären oder auch die eigenen Kompetenzen erweitern. Der BCL-Quellcode ist als Open Source über die folgende Webseite frei verfügbar:

https://github.com/dotnet/runtime

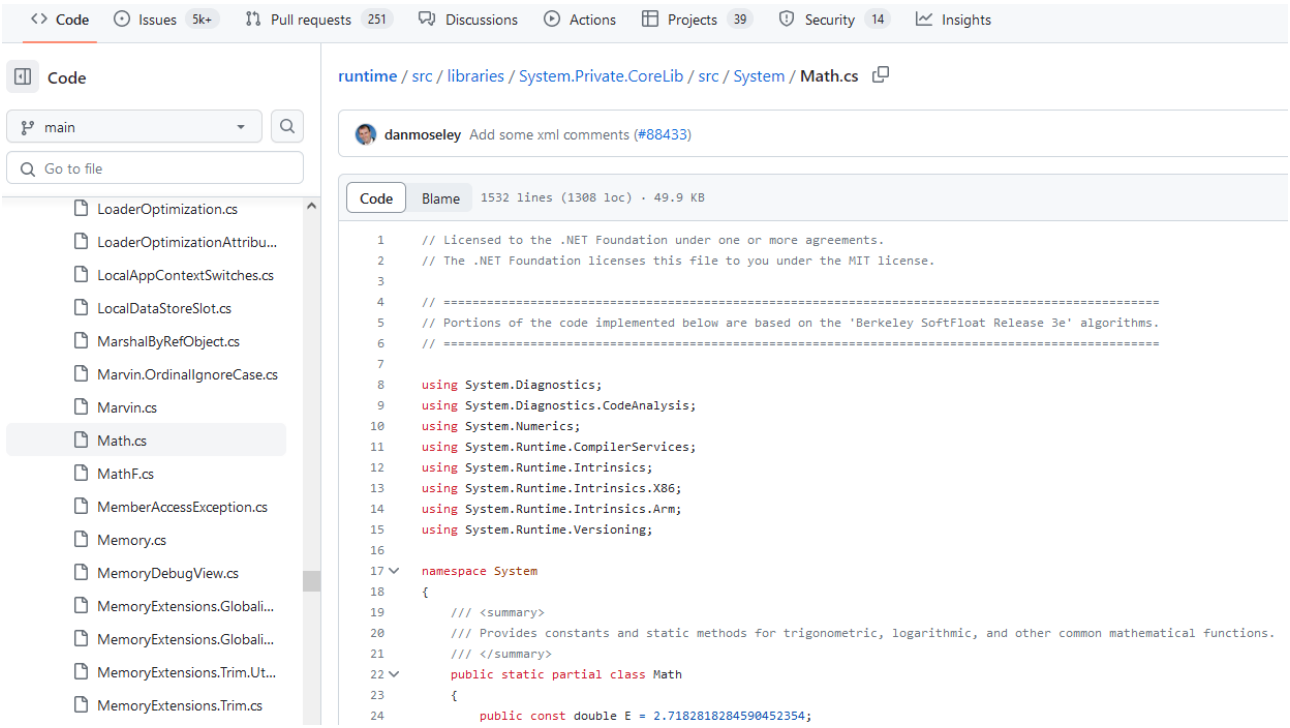
Es wird die aktuelle Version präsentiert (am 1.7.2023 z. B. die Version 7.0.9):



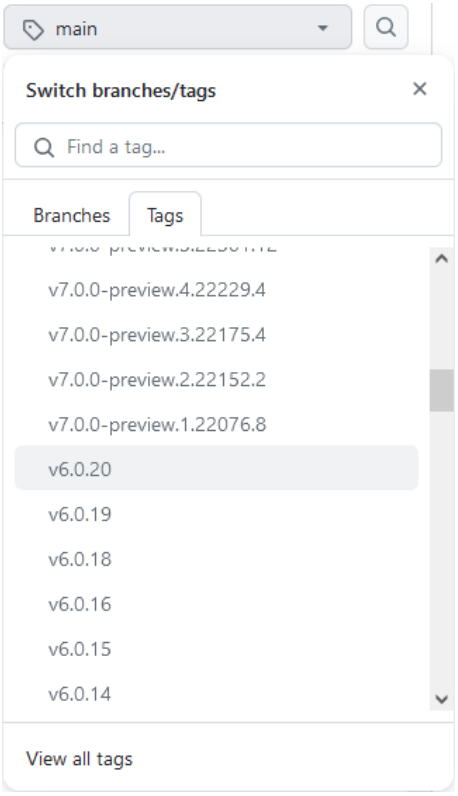
Von hier aus erreicht man z. B. den Quellcode der im Assembly **System.Private.CoreLib** implementierten Klasse **Math** auf dem folgenden Weg:

src > libraries >System.Private.CoreLib > src > System > Math.cs

Wie der Quellcode zeigt, befindet sich die Klasse im Namensraum **System**:



Um zu einer älteren .NET - Version zu wechseln, öffnet man das (auch auf der Startseite vorhandene) Drop-Down – Menü **main** und wählt die Registerkarte **Tags**, z. B.:

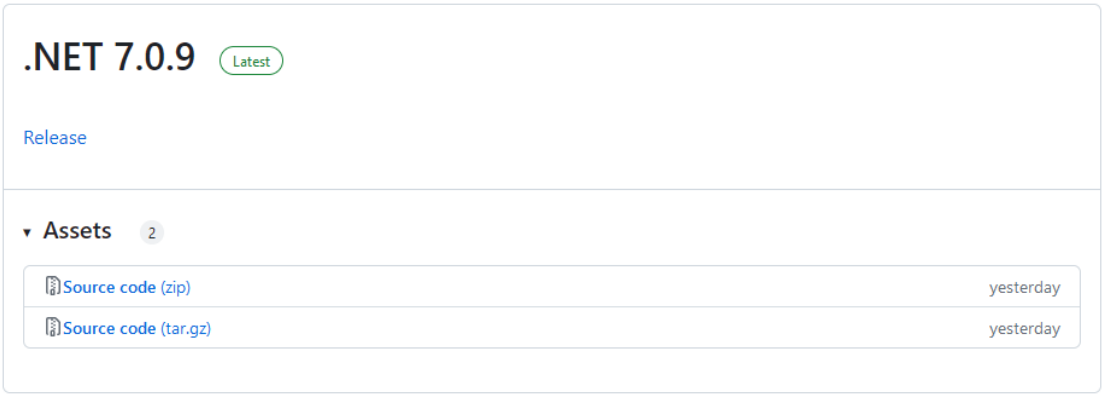


Wer den Quellcode einer .NET – Version herunterladen möchte, kann von der Startseite

https://github.com/dotnet/runtime

ausgehend über einen Klick auf **Releases**

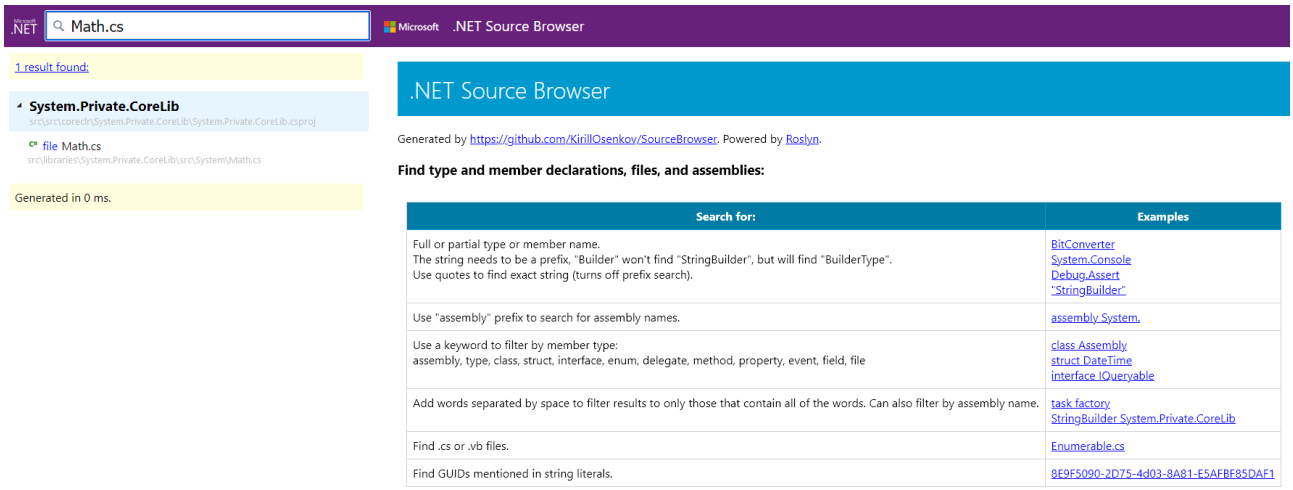
eine Liste mit den .NET – Versionen anfordern, eine Version wählen, nötigenfalls die **Assets** aufklappen und dann den **Source code** herunterladen, z. B.:



Die Webseite

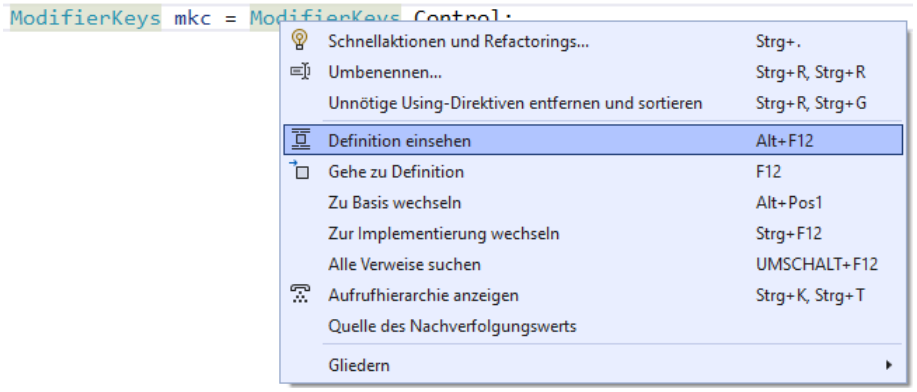
https://source.dot.net/

präsentiert den BCL-Quellcode mit besseren Suchoptionen, erlaubt aber keine Wahl der Version:



Das Visual Studio ermöglicht einen bequemen Zugang zum Quellcode des im Editor markierten (die Einfügemarke enthaltenden) BCL-Typs über …

- die Tastenkombination **Alt-F12**
- oder das Kontextmenüitem **Definition einsehen**, z. B.:



2.7 Zusammenfassung zum Kapitel 2

Als Vorteile der .NET - Technologie für die Software-Entwicklung 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 die CLS-Kompatibilität (*Common Language Specification*) geachtet wurde (vgl. Abschnitt 2.3.3).
- **Portabilität**
  .NET - Anwendungen sind nicht auf Windows beschränkt, sondern (allerdings ohne GUI) auch unter Linux und macOS einsetzbar. Seit .NET 6 sind plattformübergreifende Anwendungen mit einer grafischen Bedienoberfläche namens MAUI (*Multi-platform App UI*) möglich, die unter macOS, Windows, Android und iOS laufen.
- **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 Kapitel 2 u. a. die folgenden Begriffe kennengelernt:

- **Intermediate Language (IL)**
  .NET - Compiler (z. B. **csc.dll** bei C#) übersetzen den Quellcode in die Intermediate Language, die man als Maschinencode für eine virtuelle CPU auffassen und mit dem Java-Bytecode vergleichen kann (siehe Abschnitt 2.3).
- **Common Language Specification (CLS)**
  Beachtet man bei der Klassendefinition die Regeln der Common Language Specification (CLS), dann ist die Interoperabilität mit anderen CLS-kompatiblen Klassen unabhängig von der verwendeten Programmiersprache sichergestellt (siehe Abschnitt 2.3.3).
- **Assembly**
  Beim Übersetzen von (im Allgemeinen mehreren) Quellcodedateien entsteht ein Assembly, das folglich beliebig viele Klassen und sonstige Typen implementieren kann. Es enthält den IL-Code der übersetzten Typen sowie Metadaten zu den Typen und zum Assembly selbst. Seit .NET 5 wird für alle Assemblies die Dateinamenserweiterung **dll** verwendet. Die Besonderheit eines ausführbaren Assemblies im Vergleich zu einem Bibliotheks-Assembly besteht in der Anwesenheit einer Startklasse (mit einer statischen Methode namens **Main()**).
- **Metadaten**
  Ein Assembly enthält neben dem IL-Code auch Metadaten. Die **Typ-Metadaten** beschreiben …
  - die im Assembly implementierten Typen
  - und die vom Assembly referenzierten Typen aus anderen Assemblies.

  Im Manifest sind die **Assembly-Metadaten** mit Angaben zur Version, zur Kultur (Lokalisierung), zur Sicherheit und zur Abhängigkeit von anderen Assemblies enthalten.
- **Common Language Runtime (CLR) mit Just-In-Time (JIT) - Compiler**
  Die Ausführungsumgebung für IL-Code, der auch als *managed code* bezeichnet wird, besitzt einen Just-In-Time - Compiler zur Übersetzung des IL-Codes in den Maschinencode der lokalen CPU. Außerdem ist die CLR u. a. für die Freigabe des Speichers von überflüssig gewordenen Objekten und für das Multithreading zuständig.
- **Namensräume**
  Indem man eine Klasse in einen Namensraum einfügt, statt sie im globalen Namensraum zu belassen, ergänzt man ihren Namen um ein Präfix und vermeidet Namenskollisionen. Man fasst in der Regel funktional verwandte Klassen und sonstige Typen in einen gemeinsamen Namensraum zusammen.
- **Base Class Library (BCL)**
  Die BCL enthält in zahlreichen Assemblies generell benötigte Typen (z. B. **String**, **Console**). Für jede Anwendungskategorie (z. B. Windows-Desktop, ASP.NET Core, MAUI) befindet sich in der jeweiligen Variante der .NET – Laufzeitumgebung eine Erweiterung der BCL. Im Manuskript steht *BCL* für die Kombination aus der generellen und der anwendungsspezifischen Laufzeitbibliothek. Der Begriff *Bibliothek* wird im Manuskript etwas nachlässig …
  - entweder für ein *einzelnes* Assembly ohne Startklasse
  - oder für eine *Sammlung* von solchen Assemblies (z. B. die Basisklassenbibliothek)

  verwendet.

2.8 Übungsaufgaben zum Kapitel 2

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. In C# kann man nur Software für Windows entwickeln.
2. Die BCL wurde überwiegend in C# programmiert.
3. Die Klassen in einem mit C# erstellten Bibliotheks-Assembly können auch in anderen .NET - Programmiersprachen (z. B. VB.NET) genutzt werden.
4. Eine mit .NET erstellte WPF-Anwendung läuft wegen der Plattformunabhängigkeit unter Linux, macOS und Windows.
5. Für ein ausschließlich unter Windows einzusetzendes Programm ist das .NET Framework nach wie vor die passende .NET – Implementation.

2) In welcher Beziehung stehen Assemblies und Namensräume?

3) Was bedeuten die Abkürzungen IL, BCL und JIT?

3 Werkzeuge zum Entwickeln von C# - Programmen

In diesem Kapitel werden kostenlos verfügbare Werkzeuge zum Entwickeln von .NET - Applikationen in der Programmiersprache C# beschrieben. Zunächst beschränken wir uns auf einen simplen Texteditor zum Verfassen des Quellcodes und verwenden in einem Konsolenfenster das zusammen mit dem .NET SDK (*Software Development Toolkit*) installierte Programm **dotnet.exe** zum Erstellen des Assemblies, wobei im Hintergrund der Roslyn-Compiler den Quellcode in den IL-Code übersetzt. In dieser sehr übersichtlichen „Entwicklungsumgebung“ werden die grundsätzlichen Arbeitsschritte besonders deutlich. Anschließend werden mit dem Visual Studio Community 2022 und dem Visual Studio Code zwei Programme vorgestellt, die einen für die Software-Entwicklung optimierten Editor besitzen (z. B. mit der farblichen Unterscheidung von Syntaxbestandteilen und Vorschlägen zur Codevervollständigung). Zur Übersetzung des Quellcodes kommt wiederum der Roslyn-Compiler zum Einsatz.

3.1 .NET SDK

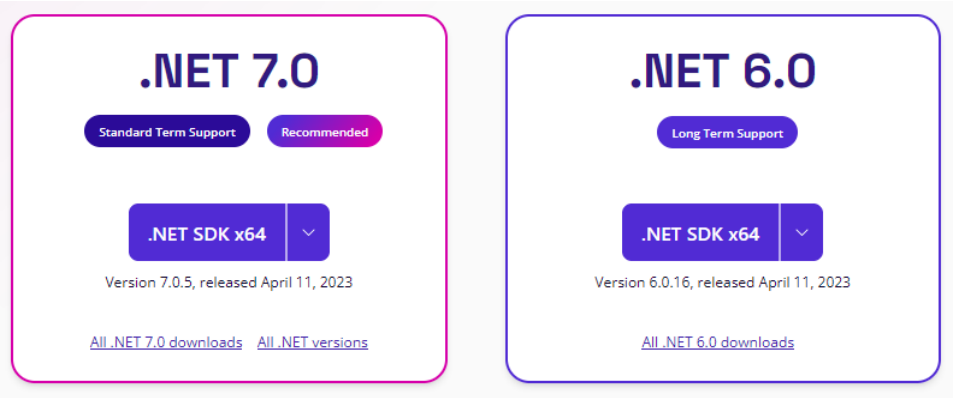
Die im frei verfügbaren .NET SDK (*Software Development Toolkit*) enthaltenen Werkzeuge ermöglichen die Entwicklung von .NET – Programmen, wenn auch nicht mit perfekter Produktivität. Es sind u. a. enthalten:

- das Erstellungssystem MSBuild
- der C# - Compiler Roslyn
- eine Laufzeitumgebung (CLR) zur Ausführung von Programmen
- das als *CLI* (*Command Line Interface*) bezeichnete Programm **dotnet.exe** mit zahlreichen Optionen zum Erstellen und Veröffentlichen von Programmen

3.1.1 Installieren

Bei der Installation von Visual Studio Community wird auch das aktuelle .NET SDK eingerichtet bzw. aktualisiert. Wer diese Entwicklungsumgebung verwendet, kann also auf eine vorherige SDK-Installation verzichten. Bei der im Abschnitt 3.3 beschriebenen Installation der Entwicklungsumgebung wird allerdings nur das SDK zu .NET 7 eingerichtet. Wir ergänzen daher an dieser Stelle das SDK zur .NET – Version 6, die dank *Long Term Support* (*LTS*) länger mit Updates versorgt wird als die Version 7 (mit *Standard Term Support*, *STS*).

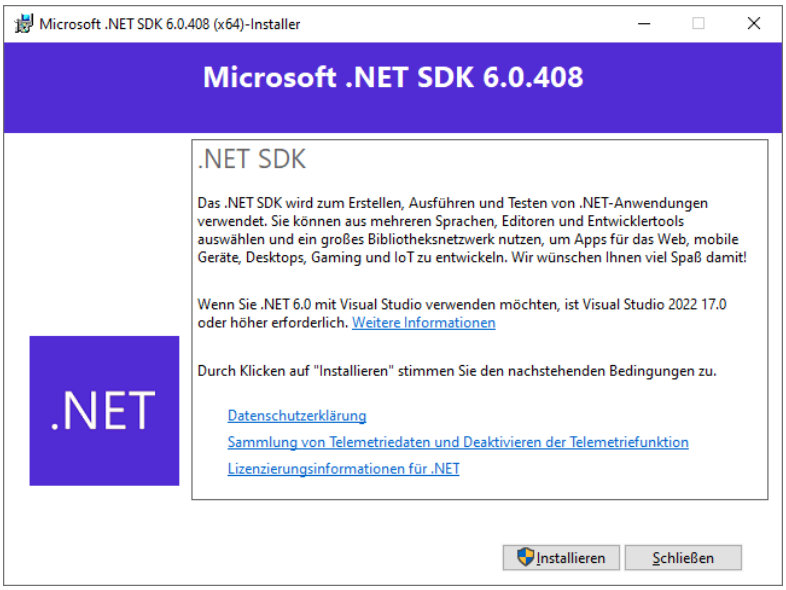
Man kann das .NET SDK samt Laufzeitumgebung von der Webseite https://dotnet.microsoft.com/ beziehen. Im Mai 2023 sind für Linux, macOS und Windows die LTS-Version 6.0 (Support bis zum 12.11.2024) sowie die STS-Version 7.0 (Support bis zum 14.5.2024) im Angebot:



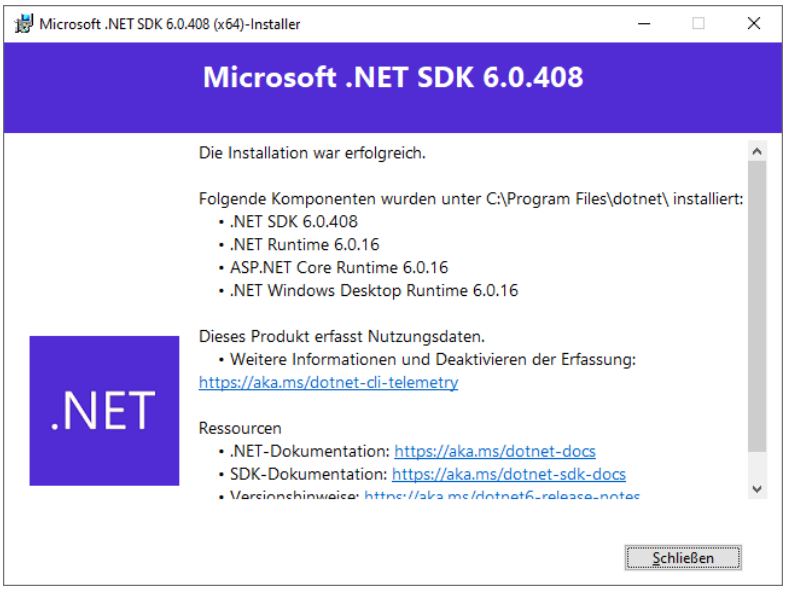
Wir installieren mit Hilfe der heruntergeladenen Datei

dotnet-sdk-6.0.408-win-x64.exe

das SDK zu .NET 6:



Neben dem SDK 6.0.408 werden auch drei anwendungsspezifische Laufzeitumgebungen in der Version 6.0.16 eingerichtet:



Ein .NET – SDK wird unter Windows in den folgenden Ordner installiert:

C:\program files\dotnet\sdk

Hier landet auch das zum Erstellen, Ausführen und Veröffentlichen von .NET-Anwendungen geeignete Kommandozeilenprogramm **dotnet.exe**, das in der englischen Literatur als *CLI* (*command line interface*, dt.: *Kommandozeilen-Schnittstelle*) zu .NET bezeichnet wird. Das Programm wird bei der Installation in den Suchpfad für ausführbare Programme aufgenommen. Es spielt auch bei der C# - Software-Entwicklung mit dem Visual Studio *Code* eine wichtige Rolle (siehe Abschnitt 3.5.4) und wird daher im Manuskript oft erwähnt, wobei *dotnet-CLI* als kompakte und eindeutige Bezeichnung dient.

Mit dem folgenden **dotnet**-Kommando lässt sich prüfen, welche .NET – SDKs auf einem Rechner vorhanden sind:

```
C:\Users\baltes>dotnet --list-sdks
3.1.426 [C:\Program Files\dotnet\sdk]
5.0.104 [C:\Program Files\dotnet\sdk]
5.0.203 [C:\Program Files\dotnet\sdk]
5.0.214 [C:\Program Files\dotnet\sdk]
5.0.303 [C:\Program Files\dotnet\sdk]
5.0.408 [C:\Program Files\dotnet\sdk]
5.0.416 [C:\Program Files\dotnet\sdk]
6.0.408 [C:\Program Files\dotnet\sdk]
7.0.304 [C:\Program Files\dotnet\sdk]
```

Jede SDK-Version befindet sich in einem eigenen Ordner, z. B.:

C:\program files\dotnet\sdk\6.0.408

In der SDK-Version geben die beiden ersten, durch einen Punkt getrennten Ziffern die unterstützte .NET – Haupt - bzw. – Nebenversion an, im Beispiel: 6.0. Dann folgt nach einem weiteren Punkt die sogenannte *Featuregruppe*, die mit 1 startet und beim vierteljährlichen Erscheinen einer neuen Visual Studio - Nebenversion ansteigt, im Beispiel: 4. Am Ende steht (ohne trennenden Punkt) die beim monatlichen Wartungs-Update heraufgesetzte Patch-Version, im Beispiel 08.[1]

Mit dem folgenden Kommando erfahren wir, welche .NET – Laufzeitumgebungen vorhanden sind:

```
C:\Users\baltes>dotnet --list-runtimes
Microsoft.AspNetCore.App 3.1.32 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 5.0.4 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 5.0.5 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 5.0.9 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 5.0.17 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 6.0.18 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 7.0.7 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.NETCore.App 3.1.32 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 5.0.4 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 5.0.5 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 5.0.9 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 5.0.17 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 6.0.18 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 7.0.7 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.WindowsDesktop.App 3.1.32 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 5.0.4 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 5.0.5 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 5.0.9 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 5.0.17 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 6.0.18 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 7.0.7 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
```

Auf dem Beispielrechner befinden sich Laufzeitumgebungen für drei verschiedene **Anwendungstypen**:

- .NET - Runtime
  Diese Laufzeitumgebung genügt für Konsolenanwendungen und ist in den Laufzeitumgebungen für die beiden anderen Anwendungstypen enthalten.
- ASP.NET Core - Runtime
  Diese Laufzeitumgebung führt ASP.NET Core – Anwendungen (Web-Anwendungen bzw. -Dienste) aus.
- Desktop-Runtime
  Diese Laufzeitumgebung ist nur unter Windows vorhanden und führt Anwendungen mit grafischer Bedienoberfläche aus, wobei die GUI-Bibliotheken WinForms und WPF unterstützt werden.

Die Versionierung der Laufzeitumgebungen verwendet das Schema

Hauptversion.Nebenversion.Patch

[1] https://learn.microsoft.com/en-us/dotnet/core/porting/versioning-sdk-msbuild-vs

Jede Laufzeitumgebung befindet sich in einem eigenen Ordner, z. B.:

C:\program files\dotnet\shared\Microsoft.WindowsDesktop.App\6.0.16

Seit .NET 6 lässt sich mit dem folgenden Kommando die Aktualität der installierten SDK- und Laufzeitversionen prüfen:

```
C:\Users\baltes>dotnet sdk check
.NET SDKs:
Version      Status
-----------------------------------------------
3.1.426      .NET 3.1 wird nicht mehr unterstützt.
5.0.104      .NET 5.0 wird nicht mehr unterstützt.
 . . .
5.0.416      .NET 5.0 wird nicht mehr unterstützt.
6.0.410      Patch 6.0.411 ist verfügbar.
7.0.304      Patch 7.0.305 ist verfügbar.

Testen Sie die neuesten .NET SDK-Features mit .NET 8.0.100-preview.5.23303.2.

.NET-Runtimes:
Name                              Version      Status
---------------------------------------------------------------------------------
Microsoft.AspNetCore.App          3.1.32       .NET 3.1 wird nicht mehr unterstützt.
 . . .
Microsoft.WindowsDesktop.App      5.0.17       .NET 5.0 wird nicht mehr unterstützt.
Microsoft.AspNetCore.App          6.0.18       Patch 6.0.19 ist verfügbar.
Microsoft.NETCore.App             6.0.18       Patch 6.0.19 ist verfügbar.
Microsoft.WindowsDesktop.App      6.0.18       Patch 6.0.19 ist verfügbar.
Microsoft.AspNetCore.App          7.0.7        Patch 7.0.8 ist verfügbar.
Microsoft.NETCore.App             7.0.7        Patch 7.0.8 ist verfügbar.
Microsoft.WindowsDesktop.App      7.0.7        Patch 7.0.8 ist verfügbar.
```

Wie sich im weiteren Verlauf von Abschnitt 3.1 zeigen wird, dient das dotnet-CLI vielen Zwecken (z. B. Projekt anlegen, Assembly erstellen und ausführen). Über die komplette Funktionalität informiert z. B. die folgende Webseite:

https://learn.microsoft.com/en-us/dotnet/core/tools/

Zum Erstellen von Software für mobile Geräte (Smartphones und Tablets) sowie zur Erstellung von Multiplattform-Anwendungen, die unter Android, iOS, macOS und Windows laufen, wird .NET MAUI benötigt (*Multi-platform App UI*):

https://dotnet.microsoft.com/en-us/apps/maui

Eine SDK-Installation muss zur Unterstützung von .NET MAUI erweitert werden.[1]

3.1.2 Projektordner anlegen

Wie angekündigt, verwenden wir im Abschnitt 3.1 zum Erstellen eines C# - Programms das zum .NET SDK gehörende dotnet-CLI. Neben der C# - Quellcodedatei ist dabei auch eine Projektdatei beteiligt, wobei das vom dotnet-CLI automatisch erstellte Exemplar für unsere Zwecke genügt.

Sind auf einem Rechner *mehrere* .NET SDK – Versionen vorhanden, dann wird vom (versions-unabhängigen!) dotnet-CLI per Voreinstellung die neueste verwendet. Das ist z. B. dann *nicht* erwünscht, wenn eine Vorschauversion (z. B. .NET 8.0.100-preview.5.23303.2) installiert worden ist, die aber für kommerzielle Projekte nicht verwendet werden soll. Über eine Datei namens **global.json** im aktuellen Ordner oder in einem übergeordneten Ordner lässt sich die zu verwendende SDK-Version vorschreiben, z. B.:

---

[1] Siehe z. B. https://learn.microsoft.com/en-us/dotnet/maui/ios/cli

```
{
  "sdk": {
    "version": "6.0.408"
  }
}
```

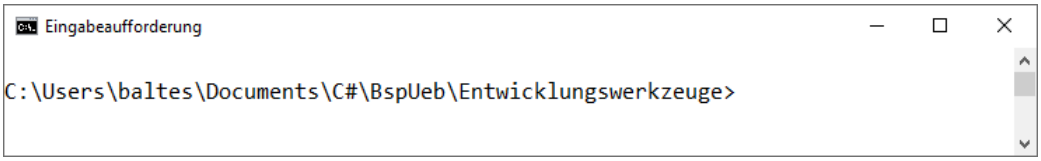
Man kann diese Datei mit dem Kommando **dotnet new globaljason** erstellen lassen, z. B.:

```
>dotnet new globaljson --sdk-version 6.0.408
```

Welche Version vom dotnet-CLI (eventuell aufgrund einer Steuerung per **global.json**) verwendet wird, ist so zu erfahren:

```
>dotnet --version
```

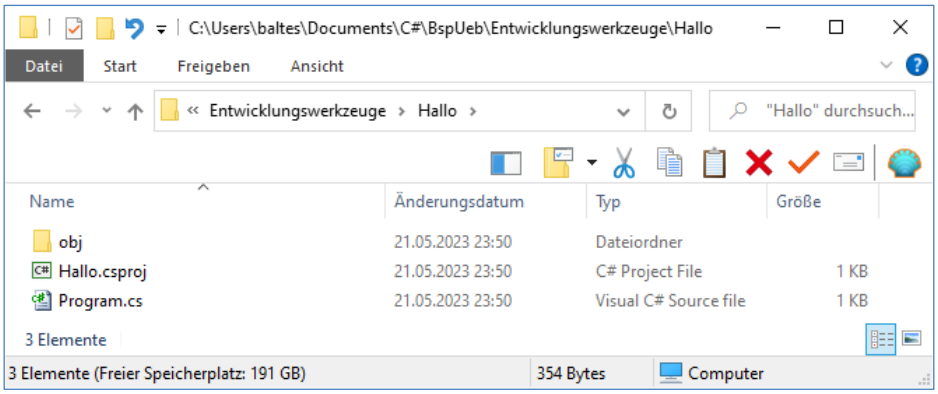
Wir positionieren ein Konsolenfenster auf ein Verzeichnis, in dem der Projektordner angelegt werden soll, z. B.:



Mit dem Kommando

```
>dotnet new console -n Hallo
```

lassen wir unter Verwendung der Vorlage **console** ein Projekt mit dem Namen `Hallo` anlegen. Es soll ein Programm entstehen, das auf der Konsole eine freundliche Meldung ausgibt. Der neu angelegte Projektordner



enthält:

- die Projektdatei **Hallo.csproj**
  Die Projektdatei (mit der Namenserweiterung **csproj**) steuert das Verhalten der **MSBuild**-Technik zum Erstellen von Anwendungen (vgl. Abschnitt 2.3.1). Daher ist die **csproj**-Datei bei allen Verfahren zum Entwickeln von .NET – Software (dotnet-CLI, Visual Studio, VS Code) involviert, und im Manuskript taucht die Projektdatei oft auf.
- die Quellcodedatei **Program.cs**
  Man darf den Namen und den Inhalt der Datei verändern. Beim Erstellen entsteht aus allen Quellcodedateien im Projektordner ein gemeinsames Assembly.
- der Unterordner **obj** mit Hilfsdateien für die Erstellung des Programms
  Für die Dateien in diesem Ordner werden wir uns eher selten interessieren.

Im Beispiel ist die automatisch erstellte Projektdatei angenehm kurz,

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

und von den vier Einstellungen im Element **PropertyGroup** können wir zwei bereits komplett verstehen:

- Als **OutputType** ist **Exe** eingestellt (= Konsolenanwendung), weil wir im Kommando **dotnet new** die Vorlage **console** verwendet haben. Beim Erstellen des Projekts wird also eine Konsolenanwendung entstehen. Alternative Werte sind:
  - **Library**
    Es entsteht ein Bibliotheks-Assembly, das Klassen und andere Typen enthält, aber keine Startklasse.
  - **Winexe**
    Es entsteht ein ausführbares Windows-Programm. Im Unterschied zum Typ **Exe** wird bei der Ausführung kein Konsolenfenster angezeigt, was im `Hallo`-Beispiel zu einem sinnlosen Programm ohne jeglichen Bildschirmauftritt führen würde.
- Als **TargetFramework** ist **net6.0** eingestellt, weil bei der Projekterstellung eine Datei namens **global.json** anwesend war und die SDK-Version 6.0.408 vorgeschrieben hat. Diese SDK-Version hat wiederum **net6.0** als das vom resultierenden Programm vorausgesetzte Zielframework gewählt.

3.1.3 Quellcode editieren

Zum Editieren einer Quellcodedatei kann z. B. das im Windows-Zubehör enthaltene Programm **Notepad** (alias **Editor**) verwendet werden. Die vom Kommando

```
>dotnet new console
```

erstellte Datei **Program.cs** enthält einen Kommentar und eine Anweisung, die einen Text auf die Konsole schreibt:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Dass aus der Vorlage **console** ein Programm mit einer simplen Konsolenausgabe entstanden ist, verwundert nicht, das Fehlen einer Klassendefinition erstaunt aber schon. Nach den Erfahrungen aus dem Abschnitt 1.5 war für ein `Hallo`-Programm ungefähr der folgende Quellcode zu erwarten (mit einem etwas kreativeren Meldungstext):

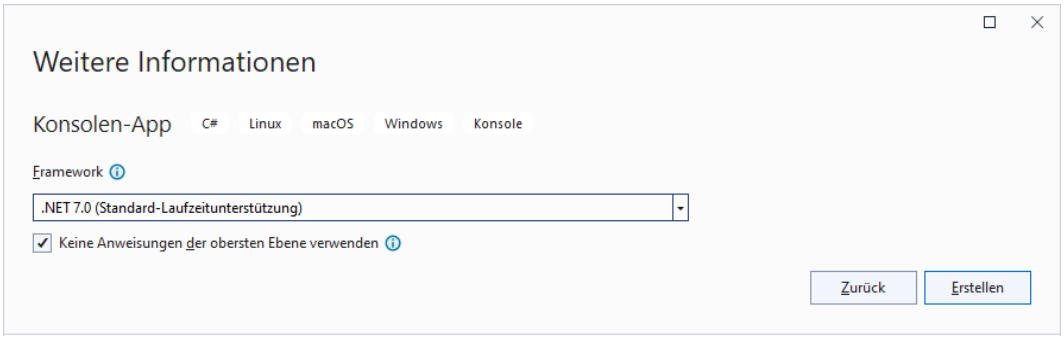
```
using System;

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

Seit C# 9 darf man sich tatsächlich bei einer Konsolenanwendung auf die Anweisung(en) in der **Main()** – Methode beschränken,

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

und der Compiler ergänzt den Rest. Dabei sind Vereinfachungen im Spiel, die grundsätzlich zu begrüßen sind, aber für Einsteiger die Orientierung erschweren, weil die Programmstruktur teilweise im Verborgenen bleibt. Es wird nur scheinbar die Regel verletzt, dass in C# keine Anweisungen außerhalb einer Klassendefinition erlaubt sind. Tatsächlich erstellt der Compiler im Hintergrund eine Klasse samt **Main()** – Methode. Die dabei entstehenden Zeilen sind bei allen Programmen identisch und damit eine lästige Pflichtübung. In der englischen Literatur spricht man in dieser Situation von *boilerplate code*. Sobald wir die Struktur eines C# - Programms internalisiert und die vom Compiler erbrachten Ergänzungen verstanden haben, werden wir die seit C# 9 erlaubte Bequemlichkeit nutzen.

Die mit C# für Konsolenanwendungen eingeführte Kurzform der Startklasse ist gemeint, wenn Microsoft von *Anweisungen der obersten Ebene* (engl.: *top level statements*) spricht, z. B. im Dialog zum Erstellen einer Konsolenanwendung im Visual Studio (siehe Abschnitt 3.3.5):



Im Einleitungsbeispiel (siehe Kapitel 1) wurde einiger Aufwand in Kauf genommen, um einen halbwegs realistischen Eindruck von der objektorientierten Programmierung (OOP) zu vermitteln. Das `Hallo`-Beispiel (in der kompletten Form, inkl. boilerplate code) ist zwar angenehm einfach aufgebaut, kann aber als „pseudo-objektorientiert“ (POO) kritisiert werden. Es ist eine einzige Klasse namens `Hallo` mit einer 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 3 und 4 werden wir solche pseudo-objektorientierten (POO-) Programme benutzen, um elementare Sprachelemente in einer möglichst einfachen Umgebung kennenzulernen.

Aus den letzten Ausführungen ergibt sich, dass C# zwar eine objektorientierte Programmierweise nahelegen und unterstützen, aber nicht erzwingen kann. Seit C# 9 darf von einem POO-Programm der aus *boilerplate code* bestehende Anschein der Objektorientierung weggelassen werden.

Das (komplette) `Hallo`-Programm eignet sich aufgrund seiner Kürze zum Erläutern wichtiger Regeln, an die Sie sich so langsam gewöhnen sollten. Alle in der folgenden Auflistung enthaltenen Themen werden aber später noch einmal ausführlich behandelt:

- In der ersten Zeile wird per **using**-Direktive der Namensraum **System** importiert, damit die dort enthaltene Klasse **Console** im Programm ohne Namensraumpräfix angesprochen werden kann (vgl. Abschnitt 2.6). 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 weitgehend 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 (bestehend aus dem Schlüsselwort **class** und dem Namen der Klasse) 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 automatisch aufgerufen und dient bei „echten" OOP-Programmen oft dazu, Objekte (direkt oder indirekt) 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 1.2). Die beim Programmstart automatisch ausgeführte **Main()** - Methode der Startklasse muss auf jeden Fall durch den Modifikator **static** als Klassenmethode gekennzeichnet sein. In einem objektorientierten Programm hat sie oft die Aufgabe, die ersten Objekte zu erzeugen (siehe unsere Klasse `Bruchaddition` auf Seite 12).
    - **void**
      Im Beispiel wird für die Methode **Main()** der Rückgabetyp **void** verwendet, weil die Methode ihrem Aufrufer keinen Rückgabewert liefert. Mit den Rückgabewerten von Methoden werden wir uns noch gründlich beschäftigen.
- Per **Parameterliste** kann man Daten an eine Methode übergeben, um z. B. ihre Arbeitsweise zu beeinflussen. Hinter dem Methodennamen muss auf jeden Fall eine durch runde Klammern eingerahmte Parameterliste angegeben werden, gegebenenfalls eine leere.
- Dem Kopf einer Methodendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf mit Variablendeklarationen und sonstigen Anweisungen.

---

[1] Diese längliche Fußnote sollte nur lesen, wer im `Hallo`-Beispielprogramm (z. B. aufgrund von Erfahrungen mit anderen C# - Beschreibungen) den Modifikator **public** am Anfang der Klassen- und der Methodendefinition vermisst.

Die Methode **Main()** wird beim Programmstart von der Laufzeitumgebung aufgerufen. Weil es sich bei der Laufzeitumgebung aus Sicht des Programms um einen *externen* Akteur handelt, liegt es nahe, die Methode **Main()** explizit über den Modifikator **public** für die Öffentlichkeit freizugeben. 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 (z. B. bei Gunnerson 2002, Louis et al. 2002, Mössenböck 2019) viele `Hallo`-Beispielprogramme 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 die statische Methode **Main()** grundsätzlich nur von der Laufzeitumgebung aufgerufen, und die ist *nicht* mit einer fremden Klasse gleichzusetzen. Laut C# - Sprachdefinition (ECMA 2022) 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. Richter 2012) wird 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 die einzige Klasse der `Hallo`-Beispielprogramme als Startklasse nur von der Laufzeitumgebung gesucht und genutzt wird.

- In der **Main()** - Methode unserer `Hallo`-Klasse wird die statische **WriteLine()** - Methode der Klasse **Console** dazu benutzt, um einen Text auf der Konsole auszugeben. 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 statischen Methode **Console.WriteLine()** einen Parameter, damit der gewünschte Effekt erzielt wird. Wir geben eine durch doppelte Anführungszeichen begrenzte Zeichenfolge an, die auf der Konsole erscheinen soll.
- Bei einem Methodenaufruf handelt es sich um eine **Anweisung**, und die ist in C# mit einem Semikolon abzuschließen.

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.

Wir entscheiden uns für den Quellcode mit vollständiger Klassendefinition und speichern ihn in einer Datei mit dem Namen **Hallo.cs**, z. B.

C:\Users\baltes\Documents\C#\BspUeb\Entwicklungswerkzeuge\Hallo\Hallo.cs

Im Unterschied zu anderen objektorientierten Programmiersprachen (z. B. Java) müssen in C# der Klassen- und der Dateiname *nicht* übereinstimmen, zwecks Übersichtlichkeit sollten sie es in der Regel aber doch tun.

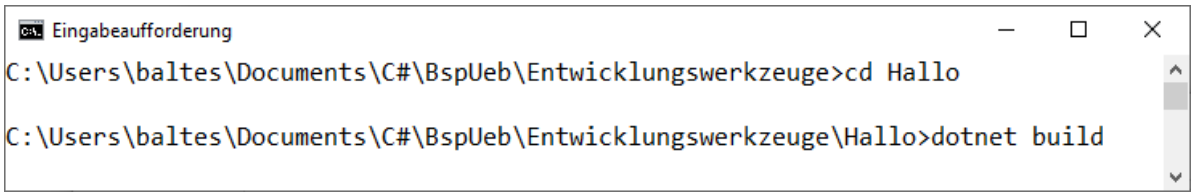
Wir löschen die automatisch angelegte Datei **Program.cs**, weil ansonsten das resultierende Assembly eine nutzlose Klasse namens `Program` enthalten würde.

3.1.4 Quellcode in die IL übersetzen und Programm erstellen

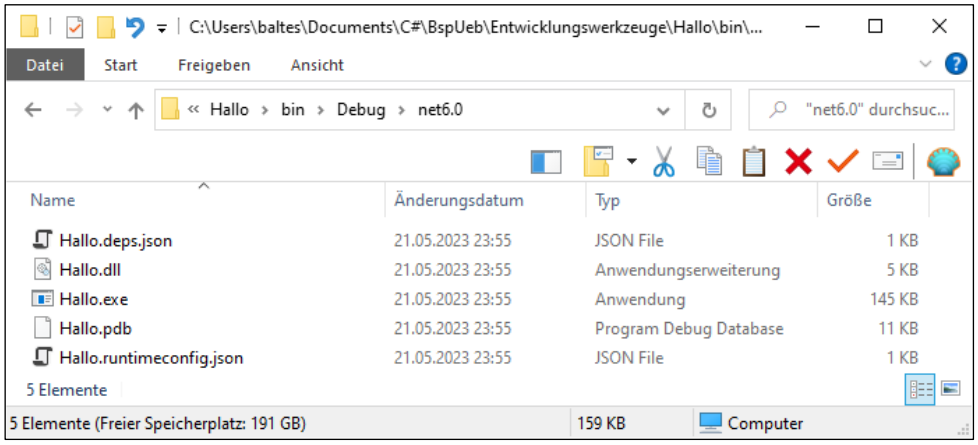
Wir wechseln mit dem Konsolenfenster zum Projektordner (mit der Projektdatei) und fordern mit dem Kommando

```
>dotnet build
```

die Erstellung des Programms an:



Es resultiert der Unterordner **…\ bin\Debug\net6.0** mit dem ausführbaren Assembly bzw. Programm:



Von den beiden Dateien mit dem Namen **Hallo** beinhaltet die kleinere (mit der Namenserweiterung **dll**) das Assembly mit dem IL-Code der Klasse `Hallo`:[1]

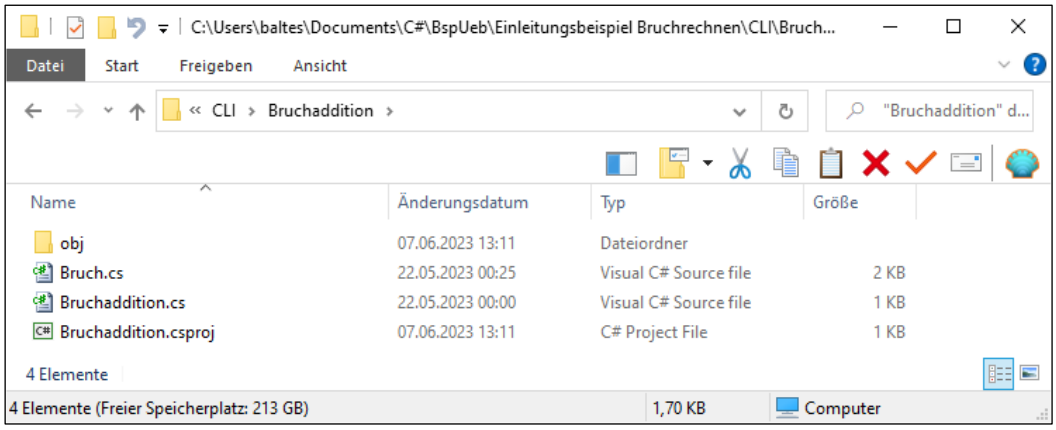
```
.class private auto ansi beforefieldinit Hallo
   extends [System.Runtime]System.Object
{
   // Methods
   .method private hidebysig static
      void Main () cil managed
   {
      // Method begins at RVA 0x2050
      // Header size: 1
      // Code size: 13 (0xd)
      .maxstack 8
      .entrypoint

      IL_0000: nop
      IL_0001: ldstr "Hallo, echt .NET hier!"
      IL_0006: call void [System.Console]System.Console::WriteLine(string)
      IL_000b: nop
      IL_000c: ret
   } // end of method Hallo::Main

   .method public hidebysig specialname rtspecialname
      instance void .ctor () cil managed
   {
      // Method begins at RVA 0x205e
      // Header size: 1
      // Code size: 8 (0x8)
      .maxstack 8

      IL_0000: ldarg.0
      IL_0001: call instance void [System.Runtime]System.Object::.ctor()
      IL_0006: nop
      IL_0007: ret
   } // end of method Hallo::.ctor

} // end of class Hallo
```

Die erheblich größere Datei **Hallo.exe** ist nur unter Windows vorhanden und ermöglicht einen bequemen Programmstart (Abschicken des Dateinamens in einem Konsolenfenster oder Doppelklick auf den Dateinamen).

Im Hintergrund wurde vom Erstellungssystem MSBuild der C# - Compiler aus dem Roslyn-Projekt zur Übersetzung des Quellcodes in den IL-Code verwendet.

Befinden sich (wie im Beispiel aus dem Abschnitt 2.3) *mehrere* Quellcodedateien im aktuellen Verzeichnis,

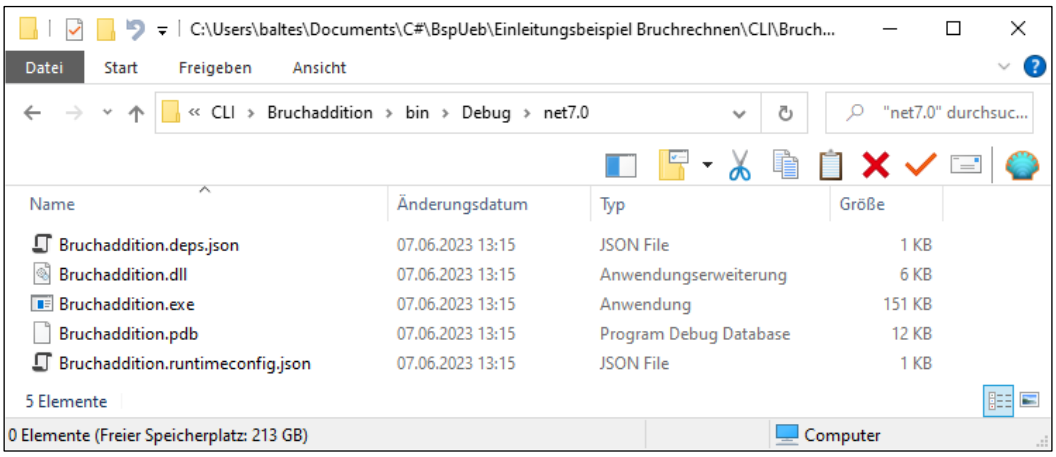
[1] Wie man das im Manuskript bereits mehrfach verwendete Programm **ILSpy** dazu bringt, den IL-Code in der Datei **Hallo.dll** anzuzeigen, wird im Abschnitt 3.4 erläutert.



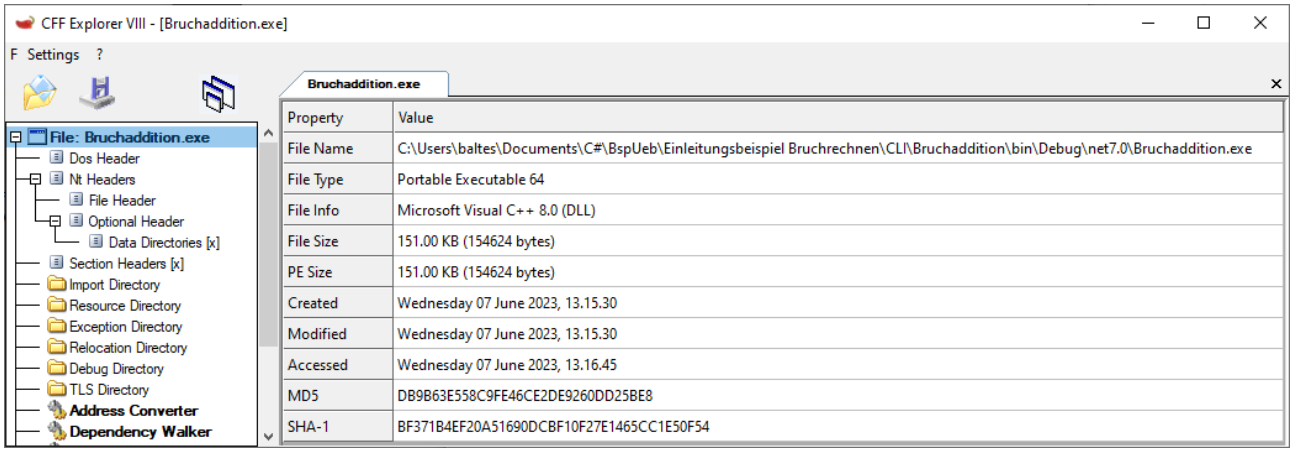
dann werden durch das Kommando

```
>dotnet build
```

alle Quellcodedateien in ein gemeinsames Assembly übersetzt, das seinen Namen von der Startklasse übernimmt, z. B.:



Auf einem Rechner mit 64-Bit - Windows als Betriebssystem resultiert zum bequemen Starten des Assemblies eine 64-Bit - Anwendung, wie der kostenlose **CFF-Explorer** von Daniel Pistelli zeigt:[1]



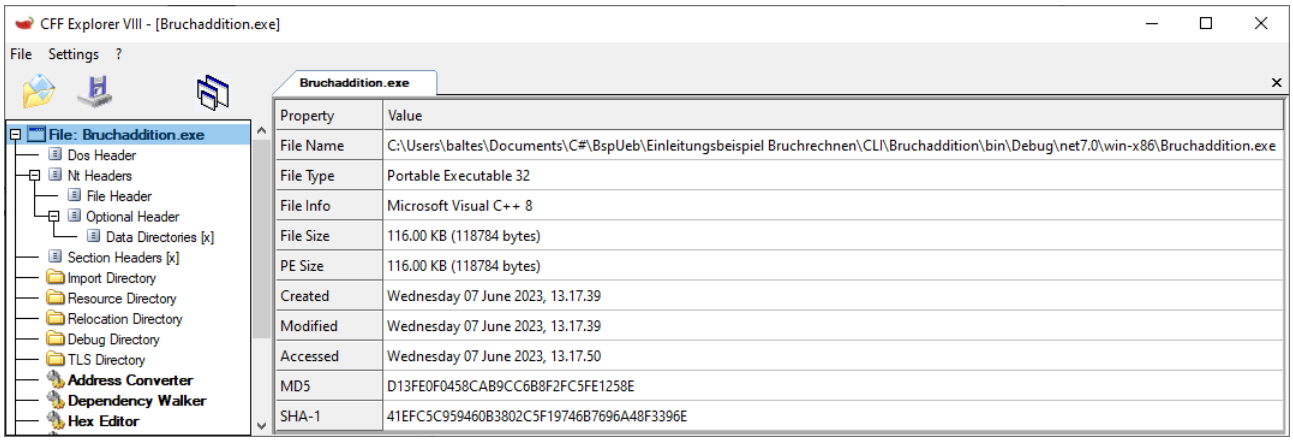
Die (mittlerweile sehr seltenen) Rechner mit 32-Bit – Windows können dieses Startprogramm also nicht ausführen, das plattformübergreifende Assembly aber sehr wohl (siehe Abschnitt 3.1.5 zum Assembly-Start per dotnet-CLI). Wird bei der Programmerstellung die Architektur **x86** angefordert,

---

[1] Das betagte, aber immer noch nützliche Programm ist hier verfügbar:
https://ntcore.com/?page_id=388

```
>dotnet build --arch x86
```

dann resultiert ein 32-Bit – Startprogramm, und das arbeitet auf *jedem* Windows-Rechner:



Über weitere Erstellungsoptionen informieren das folgende Kommando

```
>dotnet build -?
```

und die Webseite

https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-build

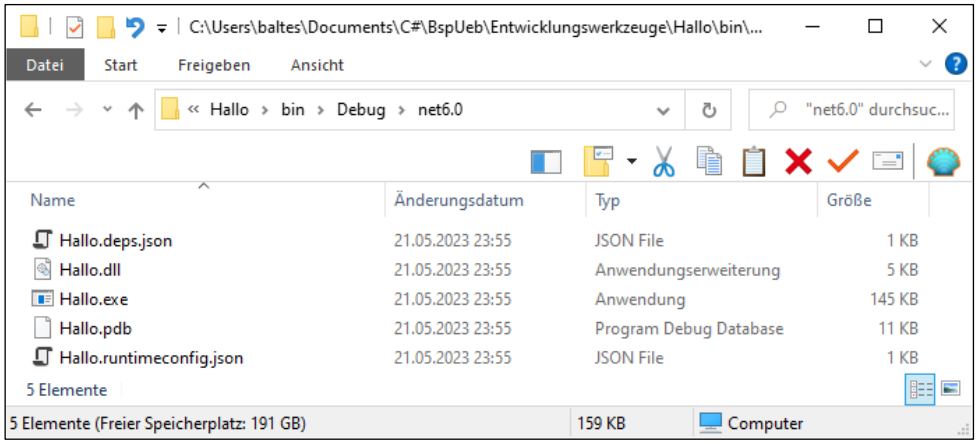
Wir werden eine Entwicklungsumgebung zum Programmieren verwenden und auch die Erstellungsoptionen in diesem Rahmen festlegen (siehe Abschnitt 3.3.8).

3.1.5 Ausführen

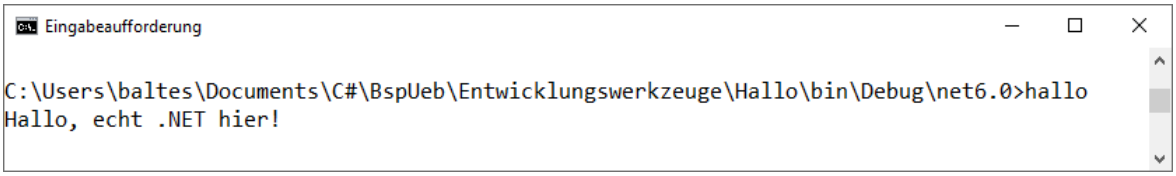
Bei der Projekterstellung mit dem Ausgabetyp **Exe** entstehen unter Windows (vgl. Abschnitt 3.1.4):

- ein Assembly in einer Datei mit der Namenserweiterung **dll**
- ein Windows-Programm in einer Datei mit der Namenserweiterung **exe**, das ein bequemes Starten des Assemblies ermöglicht (durch Abschicken des Dateinamens in einem Konsolenfenster oder per Doppelklick auf den Dateinamen).

Hier sind die Erstellungsergebnisse zum **Hallo**-Projekt noch einmal zu sehen:

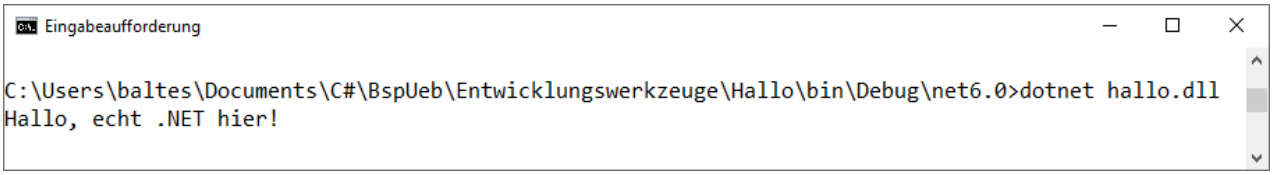


Ist ein Konsolenfenster auf den Ordner mit den Erstellungsergebnissen positioniert, dann kann das Programm über die Datei **Hallo.exe** auf windows-übliche Weise gestartet werden:



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

Per dotnet-CLI lässt sich das Assembly in der Datei **Hallo.dll** auch *ohne* die Datei **Hallo.exe** starten:



Nicht verzichten kann man hingegen auf die zusammen mit dem Assembly erstellte Datei **Hallo.runtimeconfig.json**, die darüber informiert, welche Laufzeitumgebung hinsichtlich Typ und Version zur Ausführung des Assemblies erforderlich ist:

```
{
  "runtimeOptions": {
    "tfm": "net6.0",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "6.0.0"
    }
  }
}
```

Fehlt diese Datei bei einer Konsolenanwendung, also bei einem Projekt mit der folgenden Einstellung

```
<OutputType>Exe</OutputType>
```

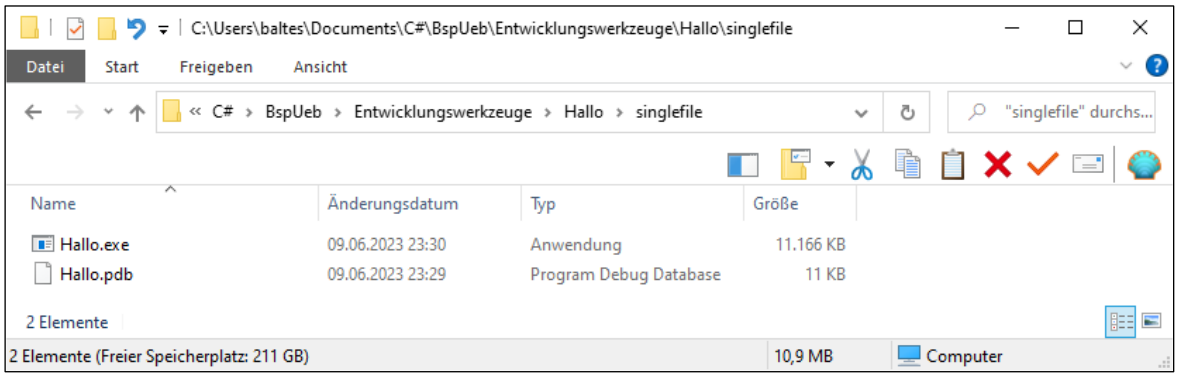
in der **csproj**-Datei, dann scheitert der Programmstart mit der Fehlermeldung:

```
A fatal error was encountered.
```

Obwohl wir uns aktuell noch nicht mit Details der Software-*Veröffentlichung* beschäftigen sollten, wird ein Kommando zum Erstellen eines eigenständigen Programms (engl.: *self-contained app*) angegeben, das auf jedem Rechner mit 64-Bit – Windows läuft:

```
>dotnet publish -r win-x64 -o singlefile -p:PublishSingleFile=true -p:PublishTrimmed=true
```

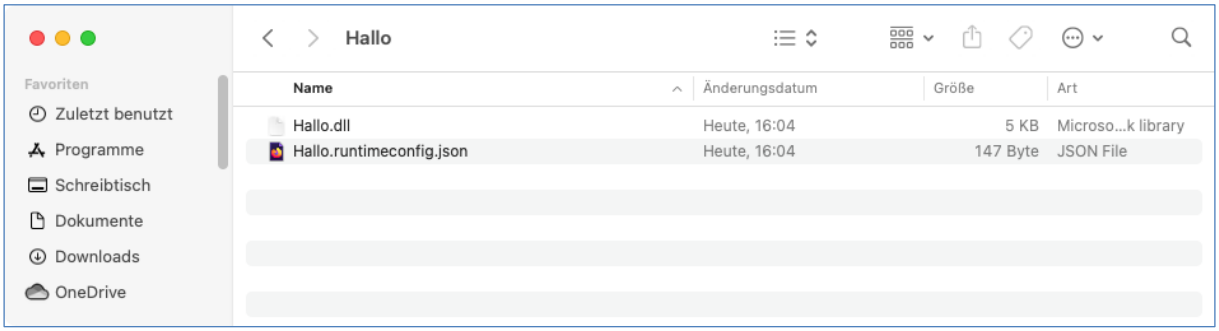
Die resultierende eigenständige Datei beinhaltet eine angepasste .NET - Laufzeitumgebung und ist daher mit ca. 11 MB erheblich größer als das ursprünglich erstellte, als *framework-abhängig* (engl.: *framework-dependent*) bezeichnete und nur 5 KB große Assembly:[1]



[1] Mit dem .NET Framework (vgl. Abschnitt 2.2.1) hat der Begriff *framework-abhängig* nichts zu tun.

Über die Option **PublishTrimmed** wird dafür gesorgt, dass die Größe der eigenständigen Datei durch das Löschen von nicht genutztem Code von 62 MB auf 11 MB schrumpft.[1]

Neben dem kompakten Format hat das Assembly den Vorteil, auch unter macOS und Linux zu laufen. Installiert man z. B. auf einem Mac eine passende .NET – Version (zu beziehen von der Webseite https://dotnet.microsoft.com/) und überträgt die beiden Dateien **Hallo.dll** und **Hallo.runtimeconfig.json** dorthin,



dann lässt sich das Assembly ausführen:

Diese erneute Abweichung vom Pfad zur baldigen Beschäftigung mit der Programmiersprache C# ist vertretbar, weil die Erfahrung der Portabilität von .NET - Software für eine gesteigerte Lernmotivation sorgen sollte.

3.1.6 Programmfehler

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 bemerkt und sind daher schon vor dem Starten eines Programms leicht zu beseitigen.
- **Logikfehler (Semantikfehler)**
  Hier liegt *kein* Syntaxfehler vor, aber das Programm verhält sich anders als erwartet, wiederholt z. B. ständig eine nutzlose Aktion („Endlosschleife“) oder stürzt mit einem Laufzeitfehler ab. In jedem Fall sind die Benutzer verärgert, wenn sie auf einen solchen Fehler stoßen.

Die C# - Designer haben (z. B. durch die Verwendung von streng definierten und unveränderlichen Datentypen) dafür gesorgt, dass möglichst viele Fehler vom Compiler aufgedeckt werden können, also zur Kategorie der Syntaxfehler gehören.[2]

---

[1] Ein KB enthält 1000 Bytes, und ein MB enthält 1000 KB. Manchmal wird ein KB mit 1024 (= 2^{10}) Bytes und ein MB mit 1024 KB gleichgesetzt.

[2] Seit C# 4.0 ist mit dem Datentyp **dynamic** eine praktischen Zwängen (z. B. bei der Kooperation mit typfreien Skriptsprachen) geschuldete Ausnahme von der strengen Typisierung vorhanden. Anstelle des Compilers ist hier die CLR für die Typprüfung verantwortlich, was die Wahrscheinlichkeit von Laufzeitfehlern erhöht. Dieser Datentyp sollte nur in begründeten Ausnahmefällen verwendet werden; im Kurs kommt er nicht zum Einsatz.

Wir wollen am Beispiel eines provozierten Syntaxfehlers überprüfen, ob der C# - Compiler hilfreiche Fehlermeldungen produziert. Wenn in der **Main()** -Methode der Klasse `Hallo` der Bezeichner **Console** fälschlicherweise mit kleinem Anfangsbuchstaben geschrieben wird,

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

dann meldet der Compiler:

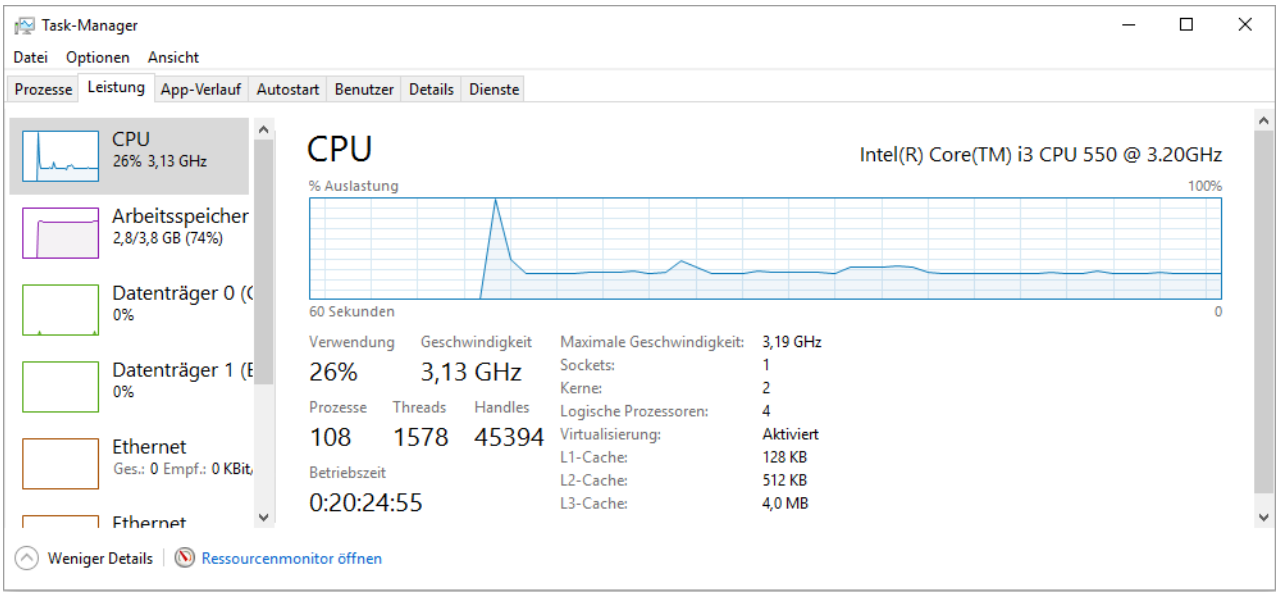
```
Hallo.cs(4,3): error CS0103: Der Name 'console' ist im aktuellen Kontext nicht
         vorhanden.
```

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 ist gut zu verstehen. Wer Erfahrungen mit Programmiersprachen wie Visual Basic oder Delphi hat, muss sich noch daran gewöhnen, dass in C# (wie in den meisten modernen Programmiersprachen) die Groß-/Kleinschreibung signifikant ist.

Während Syntaxfehler nur den Programmierer betreffen, automatisch entdeckt und leicht beseitigt werden können, verursachen Logikfehler für Entwickler *und* Anwender oft einen sehr großen Schaden. Simons (2004, S. 43) schätzt, dass viele Logikfehler tausendfach mehr Aufwand verursachen als der übelste Syntaxfehler. Während sich in die äußerst simple Klasse `Hallo` kaum ein Logikfehler einbauen lässt, ist das im Bruchadditionsbeispiel aus dem Kapitel 1 leicht zu bewerkstelligen. Wird z. B. in der `Nenner` - Eigenschafts-Implementierung bei der Absicherung gegen Nullwerte der Ungleich-Operator (!=) durch sein Gegenteil (==) ersetzt, dann ist keine Syntaxregel verletzt:

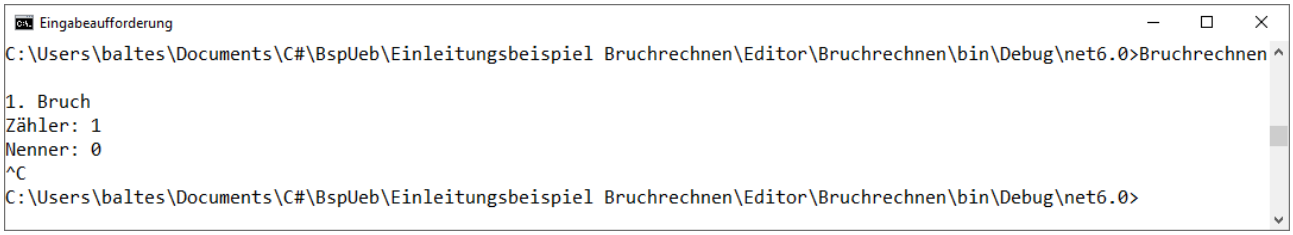
```
public int Nenner {
    get {
        return nenner;
    }
    set {
        if (value == 0)    // Logikfehler
            nenner = value;
    }
}
```

Bei Eingabe kritischer „Brüche“ (wie z. B. $\frac{1}{0}$) verursacht die fehlerhafte Definition der Klasse `Bruch` aber ein unerwünschtes Verhalten des Programms `Bruchaddition`: Die Methode `Kuerze()` (vgl. Abschnitt 1.3) gerät in eine Endlosschleife, und das Programm verbraucht dabei reichlich Rechenzeit, wie der Windows-Taskmanager auf einem Rechner mit der Intel-CPU Core i3 (mit 4 logischen Kernen) zeigt:



Das sinnlos agierende Programm lastet einen logischen CPU-Kern komplett aus, vergeudet also ca. 25% der CPU-Zeit.

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



3.2 Wahl einer Entwicklungsumgebung

Auf die Dauer ist das Hantieren mit einem Texteditor und einem Konsolenfenster beim Entwickeln von C# - Software keine ernsthafte Option. Bei der Suche nach einer leistungsfähigen, gut bedienbaren und kostenlosen Entwicklungsumgebung kommen zwei Kandidaten in die engere Wahl, die *beide* von der Firma Microsoft stammen:

- Visual Studio Community 2022
  Dies ist die kostenlose, aber für unsere Zwecke voll ausreichende Edition von Microsofts Entwicklungsumgebung für den professionellen Einsatz. Es sind alle wichtigen Funktionen vorhanden, um C# - Programme bequem erstellen, testen und verteilen zu können. Das Visual Studio wird primär für Windows entwickelt, doch existiert auch eine Version für macOS, die auf dem Xamarin Studio für macOS basiert.
- Visual Studio Code (meist kurz als *VS Code* bezeichnet)
  Dieses Open Source – Programm ist eigentlich ein auf Quellcode spezialisierter Texteditor mit einer guten Plugin-Schnittstelle. Viele Funktionen einer vollausgebauten Entwicklungsumgebung (z. B. die Code-Analyse) fehlen zunächst, können aber dank einer umfangreichen Bibliothek mit hochwertigen Plugins nachgerüstet werden. Das Programm benötigt im Vergleich zum Visual Studio weniger Speicherplatz und Rechenzeit. Passend zur plattformübergreifenden .NET - Technologie ist VS Code für Windows, macOS und Linux verfügbar.

Beide Programme unterstützen die Programmierung durch einen Editor mit einer syntaxorientierten Textdarstellung und einer kontextsensitiven Auflistung von Optionen zur Syntaxvervollständigung. Während die traditionelle, von Microsoft als *IntelliSense* bezeichnete Syntaxvervollständigung eine

alphabetische Liste der zulässigen Fortsetzungen anbietet, beherrschen die beiden zu vergleichenden Entwicklungsumgebungen eine als *IntelliCode* bezeichnete, mit KI-Techniken arbeitende Verbesserung, die von Microsoft folgendermaßen beschrieben wird:[1]

> IntelliCode platziert das, was Sie am wahrscheinlichsten verwenden, oben in der IntelliSense-Vervollständigungsliste. Diese zeitsparenden Empfehlungen basieren auf der Analyse tausender Open-Source-Beiträge auf GitHub.

IntelliCode macht im Visual Studio auch Vorschläge zu einer Fortsetzung des Programms, wenn man bei der Eingabe kurz wartet, z. B.:

```
int a = 4711;
int b =
```

Man kann die an grauer Schrift zu erkennenden Vorschläge per Tabulator-Taste übernehmen oder ignorieren.

Die Entscheidung zwischen den beiden Entwicklungsumgebungen fällt nicht leicht und wird im Internet intensiv diskutiert. Eine vollständige Ermittlung und faire Integration der zahlreichen Argumente ist anspruchsvoll und möglicherweise ein geeignetes Betätigungsfeld für die künstliche Intelligenz in der am 30.11.2022 von der Firma OpenAI veröffentlichten KI-Lösung ChatGPT (*Generative Pretrained Transformer*). Am 2.5.2023 haben wir 10 spontan formulierte Fragen zum Vergleich der konkurrierenden Programme gestellt:

1 Welche Entwicklungsumgebung eignet sich am besten für C#?
2 Wer bietet die beste IntelliSense-Funktion für C#: Visual Studio Community oder VS Code?
3 Wird das Debugging bei C# - Projekten im Visual Studio Community oder im VS Code besser unterstützt?
4 Wer bietet die beste Refaktorierung für C#: Visual Studio Community oder VS Code?
5 Werden Datenbankprojekte im Visual Studio Community oder in VS Code besser unterstützt?
6 Wird das Unit Testing bei C# - Projekten im Visual Studio Community oder in VS Code besser unterstützt?
7 Werden Razor-Projekte im Visual Studio Community oder in VS Code besser unterstützt?
8 Werden Blazor-Projekte im Visual Studio Community oder in VS Code besser unterstützt?
9 Wird das Profiling von C# - Projekten im Visual Studio Community oder im VS Code besser unterstützt?
10 Wer bietet die beste Unterstützung beim Deployment von C# - Programmen: Visual Studio Community oder VS Code?

Bei allen Fragen hat ChatGPT das Visual Studio favorisiert, sodass man als Ergebnisse dieser kritisierbaren Methode zur Bewertungsintegration die folgenden Argumente für das Visual Studio notieren kann:[2]

---

[1] https://visualstudio.microsoft.com/de/services/intellicode/

[2] Bei den aktuellen Large Language Models ist im Zweifelsfall eine Fehlerquote von ca. 20% anzunehmen, wobei die exakte Quote schwer zu ermitteln ist (Gieselmann 2023, S. 17). Allzu ernst sollte man also die Empfehlungen von ChatGPT nicht nehmen.

- Bessere IntelliSense-Funktion (Vorschläge zur Code-Vervollständigung)
- Bessere Refaktorierung (z. B. beim Umbenennen von Klassen oder Methoden)
- Bessere Unterstützung für Datenbankprojekte
- Bessere Unterstützung für Web-Projekte (Razor, Blazor)
- Bessere Werkzeuge zur Unterstützung der Software-Entwicklung (Unit Testing, Debugging, Profiling)
- Bessere Werkzeuge für die Veröffentlichung (Auslieferung) von Software

Weiterhin sprechen für das Visual Studio die folgenden Fakten:

- Unterstützung von WPF- und WinForms-Projekten durch einen grafischen Designer für die Bedienoberfläche
- Unterstützung für MAUI-Projekte
- Verfügbarkeit eines visuellen Klassen-Designers (siehe Anwendungsbeispiel im Abschnitt 1.2)

Als Vorteile von VS Code sind zu nennen:

- Verfügbarkeit für Linux, macOS und Windows
- Geringer Massenspeicherbedarf
- Schnelle Installation
- Kurze Startzeit und flotter Betrieb
- Unterstützung vieler Programmiersprachen

Von diesen Argumenten sind allerdings für uns einige nicht zwingend, weil …

- wir außer C# keine andere Programmiersprache verwenden,
- und weil das Visual Studio Community selbst auf einem 12 Jahre alten Windows-Rechner mit 8 GB RAM und SSD bei den Manuskriptbeispielen keine Ressourcen- oder Zeitprobleme verursacht.

Die Plattformunabhängigkeit ist allerdings ein gravierender Vorteil von VS Code, auf den wir aufgrund der zahlreichen Pluspunkte von Visual Studio Community schweren Herzens verzichten in der Annahme, dass die meisten Leser einen Windows-Rechner verwenden. Wir erläutern im Manuskript auch die Verwendung von VS Code (siehe Abschnitt 3.5), widmen aber insgesamt dem Visual Studio Community mehr Aufmerksamkeit.

3.3 Microsoft Visual Studio Community 2022 für Windows

Im Mai 2023 ist Visual Studio Community 2022 in der Version 17.6.0 verfügbar geworden (inkl. Unterstützung für .NET 7.0). Wenn Sie diesen Text lesen, ist mit Sicherheit eine höhere Version aktuell. Erfahrungsgemäß bleiben aber die im Manuskript enthaltenen Erläuterungen zur Entwicklungsumgebung größtenteils gültig.

Neben der von uns verwendeten kostenlosen Community-Edition existieren noch zwei *kostenpflichtige* Editionen:

- Visual Studio Professional
- Visual Studio Enterprise

Wie die folgende Webseite

https://visualstudio.microsoft.com/de/vs/compare/

zeigt, unterscheiden sich die Editionen Community und Professional im Wesentlichen durch das Lizenzmodell, während die Enterprise-Edition zusätzliche Leistungen enthält.

Die Lizenzbedingungen für die Community-Edition sind liberal. So ist es z. B. einem *einzelnen* Entwickler (im Unterschied zu einer Software-Firma) erlaubt, mit dem Visual Studio Community kostenlose und kostenpflichtige Anwendungen zu erstellen.[1]

3.3.1 Systemvorraussetzungen

Microsoft nennt die folgenden (empfohlenen) Systemvoraussetzungen für Visual Studio 2022:[2]

- Windows 10 oder 11 sowie die korrespondierenden Versionen von Windows Server
- Prozessor mit mindestens vier Kernen empfohlen[3]
- 4 GB RAM (16 GB empfohlen)
- Für unsere Zwecke genügen ca. 6 GB Festspeicher (auf der SSD oder Festplatte)

3.3.2 Bezugsquelle

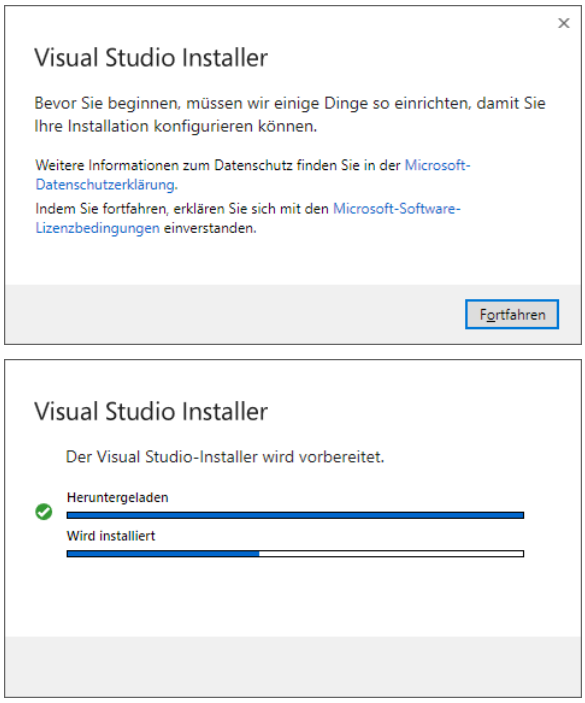
Von der Webseite

https://visualstudio.microsoft.com/downloads/

kann man einen Web-Installer herunterladen, der die benötigten Dateien während der Installation aus dem Internet bezieht.

3.3.3 Installation

Der Web-Installer **VisualStudioSetup.exe** führt nach dem Start einige Vorbereitungen aus, die wenige Minuten in Anspruch nehmen:



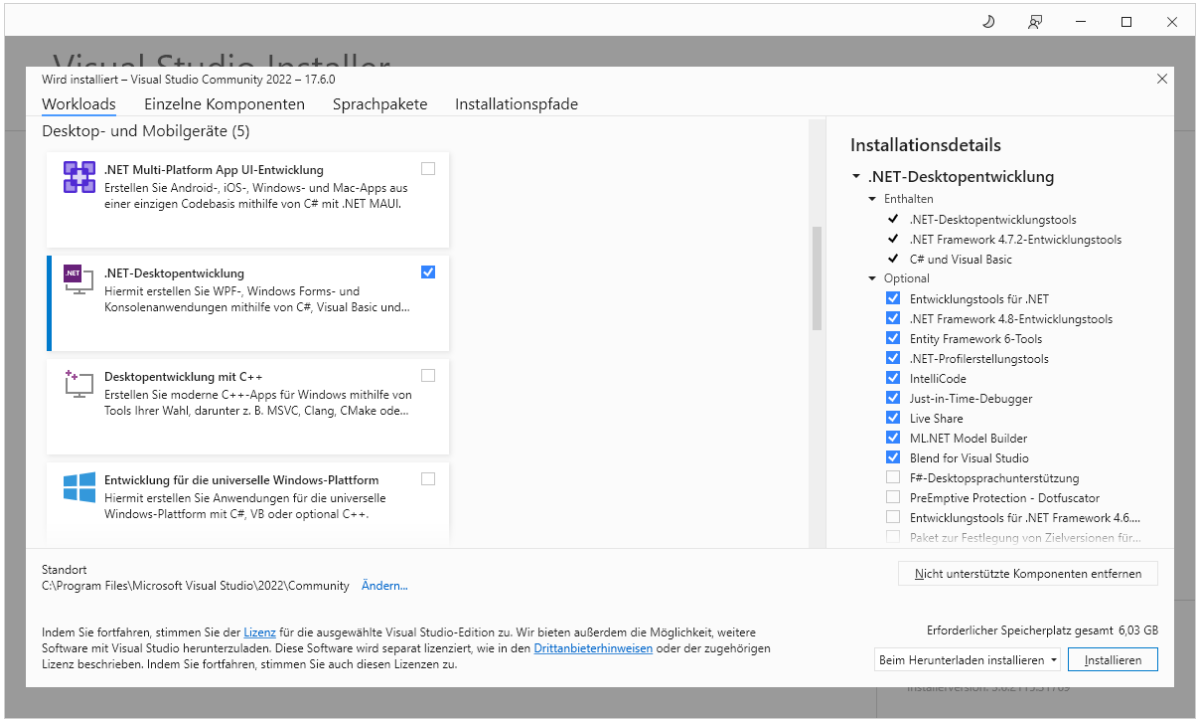
Anschließend erscheint der **Visual Studio Installer** mit dem folgenden Dialog zur Wahl der **Workloads** (geplanten Einsatzfelder):

---

1 https://visualstudio.microsoft.com/de/vs/community/

2 https://learn.microsoft.com/en-us/visualstudio/releases/2022/system-requirements

3 Alle Manuskriptbeispiele sind auf einem Rechner mit zwei realen Kernen plus Hyperthreading (also insgesamt vier virtuellen Kernen) entstanden.



Wir beschränken uns auf die **.NET - Desktopentwicklung** und *verzichten* vorläufig auf:

- **ASP.NET und Webentwicklung**
- **Azure-Entwicklung**

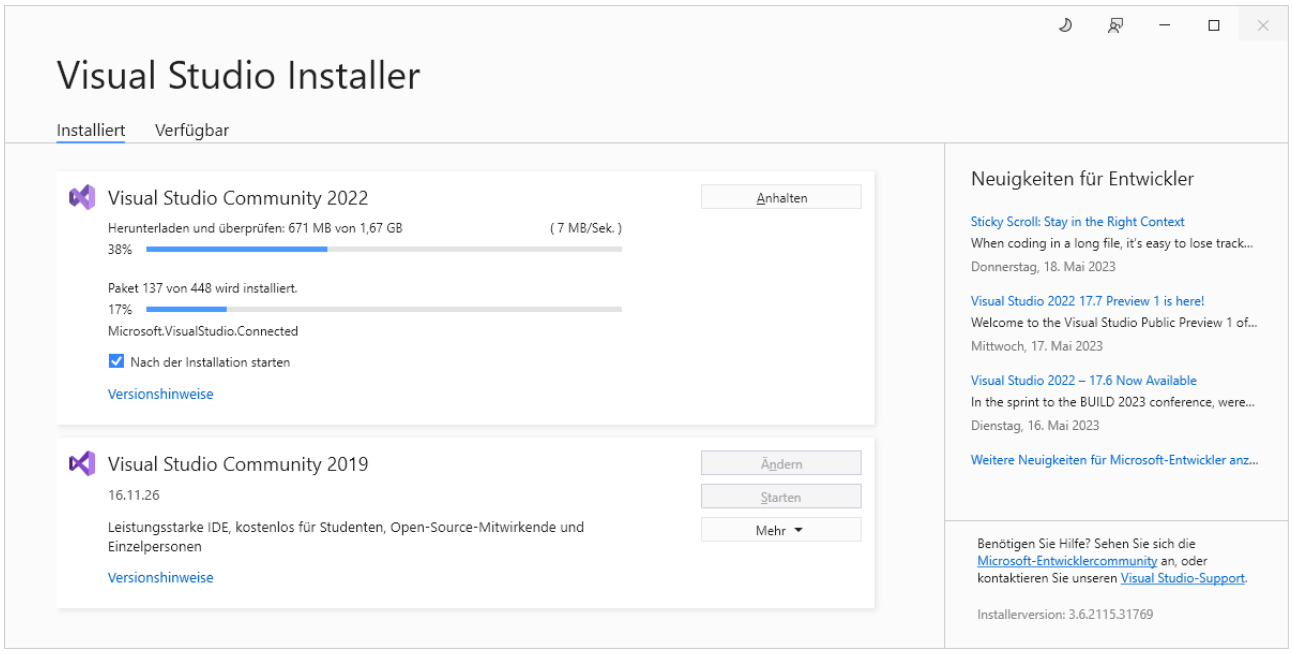
Die damit erforderlichen 6,03 GB Festspeichervolumen sind akzeptabel, sodass wir die **optionalen** Bestandteile und die **einzelnen Komponenten** nicht auf weitere Einsparpotenziale hin untersuchen.

Bei Bedarf kann der **Visual Studio Installer** später (über einen Link im Windows-Startmenü) erneut gestartet werden, um den Installationsumfang zu erweitern (siehe Abschnitt 3.3.6).

Auf einem betagten Rechner mit ...

- Windows 10 (64 Bit), Version 22H2
- einer Intel-CPU Core i3 aus dem Jahr 2010
- 8 GB RAM
- einer SSD als Festspeicher
- und einer vorhandenen Installation von Visual Studio 2019

hat die Installation ca. 11 Minuten gedauert:



Die Entwicklungsumgebung landet im Ordner:

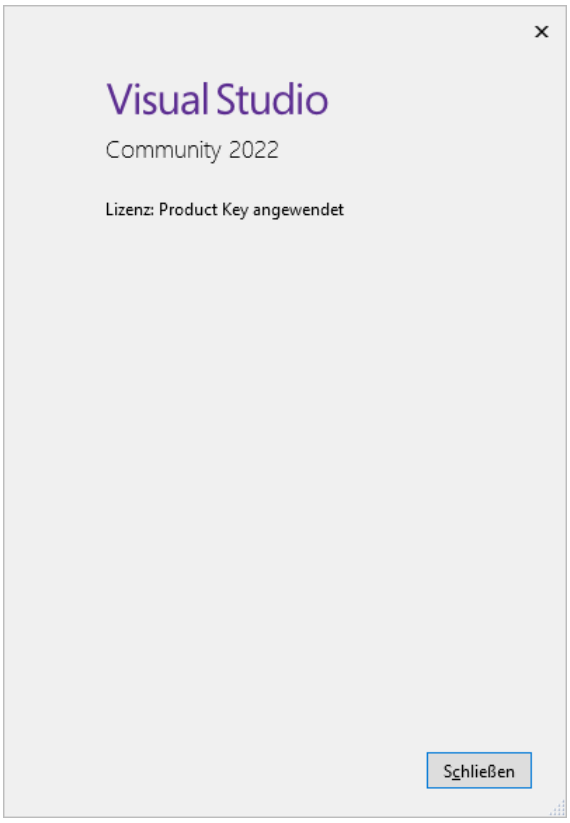
C:\Program Files\Microsoft Visual Studio\2022\Community

Bei Bedarf werden das .NET SDK in der aktuellen Version 7 sowie die .NET – Laufzeitumgebung in der aktuellen STS-Version 7 sowie in der LTS-Version 6 installiert (vgl. Abschnitt 3.1.1).

3.3.4 Registrierung und Initialisierung

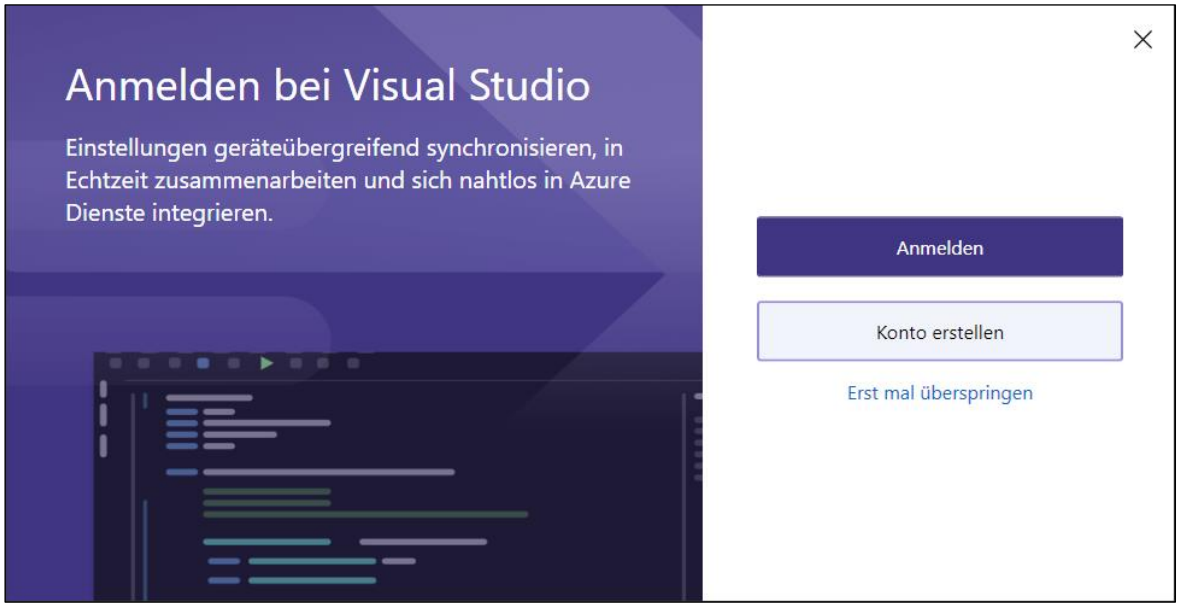
Per Voreinstellung startet das Visual Studio Community 2022 nach der Installation:

Die folgende Information



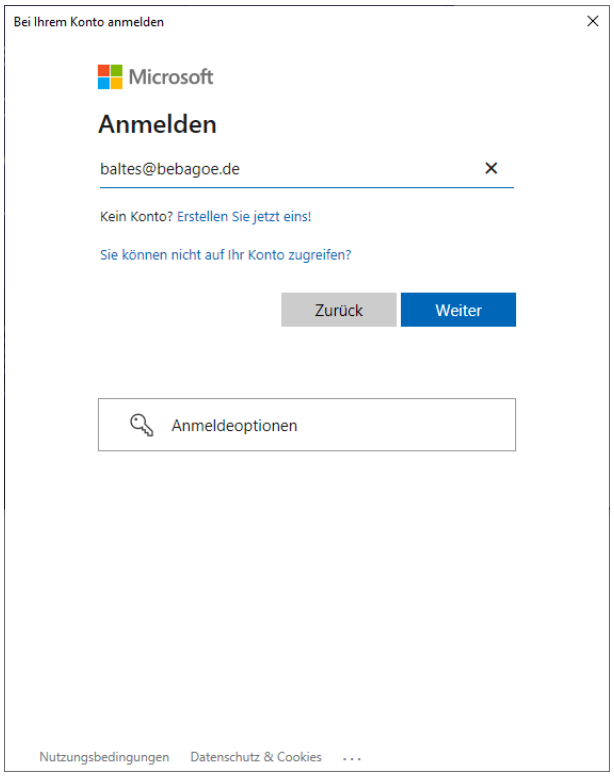
besagt offenbar, dass der zur Aktivierung einer Visual Studio - Installation benötigte **Product Key** im Fall der Community-Edition automatisch spendiert und angewendet wird.

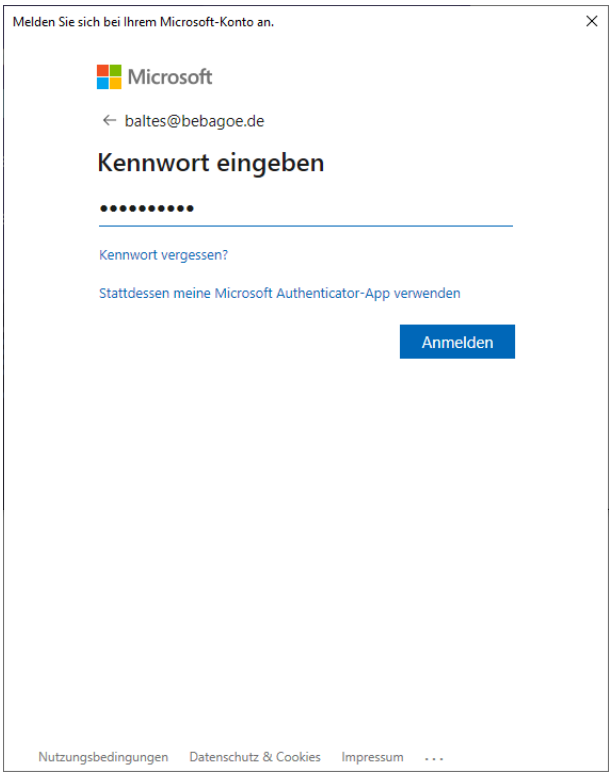
Microsoft verlangt für die kostenlose Variante seiner Entwicklungsumgebung eine Registrierung durch die Anmeldung mit einem Microsoft-Konto, die sich 30 Tage lang aufschieben lässt (Klick auf **Erst mal überspringen**):[1]



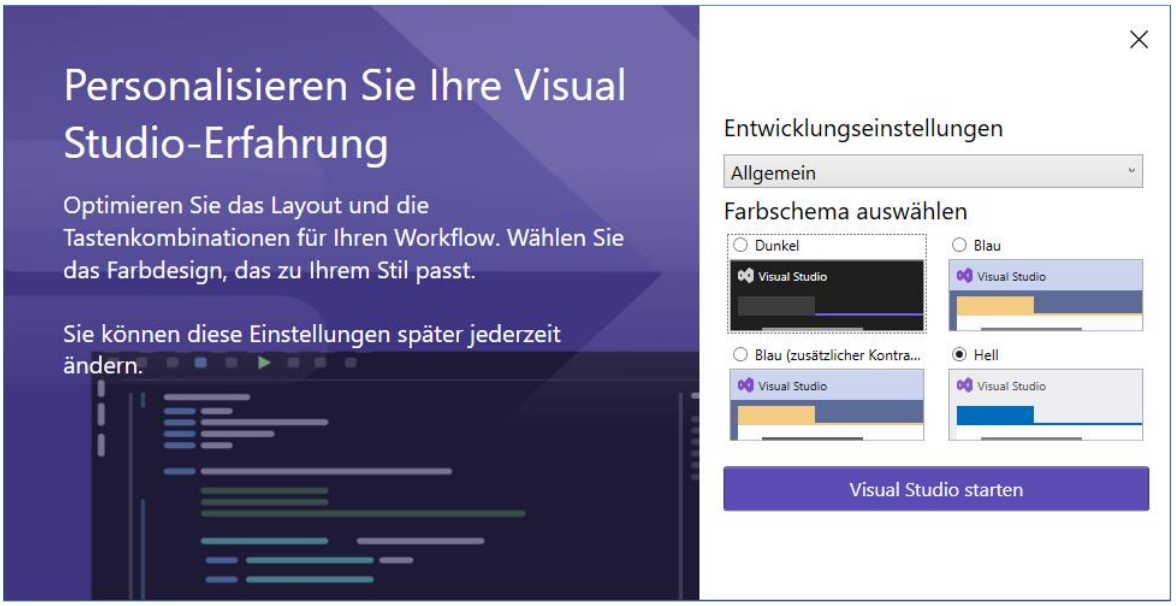
[1] Die 30 Tage mit unregistrierter Nutzungsdauer werden allerdings nicht pro Benutzer gewährt, sondern pro PC.

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

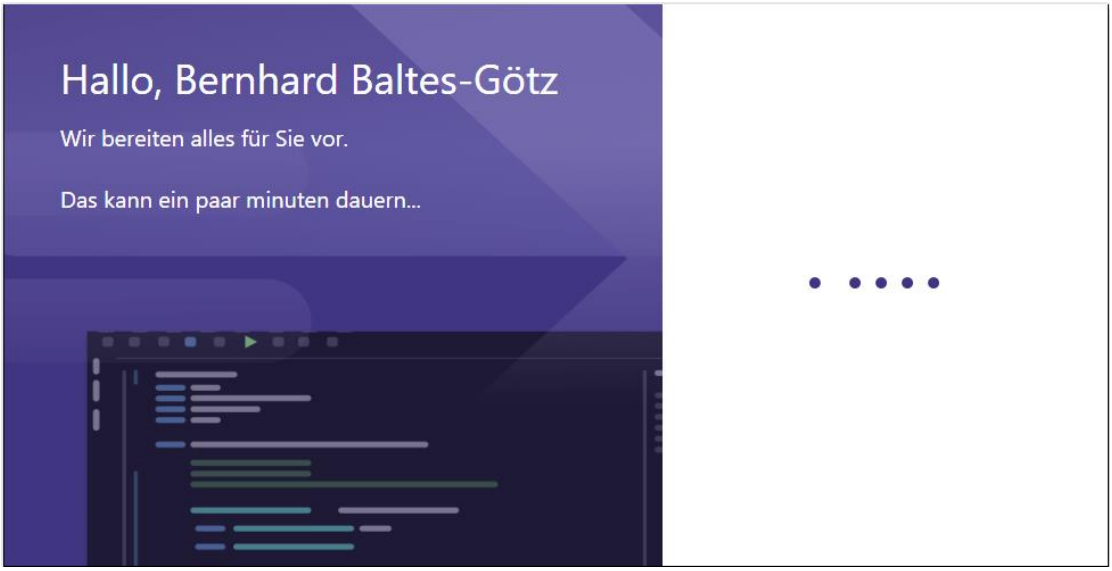




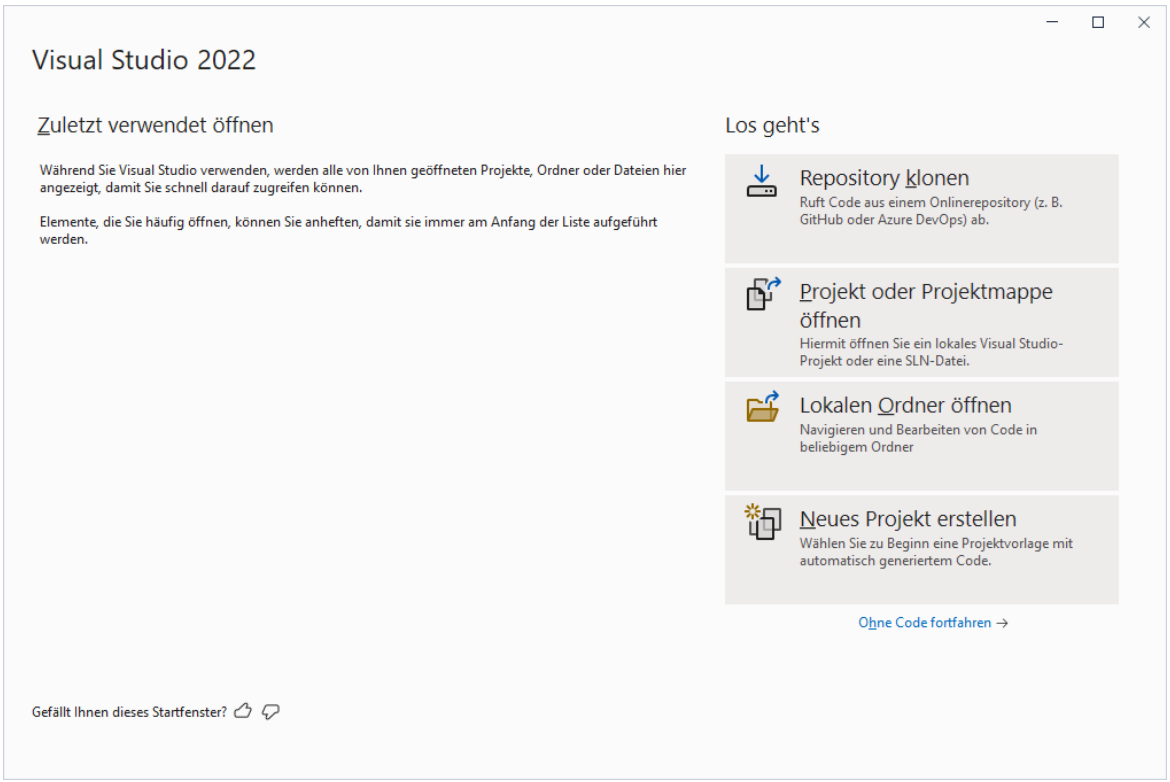
Wenn keine Einstellungen von einer Vorversion vorhanden sind, dann kann man ein Farbschema für die Bedienoberfläche der Entwicklungsumgebung wählen:



Beim ersten Start der Entwicklungsumgebung sind einmalige Vorbereitungen erforderlich:



Schließlich kann es losgehen:



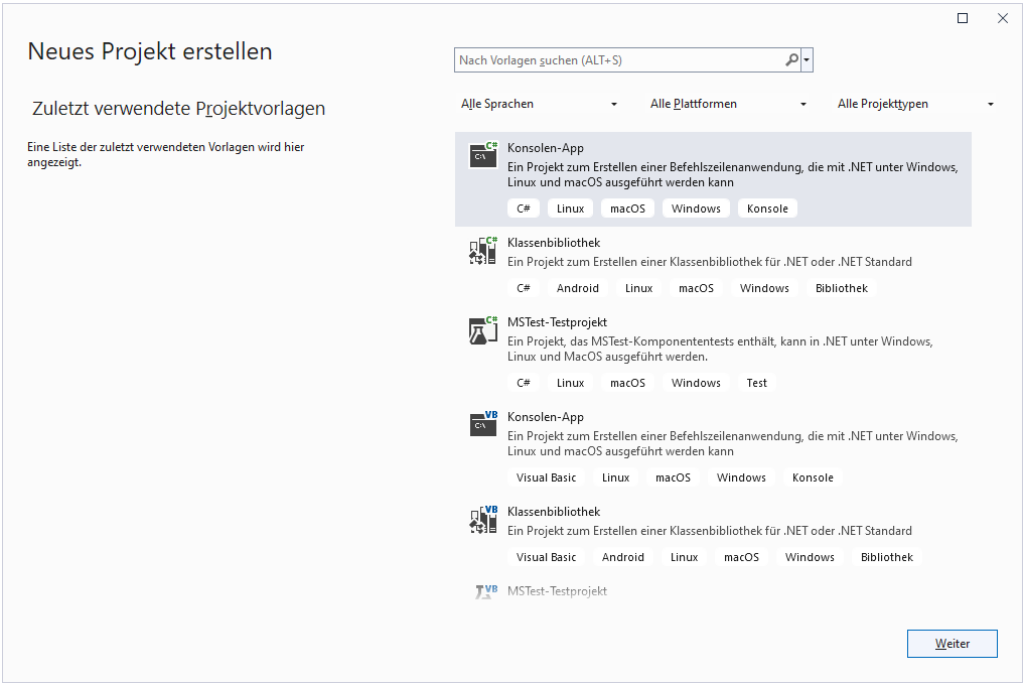
3.3.5 Ein erstes Konsolenprogramm

3.3.5.1 Projekt anlegen

Wir öffnen über die Option **Neues Projekt erstellen** im Begrüßungsdialog (siehe letztes Bildschirmfoto im Abschnitt 3.3.4) oder bei bereits vorhandenem Visual Studio - Anwendungsfenster mit dem Schalter  oder mit dem Menübefehl

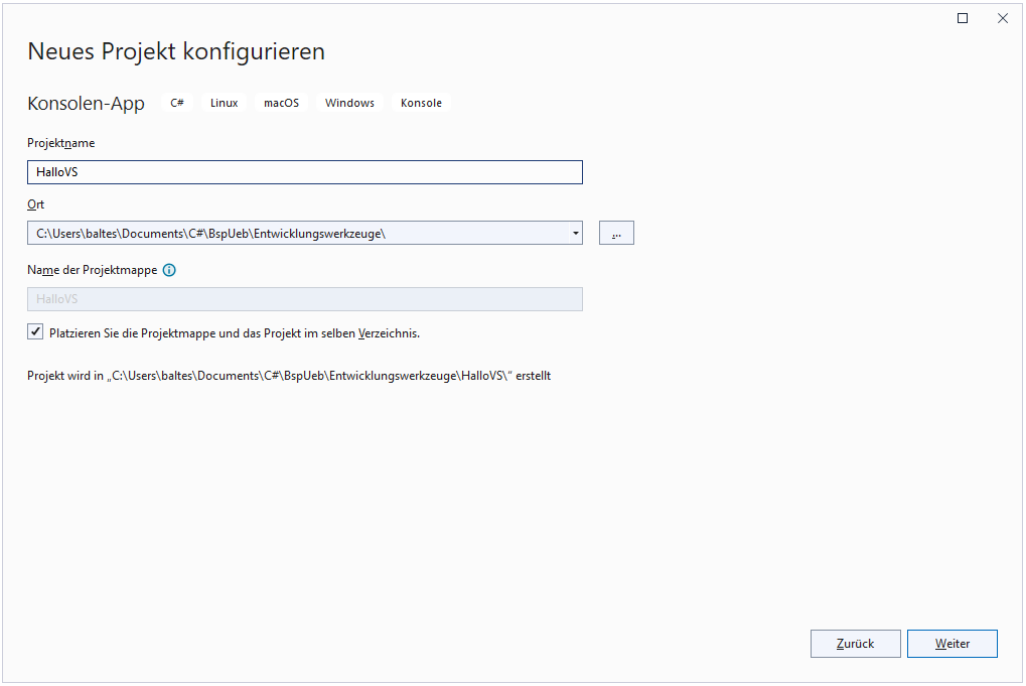
Datei > Neu > Projekt

den Dialog zum Erstellen von neuen Projekten:



Analog zur Wahl der Option **new console** im **dotnet**-Kommando (vgl. Abschnitt 3.1.2) entscheiden wir uns für die **Konsolen-App** in C#, um eine unter Linux, macOS und Windows einsetzbare Konsolenanwendung zu erstellen.

Im nächsten Dialog tragen wir den **Projektnamen** `HalloVS` ein, der als **Name der Projektmappe** übernommen wird:



Als **Ort**, an dem der Projektordner angelegt werden soll, schlägt das Visual Studio beim Windows-Benutzer `baltes` vor:

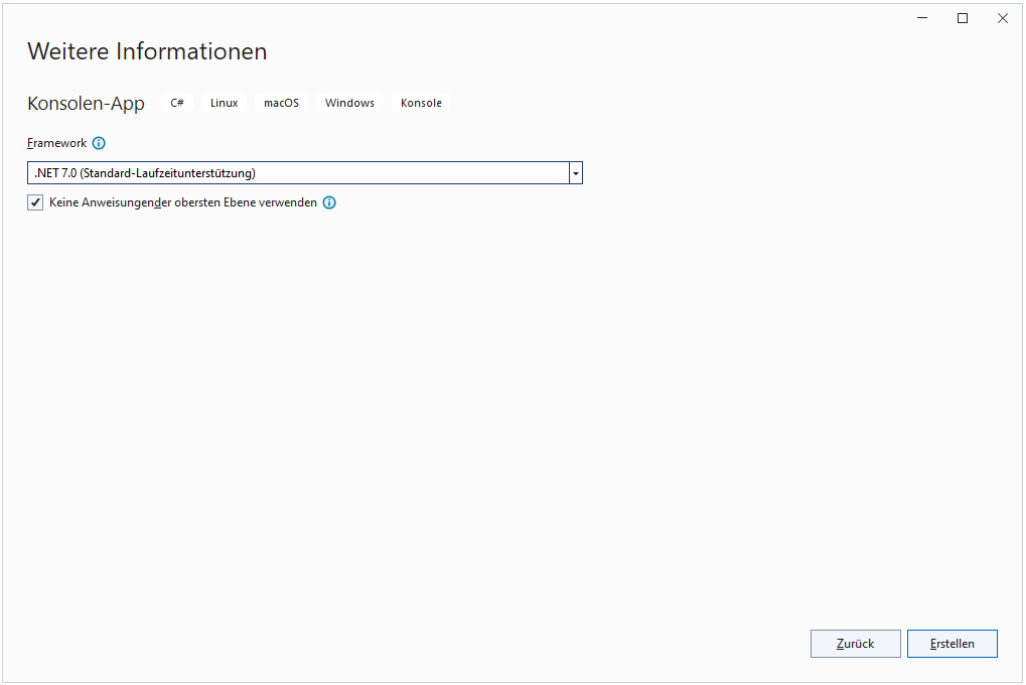
C:\Users\baltes\source\repos

Der folgende **Ort** hat gegenüber dem Vorschlag der Entwicklungsumgebung u. a. den Vorteil, dass er von manchen Windows-Sicherungsprozeduren automatisch einbezogen wird (z. B. **Dateien auf OneDrive sichern**):

C:\Users\baltes\Documents\C#\BspUeb\Entwicklungswerkzeuge

Jedes Visual Studio - Projekt gehört zu einer **Projektmappe**, die eine *Familie* von zusammengehörigen Projekten (z. B. Client- 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 meist nur ein einziges Projekt enthalten.[1] In dieser Situation ist es überflüssig, im Ordner einer Projektmappe einen Unterordner für das einzige enthaltene Projekt anzulegen. Daher markieren wir das Kontrollkästchen **Platzieren Sie die Projektmappe und das Projekt im selben Verzeichnis**. Im gemeinsamen Verzeichnis für das Projekt und die Projektmappe erscheint dann neben der Projektdatei (im Beispiel **HalloVS.csproj**) die Projektmappendatei (im Beispiel **HalloVS.sln**) mit einer von der englischen Bezeichnung *solution* abgeleiteten Namenserweiterung.

Wir machen **weiter**, und wählen die aktuelle .NET – Version 7 als Laufzeitumgebung. Bei einer an Kunden auszuliefernden Software kommt eher die LTS-Version 6 in Frage (vgl. Abschnitt 2.2.2.1).[2] Außerdem verzichten wir auf die **Anweisungen der obersten Ebene**, weil das Verstecken der Klassendefinition für Einsteiger aus didaktischen Gründen keine gute Option ist (vgl. Abschnitt 3.1.3). Sobald wir genau wissen, was der Compiler im Hintergrund beisteuert, werden wir die mit C# 9 eingeführten Schreibvereinfachungen nutzen.



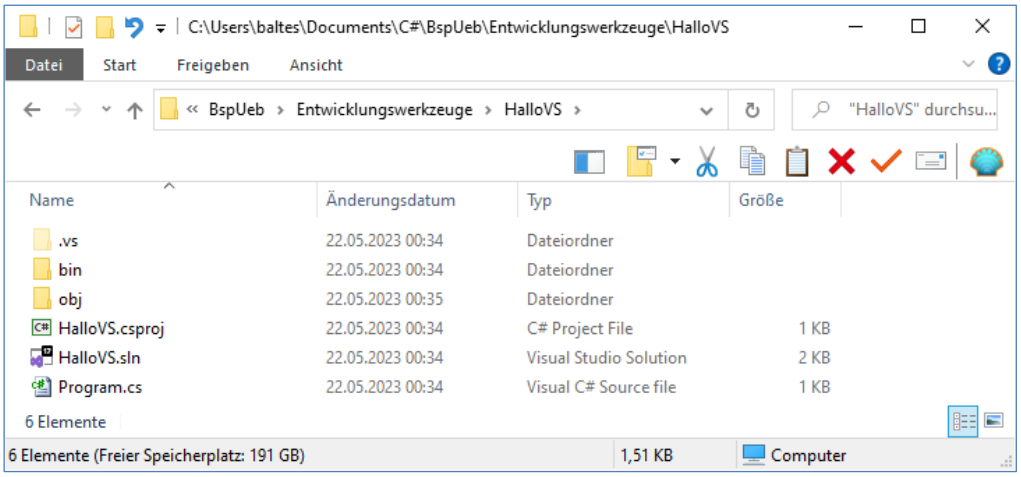
Nach einem Mausklick auf **Erstellen** entsteht im Projektordner

C:\Users\baltes\Documents\C#\BspUeb\Entwicklungswerkzeuge\HalloVS

---

[1] Im Abschnitt 3.3.9 befindet sich als Ausnahme von dieser Regel eine Mappe mit *zwei* Projekten, um einen Projektverweis demonstrieren zu können.

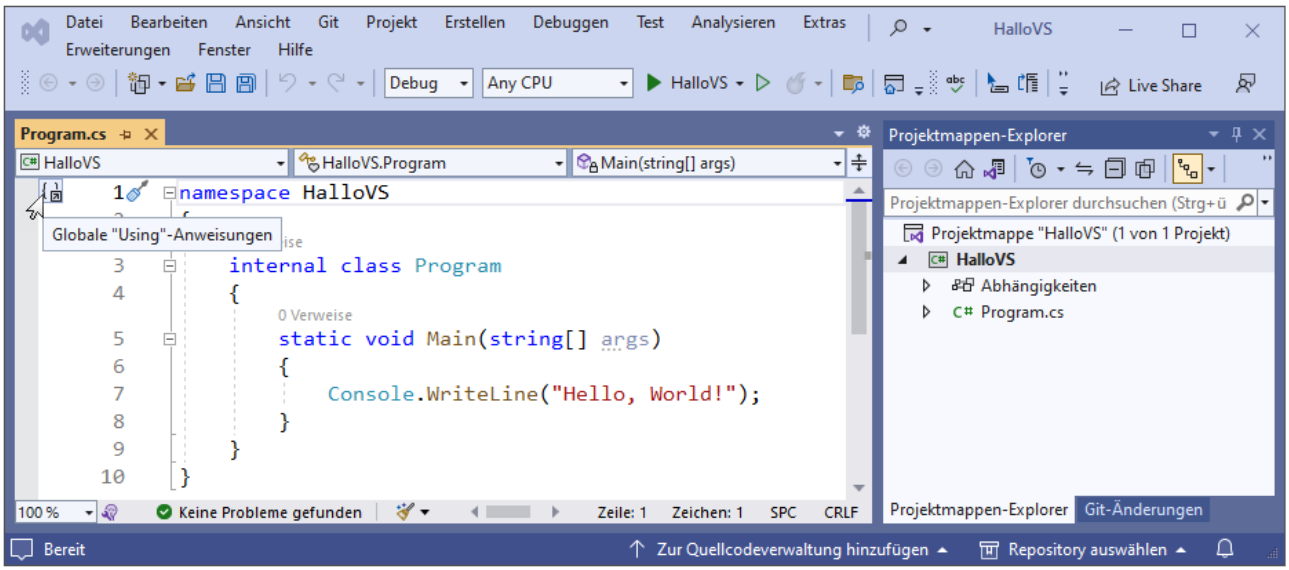
[2] Ein für .NET 7 erstelltes, framework-abhängiges Assembly kann z. B. auch von der im November 2023 zu erwartenden LTS-Version 8 der .NET – Laufzeitumgebung ausgeführt werden. Bei einer *inklusive Laufzeitumgebung* ausgelieferten Software (self-contained app, siehe Abschnitt 3.1.5) ist das Auslaufen der Unterstützung für die mitgelieferte Laufzeitumgebung allerdings ein Anlass, die Kunden mit einem Update zu versorgen. In diesem Fall ist ein möglichst später Zeitpunkt für alle Beteiligten von Vorteil.



das neue Projekt mit den folgenden Dateien, über die sich das Projekt später (z. B. per Doppelklick) öffnen lässt:

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

Das Visual Studio hat den Quellcode eines Hallo-Programms verfasst:



Im Vergleich zu unserem selbst codierten Hallo-Programm (siehe Abschnitt 3.1.3) fallen einige Unterschiede auf:

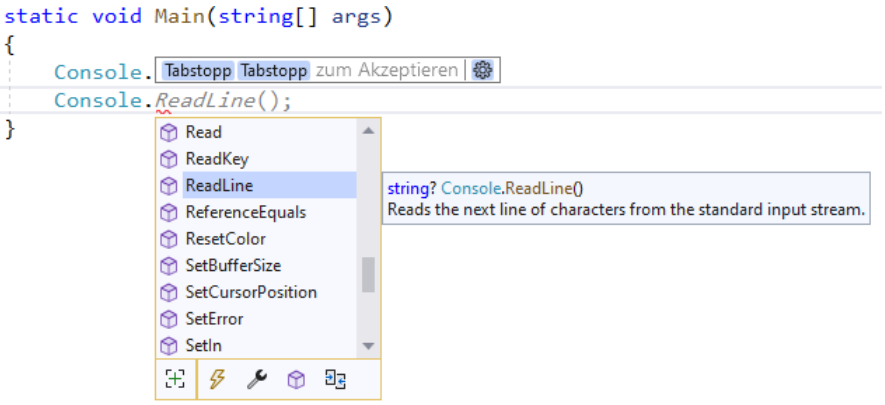
- Es ist keine **using**-Direktive für den Namensraum **System** vorhanden. Seit C# 10 sind für ein Projekt implizite **using**-Direktiven möglich, und einige sind zudem voreingestellt (siehe Abschnitt 2.6.3.2). Folglich wird im Beispielprogramm der Namensraum **System** doch importiert.
- Es wird ein eigener Namensraum definiert, was bei einem kleinen Übungsprogramm überflüssig ist.
- Für die Klasse mit dem Namen `Program` wurde die Schutzstufe **internal** deklariert, die ohnehin voreingestellt ist (siehe Abschnitt 5.12).

3.3.5.2 Editieren mit Komfort

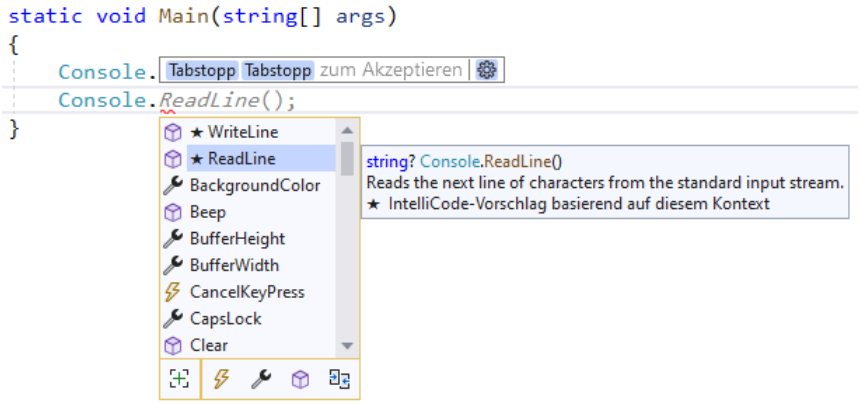
Damit bei einem Programmstart per Doppelklick auf die erstellte Datei **HalloVS.exe** das automatisch erscheinende Konsolenfenster nach der Ausgabe *nicht* spontan verschwindet, sondern bis zur Betätigung der **Enter**-Taste stehen bleibt, bitten wir am Ende der **Main()** - Methodendefinition die

Klasse **Console** darum, ihre Methode **ReadLine()** auszuführen. Diese Methode wartet auf die **Enter**-Taste und verhindert so, dass die Konsolenanwendung nach der Ausgabe sofort verschwindet.

Wir fügen im Quellcodeeditor am Ende der Methode **Main()** eine neue Zeile ein, schreiben den Namen der Klasse **Console** und setzen einen Punkt dahinter. Daraufhin erscheinen die erlaubten Fortsetzungen, also im konkreten Fall die öffentlichen Mitglieder der Klasse **Console**. Noch besitzt die Entwicklungsumgebung keine Informationen zu der mutmaßlich geplanten Fortsetzung, sodass die syntaktisch zulässigen Fortsetzungen in alphabetischer Reihenfolge erscheinen (IntelliSense):



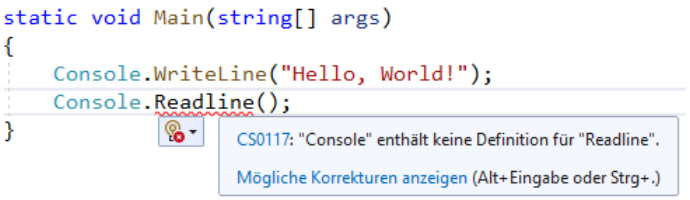
Wiederholt man die Anforderung (z. B. Punkt hinter dem Klassennamen löschen und neu setzen), dann startet die Angebotsliste mit kontext-abhängigen Favoriten (IntelliCode):



Wir wählen die Methode **ReadLine()** und betätigen die Tabulatortaste zweimal.

Der Quellcodeeditor unserer Entwicklungsumgebung bietet außerdem …

- die farbliche Unterscheidung verschiedener Sprachbestandteile,
- eine automatische Quellcode-Formatierung (z. B. bei Einrückungen)
- und eine automatische Syntaxprüfung, z. B.:

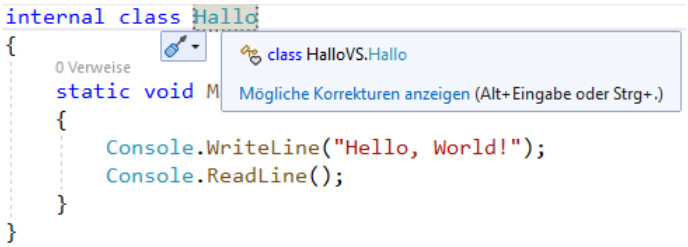


Im Beispiel wurde ein Methodenname falsch geschrieben. Die Erläuterung zum Fehler und das Unterstützungsangebot erscheinen, sobald sich der Mauszeiger über der unterschlängelten Fehlerstelle befindet. Über die Tastenkombination **Alt + Enter** sind Korrekturvorschläge verfügbar:

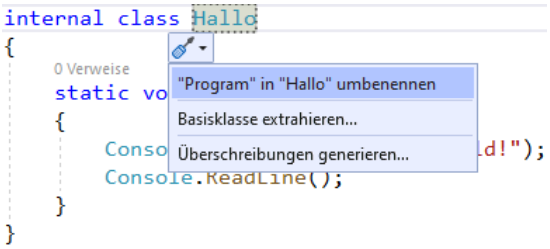
Im Beispiel soll die Startklasse den passenderen Namen `Hallo` erhalten. Das Umbenennen einer Klasse, Methode oder Variablen kann in einem komplexen Projekt diverse Quellcodeänderungen (in verschiedenen Dateien) erfordern, also aufwändig und fehleranfällig sein. Zum Glück kann unsere Entwicklungsumgebung solche Umgestaltungen, die man zu den *Refaktorierungen* rechnet, sehr gut unterstützen: Wenn man …

- die Klasse umbenennt,
- mit der Maus auf den nunmehr eingerahmten Klassennamen klickt
- und den Mauszeiger über dem eingerahmten Klassennamen bewegt,

dann erscheint ein Werkzeugsymbol,



das unter den **Schnellaktionen** u. a. das projekt-globale Umbenennen der Klasse anbietet:



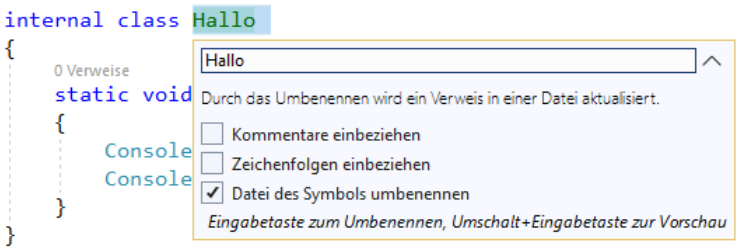
Im aktuellen, sehr einfachen Beispielprogramm ist kein weiteres Auftreten des Klassennamens zu aktualisieren. Sie werden aber bald die Möglichkeit schätzen lernen, in einem komplexen Programm einen Bezeichner zu ändern, ohne über die damit an vielen Stellen erzwungenen Aktualisierungen nachdenken zu müssen.

Statt durch Umbenennen eine problematische Lage zu provozieren, auf die das Visual Studio mit dem Angebot von Schnellaktionen 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** (**Strg**-Taste festhalten, zweimal **R**-Taste drücken, **Strg**-Taste loslassen) oder mit dem folgenden Menübefehl:

  Bearbeiten > Umgestalten > Umbenennen

- Ändern Sie den Bezeichner, und drücken Sie die **Enter**-Taste:



Aufgrund des markierten Kontrollkästchens **Datei des Symbols umbenennen** wird auch die zur Klasse gehörige Quellcodedatei umbenannt (in **Hallo.cs**).

Das Projekt kann jederzeit über den Schalter oder den Menübefehl

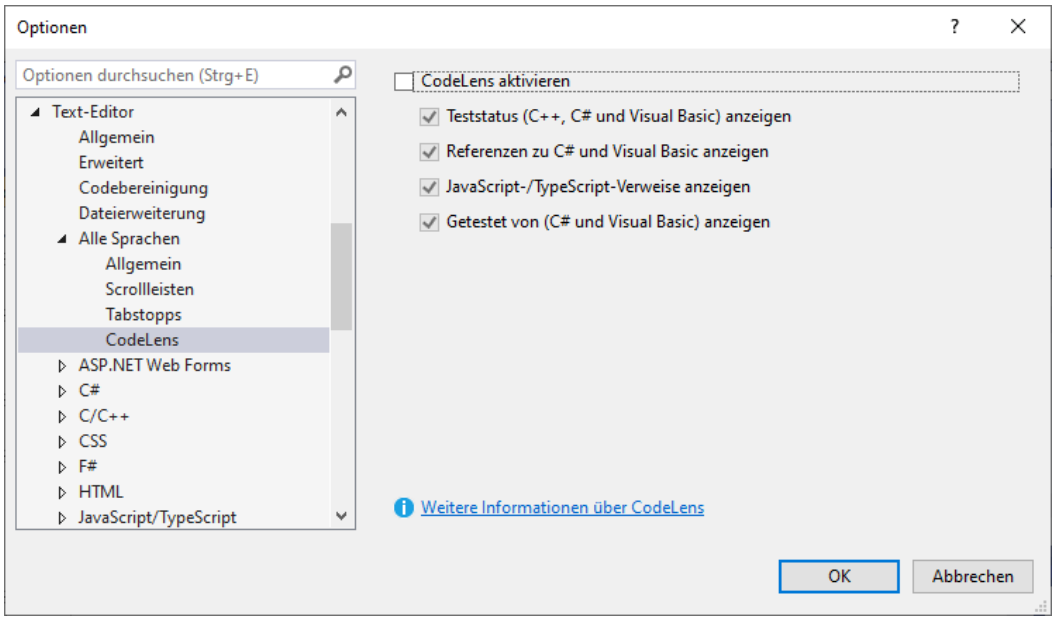
Datei > Alles Speichern

gespeichert werden. Allerdings speichert die Entwicklungsumgebung auch unaufgefordert zu passenden Gelegenheiten (z. B. beim Erstellen des Programms).

Vor den Definitionsköpfen der Klasse bzw. der **Main()** - Methode hat die per Voreinstellung aktive **CodeLens**-Funktion (nach einiger Bedenkzeit) die Anzahl der im Quellcode des Projekts vorhandenen **Verweise** (Verwendungen der Klasse bzw. Methode) eingetragen. Wenn das stört, kann die Funktion nach

Extras > Optionen > Text-Editor > Alle Sprachen > CodeLens

abgeschaltet werden:



3.3.5.3 Programm erstellen und starten

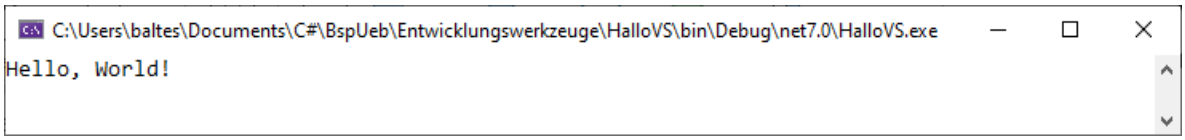
Nach der Aufforderung durch die Funktionstaste **F5**, die Schaltfläche HalloVS oder den Menübefehl

Debuggen > Debugging starten

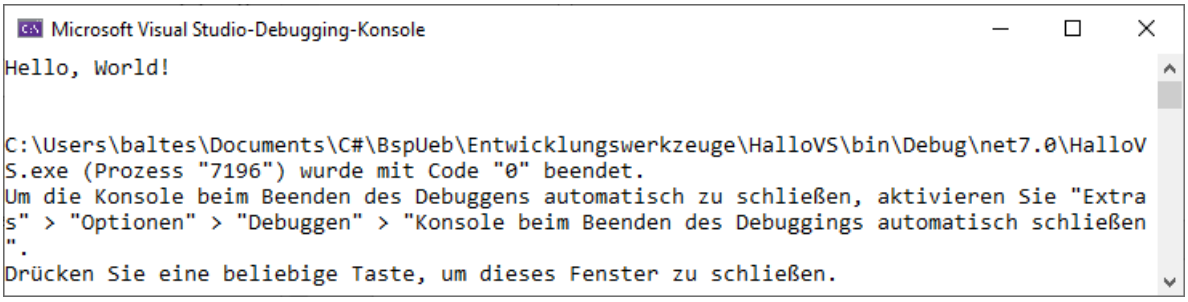
veranlasst das Visual Studio über das MSBuild-System die folgenden Aktionen:

- Der Quellcode wird vom Roslyn-Compiler übersetzt.
- Das Programm wird erstellt (inkl. **HalloVS.dll** und **HalloVS.exe**).

Schließlich startet das fertige Programm in einem Konsolenfenster:



Sobald Sie die **Enter**-Taste drücken, kehrt die von der Klasse **Console** ausgeführte **ReadLine()** - Methode zurück. Danach endet mit der **Main()** - Methode das Programm. Um das aus dem Visual Studio gestartete Konsolenfenster zu schließen,

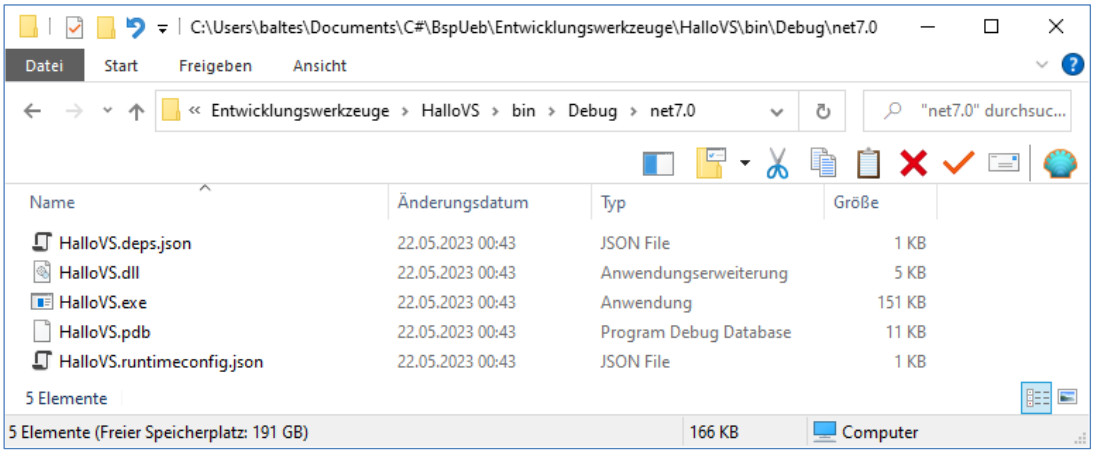


ist noch eine weitere Betätigung der **Enter**-Taste erforderlich.

MSBuild erstellt im folgenden Projektunterordner

...\bin\Debug\net7.0

ein gut testbares, aber nicht leistungsoptimiertes Assembly, z. B.:

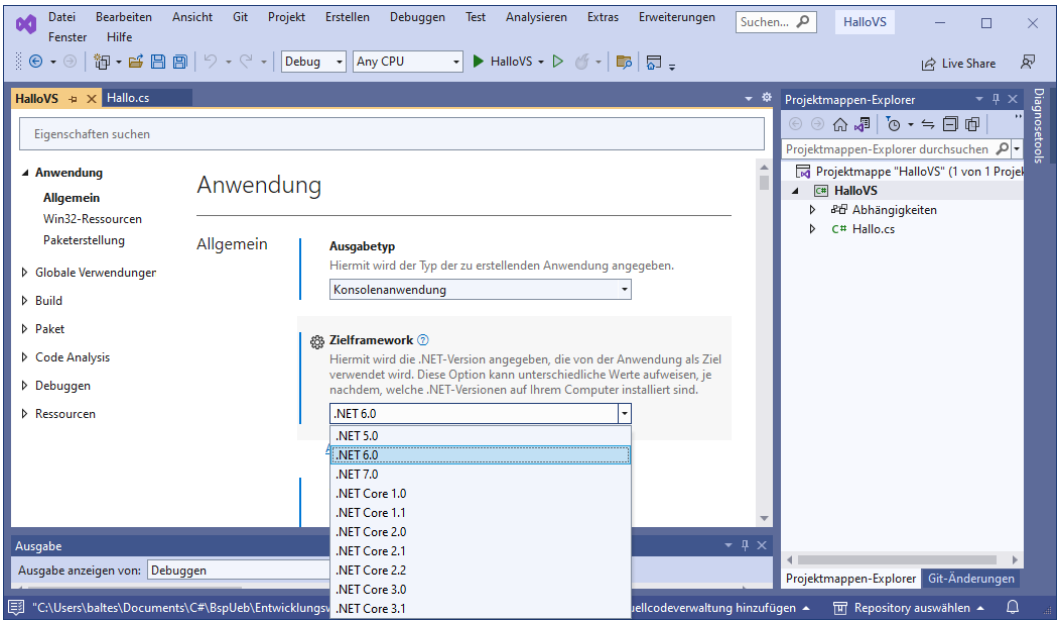


Es entstehen dieselben Dateien, die wir im Abschnitt 3.1.4 bei der Programmerstellung über das dotnet-CLI festgestellt haben, weil in beiden Fällen im Hintergrund das MSBuild-System arbeitet.

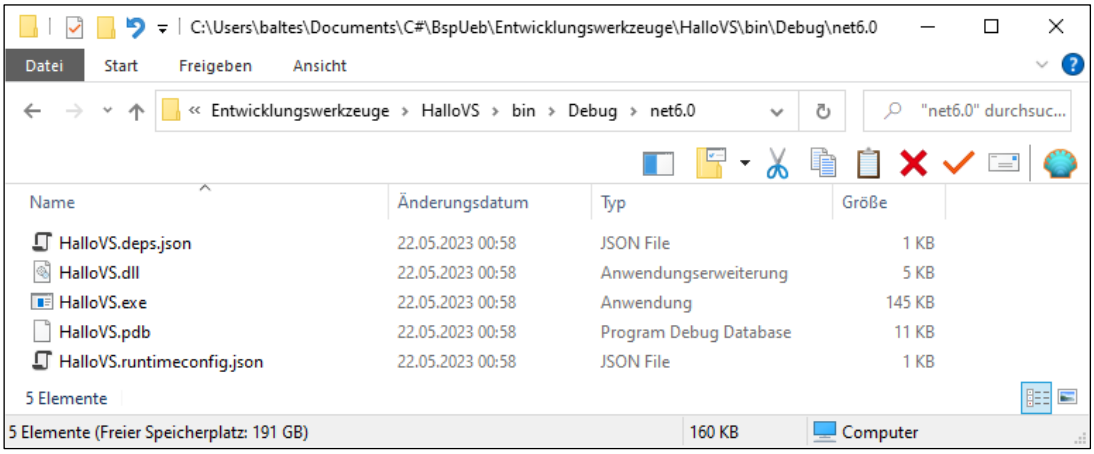
Ersetzt man über

Projekt > HalloVS-Eigenschaften

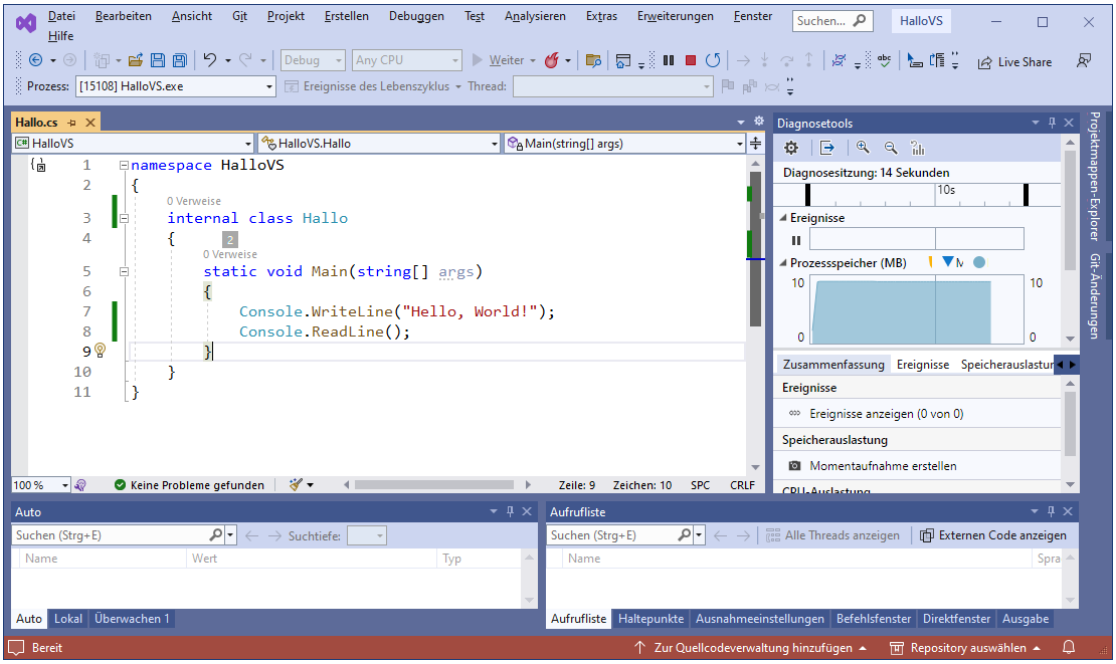
als **Zielframework** das beim Anlegen des Projekts gewählte **.NET 7.0** z. B. durch **.NET 6.0**,



dann ändert sich der Ausgabeordner für die Programmerstellung:



Zu einem mit der Funktionstaste **F5** oder mit der Schaltfläche ▶ HalloVS gestarteten Programm zeigt das Visual Studio diagnostische Informationen über das Laufzeitverhalten an, z. B.:

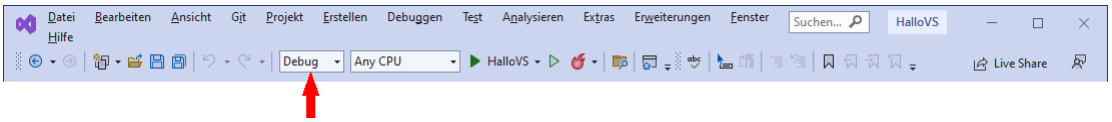


In der Regel benötigen wir diese Informationen nicht und starten Programme daher mit der Tastenkombination **Strg+F5** oder mit dem Menübefehl

Debuggen > Starten ohne Debugging

Dann verzichtet das Visual Studio während des Programmlaufs auf die Anzeige diagnostischer Informationen.

Das Erstellungssystem MSBuild verwendet aber weiterhin die **Debug-Konfiguration**



und erstellt im Ausgabeordner

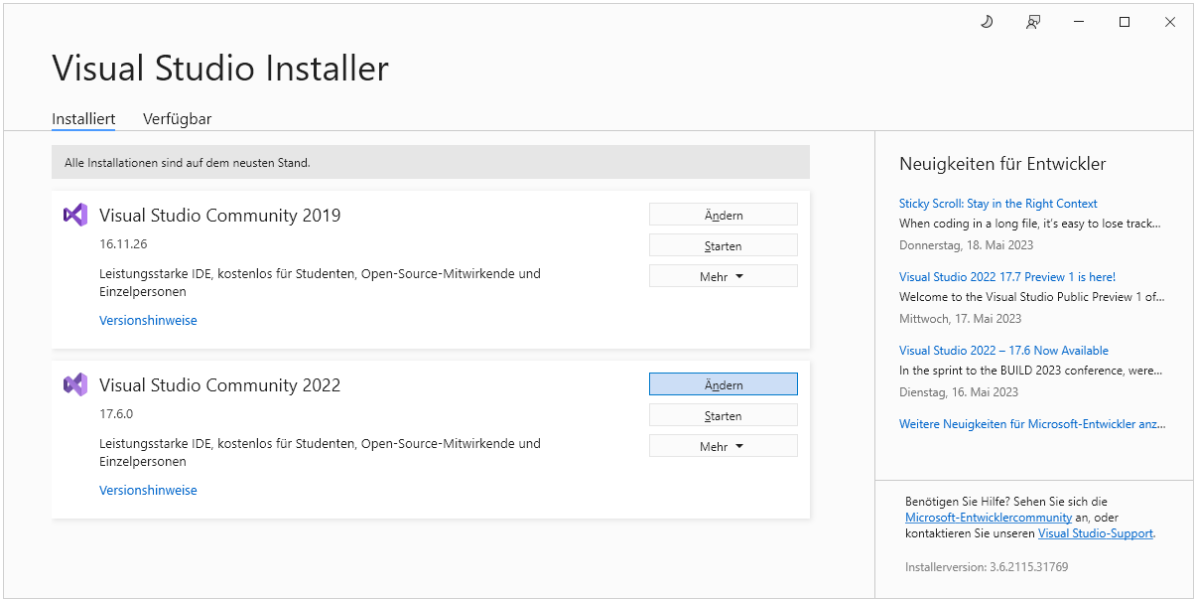
...\bin\Debug\net7.0

ein gut testbares, aber nicht leistungsoptimiertes Assembly. Mit der bei auslieferungsbereiten Programmen sinnvolleren **Release-Konfiguration** werden wir uns im Abschnitt 3.3.7.4 beschäftigen.

3.3.6 Installation modifizieren und aktualisieren

Eine Visual Studio - Installation lässt sich flexibel ändern, indem z. B. Sprachpakete oder Einzelkomponenten (wie der UML - Klassen-Designer) aufgenommen oder entfernt werden.

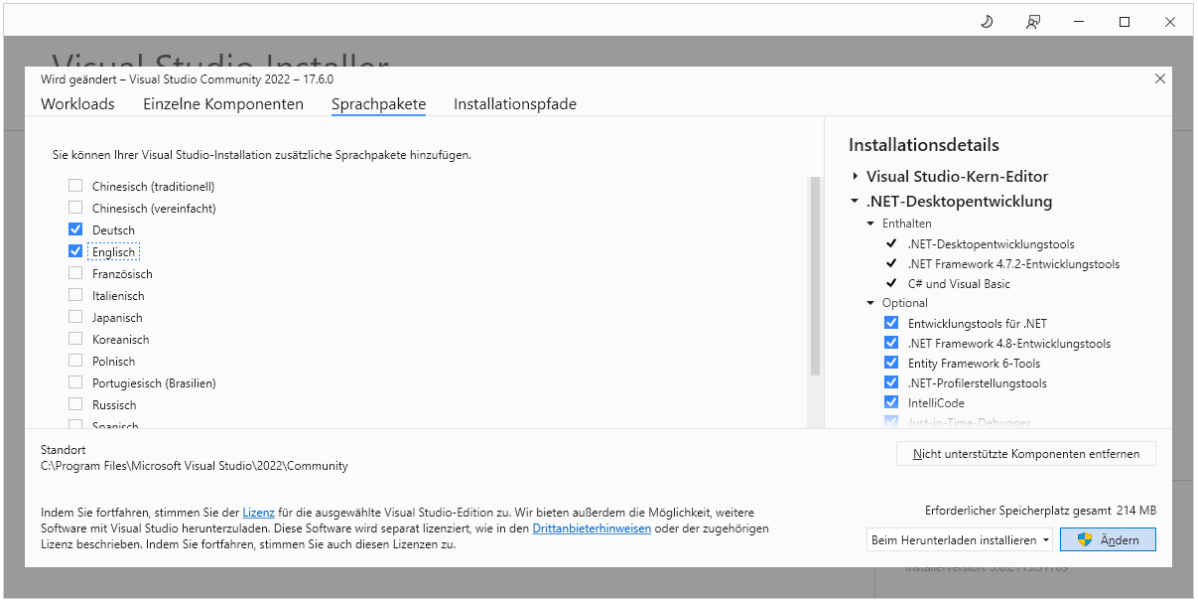
Starten Sie den **Visual Studio Installer** über seinen Eintrag im Startmenü,



und **ändern** Sie das Visual Studio Community 2022.

3.3.6.1 Bedienoberfläche in englischer Sprache

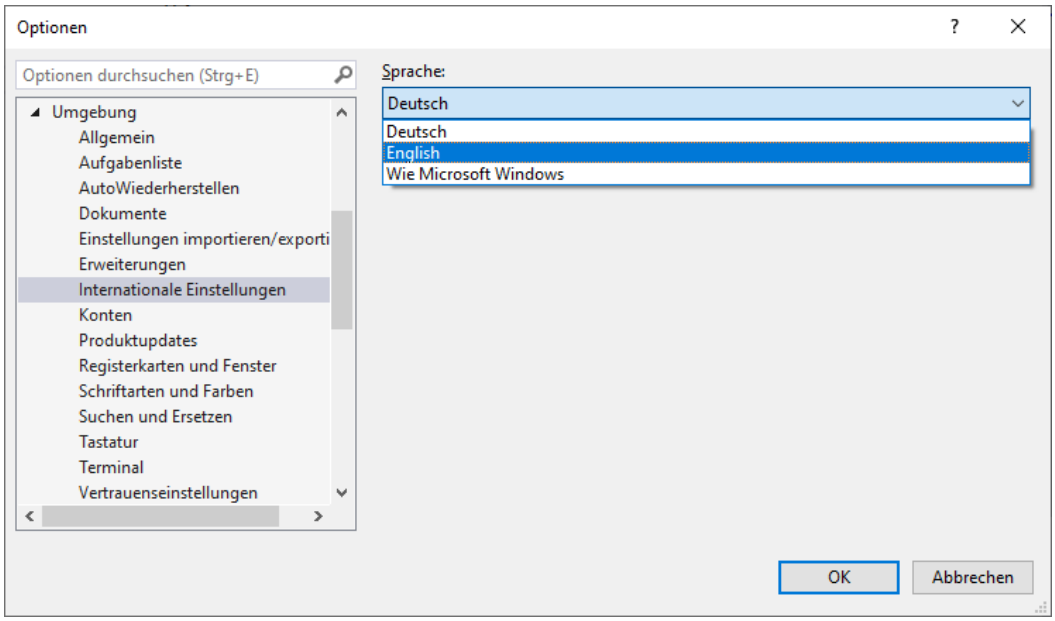
Ein (temporäres) Umschalten auf eine englische Bedienoberfläche kann nützlich sein, weil sich die meisten Anleitungen und Tipps auf die englische Bedienoberfläche beziehen. Daher sollten Sie auf der Registerkarte **Sprachpakete** das englische Sprachpaket hinzufügen:



Die mit einem Klick auf **Ändern** gestartete Installationserweiterung dauert einige Minuten. Nach dem nächsten Start der Entwicklungsumgebung kann über

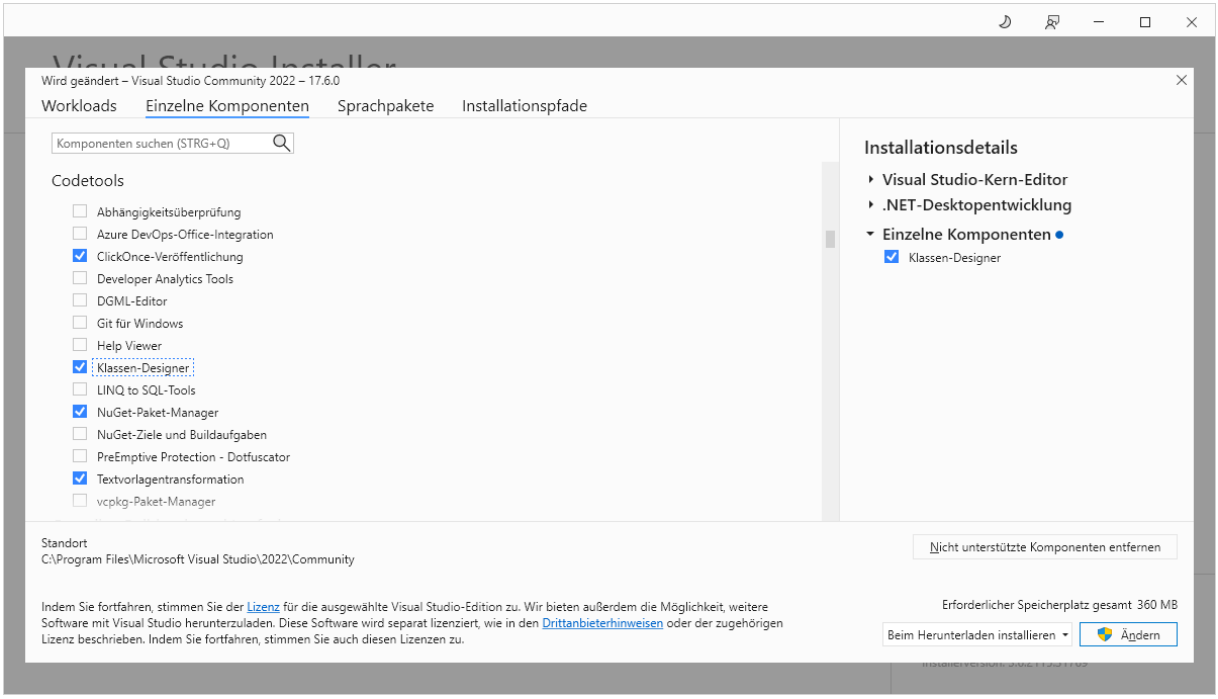
Extras > Optionen > Umgebung > Internationale Einstellungen

die Sprache der Bedienoberfläche geändert werden:



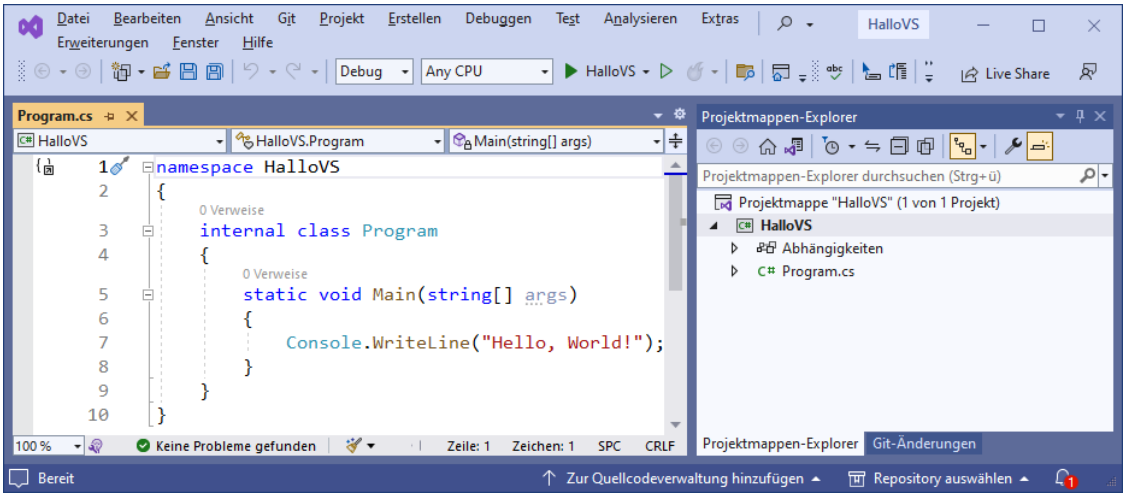
3.3.6.2 Klassen-Designer

Auf der Registerkarte mit den **einzelnen Komponenten** hält der **Visual Studio Installer** einige attraktive Optionen bereit. Im Bereich mit den **Codetools** sollten Sie den **Klassen-Designer** ergänzen, um in Ihren Projekten UML-Diagramme erstellen zu können (vgl. Abschnitt 1.2):

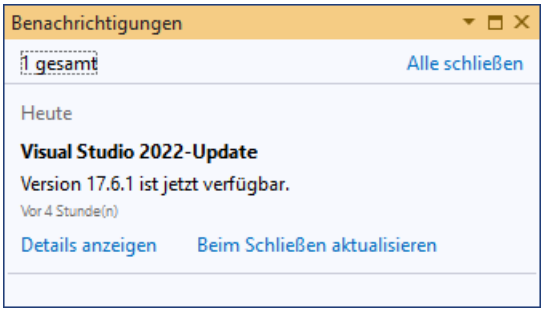


3.3.6.3 Kontinuierliche Aktualisierungen

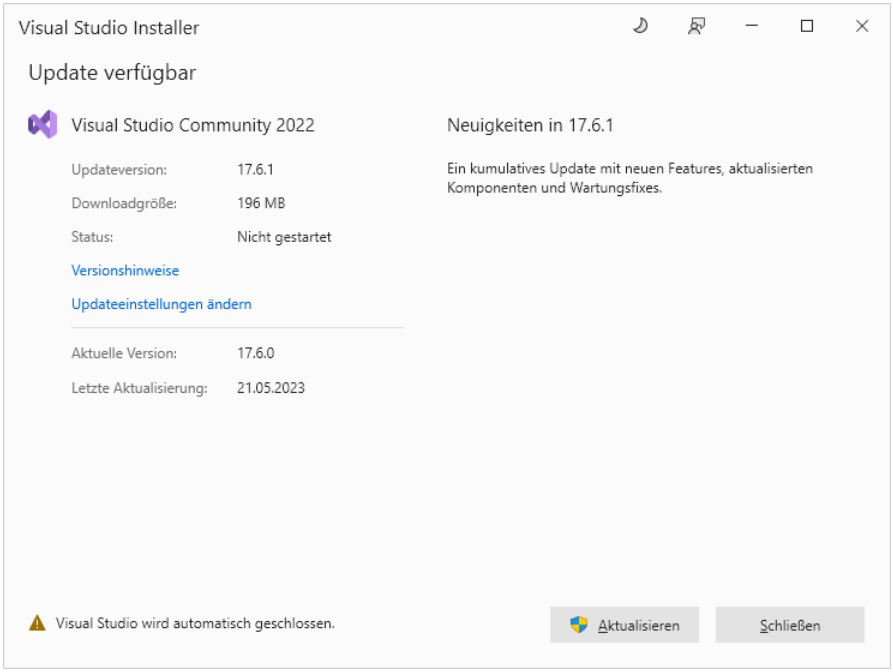
Im Abstand von wenigen Wochen erscheinen Aktualisierungen der Entwicklungsumgebung, die sich mit dem **Visual Studio Installer** bequem installieren lassen. Das Visual Studio informiert ggf. am rechten Rand der Statuszeile über neue Nachrichten:



Nach einem Klick auf das Glockensymbol mit der rot hinterlegten Anzahl neuer Nachrichten erfährt man nicht selten von einer verfügbaren Aktualisierung, z. B.:



Mit einem Klick auf **Details anzeigen** startet man den **Visual Studio Installer**:



Über den Schalter **Aktualisieren** veranlasst man den Installer, ...

- das Visual Studio zu schließen,
- das Update zu installieren,
- das Visual Studio wieder zu starten.

3.3.7 Ein erstes GUI-Programm

Dieser Abschnitt bietet zwecks Steigerung Ihrer Motivation einen Vorausblick auf die Erstellung von Programmen mit GUI-Bedienung (*Graphical User Interface*). Ein Ordner mit dem fertigen Visual Studio – Projekt ist hier zu finden:

…\BspUeb\Entwicklungswerkzeuge\DmToEuro

3.3.7.1 Projekt anlegen

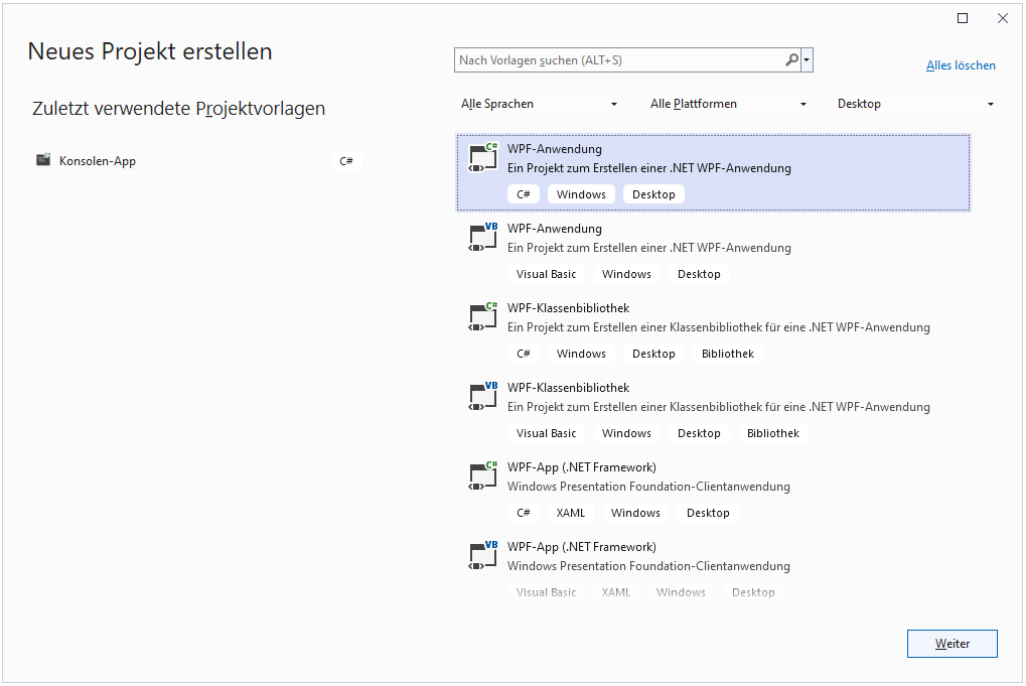
Es entsteht ein Währungskonverter mit grafischer Bedienoberfläche, der DM-Beträge in Euro-Beträge wandeln kann:[1]

Im aktiven Visual Studio wählen wir nach dem Menübefehl

Datei > Neu > Projekt

im folgenden Dialog aus der **Desktop**-Gruppe (Drop-Down – Menü oben rechts)

[1] Benötigt wird ein solches Programm z. B. noch bei der Auszahlung von Erbschaftsbeträgen aus Testamenten, die in der DM-Ära verfasst worden sind.



die Projektvorlage für **WPF-Anwendungen** in C#. Damit verwendet das Programm die GUI-Bibliothek *Windows Presentation Foundation* (*WPF*), die im Manuskript ...

- gegenüber der älteren GUI-Bibliothek *WinForms*,
- gegenüber der wenig erfolgreichen *Universellen Windows Plattform* (*UWP*)[1]
- und auch gegenüber .NET MAUI

bevorzugt wird.

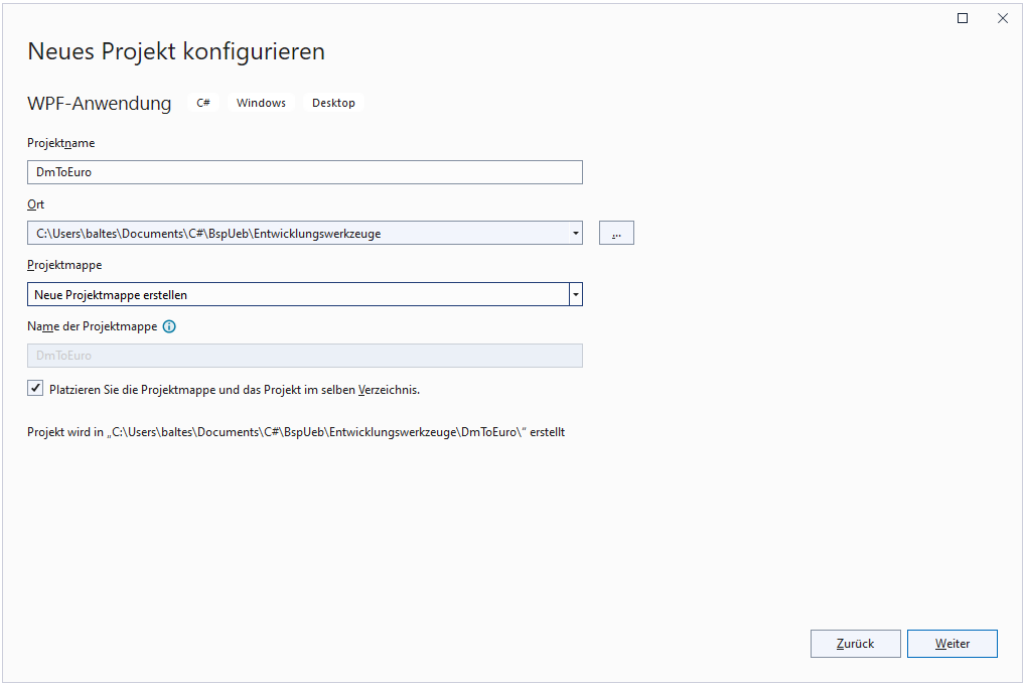
Wir machen **weiter** und legen einen **Namen** für das Projekt sowie einen **Ort** (ein übergeordnetes Verzeichnis) für den Projektordner fest, z. B.

C:\Users\baltes\Documents\C#\BspUeb\Entwicklungswerkzeuge

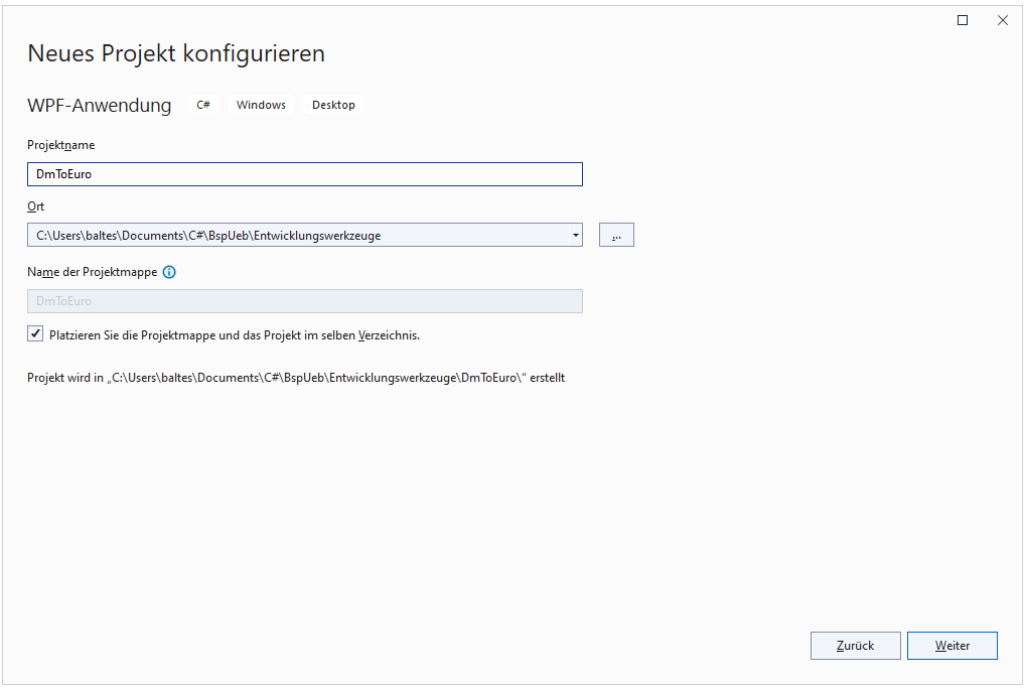
Jedes Visual Studio – **Projekt** befinden sich in einer **Projektmappe**, und zu großen Lösungen gehört eine Mappe mit *mehreren* Projekten. Bei unseren Beispielen entsteht aber fast immer eine Mappe mit einem einzigen Projekt, und beide sollten sich **im selben Verzeichnis** befinden. Weitere Informationen über Projektmappen sind im Abschnitt 3.3.5.1 zu finden.

Ist im Visual Studio bereits ein Projekt und damit auch eine Projektmappe geöffnet, dann sollten Sie eine **neue Projektmappe erstellen**, statt das neue Projekt in die vorhandene Projektmappe aufzunehmen:

[1] https://www.thurrott.com/dev/206351/microsoft-confirms-uwp-is-not-the-future-of-windows-apps



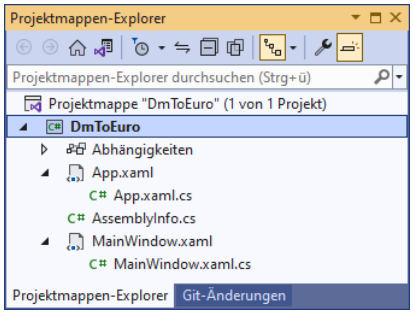
Ist keine Projektmappe geöffnet, dann wird auf jeden Fall eine neue Projektmappe angelegt:



Als Laufzeitumgebung wählen wir aus den im Abschnitt 3.3.5.1 diskutierten Gründen .NET 7.0:

Einige Sekunden nach einem Mausklick auf **Erstellen** präsentiert das Visual Studio am rechten Fensterrand im **Projektmappen-Explorer** eine Baumansicht zur Projektmappenverwaltung. Hier erscheint ein Eintrag für jedes Projekt in der potenziell aus mehreren Projekten bestehenden

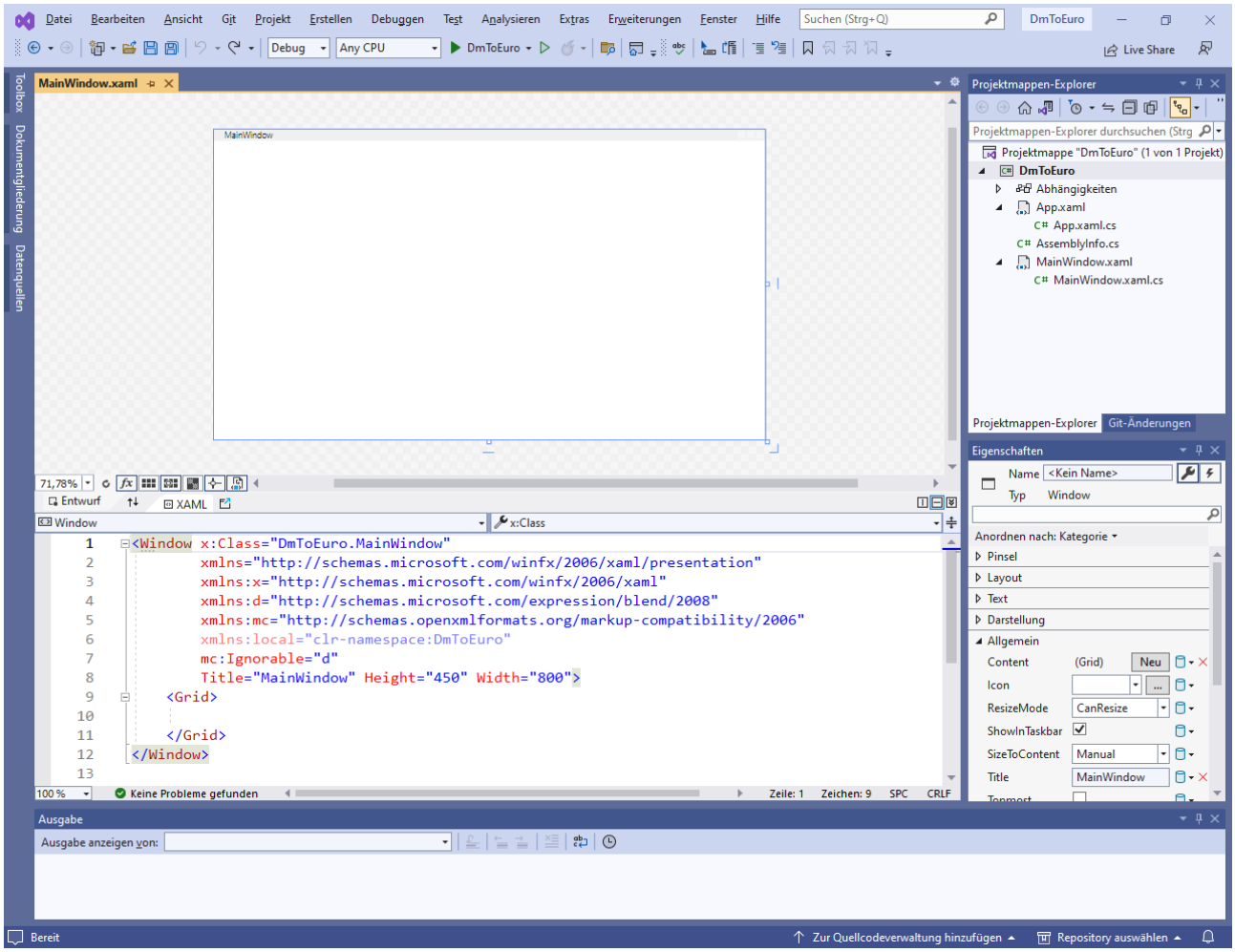
Projektmappe. Die Mappen zu unseren Beispielen enthalten in der Regel nur *ein* Projekt, und die Namen von Mappe und Projekt sind identisch, z. B.:



Zum Projekt werden aufgelistet:

- die vorausgesetzten Bibliothek (**Abhängigkeiten**)
  Damit werden wir uns bei passender Gelegenheit beschäftigen.
- die Quellcodedateien
- bei einer WPF-Anwendung die XAML-Dateien zur Bedienoberfläche und zur Anwendung

In der Editorzone des Visual Studio – Fensters präsentiert der WPF-Designer einen Rohling für das Fenster der Anwendung:



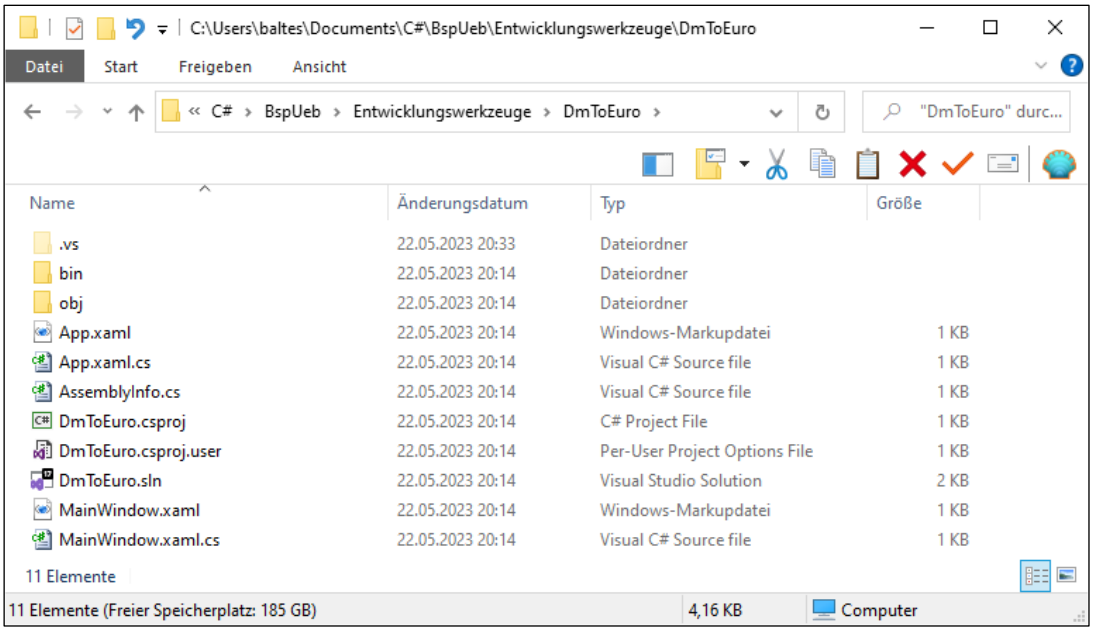
Das **Eigenschaften**-Fenster unter dem **Projektmappen-Explorer** ist beim Entwickeln einer WPF-Anwendung sehr nützlich. Schalten Sie es bitte nötigenfalls mit dem Menübefehl

Ansicht > Eigenschaftenfenster

oder mit der Funktionstaste **F4** ein. Im Beispiel wurde das zum Anwendungsfenster gehörende **Window**-Objekt über sein Element im XAML-Code markiert, um den Zweck des

Eigenschaften-Fenster zu demonstrieren: Dort sind alle Eigenschaften des aktuell markierten WPF-Objekts sicht- und modifizierbar.

Weil wir uns *gegen* einen Projektunterordner im Projektmappenordner entschieden haben, resultieren im Beispiel die folgenden Ordner und Dateien:



Wie beim Konsolenprogramm aus dem Abschnitt 3.3.5 ist für die Mappe und für das Projekt jeweils eine Konfigurationsdatei vorhanden:

- **DmToEuro.csproj** (C#-Projekt)
- **DmToEuro.sln** (Projektmappe, die Namenserweiterung **sln** steht für *Solution*)

Um das Projekt über den Windows-Explorer zu öffnen, kann man z. B. einen Doppelklick auf die Projekt- oder auf die Projektmappendatei setzen.

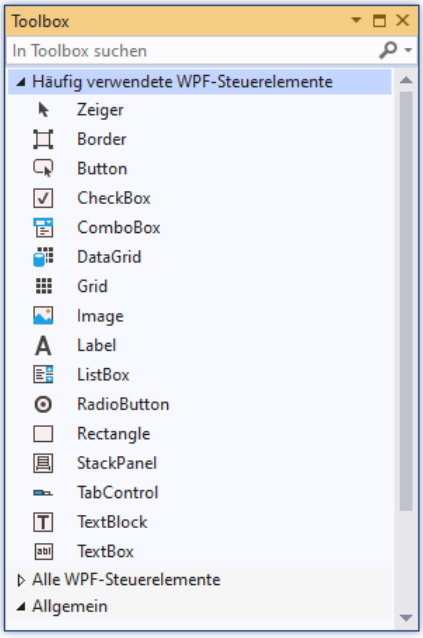
3.3.7.2 Bedienoberfläche entwerfen

Wir machen uns nun daran, das Anwendungsfenster unseres Währungskonverters mit den benötigten Bedienelementen auszustatten, die auch als *Steuerelemente* (engl.: *controls*) bezeichnet werden. Während wir mit dem WPF-Designer fast wie mit einem Grafikprogramm arbeiten, erstellt und pflegt dieser Assistent eine Deklaration der Bedienoberfläche in der *EXtensible Application Markup Language* (XAML). Mit zunehmendem Wissen über die XAML-Beschreibungssprache werden wir später unsere Abhängigkeit vom Assistenten reduzieren. In der aktuellen Lernphase bearbeiten 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 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**:

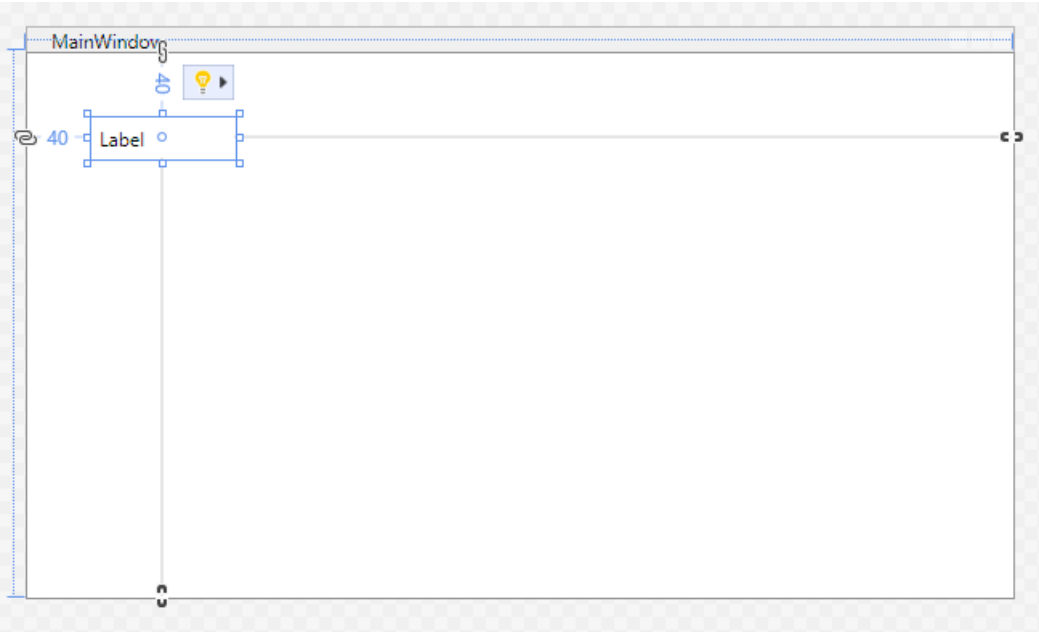


Erstellen Sie ein **Label**-Objekt (die Bezeichnung *Objekt* ist durchaus im Sinn von Kapitel 1 gemeint) auf dem Formular per Drag & Drop (Ziehen und Ablegen), indem Sie einen linken Mausklick auf den **Toolbox**-Eintrag **Label** setzen, die Maus dann mit gedrückter linker Taste zum Ziel bewegen und dort die Taste wieder loslassen.

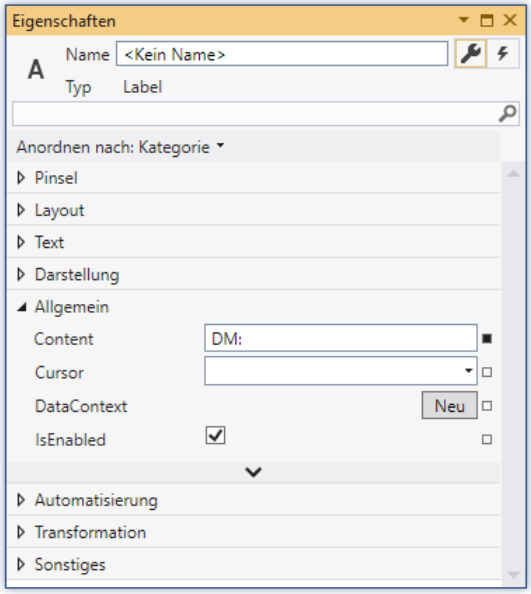
Durch die Wahl einer passenden Mauszeigerposition erhält man eines von den folgenden Werkzeugen:

- zum Bewegen von Objekten
- zur horizontalen Größenänderung
- zur vertikalen Größenänderung
- zur horizontalen *und* vertikalen Größenänderung
- zum Drehen

Damit lässt sich der folgende Zustand herstellen:

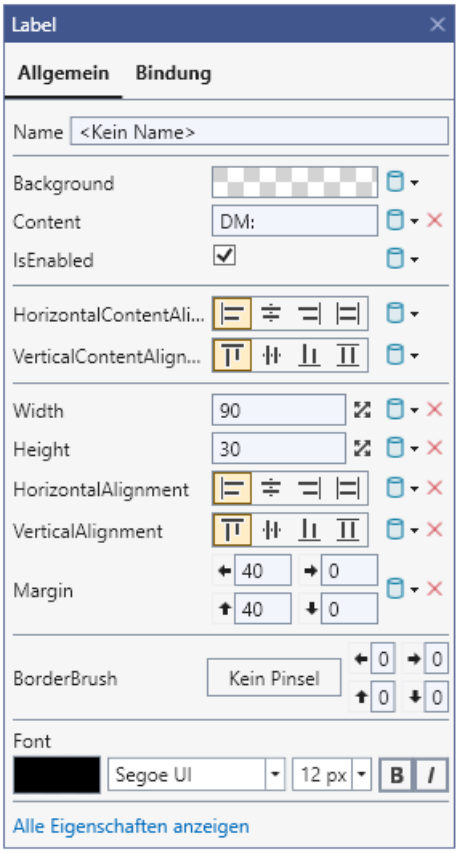


Ändern Sie die Beschriftung des **Label**-Objekts, indem Sie bei markiertem **Label**-Objekt im **Eigenschaften**-Fenster (unten rechts) einen passenden Wert für die **Allgemein**-Eigenschaft **Content** eintragen:



Dabei ist *Eigenschaft* im Sinn von Abschnitt 1.2 gemeint.

Über die Glühbirne, die im WPF-Designer zum markierten **Label**-Objekt erscheint, ist auch eine Kontextmenüversion des **Eigenschaften**-Fensters verfügbar:

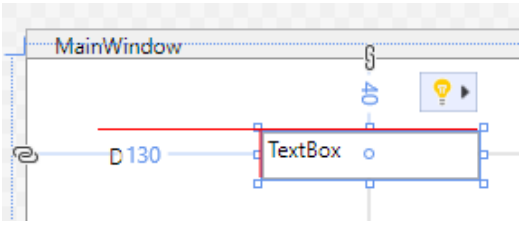


Hier kann man u. a. …

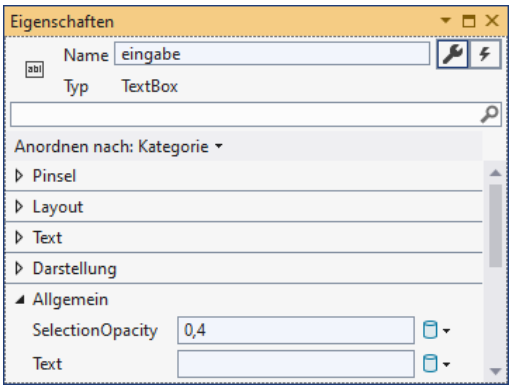
- die Breite und die Höhe über die Eigenschaften **Width** und **Height**
- sowie den Außenrand (die Abstände zum Fensterrahmen) über die **Margin**-Werte

bequemer einstellen als durch filigrane Mausarbeit.

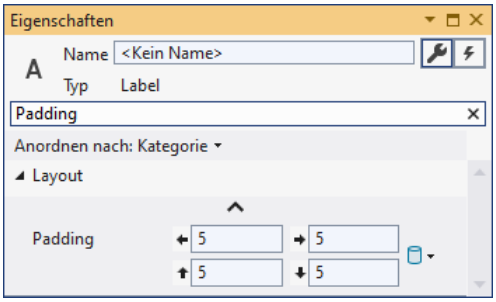
Setzen Sie ein Objekt der Klasse **TextBox** rechts neben das **Label**-Objekt, wobei die Entwicklungsumgebung die Anpassung von Position und Größe durch Hilfslinien erleichtert. In das **TextBox**-Steuerelement sollen die Benutzer den zu konvertierenden DM-Betrag schreiben:



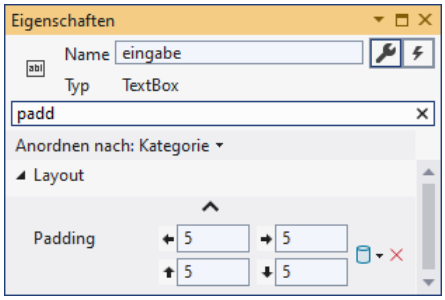
Der initial vorhandene Text stört und sollte gelöscht werden (Eigenschaft **Text** in der Kategorie **Allgemein** des **Eigenschaften**-Fensters). Weil das **TextBox**-Objekt in der gleich zu erstellenden Konvertierungsmethode angesprochen werden muss, erhält es einen **Namen** (z. B.: `eingabe`):



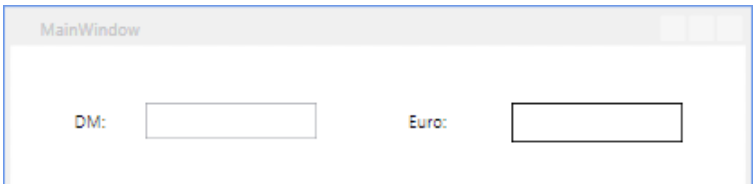
Die **Label**-Objekte haben per Voreinstellung einen Innenrand (**Padding**), der sich u. a. auf die Texthöhe auswirkt:



Damit der Text im **TextBox**-Element auf derselben Höhe erscheint, sollte hier derselbe Innenrand eingerichtet werden:



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

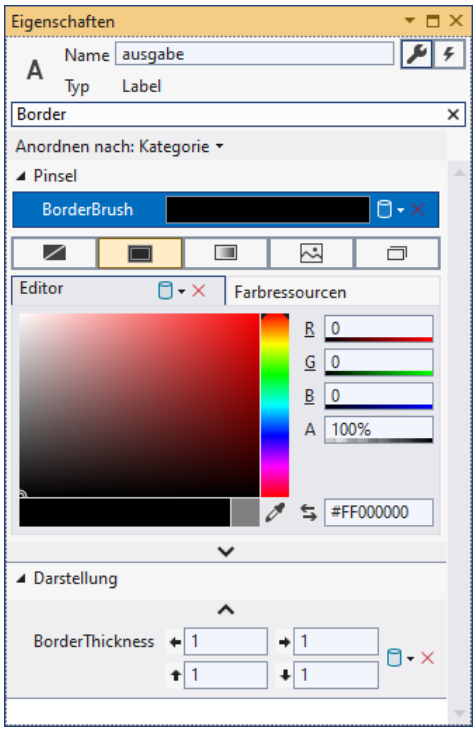


Während das erste **Label**-Objekt wie das DM-Pendant zur Beschriftung dient, soll das zweite den Ergebnisbetrag anzeigen, also insbesondere beim Programmstart leer sein. Wählen Sie passende **Content**-Eigenschaftsausprägungen für die Objekte.

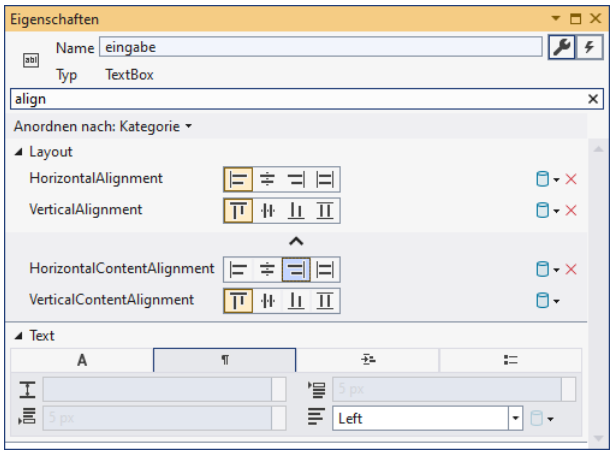
Um dem zweiten **Label**-Objekt auch initial einen optischen Auftritt zu verschaffen, sollte es eine Umrandung erhalten. Dazu ist über die Eigenschaft **BorderBrush** eine Randfarbe und über die Eigenschaft **BorderThickness** eine Randstärke festzulegen. Um eine Eigenschaft zu lokalisieren, können Sie im **Eigenschaften**-Fenster ...

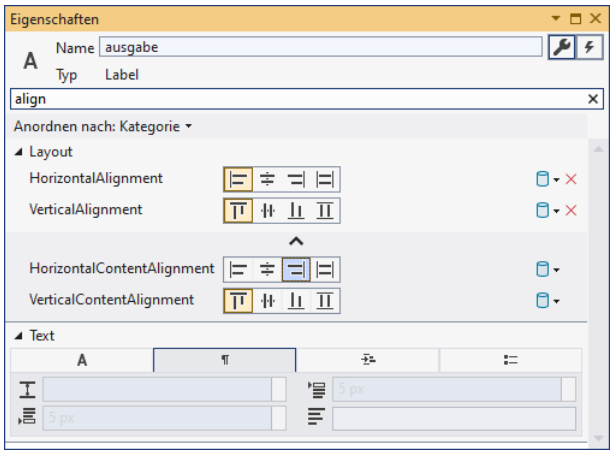
- die zugehörige Kategorie raten und aufklappen,
- ein **Anordnen** der Eigenschaften nach dem **Namen** veranlassen,
- nach der Eigenschaft suchen.

Das zur Anzeige des resultierenden Euro-Betrags vorgesehene **Label**-Objekt benötigt außerdem einen **Namen** (z. B. `ausgabe`), damit es in der gleich zu erstellenden Konvertierungsmethode angesprochen werden kann:

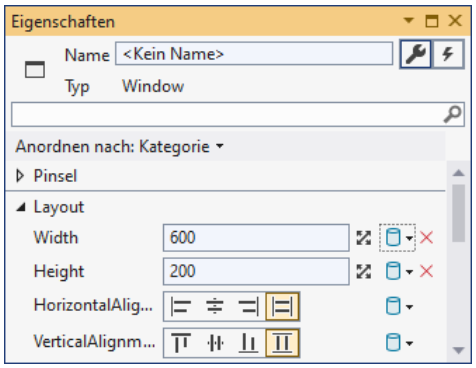


Außerdem sollte für die beiden Steuerelemente mit Geldbeträgen eine rechtsseitige Ausrichtung des jeweiligen Inhalts über die Eigenschaft **HorizontalContentAlignment** veranlasst werden:

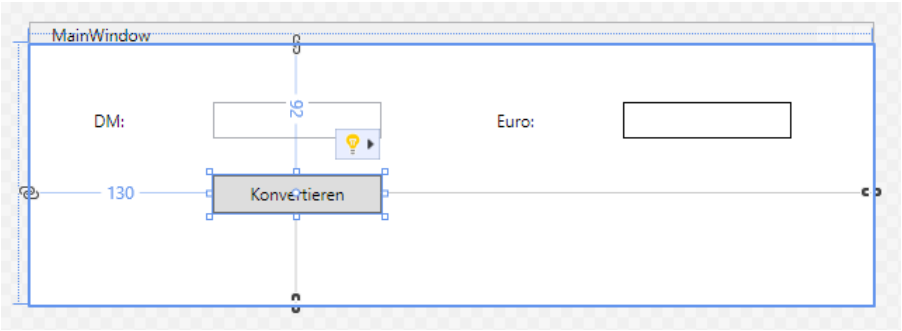




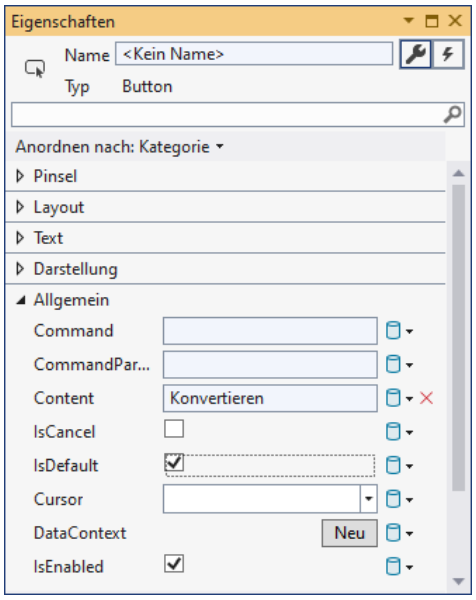
Mittlerweile hat sich gezeigt, dass die vorgegebene Fensterbreite überdimensioniert ist. Markieren Sie das gesamte Fenster (oder das **Window**-Element im XAML-Code), nicht etwa den **Grid**-Layoutcontainer (das **Grid**-Element im XAML-Code). Tragen Sie im **Eigenschaften**-Fenster passende Werte für die **Layout**-Eigenschaften **Width** und **Height** ein, z. B.:



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



Auch beim **Button**-Objekt sorgt man über die **Content**-Eigenschaft für eine passende Beschriftung. Wenn die Eigenschaft **IsDefault** in der Kategorie **Allgemein** per Kontrollkästchen den Wert **true** erhält, dann kann der Schalter im Programm auch per **Enter**-Taste betätigt werden:



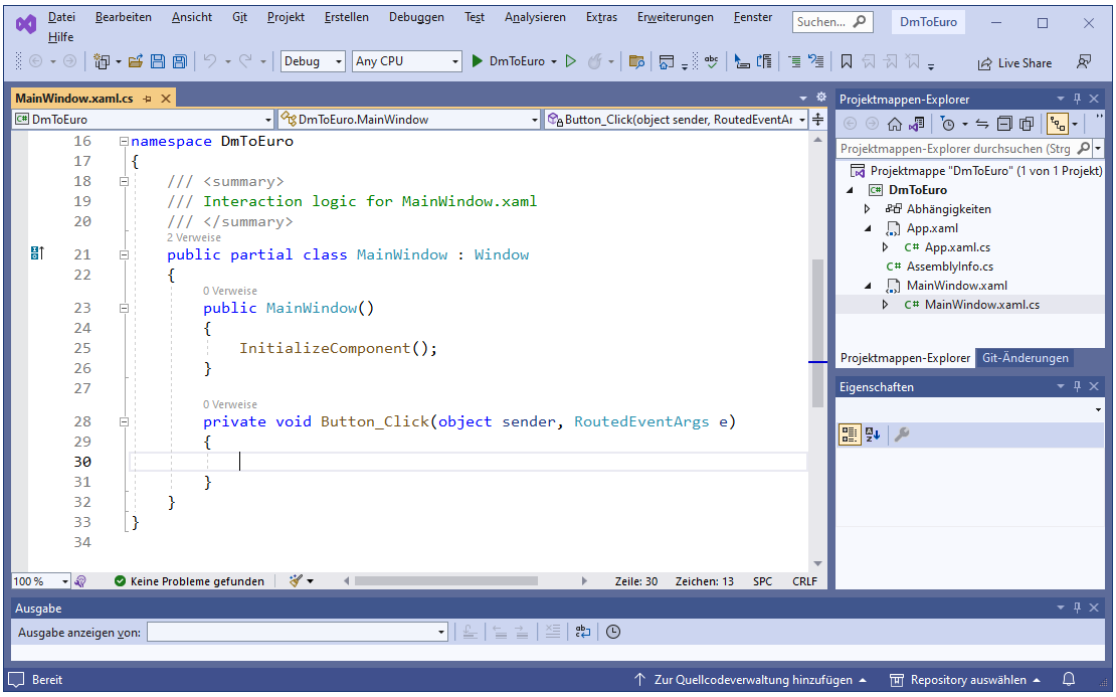
3.3.7.3 Behandlungsmethode zum Click-Ereignis des Befehlsschalters erstellen

Nun ist die Bedienoberfläche des entstehenden Programms in einem akzeptablen Zustand, und wir können uns um die Funktionalität kümmern. Dazu wird eine Methode benötigt, die bei einem Mausklick auf den Befehlsschalter auszuführen ist. Sie soll …

- 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.

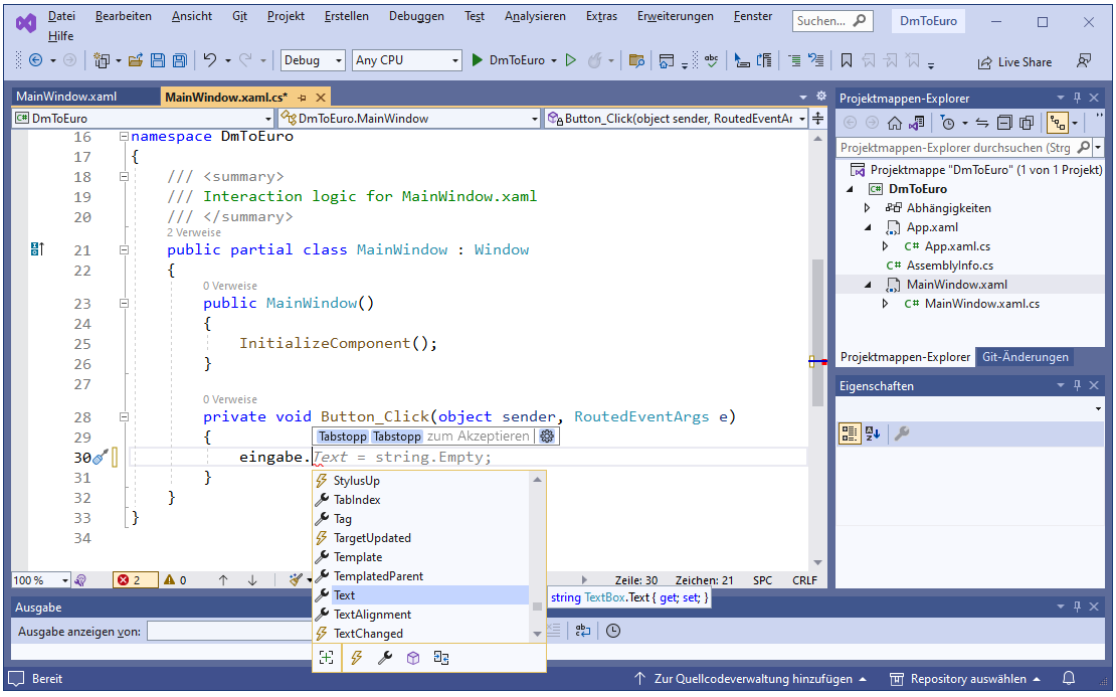
Sobald Sie einen Doppelklick auf das **Button**-Objekt setzen, öffnet das Visual Studio im Quellcode-Editor die partielle Definition der Klasse `MainWindow`, die wir gerade mit Assistentenhilfe definieren, und fügt dort eine Methode namens `Button_Click()` ein, die im fertigen Programm nach jedem Mausklick auf den Schalter ausgeführt wird:[1]

[1] In C# kann die Definition einer Klasse über mehrere Quellcodedateien verteilt werden, sodass mehrere *partielle* Klassendefinitionen entstehen. Das ist vorteilhaft, wenn wir zusammen mit einem Assistenten der Entwicklungsumgebung für eine Klasse verantwortlich sind. Dieses Thema wird später offiziell behandelt.



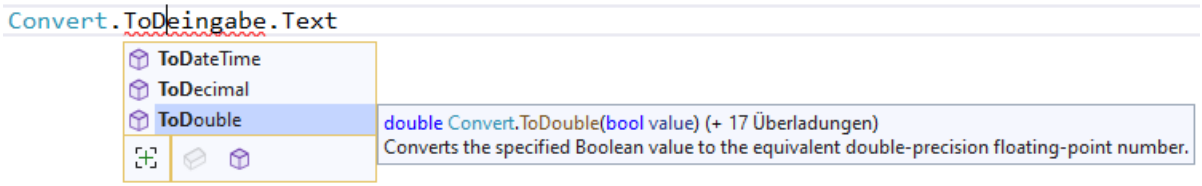
Die Entwicklungsumgebung hat für das Projekt den Namensraum `DmToEuro` definiert, wobei die Bezeichnung mit dem Projektnamen übereinstimmt.

Sobald Sie im Rumpf der Methode `Button_Click()` den Namen des **TextBox**-Objekts eintippen, um sich über seine **Text**-Eigenschaft nach der vom Benutzer eingetippten Zeichenfolge zu erkundigen, und einen Punkt hinter den Feldnamen setzen, bietet die IntelliSense-Technik der Entwicklungsumgebung u. a. die Methoden und Eigenschaften des Objekts zur Fortsetzung der Anweisung an:



Wählen Sie die **Text**-Eigenschaft per Tabulatortaste.

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



Nach einem Doppelklick auf **ToDouble** müssen wir lediglich die runden Klammern um die als Parameterwert zu übergebende **Text**-Eigenschaft ergänzen, um den Methodenaufruf zu komplettieren.

Wie Sie bereits wissen, bietet der Quellcode-Editor unserer Entwicklungsumgebung noch weitere Unterstützungsleistungen an:

- farbliche Unterscheidung verschiedener Sprachbestandteile
- automatische Quellcode-Formatierung (z. B. bei Einrückungen)
- automatische Syntaxprüfung, z. B.:

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

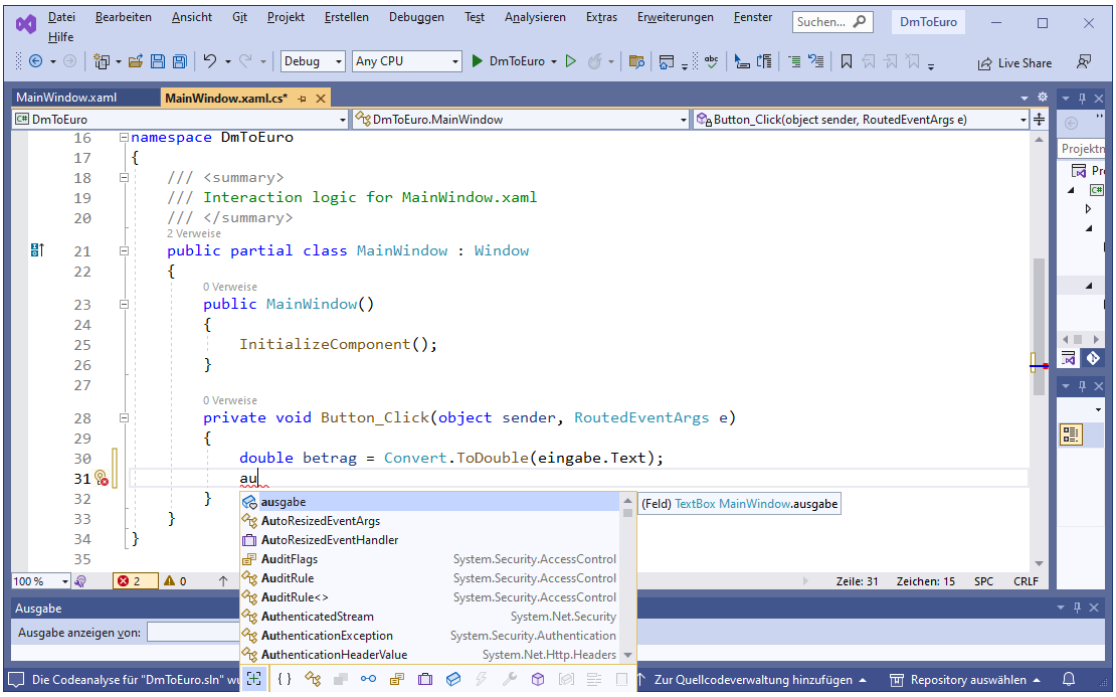
  Wir haben mittlerweile einen Methodenaufruf erstellt, der aber keine vollständige Anweisung ist, was unsere Entwicklungsumgebung durch rotes Unterschlängeln der Fehlerstelle reklamiert. Wir ergänzen das an dieser Stelle erforderliche Semikolon.

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 4.3.4):

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

Weil bislang kein lesender Zugriff auf die lokale Variable `betrag` stattfindet, wird sie durch Punkte unter dem Variablennamen als überflüssig kritisiert.

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 **ToString()** der Klasse **Convert** in eine Zeichenfolge (ein Objekt der Klasse **String**) wandeln. Das Endergebnis wird dem **Label**-Objekt `ausgabe` als neuer Wert der Eigenschaft **Content** übergegeben. Schon nach der Eingabe von zwei Anfangsbuchstaben des Feldnamens schlägt das Visual Studio die intendierte Fortsetzung vor, die wir z. B. per Tabulatortaste übernehmen:



So sieht die fertige Methode `Button_Click()` aus:[1]

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

Im Quellcode ist die Zahl 1,95583 mit einem Dezimal*punkt* zu schreiben.

Bislang wurden Methoden(aufrufe) als Nachrichten in der Kommunikation zwischen Klassen und Objekten dargestellt (siehe z. B. Abschnitt 1.2). Die Methode `Button_Click()` wird jedoch kaum im Quellcode direkt aufgerufen. Es handelt sich um eine sogenannte *Rückrufmethode* (engl.: *callback method*) im Rahmen der Ereignisbehandlung. Sie wird von der CLR aufgerufen, wenn ein Anwender das Klickereignis des Befehlsschalters auslöst.

3.3.7.4 Testen und verbessern

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

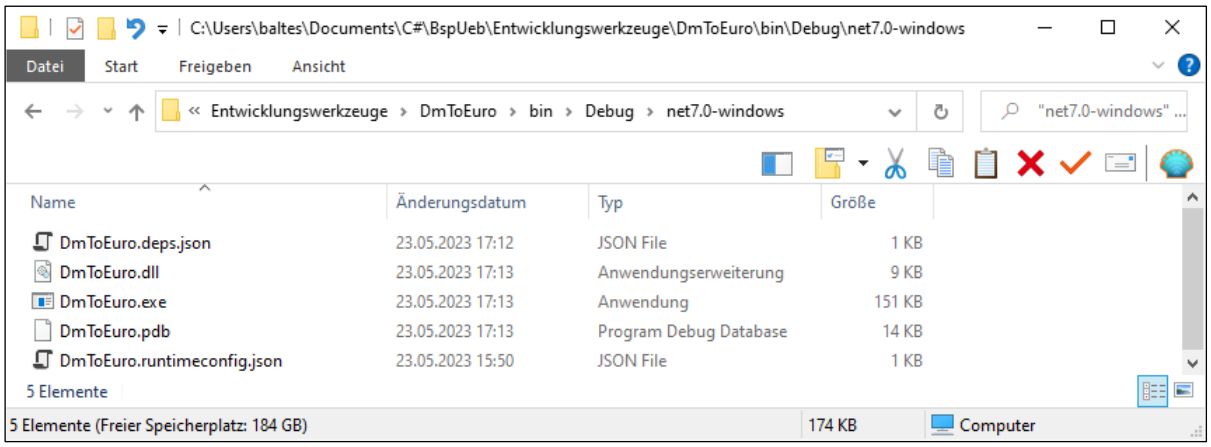
[1] Über den Grund für die farblichen Abweichungen zwischen dem Quellcode in den Bildschirmfotos und dem via Windows-Zwischenablage in den Text übernommenen Quellcode informiert Abschnitt 3.3.11.2.

Die Optik des Programms wird durch den wenig informativen Fenstertitel beeinträchtigt. Markieren Sie bei aktivem WPF-Designer das Hauptfenster, und ändern Sie per **Eigenschaften**-Fenster die **Title**-Eigenschaft (Kategorie **Allgemein**):[1]

In der Endversion des Programms werden wir einen benutzerfreundlich gerundeten Euro-Betrag ausgeben.

3.3.7.5 Erstellte Dateien

Beim Erstellen eines WPF-Programms resultieren fünf Dateien, z. B.:



Das gilt auch für ein Konsolenprojekt (siehe Abschnitt 3.1.4), und außerdem bestehen keine Unterschiede zwischen der Debug- und der Release-Konfiguration (siehe Abschnitt 3.3.8.3).

Auch im Hinblick auf eine Weitergabe von Programmen müssen wir uns mit diesen Dateien genauer beschäftigen, als es z. B. bei den nur temporär (während der Erstellung) relevanten Dateien im **obj**-Projektunterordner der Fall ist. Anschließend werden die bisherigen Informationen zusammengetragen und ergänzt:[2]

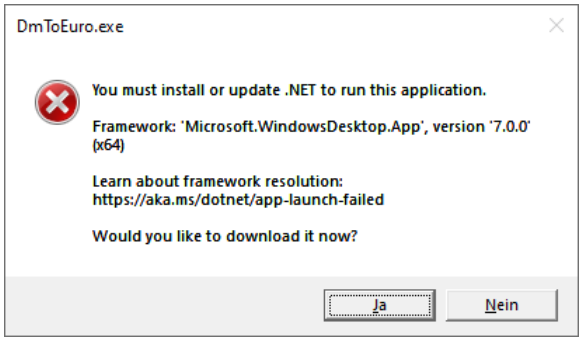
- **DmToEuro.dll**
  Das erstellte Assembly mit dem ausführbaren Programm befindet sich in der Datei **DmToEuro.dll**.
- **DmToEuro.exe**
  Diese Datei ermöglicht den bequemen Assembly-Start durch das Abschicken des Dateinamens in einem Konsolenfenster oder per Doppelklick auf den Dateinamen. Wenn diese Datei fehlt, kann das Assembly per dotnet-CLI trotzdem gestartet werden (siehe Abschnitt 3.1.5).

---

[1] Falls jemand wissen möchte, wie man den rechtsgerichteten Pfeil eintippt: **Alt**-Taste drücken und festhalten, auf dem Ziffernblock der Tastatur nacheinander die Tasten **2** und **6** drücken, **Alt**-Taste loslassen.

[2] Einige Details stammen von: https://natemcmaster.com/blog/2017/12/21/netcore-primitives/

- **DmToEuro.runtimeconfig.json**
  Diese für die Ausführung des Assemblies unverzichtbare Datei informiert darüber, welche Laufzeitumgebung hinsichtlich Typ und Version benötigt wird. Außerdem können Einstellungen zum Verhalten der Laufzeitumgebung vorhanden sein (z. B. zur (De-)aktivierung der mehrstufigen Arbeitsweise des JIT-Compilers, vgl. Abschnitt 2.5.2). Wenn auf einem Kundenrechner die benötigte Version der Laufzeitumgebung fehlt, dann erscheint beim Startversuch die folgende Fehlermeldung:



- **DmToEuro.deps.json**
  Diese Datei informiert über Assemblies außerhalb der Standardbibliothek, von denen das Assembly **DmToEuro** abhängig ist. Solche externen Assemblies stammen oft aus NuGet-Paketen (siehe Abschnitt 3.3.9). Das Beispielprogramm kommt ohne solche Abhängigkeiten aus, sodass die Datei **DmToEuro.deps.json** zum Starten nicht erforderlich ist.
- **DmToEuro.pdb**
  Die **pdb**-Datei (*Program Debug Database*) enthält Informationen, die das Debugging (die Fehlersuche) und das Profiling (die Leistungsoptimierung) unterstützen. Sie ist für die Ausführung des Assemblies nicht erforderlich.

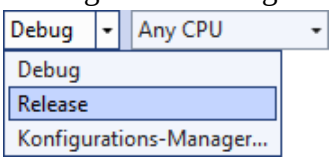
Um das Beispielprogramm auszuführen, genügen also die beiden folgenden Dateien:

- **DmToEuro.dll**
- **DmToEuro.runtimeconfig.json**

3.3.8 Erstellungsoptionen

Aus einer C# - Quellcodedatei lassen sich unterschiedliche Produkte erstellen, die sich in verschiedenen Systemumgebungen bewähren müssen. Während der Entwicklung sind für die Übersetzung in IL-Code andere Kriterien (z. B. gute Testbarkeit) relevant als bei der Erstellung der auszuliefernden Version (z. B. hohe Performanz). Folglich existieren zahlreiche Optionen für den Erstellungsprozess, denen wir an verschiedenen Stellen begegnen (z. B. in der Projektdatei, in Dialogen der Entwicklungsumgebung). Weil die Voreinstellungen gut zum Ziel passen, die Programmiersprache C# zu erlernen, dürfen Sie die Erstellungsoptionen vorläufig ignorieren. Folglich müssen Sie den aktuellen Abschnitt nur bei einem konkreten Informationsbedarf konsultieren, z. B.:

- Sie interessieren sich für die Auswirkungen der Konfigurationsauswahl im Visual Studio.



- Für ein auszuliefernde Programm sollen günstige Erstellungsoptionen gewählt werden.
- Sie wollen für ein Programm eine bestimmte C# - Version vorschreiben.

Im Abschnitt 3.1 haben wir zur Projekterstellung das dotnet-CLI verwendet und dabei einige Erstellungsoptionen kennengelernt, z. B.:

- Ausgabetyp (z. B. **Exe** oder **Library**),
- vorausgesetzte .NET - Version (**TargetFramework**)
- Name des resultierenden Assemblies

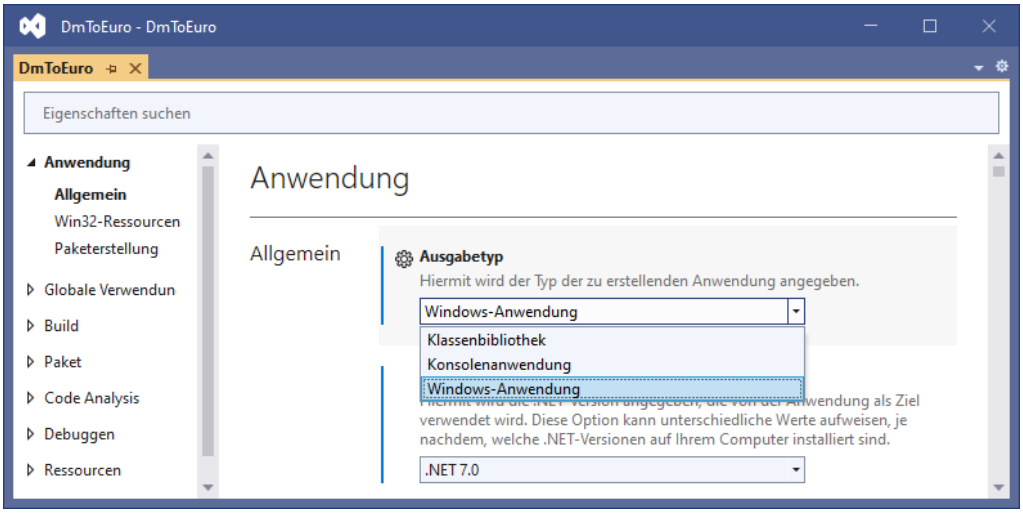
Diese Einstellungen landen in der Projektdatei mit der Namenserweiterung **csproj**, z. B.:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net7.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

</Project>
```

Man kann diese Datei im Visual Studio per Doppelklick auf den Projektnamen im **Projektmappen-Explorer** öffnen und editieren. Etwas bequemer lassen sich die Erstellungsoptionen über den Projekteigenschaftendialog ändern, der z. B. über den folgenden Menübefehl

Projekt > DmToEuro-Eigenschaften

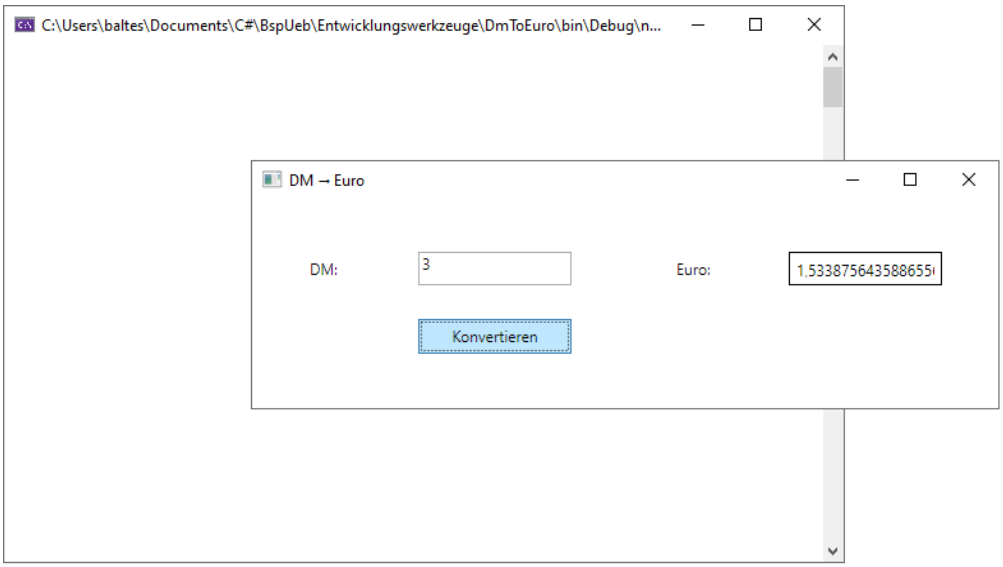
oder über das Item **Eigenschaften** im Kontextmenü zum Projekt im **Projektmappen-Explorer** erreichbar ist. Hier kann man z. B. den Ausgabetyp einstellen, ohne die korrekten Namen der Optionen (**Exe**, **WinExe**, **Library**) kennen zu müssen:



In diesem Dialog vorgenommene Änderungen wirken sich sofort auf die Projektdatei aus und werden somit an das MSBuild-System sowie an den C# - Compiler weitergegeben.

3.3.8.1 Ausgabetyp, Assembly-Name, Zielframework und WPF-Unterstützung

Im Beispiel aus dem Abschnitt 3.3.7 erscheint neben dem Anwendungsfenster mit den GUI-Bedienelementen zusätzlich ein Konsolenfenster,



wenn in der Projektdatei **DmToEuro.csproj** der Ausgabetyp (**OutputType**) **Exe** eingetragen ist:

```
<OutputType>Exe</OutputType>
```

Um den Konsolenauftritt zu verhindern, trägt man (z. B. mit Hilfe des Projekteigenschaftendialogs, siehe oben) den Ausgabetyp **WinExe** ein:

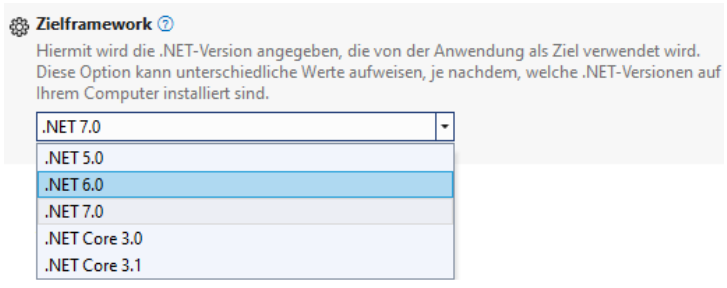
```
<OutputType>WinExe</OutputType>
```

Bei einem GUI-Programm ist ein Konsolenfenster nicht grundsätzlich überflüssig, weil hier z. B. Laufzeitfehlermeldungen erscheinen:

Außerdem kann ein Programm neben seinen GUI-Aktivitäten auch Konsolenausgaben vornehmen (z. B. über **WriteLine()** - Aufrufe). Ohne Konsolenfenster bleiben Laufzeitfehlermeldungen mit ihren diagnostischen Informationen (z. B. zur Fehlerstelle in der Quellcodedatei) sowie **WriteLine()** – Ausgaben im Verborgenen.

Im **Anwendungs**-Bereich des Projekteigenschaftsdialogs lassen sich neben dem **Ausgabetyp** noch weitere Erstellungsoptionen beeinflussen, z. B.:

- **Zielframework**
  Welche .NET – Version soll zum Erstellen verwendet und damit auch minimal auf einem zu versorgenden Rechner vorausgesetzt werden? Mit der Version wachsen die Leistungsumfänge der Programmiersprache C# und der BCL sowie die Anforderungen an einen Einsatzrechner, falls die Laufzeitumgebung nicht zusammen mit dem Programm ausgeliefert wird. Wenn eine gewünschte Version in der angebotenen Liste



  fehlt, dann hilft z. B. eine Ergänzungsinstallation über den **Visual Studio Installer** (siehe Abschnitt 3.3.6). Die freie Wahl der .NET – Version kann durch die Anwesenheit der Datei

global.json im Projektordner oder einem übergeordneten Ordner eingeschränkt sein (vgl. Abschnitt 3.1.2).

- **Windows Presentation Foundation**
  Wird ein Projekt unter Verwendung einer WPF-Vorlage angelegt, dann …
  - ist das Kontrollkästchen zur **Windows Presentation Foundation** markiert,
  - erscheint in der Projektdatei das folgende **PropertyGroup**-Element
    ```
    <UseWPF>true</UseWPF>
    ```
  - und die erforderlichen Bibliotheken werden eingebunden.
- **Assemblyname**
  Per Voreinstellung übernimmt das erstellte Assembly seinen Namen vom Projekt.

Der Projekteigenschaftsdialog enthält neben den Erstellungsoptionen etliche weitere Einstellungen, die wir bei passender Gelegenheit behandelt werden, z. B.:

- **Standardnamespace**
  Für Projekte mit vielen Typdefinitionen ist ein eigener Namensraum sinnvoll. Von der Entwicklungsumgebung angelegte Typdefinitionen verwenden den Eintrag im Feld **Standardnamespace**, der initial mit dem Projektnamen identisch ist. Eine Änderung wirkt sich nur auf *neue* Quellcodedateien aus.
- **Symbol**
  Zur Verwendung unter Windows kann eine ICO-Datei mit dem Programmsymbol gewählt werden.

3.3.8.2 C# - Sprachversion

Bis zur Version 2017 unserer Entwicklungsumgebung konnte man die in einem Programm zu verwendende C# - Version im Projekteigenschaftendialog einsehen und ändern. Seit der Version 2019 wird die C# - Version nur noch angezeigt,

und auf der verlinkten Webseite

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version

wird das aktuelle Verfahren zur Wahl der C# - Version für ein Projekt erläutert:

- Der Compiler verwendet aufgrund des im Projekt eingestellten Zielframeworks eine voreingestellte C# - Version:

Zielframework	Version	Voreingestellte C# - Version
.NET	7.x	11
.NET	6.x	10
.NET	5.x	9.0
.NET Core	3.x	8.0
.NET Core	2.x	7.3
.NET Standard	2.1	8.0
.NET Standard	2.0	7.3
.NET Framework	alle	7.3

- Über die Projektdatei kann die Voreinstellung des Compilers dominiert werden. Wird z. B. per **LangVersion**-Element in der Projektdatei die Sprachversion 8 eingestellt,

```
<LangVersion>8.0</LangVersion>
```

  dann ist die implizite Definition einer Startklasse mit **Main()** – Methode nicht mehr möglich:

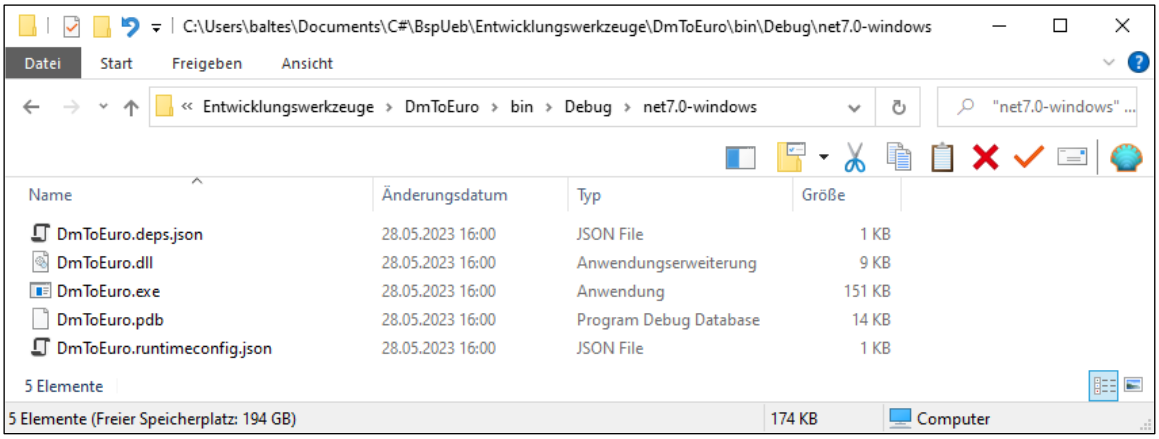
  > Das Feature "Anweisungen der obersten Ebene" ist in C# 8.0 nicht verfügbar. Verwenden Sie Sprachversion 9.0 oder höher.

Die automatische Ableitung einer voreingestellten C# - Version aus dem Zielframework ist plausibel, und wir werden nichts dagegen unternehmen.

3.3.8.3 Debug- und Release-Konfiguration

Bei der Programmerstellung hängt das Verhalten des C# - Compilers von der Projektkonfiguration ab. Per Voreinstellung ist die **Debug**-Konfiguration aktiv, sodass der Compiler …

- auf Optimierungen verzichtet,
- zur Unterstützung der Fehlersuche eine umfangreiche **pdb**-Datei (*Program Debug Database*) erstellt
- und die Ergebnisse seiner Arbeit im Projektunterordner **…\bin\Debug** ablegt, z. B.:

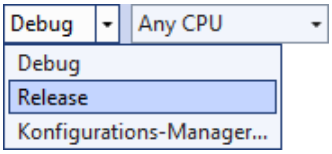


Zu den vom Compiler konfigurationsabhängig eingesetzten oder unterlassenen Optimierungsmaßnahmen gehört z. B. das sogenannte *Inlining*. Bei kurzen Methoden lohnt es sich, die mit einem Aufruf verbundenen Kosten einzusparen, indem der IL-Code in die aufrufenden Methoden integriert wird. Solche Optimierungsmaßnahmen steigern die Performanz, erschweren aber die

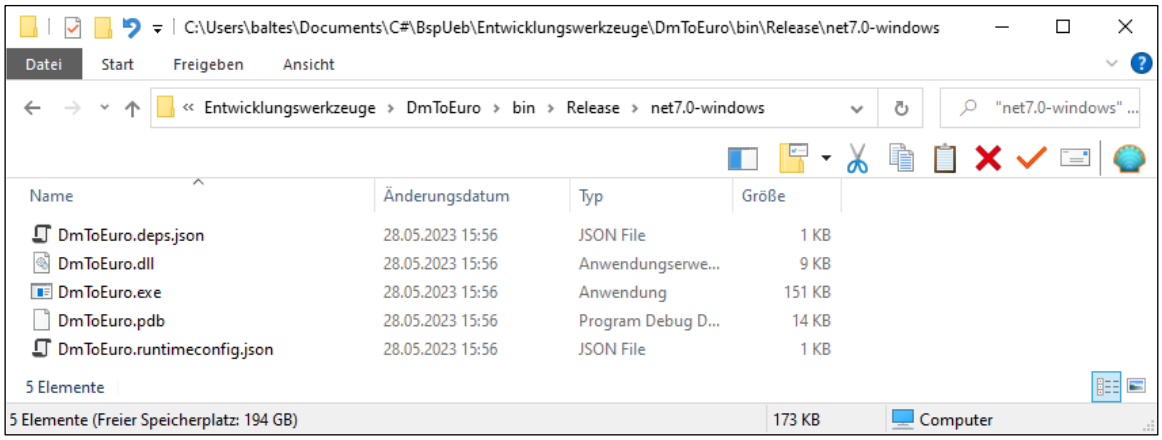
Fehlersuche (engl.: das *debugging*) durch die komplexere Beziehung zwischen dem Quellcode und dem IL-Code.

In der **pdb**-Datei (*Program Debug Database*) stecken Informationen, die bei der Fehlersuche und bei der Leistungsoptimierung (engl.: beim *profiling*) benötigt werden. Zwar findet die Fehlersuche hauptsächlich mit der Debug-Konfiguration statt, doch kommt auch mit der Release-Konfiguration (z. B. auf einem Kundenrechner) eine Fehlersuche in Betracht. Bei der Leistungsoptimierung ist die Release-Konfiguration der attraktivere Anwendungsfall. Insgesamt gibt es also gute Gründe dafür, für *beide* Konfigurationen eine **pdb**-Datei zu erstellen. Per Voreinstellung fällt diese Datei bei der Debug-Konfiguration etwas umfangreicher aus.

Zur Änderung der Konfiguration steht in der Visual Studio - Symbolleiste ein Bedienelement bereit:



Ist die **Release**-Konfiguration aktiv, dann erstellt der Compiler ein leistungsoptimiertes, aber weniger gut testbares Assembly und legt es im Projektunterordner **…\bin\Release** ab, z. B.:



Im Beispiel unterscheiden sich die Ausgabeordner der beiden Konfigurationen nur wenig:

- Das Assembly der Release-Konfiguration ist kleiner (8704 statt 9216 Bytes).
- Die **pdb**-Datei der Release-Konfiguration ist unwesentlich kleiner (13592 versus 13684 Bytes).

Das Assembly **DmToEuro.dll** und der Starthelfer **DmToEuro.exe** aus beiden Konfigurationen sind auf jedem aktuellen Windows-Rechner mit installiertem .NET 7 ausführbar, wobei die Einschränkung auf Windows aus der Nutzung der WPF-Bibliothek resultiert. In diesem Sinn sind die *beiden* Ausgabeordner auslieferungsbereit. Wegen der besseren Performanz sollten Kunden aber in der Regel den Ausgabeordner der Release-Konfiguration erhalten. Die **pdb**-Datei wird auf einem Kundenrechner nur dann benötigt, wenn dort nach der Ursache für einen Fehler gesucht werden soll.

3.3.8.4 Zielplattform

Wird im Visual Studio 2022 für ein neues Projekt in einer eigenen Projektmappe als Vorlage entweder **Konsolen-App** oder **WPF-Anwendung** verwendet, dann ist **Any CPU** als **Zielplattform** voreingestellt. Bei

- (**Windows-**)**Anwendung** als **Ausgabetyp**,
- .NET 7.0 als **Zielframework**
- und Windows als **Zielbetriebssystem**

werden dann auf einem Entwicklungsrechner mit 64-Bit - Windows als Betriebssystem (bei aktiver **Debug**- oder **Release**-Konfiguration) von MSBuild erstellt:

- ein Assembly mit IL-Code in einer Datei mit der Namenserweiterung **dll**
- ein **Win-64** - Programm in einer Datei mit der Namenserweiterung **exe**, das den Start vereinfacht (durch Abschicken des Dateinamens in einem Konsolenfenster oder per Doppelklick auf den Dateinamen)

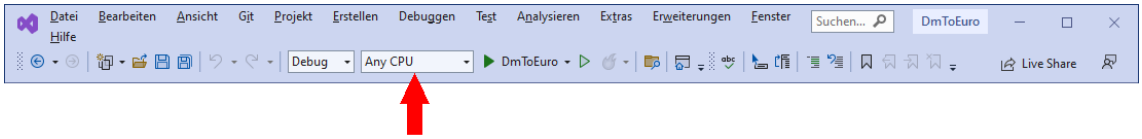
Man kann die **Zielplattform** zu einem Projekt über

Projekt > Eigenschaften > Build > Allgemein

einsehen und modifizieren, z. B.:

Wird **x86** als **Zielplattform** gewählt, dann entsteht unter den eben beschriebenen Voraussetzungen neben dem IL-Assembly ein **Win-32** – Startprogramm, das auf jedem Windows-Rechner läuft.

Eine Projektmappe kann *mehrere* Projekte enthalten, was im Manuskript keine nennenswerte Rolle spielt, bei komplexen Aufgabenstellungen aber schon. Sind mehrere Projekte vorhanden, dann sollen in der Debug- und/oder Release-Konfiguration eventuell bei der Programmerstellung nicht alle Projekte identisch behandelt werden. Um derartige Erstellungskonfigurationen unterstützen zu können, besitzt das Visual Studio das Konzept der **Projektmappenplattform** und zeigt die aktuelle Einstellung unter der Menüzeile an, z. B.:



In der zu Beginn des Abschnitts beschriebenen Situation ist **Any CPU** die einzige definierte **Projektmappenplattform**, und alle in der Mappe enthaltenen Projekte besitzen **Any CPU** als **Zielplattform**. Wir werden im Manuskript keinen Anlass finden, diese Voreinstellung zu ändern.

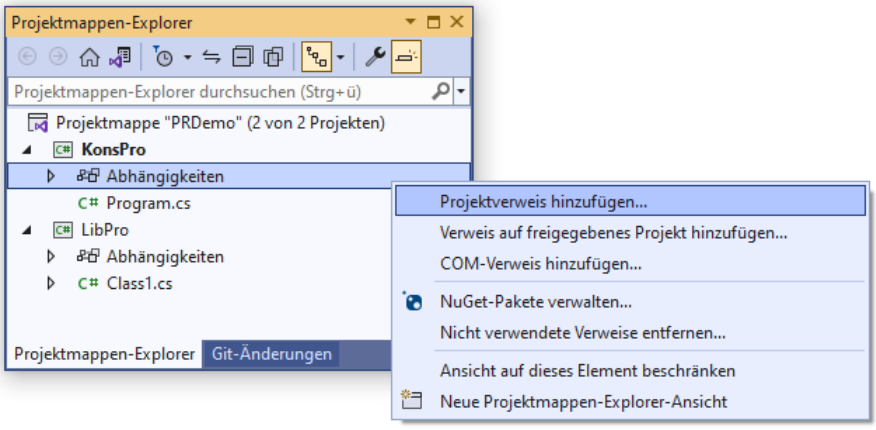
3.3.9 Projektverweise und NuGet-Pakete

Fast jedes Assembly ist von anderen Assemblies abhängig, weil es dort implementierte Klassen und/oder andere Typen benutzt. Es muss dafür gesorgt werden, dass die vorausgesetzten Assemblies zur Entwicklungs- und zur Laufzeit gefunden werden. Bei den Assemblies in der BCL ist das sichergestellt. Die Aufnahme von Verweisen auf zusätzliche Bibliotheken wird früher oder später relevant, doch muss man sich nicht unbedingt in einer frühen Phase des Einstiegs in die Programmierung damit beschäftigen. Somit kann man die Lektüre des aktuellen Abschnitts bis zum ersten Anwendungsfall aufschieben.

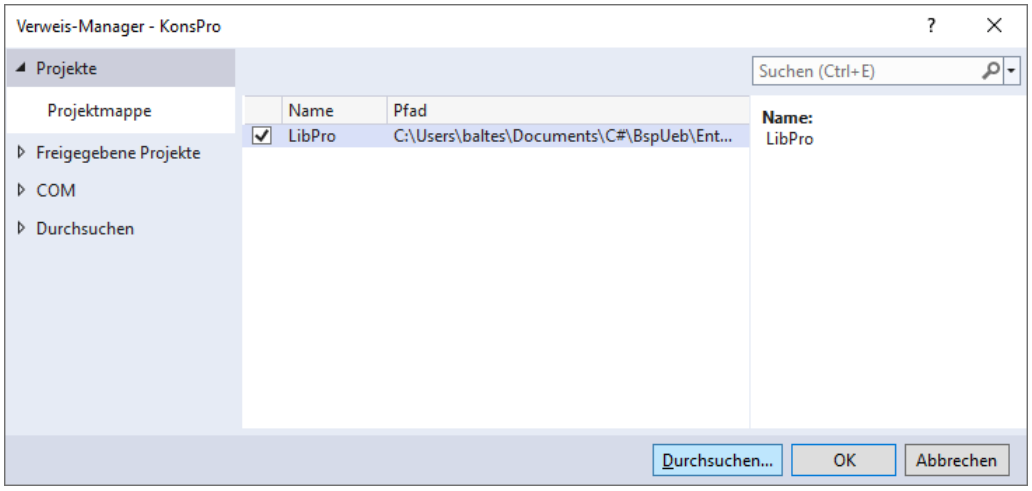
Anschließend wird die Verwaltung von Projektverweisen und NuGet-Paketen im Visual Studio beschrieben, wo hilfreiche Assistenten verfügbar sind. Es werden aber auch Hinweise zur Verwendung im VS Code gegeben (Projektdatei editieren und das dotnet-CLI nutzen).

3.3.9.1 Projektverweise

Um das Assembly eines anderen Projekts aus derselben Projektmappe als Bibliothek verwenden zu können, sollte ein sogenannter **Projektverweis** (engl.: *project reference*) eingefügt werden. Dazu wählt man im **Projektmappen-Explorer** aus dem Kontextmenü zum Projektknoten mit den **Abhängigkeiten** das Item **Projektverweis hinzufügen**, z. B.:



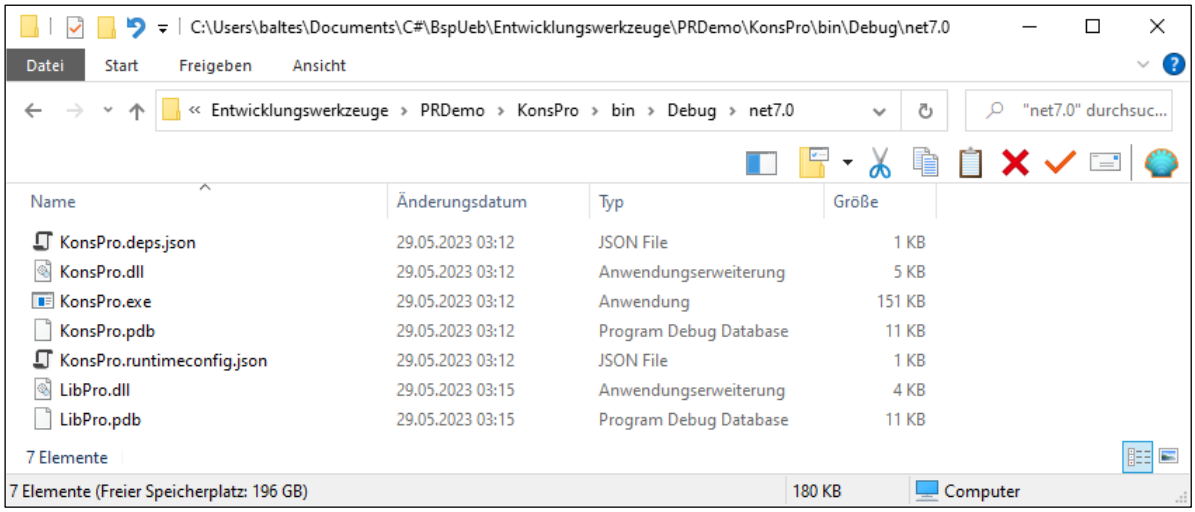
Im **Verweis-Manager** wird das vorausgesetzte Projekt markiert:



Nach dem Quittieren des **Verweis-Manger** – Eintrags erscheint in der Projektdatei ein **ProjectReference**-Element:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\LibPro\LibPro.csproj" />
  </ItemGroup>

</Project>
```

Das im Beispiel vorausgesetzte Bibliotheks-Assembly aus dem Projekt `LibPro` taucht nach der Erstellung im Ausgabeordner des nutzenden Projekts `KonsPro` auf:



Auf diese Weise ist sichergestellt, dass bei der Ausführung des Assemblies **KonsPro.dll** die aus dem Bibliotheks-Assembly **LibPro.dll** benötigten Klassen und sonstigen Typen geladen werden können.

Ein Projektmappenordner mit dem Bibliotheks-Projekt `LibPro` und dem Anwendungsprojekt `KonsPro` ist hier zu finden:

…\BspUeb\Entwicklungswerkzeuge\Verweise und Abhängigkeiten\PRDemo

Wenn ein in die Verweisliste aufzunehmendes Assembly nur als **dll**- oder **exe**-Datei, aber nicht als Projekt vorhanden ist, dann muss man sich mit einem sogenannten *Dateiverweis* behelfen. Dazu wird per **Verweis-Manager** (siehe oben) nach einem Klick auf den Schalter **Durchsuchen** der Pfad zum Assembly bekanntgegeben.

3.3.9.2 NuGet-Pakete

Um die Erstellung, Verteilung und Nutzung von Paketen mit .NET - Bibliotheks-Assemblies zu unterstützen, hat Microsoft unter dem Namen **NuGet** …

- eine Cloud-Infrastruktur,
- eine Reihe von Werkzeugen mit guter Integration in den .NET - Erstellungsprozess per dotnet-CLI oder Visual Studio
- und Spezifikationen für alle beteiligten Aufgaben

geschaffen.[1] Frei verfügbare Pakete werden auf dem öffentlichen Server

https://www.nuget.org/

angeboten. Die NuGet-Spezifikation lässt aber auch private Server zu.

Wir nutzen in einem Beispiel das sehr populäre NuGet-Paket **Newtonsoft.Json** (auch bekannt unter dem Namen *Json.NET*) mit Typen zur Unterstützung des *JSON*-Formats (*JavaScript Object Notation*) bei der (De-)Serialisierung von Objekten.[2] Das Serialisieren kann z. B. zum Speichern von Objekten in Dateien verwendet werden (siehe Abschnitt 16.5 in Baltes-Götz 2021). Das Paket **Newtonsoft.Json** nimmt (am 25.11.2023) in der Hitliste der beliebtesten NuGet-Pakete den ersten Platz ein mit großem Abstand vor dem Zweitplatzierten.[3]

---

1 https://learn.microsoft.com/en-us/nuget/what-is-nuget

2 https://learn.microsoft.com/en-us/nuget/quickstart/install-and-use-a-package-in-visual-studio

3 https://www.nuget.org/stats/packages

Im weiteren Verlauf des Abschnitts werden die Installation und die Nutzung eines NuGet-Pakets in einem Visual Studio - Projekt vorgeführt. Der Projektordner des Beispiels ist hier zu finden:

…\BspUeb\Entwicklungswerkzeuge\Verweise und Abhängigkeiten\NuGetJson

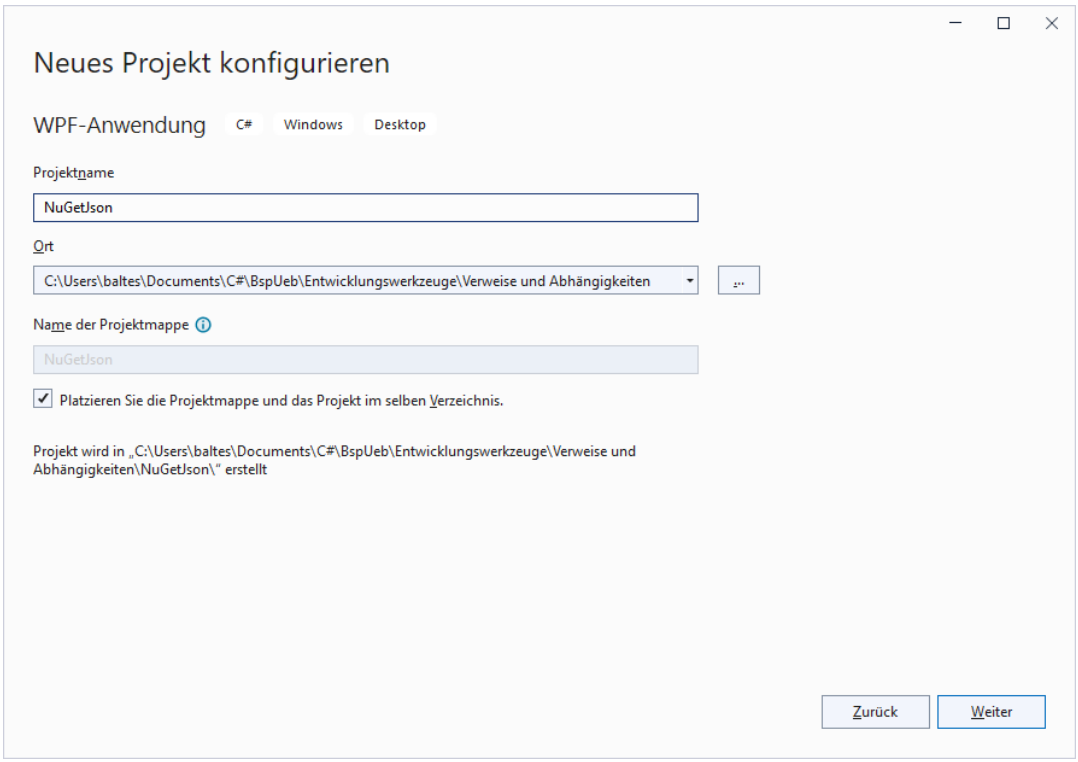
Wer das VS Code bevorzugt, kann zur Installation von NuGet-Paketen das Kommando **dotnet add package** verwenden, z. B.:[1]

```
>dotnet add package Newtonsoft.Json
```

Das Kommando ist auf ein Projekt anzuwenden, also in einem Konsolenfenster aufzurufen, das auf den Projektordner positioniert ist. In das VS Code ist ein Terminal integriert, sodass diese Bedingung leicht zu erfüllen ist (siehe Abschnitt 3.5.4).

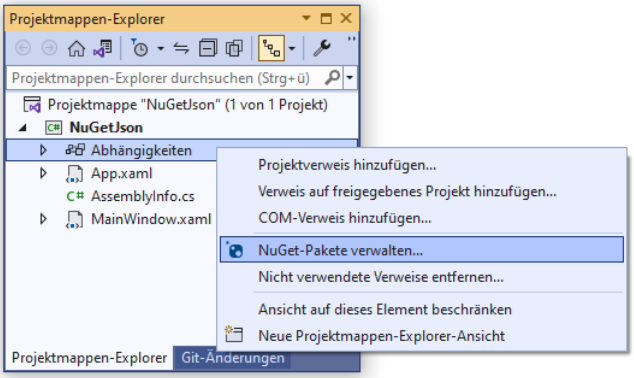
3.3.9.2.1 Beispielprojekt mit NuGet-Paketinstallation

Wie im Abschnitt 3.3.7 nutzen wir die C# - Projektvorlage **WPF-Anwendung**, um ein Programm mit grafischer Bedienoberfläche in WPF-Technik (Windows Presentation Foundation) zu erstellen:



Wir wählen im **Projektmappen-Explorer** aus dem Kontextmenü zu den **Abhängigkeiten** des Projekts das Item **NuGet-Pakete verwalten**:

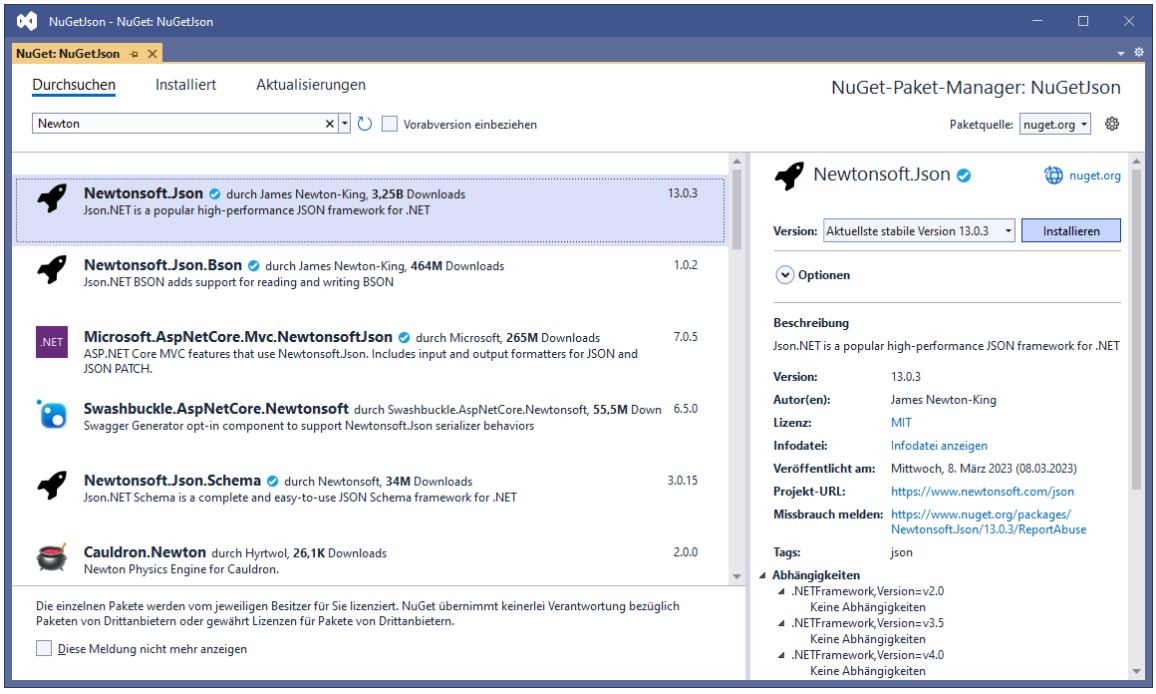
[1] https://learn.microsoft.com/en-us/nuget/consume-packages/install-use-packages-dotnet-cli



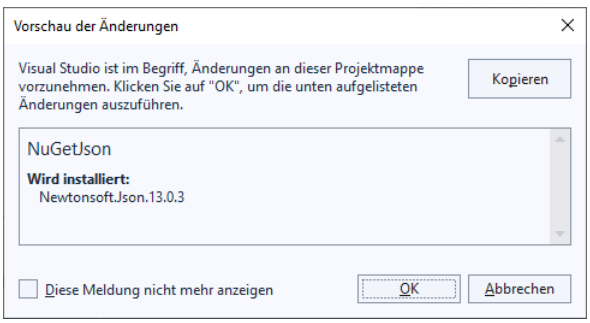
Daraufhin erscheint der **NuGet-Paket-Manager**, der folgende Leistungen anbietet:

- die **Paketquelle** ändern (Voreinstellung: https://www.nuget.org/)
- die **Paketquelle** nach Paketen **durchsuchen**
- die im Projekt **installierten** Pakete verwalten
- die Pakete des Projekts **aktualisieren**

Mit Hilfe des Suchfelds ist das benötigte Paket schnell gefunden:

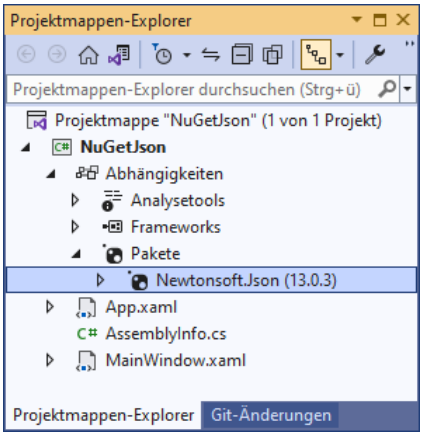


Wir veranlassen das **Installieren** der stabilen Version 13.0.3 und bestätigen per Klick auf **OK**:



In der Liste der mit **Newtonsoft.Json** 13.0.3 kompatiblen .NET - Versionen taucht .NET 7.0 nicht auf, aber die Vorgängerversion .NET 6.0 (mit Langzeitunterstützung). Außerdem wird die Version 2.0 von .NET Standard genannt, und dieser Standard wird von .NET 7.0 erfüllt (siehe Abschnitt 2.2).

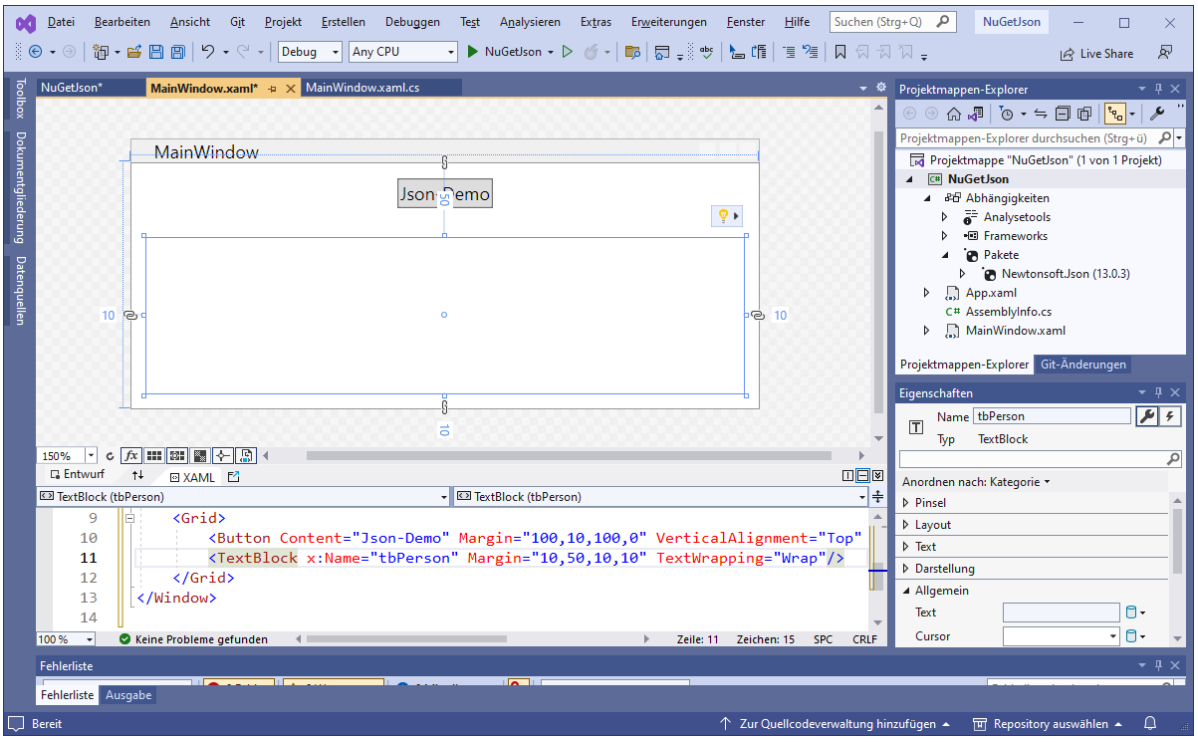
Im **Projektmappen-Explorer** wird anschließend die neue **Abhängigkeit** angezeigt:



In der Projektdatei **NuGetJson.csproj** befindet sich nach dem Speichern ein **PackageReference**-Element, das die Abhängigkeit des Projekts vom Paket **Newtonsoft.Json** 13.0.3 dokumentiert:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net7.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="13.0.3" />
  </ItemGroup>

</Project>
```

Wir setzen per WPF-Designer mit Hilfe der **Toolbox** ein unbenanntes **Button**-Objekt und ein **TextBlock**-Objekt mit dem Namen `tbPerson` auf das Anwendungsfenster (siehe Abschnitt 3.3.7.2):

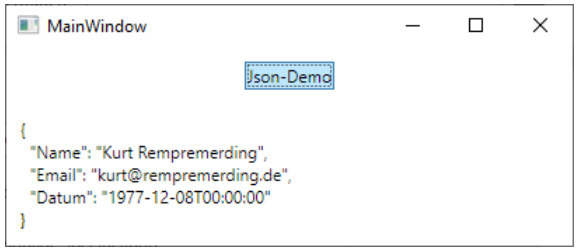


Analog zum Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.** lassen wir per D oppelklick auf den Schalter eine Behandlungsmethode zu seinem **Click**-Ereignis erstellen, die vom Visual Studio den Namen `Button_Click()` erhält. In der Quellcodedatei **MainWindow.xaml.cs** definieren wir (zusätzlich zur Klasse `MainWindow`) eine rudimentäre Klasse namens `Person`, wobei der Einfachheit halber auf die Datenkapselung verzichtet wird.[1]

```
using System;
using System.Windows;
using Newtonsoft.Json;

namespace NuGetJson;

public class Person {
    public string Name;
    public string Email;
    public DateTime Datum;
}

public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
    }

    private void Button_Click(object sender, RoutedEventArgs e) {
        Person person = new Person();
        person.Name = "Kurt Rempremerding";
        person.Email = "kurt@rempremerding.de";
        person.Datum = new DateTime(1977, 12, 8);
        tbPerson.Text = JsonConvert.SerializeObject(person, Formatting.Indented);
    }
}
```

In der Methode `Button_Click()` wird per **new**-Operator ein Objekt der Klasse `Person` erstellt. Aus seinen Merkmalsausprägungen entsteht mit Hilfe der statischen Methode **SerializeObject()** der Klasse **JsonConvert** aus dem Namensraum **Newtonsoft.Json** eine Zeichenfolge im JSON-Format, die anschließend an die **Text**-Eigenschaft des **TextBlock**-Objekts übergeben wird.

Wir lassen das Programm über die Tastenkombination **Strg+F5** erstellen und ausführen. Nach einem Kick auf den Schalter erscheint das Ergebnis der Objektserialisierung im JSON-Format:



3.3.9.2.2 Ablage der NuGet-Pakete im Benutzerprofil

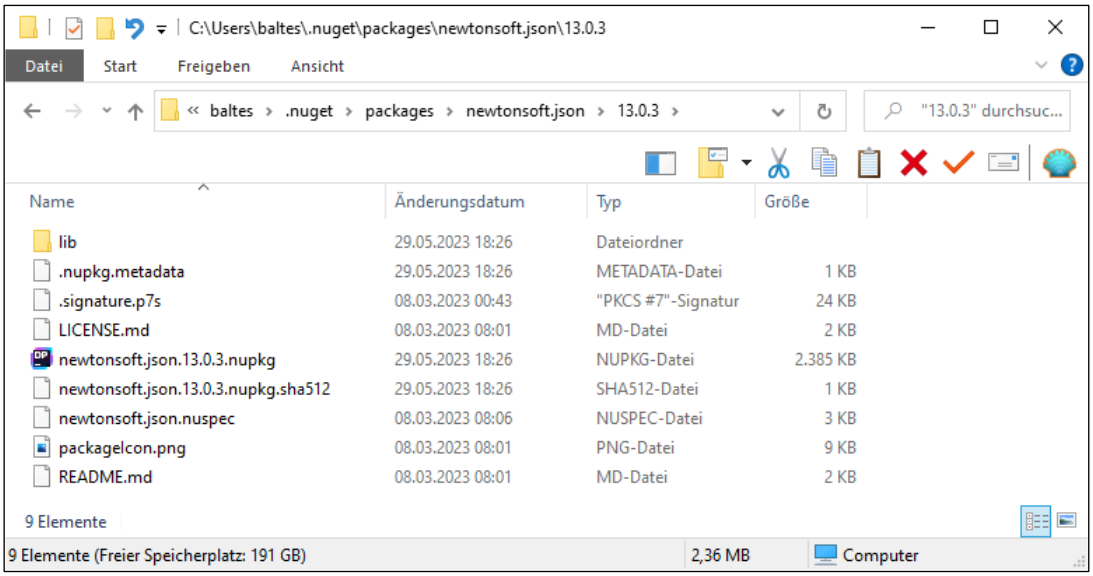
Bei der Installation des NuGet-Pakets **Newtonsoft.Json** im Beispielprojekt hat der **NuGet-Paket-Manager** …

- eine benutzer-bezogene Installation vorgenommen
- und ein **PackageReference**-Element in die Projektdatei eingetragen.

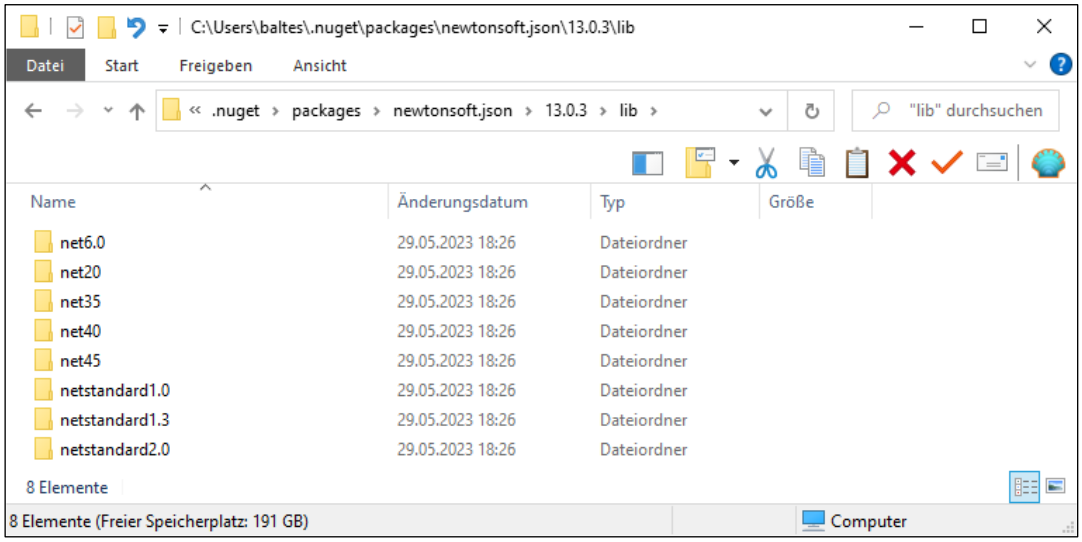
---

[1] Wir erlauben uns diese Nachlässigkeit, damit das Beispiel einfach bleibt und möglichst wenige Vorgriffe auf spätere Kapitel benötigt.

Die NuGet-Pakete landen unter Windows im benutzereigenen Ordner **.nuget**, z. B.:



Das bezogene ZIP-Archiv **newtonsoft.json.13.0.3.nupkg** wurde ausgepackt, und im Unterordner **lib** befindet sich das Bibliotheks-Assembly **Newtonsoft.Json.dll** in Varianten für diverse .NET - Implementationen bzw. -Versionen:



3.3.9.2.3 Versionierung von NuGet-Paketen

Zur Versionierung von NuGet-Paketen schlägt Microsoft das folgende Schema vor:[1]

*Major***.***Minor***.***Patch*[-*Suffix*]

Die zur Beschreibung der Syntax verwendeten metasprachlichen Regeln sind aus dem Abschnitt 2.6.1 bekannt:

- Platzhalter sind an kursiver Schrift zu erkennen, statische Elemente an fetter Schrift. Im Beispiel treten der Punkt und der Bindestrich als statische Elemente auf.
- Die eckigen Klammern dienen als Metazeichen, gehören also *nicht* zur Namensraumbezeichnung. Sie begrenzen optionale Elemente.

---

[1] https://learn.microsoft.com/en-us/nuget/concepts/package-versioning

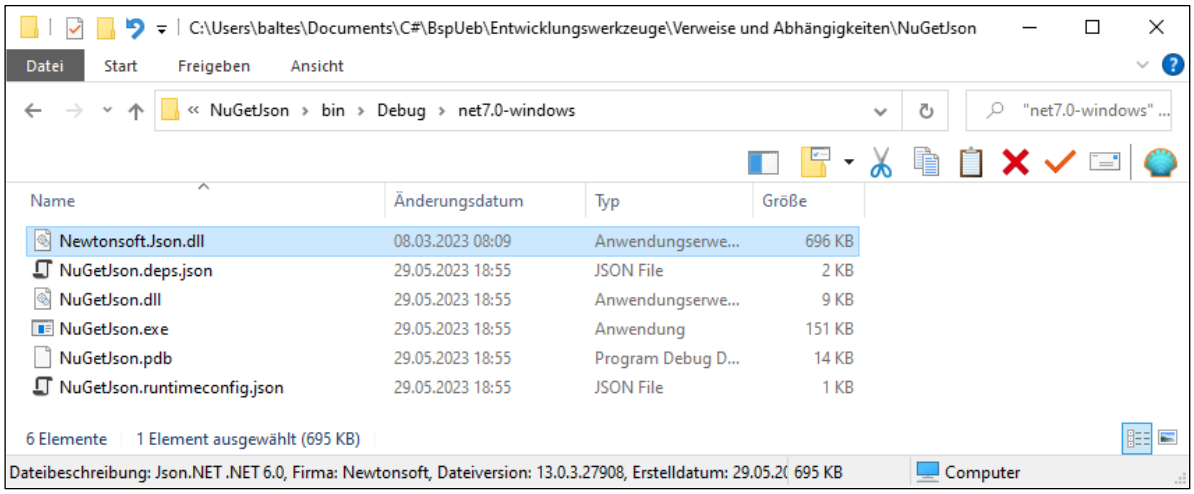
Beispiele:

- 13.0.3
- 3.1.4-beta

Das NuGet-Paket **Newtonsoft.Json** 13.0.3 enthält das Assembly **Newtonsoft.Json** in der Version 13.0.0.0. Wie von Microsoft empfohlen, stimmen die Major-Bestandteile der NuGet-Paketversion und der Assembly-Version (vgl. Abschnitt 2.4.2.2) überein.[1]

3.3.9.2.4 Übernahme der NuGet-Assemblies in den Erstellungs-Ausgabeordner

In den Erstellungs-Ausgabeordner des Beispielprojekts wurde die Variante für .NET 6.0 kopiert, sodass ein startfähiges (framework-abhängiges) Programm vorliegt:



Fehlt das NuGet-Paket beim Öffnen des Projekts, dann wird es vom Visual Studio automatisch wiederhergestellt. Das Restaurieren (engl.: *restoring*) kann auch explizit angefordert werden über das Item

NuGet-Pakete wiederherstellen

aus dem Kontextmenü zur Projektmappe.

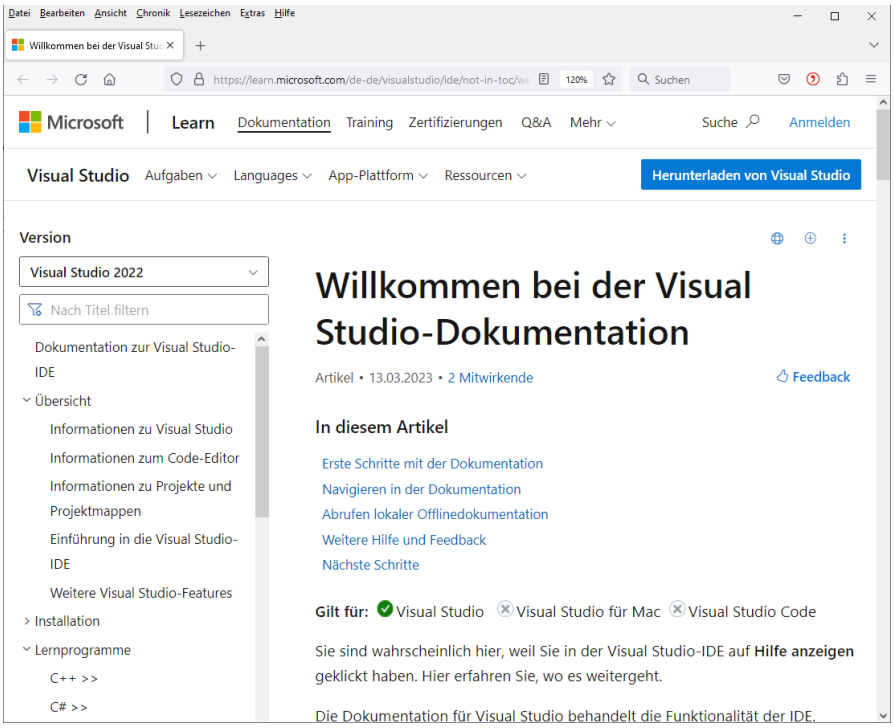
3.3.10 BCL-Dokumentation und andere Hilfeinhalte

Die über den Menübefehl

Hilfe > Hilfe anzeigen

angeforderten Informationen werden per Voreinstellung aus dem Internet bezogen und vom Standard-Browser angezeigt, z. B.:

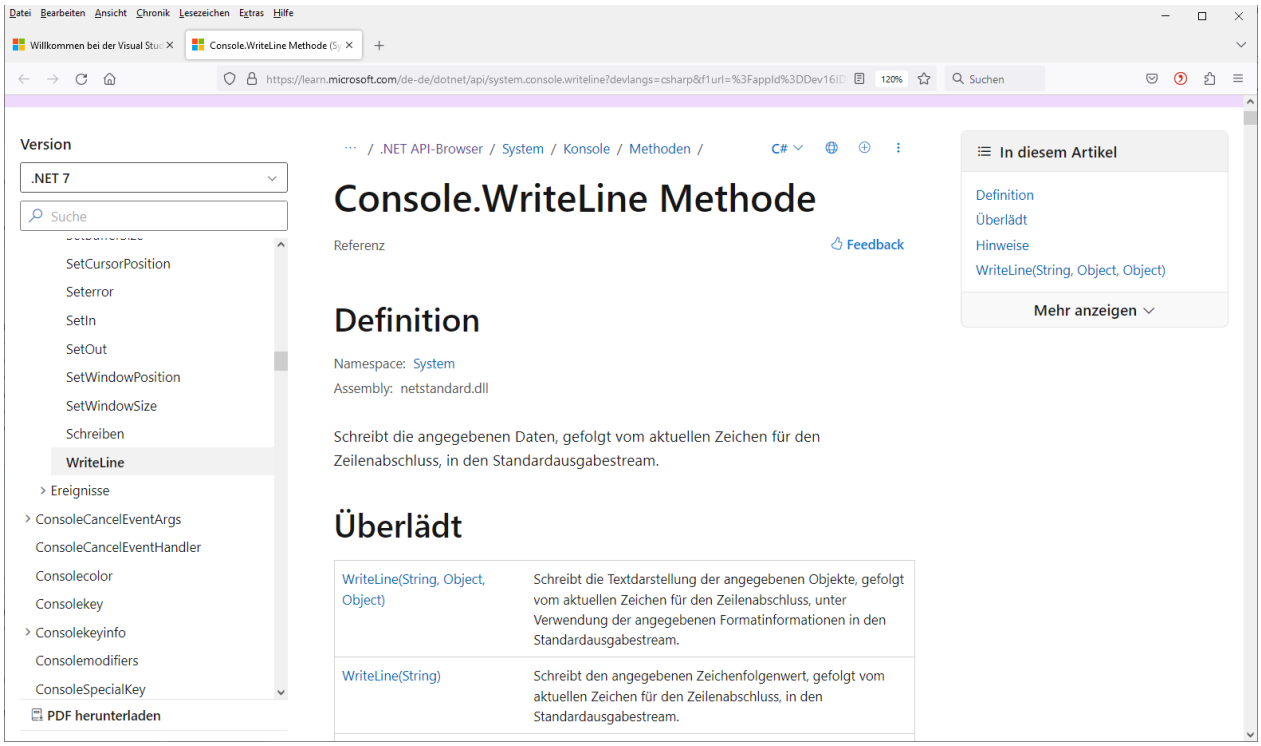
[1] https://learn.microsoft.com/en-us/dotnet/standard/library-guidance/versioning



Wir benötigen die Hilfefunktion vor allem dazu, um Informationen über C# und über das API von BCL-Typen einzuholen. Zu einem im Quellcode-Editor markierten (die Einfügemarke enthaltenden) C# - Schlüsselwort oder BCL-Bezeichner erreicht man die zugehörige Dokumentation bequem über die Funktionstaste **F1**. In der folgenden Situation

```
static void Main(string[] args)
{
    Console.WriteLine("Hello, World!");
    Console.ReadLine();
}
```

erhält man über einen **F1**-Tastendruck ausführliche Informationen zur Methode **WriteLine()** der Klasse **Console** in einem Fenster des Standard-Browsers:



Man erfährt u. a., dass …

- sich die Klasse **Console** im Namensraum **System** befindet
- und im Assembly **netstandard.dll** implementiert wird.

3.3.11 Einstellungen

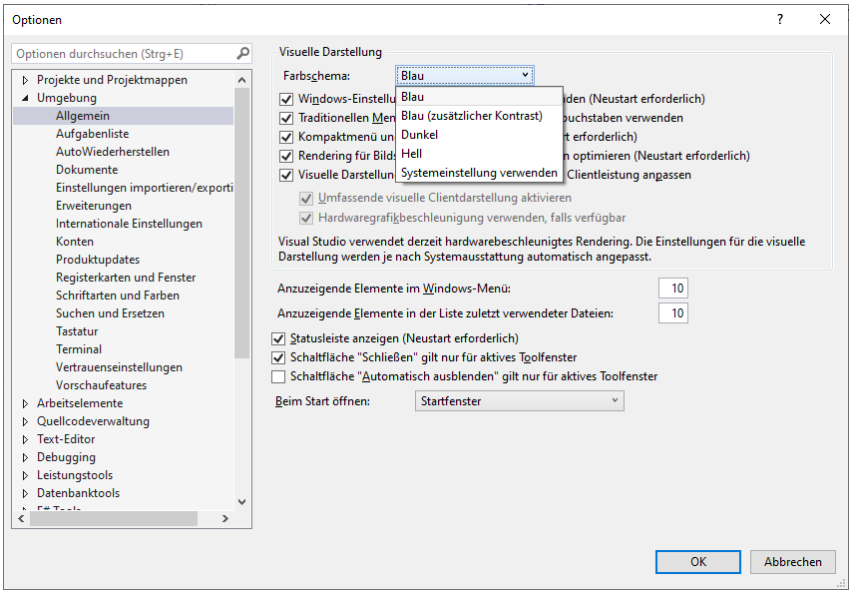
Anschließend werden einige vom Beginn an relevante Visual Studio - Einstellungen beschrieben. Andere Einstellungen werden später bei Bedarf behandelt.

3.3.11.1 Farbschema der Bedienoberfläche

Wer das beim ersten Start der Entwicklungsumgebung gewählte Farbschema der Bedienoberfläche später ändern möchte, kann das nach

Extras > Optionen > Umgebung > Allgemein

tun:

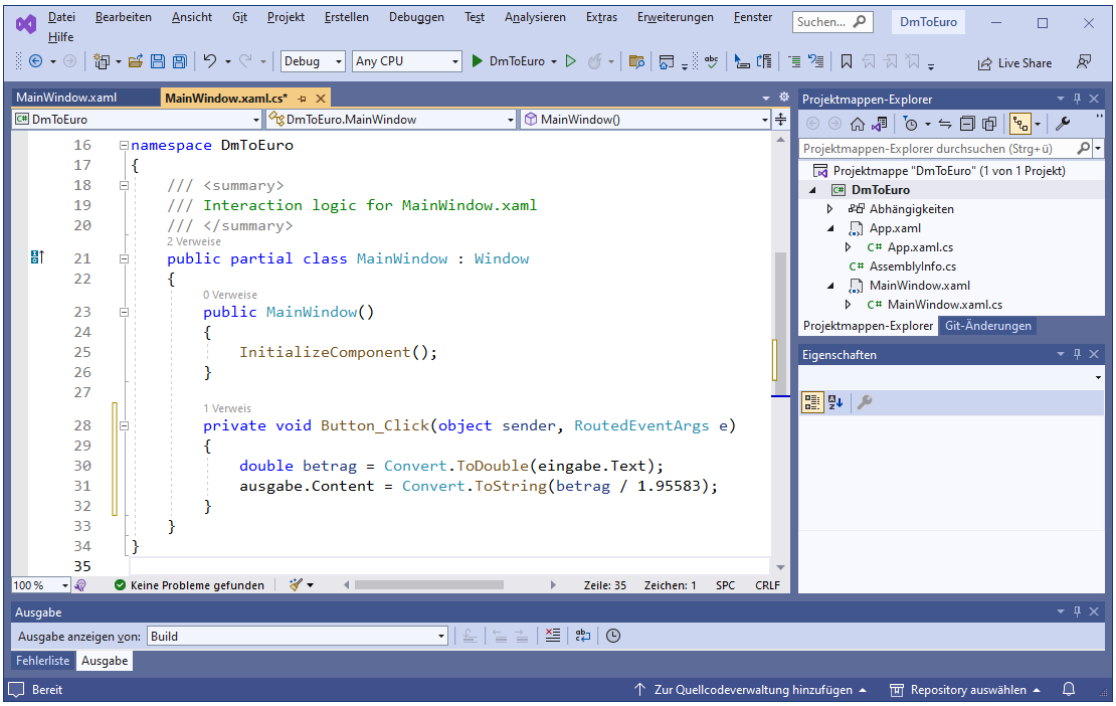


3.3.11.2 Farbschema des C#-Editors

Beim Zwischenablagentransfer von Quellcode in ein Textverarbeitungsprogramm gehen leider auch dann einige Farben von Textbestandteilen verloren, wenn nach

Extras > Optionen > Text Editor > Erweitert

das Kontrollkästchen **Beim Kopieren/Ausschneiden Rich-Text kopieren** markiert worden ist. Ist das voreingestellte C# - Farbschema im Einsatz, dann kommt z. B. von diesem Editor-Fenster



beim Zwischenablagentransfer (per Copy & Paste) das folgende Ergebnis in Microsoft Word 365 an:

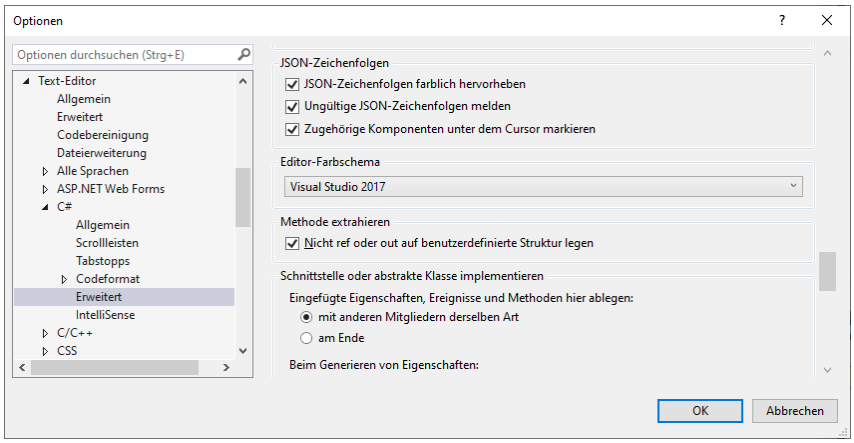
```
namespace DmToEuro
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

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

Die Methodennamen und die meisten Klassennamen verlieren ihre Farbe.

Um für die zahlreichen Quellcode-Darstellungen im Manuskript die Anzahl der zu korrigierenden Farbfehler in Grenzen zu halten, wurde nach

Extras > Optionen > Text Editor > C# > Erweitert > Editor-Farbschema

das **Editor-Farbschema** von Visual Studio 2017 eingestellt, weil dort die Namen der Methoden schwarz dargestellt werden, sodass beim Zwischenablagentransfer keine Farbe verloren gehen kann.



Bei fast allen Klassennamen im Manuskript musste die verlorene Farbe manuell wiederhergestellt werden, was sicher nicht lückenlos geglückt ist.

3.4 Assembly-Inspektion und -Dekompilierung

In diesem Abschnitt werden zwei Programme vorgestellt, die etliche Kompetenzen zur gründlichen Analyse von Assemblies besitzen, z. B.:

- Anzeige des IL-Codes der enthaltenen Klassen
- Dekompilierung (Rückübersetzung) des IL-Codes in C#-Code
- Anzeige der Metadaten

3.4.1 ILSpy

ILSpy ist eine vom SharpDevelop-Team[1] entwickelte Open Source – Software, die eigenständig verwendet[2] oder als Erweiterung (Plugin) in das Visual Studio integriert werden kann. Wir verwenden die zuletzt genannte Nutzungsart und starten die Installation der Erweiterung über den Menübefehl

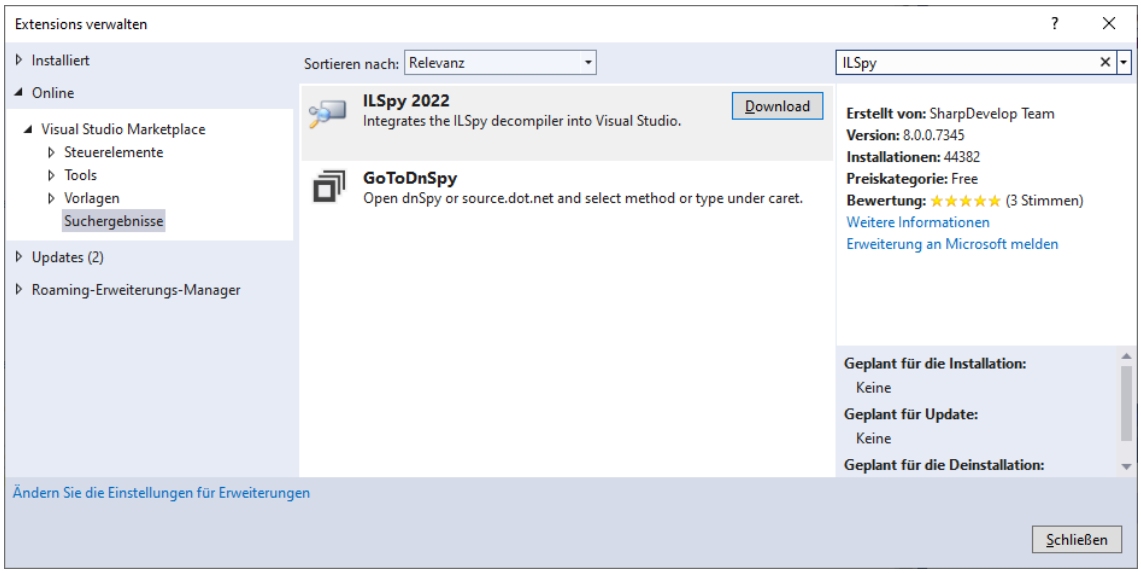
Erweiterungen > Erweiterungen verwalten

Wir suchen im **Visual Studio Marketplace** nach der Erweiterung **ILSpy 2022** und laden sie herunter:

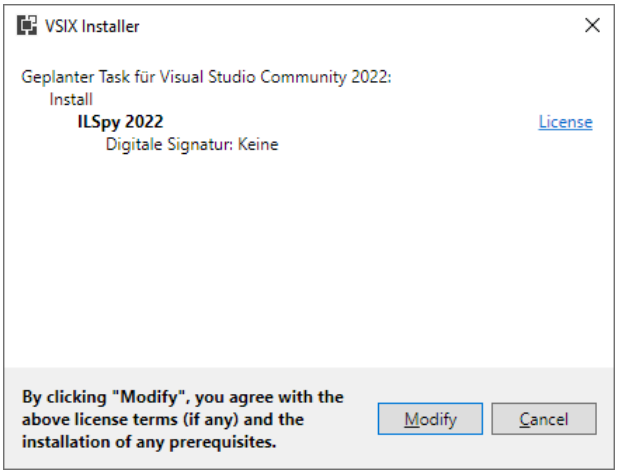
---

[1] Wer sich schon länger mit der .NET – Szene beschäftigt, kann sich vielleicht an die Entwicklungsumgebung **SharpDevelop** erinnern (https://sourceforge.net/projects/sharpdevelop/files/).

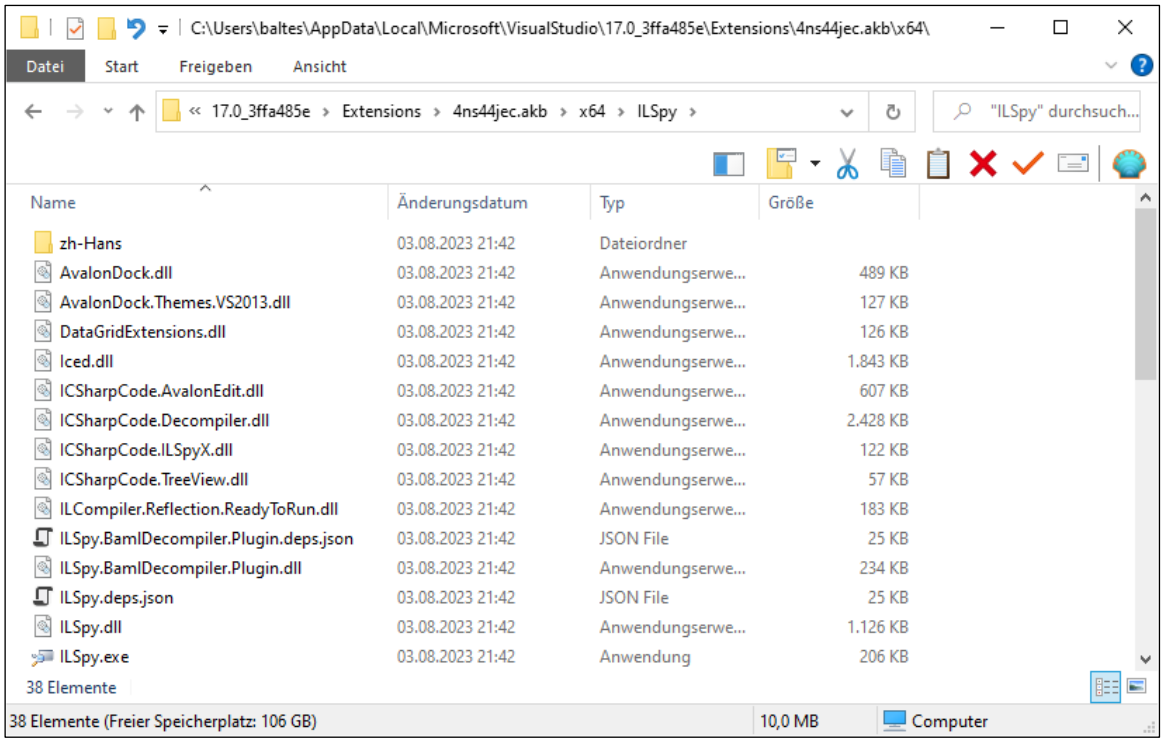
[2] https://sourceforge.net/projects/ilspy.mirror/



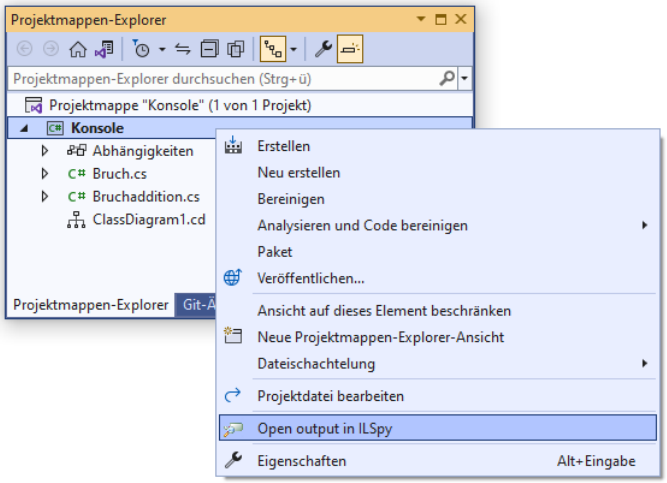
Zur Installation der Erweiterung durch den **VSIX Installer** muss das Visual Studio beendet werden:



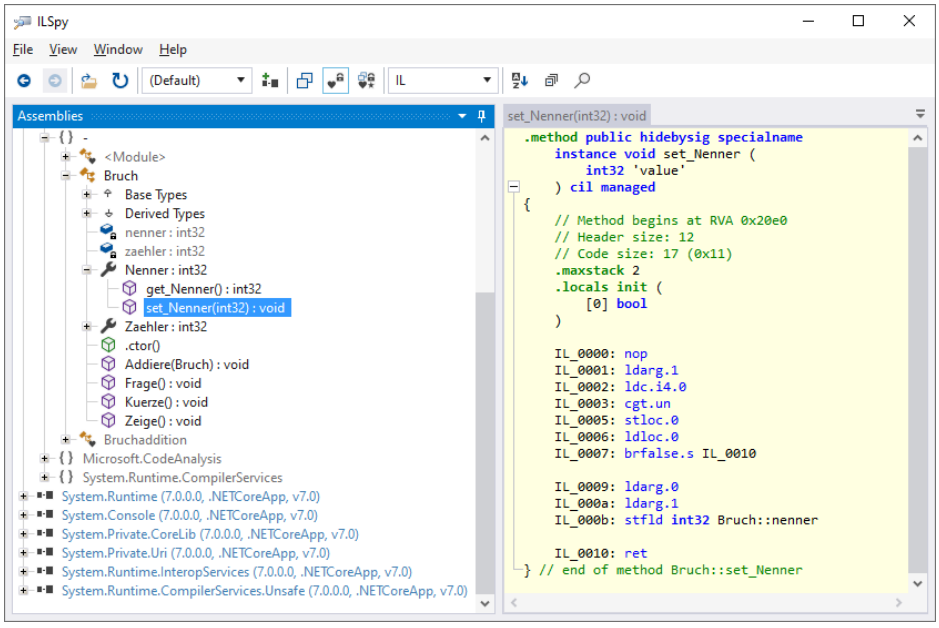
Die Installation landet im Windows-Benutzerprofil, z. B.:



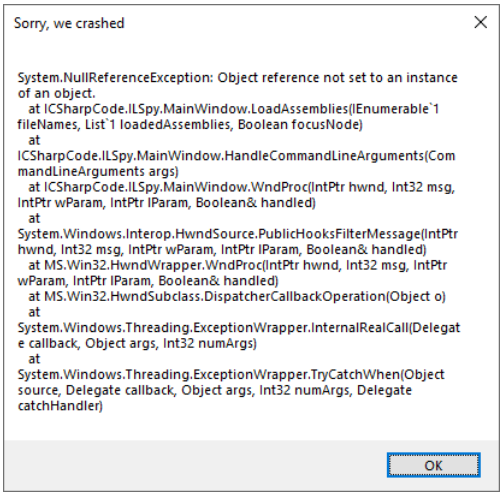
Nach dem nächsten Start kann im Visual Studio über das Kontextmenü zu einem Projekt



das zugehörige Assembly in ILSpy geöffnet werden. Hier ist der IL-Code zur Methode `set_Nenner()` in der Klasse `Bruch` zu sehen (vgl. Abschnitt 2.3):

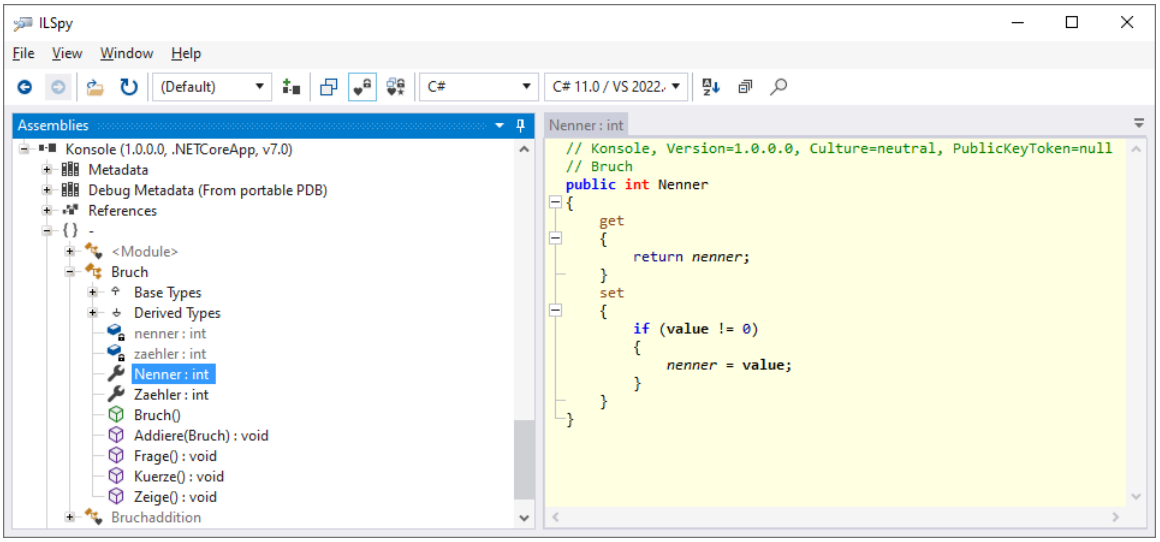


Ein (mit Visual Studio 17.6.x und ILSpy 8.0.0.7345 beobachteter) gelegentlicher Ausnahmefehler beim Start der Erweiterung



behindert die Nutzung nicht.

Ist von einem Assembly nur der IL-Code vorhanden (die ausführbare **dll**-Datei), dann kann ILSpy den Quellcode ziemlich perfekt rekonstruieren, z. B.:



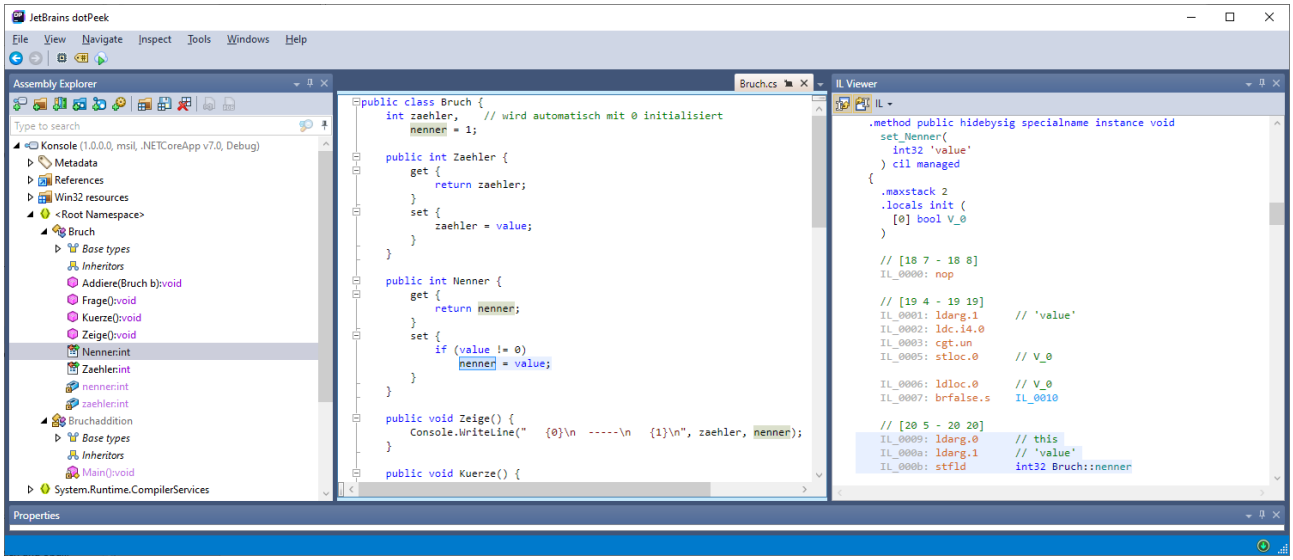
Wer seinen C# - Quellcode geheim halten (sein geistiges Eigentum schützen) möchte, muss einen sogenannten *Obfuskator* verwenden.[1] Solche Programme verändern den IL-Code, um die Rückübersetzung zu verhindern.

3.4.2 dotPeek

Eine Alternative zu ILSpy ist das auf dieser Webseite

https://www.jetbrains.com/de-de/decompiler/

kostenlos verfügbare Programm **dotPeek** der Firma **JetBrains**. Es kann u. a. zu einer markierten Stelle im C# - Code die korrespondierende Stelle im IL-Code anzeigen:

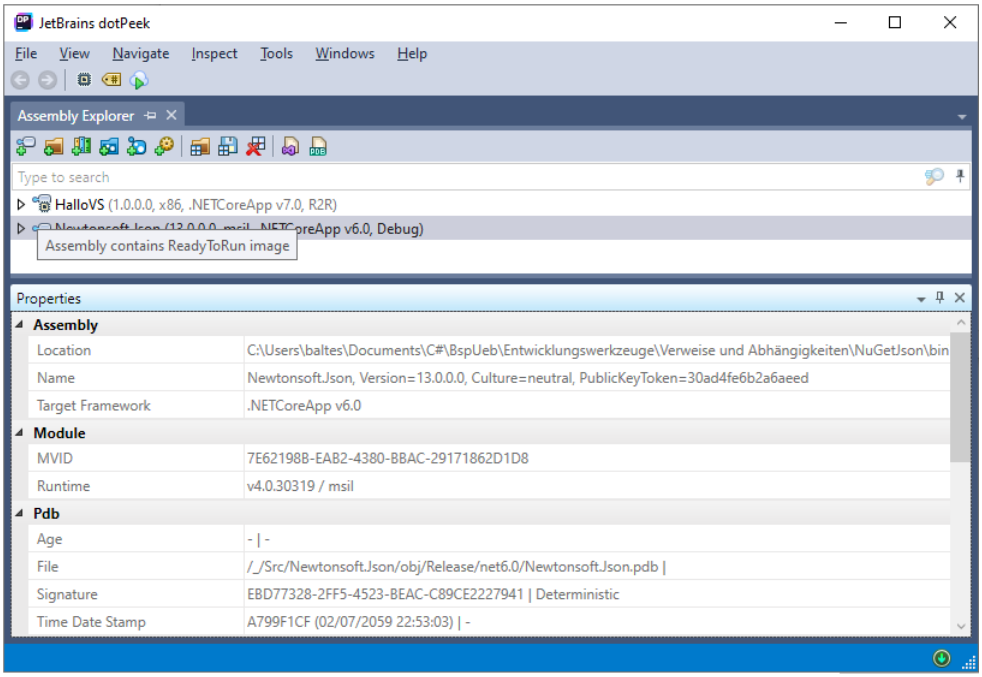


Im konkreten Fall zeigt sich erneut, dass zum **set**-Block einer C# - Eigenschaftsdefinition eine IL-Methodendefinition mit dem Namensanfang **set** gehört (im Beispiel: `set_Nenner()`).

Ein R2R-Assembly (vgl. Abschnitt 2.5.2) ist am Symbol schnell zu erkennen, z. B.:

---

[1] https://de.wikipedia.org/wiki/Obfuskation_(Software)#Java-Bytecode_und_MSIL
https://learn.microsoft.com/en-us/visualstudio/ide/dotfuscator/



3.5 Microsoft Visual Studio Code

Microsoft bietet unter dem Namen *VS Code* eine leichtgewichtige Alternative zum Visual Studio an, die einige Pluspunkte aufzuweisen hat:

- Verfügbarkeit für Linux, macOS und Windows
- Geringe Hardwareanforderungen
- Eine für unsere Zwecke ausreichende Unterstützung bei der Software-Entwicklung mit C# (inklusive Editor mit IntelliSense, Refaktorierung etc.)

Trotzdem sind wir im Abschnitt 3.2 zur Entscheidung gekommen, dass unter Windows für die Software-Entwicklung mit C# das Visual Studio die beste Wahl ist (z. B. wegen des integrierten Fenster-Designers, der die Entwicklung von Anwendungen mit grafischer Bedienoberflächen erleichtert). Unter Linux ist das VS Code hingegen ohne ernsthafte Alternative.[1]

3.5.1 Voraussetzungen

Die im Mai 2023 genannten Systemvoraussetzungen für das VS Code sind sehr bescheiden:[2]

- Betriebssystem:
  - Linux:
    - Debian: Ubuntu Desktop 16.04, Debian 9
    - Red Hat: Red Hat Enterprise Linux 7, CentOS 7, Fedora 34
  - macOS: ab 10.13
  - Windows: 10 oder 11 (32-Bit oder 64-Bit)
- Prozessor mit 1,6 GHz
- 1 GB RAM
- Für die C# - Entwicklung werden unter Windows (ohne .NET SDK 7) ca. 1,5 GB Festspeicherplatz (auf einer SSD oder Festplatte) benötigt.

Unter Linux sind die folgenden Bibliotheken erforderlich:

---

[1] Das VS Code konkurriert auf dem Mac mit einer speziellen Visual Studio - Version, die auf dem Xamarin Studio basiert. Mangels Erfahrung sind dazu keine Empfehlungen möglich.

[2] https://code.visualstudio.com/docs/supporting/requirements

- GLIBCXX ab Version 3.4.21
- GLIBC ab Version 2.17

3.5.2 Bezugsquelle

Von der Webseite

https://code.visualstudio.com/download

ist das Visual Studio Code für Linux, macOS und Windows zu beziehen. Unter Linux und Windows wird zwischen einer benutzer- und einer system-seitigen Installation unterschieden, wobei die benutzer-seitige Installation wegen der größeren Bequemlichkeit bei der Aktualisierung empfohlen wird.

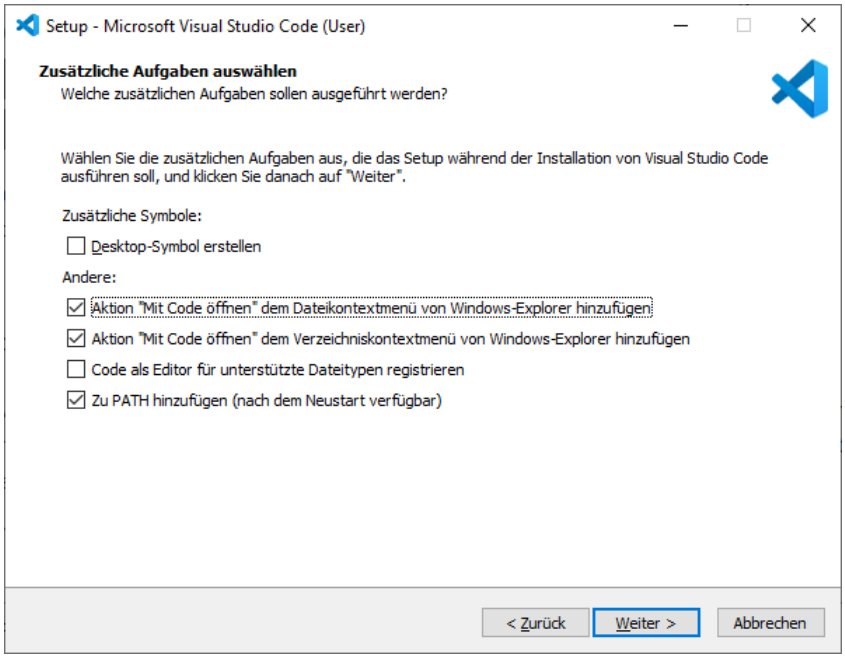
3.5.3 Installation

Microsoft unterstützt bei VS Code eine benutzer- und eine system-bezogene Installation mit einer klaren Empfehlung für die benutzer-bezogene Variante, weil dabei sowohl die Installation als auch die oft angebotene Aktualisierung ohne Administratorrechte möglich sind.

Das benutzer-seitige Windows-Installationsprogramm (z. B. **VSCodeUserSetup-x64-1.79.2.exe**) verwendet für einen Benutzer namens `baltes` per Voreinstellung den folgenden Installationsordner:

C:\Users\baltes\AppData\Local\Programs\Microsoft VS Code

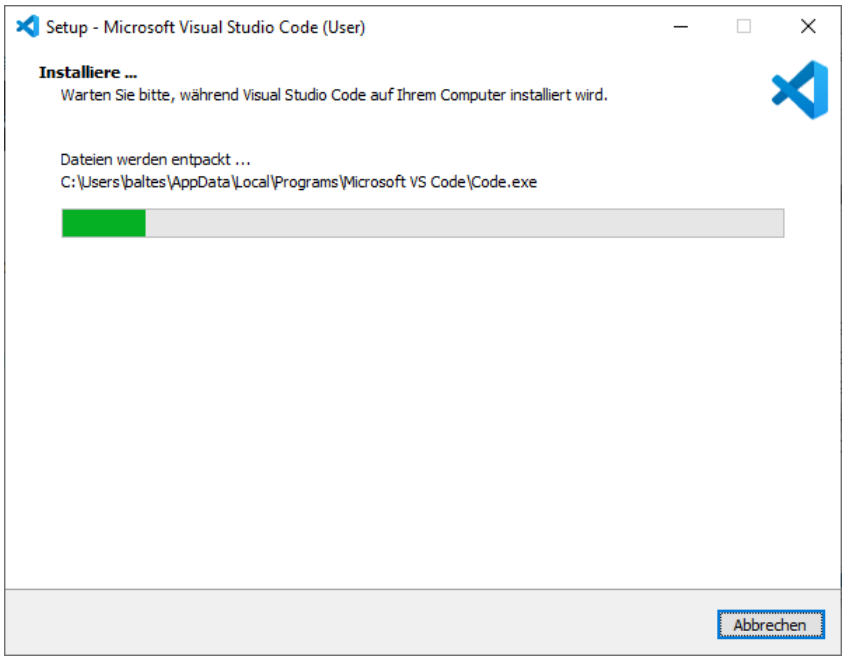
Werden als **zusätzliche Aufgaben** die beiden Optionen **Mit Code öffnen** markiert,



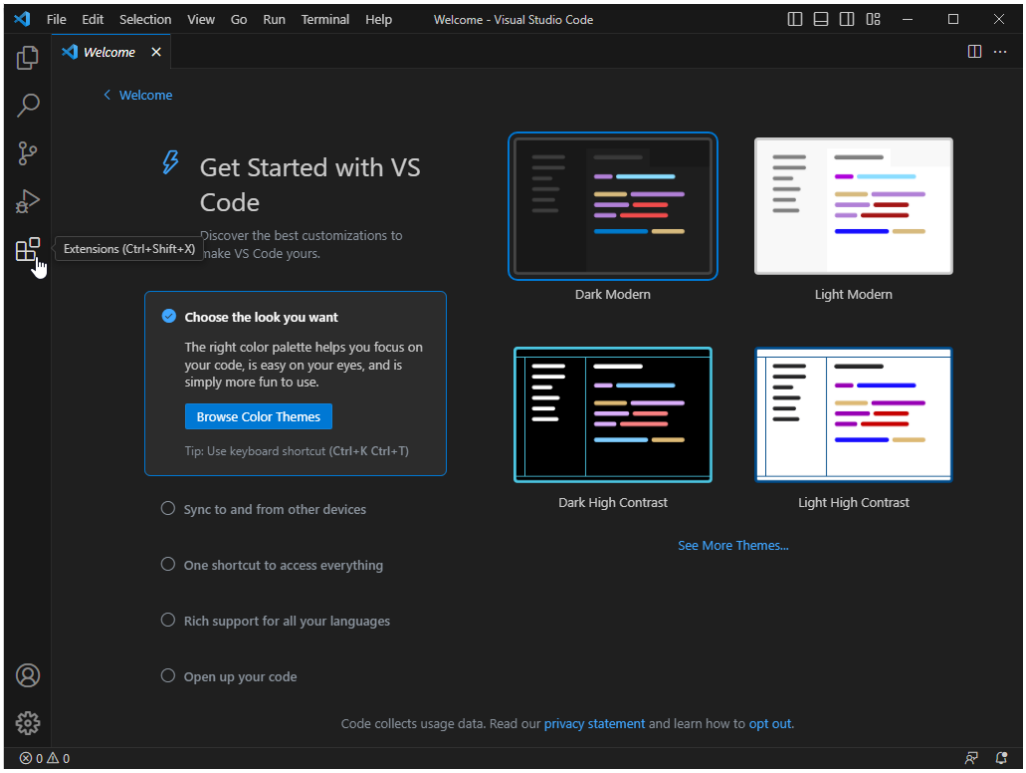
dann lassen sich Projekte per Explorer-Fenster im VS Code öffnen:

- über das Kontextmenü zum Namen des Projektordners
- oder über das Kontextmenü zum Hintergrund des geöffneten Projektordners

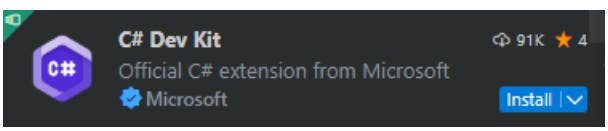
Die Installation dauert nur wenige Sekunden:



Für die Entwicklung von Software in C# wird in VS Code eine Erweiterung benötigt, die nach dem Start der Entwicklungsumgebung installiert werden kann. Man klickt zunächst auf den für **Extensions** zuständigen Schalter:



Dann wählt man (bei Bedarf unterstützt durch die Eintragung **C#** im Suchfeld) die Erweiterung **C# Dev Kit**:

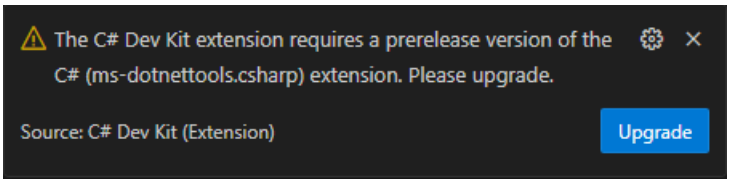


Für den Windows-Benutzer `baltes` werden VS Code - Erweiterungen in den folgenden Ordner installiert:

C:\Users\baltes\.vscode\extensions

VS Code legt Erweiterungen grundsätzlich benutzer-bezogen ab, sodass Aktualisierungen mit normalen Benutzerrechten möglich sind. Solche Aktualisierungen finden nötigenfalls automatisch beim Zugriff statt.

Der Bitte, die C# - Erweiterung zu aktualisieren, kommen wir nach:



Die VS Code - Konfiguration des Windows-Benutzers `baltes` landet im folgenden Ordner:

C:\Users\baltes\AppData\Roaming\Code

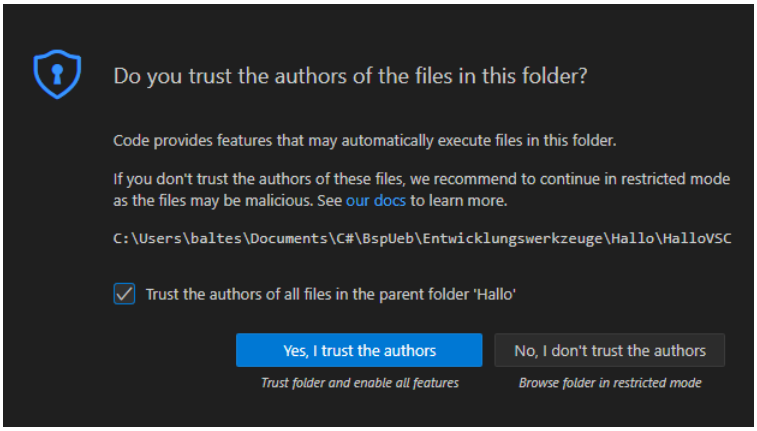
Zusätzlich zu VS Code wird das .NET SDK in einer aktuellen Version benötigt (zur Installation siehe Abschnitt 3.1.1). Für das Arbeiten mit dem VS Code muss das dotnet-CLI (also unter Windows das Programm **C:\Program Files\dotnet\dotnet.exe**) über den Suchpfad für ausführbare Programme erreichbar sein, was nach einer .NET SDK - Installation der Fall ist.

3.5.4 Ein erstes Konsolen-Projekt

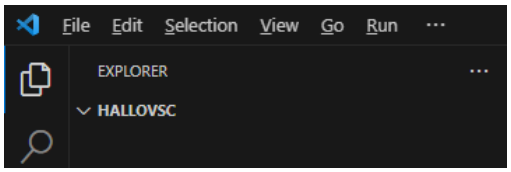
3.5.4.1 Projekt erstellen

Wir erstellen ein Konsolenprogramm für .NET 7 mit dem Namen `HalloVSC`:

- Wählen Sie im VS Code den Menübefehl **File > Open Folder**
- Legen Sie im Dialog **Open Folder** einen neuen Ordner an, und **wählen** Sie diesen **aus**.
- Bestätigen Sie Ihr Vertrauen gegenüber den Autoren der im neuen Ordner sowie im übergeordneten Ordner befindlichen Dateien:



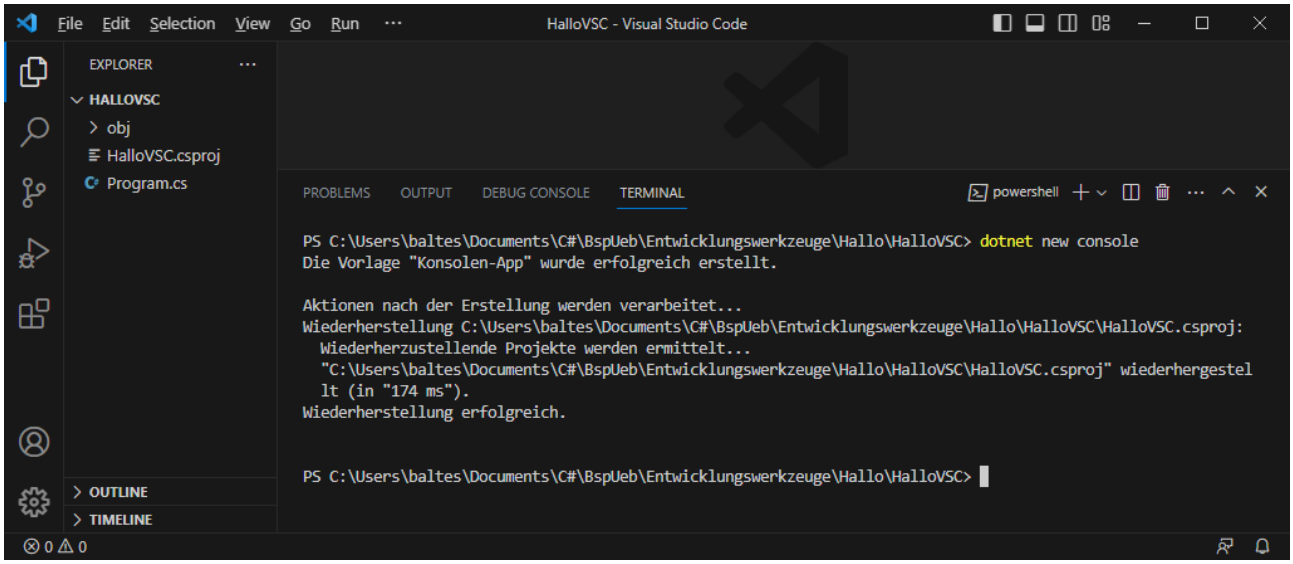
Der **EXPLORER**, den man bei Bedarf über sein Symbol  in den Vordergrund holt, zeigt den Projektordner an:



- Wählen Sie in VS Code den Menübefehl **View > Terminal**.
- Schicken Sie im **TERMINAL** per **Enter**-Taste den folgenden Befehl ab, der ein Projekt unter Verwendung der Vorlage **console** erstellt (vgl. Abschnitt 3.1.2):

```
dotnet new console
```

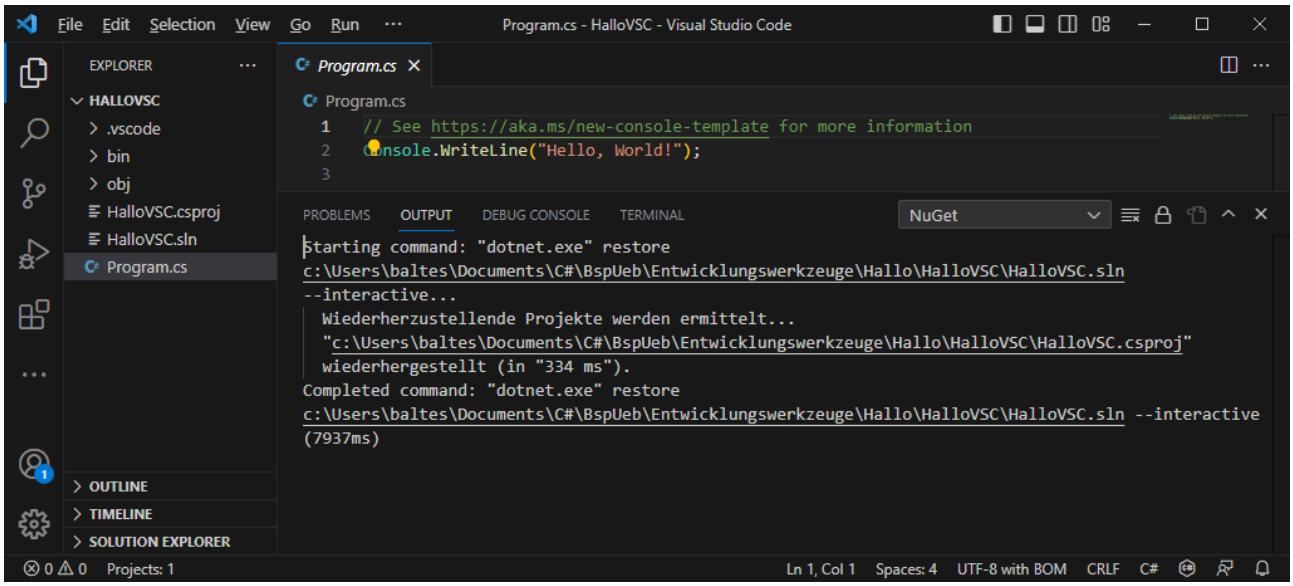
Im **TERMINAL** erscheint eine Erfolgsmeldung,



und der **EXPLORER** zeigt zum neuen Projekt:

- den Unterordner **obj** mit Hilfsdateien für die Erstellung des Programms
- die Projektdatei **HalloVSC.csproj**
- die C# - Quellcodedatei **Program.cs**

Nach einem Mausklick auf die Quellcodedatei **Program.cs** erscheint ein automatisch erstelltes Hallo-Programm im Editor:



Hier wird die seit C# 9 bestehende Möglichkeit genutzt, durch Anweisungen der obersten Ebene dem Compiler die Definition einer Startklasse mit **Main()** – Methode zu überlassen.

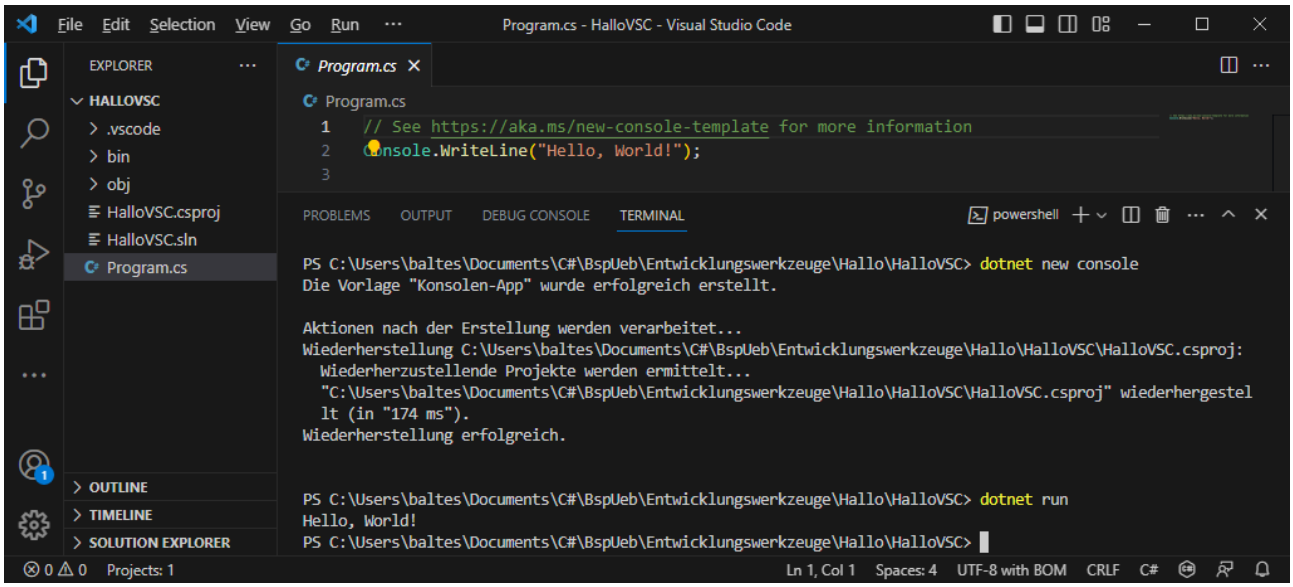
Außerdem werden bei dieser Gelegenheit einige Arbeiten an der Projektkonfiguration ausgeführt.

3.5.4.2 Programm erstellen und ausführen

Schicken Sie im **TERMINAL** den folgenden Befehl

```
dotnet run
```

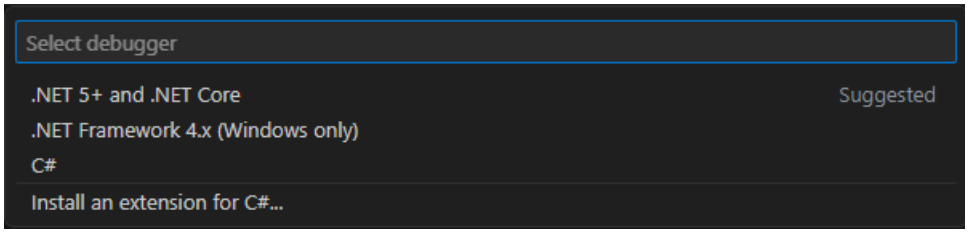
ab, um das Programm zu erstellen und zu starten:



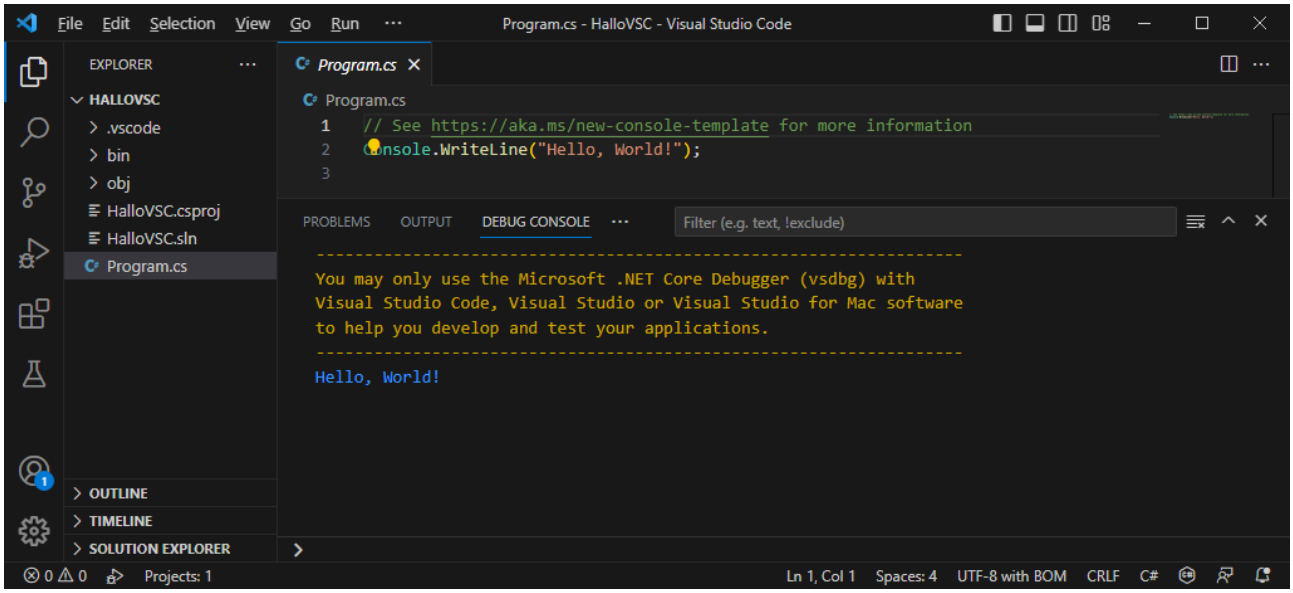
Das Programm lässt sich auch ohne **TERMINAL** starten:

- Menübefehl **Run > Without Debugging**
- oder Tastenkombination **Strg-F5**

Auf Nachfrage ist (einmalig pro Projekt) die Debug-Konfiguration **C#** zu wählen:

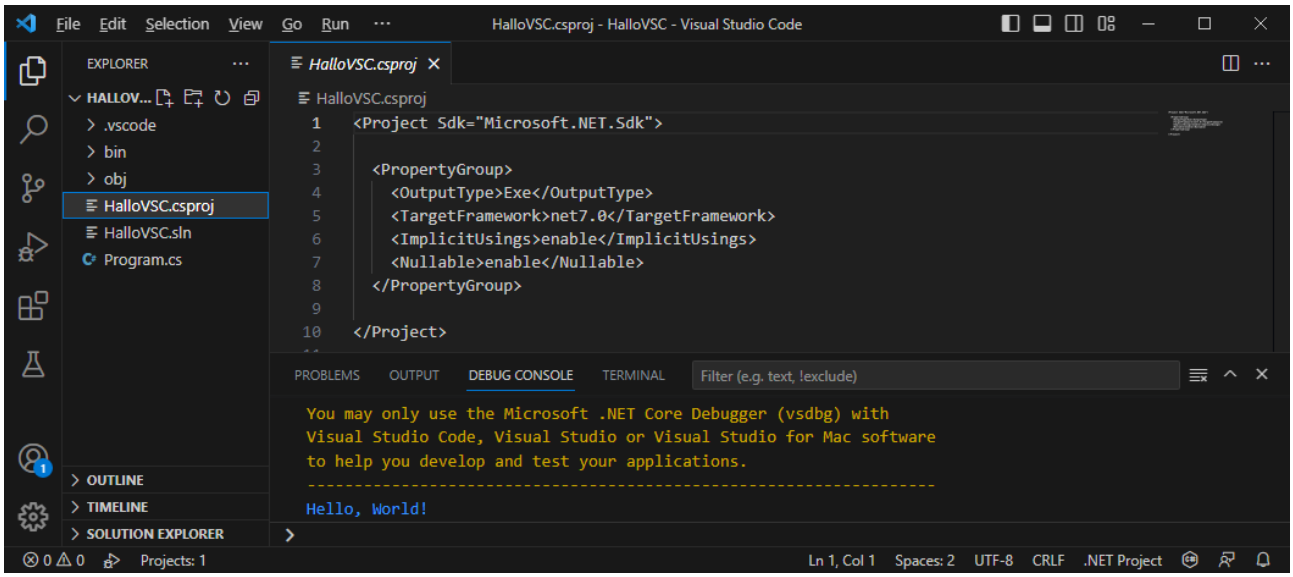


Diesmal erscheint die Ausgabe in der **DEBUG CONSOLE**:



3.5.4.3 Projektkonfiguration

Um die Einstellungen für ein Projekt zu überprüfen oder zu verändern, öffnet man die Projektdatei per Mausklick, z. B.:



Den Inhalt kennen wir schon, weil das Projekt per dotnet-CLI erstellt worden ist (wie im Abschnitt 3.1.2).

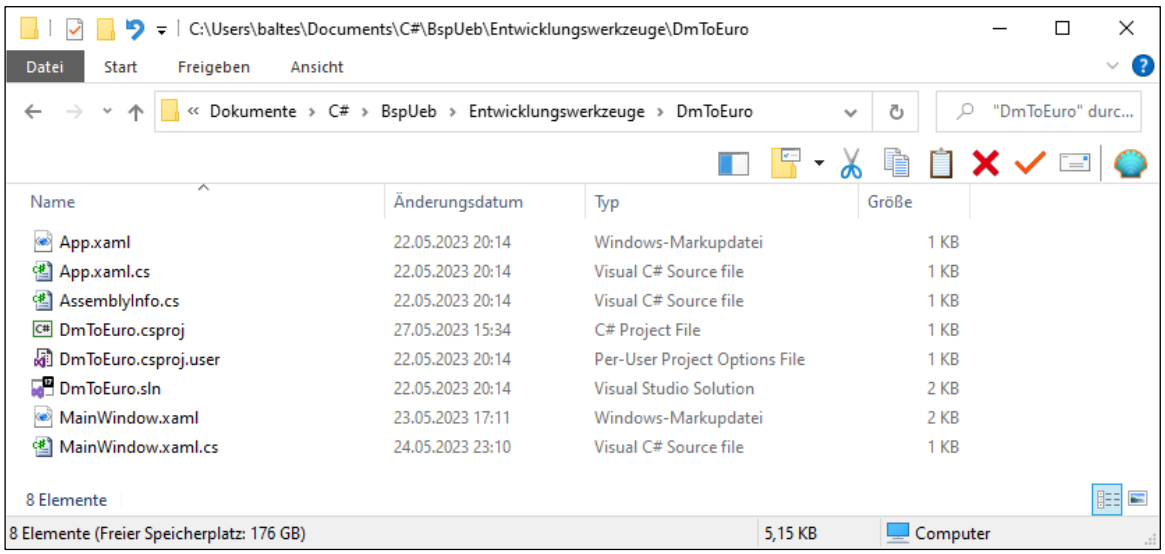
3.6 Projekte mit Beispielen und Übungen

Zusammen mit dem Manuskript wird das Zip-Archiv **BspUeb.zip** zur Verfügung gestellt (siehe Vorwort zum Bezug). Dort befinden sich Visual Studio - Projekte mit Beispielen und Lösungsvorschlägen zu Übungsaufgaben.

Im Manuskript werden zu vielen Beispielen und Übungsaufgaben die zugehörigen Visual Studio – Projektordner angegeben, z. B.:

…\BspUeb\Entwicklungswerkzeuge\DmToEuro

Nach dem Entpacken der Datei **BspUeb.zip** sind die Projektordner im Windows-Explorer verfügbar, z. B.:



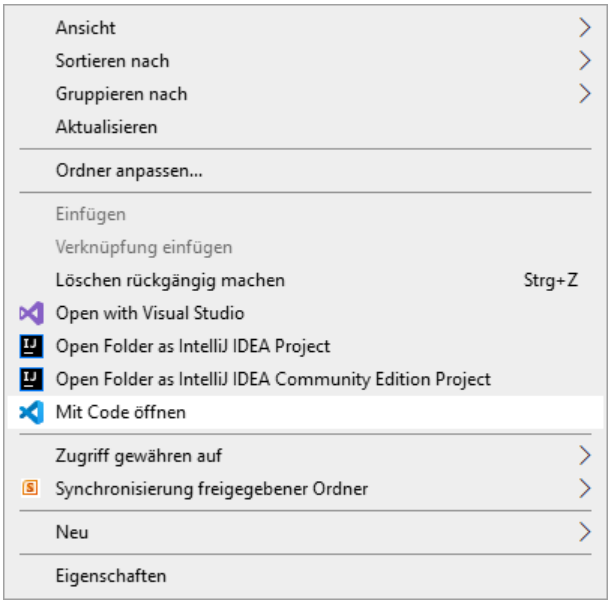
Nun lassen sich die Projekte mit den windows-üblichen Verfahren öffnen:

- im Visual Studio z. B. über die Projektdatei,
- im VS Code über das Kontextmenü zum Fensterhintergrund (siehe Abschnitt 3.6.1).

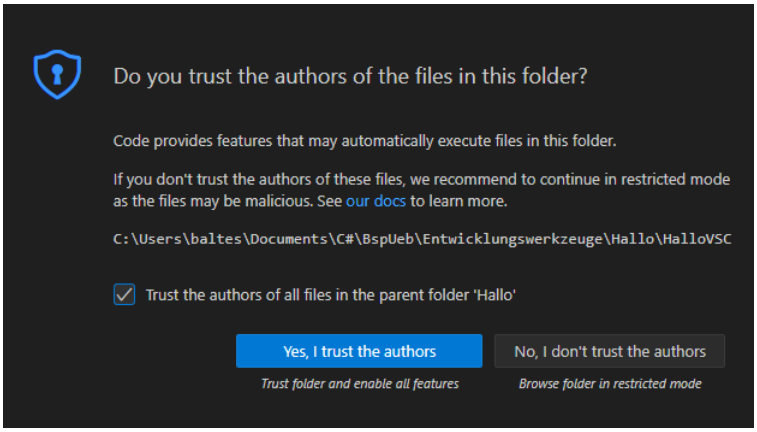
3.6.1 Projekte im VS Code öffnen

Weil das VS Code dieselbe Projektordnerstruktur verwendet wie das Visual Studio, lassen sich die aus dem Archiv **BspUeb.zip** stammenden Projekte auch mit VS Code öffnen:

- Wählen Sie aus dem Kontextmenü zum Namen des Projektordners oder zur Hintergrundfläche des Projektordners das Item **Mit Code öffnen**, z. B.:



- Bestätigen Sie ggf. Ihr Vertrauen gegenüber den Autoren der im Projektordner sowie im übergeordneten Ordner befindlichen Dateien:



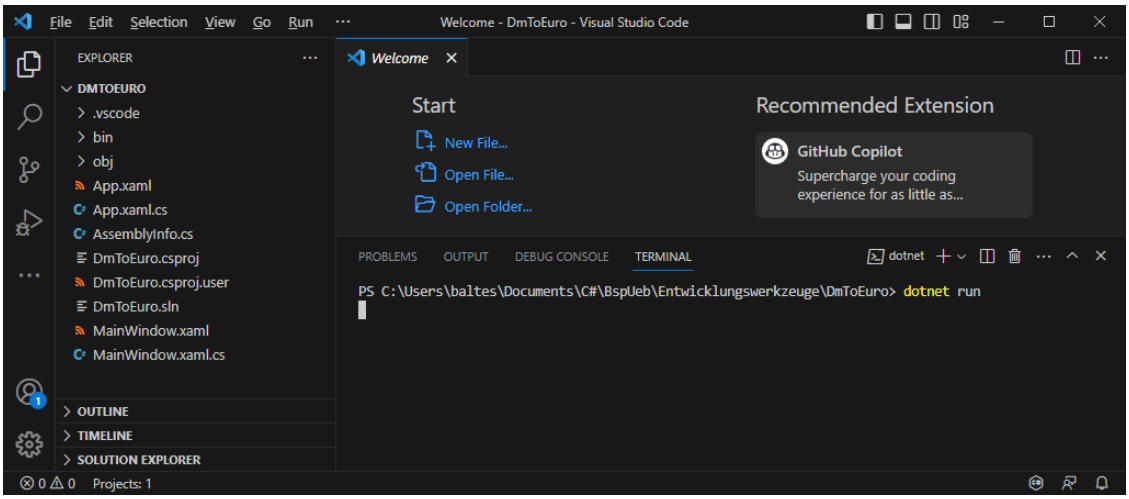
- Lassen Sie im Terminalfenster das folgende Kommando ausführen:

  ```
  dotnet run
  ```

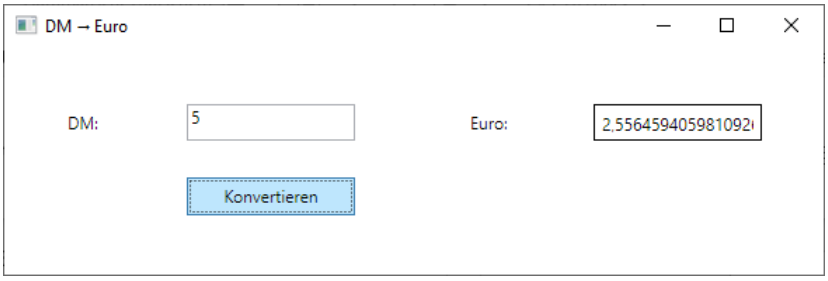
  Wird das Projekt im Ordner

  …\BspUeb\Entwicklungswerkzeuge\DmToEuro

  gestartet,



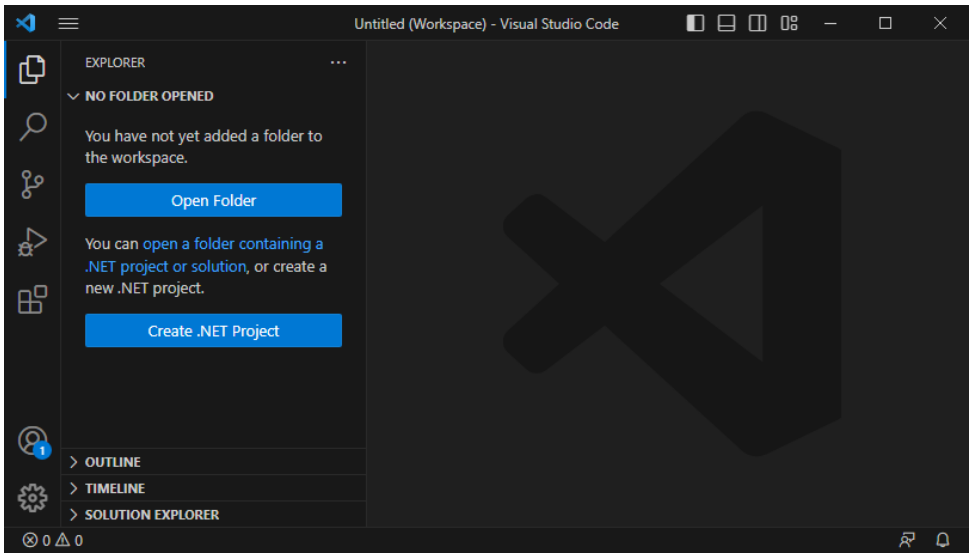
dann erscheint ein mit WPF-Technik erstelltes Anwendungsfenster (vgl. Abschnitt 3.3.7):



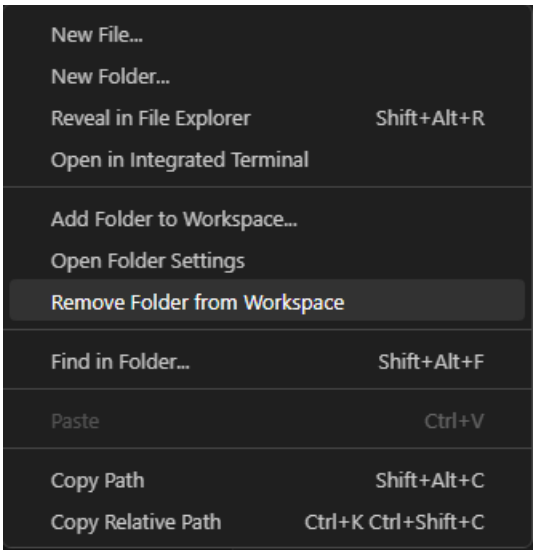
Offenbar lassen sich auch WPF-Projekte in VS Code öffnen, wobei man zur Gestaltung der Bedienoberfläche den XAML-Code editieren muss.

Im Manuskript spielen die vom Visual Studio unterstützten Projektmappen mit *mehreren* Projekten keine wesentliche Rolle. Als Ausnahme enthält der Abschnitt 3.3.9 eine Mappe mit *zwei* Projekten, um einen Projektverweis demonstrieren zu können. Eine Visual Studio – Projektmappe lässt sich auf einen VS Code - **Workspace** (dt.: *Arbeitsbereich*) abbilden, was anschließend für das Beispiel aus dem Abschnitt 3.3.9 vorgeführt wird:

- Starten Sie VS Code, und sorgen Sie dafür, dass kein Arbeitsbereich geöffnet ist:



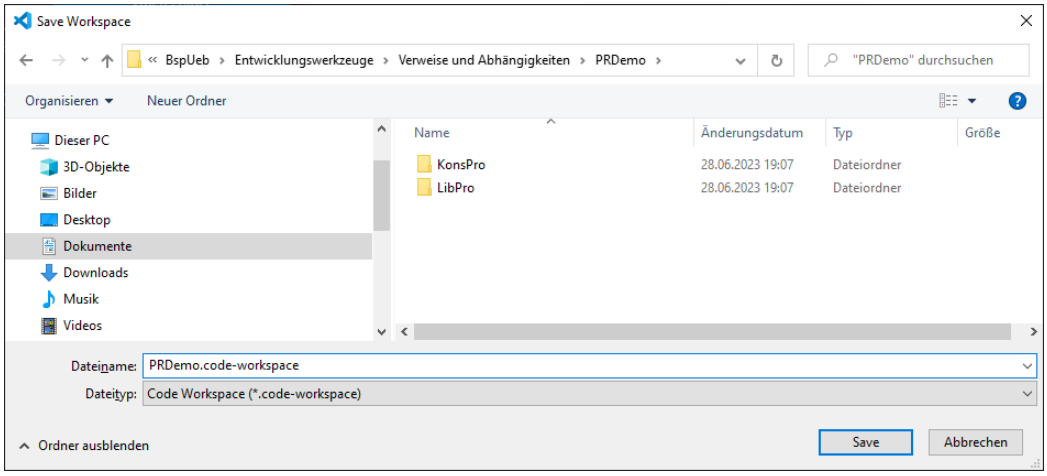
Dieser Zustand lässt sich z. B. über das Item **Remove Folder from Workspace** aus dem Kontextmenü zur **EXPLORER**-Fensterzone herbeiführen.



- Erstellen Sie nach dem Menübefehl

  File > Save Workspace As

  im Ordner mit den Projekten, die zum Arbeitsbereich gehören sollen, eine Arbeitsbereichsdatei mit der Namenserweiterung **code-workspace**, z. B.:



- Öffnen Sie das Projekt `KonsPro` mit der auszuführenden Anwendung über den Menübefehl

  File > Add Folder to Workspace

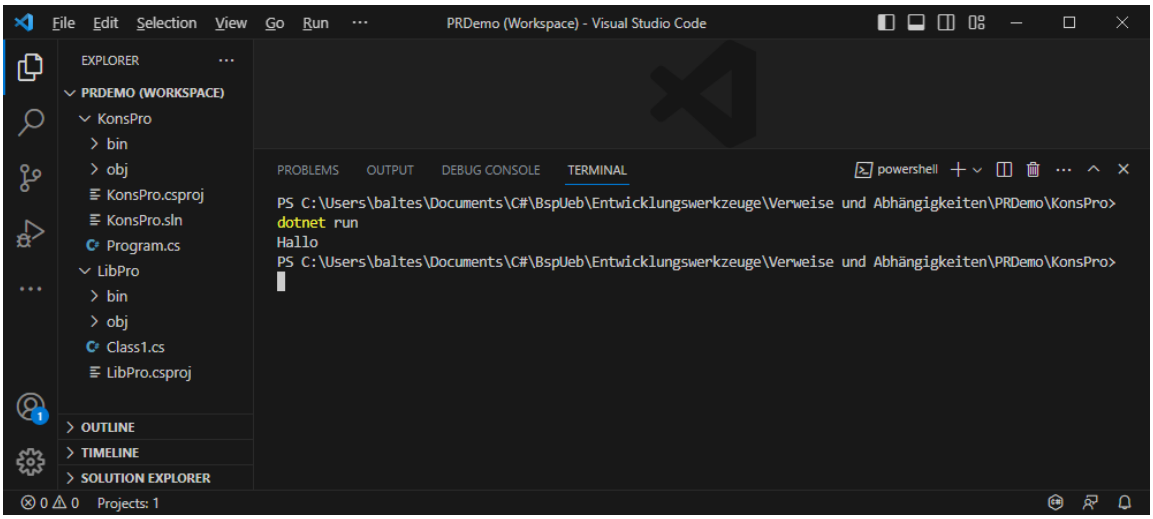
- Öffnen Sie das Projekt `LibPro` über den Menübefehl

  File > Add Folder to Workspace

- Lassen Sie im Terminalfenster das Kommando

  ```
  dotnet run
  ```

  ausführen:



3.6.2 Übernahme von Quellcode aus der PDF-Datei mit dem Manuskript

Bei der Übernahme von Quellcode aus der PDF-Datei mit dem Manuskript via Windows-Zwischenablage geht leider die Formatierung verloren. Optische Verluste kann das Visual Studio beim Einfügen durch die automatische Formatierung wieder egalisieren. Wenn allerdings beim Transfer Leerzeichen aus einer Zeichenfolge entfernt worden sind, dann verhält sich der eingefügte Quellcode nicht mehr korrekt. Daher sollte der Quellcode in der Regel aus dem ausgepackten Archiv **BspUeb.zip** übernommen werden. Seine Ordner bilden ungefähr die Kapitel und Abschnitte des Manuskripts ab.

3.7 Übungsaufgaben zum Kapitel 3

1) Installieren Sie nach Möglichkeit auf Ihrem privaten PC das Visual Studio Community 2022 (gemäß Abschnitt 3.3) und/oder das Visual Studio Code (gemäß Abschnitt 3.5).

2) Beseitigen Sie die Fehler in der folgenden Variante des `Hallo`-Programms:

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

3) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Bei der Programmerstellung per dotnet-CLI wird von den installierten .NET – Versionen stets die neueste verwendet.
2. Mit dem Visual Studio Community dürfen Einzelentwickler (nicht aber Software-Firmen) auch kommerzielle Programme erstellen.
3. Im VS Code verwendet man zum Erstellen eines Projekte das dotnet-CLI in einem Terminal-Fenster.
4. Eine mit dem Visual Studio erstellte WPF-Anwendung kann mit VS Code nicht geöffnet und verändert werden, weil das VS Code keinen grafischen WPF-Designer besitzt.
5. Zu einem im Quellcode-Editor markierten (die Einfügemarke enthaltenden) C# - Schlüsselwort oder BCL-Bezeichner liefert das Visual Studio über die Funktionstaste **F1** die zugehörige Dokumentation.

4) Welche Dateien im Ausgabeordner einer Projekterstellung (z. B. **…\bin\Debug\net7.0**) müssen mindestens an Kunden ausgeliefert werden?

4 Elementare Sprachelemente

Im Kapitel 1 wurde anhand eines halbwegs realistischen Beispiels versucht, einen ersten Eindruck von der objektorientierten Software-Entwicklung 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 in C# nicht wesentlich anders aus als in älteren, *nicht* objektorientierten Sprachen (z. B. in C).

4.1 Einstieg

Der aktuelle Abschnitt dient u. a. dazu, ein minimalistisches, zum Erlernen von elementaren Sprachbestandteile geeignetes Konsolenprogramm zu beschreiben. Dabei kommen grundlegende Konstruktionsprinzipien von C# - Programmen zur Sprache, die Sie bald beherrschen werden.

4.1.1 Aufbau von einfachen C# - Konsolenprogrammen

Unsere vorläufige Vorstellung vom Aufbau eines C# - Konsolenprogramms kann folgendermaßen zusammengefasst werden:

- Ein C# - Programm besteht aus **Klassen**. Für das Bruchrechnungsbeispiel im Kapitel 1 wurden die Klassen `Bruch` und `Bruchaddition` definiert. In den Methoden der beiden Klassen kommen Klassen aus der Base Class Library (BCL) zum Einsatz (**Console**, **Math** und **Convert**).
- Eine **Klassendefinition** besteht aus …
  - dem **Kopf**
    Er enthält nach dem Schlüsselwort **class** den Namen der Klasse. Soll eine Klasse für beliebige andere Klassen (in fremden Assemblies) nutzbar sein, dann muss dem Schlüsselwort **class** der Zugriffsmodifikator **public** vorangestellt werden, z. B.:
    ```
    public class Bruch {
      . . .
    }
    ```
  - und dem **Rumpf**
    Begrenzt durch ein Paar geschweifter Klammern befinden sich hier …
    - die Deklarationen der **Felder**
    - sowie die Definitionen der **Methoden** und **Eigenschaften**.
- Auch eine **Methodendefinition** besteht aus …
  - dem **Kopf**
    Seine Bestandteile sind:
    - Modifikatoren
      Mit dem Zugriffsmodifikator **public** wird z. B. vereinbart, dass eine Methode für beliebige Klassen (in beliebigen Assemblies) nutzbar sein soll.
    - Rückgabetyp
    - Name der Methode
    - Parameterliste

    Diese Bestandteile werden noch ausführlich erläutert.
  - und dem **Rumpf**
    Begrenzt durch ein Paar geschweifter Klammern befinden sich hier **Anweisungen** zur Realisation von Algorithmen. Dabei werden lokale Variablen zum Speichern von Zwischenergebnissen verwendet und oft auch Instanzvariablen des agierenden Objekts verändert. Der Unterschied zwischen Instanzvariablen (Merkmalen von Objekten), statischen Variablen (Merkmalen von Klassen) und lokalen Variablen von Methoden wird im Abschnitt 4.3 erläutert.

Details zur Definition einer Methode und zur Definition einer Eigenschaft, die als Paar von Methoden für den lesenden und den schreibenden Zugriff auf ein Feld aufgefasst werden kann, folgen im Abschnitt 4.1.3. Die offizielle und umfassende Behandlung dieser Begriffe erfolgt im Kapitel 5.

- Eine **Anweisung** ist die kleinste ausführbare Einheit eines Programms. In C# sind bis auf wenige Ausnahmen alle Anweisungen mit einem **Semikolon** abzuschließen.
- Von den Klassen eines Programms muss eine **startfähig** sein. Dazu benötigt sie eine Methode mit dem Namen **Main()** und folgenden Besonderheiten:
  - Modifikator **static**
    Damit gehört die Methode **Main()** zum Handlungsrepertoire der Klasse.
  - Rückgabetyp **int** oder **void**

  Diese Methode wird beim Programmstart von der Laufzeitumgebung (der CLR) aufgerufen. Wenn die **Main()** - Methode ihrem Aufrufer per **return**-Direktive (siehe Abschnitt 5.3.1.2) 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 Rückgabetyp **void** anzugeben. Im Bruchadditionsbeispiel ist die Klasse `Bruchaddition` startfähig. Ihre **Main()** - Methode besitzt den Rückgabetyp **void**.
- Meist bringt man den Quellcode einer Klasse in einer eigenen Datei unter, die den Namen der Klasse übernimmt und **.cs** als Namenserweiterung erhält.
- Die ersten Zeilen einer C# - Quellcodedatei enthalten meist **using**-Direktiven zum Importieren von **Namensräumen**, damit die dortigen Klassen später ohne Namensraumpräfix vor dem Klassennamen angesprochen werden können.
- Die zu einem Programm gehörigen Quellcodedateien werden gemeinsam vom **Compiler** in die **Intermediate Language** (IL) übersetzt. Das resultierende Assembly übernimmt seinen Namen per Voreinstellung von der Startklasse. Es enthält neben dem IL-Code auch Typ- und Assembly-Metadaten.

Für die Beschäftigung mit elementaren C# - Sprachelementen eignen sich Konsolenprogramme mit einer sehr einfachen und nicht sonderlich objektorientierten Struktur, die Sie schon aus dem `Hallo`-Beispiel kennen (siehe z. B. Abschnitt 3.1.3):

```
using System;
class Prog {
   static void Main() {
      Console.WriteLine("Echt .NET hier!");
   }
}
```

Es wird nur *eine* Klasse definiert, und diese Klasse enthält nur eine einzige Methodendefinition. Weil die Klasse startfähig sein muss, liegt **Main()** als Name der Methode fest.

Damit die wenig objektorientierten Beispiele Ihren Programmierstil nicht prägen, wurde zu Beginn des Kurses (im Kapitel 1) eine Anwendung vorgestellt, die bereits etliche OOP-Prinzipien realisiert.

Wie man im Visual Studio ein zum Üben von elementaren Sprachelementen geeignetes Konsolenprojekt erstellt, wurde im Abschnitt 3.3.5 beschrieben. Ein fertiges Übungsprojekt mit .NET 7 als **TargetFramework** (vgl. Abschnitt 3.3.8) ist hier zu finden:

…\BspUeb\Elementare Sprachelemente\Prog

---

[1] Nach einem beendeten Programmeinsatz unter Windows in einem per **cmd.exe** gestarteten Konsolenfenster befindet sich der **return**-Wert in der Pseudo-Umgebungsvariablen **errorlevel**.

4.1.2 Anweisungen auf oberster Ebene

Vermutlich finden Sie das `Hallo`-Beispiel allmählich langweilig, weil es bis auf den **WriteLine()** – Aufruf komplett determiniert ist. Seit C# 10 kann man sich tatsächlich auf den **WriteLine()** – Aufruf beschränken.

Dass man seit C# 10 in einer Quellcodedatei ohne explizite **using**-Direktiven auskommt, wurde schon im Abschnitt 2.6.3.2 erläutert.

Bereits seit C# 9 kann man es bei Konsolenprogrammen dem Compiler überlassen, eine Startklasse zu definieren und deren **Main()** – Methodenkopf zu verfassen, sodass sich der Programmierer nur noch um die Anweisungen der **Main()** – Methode kümmern muss, die in diesem Kontext als *Anweisungen auf oberster Ebene* (engl.: *top-level statements*) bezeichnet werden.

Dank dieser Vereinfachungsmöglichkeiten lässt sich das `Hallo`-Programm so formulieren:

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

Es wirkt etwas befremdlich oder hochstaplerisch, eine einzelne Anweisung als *Programm* zu bezeichnen. Aus dem Manuskript konnten im Vergleich zur letzten Ausgabe die folgenden Zeilen

```
using System;
class Prog {
   static void Main() {
   }
}
```

aus ca. 180 Beispielen gestrichen werden. Die verbliebenen, substanziellen Zeilen werden im Manuskript oft als *Programm* bezeichnet, manchmal mit dem Hinweis auf das seit C# 9/10 bei Konsolenprogrammen erlaubte Streichen von uninformativen Standardzeilen.

Wir haben im Abschnitt 4.1.1 das zum Erlernen von elementaren Sprachelementen geeignete, minimalistische C# - Konsolenprogramm *vollständig* dargestellt, damit die (vom Compiler im Hintergrund kompensierten) Auslassungen keine Unsicherheit bzgl. der obligatorischen Struktur eines C# - Programms zur Folge haben.

Wenn ein Beispielprogramm in der Datei **BspUeb.zip** enthalten ist (meist als Visual Studio – Projekt), dann erscheint es im Manuskript *inklusive* Klassendefinitionskopf, damit der Klassenname zu sehen ist, und das Auffinden der gleichnamigen Quellcodedatei erleichtert wird. Der **Main()** – Definitionskopf ist dann ebenfalls erforderlich, und aus dem Verzicht auf die mit C# 10 eingeführten *impliziten* **using**-Direktiven resultiert das traditionelle Erscheinungsbild eines C# - Konsolenprogramms.

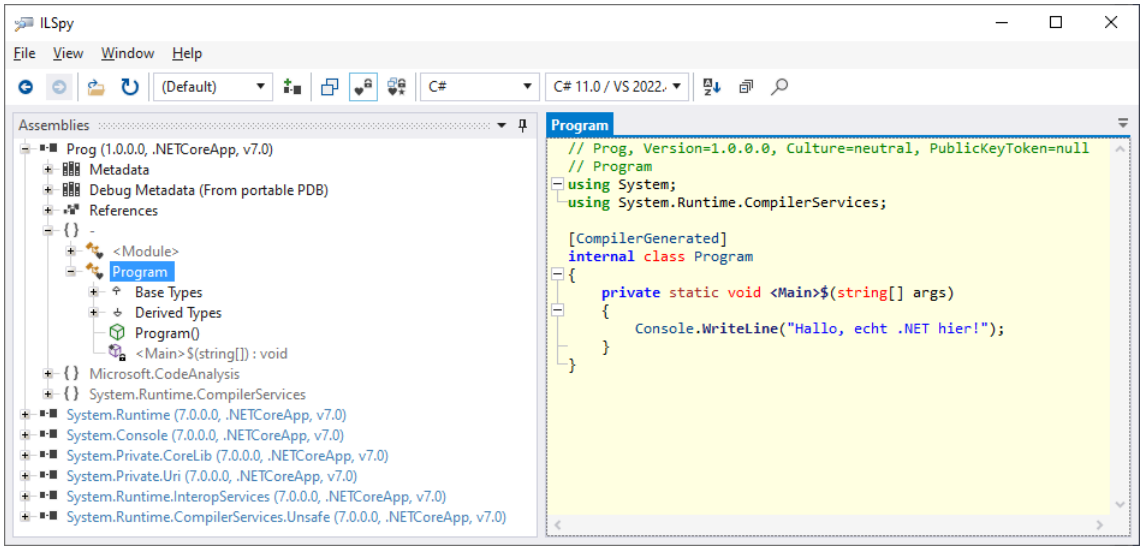
Für die Anweisungen auf oberster Ebene, die wir schon im Abschnitt 1.5 demonstriert haben, sind noch einige Regeln nachzutragen:[1]

- In einem Projekt darf die Technik der Anweisungen auf oberster Ebenen nur in *einer* Quellcodedatei genutzt werden. Die Verwendung der Technik in weiteren Quellcodedateien würde zur Existenz von *mehreren* **Main()** – Methoden führen.
- Explizite **using**-Direktiven sind in einer Quellcodedatei mit Anweisungen auf oberster Ebene nur am Dateianfang erlaubt.
- Die Anweisungen auf oberster Ebene befinden sich im globalen Namensraum.
- Nach den Anweisungen auf oberster Ebene sind **namespace**-Direktiven und Typdefinitionen erlaubt.

[1] https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/top-level-statements

- Vom Programmbenutzer übergebene Befehlszeilenargumente befinden sich in einer Variablen mit dem Typ **String[]** (Serie von Zeichenfolgen) und dem Namen **args**. Mit Befehlszeilenargumenten werden wir uns im Abschnitt 4.7.2.3.2 beschäftigen.
- Die nächste Regel wird der Vollständigkeit halber aufgelistet, obwohl man sie noch nicht verstehen kann: Der Aufruf von asynchronen Methoden über das Schlüsselwort **await** ist erlaubt. Die asynchrone Programmierung mit Hilfe der Schlüsselwörter **async** und **await** wird im Abschnitt 17.5 in Baltes-Götz (2021) behandelt.
- Um beim Beenden der Anwendung einen Exit Code an das Betriebssystem zu übergeben, darf man die **return**-Anweisung wie in einer **Main()** – Methode mit dem Rückgabetyp **int** verwenden (siehe Abschnitt 13.1).
- Der vom Compiler im Hintergrund erstellte **Main()** – Definitionskopf hängt davon ab, ob …
  - die **return**-Anweisung zum Beenden des Programms verwendet wird,
  - per **await**-Schlüsselwort eine asynchrone Methode aufgerufen wird.

Eine Inspektion das `Hallo`-Assemblies mit dem Diagnoseprogramm ILSpy



zeigt, dass der Compiler die weggelassenen Bestandteile in leicht geänderter Form eingefügt hat:

- Er hat den BCL-Namensraum **System** durch eine **using**-Direktive importiert.
- Er hat eine Startklasse mit dem Namen **Program** definiert und durch das Attribut
  `[CompilerGenerated]`
  dekoriert.[1]
- Er hat in der Klasse **Program** eine statische Methode mit dem ungewöhnlichen Namen **<Main>$()** definiert, der gegen die Benennungsregeln verstößt und folglich von uns nicht vergeben werden dürfte.[2]
- Weil das Programm keine **return**-Direktive zum Beenden enthält, hat die Methode **<Main>$()** den Rückgabetyp **void** erhalten.

---

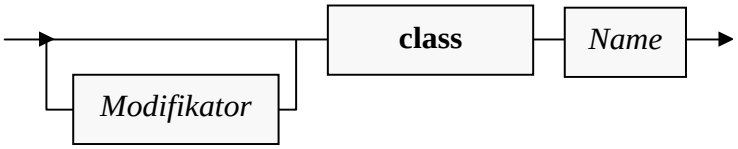
[1] Mit Attributen werden wir uns im Kapitel 14 beschäftigen.

[2] Die Verwendung eines **string[]** -Parameters, der für eine Serie von Zeichenfolgen steht, ist die bei **Main()** – Methoden generell erlaubte Alternative zur leeren Parameterliste.

4.1.3 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 in Pfeilrichtung durch das Syntaxdiagramm und gelangt dabei zu Rechtecken, die die an der jeweiligen Stelle zulässigen Sprachbestandteile angeben, wie z. B. im folgenden Syntaxdiagramm zum Kopf einer Klassendefinition:



- 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 an *kursiver* Schrift zu erkennen. Im konkreten Quellcode muss anstelle des Platzhalters eine zulässige Realisation stehen, und die zugehörigen Bildungsregeln sind an anderer Stelle (z. B. in einem anderen Syntaxdiagramm) erklärt.
- Bei einer Verzweigung kann man sich für eine Richtung entscheiden, wenn nicht per Pfeil eine Bewegungsrichtung vorgeschrieben ist. Zulässige Realisationen zum obigen Segment sind also z. B.:
  - `class Bruchaddition`
  - `public class Bruch`

  Verboten sind hingegen z. B. die folgenden Sequenzen:
  - `class public Bruchaddition`
  - `Bruchaddition public class`
- Als Klassenmodifikator ist uns bisher nur der Zugriffsmodifikator **public** begegnet, der für die allgemeine Verfügbarkeit einer Klasse (in beliebigen Assemblies) sorgt. Später werden Sie noch weitere Klassenmodifikatoren kennenlernen. 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.
- Bei den Syntaxdiagrammen im Manuskript wird angestrebt, dass sie möglichst übersichtlich sind und nur zulässige Syntax erlauben. Es ist hingegen *nicht* garantiert, dass jede erlaubte Syntax berücksichtigt ist.

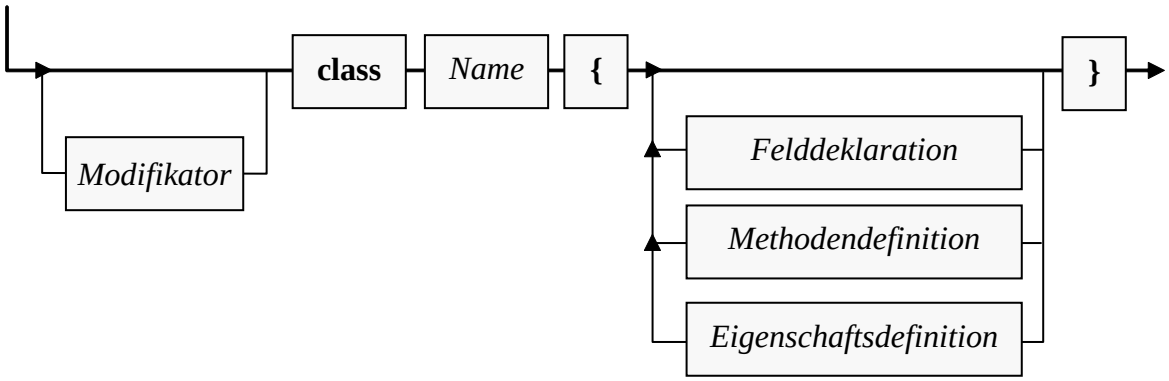
Als Beispiele betrachten wir anschließend die Syntaxdiagramme zur Definition von Klassen, Methoden und Eigenschaften. Aus didaktischen Gründen zeigen die Diagramme nur solche Sprachbestandteile, die bisher in einem Beispiel verwendet oder im Text beschrieben worden sind, sodass sie langfristig nicht 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 4.1.3 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.

4.1.3.1 Klassendefinition

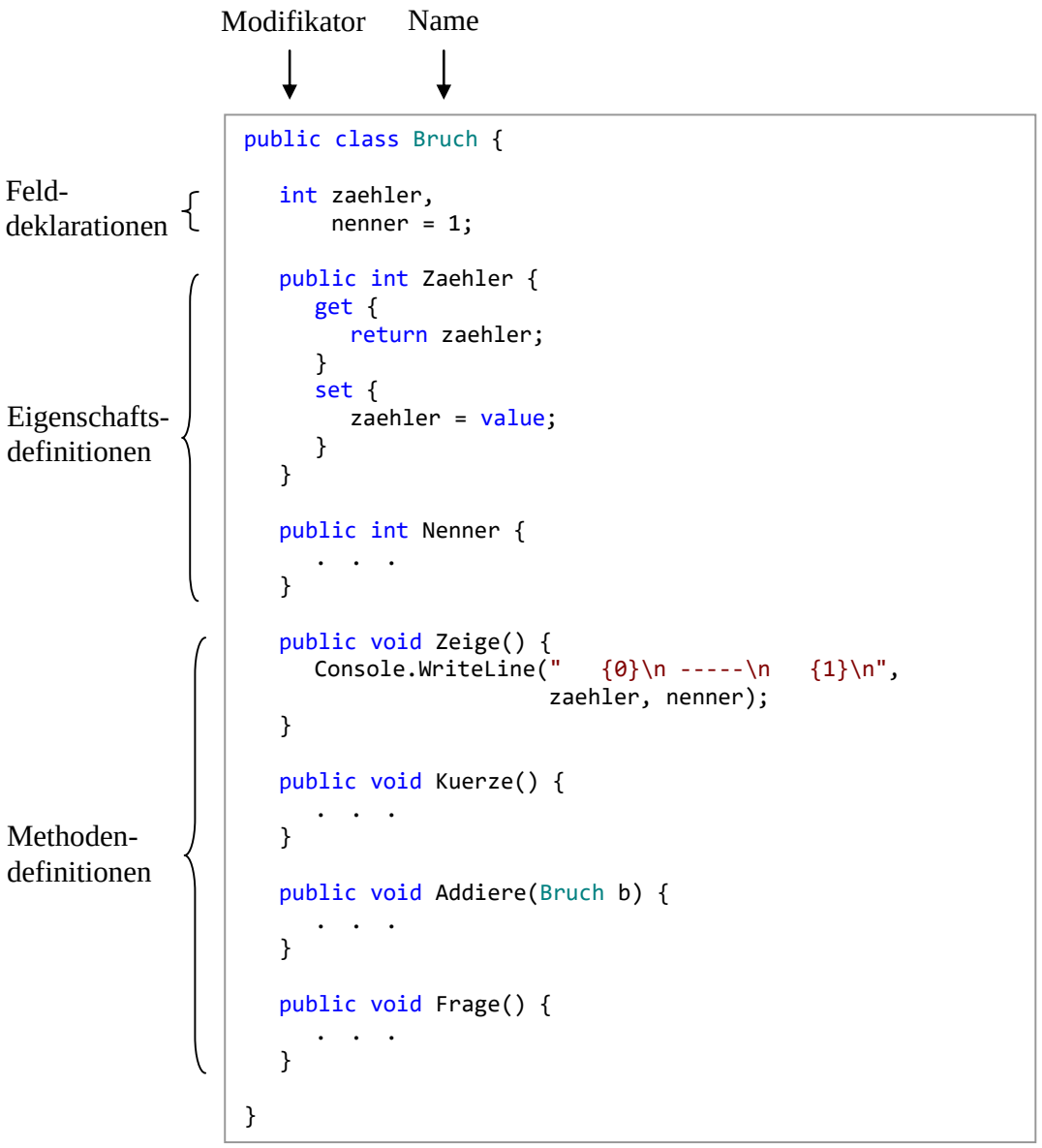
Wir arbeiten vorerst mit dem folgenden, leicht vereinfachten Klassenbegriff:

Klassendefinition



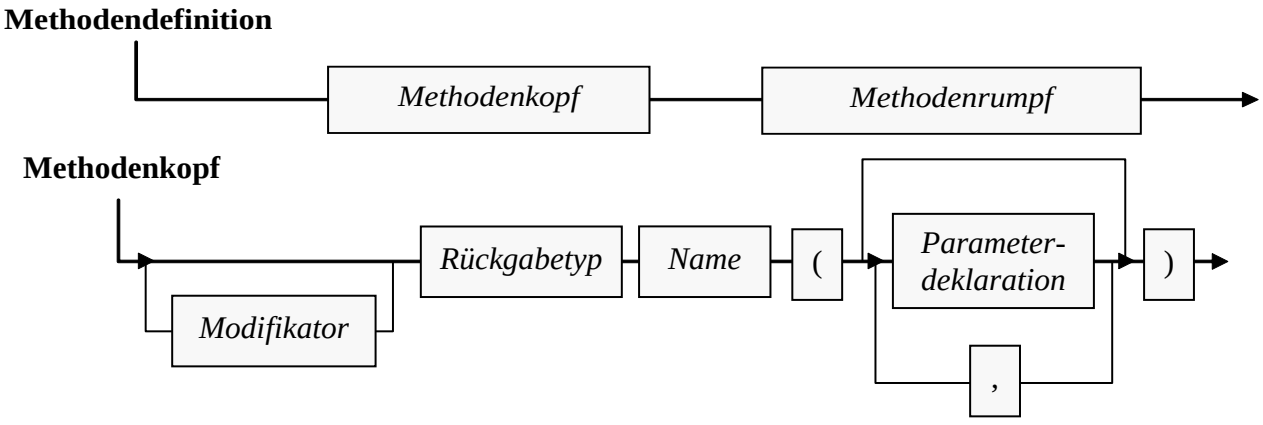
Solange man sich auf zulässigen Pfaden bewegt (immer in Pfeilrichtung, eventuell auch in Schleifen), an den Stationen (Rechtecken) entweder den konstanten Sprachbestandteil exakt übernimmt oder den Platzhalter auf zulässige (an anderer Stelle erläuterte) Weise ersetzt, entsteht eine syntaktisch korrekte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

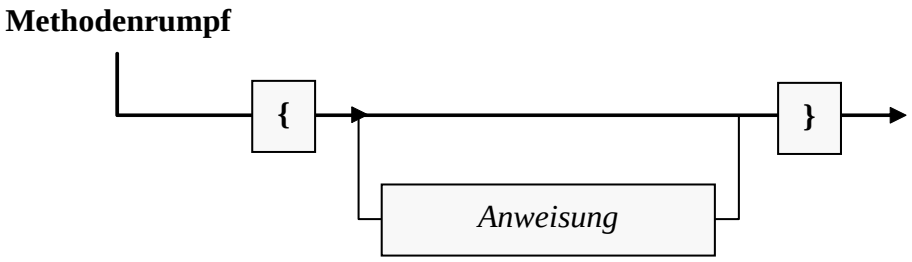
Als Beispiel betrachten wir die im Kapitel 1 vorgestellte Klasse `Bruch`:



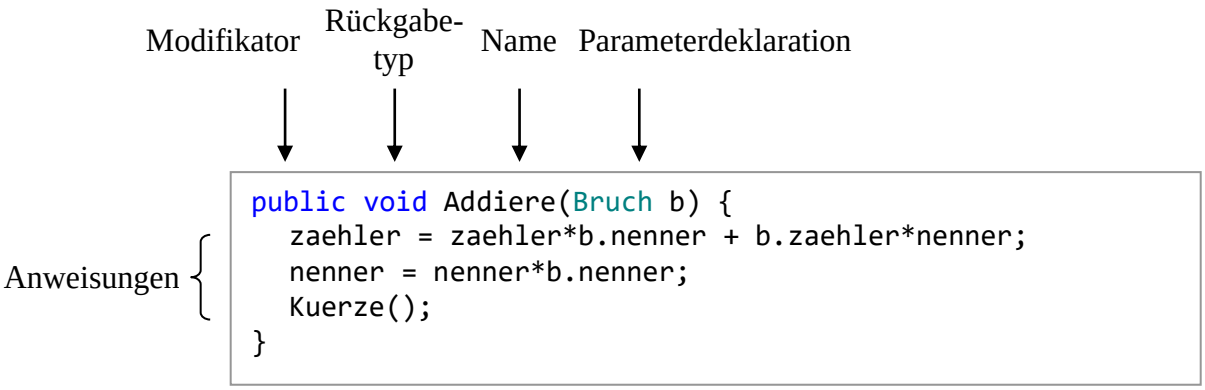
4.1.3.2 Methodendefinition

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





Als Beispiel aus der Klasse `Bruch` eignet sich vor allem die Methode `Addiere()`, die eine nichtleere Parameterliste besitzt:



Zur Erläuterung des Begriffs *Parameterdeklaration* beschränken wir uns vorläufig auf das Beispiel in der `Addiere()` - Definition. Es enthält:

- einen Datentyp (Klasse `Bruch`)
- und einen Parameternamen (`b`).

Dass in einer `Bruch`-Methodendefinition ein Parameter vom Typ `Bruch` verwendet wird, ist übrigens weder „zirkulär“ noch ungewöhnlich. Es ist vielmehr unvermeidlich, wenn `Bruch`-Objekte miteinander kooperieren sollen.

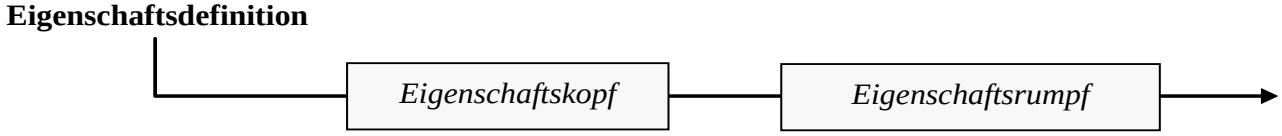
In vielen Methoden werden sogenannte *lokale Variablen* (vgl. Abschnitt 4.3.6) deklariert, z. B. in der `Bruch`-Methode `Kuerze()`:

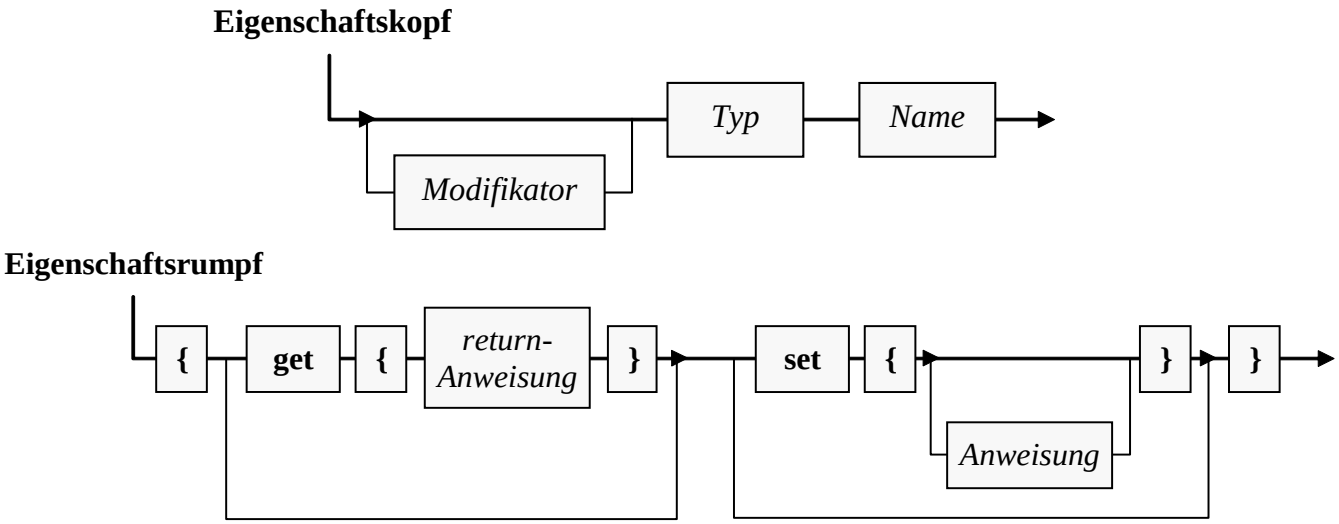
```
public void Kuerze() {
   if (zaehler != 0) {
      int az = Math.Abs(zaehler);
      int an = Math.Abs(nenner);
        .  .  .
   }
}
```

Weil wir bald u. a. die *Variablendeklarationsanweisung* kennenlernen werden, benötigt das Syntaxdiagramm zum Methodenrumpf jedoch (im Unterschied zum Klassendefinitionsdiagramm) *kein* separates Rechteck für die Variablendeklaration.

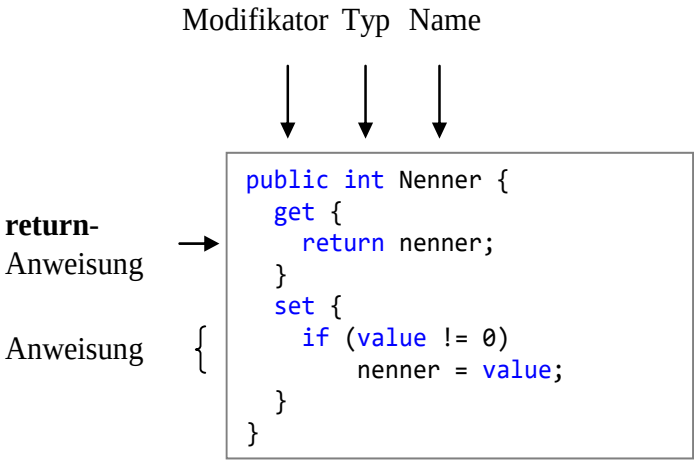
4.1.3.3 Eigenschaftsdefinition

Auch beim Syntaxdiagramm für den Eigenschaftsbegriff gehen wir schrittweise vor:





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



In der **set**-Definition spricht das Schlüsselwort **value** den Wert an, den der Aufrufer der Eigenschaft zuweisen möchte.

4.1.4 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.
- Schlüsselwörter und Namen (siehe Abschnitt 4.1.6) müssen durch **Trennzeichen** separiert werden, wobei das Leerzeichen, das Tabulatorzeichen und der Zeilenumbruch erlaubt sind. Vermutlich käme niemand auf die Idee, z. B. den Kopf der `Addiere()` – Methode in der Klasse `Bruch` unter Missachtung der Trennungsregel so zu formulieren:
  ```
  publicvoidAddiere(Bruch b) { ... }
  ```
  Die Trennzeichen dürfen sogar in beliebigen Anzahlen und Kombinationen auftreten. *Innerhalb* eines Sprachbestandteils (z. B. Namens) sind Trennzeichen 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 Quellcode-Beispiele. Es handelt sich um die Syntaxhervorhebungen der Entwicklungsumgebung Visual Studio, die via Windows-Zwischenablage in den Text übernommen wurden.[1]

Manche Programmierer setzen die öffnende geschweifte Klammer zum Rumpf einer Klassen-, Methoden- oder Eigenschaftsdefinition ans Ende der Kopfzeile (siehe linkes Beispiel), andere bevorzugen den Anfang der Folgezeile (siehe rechtes Beispiel):

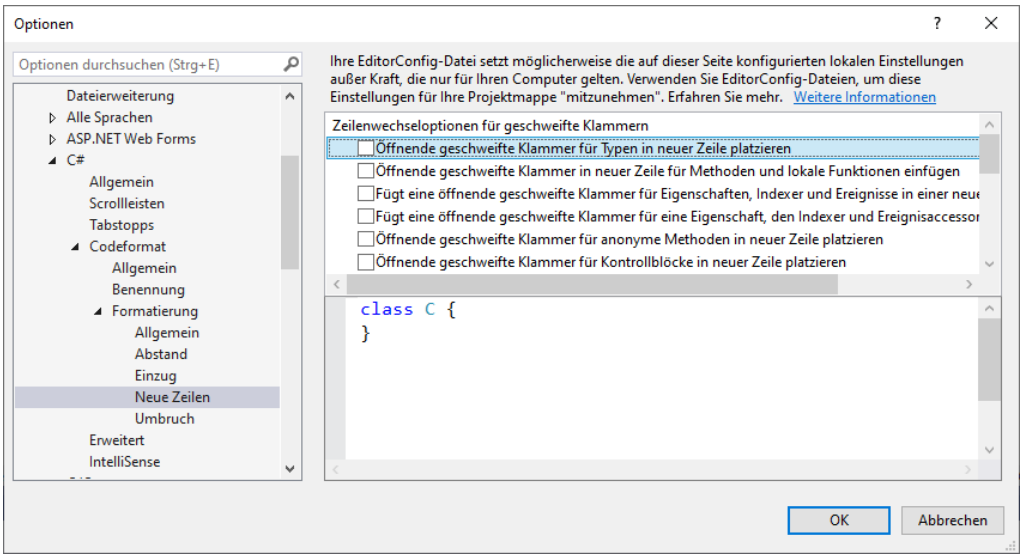
```
class Hallo {
  static void Main() {
    System.Console.WriteLine("Hallo");
  }
}
```

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

Das Visual Studio verwendet per Voreinstellung die *rechte* Variante, kann aber nach

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

umgestimmt werden:



Im Manuskript wird die Platz sparende *linke* Variante bevorzugt.

Weitere Hinweise zur übersichtlichen Gestaltung des C# - Quellcodes finden sich z. B. in Microsofts Online-Informationsangebot zu C#.[2]

---

1 Über **Extras > Optionen > Text-Editor > C# > Erweitert** kann man im Visual Studio beim **Editor-Farbschema** zwischen den Optionen **2017** und **2019** wählen. Leider gehen beim Transfer von Quellcode über die Windows-Zwischenablage einige Farben verloren. Beim Farbschema 2017 sind die Verluste weniger gravierend, sodass diese Variante für das Manuskript gewählt wurde. Nichtsdestotrotz musste die verlorene Cyan-Färbung von Typnamen manuell restauriert werden, was vermutlich nicht vollständig gelungen ist.

2 https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

4.1.5 Kommentar

C# bietet mehrere Möglichkeiten zum Kommentieren von Quellcode:

- **Zeilenrestkommentar**
  Alle Zeichen vom ersten doppelten Schrägstrich (**//**) bis zum Ende der Zeile gelten als Kommentar, z. B.:

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

  Im Beispiel wird eine Variablendeklarationsanweisung in derselben Zeile kommentiert.

- **Explizit terminierter, meist mehrzeiliger Kommentar**
  Ein durch **/*** eingeleiteter Kommentar muss explizit durch ***/** terminiert werden. In der Regel wird diese Syntax für einen mehrzeiligen Kommentar verwendet, z. B.:

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

  Ein mehrzeiliger Kommentar eignet sich 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 */}
```

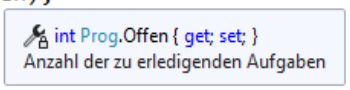
  Weil der explizit terminierte Kommentar (jedenfalls ohne farbliche Hervorhebung der auskommentierten Passage) etwas unübersichtlich ist, wird er selten verwendet.

- **Dokumentationskommentar**
  Neben den Kommentaren, die das Verständnis des Quellcodes unterstützen sollen, kennt C# noch den Dokumentationskommentar. Er 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>
/// Anzahl der zu erledigenden Aufgaben
/// </summary>
public int Offen { get; set; }
```

  Der Inhalt eines **summary**-Elements wird von Entwicklungsumgebungen als Quick-Info - Zusatzinformation angezeigt, z. B.:

```
Console.WriteLine(p.Offen);
```



  Durch **/**** eingeleitete und durch ***/** terminierte *mehrzeilige* Dokumentationskommentare erfordern die Beachtung spezieller Regeln und sind wegen des damit verbundenen Fehlerrisikos *nicht* zu empfehlen.
  Das folgende **PropertyGroup**-Element in der Projektdatei

```
<GenerateDocumentationFile>True</GenerateDocumentationFile>
```

  sorgt dafür, dass Dokumentationskommentare bei der Programmerstellung in eine separate XML-Dokumentationsdatei geschrieben werden. Bei einem Projekt namens `Prog` entsteht im Ausgabeordner die Datei **Prog.xml**. Im Visual Studio 2022 fordert man die Erstellung der XML-Dokumentationsdatei folgendermaßen an:
  Projekt > Eigenschaften > Build > Ausgabe > Dokumentationsdatei

Eine XML-Dokumentationsdatei dient nicht nur als Quick-Info – Quelle, sondern auch als Basis für die Erstellung einer leicht lesbaren Dokumentation (z. B. im HTML-Format). In Microsofts Entwicklungsumgebungen fehlt leider eine entsprechende Funktionalität.[1]

Wenn man im Visual Studio - Editor das Auskommentieren eines markierten Blocks mit dem Menübefehl

Bearbeiten > Erweitert > Auswahl auskommentieren

bzw. mit der Tastenkombination **Strg+K**, **Strg+C** veranlasst, dann werden doppelte Schrägstriche vor jede Zeile gesetzt.[2] Ist allerdings nur ein Teil *einer* Zeile markiert, dann bewirkt die Tastenkombination **Strg+K**, **Strg+C** einen explizit terminierten Kommentar.

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, dann werden die Kommentarschrägstriche entfernt.

4.1.6 Namen

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

- Die Länge eines Namens ist nicht begrenzt.
  Zwar fördern kurze Namen die Übersicht im Quellcode, doch ist die Verständlichkeit eines Namens noch wichtiger als die Kürze.
- 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 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.

  ```
  Anzahl   anzahl   ANZAHL
  ```

  grundverschiedene Namen.

---

[1] Ein ursprünglich von Microsoft zur Dokumentationserstellung entwickeltes Programm namens *Sandcastle* wird als GitHub-Projekt weiterentwickelt:

https://github.com/EWSoftware/SHFB

[2] Mit der Tastenkombination **Strg+K**, **Strg+C** ist gemeint:

- **Strg**-Taste drücken und festhalten
- Taste **K** drücken und loslassen
- Taste **C** drücken und loslassen
- **Strg**-Taste loslassen

- Die folgenden **reservierten Schlüsselwörter** dürfen nicht als Namen verwendet werden:[1]

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();
```

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

add	and	alias	args	ascending	async	await	by	descending
dynamic	equals	file	from	get	global	group	init	into
join	let	managed	nameof	nint	not	notnull	nuint	on
or	orderby	partial	record	remove	required	scoped	select	set
unmanaged	value	var	when	where	with	yield		

  Vorsichtshalber sollte man sie generell als Namen vermeiden.

- Namen müssen im aktuellen Kontext (siehe Abschnitt 4.3.8) eindeutig sein.

Während Sie obige Regeln einhalten *müssen*, ist die Beachtung der folgenden Konventionen freiwillig, aber empfehlenswert (vgl. Mössenböck 2019, S. 14):

- Die Namen von *lokalen* (methodeninternen) Variablen (siehe Abschnitt 4.3.3), Methodenparametern und *privaten* (gekapselten, nur klassenintern ansprechbaren) Feldern werden *klein* geschrieben, z. B.:

`az`	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 Buchstaben, z. B.:

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

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/index

[2] https://msdn.microsoft.com/en-us/library/the35c6y.aspx, https://msdn.microsoft.com/en-us/library/bb310804.aspx

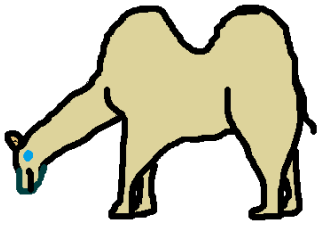
- Bei zusammengesetzten Namen beginnt jedes Wort mit einem Großbuchstaben (*Pascal Casing*), z. B.:[1]

  ```
  WriteLine()
  ```

  Eine Ausnahme stellen die nach obiger Empfehlung mit einem Kleinbuchstaben 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:

  Tritt eine aus zwei Buchstaben bestehende Abkürzung als Namensbestandteil auf, dann werden bei Verwendung von Pascal Casing *beide* Buchstaben großgeschrieben, z. B.:

  ```
  DBConnector
  ```

  Bei Verwendung von Camel Casing werden beide Buchstaben klein geschrieben, z. B.:

  ```
  dbConnector
  ```

  Gelegentlich wird bei zusammengesetzten Namen der Unterstrich zur Verbesserung der Lesbarkeit verwendet, was der folgende Methodenname demonstriert, den ein Visual Studio – Assistent im `DmToEuro`-Beispielprogramm erstellt hat (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**):

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

  Das Visual Studio schlägt den Unterstrich insbesondere in den Namen von Ereignisbehandlungsmethoden vor, um die Ereignisquelle von der Ereignisklasse abzugrenzen (siehe Beispiel).

- Einige Programmierer verwenden …
  - einen Unterstrich als Präfix für die Namen von Feldern mit der Schutzstufe **private** (verfügbar in der eigenen Klasse) oder **internal** (verfügbar im eigenen Assembly), z. B.:

    ```
    private int _nr;
    ```

  - das Präfix **s_** für die Namen von statischen Feldern mit der Schutzstufe **private** oder **internal**, z. B.:

    ```
    private static int s_count;
    ```

Auf der folgenden Webseite gibt Microsoft detaillierte Empfehlungen zur Verwendung von Namen im C# - Quellcode:

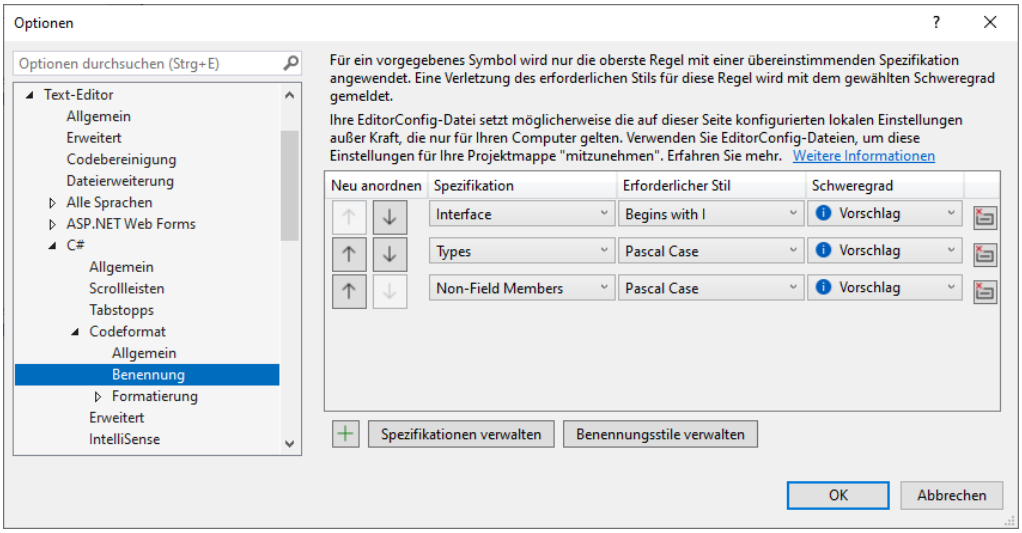
https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions#naming-conventions

Im Visual Studio ist über

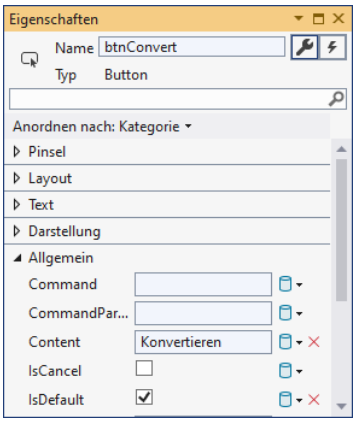
Extras > Optionen >Text-Editor > C# > Codeformat > Benennung

[1] Ob diese Benennungskonvention auf die altehrwürdige Programmiersprache Pascal zurückgeht, lässt sich nicht eindeutig belegen. Wir beschäftigen uns nicht weiter mit der etymologischen Frage (nach der Wortherkunft).

ein Einstellungsdialog mit Benennungsregeln erreichbar, die zur Code-Beurteilung dienen:



Lässt man den Assistenten für WPF-Projekte zu einem **Button**-Objekt, das durch einen klein geschriebenen Feldnamen ansprechbar ist,



eine **Click**-Behandlungsmethode erstellen, dann resultiert ein Methodenname, der mit einem Kleinbuchstaben startet. Aufgrund der dritten Regel im obigen Dialog kritisiert anschließend das Visual Studio seine Produktion (erkennbar an einer punktierten Unterstreichung):

```
private void btnConvert_Click(object sender, RoutedEventArgs e) {

}
```

4.1.7 Übungsaufgaben zum Abschnitt 4.1

1) Welche von den folgenden Methoden sind zum Starten eines Programms geeignet?

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

Wer die Technik der Anweisungen auf oberster Ebene benutzt (siehe Abschnitt 4.1.2), delegiert die korrekte Formulierung des **Main()** – Methodenkopfs an den Compiler.

2) Welche von den folgenden Namen sind unzulässig?

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

4.2 Ausgabe bei Konsolenanwendungen

Eine formatierte Konsolenausgabe lässt sich in C# recht bequem über die Methode **Console.WriteLine()** erzeugen. Das folgende Beispiel stammt aus der Methode `Zeige()` unserer Klasse `Bruch` (siehe Abschnitt 1.3):

```
Console.WriteLine($"   {zaehler}\n  -----\n   {nenner}\n");
```

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

- Der Namensraum **System** muss (explizit oder implizit) 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 Spezialisierungen der **WriteLine()** - Methode (später werden wir von *Überladungen* sprechen):

- Im obigen Beispiel ist die *formatierte* Ausgabe über die sogenannte *Zeichenfolgeninterpolation* zu sehen (vgl. Abschnitt 4.2.2.2).
- Oft reicht die im Abschnitt 4.2.1 behandelte Ausgabe einer (zusammengesetzten) Zeichenfolge.

4.2.1 Ausgabe einer (zusammengesetzten) Zeichenfolge

Im `Hallo`-Beispiel (vgl. Abschnitt 3.1.3) 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 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()** - Ausgabe recht flexibel. Außerdem erlauben die folgenden **Escape-Sequenzen** (vgl. Abschnitt 4.3.10.4), die wie gewöhnliche Zeichen in die Ausgabezeichenfolge geschrieben werden, eine Gestaltung der Ausgabe:

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

Beispiel:

Quellcode	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.

4.2.2 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** oder per Zeichenfolgeninterpolation (siehe Abschnitt 4.2.2.2) formatierte Zeichenfolgen erzeugen, die z. B. im Rahmen einer grafischen Bedienoberfläche zum Einsatz kommen (siehe Übungsaufgabe im Abschnitt 4.3.11).

4.2.2.1 Traditionelle Variante mit Platzhaltern

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

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

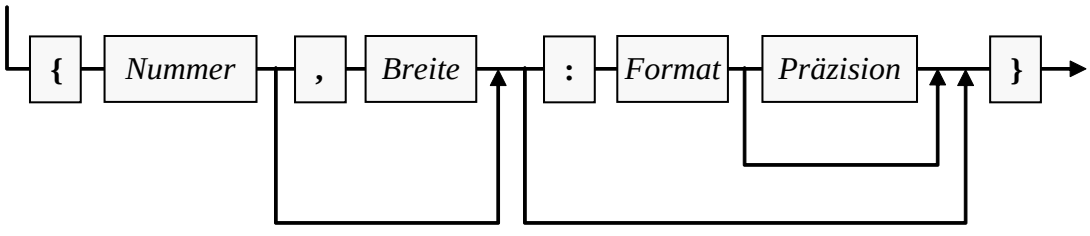
Für ein Objekt der Klasse `Bruch` mit dem Zähler 5 und dem Nenner 8 resultiert bei dieser Variante der Methode `Zeige()` die Ausgabe:

```
  5
-----
  8
```

Der einleitende Zeichenfolgenparameter beschreibt, wie die beiden restlichen Parameter auszugeben sind.

Für einen Platzhalter in der Formatierungszeichenfolge ist die folgende Syntax vorgeschrieben:

Platzhalter für die formatierte Ausgabe



Darin bedeuten:

Nummer	Fortlaufende Nummer des auszugebenden Parameters, bei 0 beginnend
Breite	Ausgabebreite für den Parameter 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)

Bei der Ausgabe von Zahlen werden u. a. die folgenden Formate unterstützt:[1]

Format	**Beschreibung**	**Beispiele mit den Variablen** `int i = 32483; float f = 21.415926f;`	
		WriteLine-Parameterliste	Ausgabe
d, D	**Ganze Dezimalzahl**	`("{0,7:d}", i)`	`32483`
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)`	`32483` `21,41593`

In der Formatierungszeichenfolge sind auch auszugebende gewöhnliche Zeichen und Escape-Sequenzen (vgl. Abschnitt 4.3.10.4) erlaubt:

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

Beispiel:

Quellcode

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

Ausgabe

```
Feste Breite:
      47
    1771
   3,142

Tabulatoren:
        47
        1771
        3,1415926
```

Auf eine Formatierungszeichenfolge mit *k* verschiedenen Platzhalternummern müssen entsprechend viele Ausdrücke (z. B. Variablen) mit einem zum jeweiligen Platzhalterformat passenden Datentyp folgen. Eine Platzhalternummer darf in der Formatierungszeichenfolge mehrfach auftreten, wenn der entsprechende Parameter mehrfach auszugeben ist.

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

Quellcode

```
int i = 47, j = 1771;
Console.WriteLine("{{{0}, {1}}}", i, j);
```

Ausgabe

```
{47, 1771}
```

[1] Hier dokumentiert Microsoft die Formatzeichenfolgen für Zahlen:
https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings
Hier dokumentiert Microsoft die Formatzeichenfolgen für Datum und Uhrzeit:
https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-date-and-time-format-strings

4.2.2.2 Zeichenfolgeninterpolation

Seit der C# - Version 6.0 erlaubt die sogenannte *Zeichenfolgeninterpolation* eine Vereinfachung bei der formatierten Ausgabe. Statt die formatiert auszugebenden Ausdrücke *hinter* die Formatierungszeichenfolge zu schreiben, setzt man sie *in* die Zeichenfolge an Stelle der bisher verwendeten Platzhalternummern. Damit der Compiler die neue Syntax von der traditionellen unterscheiden kann, muss ein **$**-Zeichen als Präfix vor die Formatierungszeichenfolge gesetzt werden. Im Beispielprogramm aus dem letzten Abschnitt kann der **WriteLine()** - Aufruf mit der neuen Technik übersichtlicher formuliert werden:

Quellcode	Ausgabe

`int i = 47, j = 1771;`
`double d = 3.1415926;`
`Console.WriteLine($"Feste Breite:\n{i,8}\n{j,8}\n{d,8:f3}" +`
`$"\n\nTabulatoren:\n\t{i}\n\t{j}\n\t{d}");`
	`Feste Breite:`
	`  47`
	`1771`
	`3,142`
	
	`Tabulatoren:`
	`  47`
	`  1771`
	`  3,1415926`

Eine interpolierte Zeichenfolge darf einer **String**-Variablen zugewiesen und später beliebig verwendet werden, z. B.:

```
String s = $"Feste Breite:\n{i,8}\n{j,8}\n{d,8:f3}" +
                  $"\n\nTabulatoren:\n\t{i}\n\t{j}\n\t{d}";
Console.WriteLine(s);
```

Die durch ein Paar geschweifter Klammern begrenzten, formatiert auszugebenden Ausdrücke werden als *Interpolationsausdrücke* (engl. *interpolation expressions*) bezeichnet, z. B.:

```
{d,8:f3}
```

Seit C# 11 darf ein Interpolationsausdruck auch Leerzeilen enthalten, sodass eine übersichtliche Darstellung von komplexen Ausdrücken möglich ist. Im folgenden Beispiel wird ein **switch**-Ausdruck mit Musterabgleich in einen Interpolationsausdruck integriert (siehe Abschnitt 4.7.2.4 zum **switch**-Ausdruck und Abschnitt 15.2.7 zum Relationsmuster):[1]

```
int gewicht = 1500;
Console.WriteLine($"Es sind {gewicht switch {
                                    <= 1000 => 10.0m,
                                    <= 2000 => 18.5m,
                                    <= 7500 => 25.2m,
                                    _       => 30.8m
                                  }:f2} Euro als Maut zu entrichten.");
```

4.2.3 Übungsaufgaben zum Abschnitt 4.2

1) Wie ist das fehlerhafte „Rechenergebnis“ im folgenden Beispiel zu erklären?

---

[1] Im Beispiel (nach https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated) besteht allerdings kein zwingender Grund für die Aufnahme des Musterabgleichs in den Interpolationsausdruck. Mit einer Hilfsvariablen wird das Beispiel übersichtlicher:

```
int gewicht = 1500;
decimal maut = gewicht switch {
    <= 1000 => 10.0m,
    <= 2000 => 18.5m,
    <= 7500 => 25.2m,
    _       => 30.8m
};
Console.WriteLine($"Es sind {maut:f2} Euro als Maut zu entrichten.");
```

Quellcode	Ausgabe
`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 im Abschnitt 4.5.10 behandeln werden.

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

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

mit *einem* **WriteLine()** - Aufruf diese Ausgabe produziert:

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

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

4.3 Variablen und Datentypen

Während ein Programm läuft, müssen zahlreiche Daten im Arbeitsspeicher des Rechners abgelegt werden und anschließend mehr oder weniger lange für lesende und schreibende Zugriffe verfügbar sein, z. B.:

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

Zum Speichern eines Werts (z. B. einer ganzen Zahl) wird eine sogenannte **Variable** verwendet, worunter Sie sich einen **benannten Speicherplatz für einen Wert mit einem bestimmten Datentyp** (z. B. Ganzzahl) vorstellen können.

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

```
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 4.1.6 erlaubt.
- **Datentyp**
  Damit sind festgelegt: Zulässige Werte (hinsichtlich Art und Größe), Speicherplatzbedarf, 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 Speicher*regionen* unterscheiden (*Stack* und *Heap*). Dieses Hintergrundwissen hilft z. B., wenn eine **StackOverflowException** gemeldet wird.

4.3.1 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.

4.3.1.1 Explizite Deklaration

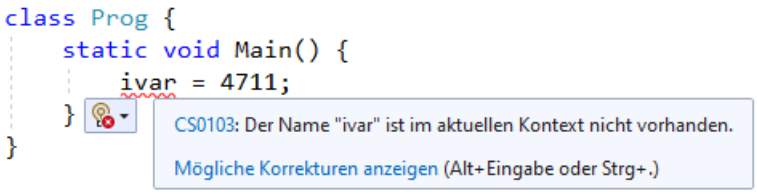
Variablen müssen **explizit deklariert** werden. In der folgenden Anweisung

```
int ivar;
```

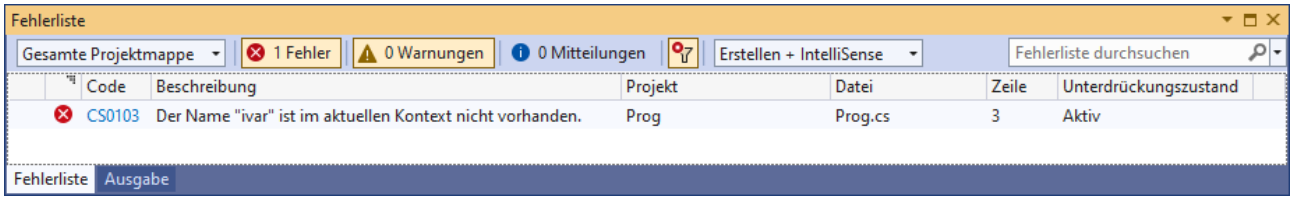
wird die Variable `ivar` vom Typ **int** (zum Speichern einer ganzen Zahl) deklariert. Wenn Sie versuchen, eine nicht deklarierte Variable zu verwenden, beschwert sich der über das Kommando **dotnet build** (vgl. Abschnitt 3.1.4) beauftragte Compiler in einem Konsolenfenster, z. B.:

```
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 (z. B. angefordert über **Strg+F5**), dann erscheint die Compiler-Reklamation in der **Fehlerliste**:



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

VB.NET - Quellcode	Ausgabe

`Option Explicit Off`
 	
`Module Program`
`  Sub Main()`
`  i1 = 12  ' Die Variable i1 wird ohne Deklaration verwendet.`
`  il = i1 + 1  ' Der Tippfehler fällt nicht auf (l statt 1).`
`  Console.WriteLine(i1)  ' Die Variable i1 hat einen falschen Wert.`
`  End Sub`
`End Module`
	`12`

4.3.1.2 Statische Typisierung

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

- welche Art von Daten (z. B. ganze Zahlen, Gleitkommazahlen, 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 einem unpassenden 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 geeignet ist (siehe Abschnitt 4.3.4).

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

- Kooperation mit typfreien bzw. dynamisch typisierten Sprachen wie z. B. JavaScript
- Nutzung von COM (*Component Object Model*) - APIs (z. B. zur Office-Automatisierung)
- Zugriffe auf das HTML-Dokumentobjektmodell (DOM)

Anstelle des Compilers ist hier die CLR für die Typprüfung verantwortlich, und Fehler werden zur Laufzeit entdeckt. Das ist ärgerlich für die Benutzer und peinlich für die Entwickler. Der Datentyp **dynamic** sollte nur in begründeten Ausnahmefällen zum Einsatz kommen.[2]

4.3.1.3 Initialisierung

In der folgenden Anweisung

```
int ivar = 4711;
```

erhält die Variable `ivar` bei der Deklaration 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 4.3.6). Weil ein Verstoß gegen diese Regel vom C# - Compiler verhindert wird, und weil die zu einem Objekt oder

---

[1] Halten Sie bitte die *statische Typisierung* (im Sinn von *unveränderlicher* Typfestlegung zur Übersetzungszeit) in begrifflicher Distanz zu den bereits erwähnten *statischen Variablen* (im Sinn von *klassenbezogenen* Variablen). Das Wort *statisch* ist eingeführter Bestandteil bei beiden Begriffen, sodass es nicht sinnvoll erscheint, zur Vermeidung der Doppelbedeutung eine andere Bezeichnung vorzunehmen.

[2] Hier finden Sie die Online-Dokumentation zum Typ **dynamic**:
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types#the-dynamic-type

zu einer Klasse gehörenden Variablen (siehe unten) automatisch initialisiert werden, hat in C# jede Variable beim Lesezugriff stets einen definierten Wert.

4.3.2 Wert- und Referenztypen

Bei der objektorientierten Programmierung werden neben den traditionellen Variablen zur Aufbewahrung von Zahlen, Zeichen oder Wahrheitswerten auch Variablen benötigt, die die Speicheradresse eines Objekts aufnehmen und die Kommunikation mit diesem Objekt ermöglichen.

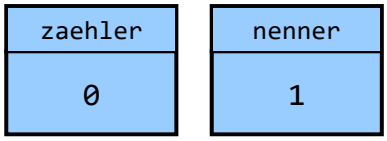
4.3.2.1 Werttypen

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 Methoden), obwohl sie „nur“ zur Verwaltung ihres Inhalts dienen und keine Rolle bei der Kommunikation mit Objekten spielen.

In der `Bruch`-Klassendefinition (siehe Abschnitt 1.3) 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 expliziten Initialisierungswert erhält:

```
int zaehler, nenner = 1;
```

Beim Feld `zaehler` wird auf die explizite Initialisierung verzichtet, sodass die automatische Null-Initialisierung von Feldern greift. Für ein frisch erzeugtes `Bruch`-Objekt befinden sich im Arbeitsspeicher die folgenden Instanzvariablen (Felder):



In der `Bruch`-Methode `Kuerze()` tritt u. a. die lokale Variable `az` auf, die ebenfalls den Werttyp **int** besitzt:

```
int az = Math.Abs(zaehler);
```

Wie Sie aus dem Abschnitt 4.3.1.3 wissen, ist bei *lokalen* (innerhalb einer Methode deklarierten) Variablen vor dem ersten Lesezugriff eine Initialisierung erforderlich. Im Beispiel findet die gleich bei der Deklaration statt (siehe Abschnitt 4.3.6).

4.3.2.2 Referenztypen

Eine Variable mit Referenztyp nimmt die **Speicheradresse eines Objekts** aus einer bestimmten Klasse auf. Sobald ein solches Objekt erzeugt und seine Speicheradresse der Referenzvariablen zugewiesen worden ist, kann das Objekt über die Referenzvariable angesprochen werden. Von einer Variablen mit Werttyp unterscheidet sich eine Referenzvariable also …

- durch ihren speziellen Inhalt (Objektadresse)
- und durch ihre Rolle bei der Kommunikation mit einem Objekt.

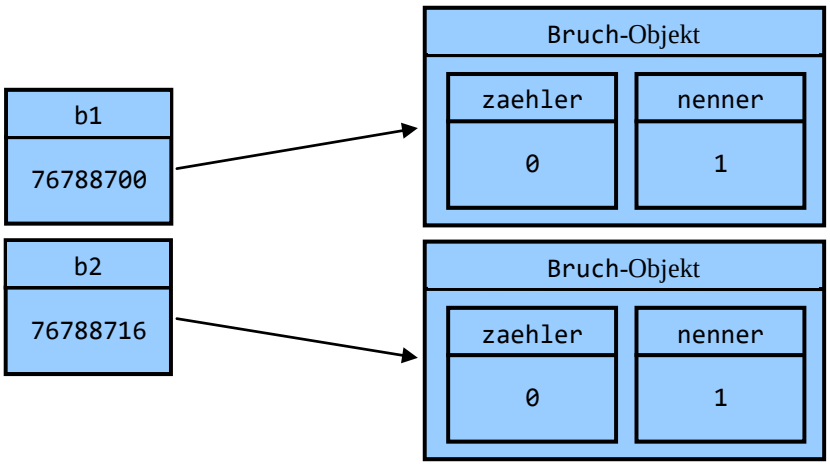
Man kann jede Klasse (aus der BCL übernommen oder selbst definiert) als Datentyp verwenden, also Referenzvariablen von diesem Typ deklarieren.[1] In der **Main()** - Methode der im Abschnitt 1.5 vorgestellten Klasse `Bruchaddition` werden z. B. die Referenzvariablen `b1` und `b2` aus der Klasse `Bruch` deklariert:

---

[1] Hier wird aus didaktischen Gründen etwas gemogelt: Die später kurz erwähnten statischen Klassen lassen sich *nicht* als Datentyp verwenden (siehe Abschnitt 5.7.5).

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

Sie erhalten als Initialisierungswert jeweils eine Referenz auf ein (per **new**-Operator, siehe Abschnitt 5.4.2) 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 untergebracht. Wir müssen diese Adresse nicht kennen, sondern sprechen das dort abgelegte Objekt über die Referenzvariable 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 *Referenzvariable* 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). Nach einer späteren Erweiterung der `Bruch`-Klassendefinition werden ihre Objekte auch ein Feld mit Referenztyp besitzen.
- Eine Referenzvariable dient zur Aufnahme einer Objektadresse.
  So kann z. B. eine Variable vom Datentyp `Bruch` die Adresse eines `Bruch`-Objekts aufnehmen und zur Kommunikation mit diesem Objekt dienen. 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 (und damit zum potenziellen Opfer des im Abschnitt 2.5 erwähnten Garbage Collectors), wenn im gesamten Programm keine einzige Referenz (Kommunikationsmöglichkeit) mehr vorhanden ist. Referenzvariablen werden als Felder von Klassen und als lokale Variablen von Methoden oder Eigenschaften benötigt.

Während eine Variable mit Werttyp „nur“ ein benannter Ort im Speicher zur Aufnahme eines Werts von bestimmtem Typ ist, kommt bei einer Referenzvariablen im arbeitsfähigen Zustand noch das (ebenfalls im Speicher befindliche) Objekt einer bestimmten Klasse hinzu.

Wir betrachten Objekte momentan bevorzugt abstrakt-metaphorisch, doch sind auch Details der technischen Realisation von Interesse. Über den Speicherbereich zur Aufnahme von Objekten wird gleich berichtet. Weitere technische Details verschwinden aus didaktischen Gründen vorläufig in einer Fußnote.[1]

---

[1] Neben den im Abschnitt 4.3.2.2 abgebildeten Instanzvariablen enthält ein Objekt auch Metadaten:

4.3.3 Klassifikation von Variablen nach der 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 oder Eigenschaft deklariert. Ihre Gültigkeit beschränkt sich auf die Methode bzw. Eigenschaft, genauer: auf einen Anweisungsblock innerhalb der Methode bzw. Eigenschaft (siehe Abschnitt 4.3.8).

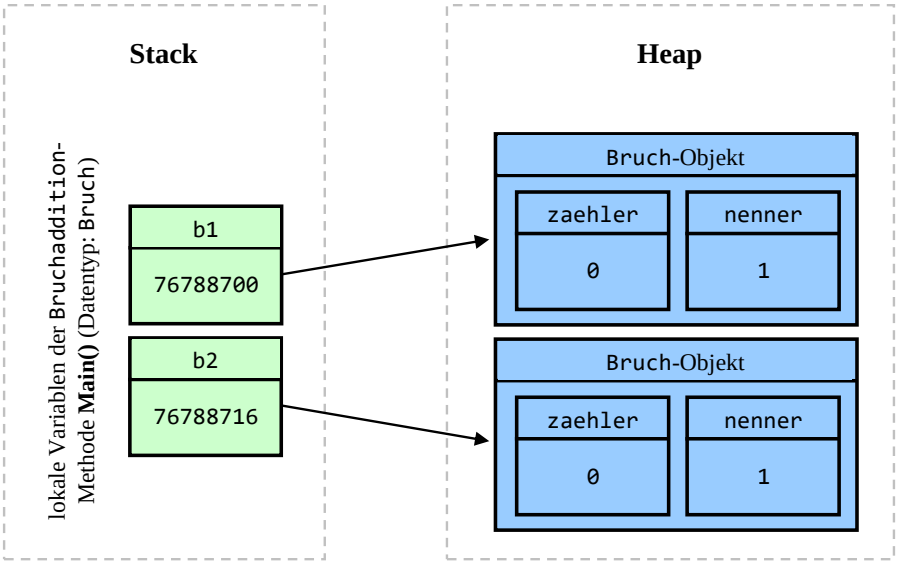
  Solange eine Methode bzw. Eigenschaft ausgeführt wird, befinden sich ihre Variablen in einem Speicherbereich, den man als **Stack** (dt.: *Stapel*) bezeichnet. Die Abbildung im Abschnitt 4.3.2.2 zeigt die lokalen Variablen `b1` und `b2` aus der **Main()** - Methode der Klasse `Bruchaddition`, die als Referenzvariablen auf Objekte der Klasse `Bruch` zeigen.
- **Instanzvariablen (nicht-statische Felder)**
  Instanzvariablen werden außerhalb jeder Methode bzw. Eigenschaft deklariert. 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`.

  Solange ein Objekt existiert, befinden es sich mit all seinen Instanzvariablen in einem Speicherbereich, den man als **Heap** (dt.: *Haufen*) bezeichnet.
- **Klassenvariablen (statische Felder)**
  Klassenvariablen werden außerhalb jeder Methode bzw. Eigenschaft deklariert und erhalten dabei den Modifikator **static**. 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 bei einem Programmeinsatz bereits 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. mit der Methodentabelle) auf dem Heap abgelegt.

Die im Wesentlichen schon aus dem Abschnitt 4.3.2.2 bekannte Abbildung zur Lage im Arbeitsspeicher bei Ausführung der **Main()** - Methode der Klasse `Bruchaddition` aus unserem OOP-Standardbeispiel (vgl. Kapitel 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 Adresse eines Objekts der Klasse **Type** mit Informationen über den eigenen Typ. Wir werden gelegentlich mit dem **typeof**-Operator auf dieses Typobjekt zugreifen.
- Beim Einsatz als Sperrobjekt zur Thread-Synchronisation (siehe Abschnitt 17.1.4 in Baltes-Götz (2021)) ist ein Verweis auf die zur Synchronisation verwendete Datenstruktur vorhanden.



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 Werttyp **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.

Im Kapitel 4 arbeiten wir ausschließlich mit *lokalen* Variablen. Im Zusammenhang mit der systematischen Behandlung der objektorientierten Programmierung im Kapitel 5 werden die Instanz- und Klassenvariablen ausführlich erläutert.

Im Unterschied zu anderen Programmiersprachen (z. B. C++) ist es in C# *nicht* möglich, sogenannte *globale* Variablen außerhalb von Klassen zu deklarieren. Eventuell aufgrund der in C# 9 eingeführten *Anweisungen auf oberster Ebene* aufkommende Zweifel an dieser Aussage werden im Abschnitt 4.1.2 ausgeräumt.

4.3.4 Elementare Datentypen

Als *elementare Datentypen* werden die in C# vordefinierten Werttypen zur Aufnahme von einzelnen Zahlen, Zeichen oder Wahrheitswerten bezeichnet.[1] Speziell für Zahlen existieren diverse Datentypen, die sich hinsichtlich Speichertechnik, Wertebereich und Platzbedarf unterscheiden.[2] 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:

---

[1] Mit Ausnahme von **decimal** werden die elementaren Datentypen auch als *primitiv* bezeichnet (wie in Java), siehe: https://learn.microsoft.com/en-us/dotnet/api/system.type.isprimitive

[2] Die seit C# 9 erlaubten, plattformabhängig mit 32 oder 64 Bit realisierten Ganzzahltypen **nint** und **nuint** fehlen in der Tabelle, siehe: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types

Typ	Beschreibung	Werte	Speicherplatzbedarf in Bits
sbyte	Diese Variablentypen speichern ganze Zahlen *mit* Vorzeichen, sodass auch negative Werte aufgenommen werden können. 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 in der Mantisse. Beispiel: `float pi = 3.141593f;` **float**-Literale benötigen das Suffix **f** oder **F** (siehe Abschnitt 4.3.10.2).	Minimum: $-3{,}402823 \cdot 10^{38}$ Maximum: $3{,}402823 \cdot 10^{38}$ Kleinster positiver Betrag: $1{,}401298 \cdot 10^{-45}$	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 in der Mantisse. Beispiel: `double ph=1.57079632679490;`	Minimum: $-1{,}79769313486232 \cdot 10^{308}$ Maximum: $1{,}79769313486232 \cdot 10^{308}$ Kleinster positiver Betrag: $4{,}94065645841247 \cdot 10^{-324}$	64 1 für das Vorz., 11 für den Expon., 52 für die Mantisse
decimal	Variablen vom Typ **decimal** speichern Gleitkommazahlen mit einer Genauigkeit von mind. 28 signifikanten Dezimalstellen in der Mantisse und eignen sich besonders für die **Finanzmathematik**, wo Rundungsfehler zu vermeiden sind. Beispiel: `decimal p = 2344.2554634m;` **decimal**-Literale (siehe unten) benötigen das 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 positiver Betrag: 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.

Typ	Beschreibung	Werte	Speicherplatz-bedarf in Bits
char	Variablen vom Typ **char** nehmen ein Unicode-Zeichen auf. Im Speicher landet aber nicht die Gestalt des Zeichens, sondern seine Nummer im Unicode-Zeichensatz. Daher zählt **char** zu den ganzzahligen (integralen) Datentypen. Beispiel: `char zeichen = 'j';` **char** - Literale (siehe unten) sind durch *einfache* Anführungszeichen zu begrenzen.	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**	8

Weil die Typen **sbyte**, **ushort**, **uint** und **ulong** nicht CLS-kompatibel sind (*Common Language Specification*), sollten sie nach Möglichkeit vermieden werden.[1]

Eine Variable mit einem integralen Datentyp (z. B. **int** oder **byte**) speichert eine ganze Zahl *exakt*, sofern es nicht durch eine Wertebereichsüberschreitung zu einem Überlauf und damit zu einem sinnlosen Speicherinhalt kommt (siehe Abschnitt 4.6.1).

Über den Speicherplatzbedarf von einem Byte (= 8 Bits) für eine **bool**-Variable informiert z. B. der **sizeof**-Operator (siehe Abschnitt 5.3.1.3.2.3).

Eine Variable zur Aufnahme einer *Gleitkommazahl* (synonym: *Gleitpunkt-* oder *Fließkommazahl*, engl.: *floating point number*) dient zur *approximativen* Darstellung einer reellen Zahl. Dabei werden drei Bestandteile separat gespeichert: Vorzeichen, Mantisse und Exponent. Diese ergeben nach der folgenden 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 resultiert im Vergleich zur Festkommadarstellung bei gleichem Speicherplatzbedarf ein erheblich größerer Wertebereich.[2] 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 tatsächlich durch die Mantisse gleiten lassen. Allerdings wird in der Regel durch eine Restriktion der Mantisse (z. B. auf das Intervall [1; 2)) für eine eindeutige Darstellung im Speicher gesorgt. Bei Gleitkommaliteralen (Gleitkommazahlen im Quellcode, siehe Abschnitt 4.3.10.2) ist die freie Wahl des Exponenten und damit das gleitende Dezimalkomma aber möglich.

---

[1] In C# sind diese Typen erlaubt, aber sie sind nicht in der CLS vorgeschrieben. Es kann also eine CLS-Sprache geben, die einen Typ aus der Liste nicht unterstützt. Man möchte aber Assemblies erstellen, die in allen Sprachen verwendbar sind, siehe z. B.:
https://docs.microsoft.com/en-us/dotnet/standard/language-independence-and-language-independent-components?redirectedfrom=MSDN

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

Weil der verfügbare Speicher für die Mantisse und den Exponenten begrenzt ist (siehe obige Tabelle), bilden die Gleitkommazahlen nur eine endliche (aber für die meisten praktischen Zwecke ausreichende) Teilmenge der reellen Zahlen.

Die binären Gleitkommatypen **float** und **double** sind optimiert hinsichtlich Speicherplatzbedarf, Wertebereich und Verarbeitungsgeschwindigkeit, doch ist ihre beschränkte Genauigkeit speziell bei finanzmathematischen Anwendungen oft inakzeptabel.

In einer Variablen vom dezimalen Gleitkommatyp **decimal** kann jede Dezimalzahl mit maximal 28 Stellen (Vorkommastellen + Nachkommastellen) exakt gespeichert werden. Allerdings sind der Speicherplatzbedarf und der Zeitaufwand zur Verarbeitung im Vergleich zu den binären Gleitkommatypen deutlich erhöht.

Nähere Informationen über die Darstellung von Gleitkommazahlen im Arbeitsspeicher eines Computers folgen gleich im Abschnitt 4.3.5.

4.3.5 Darstellung von Gleitkommazahlen im Arbeitsspeicher

Die binären Gleitkommatypen werden vor allem für mathematische Algorithmen (z. B. in Grafikprogrammen) benötigt. In finanzmathematischen Anwendungen sind die dezimalen Gleitkommatypen unverzichtbar.

4.3.5.1 Binäre Gleitkommadarstellung

Dieser Abschnitt wird vermutlich von vielen Lesern als zu mathematisch empfunden. Er enthält allerdings wichtige Details zu den binären Gleitkommatypen und sollte ertragen werden bis zur Aufklärung der Genauigkeitseinschränkungen beim Speichern von Gleitkommazahlen.

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

Quellcode	Ausgabe

`float df13 = 1.3f;`
`float df125 = 1.25f;`
`Console.WriteLine($"{df13,20:f16}");`
`Console.WriteLine($"{df125,20:f16}");`
	`1,2999999523162842`
	`1,2500000000000000`

Für die im Quellcode auftauchenden expliziten Werte 1,3 und 1,25 wird durch das Suffix **f** der Datentyp **float** angeordnet (mit immerhin 7 signifikanten Dezimalstellen), damit die Ablage in **float**-Variablen möglich ist.[1] Während die Zahl 1,25 im **float**-Format fehlerfrei gespeichert werden kann, gelingt das bei der Zahl 1,3 *nicht*.[2]

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:[3]

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

---

[1] Bei der Konstruktion des Beispiels war zu verhindern, dass die Genauigkeitsprobleme beim Speichern in der **WriteLine()** – Ausgabe weggerundet und somit versteckt werden.

[2] Man könnte einwenden, dass bei der Darstellung des wahren Wertes `1,3000000000000000` durch die Approximation `1,2999999523162842` nur eine Stelle korrekt ist und damit die Zusicherung von sieben signifikanten Dezimalstellen in Zweifel ziehen. Allerdings ist die Zusicherung so gemeint, dass beim *Runden* auf bis zu sieben Stellen alle Ziffern stimmen, was im Beispiel der Fall ist. Bei der Ermittlung der korrekten Dezimalstellen wird auch die Vorkommastelle mitgezählt.

[3] https://de.wikipedia.org/wiki/IEEE_754

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 reserviert (z. B. für die gleich erläuterte denormalisierte Darstellung sowie für +/- Unendlich). 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, dann landet der Wert

$$01111111_{bin} = 127$$

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.:

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

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

Zur Speicherung der Mantisse werden 23 (**float**) bzw. 52 (**double**) Bits verwendet. Das *i*-te Mantissen-Bit (von links nach rechts mit 1 beginnend nummeriert) hat die Wertigkeit 2^{-i}, sodass sich der *dezimale* Mantissenwert folgendermaßen ergibt:

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

Der Summenindex *i* startet mit 1, weil die führende 1 (= 2^0) der normalisierten Mantisse *nicht* abgespeichert wird (*hidden bit*). Daher 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.

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:

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

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,

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

gelingt dies bei der Restmantisse 0,3 nur approximativ:

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

Sehr aufmerksame Leser werden sich darüber wundern, wieso die Tabelle mit den elementaren Datentypen im Abschnitt 4.3.4 z. B.

$$1{,}40129846\cdot 10^{-45}$$

als betragsmäßig kleinsten positiven **float**-Wert nennt, obwohl der minimale Exponent nach obigen Überlegungen -126 (= 1 - 127) beträgt, was zum (gerundeten) dezimalen Exponentialfaktor

$$1{,}175\cdot 10^{-38}$$

führt. Dahinter steckt die *denormalisierte* (synonym: *subnormale*) 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 dieselben 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 komplett 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 Genauigkeit mit der Annäherung an die Null. Im folgenden Beispiel ist für einen denormalisiert gespeicherten **float**-Wert die Anzahl der signifikanten Dezimalstellen in der Mantisse deutlich kleiner als 7:

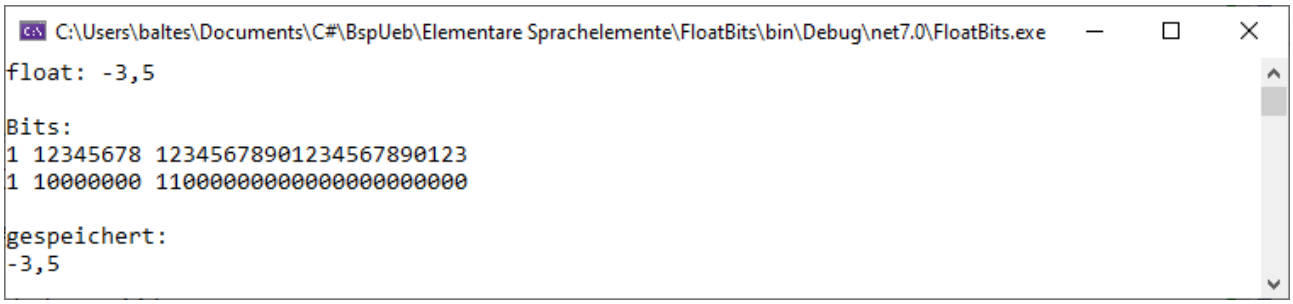
Quellcode	Ausgabe

`float fs = 1.234567e-38f;`
`Console.WriteLine($"normalisiert:\n{fs,20:e10}");`
`fs = 1.234567e-45f;`
`Console.WriteLine($"denormalisiert:\n{fs,20:e10}");`
	`normalisiert:`
	`  1,2345670685e-038`
	`denormalisiert:`
	`  1,4012984643e-045`

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 Abschnitt 14.5.5) im Kurs ansonsten keine Rolle spielen. Eine Beispielausgabe des Programms `FloatBits`:



4.3.5.2 Dezimale Gleitkommadarstellung

Neben den für mathematische Algorithmen sehr gut geeigneten *binären* Gleitkommatypen (mit der Basis 2 für die Mantisse und den Exponenten) bietet C# auch den für finanzmathematische Algorithmen konzipierten *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:

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

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

$$\overbrace{9999999999999999999999999999}^{\text{28 Stellen}} < 2^{96}\text{-}1 = \overbrace{79228162514264337593543950335}^{\text{29 Stellen}}$$

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

Wie im Abschnitt 4.3.5.1 zu sehen war, wird die Zahl 1,3 im *binären* Gleitkommaformat durch eine prinzipiell unendliche Serie von Brüchen mit einer Zweierpotenz im Nenner dargestellt, sodass 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 hingegen exakt gespeichert:

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

Die folgenden Anweisungen demonstrieren den Genauigkeitsvorteil des Datentyps **decimal** im finanzmathematisch relevanten Wertebereich gegenüber den binären Gleitkommatypen **float** und **double**:

---

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

Quellcode	Ausgabe

`Console.WriteLine(10.0f - 9.9f);`
`Console.WriteLine(10.0 - 9.9);`
`Console.WriteLine(10.0m - 9.9m);`
	`0,10000038`
	`0,09999999999999964`
	`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 (wenn auch nicht 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:    -24,516295194625854
    Benöt. Zeit:   3241 Millisek.
  decimal:
    Abweichung:    0,0
    Benöt. Zeit:   21835 Millisek.
  ```

  Die gut bezahlten Verantwortlichen vieler Banken, die sich gerne als „Global Player“ betätigen und dabei den vollen Sinn der beiden Worte ausschöpfen (mit Niederlassungen in 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, sodass man bei finanzmathematischen Aufgaben trotz des erhöhten Zeitaufwands (im Beispiel: ca. Faktor 7) den Datentyp **decimal** verwenden sollte.

  Sind in einem Algorithmus nur die Addition und die Subtraktion von ganzen Zahlen (z. B. Rechnungsbeträge in Cent) erforderlich, dann taugen auch die Ganzzahltypen **int** und **long** für monetäre Berechnungen. Sie verursachen sehr wenig Aufwand und bieten eine perfekte Genauigkeit, sofern ihr Wertebereich nicht verlassen wird.
- Keine Unterstützung für Sonderfälle wie +/- Unendlich und NaN (***N**ot **a N**umber*) (vgl. Abschnitt 4.6)

4.3.6 Variablendeklaration, Initialisierung und Wertzuweisung

Wie Sie aus dem Abschnitt 4.3.1 wissen, muss jede Variable vor ihrer Verwendung deklariert werden. Dabei sind auf jeden Fall ein Datentyp und ein Name anzugeben, und optional kann die neue Variable auch gleich initialisiert (auf einen Wert gesetzt) werden. Bei der Deklaration beschränken wir uns vorläufig auf die *lokalen* Variablen. Ihre Deklaration darf im Rumpf einer Methoden- bzw. Eigenschaftsdefinition an beliebiger Stelle *vor* der ersten Verwendung erscheinen. Um den (im

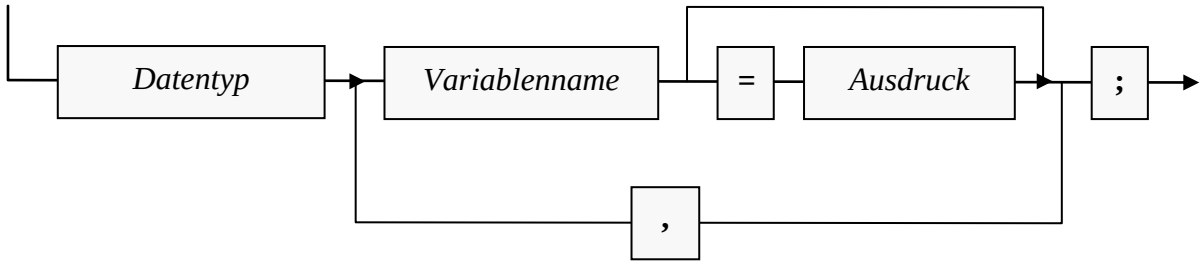
---

[1] Gemessen auf einem PC mit Intel-CPU Core i3 550 unter Windows 10 (64 Bit)
Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
…\BspUeb\Elementare Sprachelemente\DoubleDecimal

Abschnitt 4.3.8 behandelten) Gültigkeitsbereich einer lokalen Variablen zur Vermeidung von Fehlern zu minimieren, sollte sie unmittelbar vor der ersten Verwendung deklariert werden (Bloch 2018, S. 261).

Es folgt das Syntaxdiagramm zur Deklaration einer lokalen Variablen, wobei der Übersichtlichkeit halber die mit C# 3.0 eingeführte implizierte Typisierung ignoriert wird (siehe Abschnitt 4.3.7.1):

Deklaration einer lokalen Variablen



Als Datentypen sind erlaubt (vgl. Abschnitt 4.3.2):

- Werttypen, z. B.

```
int i;
```

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

```
Bruch b;
```

Es ist üblich, die Namen von lokalen Variablen mit einem Kleinbuchstaben beginnen zu lassen, also das sogenannte *Camel Casing* zu verwenden (vgl. Abschnitt 4.1.6).

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

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

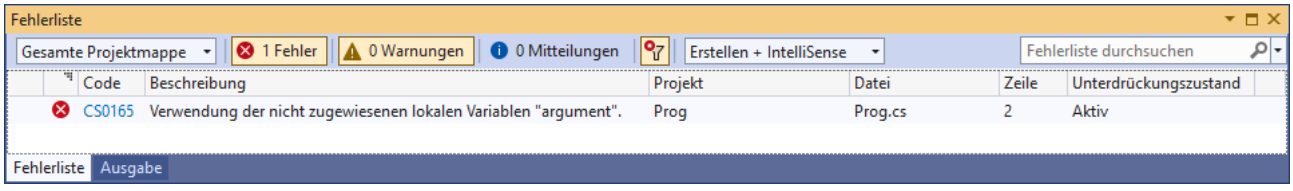
Im zweiten Beispiel wird ein `Bruch`-Objekt per **new**-Operator (siehe Abschnitt 5.4.2) erzeugt, und anschließend dessen Adresse in die Referenzvariable `b` geschrieben.

Mit der Konstruktion von gültigen *Ausdrücken*, die einen Wert von passendem Datentyp liefern, werden wir uns noch ausführlich beschäftigen.

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:

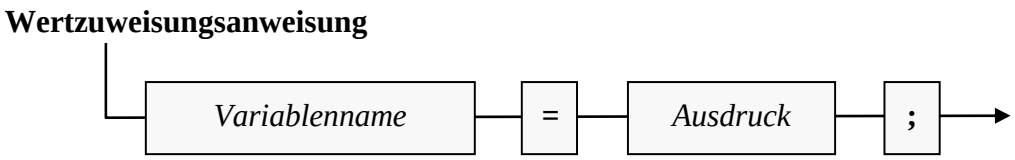
```
int argument;
Console.WriteLine("Argument = " + argument);
```

Der Compiler meint dazu:



Weil *Instanz- und Klassenvariablen* automatisch mit der typspezifischen Null initialisiert werden (siehe Abschnitt 5.2.3), 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:



Beispiel:

```
zaehler = zaehler * b.nenner + b.zaehler * nenner;
```

Durch diese Wertzuweisungsanweisung aus der `Addiere()` - Methode unserer Klasse `Bruch` (siehe Abschnitt 1.3) erhält die **int**-Instanzvariable `zaehler` den Wert des folgenden Ausdrucks:

```
zaehler * b.nenner + b.zaehler * nenner
```

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 kennengelernt:

- Deklaration einer lokalen Variablen
- Wertzuweisung

4.3.7 Implizite und zielorientierte Typisierung

4.3.7.1 Implizite Typisierung von lokalen Variablen

Bei der initialisierenden Deklaration von lokalen Variablen darf seit C# 3.0 über das Schlüsselwort **var** die implizite Typisierung verwendet werden. Dabei wird der Compiler aufgefordert, den Datentyp aus dem zugewiesenen Ausdruck zu übernehmen, z. B.:

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

Diese Arbeitserleichterung für Programmierer schwächt keinesfalls das Prinzip der strengen und statischen Typisierung. Sobald der Mauszeiger über einem Variablennamen verharrt, zeigt sich die Entwicklungsumgebung perfekt über den Datentyp informiert:

```
var d1 = 2147483647;
```

Eventuell war im Beispiel für die Variable `d1` aber der Datentyp **double** vorgesehen, und versehentlich ein **int**-Literal zum Initialisieren verwenden worden. Dann kann der falsche Datentyp zum gravierenden Problem werden, z. B.:

Quellcode	Ausgabe

`var d1 = 2147483647;`
`double d2 = 2147483647;`
`d1 = d1 + 5;`
`d2 = d2 + 5;`
`Console.WriteLine($"d1 = {d1}\nd2 = {d2}");`
	`d1 = -2147483644`
	`d2 = 2147483652`

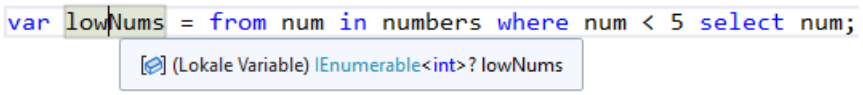
Das Addieren der Zahl 5 führt bei `d1` aufgrund der fehlerhaften impliziten Typisierung zu einem Ganzzahlüberlauf mit sinnlosem Ergebnis (siehe Abschnitt 4.6), während bei der explizit typisierten Variablen `d2` dieselbe Operation korrekt ausgeführt wird.

Bei umständlich zu schreibenden Datentypen ist **var** aber eine erhebliche Schreibvereinfachung, und der gute Vorsatz zur stets expliziten Notation von Datentypen gerät in Gefahr. Im folgenden

Beispiel dient das konkretisierte generische Interface **IEnumerable<int>** aus dem Namensraum **System.Collections.Generic** als Datentyp:[1]

```
IEnumerable<int> lowNums = from num in numbers where num < 5 select num;
```

Im Beispiel lassen sich mit dem Schlüsselwort **var** 13 Zeichen einsparen, und der Compiler behält trotzdem den Überblick:[2]



4.3.7.2 Zieltypisierte new-Ausdrücke

Obwohl wir uns mit dem **new**-Operator noch nicht systematisch beschäftigt haben (siehe Abschnitt 5.4.2), ist Ihnen die folgende Anweisung aus dem Bruchrechnungsprojekt vermutlich schon vertraut:

```
Bruch b = new Bruch();
```

Es wird die Referenzvariable `b` vom Typ der Klasse `Bruch` deklariert und initialisiert. Auf den **new**-Operator folgt ein sogenannter *Konstruktor*, der zum Erstellen und Initialisieren eines neuen `Bruch`-Objekts dient und den Namen der Klasse trägt (siehe Abschnitt 5.4). Zusammengenommen steht in obiger Anweisung rechts vom Zuweisungszeichen ein **new**-Ausdruck.

Es ist in vielen, aber keinesfalls in allen Fällen so, dass die im **new**-Ausdruck via Konstruktor verwendete Klasse auch als Datentyp genutzt wird. Sie werden es bald naheliegend finden, als Datentyp eine Schnittstelle anzugeben und im **new**-Operator den Konstruktor einer die Schnittstelle implementierenden Klasse zu verwenden (siehe Kapitel 9).

Wenn aber die Konstruktor-Klasse auch als Datentyp dient, dann ist die doppelte Nennung des Klassennamens etwas umständlich. Für *lokale* Variablen vom Typ einer Klasse ist die im Abschnitt 4.3.7.1 beschriebene implizite Typisierung unter Verwendung des Schlüsselworts **var** erlaubt, z. B.:

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

Während die implizite Typisierung für Felder verboten ist, sind die mit C# 9 eingeführten *zieltypisierten **new**-Ausdrücke* (engl.: *target-typed **new** expression*) für lokale Variablen *und* für Felder erlaubt, z. B.:[3]

```
Bruch b = new();
```

Gegen die Schreibvereinfachung durch Verwendung des Schlüsselworts **var** an Stelle des Datentyps in einer lokalen Variablendeklaration wird oft eingewendet, dass die Lesbarkeit des Quellcodes leidet. Das ist bei einem zieltypisierten **new**-Ausdruck nicht zu befürchten.

---

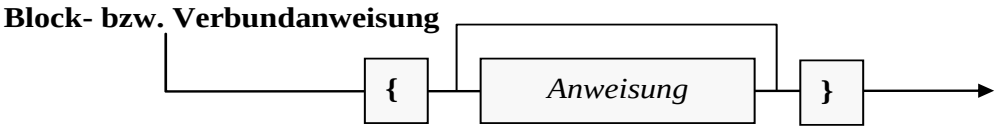
[1] Die Begriffe *generisch* und *Interface* sowie die im Ausdruck verwendete LINQ-Technik (*Language Integrated Query*) werden später behandelt.

[2] Das Fragezeichen am Ende der Typbezeichnung deutet keine Unsicherheit der Entwicklungsumgebung an, sondern steht im Zusammenhang mit der im Abschnitt 15.1 behandelten **null** - (Un)zulässigkeit bei Referenztypen. Eine Referenzvariable darf seit Menschengedenken auch den Wert **null** besitzen. C# unterstützt es optional seit der Version 8, im Quellcode bei der Deklaration einer Referenzvariablen festzulegen, ob für diese Variable der Wert **null** erlaubt sein soll oder nicht. Ist diese Option aktiviert, dann muss an eine Referenztypbezeichnung ein Fragezeichen angehängt werden, wenn der Wert **null** erlaubt sein soll. Per Voreinstellung gilt weiterhin die generelle **null**–Zulässigkeit (ohne Anforderung per Fragezeichen). In QuickInfo-Fenstern taucht das Fragezeichen aber trotzdem auf.

[3] Auf der Webseite
https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/proposals/csharp-9.0/target-typed-new
übersetzt Microsoft etwas anders: *Mit dem Ziel typisierte new Ausdrücke*.

4.3.8 Blockanweisung und Sichtbarkeitsbereich 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 untergeordnete Anweisungsblöcke gebildet werden, wiederum durch geschweifte Klammen begrenzt. Man spricht hier auch von **Block- bzw. Verbundanweisungen**, und diese können überall stehen, wo eine einzelne Anweisung erlaubt ist:



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

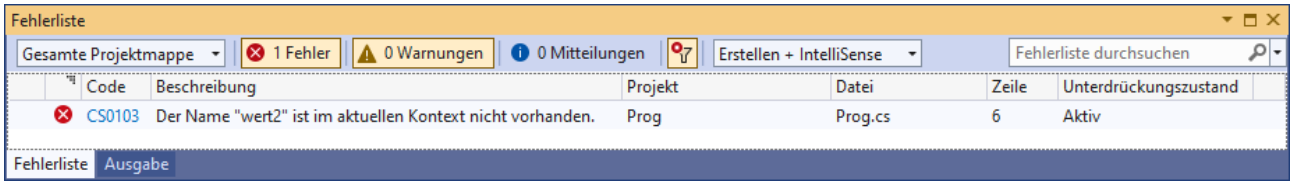
Oft treten Blöcke als Bestandteile von bedingten Anweisungen, Fallunterscheidungen und Wiederholungsanweisungen auf (siehe Abschnitt 4.7), z. B. in der Methode `Kuerze()` der Klasse `Bruch` (siehe Abschnitt 1.3):

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

Blockanweisungen haben einen wichtigen Effekt auf die Sichtbarkeit (alias: Gültigkeit) der darin deklarierten Variablen: Eine lokale Variable ist gültig von der deklarierenden Anweisung bis zur schließenden Klammer des Blocks, in dem sich die Deklaration befindet. Nur in diesem *Sichtbarkeitsbereich* (alias: *Gültigkeitsbereich* oder *Kontext*, engl. *scope*) kann sie über ihren Namen angesprochen werden. Beim Versuch, das folgende (weitgehend sinnfreie) Beispielprogramm

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

zu übersetzen, erhält man vom Compiler die Fehlermeldung:



Wird die fehlerhafte Zeile auskommentiert, lässt sich das Programm übersetzen. In dem zur **if**-Anweisung gehörenden Block ist die im übergeordneten Block der **Main()** - Methode deklarierte Variable `wert1` also sichtbar, und die folgende Anweisung bietet keinen Anlass zur Kritik:

```
Console.WriteLine("Gesamtwert = " + (wert1 + wert2));
```

Bei hierarchisch geschachtelten Blöcken ist es in C# *nicht* erlaubt, auf mehreren Stufen Variablen mit identischem Namen zu deklarieren. Diese kaum sinnvolle Option ist z. B. in der

Programmiersprache C++ vorhanden und erlaubt dort Fehler, die schwer aufzuspüren sind. In C# gehört ein eingeschachtelter Block zum Gültigkeitsbereich des umgebenden Blocks.

Der Sichtbarkeitsbereich einer lokalen Variablen sollte möglichst klein gehalten werden, um die Lesbarkeit und die Wartungsfreundlichkeit des Quellcodes zu verbessern. Vor allem wird auf diese Weise das Risiko von Programmierfehlern reduziert. Wird eine Variable zu früh deklariert, bestehen viele Gelegenheiten für schädliche Wertzuweisungen. Aus einer längst überwundenen Verpflichtung alter Programmiersprachen ist bei manchen Programmierern die Gewohnheit entstanden, alle lokalen Variablen am Blockbeginn zu deklarieren. Stattdessen sollten lokale Variablen zur Minimierung ihres Sichtbarkeitsbereichs unmittelbar vor der ersten Verwendung deklariert werden (Bloch 2018, S. 261).

Zur übersichtlichen Gestaltung von C# - Programmen ist das Einrücken von Anweisungsblöcken sehr zu empfehlen, wobei Sie die Position der einleitenden Blockklammer 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);
}
```

Das Visual Studio unterstützt Sie bei der Einhaltung einer Konvention und verwendet per Voreinstellung die *rechte* Variante. Im Abschnitt 4.1.4 wurde schon gezeigt, dass man diese Voreinstellung über

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

ändern kann. Im Manuskript wird die Platz sparende *linke* Variante bevorzugt.

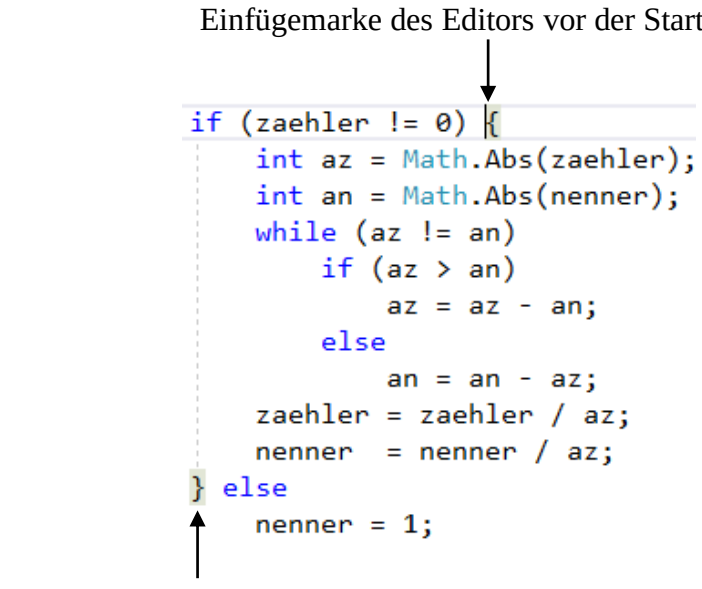
Im Quellcodeeditor der Entwicklungsumgebung kann ein markierter Block aus mehreren Zeilen mit

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, z. B:



4.3.9 Konstanten

Für einen im Programm benötigten festen Wert, der schon *zur Übersetzungszeit* feststeht und nie geändert werden soll, empfiehlt sich die Definition einer *Konstanten*, die dann im Quellcode über ihren Namen angesprochen werden kann, denn:

- Bei einer späteren Änderung des Werts ist nur die Quellcodezeile mit der Konstantendeklaration betroffen.
- Der Quellcode ist leichter zu lesen, wenn Variablennamen an Stelle von „magischen Zahlen“ stehen.

Im folgenden Beispiel wird für eine in Sekunden gemessene Dauer die Anzahl der vergangenen Tage bestimmt (ein Tag dauert 86400 Sekunden):

Quellcode	Ausgabe

`const int secDay = 86400;`
`int duration = 176000;`
`Console.WriteLine($"Vergangene ganze Tage: {duration / secDay}");`
	`2`

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 4.3.10) und andere Konstanten enthält.
- Ihr Wert kann im Programm *nicht* geändert werden, sodass insbesondere irrtümliche Wertveränderungen ausgeschlossen sind.

Manche Programmierer verwenden im Namen einer Konstanten ausschließlich Großbuchstaben und verbessern bei einem Mehrwortnamen die Lesbarkeit durch trennende Unterstriche (z. B.: `SEC_DAY`). Nach der aktuell dominierenden Meinung sollte sich in C# der Modifikator **const** jedoch *nicht* auf die Benennung auswirken, sodass die im Abschnitt 4.1.6 beschriebenen Konventionen auch für Konstanten anwendbar sind.[1]

Als Datentypen sind für Variablen mit **const**-Deklaration ausschließlich die elementaren Datentypen und der Typ **String** erlaubt bzw. sinnvoll.[2]

Seit C# 10 darf zur Initialisierung von konstanten **String**-Objekten die Zeichenfolgeninterpolation verwendet werden (vgl. Abschnitt 4.2.2.2), z. B.:

```
const string suppe = "Tomatensuppe";
const string splan = $"Als Suppe gibt es heute {suppe}";
```

In den Interpolationsausdrücken sind aber nur konstante **String**-Variablen erlaubt, weil numerische Konstanten zur Laufzeit unter Berücksichtigung der Region in Zeichenfolgen konvertiert werden.

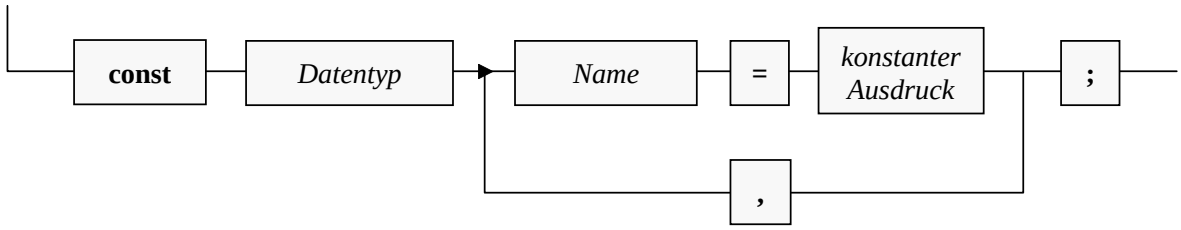
So sieht das Syntaxdiagramm zur Deklaration einer *konstanten* lokalen Variablen aus:

---

[1] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constants
https://google.github.io/styleguide/csharp-style.html

[2] Begründung:
- Nicht-elementare Strukturen (siehe Abschnitt 6.1) 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.

Deklaration von konstanten lokalen Variablen



Neben den lokalen Variablen können auch die Felder einer Klasse als konstant deklariert werden. Bei Feldern ist auch der Modifikator **readonly** erlaubt. Während **const** eine Wertfixierung zur Übersetzungszeit bewirkt, kann mit **readonly** eine Wertfixierung im Programmablauf nach der Initialisierung per Konstruktor vereinbart werden (siehe Abschnitt 5.2.5).

4.3.10 Literale

Die im Quellcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 4.3.9 wissen, ist es oft sinnvoll, Literale innerhalb von Konstanten-Deklarationen zu verwenden, z. B.:

```
const int secDay = 86400;
```

Aber auch andere Einsatzorte kommen in Frage (z. B. Variableninitialisierung, Aktualparameter).

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 Nachschlagecharakter, sodass man beim ersten Lesen nicht jedes Detail aufnehmen muss bzw. kann.

4.3.10.1 Ganzzahlliterale

Ganzzahlliterale können im dezimalen, im binären oder im hexadezimalen Zahlensystem geschrieben werden. Bei Verwendung des seit C# 7.0 unterstützten binären Zahlensystems (mit der Basis 2 und den Ziffern 0, 1) ist das Präfix **0b** oder **0B** zu verwenden. Die Anweisungen:

```
int i = 11, j = 0b11;
Console.WriteLine($"i = {i}, j = {j}");
```

liefern die Ausgabe:

```
i = 11, j = 3
```

Für das Ganzzahlliteral `0b11` ergibt sich der dezimale Wert 3 aufgrund der Stellenwertigkeiten im Binärsystem folgendermaßen:

$$11_{bin} = 1 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 2 + 1 \cdot 1 = 3$$

Bei Verwendung des hexadezimalen Zahlensystems (mit der Basis 16 und den Ziffern 0, 1, …, 9, A, B, C, D, E, F) ist das Präfix **0x** oder **0X** zu verwenden. 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:

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

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

Seit C# 7.0 dürfen dezimale Ganzzahlliterale mit dem Unterstrich als **Zifferntrennzeichen** geschrieben werden, z. B.:

```
long i = 9_000_000_000_000_000_000;
```

Seit C# 7.2 ist dies auch bei Ganzzahlliteralen im binären oder im hexadezimalen Zahlensystem erlaubt, wobei der Unterstrich auch zwischen dem Zahlensystempräfix und der Zahl stehen darf, z. B.:

```
int i127 = 0b_0111_1111;
```

Der Datentyp eines Ganzzahlliterals hängt von seinem Wert und einem eventuell vorhandenen Suffix ab:

- Ist *kein* Suffix vorhanden, dann hat das Ganzzahlliteral den ersten Typ aus der folgenden Serie, der seinen Wert aufnehmen kann:

  int, **uint**, **long**, **ulong**

  Beispiele:

Literale	Typ
2147483647	**int**
2147483648	**uint**
9223372036854775807	**long**
9223372036854775808	**ulong**

- Ein Ganzzahlliteral mit Suffix **u** oder **U** (*unsigned*, ohne Vorzeichen) hat den ersten Typ aus der folgenden Serie, der seinen Wert aufnehmen kann:

  uint, **ulong**

  Beispiele:

Literale	Typ
2147483647U	**uint**
9223372036854775807u	**ulong**

- Ein Ganzzahlliteral mit Suffix **l** oder **L** (*Long*) hat den ersten Typ aus der folgenden Serie, der seinen Wert aufnehmen kann:

  long, **ulong**

  Beispiele:

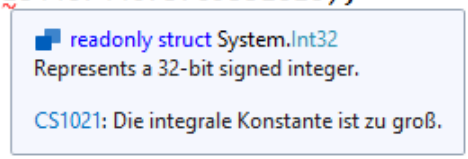
Literale	Typ
2147483647L	**long**
9223372036854775808L	**ulong**

  Der Kleinbuchstabe **l** ist leicht mit der Ziffer **1** zu verwechseln und daher als Suffix ungeeignet.

- 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, dann warnt das Visual Studio, z. B.

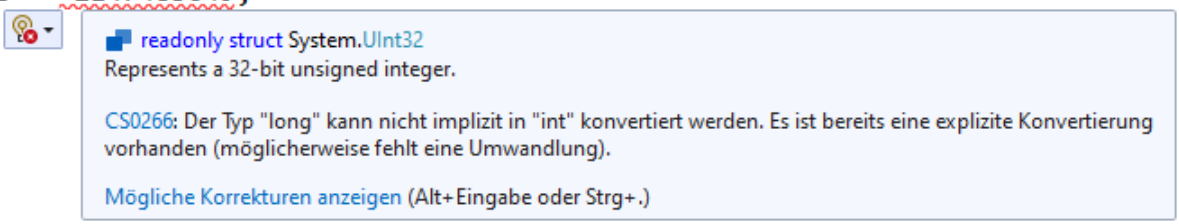
```
Console.WriteLine(18446744073709551616);
```



  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.

Per Ganzzahlliteral können nur nicht-negative Zahlen dargestellt werden, und im folgenden Beispiel

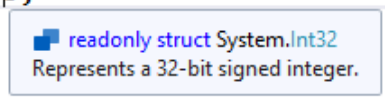
```
int l = -2147483649;
```



entsteht auf der rechten Seite des Zuweisungszeichens ein Ausdruck vom Typ **long** mit negativem Wert, indem die Vorzeichenumkehr-Operation auf ein Literal vom Typ **uint** angewendet wird.[1]

Weil der Compiler bzw. das Visual Studio einige Intelligenz bei der Verwendung von Ganzzahlliteralen zeigt, sind die eben erwähnten Suffixe oft überflüssig. Man kann z. B. einer vorzeichenlosen Ganzzahlvariablen ein Literal mit geeignetem Wert zuweisen, obwohl dieses Literal zu einem vorzeichenbehafteten Typ gehört:

```
ulong uli = 33;
```



4.3.10.2 Gleitkommaliterale

Zahlen mit Dezimalpunkt oder Exponent sind in C# vom Typ **double**, wenn nicht per Suffix ein alternativer Typ erzwungen wird:

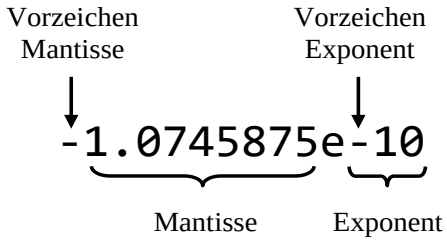
- Durch das Suffix **F** oder **f** wird der Datentyp **float** erzwungen, z. B.:
  `9.78f`
- Durch das Suffix **M** oder **m** wird der Datentyp **decimal** erzwungen, z. B.:
  `1.3m`

Mit dem Suffix **D** oder **d** wird auch bei einer Zahl *ohne* Dezimalpunkt oder Exponent der Datentyp **double** erzwungen. Warum das Suffix **d** im folgenden Beispiel für das mathematisch korrekte Rechenergebnis sorgt, erfahren Sie im Zusammenhang mit dem Unterschied zwischen der Ganzzahl- und der Gleitkommaarithmetik (siehe Abschnitt 4.5.1):

Quellcode	Ausgabe
`Console.WriteLine(5/2);` `Console.WriteLine(5d/2);`	`2` `2,5`

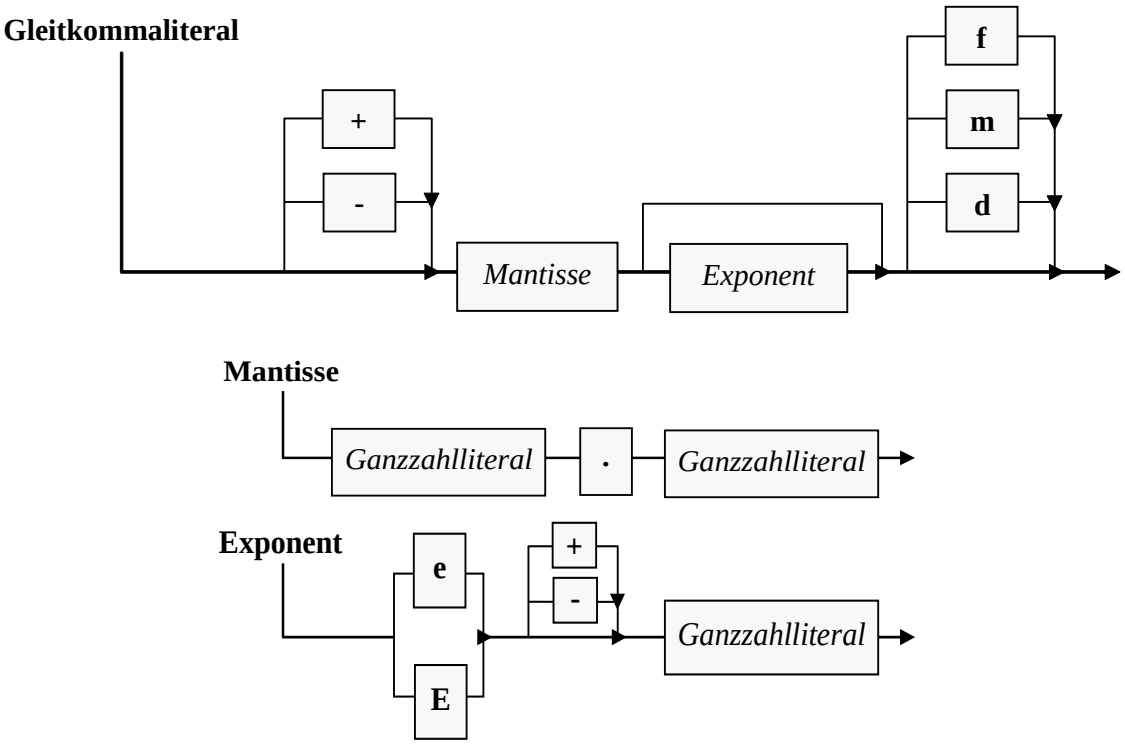
[1] Dass hinter dem elementaren Datentyp **uint** die vordefinierte BCL-Struktur **System.UInt32** steckt, erfahren Sie im Abschnitt 6.1.5.

Neben der alltagsüblichen Schreibweise (mit dem *Punkt* als Dezimaltrennzeichen) erlaubt C# bei Gleitkommaliteralen auch die wissenschaftliche Exponentialnotation (mit der Basis 10), z. B. bei der Zahl -0,00000000010745875):



Eine Veränderung des Exponenten lässt das Dezimaltrennzeichen gleiten und macht somit die Bezeichnung *Gleitkommaliteral* (engl.: *floating-point literal*) plausibel.

In den folgenden Syntaxdiagrammen werden die wichtigsten Regeln für Gleitkommaliterale beschrieben:



Für die Ganzzahlliterale als Bestandteile eines Gleitkommaliterals gelten einige selbstverständliche bzw. freundliche Regeln:

- Es ist das dezimale Zahlensystem zu verwenden.
- Die im Abschnitt 4.3.10.1 beschriebenen Präfixe (**0x**, **0X**) und Suffixe (z. B. **L**, **U**) sind verboten.
- Zifferntrennstriche sind erlaubt.
- Der Exponent wird zur Basis 10 verstanden.
- Die Länge der Ganzzahlliterale (Anzahl der Ziffern) ist praktisch unbegrenzt, sodass z. B. die Anweisung:
  ```
  Console.WriteLine(12345678901234567890000000000000000000000000000000000.0);
  ```
  akzeptiert wird und die folgende Ausgabe produziert:
  ```
  1,2345678901234568E+53
  ```

Der Einfachheit halber unterschlagen die Syntaxdiagramme die Möglichkeit, aus einem Ganzzahlliteral über das Suffix **d** ein Gleitkommaliteral herzustellen, z. B.:

```
Console.WriteLine(5d/2);
```

Der Compiler achtet bei Wertzuweisungen unter Verwendung von Gleitkommaliteralen streng auf die Typkompatibilität. Z. B. führt die folgende Deklarationsanweisung:

```
float pf = 1.3;
```

zu der Fehlermeldung:

```
Literale des Typs "Double" können nicht implizit in den Typ "float" konvertiert werden.
Verwenden Sie ein F-Suffix, um ein Literal mit diesem Typ zu erstellen.
```

Um das Problem zu lösen, muss das Gleitkommaliteral per Suffix den Datentyp **float** erhalten:

```
float pf = 1.3f;
```

Die Zahl 1,3 verlässt zwar nicht den **float**-Wertebereich, doch steckt im **double**-Literal `1.3` ein kleinerer Darstellungsfehler als in der **float**-Variablen `pf`, und der Compiler verhindert eine genauigkeits-reduzierende Zuweisung (siehe Abschnitt 4.3.5.1 zur binäre Gleitkommadarstellung).

4.3.10.3 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
`bool b = false;` `Console.WriteLine(b);`	`False`

4.3.10.4 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 sogenannte *Escape-Sequenz*:
- **Escape-Sequenzen**
  Hier dürfen einem einleitenden Rückwärts-Schrägstrich u. a. folgen:
  - ein Steuerzeichen, z. B.:
    Neue Zeile `\n`
    Tabulator `\t`
  - das Hochkomma sowie der Rückwärts-Schrägstrich:
    `\'`
    `\\`
  - das Null-Zeichen:
    `\0`
    Das Zeichen mit der Nummer 0 im Unicode-Zeichensatz wird in der Programmiersprache C (*nicht* in C#) dazu verwendet, das Ende einer Zeichenfolge zu signalisieren.

Beispiel:

Quellcode	Ausgabe
`char rs = '\\';` `Console.WriteLine("Inhalt von rs: " + rs);`	`Inhalt von rs: \`

- **Unicode-Escape-Sequenzen**
  Eine Unicode-Escape-Sequenz enthält eine Unicode-Zeichennummer als …
  - Ganzzahlliteral im Hexadezimalsystem,
  - mit vier Stellen, ggf. links mit Nullen aufgefüllt,
  - nach der Einleitung durch **\u** oder **\x**.

  Beispiel:

  ```
  const char alpha = '\u03b3';
  ```

  Dezimal interpretiert ergibt sich für das Ganzzahlliteral der Wertebereich von 0 bis $2^{16} - 1 = 65535$. Über eine Unicode-Escape-Sequenz lassen sich Zeichen ansprechen, die per Tastatur nicht einzugeben sind. 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).

4.3.10.5 Zeichenfolgenliterale

Zeichenfolgenliterale 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";
```

Zeichenfolgenliterale 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) über eine implizite oder explizite **using**-Direktive importiert worden ist, dann kann die obige Variablendeklaration auch so geschrieben werden:

```
String name = "Otto's Welt";
```

Wegen der speziellen Unterstützung der sehr wichtigen Klasse **String** durch den Compiler ist hier ausnahmsweise die Groß-/Kleinschreibung irrelevant.

Um ein doppeltes Hochkomma in ein Zeichenfolgenliteral aufzunehmen, ist eine Escape-Sequenz erforderlich, z. B.:

```
string name = "\"Eumel\"";
```

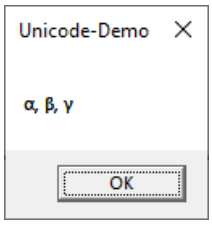
Während ein **char**-Literal stets *genau ein* Zeichen enthält, kann ein Zeichenfolgenliteral aus beliebig vielen Zeichen bestehen oder auch leer sein, z. B.:

```
string name = "";
```

Im folgenden Beispiel wird die statische Methode **Show()** der Klasse **MessageBox** aus dem Namensraum **System.Windows** zur Anzeige eines Zeichenfolgenliterals verwendet, das drei Unicode-Escape-Sequenzen enthält:[1]

```
using 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 einer Zeichenfolge das @-Zeichen voran, z. B.:

```
Console.WriteLine(@"Pfad: C:\Program Files\Map\bin");
```

Weil nun das Escape-Zeichen fehlt, sind Ersatzlösungen erforderlich:

- Um ein doppeltes Hochkomma in die Zeichenfolge einzufügen, muss man es verdoppeln, z. B.:

  ```
  string s = @"Auftretenshäufigkeit für das Wort ""Resonanz"":";
  ```

- Die Escape-Sequenz **\n** ist durch einen Zeilenumbruch im Quellcode zu ersetzen, was sich allerdings nachteilig auf die Formatierung des Quellcodes auswirkt, z. B.:

Quellcode	Ausgabe

`class Prog {`
`  static void Main() {`
`  string s = @"Literal`
`mit Zeilenwechsel";`
`  System.Console.WriteLine(s);`
`  }`
`}`
	`Literal`
	`mit Zeilenwechsel`

Das @-Zeichen darf mit dem im Abschnitt 4.2.2.2 beschriebenen Interpolationspräfix ($) kombiniert werden, z. B.:

```
string s = "Map";
Console.WriteLine($@"Pfad: C:\Program Files\{s}\bin");
```

Während bis C# 7.3 die Reihenfolge **$@** vorgeschrieben war, ist seit C# 8 die Reihenfolge der beiden Präfixzeichen beliebig.

---

[1] Damit das Beispiel unter .NET 7 ausgeführt werden kann, muss man in der Projektdatei ein windows-spezifisches **TargetFramework** einstellen und mit dem Element **UseWPF** die Verwendung der GUI-Bibliothek WPF aktivieren, z. B.:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

Seit C# 11 erlaubt das *mehrzeilige Zeichenfolgenliteral* (alias: *Rohzeichenfolgenliteral*, engl.: *raw string literal*) noch mehr Flexibilität bei Zeichenfolgenliteralen:

- Es wird eingeleitet und beendet mit *drei* doppelten Anführungszeichen.
- Als Textbestandteile sind Zeilenwechsel, Rückwärtsschrägstriche (ohne Escape-Funktion) sowie doppelte Anführungszeichen erlaubt.
- Hinter der einleitenden und vor der terminierenden """ - Sequenz fügt man einen Zeilenwechsel ein, der *nicht* zum Literal gehört.
- Die im Quellcode vor der terminierenden Sequenz stehende Anzahl von Leerzeichen wird am Anfang jeder Zeile des Literals entfernt, sodass man eine Einrückung im Quellcode erzielt.

Das folgende Programm verwendet ein @-Zeichenfolgenliteral und ein Rohzeichenfolgenliteral:

```
class Prog {
    static void Main() {
        string sk = @"Das @-Präfix ermöglicht mehrzeilige Texte
mit \-Zeichen ohne Escape-Funktion
und mit zu verdoppelndem ""-Zeichen.";
        Console.WriteLine(sk + "\n");

        string uz = """
            Das Rohzeichenfolgenliteral (alias: mehrzeilige Zeichenfolgenliteral)
            ermöglicht mehrzeilige, im Quellcode gleichmäßig eingerückte Texte
            mit \-Zeichen ohne Escape-Funktion
            und "-Zeichen ohne Sonderfunktion.
            """;
        Console.WriteLine(uz);
    }
}
```

Es resultiert die folgende Ausgabe:

```
Das @-Präfix ermöglicht mehrzeilige Texte
mit \-Zeichen ohne Escape-Funktion
und mit zu verdoppelndem "-Zeichen.

Das Rohzeichenfolgenliteral (alias: mehrzeilige Zeichenfolgenliteral)
ermöglicht mehrzeilige, im Quellcode gleichmäßig eingerückte Texte
mit \-Zeichen ohne Escape-Funktion
und "-Zeichen ohne Sonderfunktion.
```

Wie im @-Zeichenfolgenliteral sind auch im Rohzeichenfolgenliteral Interpolationsausdrücke erlaubt.

4.3.10.6 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 das Objekt zum Abräumen durch den Garbage Collector freigegeben.

Da C# eine streng typisierte Programmiersprache ist, und das Literal **null** einen Ausdruck darstellt (vgl. Abschnitt 4.5), 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, sodass man ihn im Programmieralltag getrost vergessen kann.

4.3.11 Übungsaufgaben zum Abschnitt 4.3

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 im folgenden Programm:

```
class Prog {
   static void main() {
      float pi = 3,141593;
      double radius = 2,0;
      System.Console.WriteLine('Der Flächeninhalt beträgt: {pi * radius * radius:f3}');
   }
}
```

3) Produzieren Sie bitte per **WriteLine()** – Aufruf die folgende Ausgabe:

```
Dies ist ein Zeichenfolgenliteral:
    "Hallo"
```

4) Im folgenden Quellcode auf oberster Ebene (mit impliziter **using**-Direktive für den Namensraum **System**) 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

`char zeichen = 'c';`
`int nummer = zeichen;`
`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**-Variablen zu übergeben? Wie kann man das Zeichen 'c' über eine Unicode-Escape-Sequenz ansprechen?

5) Der im Abschnitt 3.3.7 erstellte Währungskonverter arbeitet mit dem Datentyp **double**, während für monetäre Anwendungen nachdrücklich der Typ **decimal** vorgeschlagen wird (siehe Abschnitte 4.3.4 und 4.3.5.2). Sorgen Sie daher durch einen Wechsel auf den Datentyp **decimal** für mehr Genauigkeit.

Die praktische Bedeutung dieser Maßnahme ist im Beispiel begrenzt. Man muss schon mit Billionen hantieren, damit sich nach der Division der Eingabe durch 1,95583 eine Abweichung im Cent-Bereich zeigt. Wie im Abschnitt 4.3.5.2 an einem anderen Beispiel zu sehen ist, wächst der Fehler mit der Anzahl der ausgeführten Operationen.

Außerdem sollte das Programm das Ergebnis auf volle Cent-Beträge runden. Das gelingt durch die Verwendung der statischen Methode **Format()** aus der Klasse **String**. Diese Methode arbeitet mit derselben Formatierungszeichenfolge und mit derselben Parameterliste wie die **Console**-Methode **WriteLine()** (siehe Abschnitt 4.2.2.1). In der folgenden Anweisung liefert **Format()**

```
ausgabe.Content = String.Format("{0:f2}", betrag / 1.95583m);
```

das Rechenergebnis `betrag / 1.95583m` mit 2 Nachkommastellen als Zeichenfolge.

Mit Hilfe der Zeichenfolgeninterpolation (siehe Abschnitt 4.2.2.2) lässt sich eine formatierte Zeichenfolge noch einfacher erzeugen:

```
ausgabe.Content = $"{betrag / 1.95583m:f2}";
```

Die Methode **Format()** hat den (allerdings nur selten relevanten) Vorteil, dass einige Varianten (Überladungen) dieser Methode über einen Parameter vom Typ **IFormatProvider** eine von den lokalen Regeln abweichende Formatierung erlauben (z. B. beim Dezimaltrennzeichen).

4.4 Einfache Techniken für Benutzereingaben

Für unsere Übungsprogramme brauchen wir gelegentlich eine einfache Möglichkeit, Benutzereingaben entgegenzunehmen.

4.4.1 Via Konsole

In der `Frage()` - Methode der Klasse `Bruch` aus dem Einleitungsbeispiel (siehe Abschnitt 1) wird die folgende Anweisung genutzt, um einen **int**-Wert via Konsole zu erfragen:

```
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 eine Verschachtelung zweier Methodenaufrufe vor, die bei Programmierern der kompakten Schreibweise wegen beliebt ist. Die folgende Variante ist für Einsteiger leichter zu verstehen, verursacht aber mehr Schreibarbeit:

```
string eingabe;
eingabe = Console.ReadLine();
zaehler = Convert.ToInt32(eingabe);
```

Die gerade vorgestellte Datenerfassungstechnik hat ein Problem mit schlecht instruierten oder nicht kooperativen Benutzern: Wird eine nicht konvertierbare Zeichenfolge abgeschickt, dann endet ein betroffenes Programm mit einem unbehandelten Ausnahmefehler, z. B.:

Quellcode	Ausgabe (Eingaben **fett**)

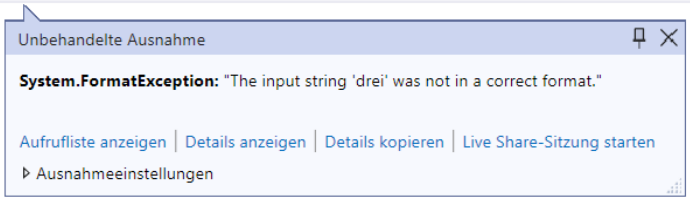
`Console.Write("Ihre Lieblingszahl? ");`
`int zahl = Convert.ToInt32(Console.ReadLine());`
`Console.WriteLine("Verstanden: " + zahl);`
	`Ihre Lieblingszahl?` **`drei`**
	
	`Unbehandelte Ausnahme:`
	`System.FormatException: Die Eingabezeichenfolge hat das falsche Format.`

Startet man das Programm aus dem Visual Studio im sogenannten *Debug-Modus*, indem man ...

- die Taste **F5** betätigt
- oder auf den Schalter ▶ klickt,

dann führt der Ausnahmefehler zur folgenden Anzeige:

```
Console.Write("Ihre Lieblingszahl? ");
int zahl = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Verstanden: " + zahl);
```



In dieser Lage lässt sich das Debugging z. B. folgendermaßen beenden:

- Tastenkombination **Umschalt+F5**
- Symbolschalter ■
- Menübefehl **Debuggen > Debugging beenden**

Um einen Programmabsturz durch fehlerhafte Eingaben zu verhindern, sind Programmiertechniken erforderlich, mit denen wir uns momentan noch nicht beschäftigen wollen, z. B.:

Quellcode	Ausgabe (Eingaben **fett**)

`int zahl;`
`bool ok;`
`do {`
`  try {`
`  Console.Write("Ihre Lieblingszahl? ");`
`  zahl = Convert.ToInt32(Console.ReadLine());`
`  Console.WriteLine("Verstanden: " + zahl);`
`  ok = true;`
`  } catch {`
`  Console.WriteLine("Falsche Eingabe!\n");`
`  ok = false;`
`  }`
`} while (!ok);`
	`Ihre Lieblingszahl? drei`
	`Falsche Eingabe!`
	
	`Ihre Lieblingszahl? 3`
	`Verstanden: 3`

Hier wird ein Ausnahmefehler per **catch**-Klausel abgefangen, damit er nicht zur Beendigung des Programms führt (siehe Abschnitt 13.2). In einer **do**-Schleife (siehe Abschnitt 4.7.3.3.2) wird die Frage nach der Lieblingszahl so lange wiederholt, bis der Benutzer eine gültige Antwort geliefert hat.

Auch die folgende Lösung unter Verwendung der vom Typ **Int32** angebotenen statischen Methode **TryParse()**, die auf einen nicht interpretierbaren ersten Parameter mit der Rückgabe **false** reagiert, benötigt unbekannte Programmiertechniken und etliche Zeilen:

```
bool ok;
do {
    Console.Write("Ihre Lieblingszahl? ");
    if (ok = Int32.TryParse(Console.ReadLine(), out int zahl))
        Console.WriteLine("Verstanden: " + zahl);
    else
        Console.WriteLine("Falsche Eingabe!");
} while (!ok);
```

In den Übungs- bzw. Demoprogrammen verwenden wir der Einfachheit halber ungesicherte **ToInt32** - Aufrufe bzw. analoge Varianten für andere Datentypen.

4.4.2 Via InputBox

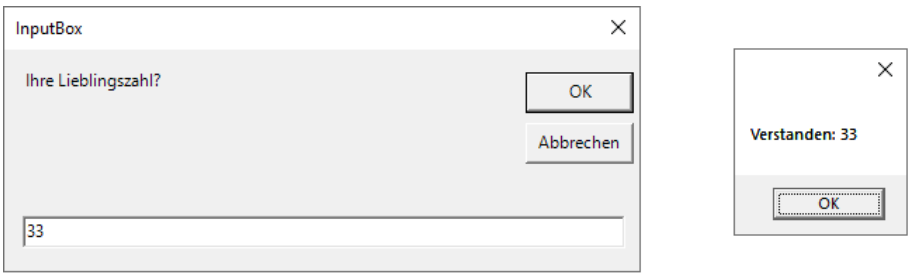
Um eine simple Anwendung mit grafischer Bedienoberfläche zu erstellen, muss man sich noch nicht mit den Details der WPF-Technik beschäftigen (vgl. Abschnitt 3.3.7):

- Die statische Methode **Show()** der Klasse **MessageBox** aus dem Namensraum **System.Windows** genügt zur Anzeige von Informationen.
- Mit Hilfe der statischen Methode **InputBox()** der Klasse **Interaction** aus dem Namensraum **Microsoft.VisualBasic** kann man Benutzereingaben per Dialogbox entgegennehmen.

Die Klasse **Interaction** ist als Migrationshilfe für Visual Basic 6 - Programmierer gedacht, kann aber auch in C# - Programmen genutzt werden. In der folgenden GUI-Alternative zum Beispielprogramm im Abschnitt 4.4.1[1]

```
using System.Windows;
using Microsoft.VisualBasic;

string eingabe = Interaction.InputBox("Ihre Lieblingszahl?", "InputBox", "", -1, -1);
int zahl = Convert.ToInt32(eingabe);
MessageBox.Show("Verstanden: " + zahl);
```

werden Ein- und Ausgabe per Dialogbox erledigt:



Damit das Programm unter .NET 7 erstellt werden kann, muss man in der Projektdatei ein windows-spezifisches **TargetFramework** einstellen und mit dem Element **UseWPF** die Verwendung der GUI-Bibliothek WPF aktivieren, z. B.:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

---

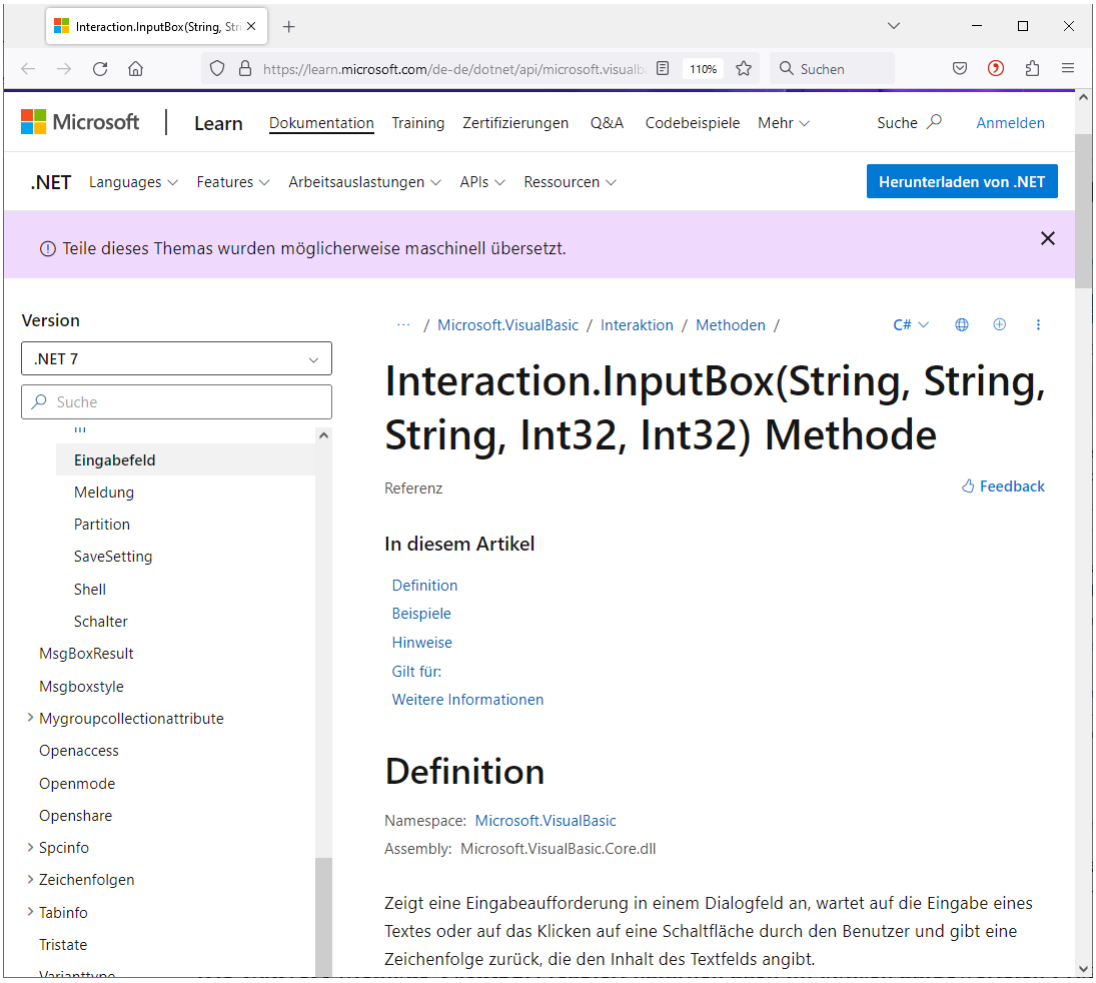
[1] Bei Ausnutzung der seit C# 9 (Anweisungen auf oberster Ebene) bzw. C# 10 (implizite **using**-Direktiven) erlaubten Weglassungen entsteht ein kurzer, aber für Einsteiger eventuell irritierender Quellcode. Daher präsentieren wir auch die vollständige Variante:

```
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);
    }
}
```

Anhand der Hilfefunktion der Entwicklungsumgebung kann man auf einfache Weise die Online-Dokumentation zur **InputBox()** – Methode anfordern (z. B. mit Informationen zu den Parametern und zum Rückgabewert). Bei markiertem Methodennamen

```
string eingabe = Interaction.InputBox("Ihre Lieblingszahl?", "InputBox", "", -1, -1);
```

ruft ein Druck auf die Taste **F1** die folgende Webseite auf:



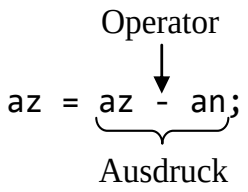
Die **Convert**-Methode **ToInt32()** reagiert natürlich auch im optisch aufgewerteten Programm auf nicht interpretierbare Benutzereingaben mit einem Ausnahmefehler (siehe Abschnitt 4.4.1). Im Kapitel 13 werden Sie lernen, solche Störungen des Programmablaufs per Ausnahmebehandlung abzufangen.

Zwar sieht die Ein- bzw. Ausgabe per GUI attraktiver aus, jedoch lohnt sich der erhöhte Aufwand bei Demo- bzw. Übungsprogrammen zu elementaren Sprachelementen kaum, sodass wir in der Regel mit Konsolenprogrammen arbeiten.

4.5 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 4.5 werden wir Ausdrücke als wichtige Bestandteile von C# - Anweisungen detailliert betrachten. Dabei lernen Sie elementare Datenverarbeitungs-Möglichkeiten kennen, die von sogenannten *Operatoren* mit ihren Argumenten realisiert werden, z. B. von den arithmetischen Operatoren (**+**, **-**, *****, **/)** für die Grundrechenarten. Im Verlauf des aktuellen Abschnitts werden Ihre Kenntnisse über die Datenverarbeitung mit C# erheblich wachsen. Der dabei zu investierende Aufwand lohnt sich, weil ein sicherer Umgang mit Operatoren und Ausdrücken eine unabdingbare Voraussetzung für das erfolgreiche Implementieren von Methoden und Eigenschaften ist. Dort werden die Handlungskompetenzen von Klassen bzw. 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** mit einem stets wohldefinierten **Datentyp** zu berechnen. Den zur Berechnung eines Werts geeigneten, aus Operatoren und zugehörigen Argumenten aufgebauten Teil einer Anweisung bezeichnet man als **Ausdruck**, z. B. in der folgenden Wertzuweisung:[1]



Durch diese Anweisung aus der `Kuerze()` - Methode unserer Klasse `Bruch` (siehe Abschnitt 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 typkompatibel sein müssen. Den Datentyp eines Ausdrucks bestimmen im Wesentlichen die Datentypen der Argumente, manchmal beeinflusst aber auch der Operator den Typ des Ausdrucks (z. B. bei einem Vergleichsoperator).

Schon bei einem Literal, einer Variablen oder einem Methodenaufruf haben wir es mit einem Ausdruck zu tun.[2]

Beispiel: `1.5`
Dieses Gleitkommaliteral ist ein Ausdruck mit dem Typ **double** und dem Wert 1,5 (vgl. Abschnitt 4.3.10.2).

Mit Hilfe diverser Operatoren entstehen komplexere Ausdrücke, deren Typen und Werte von den Argumenten und den Operatoren abhängen.

Beispiele:

- `2 * 1.5`
  Hier resultiert der **double**-Wert 3,0.
- `2 * 3`
  Hier resultiert der **int**-Wert 6.
- `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

---

[1] Im Abschnitt 4.5.8 werden Sie eine Möglichkeit kennenlernen, diese Anweisung etwas kompakter zu formulieren.

[2] Besteht ein Ausdruck aus einem Methodenaufruf mit dem Pseudorückgabetyp **void**, dann liegt allerdings *kein* Wert vor.

Operatoren haben jedoch zusätzlich einen **Nebeneffekt** auf eine als Argument fungierende *Variable*, z. B.:

```
int i = 12;
int j = i++;
```

In der zweiten Anweisung des Beispiels tritt der **Postinkrementoperator** `++` mit der **int**-Variablen `i` als Argument auf (siehe Abschnitt 4.5.1). 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 gebracht.

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:[1]

```
a + b
```

Manche Operatoren verarbeiten nur *ein* Argument und heißen daher **einstellig** bzw. **unär**. Als Beispiel haben wir eben den Postinkrementoperator kennengelernt. Ein weiteres Beispiel ist der **Negationsoperator**, der durch ein Ausrufezeichen dargestellt wird, *ein* Argument vom Typ **bool** erwartet und dessen Wahrheitswert wechselt (**true** und **false** vertauscht):

```
!cond
```

Wir werden auch noch einen *dreistelligen* (*ternären*) Operator kennenlernen.

Weil Ausdrücke von passendem Ergebnistyp als Argumente einer Operation erlaubt sind, können beliebig komplexe Ausdrücke aufgebaut werden. Unübersichtliche Exemplare sollten jedoch als potenzielle Fehlerquellen vermieden werden.

Die Operatorsymbole sind selbstisolierend, d.h. vor und nach ihnen sind keine Trennzeichen nötig. Bei den binären Operatoren ist eine Einrahmung durch Leerzeichen aber üblich, z. B.:

```
int s = a1 + a2;
```

4.5.1 Arithmetische Operatoren

Weil die arithmetischen Operatoren für die Grundrechenarten zuständig sind, müssen ihre Operanden (Argumente) einen numerischen Datentyp 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. C# kennt sieben Additionsoperator-Varianten für die eben aufgelisteten numerischen Datentypen (siehe C# - Sprachspezifikation in ECMA 2022, Abschnitt 11.9.5, S. 214ff). In der folgenden Auflistung tritt die noch ungewohnte Methodennotation für die Additionsoperator-Varianten auf, doch sind die Datentypen von Argumenten und Funktionswerten gut zu erkennen:[2]

---

[1] Wenn ein Operator in einer Klasse überladen wurde, dann kann er auch Argumente mit diesem Typ verarbeiten. Wie Sie wissen, ist in der Klasse **String** der Additionsoperator überladen (vgl. Abschnitt 4.2.1).

[2] Wir werden später von *Operatoren-Überladungen* sprechen (siehe Abschnitt 5.8.3) und z. B. auch für die Klasse `Bruch` eine Überladung des Additionsoperators definieren.

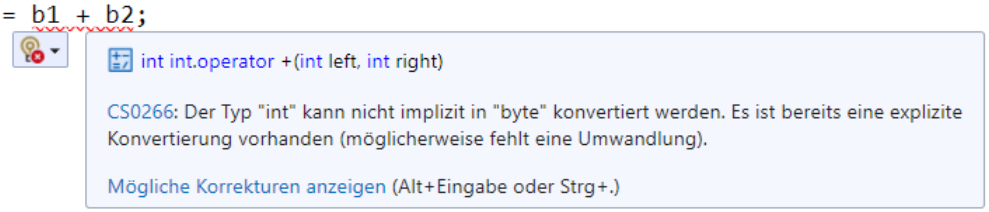
- Ganzzahladdition
  - **int operator +(int x, int y)**
  - **uint operator +(uint x, uint y)**
  - **long operator +(long x, long y)**
  - **ulong operator +(ulong x, ulong y)**
- Binäre Gleitkommaaddition
  - **float operator +(float x, float y)**
  - **double operator +(double x, double y)**
- Dezimale Gleitkommaaddition
  - **decimal operator +(decimal x, decimal y)**

Wie man sieht, ...

- gibt es nicht für jeden numerischen Datentyp eine spezielle Additionsoperator-Variante,
- müssen die beiden Argumente stets denselben Datentyp besitzen.

Wenn bei einem Argument oder bei beiden Argumenten der Datentyp nicht passt, findet nach Möglichkeit eine automatische (implizite) Typanpassung „nach oben“ statt (vgl. Abschnitt 4.5.7). 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;
byte s = b1 + b2;
```



Ist keine automatische Typanpassung möglich, beschwert sich der Compiler, z. B.:

```
double d = 1.25 + 1.3m;
```

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

`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 Verhalten beim Überschreiten des Wertebereichs (dem sogenannten *Überlauf*, siehe Abschnitt 4.6) werden Sie noch weitere Unterschiede zwischen der Ganzzahl- und der Gleitkommaarithmetik kennenlernen.

In der folgenden 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	**Ausgabe**
-Num	**Vorzeichenumkehr**	`int i = 2 , j = -3;` `Console.WriteLine($"{-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 erhöht *Var* um 1 *und* liefert *Var* + 1
	−−Var reduz. *Var* um 1 *und* liefert *Var* − 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 und* erhöht *Var* um 1 *Var−−* liefert *Var und* 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 4.7), wenn man ein Semikolon dahinter setzt, z. B.:

Quellcode	Ausgabe

`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 einige sinnvolle Anwendungen, z. B.:

- Man kann für eine ganze Zahl bequem feststellen, ob sie gerade (durch 2 teilbar) ist. Dazu prüft man, ob der Rest aus der Division durch 2 gleich 0 ist:

Quellcode	Ausgabe
`int i = 19;` `Console.WriteLine(i % 2 == 0);`	`False`

- Man kann bei einer Gleitkommazahl den gebrochenen Anteil ermitteln bzw. abspalten:[1]

Quellcode	Ausgabe

`double a = 7.1248239;`
`double rest = a % 1.0;`
`double ganz = a - rest;`
`Console.WriteLine($"{a} = {ganz:f0} + {rest:f7}");`
	`7,1248239 = 7 + 0,1248239`

4.5.2 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 im Abschnitt 4.4.1 vorgestellte Methode **Convert.ToInt32()** liefert z. B. einen **int**-Wert, sofern die als Parameter übergebene Zeichenfolge als ganze Zahl im **int**-Wertebereich (siehe Tabelle im Abschnitt 4.3.4) interpretierbar ist. Bei einer Methodendefinition ist der Datentyp der Rückgabe anzugeben (siehe Syntaxdiagramm im Abschnitt 4.1.3.2).
- Liefert eine Methode dem Aufrufer *kein* Ergebnis, dann ist in der Definition der Pseudo-Rückgabetyp **void** anzugeben.
- Neben der Wertrückgabe hat ein Methodenaufruf oft weitere Effekte, z. B. auf die Merkmalsausprägungen eines 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) Bindungskraft (Priorität) besitzt (vgl. Abschnitt 4.5.10).

Bei passendem Rückgabetyp darf ein Methodenaufruf auch als Argument für komplexere Ausdrücke oder für übergeordnete Methodenaufrufe verwendet werden (siehe Abschnitt 5.3.1.2). Bei einer Methode *ohne* Rückgabe resultiert ein Ausdruck vom Typ **void**, der nicht als Argument für Operatoren oder andere Methoden taugt.

Ein Methodenaufruf mit angehängtem Semikolon stellt eine **Anweisung** dar (vgl. Abschnitt 4.7), was Sie z. B. bei den zahlreichen Einsätzen der statischen Methode **WriteLine()** aus der Klasse **Console** in unseren Beispielprogrammen beobachten konnten.

---

[1] Der (gerundete) ganzzahlige Anteil eines **double**- oder **decimal**-Werts 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

Mit den arithmetischen Operatoren lassen sich nur elementare mathematische Probleme lösen. Darüber hinaus stellt die BCL eine große Zahl mathematischer Standardfunktionen (z. B. Potenzfunktion, Logarithmus, Wurzel, trigonometrische Funktionen) über statische Methoden der Klasse **Math** im Namensraum **System** zur Verfügung. Im folgenden Beispiel wird die Methode **Pow()** zur Berechnung der allgemeinen Potenzfunktion (b^e) genutzt:

Quellcode	Ausgabe
`Console.WriteLine(Math.Pow(2.0, 3.0));`	`8`

Alle **Math**-Methoden sind als **static** definiert, werden also von der Klasse selbst ausgeführt.

Im Beispiel 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 Einsteiger leichter nachvollziehbares Äquivalent zum obigen **WriteLine()** - Aufruf könnte z. B. so aussehen:

```
double d = Math.Pow(2.0, 3.0);
Console.WriteLine(d);
```

4.5.3 Vergleichsoperatoren

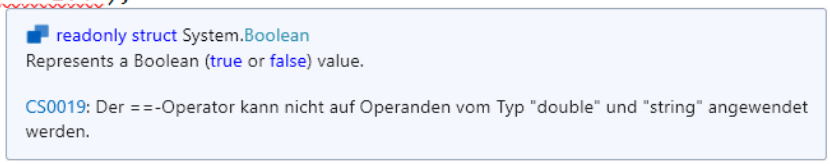
Durch die Anwendung eines *Vergleichsoperators* auf zwei komparable (miteinander vergleichbare) Ausdrücke entsteht ein **Vergleich**. Dies ist ein einfacher **logischer Ausdruck** (vgl. Abschnitt 4.5.5). Folglich kann ein Vergleich die booleschen Werte **true** (wahr) und **false** (falsch) annehmen und zur Formulierung einer Bedingung verwendet werden. Das folgende Beispiel dürfte verständlich sein, obwohl die **if**-Anweisung noch nicht behandelt wurde:

```
if (arg > 0)
  Console.WriteLine(Math.Log(arg));
```

In der anschließenden Tabelle mit den von C# unterstützten Vergleichsoperatoren stehen ...

- *Expr1* und *Expr2* für miteinander vergleichbare Ausdrücke
  Es muss möglich sein, den Typ eines Operanden per Typumwandlung (siehe Abschnitt 4.5.7) in den Typ des anderen Operanden zu konvertieren. Das ist im folgenden Beispiel *nicht* möglich:

```
Console.WriteLine(2.4 == "2.4");
```



- *Num1* und *Num2* für numerische Ausdrücke (mit dem Datentyp **sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**, **float, double** oder **decimal**)

Operator	**Bedeutung**	Beispiel	
		Quellcode	**Ausgabe**
Expr1 == *Expr2*	**gleich**	`string a = "2.4",s = "2.",b = s + "4";` `Console.WriteLine(a == b);`	`True`
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 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 unangenehmer Logikfehler resultieren, also ein irreguläres Verhalten des Programms (vgl. Abschnitt 3.1.6 zur Unterscheidung von Syntax- und Logikfehlern). Im ersten **WriteLine()** - Aufruf des folgenden Beispiels wird das Ergebnis eines Vergleichs auf die Konsole geschrieben:[1]

Quellcode	Ausgabe

`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 4.5.8) mit dem Typ **int** und dem Wert 2:

Quellcode	Ausgabe

`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 Programms unangenehme Folgen haben kann.

Bei einem intendierten Vergleich einer Variablen mit einem *Literal* schützt das Vertauschen der Operanden vor dem Fehler durch ein vergessenes Gleichheitszeichen, weil die versehentlich zustande gekommene Zuweisung (siehe Abschnitt 4.5.8) als linken Operanden eine Variable erwartet und folglich von der Entwicklungsumgebung bzw. vom Compiler als fehlerhaft erkannt wird:[2]

```
Console.WriteLine(2 = i);
```

readonly struct System.Int32
Represents a 32-bit signed integer.

CS0131: Die linke Seite einer Zuweisung muss eine Variable, eine Eigenschaft oder ein Indexer sein.

Dass der Vergleich der beiden **String**-Variablen **a** und **b** zum Ergebnis **True** führt, ist durchaus nicht selbstverständlich.

```
string a = "2.4", s = "2.", b = s + "4";
Console.WriteLine(a == b);
```

In der Programmiersprache Java (siehe z. B. Baltes-Götz & Götz 2023) werden die beiden **String**-Referenzvariablen vom Identitätsoperator als *ungleich* beurteilt, weil sie auf verschiedene Objekte zeigen. In C# orientiert sich der Identitätsoperator jedoch nicht an den Adressen der Objekte, sondern an den Inhalten.

4.5.4 Identitätsprüfung bei Gleitkommawerten

Bei den *binären* Gleitkommatypen (**float** und **double**) sind simple Identitätstests wegen technisch bedingter Abweichungen von der reinen Mathematik unbedingt zu unterlassen, z. B.:

---

[1] Wir wissen schon aus dem Abschnitt 4.2.1, dass **WriteLine()** einen Ausdruck mit beliebigem Typ verarbeiten kann, wobei automatisch eine Zeichenfolgen-Repräsentation erstellt wird.

[2] Diese schlaue Idee stammt von Marian Peters.

Quellcode	Ausgabe

`const double usd = 1.0e-14; // Unterschiedsschwelle 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 des Datentyps **double** (mit mindestens 15 signifikanten Dezimalstellen) zum Ergebnis `False`. Wenn man die im Abschnitt 4.3.5.1 beschriebenen Genauigkeitsprobleme bei der Speicherung von binären Gleitkommazahlen berücksichtigt, dann 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:

Quellcode	Ausgabe

`double d1 = 10.0 - 9.9;`
`double d2 = 0.1;`
`Console.WriteLine($"{d1:f20}");`
`Console.WriteLine($"{d2:f20}");`
	`0,09999999999999964473`
	`0,10000000000000000555`

Im gespeicherten Berechnungsergebnis 10,0 - 9,9 steckt ein anderer Fehler als im Speicherabbild der Zahl 0,1. Weil die Vergleichspartner nicht Bit für Bit identisch sind, meldet der Identitätsoperator das Ergebnis **false**.

Für Anwendungen im Bereich der Finanzmathematik wurde schon im Abschnitt 4.3.5 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 des relativ kleinen Wertebereichs, des großen Speicherbedarfs und des hohen Rechenzeitaufwands nicht für alle Anwendungen.

Den etwas anstrengenden Rest des Abschnitts kann überspringen, wer aktuell keinen Algorithmus mit auf numerische Identität zu prüfenden **double**-Werten zu implementieren hat.

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 4.3.5.1) dann als **numerisch identisch** betrachtet, wenn der relative Abweichungsbetrag kleiner als $1{,}0 \cdot 10^{-14}$ ist:

$$\left|\frac{d_1 - d_2}{d_1}\right| < 1{,}0 \cdot 10^{-14}$$

Die Vergabe der d_1-Rolle, also die Wahl des Nenners, 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 gelegentlich anzutreffenden Vorschlag, die betragsmäßige Differenz

$$|d_1 - d_2|$$

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:

$$d_1 = (1 + m + e_1)\ 2^k \text{ und } d_2 = (1 + m + e_2)\ 2^k$$

Bei einem normalisierten **double**-Wert (mit 52 Mantissen-Bits) kann aufgrund der begrenzten Speichergenauigkeit als maximaler absoluter Mantissenfehler ε der halbe Abstand zwischen zwei benachbarten Mantissenwerten auftreten:

$$\varepsilon = 2^{-53} \approx 1{,}1 \cdot 10^{-16}$$

Für den Betrag der technisch bedingten relativen Abweichung von zwei eigentlich identischen normalisierten Werten (mit einer Mantisse im Intervall [1, 2)) gilt die Abschätzung:

$$\left|\frac{d_1 - d_2}{d_1}\right| = \left|\frac{e_1 - e_2}{1 + m + e_1}\right| \le \frac{|e_1| + |e_2|}{|1 + m + e_1|} \le \frac{2 \cdot \varepsilon}{|1 + m + e_1|} \le 2 \cdot \varepsilon \quad (\text{wegen} (1 + m + e_1) \in [1, 2))$$

Die oben vorgeschlagene Schwelle $1{,}0 \cdot 10^{-14}$ berücksichtigt über den Speicherfehler hinaus auch noch eingeflossene Rechnungsungenauigkeiten. Mit welcher Fehlerkumulation bzw. -verstärkung zu rechnen ist, hängt vom konkreten Algorithmus ab, sodass die Unterschiedlichkeitsschwelle eventuell angehoben werden muss. Immerhin hängt sie (anders als bei einem Kriterium auf Basis des Betrags 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 4.3.5.1) 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 gerichteten 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.

4.5.5 Logische Operatoren

Aus dem Abschnitt 4.5.3 wissen wir, dass jeder Vergleich (z. B. `arg > 0`) bereits ein logischer Ausdruck ist, also die Werte **true** und **false** annehmen kann. Durch Anwendung der logischen Operatoren (Negation, logisches UND, logisches ODER, exklusives ODER) 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 Erläuterungen und Beispielen zu den logischen Operatoren:

Operator	**Bedeutung**	Beispiel	
		Quellcode	**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 Operanden 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 Operanden 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 Operand 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`

Operator	Bedeutung	Beispiel	
		Quellcode	**Ausgabe**
			4

La1 | *La2*	**Logisches ODER mit unbedingter Auswertung**
	La1 | *La2* ist genau dann wahr, wenn mindestens ein Operand 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** Operand wahr ist, wenn also die Operanden verschiedene Wahrheitswerte haben.	`Console.WriteLine(true ^ true);`	`False`

Der Unterschied zwischen den beiden logischen UND-Operatoren **&&** und **&** bzw. zwischen den beiden logischen ODER-Operatoren || und | ist für Einsteiger vielleicht wenig beeindruckend, 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 **&**-Operanden 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 **&&**-Operanden das Gesamtergebnis **false** feststeht:

```
b && i++ > 3
```

Man spricht hier von einer *Kurzschlussauswertung* (engl.: *short-circuiting*). Das vom Programmierer nicht erwartete Ausbleiben einer Auswertung (z. B. bei `i++`) kann erhebliche Auswirkungen auf die Programmausführung haben.

Dank der beim Operator **&&** realisierten bedingten Auswertung kann man sich im rechten Operanden darauf verlassen, dass der linke Operand 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, dann darf der rechte Ausdruck nicht ausgewertet werden, weil eine Längenanfrage (per Eigenschaftszugriff) an ein nicht existentes Objekt zu einem Laufzeitfehler führen würde.

Mit der Entscheidung, grundsätzlich die unbedingte Operatorvariante zu verwenden, verzichtet man auf die eben beschriebene Option, im rechten Ausdruck den Wert **true** des linken Ausdrucks voraussetzen zu können, und man nimmt (mehr oder weniger relevante) Leistungseinbußen durch überflüssige Auswertungen des rechten Ausdrucks in Kauf. Eher empfehlenswert ist der Verzicht auf Nebeneffekt-Konstruktionen im Zusammenhang mit bedingt arbeitenden Operatoren.

Wie der Tabelle auf Seite 217 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 4.5.6). 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.

4.5.6 Bitorientierte Operatoren

Über unseren momentanen Bedarf hinausgehend bietet C# einige Operatoren zur bitweisen Analyse und Manipulation von Variableninhalten. Solche Techniken werden nur bei speziellen Anwendungen benötigt, sodass der Abschnitt beim ersten Lesen des Manuskripts übersprungen werden kann.

Statt einer systematischen Darstellung der verschiedenen Operatoren (siehe z. B. C# - Sprachspezifikation in ECMA 2022, Kapitel 11) beschränken wir uns auf ein Beispielprogramm, das zudem nützliche Einblicke in die Speicherung von **char**-Daten im Arbeitsspeicher eines Computers erlaubt. Allerdings sind das Beispiel und die zugehörigen Erläuterungen mit einigen technischen Details belastet. Wenn Ihnen der Sinn momentan nicht danach steht, können Sie den aktuellen Abschnitt ohne Sorge um den weiteren Lernerfolg an dieser Stelle verlassen.

Das folgende Programm `CharBits` liefert die Unicode-Codierung 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 im Abschnitt 4.7.3.1 offiziell vorgestellt wird. Im Beispiel startet die **int**-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 (Eingaben **fett**)

`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--)`
`  Console.Write((1 << i & cbit) >> i);`
`  }`
`}`
	`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` Ziffern verworfen werden und auf der rechten Seite `i` Nullen nachrücken. Von den 32 Bits, die ein **int**-Wert insgesamt belegt (siehe Abschnitt 4.3.4), 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 in C# leider in doppelter Bedeutung verwendet: Wenn beide Argumente vom Typ **bool** sind, wird **&** als *logischer* Operator interpretiert (siehe Abschnitt 4.5.5). 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 1 enthält, wenn *beide* Argumentmuster an dieser Stelle eine 1 besitzen. Hat in einem Programmablauf die **char**-Variable `cbit` z. B. den Wert `'x'` erhalten (dezimale Unicode-Zeichensatznummer 120), dann ist dieses Bitmuster

```
0000000001111000
```

im Spiel, und `1 << i & cbit` liefert z. B. bei `i = 6` das Muster:

```
0000000001000000
```

Per **Rechts-Shift - Operator** >> werden die Bits um `i` Stellen nach rechts verschoben, wobei am rechten Rand `i` Ziffern verworfen werden und auf der linken Seite `i` Nullen nachrücken:

```
(1 << i & cbit) >> i
```

Die signifikante Ziffer an der Position `i` wird zum rechten Rand befördert, sodass insgesamt ein Bitmuster vom Typ **int** entsteht, das den Wert 0 oder 1 besitzt.[1] Der Wert 1 resultiert genau dann, wenn das zum aktuellen `i`-Wert korrespondierende Bit in der Binärdarstellung des untersuchten Zeichens den Wert 1 hat.

Ein Visual Studio - Projekt mit dem Programm `CharBits` ist hier zu finden:

…\BspUeb\Elementare Sprachelemente\Bits\CharBits

4.5.7 Typumwandlung (Casting) bei elementaren Datentypen

Wie Sie aus dem Abschnitt 4.3.1 wissen, ist in C# der Datentyp einer Variablen in der Regel unveränderlich, und dieses Prinzip wird im aktuellen Abschnitt keineswegs aufgeweicht. Es gibt aber gelegentlich einen Grund dafür, z. B. den Inhalt einer **int**-Variablen in eine **double**-Variable zu übertragen. Aufgrund der abweichenden Speichertechniken ist dann eine Typanpassung fällig. Die geschieht manchmal automatisch auf Initiative des Compilers, kann aber auch vom Programmierer explizit angeordnet werden.

4.5.7.1 Automatische erweiternde Typanpassung

Bei der Auswertung des Ausdrucks

```
2.3 / 7
```

trifft der Divisionsoperator auf einen **double**- und einen **int**-Operanden. Der C# - Compiler kennt sieben vordefinierte Divisionsoperator-Varianten (siehe C# - Sprachspezifikation in ECMA 2022, Abschnitt 11.9.3, S. 211f):

- 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 Gleitkommadivision
  - **float operator /(float x, float y)**
  - **double operator /(double x, double y)**
- Dezimale Gleitkommadivision
  - **decimal operator /(decimal x, decimal y)**

Wenn bei einem Argument oder bei beiden Argumenten der Datentyp nicht passt, dann findet nach Möglichkeit eine automatische (implizite) Typanpassung „nach oben“ statt. Im Beispiel wird das **int**-Argument in den Datentyp **double** gewandelt und eine binäre Gleitkommadivision durchgeführt.

---

[1] Die runden Klammern sind erforderlich, um die korrekte Auswertungsreihenfolge zu erreichen (siehe Abschnitt 4.5.10).

In vergleichbaren Situationen (z. B. bei Wertzuweisungen) kommt es automatisch zu den folgenden **erweiternden Typanpassungen**:[1]

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
ulong	float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
float	double

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	Ausgabe

`long i = 9223372036854775313;`
`double d = i;`
`Console.WriteLine(i);`
`Console.WriteLine($"{d:f}");`
	`9223372036854775313`
	`9223372036854780000,00`

Eine Abweichung von 4687 (z. B. Meter) kann durchaus unerfreuliche Konsequenzen haben (z. B. beim Versuch, auf einem Astroiden zu landen).

Weil eine **char**-Variable die Unicode-Nummer eines Zeichens speichert, macht ihre Konvertierung in numerische Typen kein Problem, z. B.:

Quellcode	Ausgabe
`System.Console.WriteLine("x/2 \t= " + 'x' / 2);` `System.Console.WriteLine("x*0,27 \t= " + 'x' * 0.27);`	`x/2 = 60` `x*0,27 = 32,4`

4.5.7.2 Explizite Typumwandlung

Gelegentlich gibt es gute Gründe, über den sogenannten **Casting-Operator** eine *explizite* Typumwandlung zu erzwingen. Im nächsten Beispiel wird mit

```
(int)'x'
```

die **int**-erpretation des (aus dem Abschnitt 4.5.6 bekannten) Bitmusters zum kleinen „x“ vorgenommen, damit Sie nachvollziehen können, warum das Beispielprogramm im vorigen Abschnitt beim „Halbieren“ dieses Zeichens auf den Wert 60 kam:

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/conversions#implicit-conversions

Quellcode	Ausgabe

`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, weil z. B. bei einem Methodenaufruf für einen Parameter ein ganzzahliger Datentyp benötigt wird. Dabei werden die Nachkommastellen abgeschnitten.

Bei nicht-negativen Gleitkommawerten resultiert ein Runden auf die nächstgelegene ganze Zahl, wenn man vor der Typkonvertierung 0,5 addiert.

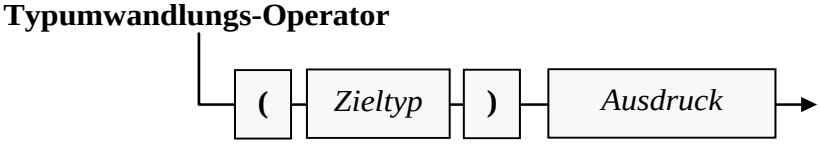
Es ist auf jeden Fall zu beachten, dass dabei eine **einschränkende Konvertierung** stattfindet, und dass die zu erwartende Gleitkommazahl im Wertebereich des Ganzzahltyps liegen muss. Wie die letzte Ausgabe zeigt, sind kapitale Programmierfehler möglich, wenn die Wertebereiche der beteiligten Variablen bzw. Datentypen nicht beachtet werden und bei der Zielvariablen ein Überlauf auftritt (vgl. Abschnitt 4.6.1). 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. Die kritische Typumwandlung hatte bei der langsameren Rakete Ariane 4 noch keine Probleme gemacht. Offenbar sind profunde Kenntnisse über elementare Sprachelemente unverzichtbar für eine erfolgreiche Raketenkonstruktion.

Später wird sich zeigen, dass auch zwischen Referenztypen gelegentlich eine explizite Wandlung (auf Verantwortung des Programmierers) erforderlich ist.

Welche expliziten Typkonvertierungen in C# erlaubt sind, ist z. B. der C# - Sprachspezifikation zu entnehmen (ECMA 2022, Abschnitt 11.3, S. 114ff).

Das Syntaxdiagramm zur expliziten Typumwandlung ist sehr übersichtlich:



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 4.5.1), ohne ein Wertebereichsproblem befürchten zu müssen, z. B.:[1]

Quellcode	Ausgabe

`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`

4.5.8 Zuweisungsoperatoren

Bei den ersten Erläuterungen zur Wertzuweisung (vgl. Abschnitt 4.3.6) blieb aus didaktischen Gründen unerwähnt, dass eine Wertzuweisung ein *Ausdruck* ist, dass wir es also mit dem binären (zweistelligen) Operator „=“ zu tun haben, für den die folgenden 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.

Im folgenden Beispiel fungiert ein Zuweisungsausdruck als Parameter für einen **WriteLine()** - Methodenaufruf:

Quellcode	Ausgabe

`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 (identisch mit dem zugewiesenen Wert), *und* die Variable `ivar` wird verändert.

Eine Zuweisung kann als Operand in einen übergeordneten Ausdruck integriert werden, z. B.:

Quellcode	Ausgabe

`int i = 1, j = 2;`
`i = j = 5;`
`Console.WriteLine(i + "\n" + j);`
	`5`
	`5`

Beim mehrfachen Auftreten des Zuweisungsoperators in einem Ausdruck ist seine **Rechts-Assoziativität** relevant (siehe Abschnitt 4.5.10). Im Beispiel

```
i = j = 5
```

kommt es zu der folgenden impliziten Klammerung (bzw. Zuordnung der Operatoren):

```
i = (j = 5)
```

Zunächst wird der Ausdruck `j = 5` ausgewertet, wobei …

- die Variable `j` den Wert 5 erhält,
- und ein Zwischenergebnis mit dem Datentyp **int** und dem Wert 5 entsteht.

---

[1] Der (gerundete) ganzzahlige Anteil eines **double**-Wertes lässt sich auch über die statische Methode **Round()** bzw. **Truncate()** aus der Klasse **Math** ermitteln.

Bei der Auswertung des verbliebenen Ausdrucks erhält die Variable `i` den Wert 5:

```
i = 5
```

Anweisungen der Art

```
i = j = 5;
```

stammen nicht aus einem Kuriositätenkabinett, sondern sind in C# - Programmen gelegentlich anzutreffen, weil im Vergleich zur Alternative

```
j = 5;
i = 5;
```

Schreibaufwand gespart wird.

Wie wir seit dem Abschnitt 4.3.6 wissen, stellt ein Zuweisungsausdruck bereits eine vollständige **Anweisung** dar, sobald man ein Semikolon dahinter setzt. Dies gilt auch für die Inkrement- bzw. Dekrementausdrücke in der Prä- oder Post-Variante (vgl. Abschnitt 4.5.1) sowie für Methodenaufrufe, jedoch *nicht* für die anderen Ausdrücke, die im Abschnitt 4.5 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, die als *Aktualisierungsoperatoren* oder als *Verbundzuweisungs-Operatoren* (engl.: *compound assignment operators*) bezeichnet werden. Im Beispiel werden durch die Verwendung des Operators **`*=`**

```
j *= i;
```

zwei Vorteile erzielt:

- prägnanterer Quellcode
- nur *ein* Zugriff auf die Variable `j` bei der Auswertung der Operanden
  Wie bei jeder binären Operation wird auch bei einer Zuweisungsoperation der linke Operand zuerst ausgewertet (siehe Abschnitt 4.5.10.1). Während bei der einfachen Zuweisungsoperation nur überprüft wird, ob es sich beim linken Operanden um eine veränderbare Variable handelt, wird bei einer Aktualisierungsoperation zusätzlich der Wert dieser Variablen gespeichert, sodass bei der Auswertung des rechten Operanden kein erneuter Zugriff auf die zu aktualisierende Variable erforderlich ist.

In der folgenden Tabelle steht *Var* für eine numerische Variable und *Expr* für einen typkompatiblen Ausdruck:

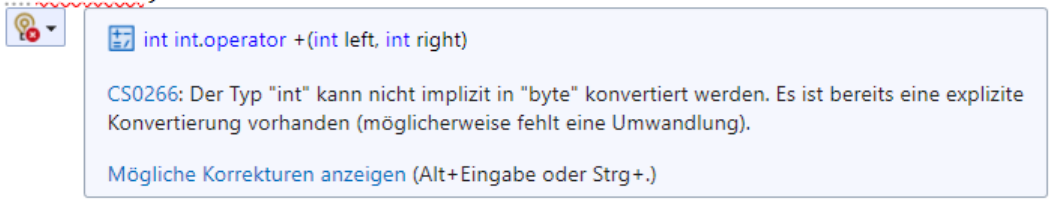
Operator	**Bedeutung**	Beispiel	
		Quellcode	**Neuer Wert von `i`**
Var **`+=`** *Expr*	*Var* erhält den neuen Wert *Var* + *Expr*.	`int i = 2;` `i += 3;`	`5`
Var **`-=`** *Expr*	*Var* erhält den neuen Wert *Var* − *Expr*.	`int i = 10, j = 3;` `i -= j * j;`	`1`
Var **`*=`** *Expr*	*Var* erhält den neuen 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 4.5.1), akzeptiert der Compiler den äquivalenten Ausdruck mit Aktualisierungsoperator:

```
b1 += b2;
```

Dabei kann es zu einem Ganzzahlüberlauf kommen, z. B.:

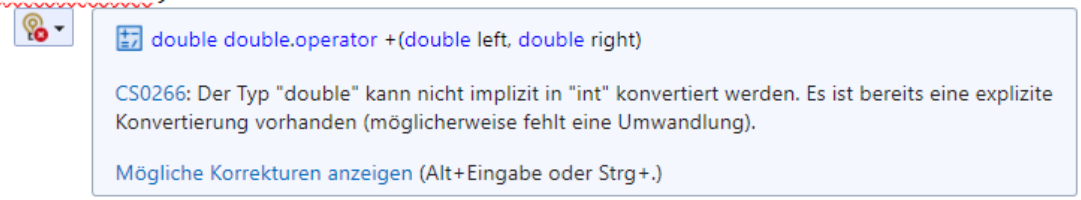
Quellcode	Ausgabe

`byte b1 = 200, b2 = 200;`
`b1 += b2;`
`Console.WriteLine(b1);`
	`144`

Wie der Abschnitt 4.6.1 zeigen wird, muss man bei der Verarbeitung von integralen Datentypen generell das Risiko eines Überlaufs beachten.

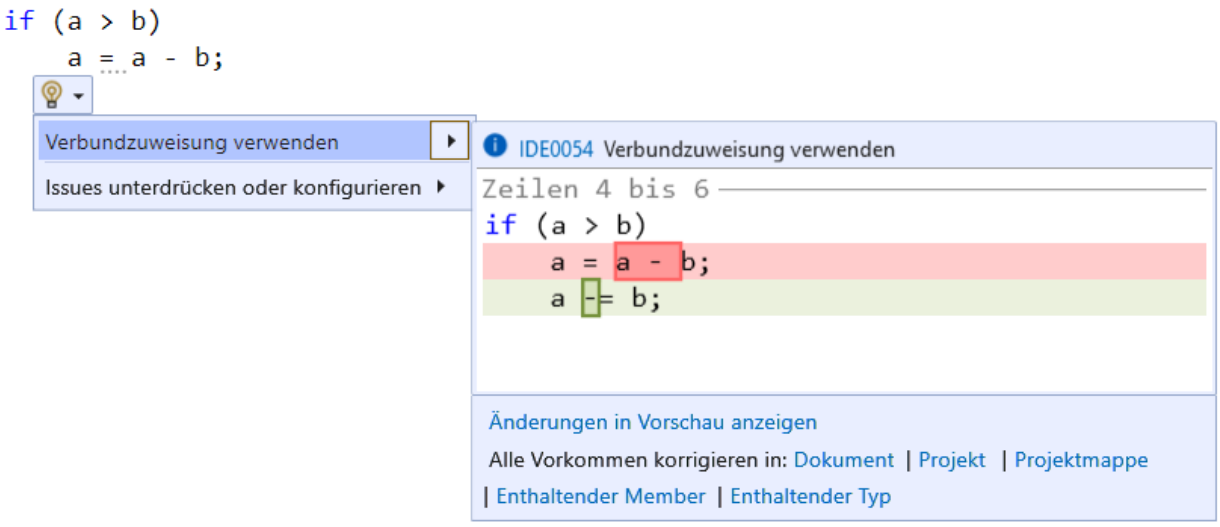
C# ist bei den Verbundzuweisungen vorsichtiger als andere Programmiersprachen und verweigert z. B. die Übersetzung der folgenden Anweisungen,

```
int ivar = 1;
double dvar = 3_000_000_000.0;
ivar += dvar;
```



während der Java-Compiler denselben Quellcode akzeptiert (siehe Baltes-Götz & Götz 2023, Abschnitt 3.5.8). Wie sich eine einschränkende Konvertierung in der Raketenentwicklung auswirken kann, wurde im Abschnitt 4.5.7.2 beschrieben.

Wer sich dafür entscheidet, auf die Verbundzuweisungen zu verzichten, wird vom Visual Studio per Voreinstellung zu ihrer Verwendung aufgefordert, z. B.:



4.5.9 Konditionaloperator

Der **Konditionaloperator** erlaubt eine sehr kompakte Schreibweise, wenn beim neuen Wert für eine Zielvariable bedingungsabhängig zwischen zwei Ausdrücken zu entscheiden ist, z. B.

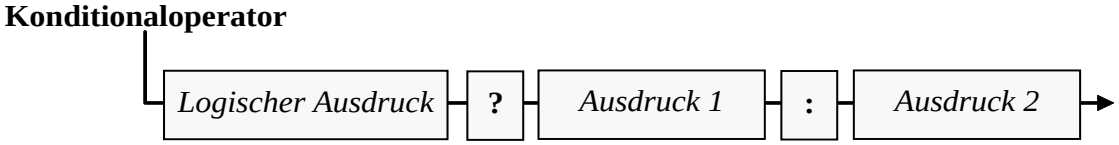
$$i = \begin{cases} i + j & \text{falls } k > 0 \\ i - j & \text{sonst} \end{cases}$$

In C# ist für diese Zuweisung mit Fallunterscheidung nur eine einzige Zeile erforderlich:

Quellcode	Ausgabe

`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*, anderenfalls den Wert von *Ausdruck 2*.

Die Frage nach dem Typ eines Konditionalausdrucks ist etwas knifflig, und in der C# - Sprachspezifikation werden etliche Fälle unterschieden (ECMA 2022, Abschnitt 11.15). Es liegt an Ihnen, sich auf den einfachsten und wichtigsten Fall zu beschränken: Wenn der zweite und der dritte Operand denselben Typ haben, dann ist dies auch der Typ des Konditionalausdrucks.

4.5.10 Auswertungsreihenfolge

Bisher haben wir zusammengesetzte Ausdrücke mit *mehreren* Operatoren und das damit verbundene Problem der Auswertungs*reihenfolge* nach Möglichkeit gemieden. Wie sich gleich zeigen wird, sind für Schwierigkeiten und Fehler bei der Verwendung zusammengesetzter Ausdrücke die folgenden Gründe hauptverantwortlich:

- Komplexität des Ausdrucks (Anzahl der Operatoren, Schachtelungstiefe)
- Operanden mit Nebeneffekten (z. B. Ausdrücke mit In- oder Dekrementoperationen)

Um Problemen aus dem Weg zu gehen, sollte man also eine übertriebene Komplexität vermeiden und auf Nebeneffekte möglichst verzichten.

4.5.10.1 Regeln

In diesem Abschnitt werden die Regeln vorgestellt, nach denen der C# - Compiler einen Ausdruck mit mehreren Operatoren auswertet.

1) Klammern

Wenn aus den anschließend erläuterten Regeln zur Bindungskraft und Assoziativität der beteiligten Operatoren nicht die gewünschte Operandenzuordnung bzw. implizite Klammerung resultiert, dann greift man mit runden Klammern steuernd ein, wobei auch eine Schachtelung erlaubt ist. Durch Klammern werden Ausdrücke zu *einem* Operanden zusammengefasst, sodass die *internen* Operationen ausgeführt sind, bevor der Klammerausdruck von einem *externen* Operator verarbeitet wird.

Durch syntaktisch überflüssige Klammern kann die Lesbarkeit eines komplizierten Ausdrucks verbessert werden.

2) Bindungskraft (Priorität)

Steht ein Operand (ein Ausdruck) zwischen zwei Operatoren, dann wird er dem Operator mit der stärkeren Bindungskraft (siehe Tabelle im Abschnitt 4.5.10.3) zugeordnet. Mit den numerischen Variablen `a`, `b` und `c` als Operanden wird z. B. der Ausdruck

```
a + b * c
```

nach der Regel *Punktrechnung geht vor Strichrechnung* interpretiert als

```
a + (b * c)
```

In der Konkurrenz um die Zuständigkeit für den Operanden `b` hat der Multiplikationsoperator Vorrang gegenüber dem Additionsoperator.

Die implizite Klammerung aufgrund der Bindungskraft kann durch eine explizite Klammerung dominiert werden:

```
(a + b) * c
```

3) Assoziativität (Orientierung)

Steht ein Operand zwischen zwei Operatoren mit *gleicher* Bindungskraft, dann entscheidet deren Assoziativität (Orientierung) über die Zuordnung des Operanden:

- Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren *links-assoziativ*. Z. B. wird

  ```
  x - y - z
  ```

  ausgewertet als

  ```
  (x - y) - z
  ```

  Diese implizite Klammerung kann durch eine explizite Klammerung dominiert werden:

  ```
  x - (y - z)
  ```

- Die Zuweisungsoperatoren sind *rechts-assoziativ*. Z. B. wird

  ```
  a += b -= c = d
  ```

  ausgewertet als

  ```
  a += (b -= (c = d))
  ```

  Diese implizite Klammerung kann *nicht* durch eine explizite Klammerung geändert werden, weil der linke Operand einer Zuweisung eine Variable oder eine Eigenschaft sein muss.

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 Assoziativgesetz, sodass 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 technischen Gründen für die Addition und die Multiplikation von *Gleitkommatypen* nur approximativ.

4) Links vor rechts bei der Auswertung der Operanden eines Operators

Bevor ein Operator ausgeführt werden kann, müssen erst seine Operanden ausgewertet sein. In C# ist sichergestellt, dass die Operanden eines Operators von links nach rechts ausgewertet werden. Bei einem binären Operator wird also zuerst der linke Operand ausgewertet, dann der rechte.[1] Im folgenden Beispiel tritt der Ausdruck `++ivar` als rechter Operand einer Multiplikation auf. Die hohe Bindungskraft (Priorität) des Präinkrementoperators (siehe Tabelle im Abschnitt 4.5.10.3) führt *nicht* dazu, dass sich der Nebeneffekt des Ausdrucks `++ivar` auf den linken Operanden der Multiplikation auswirkt:

Quellcode	Ausgabe

`int ivar = 2;`
`int erg = ivar * ++ivar;`
`Console.WriteLine($"{erg}\n{ivar}");`
	`6`
	`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:
  - Die Präinkrementoperation hat einen Nebeneffekt auf die Variable `ivar`.
  - Der Ausdruck `++ivar` hat den Wert 3.
- Die Ausführung der Multiplikationsoperation liefert schließlich das Endergebnis 6.

Wie eine Variation des letzten Beispiels zeigt, kann sich ein Nebeneffekt im *linken* Operanden einer binären Operation auf den rechten Operanden auswirken:

---

[1] In den folgenden Fällen 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 (siehe Abschnitt 4.5.5).
- Bei der Auswertung des linken Operanden kommt es zu einem Ausnahmefehler (siehe Kapitel 13).

Quellcode	Ausgabe

`int ivar = 2;`
`int erg = ++ivar * ++ivar;`
`Console.WriteLine($"{erg}\n{ivar}");`
	`12`
	`4`

Auch bei einem rechts-assoziativen Operator wird der linke Operand vor dem rechten ausgewertet, sodass im folgenden Beispiel mit der numerischen Variablen `a`

```
a += ++a
```

diese Auswertungs- bzw. Ausführungsreihenfolge resultiert:

```
a, ++a, +=
```

Als neuer Wert von `a` entsteht:

```
a + (a + 1)
```

Microsoft bevorzugt für die Regel 4 eine alternative Formulierung:[1]

> Unrelated to operator precedence and associativity, operands in an expression are evaluated from left to right.

Allerdings droht bei dieser Formulierung das Missverständnis, dass vorweg alle Operanden ausgewertet und danach alle Operationen ausgeführt würden. Das kann bei einem Ausdruck der Fall sein, gilt aber nicht allgemein.

4.5.10.2 Fallen

In diesem Abschnitt werden (hoffentlich ohne arroganten Auftritt) zwei Fehler beschrieben, die sich bei Missachtung der im vorigen Abschnitt beschriebenen Regeln durch eine beeindruckende Operator-Priorität oder einen überschätzten Klammerungseffekt ergeben können.

Wir betrachten einen Ausdruck mit der folgenden Struktur,

$$a + b * c$$

wobei *a*, *b* und *c* für beliebige numerische Operanden stehen (z. B. `++ivar`). Aus der Bindungskraftregel resultiert die folgende implizite Klammerung (Zuordnung der Operanden):

$$a + (b * c)$$

Aus der Links-vor-rechts - Regel ergibt sich für die Auswertung der Operanden bzw. Ausführung der Operatoren die folgende Reihenfolge:

$$a, b, c, *, +$$

Wenn als Operanden numerische Literale oder Variablen auftreten, dann wird bei der „Auswertung“ eines Operanden lediglich sein Wert ermittelt, und die Reihenfolge der Operandenauswertungen ist belanglos. Es bleibt ohne Folgen, wenn im Beispiel aufgrund der höheren Priorität des Multiplikationsoperators eine *falsche* Auswertungsreihenfolge unterstellt wird:

$$b, c, *, a, +$$

Wenn Operanden *Nebeneffekte* enthalten (z. B. Inkrementoperationen oder Methodenaufrufe), dann ist die Reihenfolge der Operandenauswertungen jedoch relevant, und eine falsche Vermutung kann gravierende Fehler verursachen. Im folgenden Beispiel

```
int ivar = 2;
Console.WriteLine(ivar++ + ivar * 2); // Ausgabe: 8
```

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/#operator-precedence

resultiert für den Ausdruck `ivar++ + ivar * 2` der Wert 8, denn:

- Zuerst wird der linke Operand der Addition ausgewertet:
  - Der Ausdruck `ivar++` hat den Wert 2.
  - Die Postinkrementoperation hat einen Nebeneffekt auf die Variable `ivar`.
- Dann wird der linke Operand der Multiplikation ausgewertet (Ergebnis: 3).
- Dann wird der rechte Operand der Multiplikation ausgewertet (Ergebnis: 2).
- Dann wird die Multiplikation ausgeführt (Ergebnis: 6).
- Dann wird die Addition ausgeführt (Ergebnis: 8).

Die gelegentlich anzutreffende Behauptung, Klammerausdrücke würden generell zuerst ausgewertet, ist falsch, wie das folgende Beispiel zeigt:[1]

Quellcode	Ausgabe

`int ivar = 2;`
`int erg = ivar * (++ivar + 5);`
`Console.WriteLine(erg);`
	`16`

Die Auswertung des Ausdrucks `ivar * (++ivar + 5)` verläuft so:

- Wegen Regel 4 aus dem Abschnitt 4.5.10.1 (Links-vor-rechts – Regel) wird zuerst 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.

4.5.10.3 Operatorentabelle

In der folgenden Tabelle sind die bisher behandelten Operatoren mit absteigender Bindungskraft (Priorität) aufgelistet. Gruppen von Operatoren mit gleicher Bindungskraft sind durch eine horizontale Linie voneinander getrennt. 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**)

[1] Von der IBM-Seite https://www.ibm.com/docs/en/zos/2.4.0?topic=operators-order-evaluation stammt die folgende Aussage:
Expressions within parentheses are evaluated first.

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*
+=, -=, *=, /=, %=	Verbundzuweisung	V_n, *N*

Im Anhang A finden Sie eine erweiterte Version dieser Tabelle, die zusätzlich alle Operatoren enthält, die im weiteren Verlauf des Manuskripts noch behandelt werden.

4.5.11 Übungsaufgaben zum Abschnitt 4.5

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 erhalten die Variablen `erg1` und `erg2` im folgenden Beispiel?

```
int i = 2, j = 3, erg1, erg2;
erg1 = (i++ == j ? 7 : 8) % 3;
erg2 = (++i == j ? 7 : 8) % 2;
Console.WriteLine($"erg1 = {erg1}\nerg2 = {erg2}");
```

3) Welche Wahrheitswerte erhalten im folgenden Beispiel die booleschen Variablen `la1` bis `la3`?

```
bool la1 = 3 > 2 && 2 == 2 ^ 1 == 1;
Console.WriteLine(la1);

bool la2 = ((2 > 3) && (2 == 2)) ^ (1 == 1);
Console.WriteLine(la2);

int i = 3;
char c = 'n';
bool la3 = !(i > 0 || c == 'j');
Console.WriteLine(la3);
```

Tipp: Die Negation von zusammengesetzten logischen Ausdrücken ist etwas unangenehm. Mit Hilfe der Regeln von **De Morgan** 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 BCL-Dokumentation zur Klasse **Math** im Namensraum **System** eine passende Methode.
- Verwenden Sie zum Einlesen des Arguments eine Variante der im Abschnitt 4.4 beschriebenen Technik, wobei die **Convert**-Methode **ToInt32()** geeignet zu ersetzen ist.

5) In den folgenden Ausdrücken stehen *a*, *b*, *c* und *d* für beliebige numerische Operanden:[1]

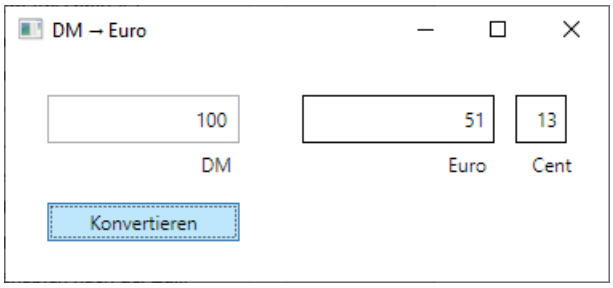
i) *a* / *b* + *c* * *d*
ii) *a* / (*b* + *c*) * *d*

In welcher Reihenfolge werden die Operanden jeweils ausgewertet bzw. die Operatoren ausgeführt?

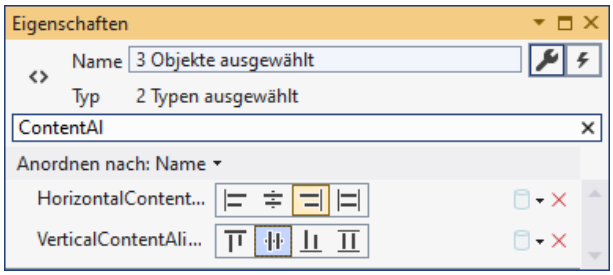
6) Entwickeln Sie den Währungskonverter aus den Abschnitten 3.3.7 und 4.3.11 so weiter, dass ganzzahlige, korrekt gerundete Werte für Euro und Cent erscheinen, z. B.:

---

[1] Die Beispiele stammen von:
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/#operator-precedence



Damit die Zahlen rechtsbündig und vertikal zentriert angezeigt werden, sollten die Eigenschaften **HorizontalContentAlignment** und **VerticalContentAlignment** angepasst werden, was für die drei betroffenen Steuerelemente gemeinsam geschehen kann:

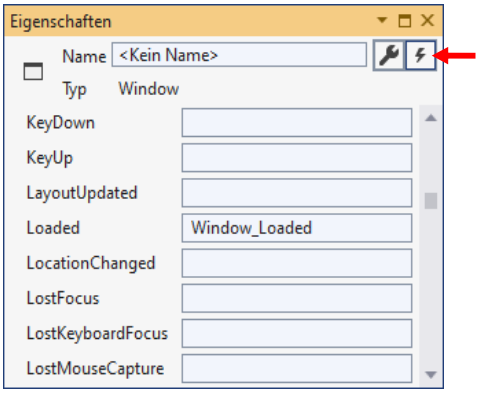


Es wäre nett, wenn nach dem Start unseres Programms das Textfeld für den DM-Betrag den Tastaturfokus hätte, sodass der Benutzer unmittelbar mit der Eingabe beginnen könnte, ohne zuvor den Tastaturfokus (z. B. per Maus) setzen zu müssen. Eine Lösungsmöglichkeit besteht darin, das zu privilegierende Steuerelement über die Methode **Focus()** der Klasse **UIElement** aufzufordern, den Fokus zu übernehmen, z. B.:

```
eingabe.Focus();
```

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 zu den Ereignissen:

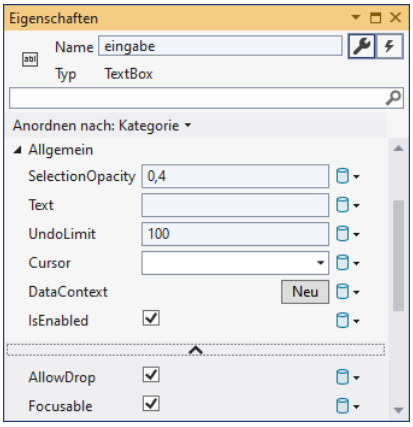


- Setzen Sie einen Doppelklick auf das Texteingabefeld zum Ereignis **Loaded**.
- Daraufhin wird in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `Window_Loaded()` zu unserer Fensterklasse `MainWindow` angelegt:

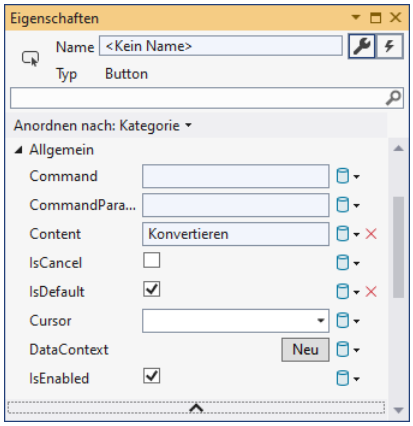
  ```
  private void Window_Loaded(object sender, RoutedEventArgs e) {

  }
  ```

  Diese Methode wird ausgeführt, wenn das Anwendungsfenster das Ereignis **Loaded** feuert.
- Ergänzen Sie im Rumpf dieser Methode den oben beschriebenen **Focus()** - Aufruf.

Außerdem muss die Eigenschaft **Focusable** des **TextBox**-Steuerelements den Wert **true** besitzen, was per Voreinstellung der Fall ist:



Wir haben (über den Wert **true** für die Eigenschaft **IsDefault**) bereits dafür gesorgt, dass sich das Klick-Ereignis des **Button**-Objekts per **Enter**-Taste auslösen lässt:



4.6 Über- und Unterlauf bei numerischen Datentypen

Wie Sie inzwischen wissen, haben die numerischen Datentypen jeweils einen bestimmten Wertebereich (siehe Tabelle im Abschnitt 4.3.4). 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, sodass Wertebereichsprobleme unbedingt vermieden oder rechtzeitig diagnostiziert werden müssen.

4.6.1 Überlauf bei Ganzzahltypen

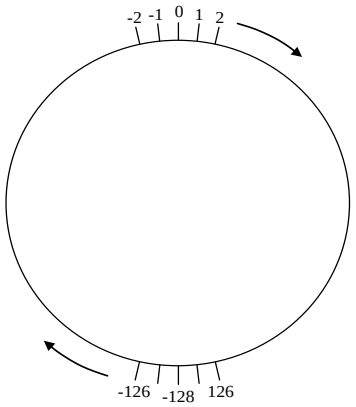
Wird z. B. zu einer ganzzahligen Variablen, die bereits den maximalen Wert ihres Typs besitzt, eine positive Zahl addiert, dann kann das Ergebnis nicht mehr korrekt abgespeichert werden. Ohne besondere Vorkehrungen stellt ein C# - Programm im Fall eines solchen Ganzzahlüberlaufs keinesfalls seine Tätigkeit ein (z. B. mit einem Ausnahmefehler), sondern arbeitet munter weiter. Der folgende Quellcode (auf oberster Ebene)

```
int i = 2_147_483_647, j = 5, k;
k = i + j;   // Überlauf!
Console.WriteLine($"{i} + {j} = {k}");
```

liefert ohne Warnung das sinnlose 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 4.5.7) 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 für eine erfolgreiche Raketenentwicklung die sichere Beherrschung der elementaren Sprachelemente erforderlich ist.

Natürlich kann nicht nur der positive Rand eines Ganzzahlwertebereichs überschritten werden, sondern auch der negative Rand, indem z. B. vom kleinstmöglichen Wert eine positive Zahl subtrahiert wird:

Quellcode	Ausgabe

`int i = -2147483648, j = 5, k;`
`k = i - j;`
`Console.WriteLine($"{i} - {j} = {k}");`
	`-2147483648 - 5 = 2147483643`

Bei Wertebereichsproblemen durch eine betragsmäßig zu große Zahl wird im Manuskript generell von einem *Überlauf* gesprochen. Unter einem *Unterlauf* soll später das Verlassen eines Gleitkommawertebereichs in Richtung null durch eine betragsmäßig zu kleine Zahl verstanden werden (vgl. Abschnitt 4.6.4).

Oft lässt sich ein Überlauf durch die Wahl eines **geeigneten Datentyps** verhindern. Mit den Deklarationen

```
long i = 2_147_483_647, j = 5, k;
```

kommt es in der Anweisung

```
k = i + j;
```

nicht zum Überlauf, weil neben den Variablen `i`, `j` und `k` nun auch der Ausdruck `i + j` den Typ **long** besitzt. Die Anweisung

```
Console.WriteLine($"{i} + {j} = {k}");
```

liefert das korrekte Ergebnis:

```
2147483647 + 5 = 2147483652
```

Im Beispiel genügt es *nicht*, für die Zielvariable `k` den Typ **int** durch **long** zu ersetzen, weil der Überlauf beim Berechnen des Ausdrucks („unterwegs“) auftritt. Mit den Deklarationen

---

[1] Mr. Spock arbeitete jahrelang als erster Offizier auf dem Raumschiff Enterprise.

```
int i = 2_147_483_647, j = 5;
long k;
```

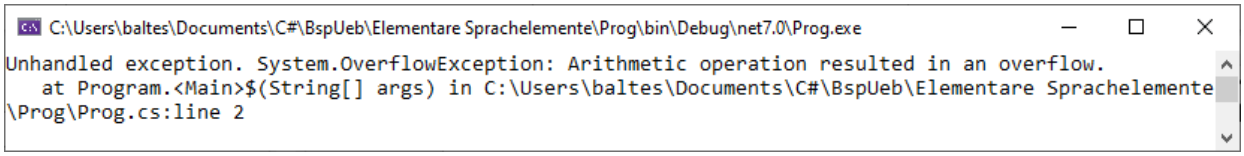
bleibt das Ergebnis falsch, denn …

- In der Anweisung
  ```
  k = i + j;
  ```
  wird der Ausdruck `i + j` berechnet, bevor die Zuweisung ausgeführt wird.
- 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 (z. B. Java) mit dem **checked**-Operator eine Möglichkeit bereit, den Überlauf bei Ganzzahlvariablen abzufangen. Eine gesicherte Variante des ursprünglichen Beispiels

```
int i = 2_147_483_647, j = 5, k;
k = checked(i + j);
Console.WriteLine($"{i} + {j} = {k}");
```

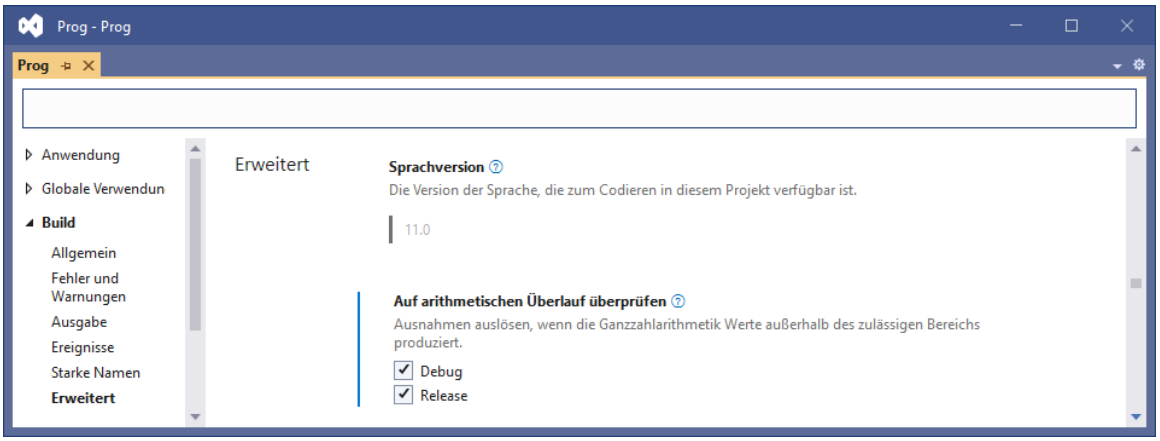
rechnet nach einem Überlauf nicht mit „Zufallszahlen“ weiter, sondern bricht mit einem Ausnahmefehler ab:[1]



Anstelle des **checked**-Operators bietet C# noch weitere Möglichkeiten, die Überlaufdiagnose für Ganzzahltypen einzuschalten:

- **checked**-Anweisung
  Man kann die Überwachung für einen kompletten Anweisungsblock einschalten, z. B.:
  ```
  checked {
      int i = 2_147_483_647, j = 5, k;
      k = i + j;
      Console.WriteLine($"{i} + {j} = {k}");
  }
  ```
- **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. Im Visual Studio vereinbart man diese Compiler-Option für ein Projekt folgendermaßen:
  - Menübefehl **Projekt > Eigenschaften**
  - Registerkarte **Build**
  - Schalter **Erweitert**
  - Kontrollkästchen **Auf arithmetischen Überlauf überprüfen**

[1] Im Kapitel 13 über die Ausnahmebehandlung werden Sie lernen, Ausnahmefehler abzufangen, damit diese nicht mehr zum Abbruch des Programms führen.



Wird die Kontrolle für die Debug- und die Release-Konfiguration eingeschaltet, dann entstehen in der Projektdatei zwei **PropertyGroup**-Elemente mit Bedingung:

```
<Project Sdk="Microsoft.NET.Sdk">

  . . .

  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'">
    <CheckForOverflowUnderflow>True</CheckForOverflowUnderflow>
  </PropertyGroup>

  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|AnyCPU'">
    <CheckForOverflowUnderflow>True</CheckForOverflowUnderflow>
  </PropertyGroup>

</Project>
```

Als Argument gegen die naheliegende Entscheidung, die Überlaufdiagnose für Ganzzahltypen *generell* einzuschalten, kommt nur der zeitliche Mehraufwand in Betracht. Bei der Aufgabe, mit 0 beginnend 2147483647 mal den **int**-Wert 1 zu addieren,

$$\sum_{i=1}^{2147483647} 1$$

ergaben sich in der *Release*-Konfiguration des Projekts basierend auf jeweils 50 Tests mit schwankenden Einzelergebnissen die folgenden mittleren Laufzeiten in Millisekunden:[1]

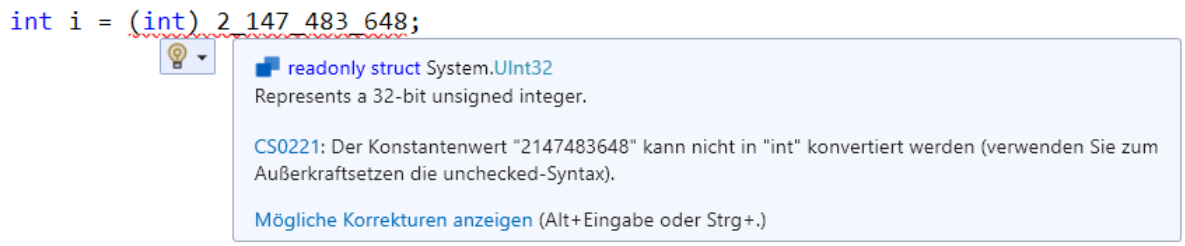
Statistiken

		Checked	Unchecked
N	Gültig	50	50
	Fehlend	0	0
Mittelwert		2206,0800	1470,6200
Median		2181,5000	1464,0000

Der zeitliche Mehraufwand aufgrund der Überlaufdiagnose beträgt immerhin ca. 50%, sodass man *nicht* empfehlen kann, die Überlaufkontrolle für Ganzzahloperationen generell einzuschalten.

Der Vollständigkeit halber soll noch der **unchecked**-Operator erwähnt werden, mit dem sich eine per Compiler-Option aktivierte Überlaufdiagnose und andere Kontrollfunktionen des Compilers abschalten lassen. Auch ohne **checked**-Instruktion verhindert der Compiler z. B., dass ein Ganzzahlliteral mit Typ **long** in den Typ **int** gewandelt wird. Das Visual Studio verrät, wie man diese Sicherung deaktivieren kann:

[1] Berechnet mit IBM SPSS Statistics 27



Wird der „Tipp“ umgesetzt, dann resultiert ein fehlerhafter Wert:

Quellcode	Ausgabe
`int i = unchecked((int) 2_147_483_648);` `Console.WriteLine(i);`	`-2147483648`

Wenn der **(u)long**-Wertebereich für eine Aufgabenstellung nicht reicht, dann muss man sich trotzdem nicht auf die Problemerkennung per Ausnahmefehlermeldung beschränken, sondern hat mit der Struktur **BigInteger** aus dem Namensraum **System.Numerics** noch einen Datentyp für *beliebig große* ganze Zahlen zur Verfügung (zu Strukturen siehe Abschnitt 6.1). Dass eine Variable vom Typ **BigInteger** erheblich mehr Speicher- und Rechenzeitaufwand verursacht als eine **int**-Variable, fällt nur beim massenhaften Auftreten von **BigInteger**-Variablen ins Gewicht. Wie das folgende Beispiel zeigt, erzielt man mit dem Datentyp **BigInteger** noch sinnvolle Ergebnisse, wenn es bei den elementaren Ganzzahltypen zum Überlauf kommt:

Quellcode	Ausgabe

`using System.Numerics;`
 	
`ulong i = 18_446_744_073_709_551_615, j = 5, k;`
`k = i + j;`
`Console.WriteLine("Rechnung mit ulong:");`
`Console.WriteLine($"{i} + {j} = {k}");`
 	
`BigInteger bigInt = i;`
`bigInt += j;`
`Console.WriteLine("\nRechnung mit BigInteger:");`
`Console.WriteLine($"{i} + {j} =\n{bigInt}");`
	`Rechnung mit ulong:`
	`18446744073709551615 + 5 = 4`
	
	`Rechnung mit BigInteger:`
	`18446744073709551615 + 5 =`
	`18446744073709551620`

4.6.2 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 erheblich 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 Beispiel:

```
double bigd = Double.MaxValue;
Console.WriteLine("Double.MaxValue =\t" + bigd);
bigd *= 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 die auf den ersten Blick verblüffende Ausgabe:

```
Double.MaxValue =       1,79769313486232E+308
Double.MaxValue * 10 =  8
Unendl. + 10 =          8
Unendl. * -13 =         -8
13.0/0.0 =              8
```

Offenbar hatte jemand die „kreative“ Idee, das in der Mathematik übliche Symbol ∞ für Unendlich durch die Ziffer 8 darzustellen.[1]

Mit Hilfe der Unendlich-Werte „gelingt“ bei der Gleitkommaarithmetik sogar die Division durch null, während bei der Ganzzahlarithmetik ein solcher Versuch zu einem Laufzeitfehler führt.[2]

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:[3]

```
double 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 die Ausgabe:

```
Unendlich - Unendlich = NaN
Unendlich / Unendlich = NaN
Unendlich * 0.0 =       NaN
0.0 / 0.0 =             NaN
```

Zu den letzten Beispielprogrammen ist noch anzumerken, dass man über das öffentliche Feld **MaxValue** der Struktur[4] **Double** aus dem Namensraum **System** den größten Wert in Erfahrung bringt, der in einer **double**-Variablen gespeichert werden kann.[5]

---

[1] Bis zum .NET Framework 3.5 erschien eine weniger kreative und weniger missverständliche Ausgabe:

```
Double.MaxValue =       1,79769313486232E+308
Double.MaxValue * 10 =  +unendlich
Unendl. + 10 =          +unendlich
Unendl. * -13 =         -unendlich
13.0/0.0 =              +unendlich
```

[2] Es wird dann ein Objekt der Klasse **DivideByZeroException** „geworfen“. Mit der Ausnahmebehandlung werden wir uns im Kapitel 13 beschäftigen.

[3] Die Bezeichnung *Programm* ist gerechtfertigt, weil die fehlenden, aus uninformativen Standardzeilen bestehenden Bestandteile eines Konsolenprogramms seit C# 9/10 weggelassen werden dürfen (siehe Abschnitt 4.1.2).

[4] 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**).

[5] Die öffentliche **MaxValue**-Verfügbarkeit stellt übrigens *keinen* Verstoß gegen das Datenkapselungs-Prinzip der objektorientierten Programmierung dar, weil das Feld als *konstant* deklariert ist (vgl. Abschnitt 4.3.9):

```
public const double MaxValue = 1.7976931348623157E+308;
```

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 Ausdruck vom Typ **double** prüfen, ob er einen unendlichen oder undefinierten Wert besitzt, z. B.:

Quellcode	Ausgabe
`double d = 0.0 / 0.0;` `Console.WriteLine(Double.IsNaN(d));`	`True`

Eben wurde eine Technik zur Beschreibung von *vorhandenen Bibliotheksmethoden* im Manuskript erstmals benutzt, die auch in der BCL-Dokumentation in ähnlicher Form Verwendung findet, z. B.:

C# Kopieren

```
public static bool IsPositiveInfinity (double d);
```

Dabei wird die Benutzung einer Methode erläutert durch Angabe von:

- 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 4.3.5.1):

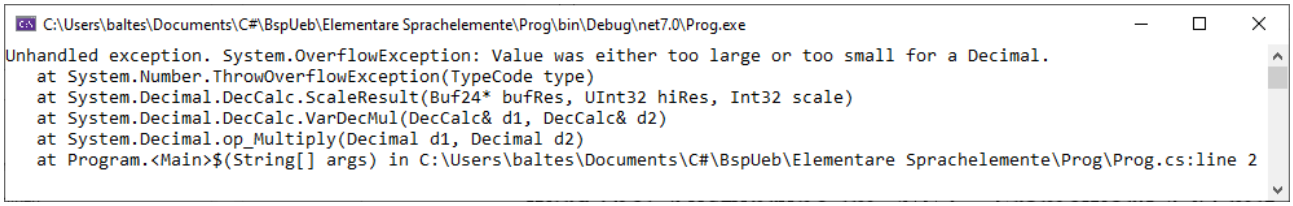
Wert	**float-Darstellung**		
	Vorz.	**Exponent**	**Mantisse**
+unendlich	0	11111111	00000000000000000000000
-unendlich	1	11111111	00000000000000000000000
NaN	0	11111111	10000000000000000000000

4.6.3 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 4.6.1) muss die Überlaufdiagnose *nicht* per **checked**-Operator, -Anweisung oder -Compiler-Option angeordnet werden. Das Programm

```
decimal d = Decimal.MaxValue;
d *= 10;
Console.WriteLine(d);
```

wird mit der folgenden Meldung abgebrochen:



4.6.4 Unterlauf bei den Gleitkommatypen

Bei den binären Gleitkommatypen **float**, **double** und **decimal** ist auch ein Unterlauf möglich, wobei eine Zahl mit einem sehr kleinen Betrag (bei **double**: $< 4{,}94065645841247 \cdot 10^{-324}$) nicht mehr dargestellt werden kann. In diesem Fall rechnet ein C# - Programm mit dem Wert 0,0 weiter, was in der Regel akzeptabel ist, z. B.:

Quellcode	Ausgabe

`double smalld = Double.Epsilon;`
`Console.WriteLine(smalld);`
`smalld /= 10.0;`
`Console.WriteLine(smalld);`
	`5E-324`
	`0`

Das öffentliche Feld **Double.Epsilon** enthält den kleinsten Betrag, der in einer **double**-Variablen gespeichert werden kann (vgl. Abschnitt 4.3.5.1 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, dann führt ein irreversibler Unterlauf zum falschen Ergebnis 0:

Quellcode	Ausgabe

`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`

Im folgenden Beispielprogramm führt der Versuch, den kleinsten positiven **decimal**-Wert (10^{-28}) zu halbieren, zum Ergebnis 0:

Quellcode	Ausgabe

`decimal smallDec = 1e-28m;`
`Console.WriteLine(smallDec);`
`smallDec /= 2m;`
`Console.WriteLine(smallDec);`
	`0,0000000000000000000000000001`
	`0`

4.7 Anweisungen (zur Ablaufsteuerung)

Wir haben uns im Kapitel 4 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.

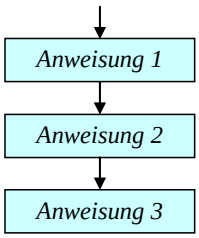
In den meisten Beispielprogrammen traten nur wenige Sorten von Anweisungen auf (Variablendeklarationen, Wertzuweisungen und Methodenaufrufe). Nun werden wir uns systematisch mit dem allgemeinen Begriff einer C# - Anweisung befassen und vor allem die wichtigen Anweisungen zur Ablaufsteuerung (Verzweigungen und Schleifen) kennenlernen.

4.7.1 Überblick

Ausführbare Programmteile, die in C# nach unserem bisherigen Kenntnisstand als Methoden oder Eigenschaften von Klassen zu realisieren sind, bestehen aus *Anweisungen* (engl. *statements*).

Am Ende des Abschnitts 4.7 werden Sie die folgenden Sorten von Anweisungen kennen:

- **Deklarationsanweisung für lokale Variablen**
  Die Deklarationsanweisung für lokale Variablen wurde schon im Abschnitt 4.3.6 eingeführt.
  Beispiel: `int i = 1, k;`
- **Ausdrucksanweisungen**
  Die folgenden Ausdrücke werden zu Anweisungen, sobald man ein Semikolon dahinter setzt:
  - **Wertzuweisung** (vgl. Abschnitte 4.3.6 und 4.5.8)
    Beispiel: `k = i + j;`
  - **Prä- bzw. Postinkrement- oder -dekrementoperation**
    Beispiel: `i++;`
    Im Beispiel ist nur der „Nebeneffekt“ des Ausdrucks `i++` von Bedeutung (vgl. Abschnitt 4.5.1). Sein Wert bleibt ungenutzt.
  - **Methodenaufruf**
    Beispiel: `Console.WriteLine(cond);`
    Besitzt die im Rahmen einer eigenständigen Anweisung aufgerufene Methode einen Rückgabewert, dann 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 syntaktisch eine Anweisung erforderlich ist, 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 im Abschnitt 4.3.8 im Zusammenhang mit dem Sichtbarkeitsbereich 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 **Main()** - Methoden der bisherigen Beispielprogramme im Kapitel 4 bestanden meist aus einer *Sequenz* von Anweisungen, die bei jedem Programmeinsatz komplett durchlaufen wurde:[1]



[1] In der Regel nutzen wir in kurzen Beispielprogrammen die mit C# 9 eingeführten Anweisungen der obersten Ebene, sodass im Quellcode lediglich die *Anweisungen* der **Main()** – Methode zu sehen sind, während der Definitionskopf vom Compiler beigesteuert wird (vgl. Abschnitt 4.1.2).

Oft möchte man jedoch ...

- die Ausführung einer Anweisung (bzw. eines Anweisungsblocks) von einer *Bedingung* abhängig machen
- oder eine Anweisung (bzw. einen Anweisungsblock) *wiederholt* ausführen lassen.

Für solche Zwecke enthält C# etliche Anweisungen zur Ablaufsteuerung, 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.

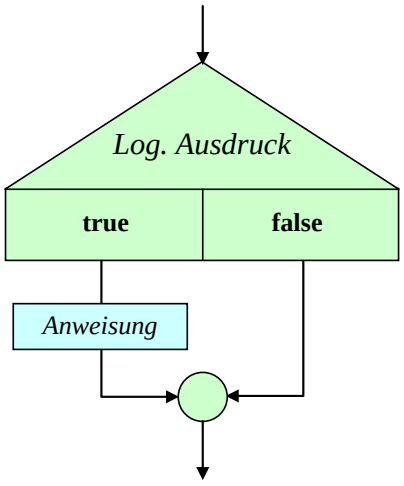
Anweisungen werden durch ein **Semikolon** abgeschlossen, sofern sie nicht mit einer schließenden Blockklammer enden.

4.7.2 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 müssen.

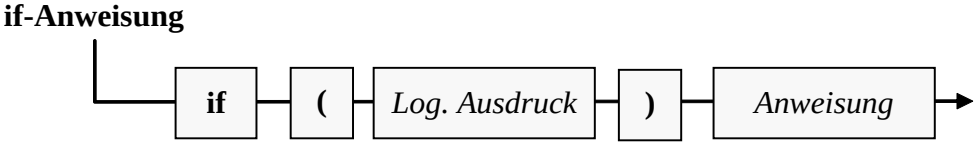
4.7.2.1 if-Anweisung

Nach dem folgenden **Programmablaufplan** (**PAP**) bzw. **Flussdiagramm** soll eine 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, obwohl sie keiner Norm entsprechen.

Während der Programmablaufplan den Zweck (die Semantik) eines Sprachbestandteils erläutert, beschreibt das vertraute Syntaxdiagramm, wie zulässige Exemplare des Sprachbestandteils zu bilden sind. Das folgende Syntaxdiagramm beschreibt die zur Realisation einer bedingten Ausführung dienende **if**-Anweisung:



Die eingebettete (bedingt auszuführende) Anweisung darf keine Variablendeklaration (im Sinn von Abschnitt 4.3.6) sein. Ein Block ist aber selbstverständlich erlaubt, und darin dürfen auch lokale

Variablen deklariert werden. Weil die beiden letzten Sätze leicht widersprüchlich klingen, sind erläuternde Beispiele angemessen:

Verboten

```
int i = 2, j = 3;
if (i == j)
    int k = 1;
```

Erlaubt

```
int i = 2, j = 3;
if (i == j) {
    int k = 1;
}
```

Es ist übrigens nicht vergessen worden, ein Semikolon ans Ende des **if**-Syntaxdiagramms zu setzen. Dort wird eine eingebettete Anweisung verlangt, wobei konkrete Beispiele oft mit einem Semikolon enden, manchmal aber auch mit einer schließenden geschweiften Klammer.

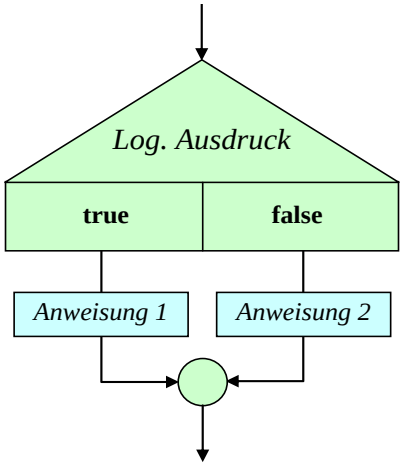
Im folgenden Beispiel wird eine Meldung ausgegeben, wenn die Variable `anz` einen Wert kleiner oder gleich null besitzt:

```
if (anz <= 0)
  Console.WriteLine("Die Anzahl muss > 0 sein!");
```

Der Zeilenumbruch zwischen dem logischen Ausdruck und der eingebetteten Anweisung dient nur der Übersichtlichkeit und ist für den Compiler irrelevant.

4.7.2.2 if-else - Anweisung

Soll auch etwas passieren, wenn der steuernde logische Ausdruck den Wert **false** besitzt,



dann 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 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 einfache Bildungsregeln (mit einer einzigen zulässigen Sequenz) sehr anschaulich beschreibt, kann das manchmal weniger anschauliche Syntaxdiagramm bei einer komplizierten und variantenreichen Syntax alle zulässigen Sequenzen kompakt und präzise dokumentieren.

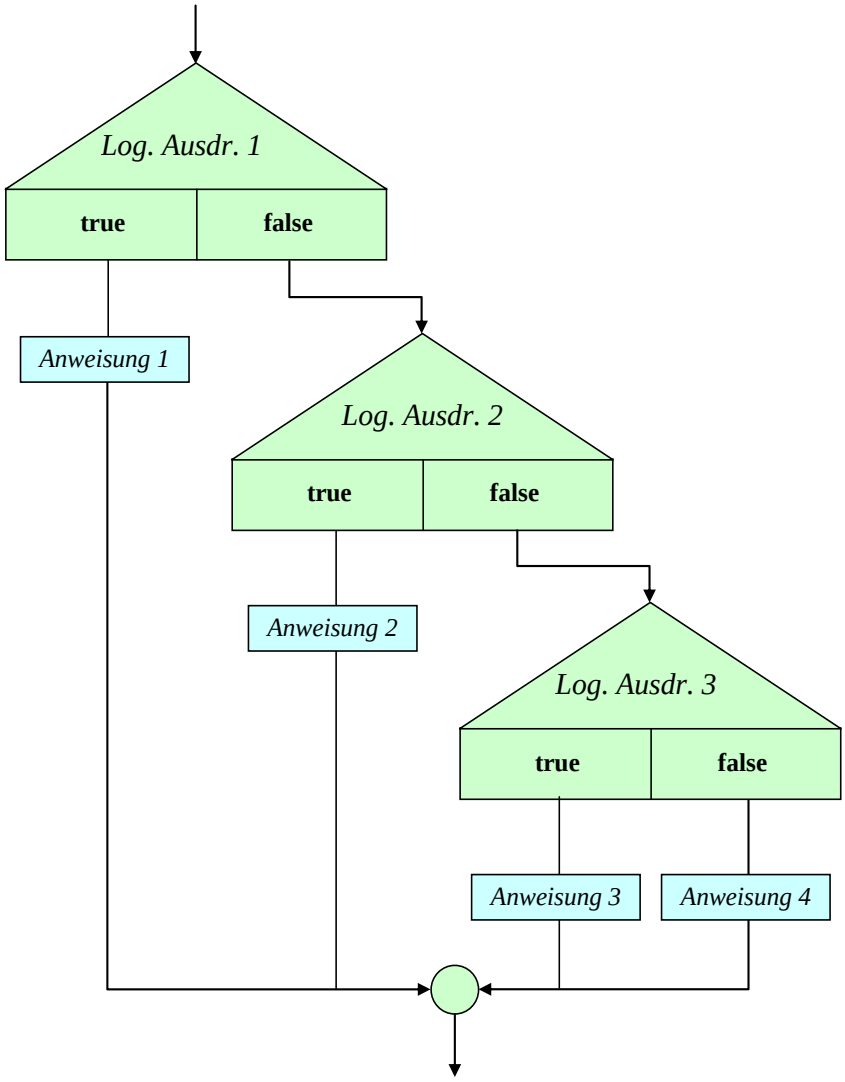
Bei den eingebetteten Anweisungen (*Anweisung 1* bzw. *Anweisung 2*) darf es sich nicht um Variablendeklarationen (im Sinn von Abschnitt 4.3.6) handeln. Wird ein *Block* als eingebettete Anweisung verwendet, sind darin aber auch Variablendeklarationen erlaubt (siehe Erläuterung zur **if**-Anweisung im Abschnitt 4.7.2.1).

Im folgenden **if-else** - Beispiel wird der natürliche Logarithmus zu einer Zahl geliefert, falls diese positiv ist. Anderenfalls erscheint eine Fehlermeldung. Das Argument wird vom Benutzer über eine **ToDouble() - ReadLine()** - Konstruktion erfragt (vgl. Abschnitt 4.4.1).

Quellcode	Ausgabe (Eingaben **fett**)

`Console.Write("Argument: ");`
`double arg = Convert.ToDouble(Console.ReadLine());`
`if (arg > 0)`
`  Console.WriteLine($"ln({arg}) = {Math.Log(arg)}");`
`else`
`  Console.WriteLine("Argument <= 0");`
	`Argument: 2`
	`ln(2) = 0,693147180559945`

Eine bedingt auszuführende Anweisung darf wiederum vom **if**- bzw. **if-else** - Typ sein, sodass 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

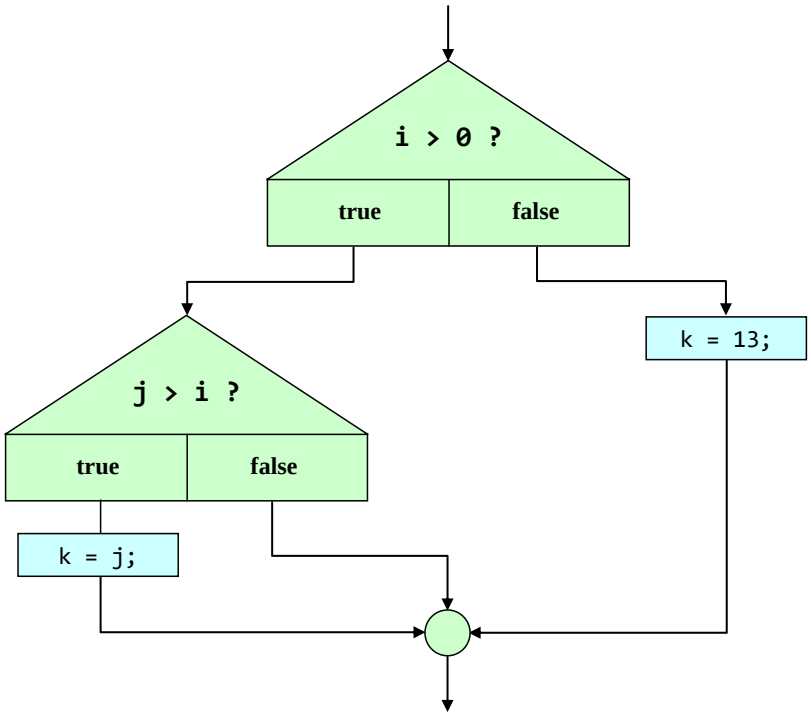
Wenn alle logischen Ausdrücke den Wert **false** annehmen, dann wird die **else**-Klausel zur letzten **if**-Anweisung ausgeführt.

Bei einer Mehrfallunterscheidung ist die im Abschnitt 4.7.2.3 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 sogenannten **dangling-else** - Problem[1] kommen, wobei ein Missverständnis zwischen Compiler und Programmierer hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Beispiel[2]

```
if (i > 0)
    if (j > i)
        k = j;
else
    k = 13;
```

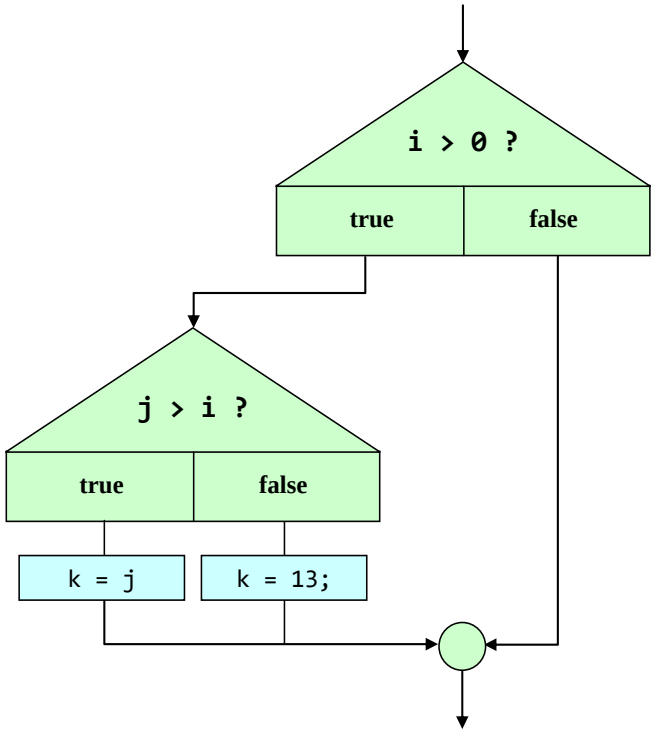
lassen die Einrücktiefen vermuten, dass der Programmierer die **else**-Klausel auf die *erste* **if**-Anweisung bezogen zu haben glaubt:



[1] Deutsche Übersetzung von *dangling*: *baumelnd*.

[2] Fügt man den Quellcode mit den „fehlerhaften“ Einrücktiefen in ein Editorfenster unserer Entwicklungsumgebung ein, dann wird der „Layout-Fehler“ übrigens automatisch behoben. Das Visual Studio verhindert also, dass der Logikfehler durch einen „Layout-Fehler“ getarnt wird.

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, sodass de facto der folgende Programmablauf resultiert:



Bei `i` ≤ 0 geht der Programmierer vom neuen `k`-Wert 13 aus, der beim tatsächlichen Programmablauf jedoch *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 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;
```

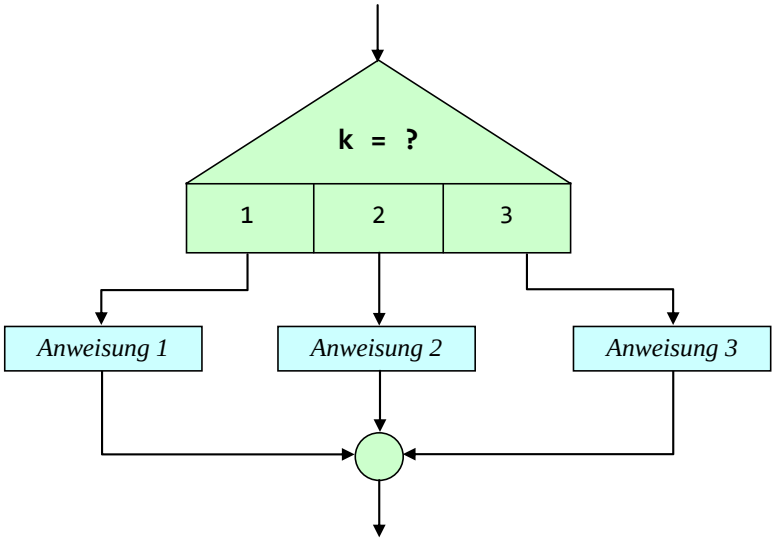
Gelegentlich kommt als Alternative zu einer simplen **if-else** - Anweisung, die zur Berechnung eines Wertes bedingungsabhängig zwei unterschiedliche Ausdrücke benutzt, der Konditionaloperator (vgl. Abschnitt 4.5.9) in Frage, z. B.:

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;`

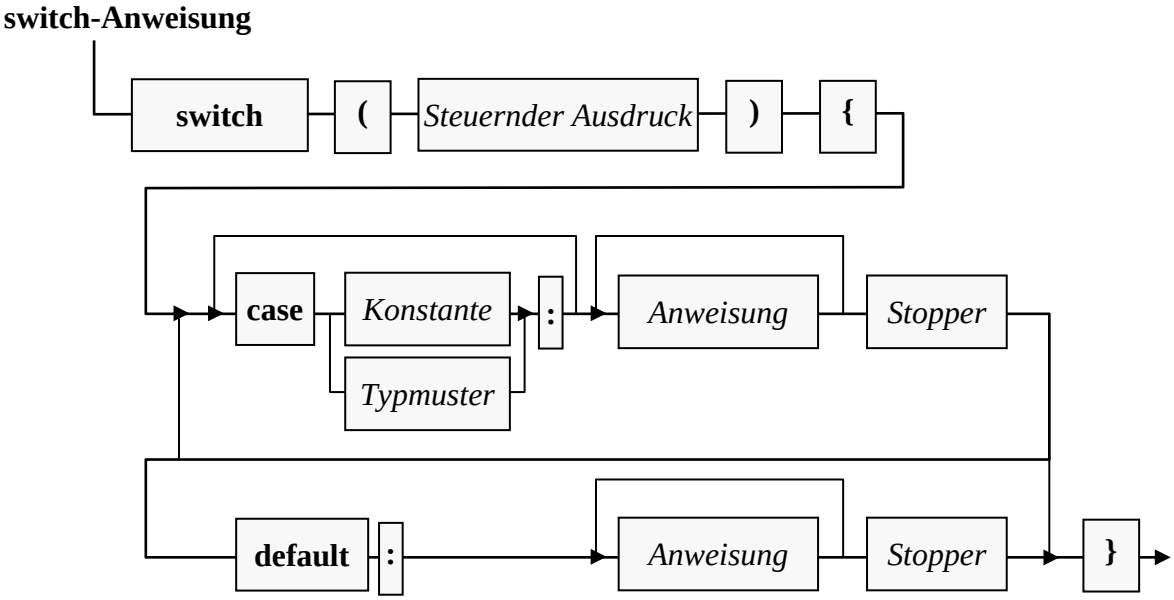
4.7.2.3 switch-Anweisung

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.

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.



Als Datentyp des steuernden Ausdrucks sind erlaubt:

- In C# 6.0:
  - Ganzzahlige (integrale) numerische Datentypen (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**). Dazu gehört auch der Datentyp **char** (vgl. Abschnitt 4.3.4).
  - **bool**
  - Aufzählungstypen (siehe Abschnitt 6.4)
  - Zeichenfolgen (Datentyp **string**)
- Seit C# 7.0 ist ein beliebiger Ausdruck erlaubt.

Mit C# 7.0 hat sich an der **switch**-Anweisung mehr geändert als die Liste der erlaubten Datentypen für den steuernden Ausdruck. Bei der **case**-Deklaration sind die traditionellen Konstanten durch die erheblich flexibleren Typmuster ergänzt worden. Außerdem ist seit C# 8 neben der **switch**-*Anweisung* auch der **switch**-*Ausdruck* verfügbar.

Wir beschreiben die Vielfalt der **switch**-Optionen dosiert und starten mit der traditionellen, bis C# 6.0 alternativlosen Variante.

4.7.2.3.1 Klassische switch-Anweisung

Auf dem Sprachniveau von C# 6.0 sind in den **case**-Deklarationen einer **switch**-Anweisung nur *konstante* Ausdrücke erlaubt, deren Ergebnis schon der Compiler ermitteln kann (z. B. Literale, Konstanten oder mit konstanten Argumenten gebildete Ausdrücke).

Stimmt zur Laufzeit der Wert des steuernden Ausdrucks mit einem **case**-Wert überein, dann werden die zugehörigen Anweisungen ausgeführt, ansonsten (falls vorhanden) die **default**-Anweisungen.

Sollen für mehrere Werte des steuernden Ausdrucks dieselben Anweisungen ausgeführt werden, dann setzt man die zugehörigen **case**-Deklarationen hintereinander und lässt die Anweisungen auf die letzte Deklaration folgen, z. B.:

```
  . . .
case 3:
case 4:
  Console.WriteLine("Fälle 3 und 4");
  break;
. . .
```

Wer in einem **case** eine *Serie* von Fällen durch Angabe der Randwerte (z. B. von *a* bis *z*) behandeln möchte, wird sich über die ab C# 7.0 verfügbaren **switch**-Erweiterungen freuen (siehe Abschnitt 4.7.2.3.3).

Jeder Fall *muss* mit einem **Stopper** abgeschlossen werden, auch der **default** - Fall:

```
    default:
        Console.WriteLine("Restkategorie");
}
```

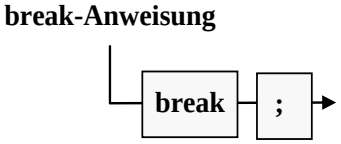
CS8070: Die Steuerung kann nicht von der abschließenden case-Bezeichnung ("default:") aus dem switch-Ausdruck übergeben werden.

Der aus anderen Programmiersprachen (z. B. C, C++, Java) bekannte „Durchfall“ zu den Anweisungen „tieferer“ Fälle ist verboten, womit viele Programmierfehler vermieden werden.

Als Stopper sind erlaubt:

- **break**-Anweisung
- **goto**-Anweisung
- **return**-Anweisung

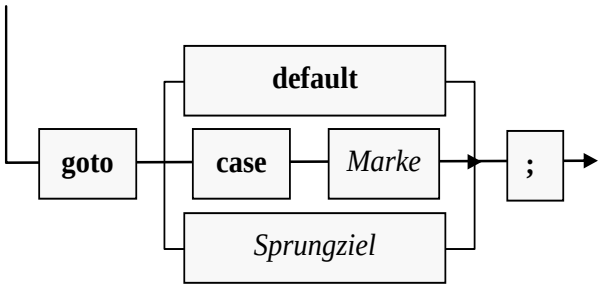
Meist stoppt man mit der **break**-Anweisung,



wobei die Methode hinter der **switch**-Anweisung fortgesetzt wird.

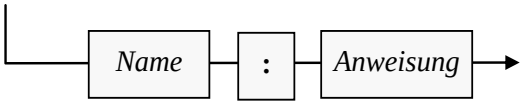
Per **goto**-Anweisung

goto-Anweisung als Stopper zu einem switch-case



kann eine **case**-Deklaration innerhalb der **switch**-Anweisung (inklusive **default**) oder ein Sprungziel an anderer Stelle innerhalb der Methode angesteuert werden:

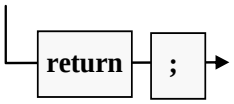
Sprungziel



Das gute (böse) alte **goto**, als Inbegriff rückständiger Programmierung („Spaghetti-Code“) aus vielen modernen Programmiersprachen verbannt, ist also in C# erlaubt. Damit lässt sich z. B. der aus anderen Programmiersprachen bekannte **switch**-Durchfall in C# trotz der Stopper-Vorschrift realisieren.

Mit der (im Abschnitt 5.3.1.2 ausführlich zu behandelnden) **return**-Anweisung wird die gesamte Methode beendet. Bei einer Methode mit dem Rückgabetyp **void** ist die folgende Syntax zu verwenden:

return-Anweisung ohne Rückgabe



Weil im nächsten Abschnitt ein praxisnahes (und damit auch etwas kompliziertes) Beispiel folgt, ist hier ein ebenso einfaches wie sinnfreies Beispiel zur Erläuterung der Syntax angemessen:

```
int zahl = 2;
switch (zahl) {
  case 1:
    Console.WriteLine("Fall 1, mit break-Stopper");
    break;
  case 2:
    Console.WriteLine("Fall 2, mit goto case - Stopper (Durchfall)");
    goto case 3;
  case 3:
  case 4:
    Console.WriteLine("Fälle 3 und 4, mit goto Sprungziel - Stopper");
    goto Fertig;
  default:
    Console.WriteLine("Restkategorie, mit return-Stopper");
    return;
}
Console.WriteLine("Nach break");
Fertig:
Console.WriteLine("Hinter Fertig");
```

Es produziert die folgende Ausgabe:

```
Fall 2, mit goto case - Stopper (Durchfall)
Fälle 3, 4, mit goto Sprungziel - Stopper
Hinter Fertig
```

4.7.2.3.2 Praxisnahes switch-Beispiel und Erstkontakt mit Befehlszeilenargumenten

Im folgenden Beispielprogramm wird die Persönlichkeit des Benutzers mit Hilfe seiner Farb- und Zahlpräferenz analysiert. Während bei einer Vorliebe für Rot oder Schwarz die Diagnose sofort feststeht, wird bei den restlichen Farben auch die Lieblingszahl berücksichtigt:[1]

```
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 (durch Leerzeichen getrennte) 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 ein Befehlszeilenargument auszuwerten und das Ergebnis in eine **String**- bzw. **int**-Variable zu befördern. Solche Anweisungen werden Sie mit Leichtigkeit selbst formulieren, sobald die beteiligten Begriffe behandelt worden sind. An dieser Stelle greifen wir späteren Erläuterungen mal wieder etwas vor (hoffentlich mit motivierendem Effekt):

---

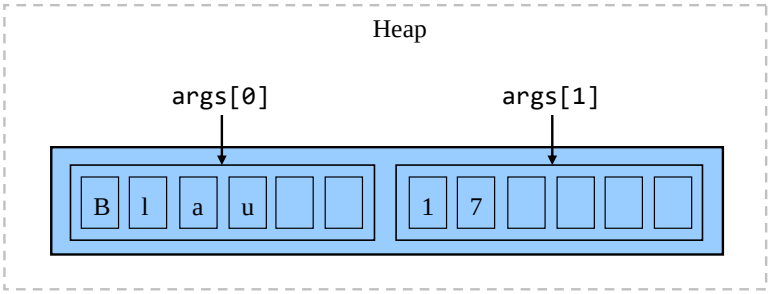
[1] Wie im Abschnitt 4.1.2 vereinbart, erscheinen die in der Datei **BspUeb.zip** enthaltenen Beispielprogramme (siehe Vorwort) im Manuskript *inklusive* Klassendefinitionskopf, damit der Klassenname zu sehen ist, und das Auffinden der gleichnamigen Quellcodedatei erleichtert wird. Der **Main()** – Definitionskopf ist dann ebenfalls erforderlich, und aus dem Verzicht auf die mit C# 10 eingeführten *impliziten* **using**-Direktiven resultiert das traditionelle Erscheinungsbild eines C# - Konsolenprogramms.

- 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, wobei es sich um Zeichenfolgen handelt. Literale mit diesem Datentyp sind uns schon öfter begegnet (z. B. `"Hallo"`).
- Über ihre Parameterliste kann man eine Methode mit Daten versorgen und/oder ihre Arbeitsweise beeinflussen.
- Besitzt die **Main()** - Methode einer Startklasse einen Parameter vom Datentyp **String[]** (Array mit **String**-Elementen), dann übergibt die CLR dieser Methode als Elemente des **String**-Arrays 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 ist jedoch frei wählbar (im Beispiel: `args`). Damit die Methode **Main()** zum Starten des Programms taugt, darf kein weiterer Parameter vorhanden sein. In der Methode **Main()** kann man auf das Array `args` genauso zugreifen wie auf eine lokale Variable.
- In einem Konsolenprogramm mit Anweisungen auf oberster Ebene (also ohne Klassen- und **Main()** – Definitionskopf im Quellcode, siehe Abschnitt 4.1.2) besitzt die vom Compiler erstellte **Main()** – Methode einen Parameter vom Datentyp **String[]** mit dem Namen **args**.
- 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]` im Programm 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()** - Rückgabe 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 lässt sich aus dieser Zeichenfolge mit der Klassenmethode **ToInt32()** der Klasse **Convert** eine Zahl vom Datentyp **int** gewinnen und anschließend der lokalen Variablen `zahl` zuweisen.

Nach einem Programmstart mit

```
perst Blau 17
```

kann man sich den **String**-Array `args`, der als Objekt im Heap-Bereich des programmeigenen Speichers abgelegt wird, ungefähr so vorstellen:[1]



[1] Hier wird aus didaktischen Gründen gemogelt: Die beiden Zeichenfolgen sind selbst Objekte und liegen „neben“ dem Array-Objekt auf dem Heap. Die Array-Elemente sind Referenzen, die auf die zugehörigen **String**-Objekte zeigen.

Ansonsten ist im Beispielprogramm noch die **return**-Anweisung von Interesse, welche die **Main()** -Methode und damit das Programm sofort beendet, wenn beim Programmstart weniger als zwei Befehlszeilenargumente angegeben wurden:

```
if (args.Length < 2) {
   Console.WriteLine("Bitte Lieblingsfarbe und -zahl angeben!");
   return;
}
```

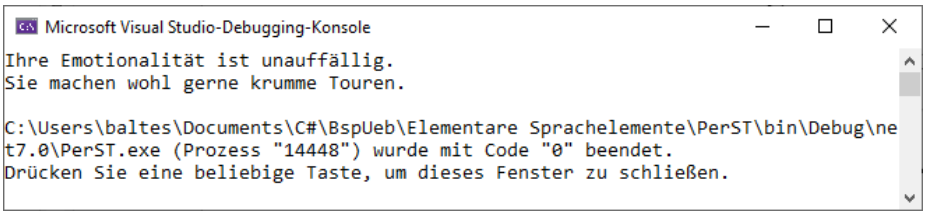
Über die Anzahl der Befehlszeilenargumente informiert das **String[]** – Objekt `args` mit seiner **Length**-Eigenschaft. Wir haben die **return**-Anweisung im Abschnitt 4.7.2.3.1 schon als Stopper im Rahmen der **switch**-Anweisung kennengelernt. Ihre „offizielle“ Behandlung erfolgt im Abschnitt 5.3.1.2 im Zusammenhang mit den Methoden.

Für den Programmstart aus dem Visual Studio kann man die Befehlszeilenargumente nach dem Menübefehl

Debuggen > *Projektname:***Debugeigenschaften**

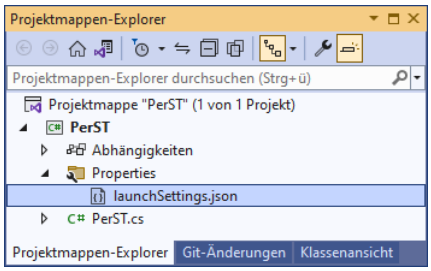
vereinbaren, z. B.:

Aus diesen Befehlszeilenargumenten leitet das Programm die folgende Persönlichkeitsbeurteilung ab:



Aufgrund der eingetragenen Befehlszeilenargumente …

- erscheint im Visual Studio - **Projektmappen-Explorer** der Knoten **Properties**



  mit der Datei **launchSettings.json**:

```
{
 "profiles": {
   "PerST": {
     "commandName": "Project",
     "commandLineArgs": "Blau 17"
   }
 }
}
```

- erscheint im Visual Studio - Projektordner der Unterordner **Properties** mit der Datei **launchSettings.json**.

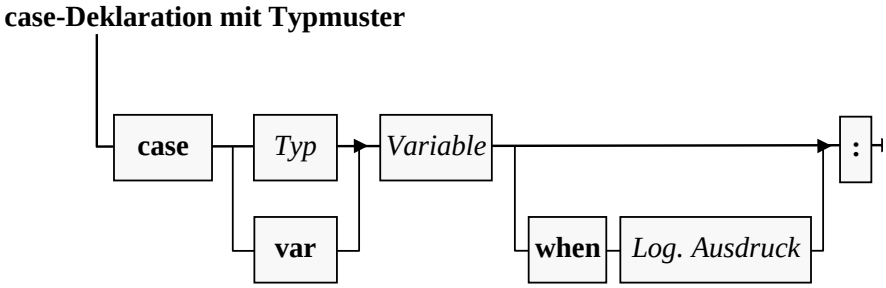
4.7.2.3.3 case-Deklaration mit Typmuster

Mit C# 7.0 hat die **switch**-Anweisung zwei Erweiterungen erfahren:

- beliebiger Datentyp für den steuernden Ausdruck
- **case**-Deklaration mit Typmuster (engl.: *type pattern*)

Die Vorteile der **case**-Deklaration mit Typmuster sind teilweise auch für Programmiereinsteiger nachvollziehbar und nützlich. Hier ist vor allem die Möglichkeit von Interesse, in einem **case** einen kompletten Wertbereich behandeln zu können, ohne die Werte einzeln auflisten zu müssen. Allerdings sind manche Details der flexiblen Typmuster-Falldeklaration für Einsteiger schwer zu verdauen. Es ist daher vertretbar, wenn Sie sich vorläufig auf die im Abschnitt 4.7.2.3.1 beschriebene klassische **switch**-Syntax beschränken und Ihre C# - Kompetenz später um die sehr empfehlenswerten **switch**-Neuerungen erweitern.

Das Syntaxdiagramm zur **case**-Deklaration mit Typmuster[1]



ergänzt das Syntaxdiagramm zur **switch**-Anweisung (siehe Beginn von Abschnitt 4.7.2.3).

Statt einen im Fall zu behandelnden Typ (z. B. eine Klasse) anzugeben, kann man mit dem Schlüsselwort **var** den deklarierten Typ des steuernden Ausdrucks wählen. Dieses sogenannte ***var**-Muster* (engl.: *var pattern*) passt natürlich immer (auch beim Wert **null**), und in der Regel fordert man für passende Fälle zusätzlich per **when**-Klausel den Wert **true** für einen logischen Ausdruck.

Die Variable enthält den Wert des steuernden Ausdrucks und steht sowohl in der Bedingung als auch in den Anweisungen des Falls zur Verfügung.

Insgesamt können die folgenden Konstellationen dazu führen, dass zur Laufzeit ein Typmuster-**case** für den Wert des steuernden Ausdrucks als passend beurteilt wird:

- Der Laufzeittyp des steuernden Ausdrucks ist der **case**-Typ oder ein daraus abgeleiteter Typ. Mit der Vererbungsbeziehung zwischen Klassen werden wir uns im Kapitel 7 beschäftigen. Dabei wird sich herausstellen, dass eine Referenzvariable nicht nur auf ein Objekt vom eigenen Typ, sondern auch auf ein Objekt einer abgeleiteten Klasse zeigen kann.
- Ist als **case**-Typ eine *Schnittstelle* (siehe Kapitel 9) angegeben, dann muss der Laufzeittyp des steuernden Ausdrucks diese Schnittstelle implementieren.

Ist eine **when**-Klausel vorhanden, dann muss auch deren Bedingung erfüllt sein, bevor die **case**-Deklaration als passend bewertet wird. In der englischen Literatur wird eine **when**-Klausel auch als *case guard* bezeichnet.

Bei der Fallauswahl per Musterabgleich können sich *mehrere* **case**-Deklarationen als passend erweisen, wobei dann die *erste* zutreffende gewinnt. Der Compiler bemüht sich um eine sinnvolle Reihenfolge der Fälle und verweigert die Übersetzung, wenn er einen nicht erreichbaren Fall feststellt. Diese Diagnose wird allerdings durch die Anwesenheit von **when**-Klauseln verhindert, sodass die Reihenfolge der **case**-Deklarationen in dieser Situation sorgfältig zu wählen ist.

---

[1] Es wird der Stand von C# 7.0 beschrieben. In C# 11 muss zu einem expliziten Typ keine Variable angegeben werden.

Der Compiler erlaubt in einer **switch**-Anweisung einen Mix aus **case**-Deklarationen mit einer Konstanten bzw. mit einem Typmuster und verhindert außerdem nach Möglichkeit, dass eine **case**-Deklaration unter einer allgemeineren steht und daher nie erreicht werden kann. Im folgenden Beispiel bemerkt der Compiler aber nicht, dass die **case**-Deklaration

```
case string s when s.Contains("ha")
```

unerreichbar ist, weil die allgemeinere Deklaration

```
case string s when s.Length > 0:
```

darüber steht:

Quellcode	Ausgabe

```
string str = "hallo";
switch (str) {
    case null:
        Console.WriteLine("null");
        break;
    case "hi":
        Console.WriteLine("string-Literal");
        break;
    case string s when s.Length > 0:
        Console.WriteLine("Typ String, Länge = " + s.Length);
        break;
    case string s when s.Contains("ha"):
        Console.WriteLine("Typ String, enthält \"ha\"");
        break;
    case string s:
        Console.WriteLine("Typ String");
        break;
}
```

```
Typ String, Länge = 5
```

In einer **case**-Deklaration mit dem Referenzliteral **null** wird eine spezielle Konstante verwendet, was auch schon in C# 6.0 (zusammen mit dem Datentyp **String**) möglich war.

Per **when**-Klausel lässt sich eine **case**-Deklaration mit Bereichsangabe realisieren, die in der klassischen **switch**-Syntax (bis C# 6.0) oft vermisst wurde, z. B.:

Quellcode	Ausgabe

```
int i= 5;
switch (i) {
    case 3:
        Console.WriteLine("3");
        break;
    case int ia when ia > 3  &&  ia <= 10:
        Console.WriteLine("von 4 bis 10");
        break;
    case 99:
        Console.WriteLine("99");
        break;
    default:
        Console.WriteLine("default");
        break;
}
```

```
von 4 bis 10
```

Weil das **var**-Muster erwähnt wurde, soll es auch in einem Beispiel vorgeführt werden, obwohl seine aktuelle Bedeutung für Ihren Lernfortschritt gering ist. Die folgende **switch**-Anweisung verarbeitet beliebige Objekte:

```
Object obj = "123";
//obj = null;
//obj = 123;
//obj = new Random();

switch (obj) {
    case null:
        Console.WriteLine("null");
        break;
    case String str:
        Console.WriteLine("String mit Länge: " + str.Length);
        break;
    case var x when x.GetType().IsPrimitive:
        Console.WriteLine("Primitiver Typ mit Wert: " + x.ToString());
        break;
    default:
        Console.WriteLine("Nicht unterstützter Typ, ToString(): " + obj.ToString());
        break;
}
```

Wenn der steuernde Ausdruck zur Laufzeit auf ein **String**-Objekt zeigt, dann wird die Länge der Zeichenfolge ausgegeben. Der Fall einer Referenz auf **null** wird abgefangen, damit beim Aufruf der Methode **GetType()** kein Laufzeitfehler vom Typ **NullReferenceException** auftreten kann. Hat der steuernde Ausdruck einen beliebigen primitiven Typ (also einen elementaren Typ ungleich **decimal**, vgl. Abschnitt 4.3.4), dann wird sein Wert ausgegeben.

Bei der schrittweisen Einführung der **switch**-Optionen in C# machen wir im gleich folgenden Abschnitt 4.7.2.4 weiter mit dem in C# 8 eingeführten **switch**-*Ausdruck*. Im Abschnitt 15.2 werden wir die sukzessive erweiterten Optionen zum Musterabgleich behandeln, von denen die **switch**-Anweisung, der **switch**-Ausdruck und der zum Typtest geeignete **is**-Operator (siehe Abschnitt 7.7) profitieren.

4.7.2.4 switch-Ausdruck

Zur Ergänzung der **switch**-Anweisung (siehe Abschnitt 4.7.2.3) wurde in C# 8 der **switch**-Ausdruck eingeführt.[1] Wie die im Abschnitt 4.7.2.3.3 beschriebene **case**-Deklaration per Typmuster ist auch der **switch**-Ausdruck eher für fortgeschrittene Programmierer zu empfehlen. Daher beschränken wir uns an dieser Stelle auf die Beschreibung der Grundidee und liefern im Abschnitt 15.2 eine vollständige Behandlung nach.

Die Verwendung eines **switch**-Ausdrucks bietet sich z. B. an, wenn eine Abbildung von einem Ausgangstyp in einen Ergebnistyp stattfinden soll. Im folgenden Beispiel wird eine Eingabe vom Typ **int** in eine Ausgabe vom Typ **String** übersetzt. In einer Personendatenbank sei der Charakter unter Verwendung der vier Temperamentstypen des griechischen Philosophen Hippokrates (melancholisch, cholerisch, phlegmatisch, sanguinisch) durch eine **int**-Variable gespeichert worden.[2] Per **switch**-Ausdruck sollen die Zahlen in aussagekräftige Zeichenfolgen übersetzt werden:

---

[1] Wir befinden uns gerade im Abschnitt über die *Anweisungen*, und die **switch**-Ausdrücke gehören eigentlich in den Abschnitt 4.5 über die *Operatoren und Ausdrücke*. Allerdings ist die Behandlung der **switch**-Ausdrücke *nach* den deutlich älteren **switch**-Anweisungen didaktisch sinnvoller.

[2] Hippokrates lebte ca. von 460 bis 370 v. Chr.

Quellcode	Ausgabe

`int charCode = 3;`
`String charLabel = charCode switch {`
`  1 => "melancholisch",`
`  2 => "cholerisch",`
`  3 => "phlegmatisch",`
`  4 => "sanguinisch",`
`  _ => "undefiniert"`
`};`
`Console.WriteLine($"Charakter: {charLabel}");`
	`Charakter: phlegmatisch`

Den Ergebnistyp eines **switch**-Ausdrucks ermittelt der Compiler per Typinferenz, sodass die implizite Typisierung von lokalen Variablen über das Schlüsselwort **var** erlaubt ist, z. B.:

```
var charLabel = charCode switch {
    1 => "melancholisch",
    2 => "cholerisch",
    3 => "phlegmatisch",
    4 => "sanguinisch",
    _ => "undefiniert"
};
```

Ein **switch**-Ausdruck kann als Operand für einen komplexeren Ausdruck verwendet bzw. in eine Anweisung integriert werden. Wie wir bereits aus dem Abschnitt 4.2.2.2 wissen, darf man seit C# 11 bei der Zeichenfolgeninterpolation auch Leerzeilen verwenden, sodass sich ein **switch**-Ausdruck übersichtlich in eine Ausgabeanweisung integrieren lässt, z. B.:

Quellcode	Ausgabe

`int charCode = 3;`
`Console.WriteLine($"Charakter: {charCode switch {`
`  1 => "melancholisch",`
`  2 => "cholerisch",`
`  3 => "phlegmatisch",`
`  4 => "sanguinisch",`
`  _ => "undefiniert"`
`}}");`
	`Charakter: phlegmatisch`

Bei **switch**-Ausdrücken sind die folgenden Syntaxregeln zu beachten:

- Zu den Fällen gehören keine Anweisungen, sondern Ausdrücke, die syntaktisch elegant mit dem Symbol => zugewiesen werden.
- Es muss *kein Durchfall* (z. B. per **break**) verhindert werden.
- Per Unterstrich können alle sonstigen Werte des steuernden Ausdrucks angesprochen werden. Der Unterstrich wird in C# oft zur Kennzeichnung von Fällen verwendet, die ausgeschlossen bzw. verworfen werden sollen (engl. *discarded*).
- Damit unter Verwendung eines **switch**-Ausdrucks eine vollständige Anweisung entsteht (z. B. eine Wertzuweisung), muss ein Semikolon am Ende der Anweisung (also hinter der schließenden Klammer des **switch**-Blocks) stehen.

Im Vergleich zur Lösung per **switch**-Ausdruck führt die traditionelle **switch**-Anweisung im Beispiel zu einem höheren Aufwand und zu einem weniger übersichtlichen Ergebnis:[1]

[1] Die Lösung mit der Hilfsvariablen `charLabel` hat auch Vorteile.

```
int charCode = 3;
String charLabel;
switch (charCode) {
    case 1:
        charLabel = "melancholisch";
        break;
    case 2:
        charLabel = "cholerisch";
        break;
    case 3:
        charLabel = "phlegmatisch";
        break;
    case 4:
        charLabel = "sanguinisch";
        break;
    default:
        charLabel = "undefiniert";
        break;
};
Console.WriteLine($"Chrakter: {charLabel}");
```

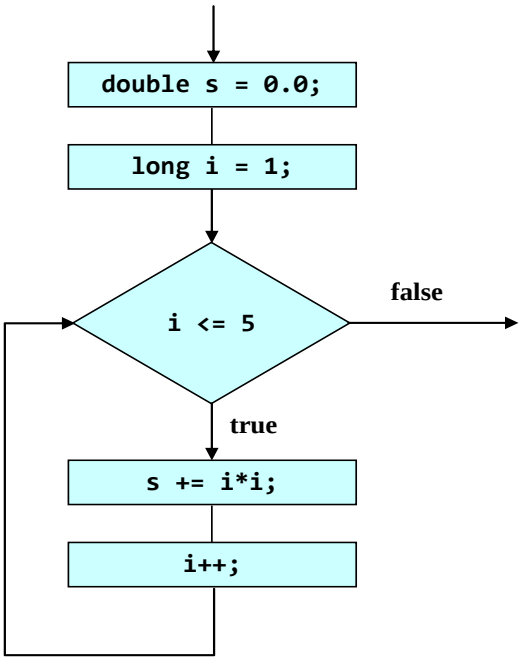
4.7.3 Wiederholungsanweisungen

Eine Wiederholungsanweisung (kurz: *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 Quellcode zu schreiben.

Im folgenden Flussdiagramm ist ein iterativer Algorithmus zu sehen, der die Summe der quadrierten natürlichen Zahlen von 1 bis 5 berechnet:[1]

[1] Das Verzweigungssymbol sieht aus darstellungstechnischen Gründen etwas anders aus als im Abschnitt 4.7.2, 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 4.6.1) kurz in Erinnerung gerufen werden. Weil die Variable `i` vom Typ **long** ist, kann der Algorithmus bis zur Laufgrenze 3.037.000.499 verwendet werden. Für größere `i`-Werte tritt beim Ausdruck `i*i` ein Überlauf auf, und das Ergebnis ist unbrauchbar. Eine einfache Möglichkeit zur Steigerung der maximalen sinnvollen Laufgrenze besteht darin, für eine Berechnung der Summanden per Gleitkommaarithmetik zu sorgen:

```
(double) i * i
```



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 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 dem 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 über- bzw. unterschritten hat.
- **Iterieren über die Elemente einer Kollektion** (**foreach**)
  Mit der **foreach**-Schleife bietet C# die Möglichkeit, eine Anweisung für jedes Element eines Arrays, einer Zeichenfolge oder einer anderen Kollektion (siehe Kapitel 11) 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 als *vor* dem 4. Schleifendurchgang, geht es bei der Entscheidung zwischen Kopf- und Fußsteuerung lediglich darum, ob auf jeden Fall ein *erster* Schleifendurchgang stattfinden soll (**do**-Schleife) oder nicht (**while**-Schleife).

Die gesamte Konstruktion aus Schleifensteuerung und (Verbund-)anweisung stellt in syntaktischer Hinsicht *eine zusammengesetzte* Anweisung dar.

4.7.3.1 Zählergesteuerte Schleife (for)

Die eingebettete 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 sind. Danach folgt die wiederholt auszuführende, eingebettete (Block-)Anweisung:

for (*Vorbereitung***;** *Bedingung***;** *Aktualisierung***)**
    Anweisung

Zu den drei Bestandteilen der Schleifensteuerung sind einige Erläuterungen erforderlich, wobei anschließend etliche weniger 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 Variablendeklaration mit Initialisierung, z. B.
    ```
    long i = 1
    ```
  - eine Wertzuweisung für eine bereits deklarierte Variable, z. B.:
    ```
    i = 1
    ```

  Im folgenden Programm, das die Summe der quadrierten natürlichen Zahlen von 1 bis 5 berechnet, findet sich die erste Variante:

Quellcode	Ausgabe

`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, ist also nur in deren Anweisung(sblock) sichtbar. Eine möglichst eingeschränkte Sichtbarkeit von lokalen Variablen mindert das Risiko von Programmierfehlern (siehe Abschnitt 4.3.8).

- ***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**, dann wird die eingebettete Anweisung (der Schleifenrumpf) ausgeführt, anderenfalls wird die **for**-Schleife verlassen. Folglich kann es auch passieren, dass überhaupt kein Schleifendurchgang zustande kommt.

- ***Aktualisierung***
  Am Ende jedes Schleifendurchgangs (nach der Ausführung der eingebetteten Anweisung) wird die Aktualisierung ausgeführt. Dabei wird meist die Laufvariable in- oder dekrementiert. In der C# - Sprachreferenz werden Alternativen für die meist als Aktualisierungsvorschrift verwendete In- oder Dekrementoperation beschrieben, z. B.:[1]
  - Wertzuweisung
  - Methodenaufruf

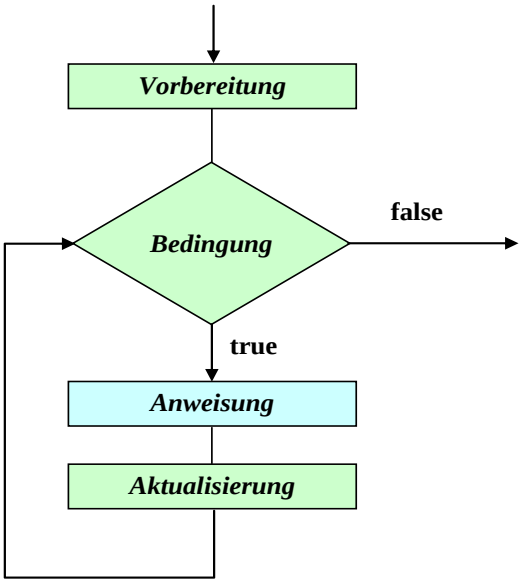
[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/for

Im folgenden Beispiel sorgt ein **WriteLine()** - Aufruf mit Postinkrementausdruck als Argument dafür, dass die Laufvariable protokolliert und inkrementiert wird:

```
for (long i = 1; i <= 5; Console.WriteLine(i++))
    s += i * i;
```

Während es für das Ergebnis der **for**-Schleife irrelevant ist, ob die Aktualisierung eine Präoder eine Postinkrementoperation enthält, hat die Wahl einen Effekt auf die **WriteLine()** - Ausgabe.[1]

Im folgenden Flussdiagramm ist das Ablaufverhalten der **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 akzeptiert, gehören **for**-Schleifenköpfe ohne Vorbereitung oder ohne Aktualisierung, wobei die trennenden Strichpunkte im Schleifenkopf 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 eingebetteten Anweisung modifiziert wird,
- die Schleife per **break**-, **goto**- oder **return**-Anweisung verlassen wird (siehe Abschnitt 4.7.3.5).

4.7.3.2 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 zu später behandelten Themen gehörenden Begriffe *Array*, *Kollektion* und *Interface* nicht beunruhigen.

---

[1] Obwohl manche Personen die Präinkrementoperation für performanter halten, wird mehrheitlich die Postinkrementoperation verwendet, und es bestehen wohl kaum relevante Performanzunterschiede, siehe z. B.:
https://www.oreilly.com/library/view/c-40-unleashed/9780132678926/h4_592.html

Die Steuerungslogik der **foreach**-Schleife:

- Es ist eine Kollektion mit einer Anzahl von Elementen vorhanden, z. B. eine Zeichenfolge mit acht Zeichen.
- Im Schleifenkopf wird eine Iterationsvariable vom Datentyp der Kollektionselemente deklariert, über die in der eingebetteten Anweisung das aktuelle Element angesprochen werden kann.
- Die eingebettete Anweisung der **foreach**-Schleife wird nacheinander für jedes Element der Kollektion ausgeführt.

Die Syntax der **foreach**-Schleife:

foreach (*Elementtyp Iterationsvariable* **in** *Kollektion***)**
    Anweisung

Als Kollektion erlaubt der Compiler:[1]

- ein Array
- eine Instanz eines Typs, der das Interface **IEnumerable** im Namensraum **System.Collections** oder das Interface **IEnumerable<T>** im Namensraum **System.Collections.Generic** implementiert.

Das Beispielprogramm `PerST` im Abschnitt 4.7.2.3.2 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** bzw. **IEnumerable <T>** nicht *explizit* implementiert werden (siehe Kapitel **9**). Es genügt, die Instanzmethode **GetEnumerator()** mit einer Rückgabe vom Typ **IEnumerator** bzw. **IEnumerator<T>** zu implementieren (siehe Abschnitt 9.5), z. B.:

```
var from0T4  = new EnumeratorFrom0To4 ();
foreach (int i in from0T4) System.Console.WriteLine(i);

class EnumeratorFrom0To4 {
  public System.Collections.Generic.IEnumerator<int> GetEnumerator() {
    for (int i = 0; i < 5; i++) yield return i;
  }
}
```

- hätte die **foreach**-Tauglichkeit von Arrays nicht explizit genannt werden müssen, weil jedes Array ein Objekt aus einer Klasse ist, welche die Schnittstelle **IEnumerable** implementiert.

Quellcode	Ausgabe

`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).[1]

Bloch (2018, S. 264ff) empfiehlt nachdrücklich, wegen der folgenden Vorteile die **foreach** - Schleife nach Möglichkeit gegenüber der traditionellen **for**-Schleife zu bevorzugen:

- Oft werden die Flexibilität (und der Aufwand) bei der Initialisierung, Überprüfung und Aktualisierung der Indexvariablen nicht benötigt. In der **foreach** - Syntax beschränkt man sich auf die Elementzugriffsvariable und erhält einen besser lesbaren Quellcode.
- Durch den Verzicht auf eine Indexvariable entfallen Fehlermöglichkeiten.
- Weil in der **foreach** - Schleife für Arrays und Kollektionen dieselbe Syntax verwendet wird, macht der Wechsel der Container-Architektur wenig Aufwand.

Eine wichtige *Einschränkung* der **foreach**-Schleife besteht darin, dass man in ihrer Anweisung über die Iterationsvariable nur *lesend* auf die Kollektionselemente zugreifen kann, sodass z. B. der folgende Versuch zum „Löschen“ der Elemente im **string**-Array `args` misslingt:

```
foreach (string s in args) {
    s = "erased";
}
```

(Lokale Variable) string s

"s" ist hier nicht NULL.

CS1656: "s" ist "foreach-Iterationsvariable". Eine Zuweisung ist daher nicht möglich.

Der Grund für dieses Verhalten besteht darin, dass die Iterationsvariable eine *Kopie* des aktuellen Kollektionselements enthält, sodass eine Änderung sinnlos wäre.[2]

Über eine **for**-Schleife ist der Plan aus dem letzten Beispiel zu realisieren, z. B.:

```
for (int i = 0; i < args.Length; i++)
    args[i] = "erased";
```

Der Versuch, die Zusammensetzung einer Kollektion während der Ausführung einer **foreach**-Schleife zu ändern, führt zu einem Laufzeitfehler vom Typ **InvalidOperationException**. Im folgenden Beispiel wird vergeblich versucht, aus einer Liste mit ganzen Zahlen (siehe Abschnitt 11.3.1 zu Listen mit Elementen eines festen Typs, hier **int**)) die geraden Zahlen zu entfernen:

---

[1] Als syntaktische Besonderheit kennt C# für den Klassennamen **String** einen Alias mit kleinem Anfangsbuchstaben. Das gilt auch für die Klasse **Object**. Im Manuskript werden bewusst beide Schreibweisen verwendet, damit Sie sich an diese spezielle Wahlfreiheit gewöhnen.

[2] Weitere Details sind hier zu finden:
http://stackoverflow.com/questions/4004755/why-is-foreach-loop-read-only-in-c-sharp

```
var liste = new List<int>() { 1, 2, 3, 4, 5, 6, 7 };
foreach (var a in liste)
    if (a % 2 == 0)
        liste.Remove(a);
```

Das Übersetzen des Codes klappt, aber die Ausführung scheitert:

```
Unhandled exception. System.InvalidOperationException: Collection was modified;
enumeration operation may not execute.
```

4.7.3.3 Bedingungsabhängige Schleifen

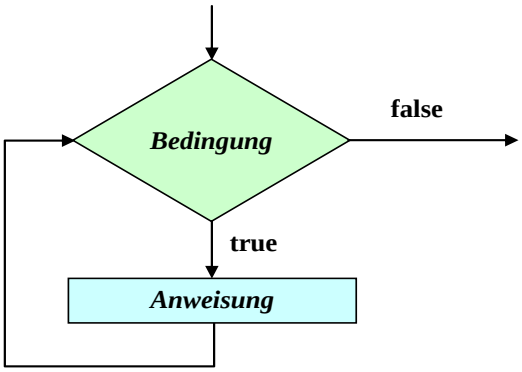
Wie die Erläuterungen zur **for**-Schleife gezeigt haben, ist die Überschrift zu diesem Abschnitt nicht sehr trennscharf, weil bei der **for**-Schleife ebenfalls eine beliebige Terminierungsbedingung angegeben werden darf. In vielen Fällen ist es eine Frage des persönlichen Geschmacks, welche Wiederholungsanweisung man zur Lösung eines konkreten Iterationsproblems einsetzt. Unter der aktuellen Abschnittsüberschrift werden traditionsgemäß die **while**- und die **do**-Schleife beschrieben.

4.7.3.3.1 while-Schleife

Die **while**-Anweisung kann als vereinfachte **for**-Anweisung beschrieben werden: Wer im Kopf einer **for**-Schleife auf Vorbereitung und Aktualisierung verzichten möchte, ersetzt besser das Schlüsselwort **for** durch **while** und erhält dann die folgende Syntax:

while (*Bedingung***)**
    Anweisung

Wie bei der **for**-Anweisung wird die Bedingung *vor Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, dann wird die eingebettete Anweisung (ein weiteres Mal) ausgeführt, anderenfalls wird die **while**-Schleife verlassen, eventuell noch vor dem ersten Durchlauf:



Das im Abschnitt 4.7.3.1 vorgestellte Beispielprogramm zur Quadratsummenberechnung mit Hilfe einer **for**-Schleife kann leicht auf die Verwendung einer **while**-Schleife umgestellt werden:

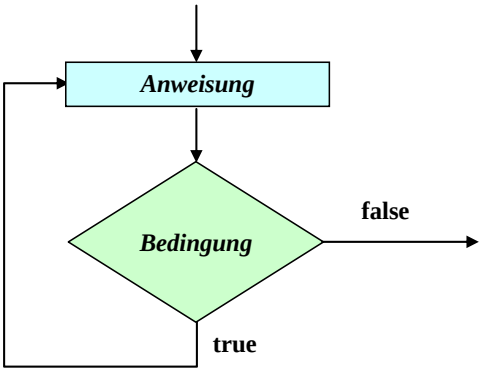
Quellcode	Ausgabe

`double s = 0.0;`
`long i = 1;`
`while (i <= 5) {`
`  s += i * i;`
`  i++;`
`}`
`Console.WriteLine("Quadratsumme = " + s);`
	`Quadratsumme = 55`

Ein Nachteil der im Beispiel verwendeten **while**-Schleife gegenüber der im Abschnitt 4.7.3.1 beschriebenen **for**-Schleife besteht darin, dass die Laufvariable `i` *außerhalb* der **while**-Schleife deklariert werden muss, was zu einem unnötig großen Gültigkeitsbereich für diese lokale Variable führt. Außerdem sind bei der **while**-Lösung der Schreibaufwand höher und die Lesbarkeit schlechter. Dieses Vergleichsergebnis gilt aber nur für Schleifen mit einer zählergesteuerten Logik.

4.7.3.3.2 do-Schleife

Bei der **do**-Schleife wird die Fortsetzungsbedingung *am Ende* der Schleifendurchläufe geprüft, sodass mindestens *ein* Durchlauf stattfindet:



Das Schlüsselwort **while** tritt auch in der Syntax der **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. Im folgenden Programm wird 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

`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).

4.7.3.4 Endlosschleifen

Bei einer **for**-, **while**- oder **do**-Schleife kann es in Abhängigkeit von der Fortsetzungsbedingung passieren, dass die eingebettete Anweisung so lange wiederholt wird, bis das Programm von außen abgebrochen wird. Solche Endlosschleifen sind in seltenen Fällen intendiert, meist aber das Ergebnis eines gravierenden Programmierfehlers. Befindet sich ein Programm in diesem Zustand, dann muss es mit Hilfe des Betriebssystems abgebrochen werden, bei unseren Konsolenanwendungen unter Windows z. B. über die Tastenkombination **Strg+C**.

Im folgenden Beispiel resultiert eine Endlosschleife aus einer ungeschickten Identitätsprüfung bei **double**-Werten (vgl. Abschnitt 4.5.3):

```
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!");
```

4.7.3.5 Schleifen(durchgänge) vorzeitig beenden

Mit der **break**-Anweisung, die uns schon im Abschnitt 4.7.2.3.1 als Bestandteil der **switch**-Anweisung begegnet ist, kann eine Schleife vorzeitig verlassen werden, wobei die Methode hinter der Schleife fortgesetzt wird.

Mit der **continue**-Anweisung erreicht man, dass der Anweisungsblock des aktuellen Schleifendurchgangs verlassen und die Bearbeitung folgendermaßen fortgesetzt wird:

- Bei der **for**-Schleife wird …
  - die Aktualisierung zum aktuellen Schleifendurchgang ausgeführt,
  - die Fortsetzungsbedingung geprüft und ggf. mit dem nächsten Schleifendurchgang begonnen.
- Bei der **while**- und der **do**-Schleife wird die Fortsetzungsbedingung geprüft und ggf. mit dem nächsten Schleifendurchgang begonnen.

Im folgenden Programm zur (relativ primitiven) Primzahlendiagnose kommt die schon im Abschnitt 4.4.1 erwähnte Ausnahmebehandlung per **try-catch** - Anweisung zum Einsatz, die im Kapitel 13 ausführlich behandelt wird. Wir leisten uns den Vorgriff, weil sich mit dieser Technik ungeeignete Benutzereingaben (negative oder zu große Werte, nicht interpretierbare Zeichenfolgen) bequem abfangen lassen, wobei ein sinnvoller **continue**-Einsatz resultiert:[1]

```
using System;
class Primitiv {
  static void Main() {
    bool tg;
    ulong i, mtk, zahl;
    Console.WriteLine("Einfacher Primzahlendetektor\n");
    while (true) {
      Console.Write("Zu untersuchende ganze Zahl von 2 bis 2^64-1 oder 0 zum Beenden: ");
      try {
        zahl = Convert.ToUInt64(Console.ReadLine());
      } catch {
        Console.WriteLine("Keine ganze Zahl (im zulässigen Bereich)!\n");
        continue;
      }
      if (zahl == 1) {
        Console.WriteLine("1 ist per Definition keine Primzahl.\n");
        continue;
      }
```

---

[1] Ein Visual Studio – Projekt mit dem Primzahlendiagnoseprogramm ist hier zu finden:
...\BspUeb\Elementare Sprachelemente\Primitiv
Wie im Abschnitt 4.1.2 vereinbart, erscheinen die in der Datei **BspUeb.zip** enthaltenen Programme im Manuskript *inklusive* Klassendefinitionskopf, damit der Klassenname zu sehen ist, und das Auffinden der gleichnamigen Quellcodedatei erleichtert wird. Der **Main()** – Definitionskopf ist dann ebenfalls erforderlich, und aus dem Verzicht auf die mit C# 10 eingeführten *impliziten* **using**-Direktiven resultiert das traditionelle Erscheinungsbild eines C# - Konsolenprogramms.

```
        if (zahl == 0)
          break;

        tg = false;
        mtk = (ulong) (Math.Sqrt(zahl) + 0.5); //Maximaler Teilerkandidat
        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!");
      Console.ReadLine();
    }
  }
```

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.

Auch nach der Eingabe 1 wird der aktuelle Schleifendurchgang per **continue** vorzeitig beendet, weil keine Prüfung erforderlich ist. Der Benutzer wird darüber informiert, dass die 1 per Definition keine Primzahl ist.

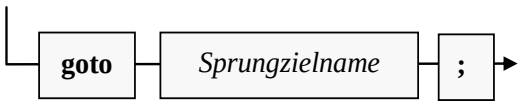
Durch Eingabe der Zahl 0 kann das Beispielprogramm beendet werden, wobei die absichtlich konstruierte **while** - „Endlosschleife“ per **break** verlassen wird.

Man hätte die **continue**- und die **break**-Anweisungen zwar vermeiden können (siehe Übungsaufgabe im Abschnitt 4.7.4), doch werden beim vorgeschlagenen Verfahren lästige Sonderfälle (unzulässige Werte, 1 als Kandidat, 0 als Terminierungssignal) auf besonders übersichtliche Weise abgehakt, bevor der Kernalgorithmus startet.

Neben der **break**-Anweisung stehen weitere, seltener benötigte Anweisungen zum vorzeitigen Verlassen einer Schleife zur Verfügung:

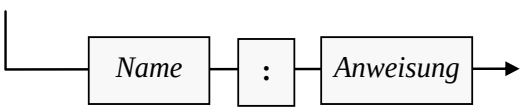
- **goto**-Anweisung
  Über die bereits im Zusammenhang mit der **switch**-Verzweigung (siehe Abschnitt 4.7.2.3.1) beschriebene **goto**-Anweisung

  goto-Anweisung

  kann ein Sprungziel

  Sprungziel

  außerhalb der Wiederholungsanweisung (aber 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.

- **return**-Anweisung
  Über die bereits mehrfach verwendete, aber noch nicht offiziell behandelte **return**-Anweisung (siehe Abschnitt 5.3.1.2) wird die Methode verlassen, was im Fall der Methode **Main()** einer Beendigung des Programms gleichkommt.

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 einen echten Teiler hat, dann besitzt sie auch einen echten Teiler kleiner oder gleich $\sqrt{z}$. Wenn man *keinen* echten Teiler kleiner oder gleich $\sqrt{z}$ gefunden hat, dann kann man die Suche einstellen, und z ist eine Primzahl.

Ist z. B. die Primzahl 23 mit der Wurzel 4,796... zu untersuchen, dann kann die Suche mit dem Teilerkandidaten 4 enden (größte ganze Zahl ≤ Wurzel). Es sind also nur die Teilerkandidaten 2, 3 und 4 zu untersuchen, sodass viel sinnloser Aufwand eingespart wird. Nach dem Teilerkandidaten 2 auch noch ein Vielfaches dieser Zahl (z. B. 4) zu untersuchen, ist natürlich nicht sehr intelligent. Daher trägt das Programm den Namen *Primitiv*.

Zur Berechnung der Quadratwurzel verwendet das Beispielprogramm die Methode **Sqrt()** aus der Klasse **Math**, über die man sich bei Bedarf in der BCL-Dokumentation informieren kann. Hinter der übertrieben aufwändig wirkenden Anweisung

```
mtk = (ulong) (Math.Sqrt(zahl) + 0.5);
```

stecken die folgenden Überlegungen:

- Der Laufindex `i` (Datentyp **ulong**) in der anschließenden **for**-Schleife wird wiederholt mit `mtk` verglichen. Damit dabei keine implizite Typumwandlung erforderlich ist, hat auch `mtk` den Typ **ulong** erhalten, sodass für das **Sqrt()** - Ergebnis eine explizite Typanpassung erforderlich ist. Dabei kann es *nicht* zum Ganzzahlüberlauf kommen, weil das **Sqrt()** - Argument eine **ulong**-Zahl ist.
- Ist eine ganze Zahl das Quadrat einer Primzahl, dann ist sie selbst keine Primzahl, sondern eine sogenannte *Sekundzahl*. Bei der Berechnung der Quadratwurzel aus einer Zahl im **ulong** - Wertebereich per Gleitkommaarithmetik kommt es in der Regel zu einer Abweichung vom mathematisch korrekten Ergebnis (vgl. Abschnitte 4.3.5 und 4.5.7.1). Bei einer Sekundzahl könnte das **Sqrt()** - Ergebnis knapp unter dem korrekten ganzzahligen Wert liegen, was bei der Wandlung in **ulong** (durch Abschneiden der Nachkommastellen) zu einem Fehler führen würde. Infolgedessen würde die Sekundzahl falsch als Primzahl erkannt. Um dies zu verhindern, wird vor der Wandlung in den Typ **ulong** der Wert 0,5 addiert und so eine Rundung erzwungen.[1]

[1] Vermutlich ist das Addieren von 0,5 überflüssig, weil der im **Math.Sqrt()** implementierte Algorithmus zum Radizieren (Wurzelziehen) bei Quadratzahlen zum korrekten (ganzzahligen) Ergebnis kommt.

4.7.4 Übungsaufgaben zum Abschnitt 4.7

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 im folgenden Quellcode 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

`bool b = false;`
`if (b = false)`
`  Console.WriteLine("b ist False");`
`else`
`  Console.WriteLine("b ist True");`
`Console.WriteLine("Kontrollausgabe von b: " + b);`
	`b ist True`
	`Kontrollausgabe von b: False`

3) Erstellen Sie eine Variante des Primzahlen-Diagnoseprogramms aus dem Abschnitt 4.7.3.5, die ohne **break** und **continue** auskommt.

4) Wie oft wird die folgende **while**-Schleife ausgeführt?

```
int i = 0;
while (i < 100);
{
   i++;
   Console.WriteLine(i);
}
```

5) Wegen der beschränkten Genauigkeit bei der Speicherung von Gleitkommazahlen (siehe Abschnitt 4.3.4) 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]

$$1{,}0 + 2^{-i} > 1{,}0$$

6) In dieser Aufgabe sollen Sie zwei 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 in der Methode `Kuerze()` der Klasse `Bruch` benutzten Algorithmus (siehe Abschnitt 1.3). Sein Problem besteht darin, dass bei stark unterschiedlichen Zahlen *u* und *v* sehr viele Subtraktions-Operationen erforderlich sind. In der meistbenutzten Variante des Euklidischen

[1] Im Abschnitt 4.3.5.1, der allerdings für Programmiereinsteiger nur bedingt geeignet ist, finden sich Hintergrundinformationen zur Aufgabe.

Verfahrens wird dieses Problem vermieden, indem an Stelle der Subtraktion die Modulo-Operation zum Einsatz kommt, basierend auf dem folgenden 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 1.4): 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* restfrei teilbar ist.
> Trifft dies zu, dann 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.

Zur Messung des Zeitaufwands eignet sich z. B. ein Objekt der Klasse **Stopwatch** aus dem Namensraum **System.Diagnostics**. Ein Messvorgang wird ...

- mit der Methode **Start()** gestartet
  public void Start()
- und mit der Methode **Stop()** beendet
  public void Stop()

Für die Beispielwerte *u* = 999000999 und *v* = 36 liefern beide Euklid-Varianten sehr verschiedene Laufzeiten (CPU: Intel Core i3 550, Betriebssystem: Windows 10 (64 Bit)):

```
GGT-Bestimmung mit Euklid (Differenz)

Erste Zahl:     999000999

Zweite Zahl:    36

GGT:            9

Benöt. Zeit:    118 Millisek.
```

```
GGT-Bestimmung mit Euklid (Modulo)

Erste Zahl:   999000999

Zweite Zahl:  36

GGT:          9

Benöt. Zeit:  0 Millisek.
```

Bei wiederholten Messungen stellt sich eine erhebliche Variabilität der Laufzeiten für den Differenz-Algorithmus heraus, wobei zufällige Schwankungen und systematische Veränderungen („Warmlaufen“) festzustellen sind. Für eine Leistungsbeurteilung mit hoher Genauigkeit ist ein erheblicher Aufwand erforderlich, den z. B. das NuGet-Paket **BenchmarkDotNet** betreibt. Beim sehr schnellen Modulo-Algorithmus zeigen sich keine Laufzeitunterschiede im Millisekundenbereich.

5 Klassen und Objekte

Software-Entwicklung mit C# besteht nach unserem bisherigen Kenntnisstand im Wesentlichen aus der Definition von **Klassen**, 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, dann kann eine handelnde Klasse diese Rolle übernehmen.[1] Sind hingegen *mehrere* Individuen einer Gattung erforderlich (z. B. mehrere Brüche in einem Bruchrechnungsprogramm oder mehrere Fahrzeuge in einer Speditionsverwaltung), dann wird eine Klasse mit Bauplancharakter benötigt.

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 Aufträge ausgeführt bzw. Algorithmen realisiert. Ein agierendes (eine Methode ausführendes) Objekt bzw. eine agierende Klasse muss nicht alles selbst erledigen, sondern kann vordefinierte (z. B. der BCL entstammende) oder im Programm definierte Klassen einspannen, z. B.:

- Eine BCL-Klasse wird beauftragt:
  ```
  int az = Math.Abs(zaehler);
  ```
- Ein explizit im Programm erstelltes Objekt aus einer im Programm definierten Klasse wird beauftragt:
  ```
  Bruch b1 = new Bruch();
  b1.Frage();
  ```

Mit dem „Beauftragen“ eines Objekts oder einer Klasse bzw. mit dem „Zustellen einer Botschaft“ ist nichts anderes gemeint als ein Methodenaufruf.

Unsere vorläufige, auch im aktuellen Kapitel 5 zugrundeliegende Vorstellung von einem Computer-Programm lässt sich so beschreiben:

- Ein Programm besteht aus Klassen, die als Baupläne für Objekte und/oder als Akteure dienen.
- Die Akteure (Objekte und Klassen) haben jeweils einen Zustand (abgelegt in Feldern).
- Sie können Botschaften empfangen und senden (Methoden ausführen und aufrufen).

Auf den bisher vermittelten Eindrücken von der objektorientierten Programmierung (OOP) aufbauend kommen wir nun zur systematischen Behandlung dieser Software-Technologie. Für die im Kapitel 1 speziell für größere Projekte empfohlene objektorientierte Analyse und Modellierung, z. B.

---

[1] Eine nur *einfach* zu besetzende Rolle von einer Klasse übernehmen zu lassen, ist keinesfalls in jeder Situation eine ideale Design-Entscheidung und wird im Manuskript hauptsächlich der Einfachheit halber bevorzugt. In einer späteren Phase auf dem Weg zum professionellen Entwickler sollte man sich unbedingt mit dem sogenannten *Singleton-Pattern* beschäftigen (siehe z. B. Bloch 2018, S. 17ff). Dabei geht es um Klassen, von denen innerhalb einer Anwendung garantiert nur *ein* Objekt entsteht. Hier fungiert also ein Objekt statt einer Klasse als Solist, was etliche Vorteile bietet, z. B.:

- Die Adresse des Solo-Objekts kann an Methoden als Parameter übergeben werden.
- Die Vererbungstechnik der OOP wird besser unterstützt (inkl. Polymorphie, siehe Kapitel 7).
- Es ist für den Solisten oft erforderlich, die abstrakten *Instanzmethoden* von Schnittstellen zu implementieren (siehe Kapitel 9).

mit Hilfe der Unified Modeling Language (UML), fehlt dabei leider die Zeit (siehe z. B. Balzert 2011; Booch et al. 2007).

5.1 Überblick, historische Wurzeln, Beispiel

5.1.1 Einige Kernideen und Vorzüge der OOP

Lahres & Rayman (2009, Kapitel 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:

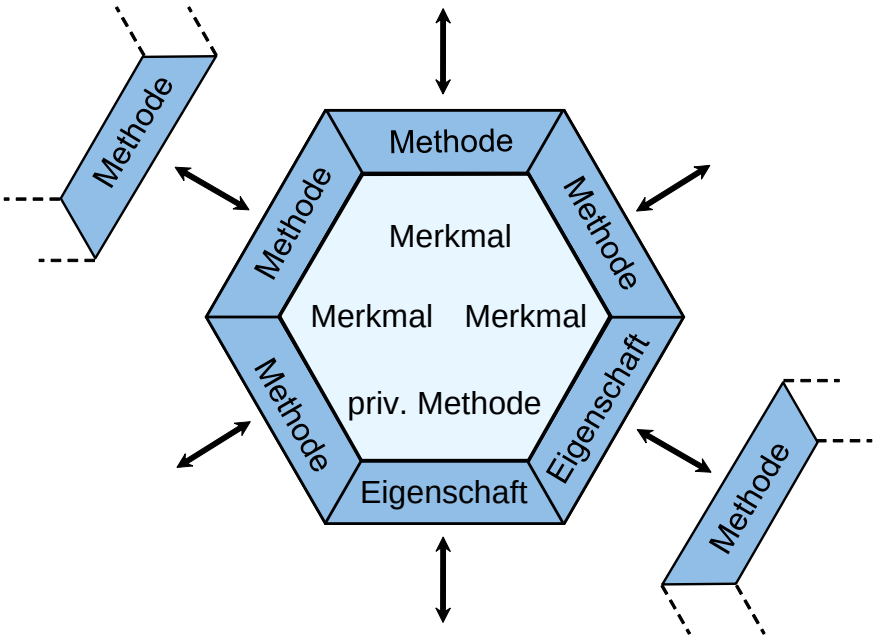
- **Datenkapselung**
  Eine Klasse erlaubt in der Regel fremden Klassen keinen direkten Zugriff auf ihre Zustandsdaten. So wird das Risiko für das Auftreten inkonsistenter Zustände reduziert. Außerdem kann der Klassendesigner Implementierungsdetails ohne Nebenwirkungen auf andere Klassen ändern. Mit der Datenkapselung haben wir uns schon im Kapitel 1 beschäftigt.
- **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 (z. B. Methoden) Modifikationen zur Anpassung an die neue Aufgabe vorzunehmen.
- **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. Welche Methode ausgeführt wird, entscheidet sich erst zur Laufzeit (späte Bindung). Dank Polymorphie ist eine lose Kopplung von Klassen möglich, und die Wiederverwendbarkeit von vorhandenem Code wird verbessert.

C# bietet sehr gute Voraussetzungen zur Nutzung dieser Konstruktionsprinzipien beim Entwurf von stabilen, wartungsfreundlichen, anpassungsfähigen und auf Wiederverwendung angelegten Software-Systemen, kann aber keinen Entwickler zur Realisation der Prinzipien zwingen.

5.1.1.1 Datenkapselung und Modularisierung

In der objektorientierten Programmierung (OOP) wird die vorher übliche Trennung von Daten und Operationen aufgegeben. Ein objektorientiertes Programm besteht 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**, engl.: **information hiding**) und so vor ungeschickten Zugriffen schützen. Die meisten Methoden und Eigenschaften einer Klasse sind hingegen von außen ansprechbar und bilden ihre **Schnittstelle** bzw. ihr API. Dies kommt in der folgenden Abbildung zum Ausdruck, die Sie im Wesentlichen schon aus dem Abschnitt 1.2 kennen:

[1] Diese Darstellung ist etwas vereinfachend. Hinter einer Eigenschaft muss nicht unbedingt ein einzelnes Feld stehen.



Es kann aber auch *private Methoden* für den ausschließlich klasseninternen Gebrauch geben. Ebenso sind *öffentliche Felder* möglich, die damit zur Schnittstelle einer Klasse gehören. Solche Felder sind oft als *konstant* deklariert (siehe Abschnitt 5.2.5) und damit vor Veränderungen geschützt. Ein Beispiel ist das **double**-Feld **PI** für die trigonometrische Konstante π (≈ 3.1416) in der BCL-Klasse **Math**.

Klassen mit Datenkapselung realisieren besser als frühere Software-Technologien (siehe Abschnitt 5.1.2) das Prinzip der **Modularisierung**, das ein unverzichtbares Mittel der Software-Entwickler zur Bewältigung von umfangreichen Projekten ist.[1]

Zugunsten einer häufigen und erfolgreichen Wiederverwendung sind Klassen mit hoher Komplexität (vielfältigen Aufgaben) und auch Methoden mit hoher Komplexität zu vermeiden. Als eine Leitlinie für den Entwurf von Klassen findet 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 breite Zustimmung (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 Interaktion mit dem Benutzer über eine grafische Bedienoberfläche kümmert.[3]

Aus der Datenkapselung und anderen Maßnahmen zur Förderung der Modularisierung (z. B. Klassendesign nach dem Prinzip einer einzigen Verantwortung) ergeben sich gravierende Vorteile für die Software-Entwicklung:

---

[1] In früheren Versionen des Manuskripts befand sich an dieser Stelle zur Auflockerung ein Verweis auf den römischen Feldherrn Julius Cäsar (100 v. Chr. - 44 v. Chr.), dem das lateinische Motto *Divide et impera* (*Teile und Herrsche*) zugeschrieben wird. Seit Wladimir Putin erneut einen völkerrechtswidrigen Überfalls auf ein Nachbarland begangen hat (wie zuvor Adolf Hitler und andere), sind Feldherrn für längere Zeit zur Auflockerung nicht mehr geeignet.

[2] Der als *Uncle Bob* bekannte Software-Berater und Autor erläutert auf der folgenden Webseite seine Vorstellungen von objektorientiertem Design: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[3] In einem sehr kleinen Programm ist es angemessen, wenn eine einzige Klasse für die „Geschäftslogik“ und die Benutzerinteraktion zuständig ist (siehe Beispielprogramm im Abschnitt 3.3.7).

- **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 verhindert, dass fremde Klassen den Nenner eines Bruchs auf den Wert 0 setzen. Fremde Klassen können einen Nenner nur über die Eigenschaft `Nenner` verändern, die aber den Wert 0 nicht akzeptiert. Bei einer anderen Klasse mag es wichtig sein, dass für eine *Gruppe* von Feldern bei jeder Änderung gewisse 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. Zur Sicherstellung wichtiger Bedingungen kann es sinnvoll sein, auch Feldzugriffe durch *klasseneigene* Methoden über die zuständigen Eigenschaften vorzunehmen. Bei der Software-Entwicklung im professionellen Umfeld spielt das systematische Testen eines Programms (**Unit Testing**) eine entscheidende Rolle. Ein objektorientiertes Softwaresystem mit Datenkapselung bietet günstige Voraussetzungen für eine möglichst umfassende Testung.
- **Innovationsoffenheit durch gekapselte Details der Klassenimplementation**
  Verborgene Details einer Klassenimplementation kann der Designer ändern, ohne die Kooperation mit anderen Klassen zu gefährden.
- **Produktivität durch wiederholt und bequem verwendbare Klassen**
  Selbständig agierende Klassen, die ein Problem ohne überflüssige Abhängigkeiten von anderen Klassen lösen, sind potenziell in vielen Projekten zu gebrauchen (Wiederverwendbarkeit). Wer als Programmierer eine Klasse verwendet, braucht sich um deren inneren Aufbau nicht zu kümmern, sodass neben dem Fehlerrisiko auch der Einarbeitungsaufwand sinkt. Man kann z. B. in einem GUI-Programm einen kompletten Rich-Text-Editor über eine Klasse aus der Standardbibliothek integrieren, ohne wissen zu müssen, wie der Text intern verwaltet wird.
- **Erfolgreiche Teamarbeit durch abgeschottete Verantwortungsbereiche**
  In großen Projekten können mehrere Programmierer nach der gemeinsamen Definition von Schnittstellen unabhängig voneinander an verschiedenen Klassen arbeiten.

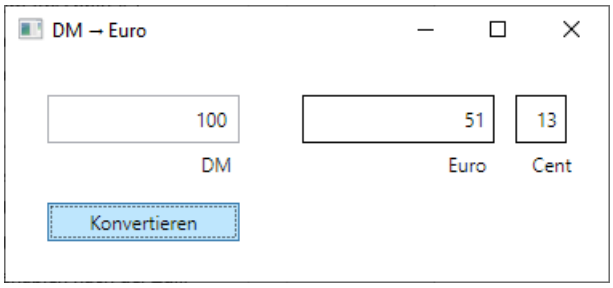
Durch die objektorientierte Programmierung werden auf vielfältige Weise **Kosten reduziert**:

- Vermeidung bzw. schnelle Aufklärung von Programmierfehlern
- gute Chancen für die Wiederverwendung von Software
- gute Voraussetzungen für die Kooperation in Teams

5.1.1.2 Vererbung

Zu den Vorzügen der „super-modularen“ Klassenkonzeption gesellt sich in der OOP ein Vererbungsverfahren, das beste Voraussetzungen für die Erweiterung von Software-Systemen bei rationeller **Wiederverwendung** der bisherigen Code-Basis schafft: Bei der Definition einer *abgeleiteten Klasse* können alle Merkmale und Handlungskompetenzen (Methoden, Eigenschaften) der *Basisklasse* übernommen werden. Es ist also leicht möglich, ein Software-System um neue Klassen mit speziellen Leistungen zu erweitern. Durch die 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.

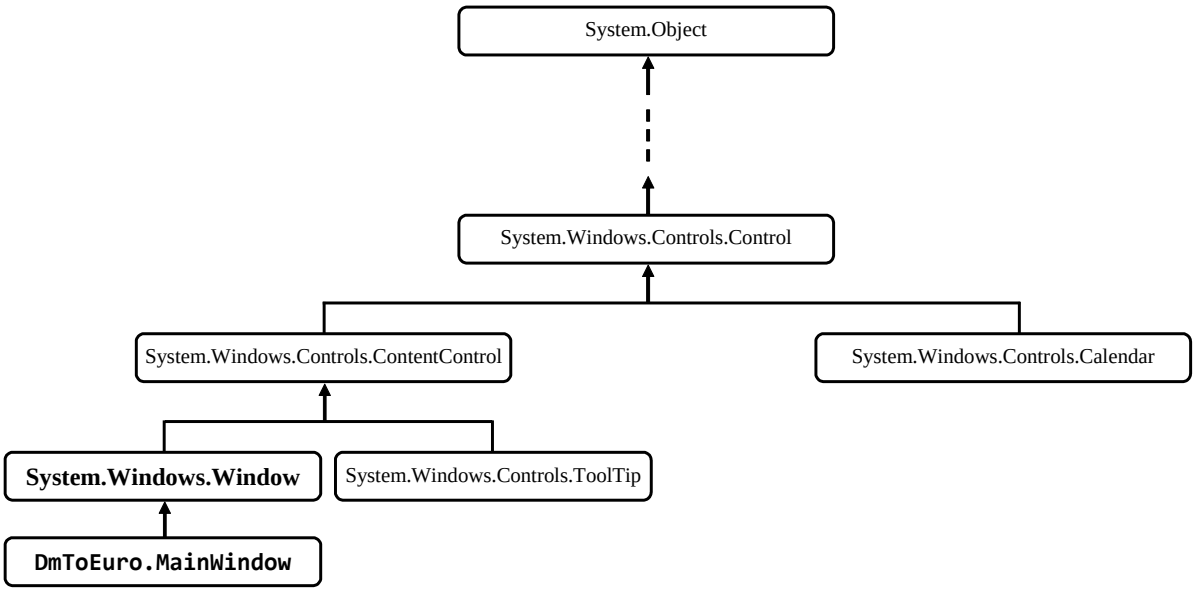
Bei dem im Abschnitt 3.3.7 erstellten und später im Rahmen einer Übungsaufgabe (siehe Abschnitt 4.5.11) weiterentwickelten Währungskonverter mit grafischer Benutzerschnittstelle



haben wir mit Hilfe der Entwicklungsumgebung die Klasse `MainWindow` definiert als Ableitung aus der Klasse **Window**:[1]

```
public partial class MainWindow : Window {
   public MainWindow() {
      InitializeComponent();
   }

   private void Button_Click(object sender, RoutedEventArgs e) {
      . . .
   }

   private void Window_Loaded(object sender, RoutedEventArgs e) {
      . . .
   }
}
```

Die Basisklasse **Window** ist selbst Bestandteil eines komplexen Stammbaums, von dem anschließend nur ein kleiner Ausschnitt zu sehen ist:



Im .NET - Typsystem wird das Vererbungsprinzip sogar auf die Spitze getrieben: Alle Klassen (und auch die Strukturen, siehe Abschnitt 6.1) 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) bezeichnet.

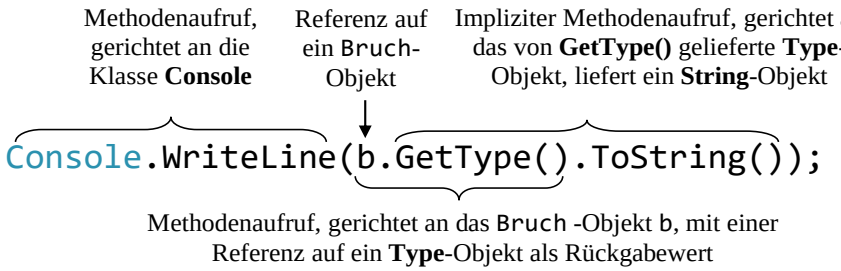
---

[1] Die Klasse `MainWindow` ist als **partial** gekennzeichnet, weil ihre Implementation ausnahmsweise auf zwei Dateien verteilt ist, um die Kooperation zwischen der Entwicklungsumgebung und dem Programmierer zu erleichtern:
- Die erste Quellcodedatei ist für den Programmierer zugänglich.
- Den restlichen Quellcode verwaltet die Entwicklungsumgebung in einer versteckten Datei.

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 Beispiel wird ein `Bruch`-Objekt (vgl. Abschnitt 1) um die entsprechende Auskunft gebeten:

Quellcode	Ausgabe
`Bruch b = new Bruch();` `Console.WriteLine(b.GetType().ToString());`	`Bruch`

Von **GetType()** erhält man als Rückgabewert eine Referenz auf ein Objekt der Klasse **Type**. Über diese Referenz wird das **Type**-Objekt gebeten, die **ToString()** – Methode auszuführen, wobei es den Wert seiner Eigenschaft **Name** mitteilt. Diese Zeichenfolge mit dem Typnamen (ein Objekt der Klasse **String**[1]) bildet schließlich den Parameter des **WriteLine()** - Aufrufs und landet auf der Konsole. In Ihrer Lernphase ist es angemessen, die komplexe Anweisung unter Beteiligung von vier Klassen (**Console**, `Bruch`, **Type**, **String**), drei Methoden (**WriteLine()**, **GetType()**) und **ToString()**), einer expliziten Referenzvariablen (`b`) und einer impliziten Referenz (**GetType()** - Rückgabewert) genau zu erläutern:[2]



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, dann kommen indirekt erworbene Erbstücke hinzu. Die als Beispiel betrachtete Klasse `Bruch` stammt direkt von der Klasse **Object** ab, und ihre Instanzen beherrschen dank Vererbung die Methode **GetType()**, obwohl in der `Bruch`-Klassendefinition davon nichts zu sehen ist.

5.1.1.3 Polymorphie

Obwohl in unseren bisherigen Beispielen die Polymorphie noch nicht zum Einsatz kam, soll doch versucht werden, die Kernidee hinter diesem Begriff schon jetzt zu vermitteln. In diesem Abschnitt sind einige Vorgriffe auf das Kapitel 7 erforderlich. Wer sich jetzt noch nicht für den Begriff der Polymorphie interessiert, kann den Abschnitt ohne Risiko für den weiteren Lernverlauf auslassen.

Beim Klassendesign ist generell das **Open-Closed - Prinzip** beachtenswert:[3]

---

[1] Eine Besonderheit der Klasse **String** ist der vom Compiler unterstützte Aliasname **string** (mit kleinem Anfangsbuchstaben).

[2] Der explizite **ToString()** - Aufruf ist im Beispiel eigentlich nicht erforderlich, weil die Methode **WriteLine()** mit Objekten (z. B. aus der Klasse **Type**) umzugehen weiß und deren garantiert vorhandene, weil bereits in der Urahnklasse **Object** definierte, Methode **ToString()** aufruft, um sich ein ausgabefähiges **String**-Objekt zu besorgen. Folglich empfiehlt sich im Beispiel der folgende **WriteLine()** – Aufruf:
`Console.WriteLine(b.GetType());`

[3] 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

- 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. In ungünstigen Fällen zieht eine Änderung am Quellcode weitere nach sich, sodass eine Kaskade von Anpassungen (eventuell unter Beteiligung von anderen Klassen) resultiert. Dadurch verursacht die Anpassung einer Klasse an neue Aufgaben hohe Kosten und das Risiko von Fehlern.

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.

In C# 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 Verhalten an spezielle Einsatzbereiche anzupassen. Ergeht ein Methodenaufruf an Objekte aus verschiedenen abgeleiteten Klassen, die 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 werden und denselben Methodenaufruf erhalten, agieren sie unterschiedlich. Welche Methode tatsächlich ausgeführt wird, entscheidet sich erst zur Laufzeit (späte Bindung). Genau in dieser Situation spricht man von *Polymorphie*, und diese Software-Technik leistet einen wichtigen Beitrag zur Realisation des Open-Closed - Prinzips.

Wird z. B. in einer Klasse zur Verwaltung von geometrischen Objekten eine Referenzvariable vom allgemeinen Typ `Figur` deklariert und beim Aufruf der Methode `MeldeInhalt()` verwendet, dann 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, wird hier die Methode `MeldeInhalt()` zwar deklariert, aber nicht implementiert, sodass eine sogenannte *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 Polymorphie ist eine lose Kopplung von Klassen möglich, und die Wiederverwendbarkeit von vorhandenem Code wird verbessert. Um die 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 darüberhinausgehende Einschränkung enthält.

Dank Vererbung und Polymorphie kann objektorientierte Software anpassungs- und erweiterungsfähig bei weitgehend fixiertem Bestands-Code, also unter Beachtung des Open-Closed - Prinzips, gestaltet werden.

---

[1] Neben den abstrakten Klassen, die mindestens *eine* abstrakte Methode (Definitionskopf ohne Implementation) enthalten, spielen bei der Polymorphie auch die sogenannten *Schnittstellen* eine wichtige Rolle als Datentypen für ein veränderungsoffenes Design. Eine Schnittstelle kann näherungsweise als Klasse mit *ausschließlich* abstrakten Methoden charakterisiert werden. Abstrakte Klassen und Schnittstellen (Interfaces) werden im Abschnitt 7.13 bzw. im Kapitel 9 ausführlich behandelt.

5.1.1.4 Realitätsnahe Modellierung

Klassen sind nicht nur ideale Bausteine für die rationelle Konstruktion von Software-Systemen, sondern sie erlauben auch eine gute Modellierung des Anwendungsbereichs. In der zentralen Projektphase der objektorientierten Analyse und Modellierung sprechen Software-Entwickler und Auftraggeber dieselbe Sprache, sodass 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 aus der Welt des Computers repräsentieren (z. B. Bildschirmfenster, Ereignisse).

5.1.2 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 in 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, dann muss jeweils nur ein Aufruf mit dem Namen des Unterprogramms und passenden Parameterwerten eingefügt werden. Durch diese Strukturierung ergeben sich kompakte und übersichtliche Programme, die mit erträglichem Aufwand erstellt, analysiert, korrigiert und erweitert werden können. Praktisch alle älteren Programmiersprachen unterstützen solche Unterprogramme (Prozeduren, Funktionen, Subroutinen), 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. Über das Schlüsselwort **struct** der Programmiersprache C oder das analoge Schlüsselwort **record** der Programmiersprache Pascal lassen sich problemadäquate Datentypen mit mehreren Bestandteilen definieren, die jeweils einen beliebigen, bereits bekannten Typ haben dürfen. So eignet sich etwa für ein Programm zur Adressenverwaltung ein neu definierter Datentyp mit Variablen 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, die oft auch als *prozedurale Programmierung* bezeichnet wird, bietet die OOP u. a. die folgenden Vorteile:

- 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 & Heinisch 2016), was bei Fehlern zu einer aufwändigen Suche entlang der Verarbeitungskette führen kann.
- Gute Voraussetzungen für die Teamarbeit
  Durch die optimierte Modularisierung wird die (vor allem in großen Projekten wichtige) Kooperation von Entwicklungsteams erleichtert.
- Rationelle (Weiter-)Entwicklung von Software nach dem Open-Closed - Prinzip durch Vererbung und Polymorphie
- Bessere Abbildung des Anwendungsbereichs
  Das erleichtert die Kommunikation zwischen dem Auftraggeber bzw. Anwender einerseits und dem Software-Architekten bzw. -Entwickler andererseits.
- Mehr Komfort für Bibliotheksbenutzer
  Jede rationelle Software-Produktion greift in hohem Maß auf Bibliotheken mit bereits vorhandenen Lösungen zurück. Dabei sind die Klassenbibliotheken der OOP einfacher zu verwenden als klassische Unterprogrammbibliotheken.
- Erleichterte Wiederverwendung
  Die komfortable Nutzung von Lösungsbibliotheken sowie die rationelle Weiterentwicklung von Software durch Vererbung und Polymorphie führen zu einer erleichterten Wiederverwendung von vorhandener Software.

Dass objektorientierte Programmiersprachen im Vergleich zu ihren strukturierten Vorgängern etwas mehr Speicherplatz und CPU-Leistung verbrauchen, spielt schon lange keine Rolle mehr.

Obwohl die objektorientierte Programmierung seit ca. 1980 als Paradigma der Software-Entwicklung dominiert, ist die strukturierte Programmierung weiterhin lebendig in darauf beschränkten Programmiersprachen (z. B. C, Fortran) sowie in hybriden Sprachen (z. B. Python). Exakte Daten zur Verbreitung der Programmierparadigmen sind selten. Dyer & Chauhan (2022) haben die dominanten Paradigmen in Python-Code untersucht und bei einer maschinellen Analyse von über 100.000 GitHub-Projekten gefunden, dass ca. 45.000 Projekte eindeutig dem objektorientierten und 33.000 Projekte dem prozeduralen Paradigma zuzuordnen waren. Diese Ergebnisse können nicht über die (sehr populäre) hybride Programmiersprache Python hinaus generalisiert werden, belegen aber doch die Lebendigkeit der strukturierten bzw. prozeduralen Programmierung.

5.1.3 Auf-`Bruch` zu echter Klasse

In C# - Programmen sind auf „globaler Ebene“ (außerhalb von Typdefinitionen) weder Variablendeklarationen noch Anweisungen erlaubt. Vor C# 9 musste im Quellcode jedes Programms eine Startklasse samt **Main()** – Methode definiert werden, wobei in einfachen Beispielprogrammen in der Regel weitgehend identische Zeilen monoton wiederholt wurden. Der Zweck eines solchen Beispielprogramms artikulierte sich erst in den lokalen Variablen und Anweisungen der **Main()** – Methode. Seit C# 9 kann mit Hilfe der sogenannten *Anweisungen auf oberster Ebene* auf solche pseudo-objektorientierte Pflichtübungen verzichten und in Konsolenanwendungen dem Compiler die Definition der Startklasse samt **Main()** – Methode überlassen. Von dieser Möglichkeit haben wir im Kapitel 4 in den Beispielprogrammen zum Erlernen elementarer Sprachelemente reichlich Gebrauch gemacht. Die im Kapitel 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. Die Klasse `Bruch` eignet sich z. B. für Programme, die Schüler beim Erlernen der Bruchrechnung unterstützen sollen. Eine objektorientierte Analyse der Problemstellung hat ergeben, dass in elementaren Bruchrechnungsprogrammen

lediglich eine Klasse zur Repräsentation von Brüchen benötigt wird. Später sind weitere Klassen zu ergänzen (z. B. Aufgabe, Übungsaufgabe, Testaufgabe, Schüler, Lernepisode, Testepisode, Fehler).

5.1.3.1 Verbesserte Definition der Klasse `Bruch`

Wir nehmen bei der `Bruch`-Klassendefinition im Vergleich zur Variante im Kapitel 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 zum Feld `etikett` eine Eigenschaft namens `Etikett` (mit großem Anfangsbuchstaben) für das Lesen und das (kontrollierte) Verändern des Felds zur Verfügung.
- Wir erlauben uns einen erneuten Vorgriff auf die im Kapitel 13 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**. Im **catch**-Block zur Behandlung von Ausnahmefehlern befindet sich eine **return**-Anweisung, die mit dem Rückgabewert **false** eine gescheiterte Ausführung der Methode signalisiert. Eine ungestörte Ausführung der Methode endet mit einer **return**-Anweisung, die dem Aufrufer den Wert **true** liefert.
- 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 im Abschnitt 4.7.4).

Im folgenden Quellcode der erweiterten bzw. renovierten Klasse `Bruch` sind die unveränderten Methoden bzw. Eigenschaften gekürzt wiedergegeben:[1]

```
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 {
      . . .
   }
```

[1] Ein Visual Studio - Projekt mit der aktuellen Entwicklungsstufe der Klasse `Bruch` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Bruch\b0

```
    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 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("Zähler: ");
          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 = "  ";
       string glz;
       for (int i = 1; i <= etikett.Length; i++)
          luecke += " ";
       if (etikett.Length == 0)
          glz = "";
       else {
          glz = " = ";
          luecke += "    ";
       }
       Console.WriteLine($" {luecke}{zaehler}\n {etikett}{glz}-----\n {luecke}{nenner}\n");
    }
 }
```

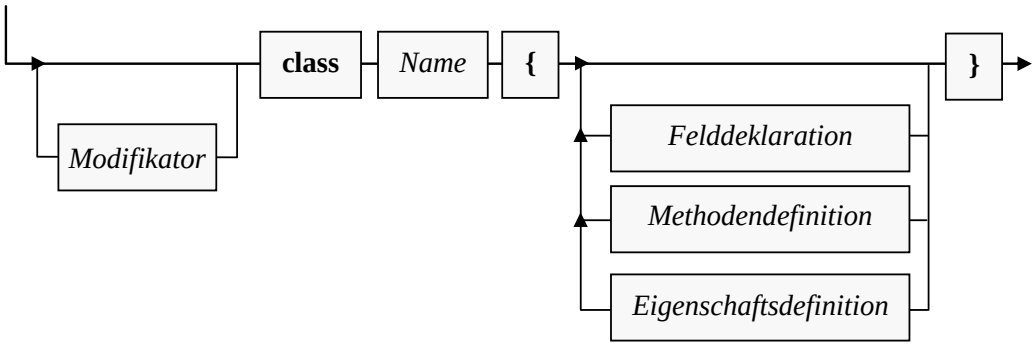
Im Unterschied zur Präsentation im Kapitel 1 wird die Definition der Klasse `Bruch` anschließend gründlich erläutert. Dabei machen die im Abschnitt 5.2 behandelten Instanzvariablen (Felder)

relativ wenig Mühe, weil wir viele Details schon von den *lokalen* Variablen her kennen. Bei den Methoden gibt es mehr Neues zu lernen, sodass wir uns im Abschnitt 5.3 auf elementare Themen beschränken und später noch wichtige Ergänzungen vornehmen.

5.1.3.2 Syntaxdiagramm und Details zum Kopf der Klassendefinition

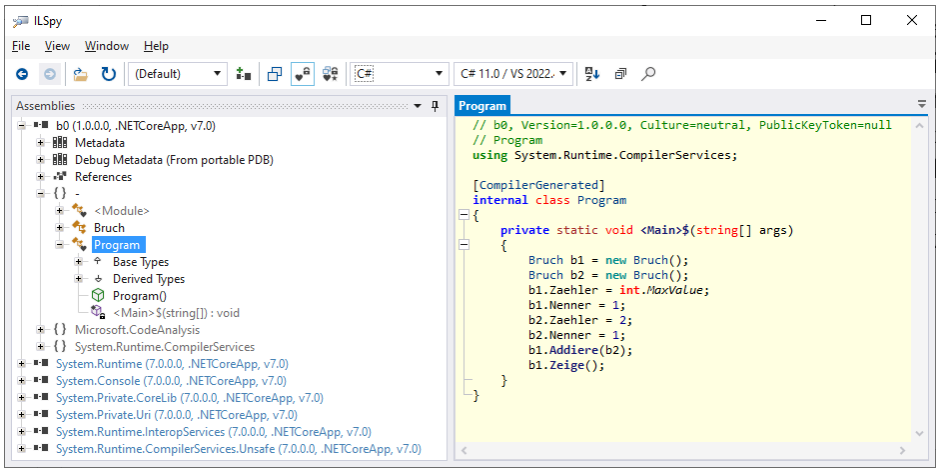
Wir arbeiten weiterhin mit dem aus dem Abschnitt 4.1.3.1 bekannten Syntaxdiagramm zur Klassendefinition, das aus didaktischen Gründen einige Vereinfachungen enthält:

Klassendefinition



Am Ende der Klassendefinition (hinter der schließenden geschweiften Klammer des Rumpfs) ist ein Semikolon erlaubt, aber nicht erforderlich.

Die bislang im Manuskript vorgestellten Modifikatoren zu einer Klasse beschäftigen sich mit der Verfügbarkeit für andere Klassen, wobei die Schlüsselwörter **public** und **internal** (= Voreinstellung) erlaubt sind (vgl. Abschnitt 5.12). Damit die Klasse `Bruch` von anderen Klassen aus beliebigen Assemblies genutzt werden kann, hat sie den Modifikator **public** erhalten. Bei der Startklasse zu einem Bruchrechnungsprogramm kommt eine Nutzung durch andere Klassen nicht in Frage. Wenn wir eine Startklasse explizit definieren, dann verzichten wir auf den (zum Starten durch die CLR nicht erforderlichen) Zugriffsmodifikator **public**. Damit ist die Voreinstellung **internal** aktiv (Verfügbarkeit für andere Klassen im selben Assembly). Bei einer vom Compiler automatisch erstellten Startklasse ist der Modifikator **internal** gesetzt, wie eine Inspektion mit dem Programm ILSpy zeigt (vgl. Abschnitt 3.4.1):



Klassennamen beginnen einer allgemein akzeptierten C# - Konvention folgend mit einem Großbuchstaben. Besteht ein Name aus mehreren Wörtern (z. B. `ArithmeticException`), dann schreibt man der besseren Lesbarkeit wegen die Anfangsbuchstaben aller Wörter groß (Pascal Casing, siehe Abschnitt 4.1.6).

Zur Dateiverwaltung wird vorgeschlagen:

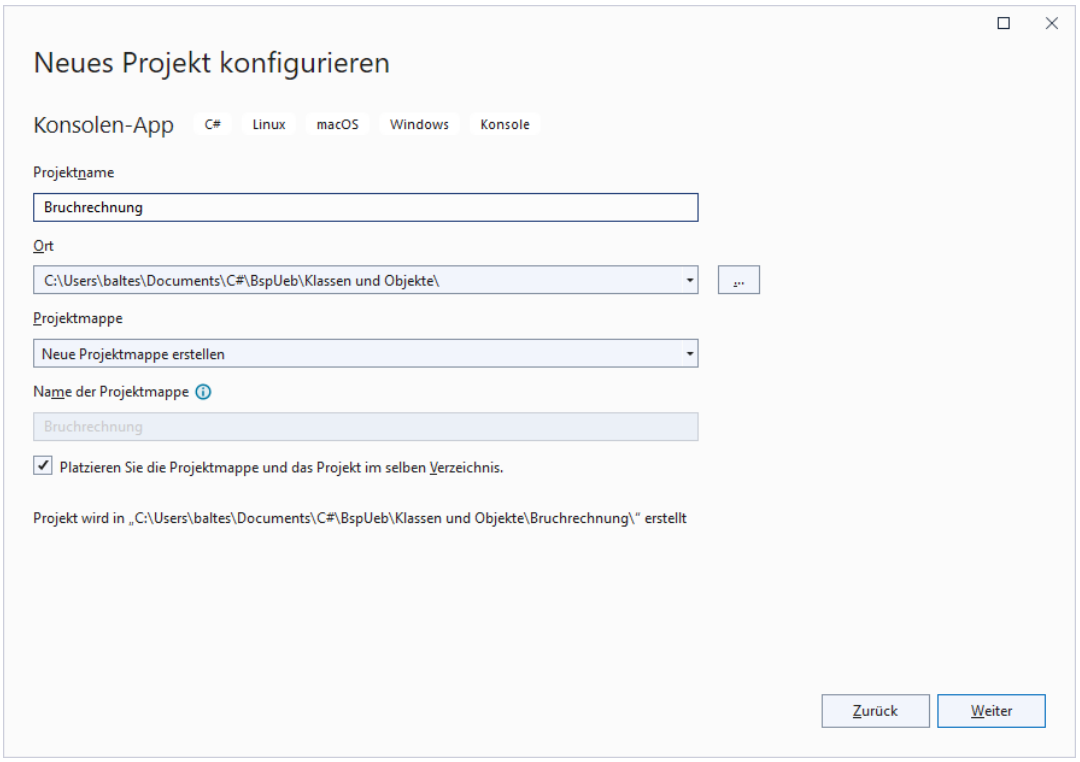
- Die Klassendefinitionen sollten in der Regel in einer eigenen Datei gespeichert werden. Seit C# 11 ermöglicht es allerdings der für nicht-geschachtelte Typen (siehe Abschnitt 5.10) erlaubte Modifikator **file**, andere Typen *in derselben Quellcodedatei* privilegiert zu berechtigen. Damit besteht gelegentlich ein Grund, mehrere Typen (z. B. Klassen) in eine Quellcodedatei aufzunehmen.
- Den Namen dieser Datei sollte man aus dem Klassennamen durch Anhängen der Erweiterung **.cs** bilden.

5.1.3.3 Bruchrechnungsprojekt im Visual Studio

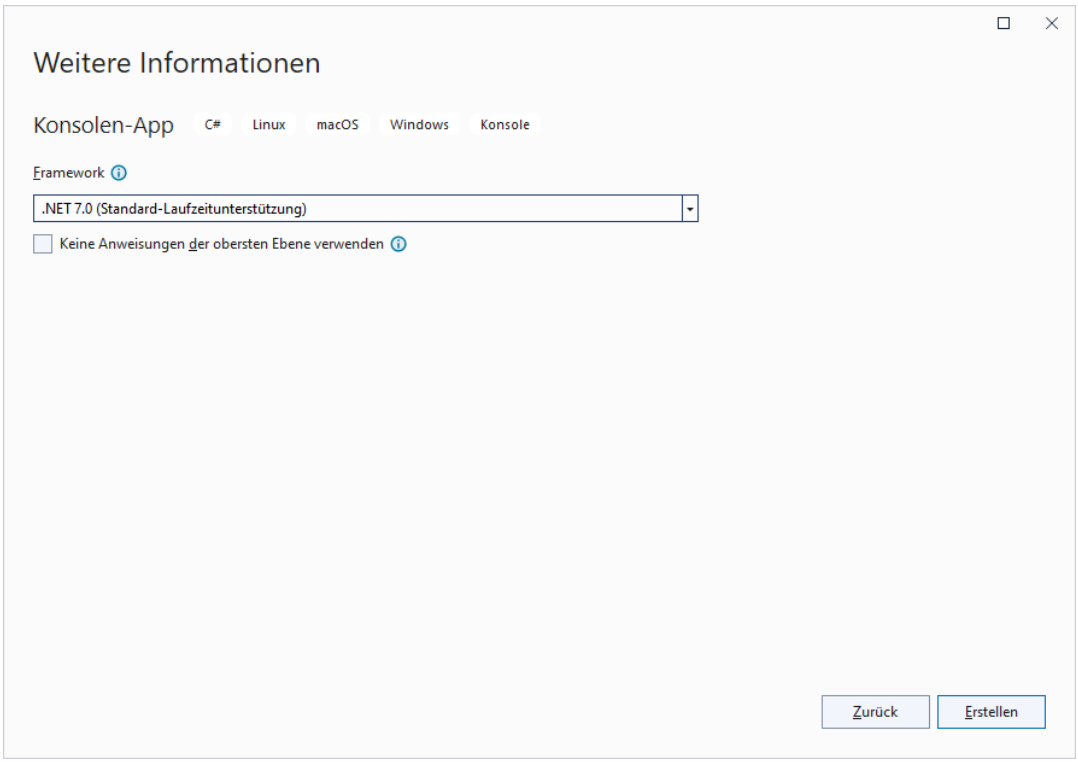
Für die Übungsbeispiele im Kapitel 5 legen wir im Visual Studio nach dem Menübefehl

Datei > Neu > Projekt

ein Konsolenprojekt an, z. B.:



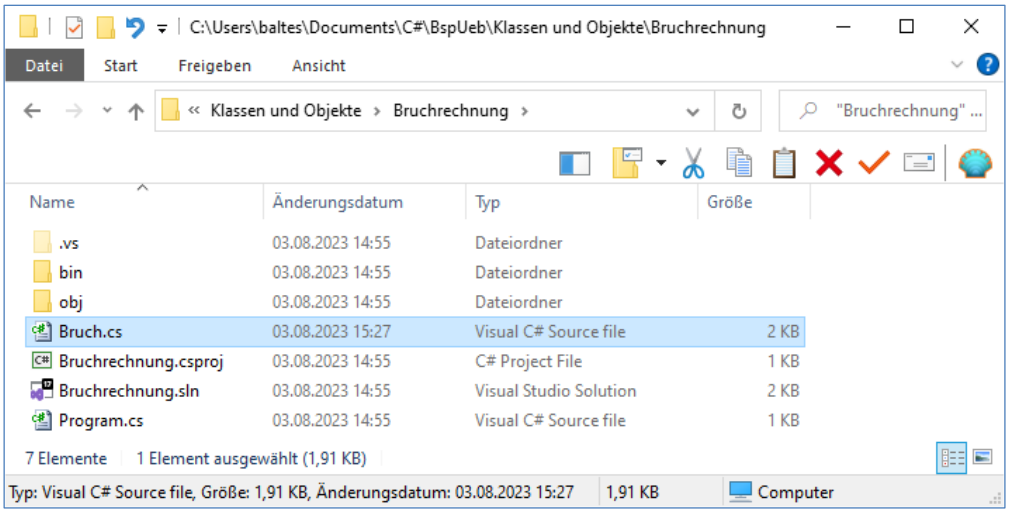
Das Kontrollkästchen zu den **Anweisungen auf oberster Ebene** hat keine nennenswerten Konsequenzen:



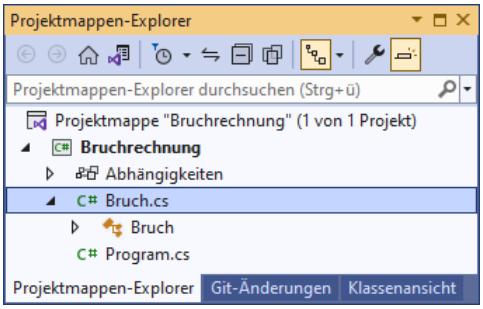
Wir kopieren die Datei

...\BspUeb\Klassen und Objekte\Bruch\b0\Bruch.cs

in den Projektordner, z. B.:



Daraufhin erscheint die Klasse `Bruch` im **Projektmappen-Explorer**:



Bei den diversen Beispielen in den folgenden Abschnitten werden wir entweder mit den Anweisungen auf oberster Ebene arbeiten, also auf die explizite Definition einer Startklasse verzichten (siehe Abschnitt 4.1.2), oder eine Klasse mit dem Namen `Bruchrechnung` und jeweils angepasster

Main() – Implementation verwenden. Im folgenden Programm zum Kürzen von Brüchen wird ein Objekt aus der Klasse `Bruch` erzeugt und mit Aufträgen versorgt.

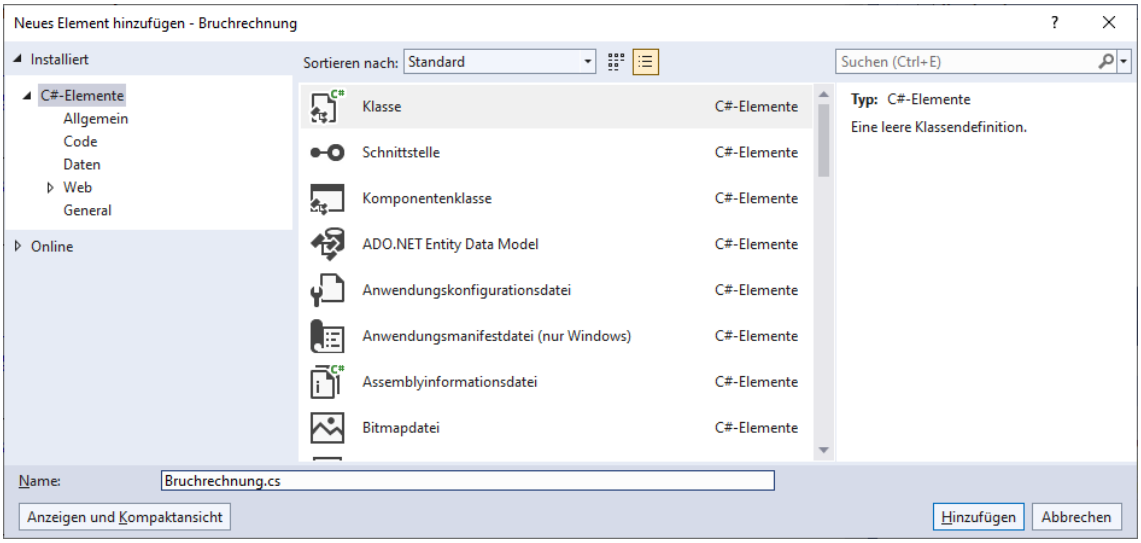
Quellcode	Eingabe (**fett**) und Ausgabe

```
using System;
class Bruchrechnung {
   static void Main() {
      Bruch b = new Bruch();
      b.Frage();
      b.Kuerze();
      b.Etikett = "b";
      b.Zeige();
   }
}
```

```
Zähler: 9
Nenner: 27
        1
 b = -----
        3
```

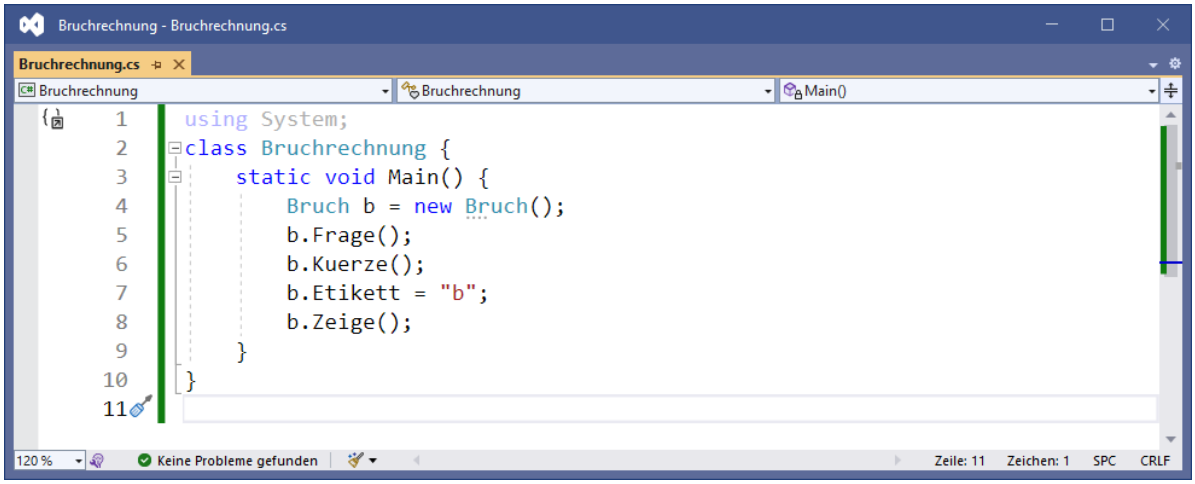
Um eine Startklasse namens `Bruchrechnung` in das Projekt aufzunehmen, kann man …

- aus dem Kontextmenü zum Projekt-Knoten im **Projektmappen-Explorer** das Item
  Hinzufügen > Neues Element
  wählen und mit dem folgenden Dialog eine **Klasse** anlegen:



- die vorhandene Datei **Program.cs** ändern und via **Projektmappen-Explorer** umbenennen in **Bruchrechnung.cs**.

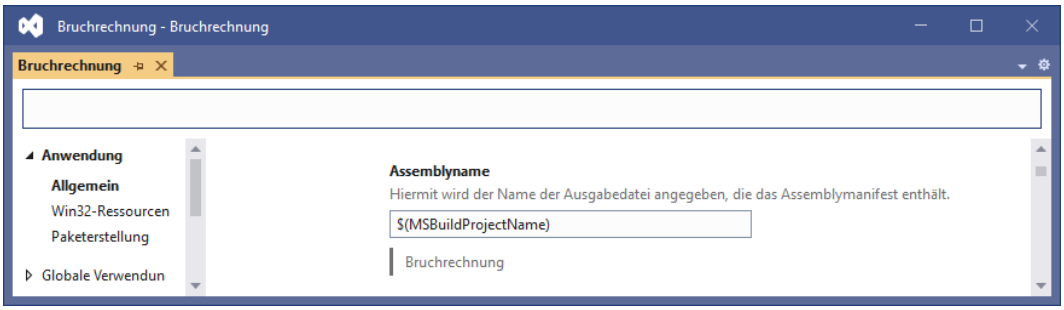
Wir wählen den zweiten Weg und verwenden den Quellcode aus dem letzten Beispiel:



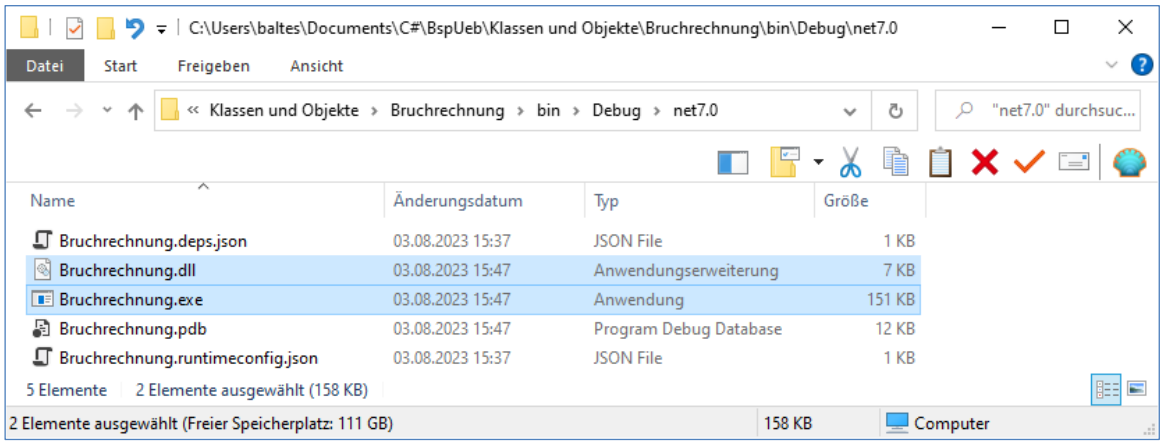
Lässt man das Programm erstellen (z. B. über die Tastenkombination **Strg+F5**), dann werden die Quellcodedateien **Bruch.cs** und **Bruchrechnung.cs** übersetzt, und es entsteht ein ausführbares Assembly, dessen Namen man im Visual Studio nach dem Menübefehl

Projekt > Eigenschaften

einstellen kann, wenn die vom Assistenten für neue Projekte vergebene Voreinstellung (= Projektname) nicht gefällt, z. B.:



Im Beispiel entstehen das ausführbare Assembly **Bruchrechnung.dll** und der Windows-Starthelfer **Bruchrechnung.exe**. (vgl. Abschnitte 3.1.5 und 3.3.7.5):



Wir nutzen das neue Projekt, um eine Schwachstelle der `Bruch`-Klassendefinition zu untersuchen. Die Instanzvariablen `zaehler` und `nenner` haben bei der Renovierung den Datentyp **int** behalten und sind daher nach wie vor mit einem potenziellen Überlaufproblem belastet, das im folgenden Programm demonstriert wird:

Quellcode	Ausgabe

`Bruch b1 = new Bruch(), b2 = new Bruch();`
`b1.Zaehler = 2147483647; b1.Nenner = 1;`
`b2.Zaehler = 2; b2.Nenner = 1;`
`b1.Addiere(b2);`
`b1.Zeige();`
	`-2147483647`
	`-----`
	`1`

Wie das Problem zu beheben ist, wurde im Abschnitt 4.6.1 beschrieben.

5.2 Instanzvariablen (Felder)

Die Instanzvariablen (bzw. -felder) einer Klasse besitzen viele Gemeinsamkeiten mit den *lokalen* Variablen, die wir im Kapitel 4 über elementare Sprachelemente ausführlich studiert 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.

5.2.1 Verfügbarkeit im Quellcode, Lebensdauer und Ablage im Hauptspeicher

Von den lokalen Variablen einer Methode unterscheiden sich die Instanzvariablen (Felder) einer Klasse vor allem hinsichtlich der *Zuordnung* (vgl. Abschnitt 4.3.3):

- lokale Variablen gehören zu einer *Methode* (oder *Eigenschaft*)
- Instanzvariablen gehören zu einem *Objekt*

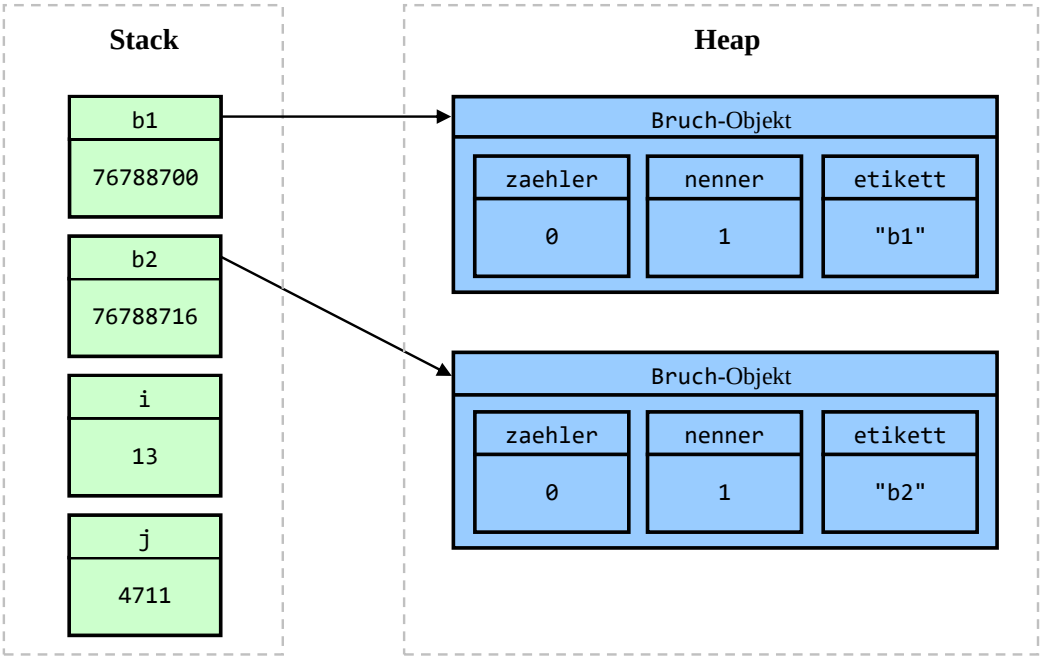
Daraus ergeben sich gravierende Unterschiede in Bezug auf die Verfügbarkeit im Quellcode, die Lebensdauer und die Ablage im Hauptspeicher:

	lokale Variable	**Instanzvariable**
Verfügbarkeit im Quellcode	Die Verfügbarkeit einer lokalen Variablen ist technisch restringiert durch ihren Sichtbarkeitsbereich (siehe Abschnitt 4.3.8). Nach der Deklarationsanweisung bleibt sie ansprechbar bis zur schließenden Klammer des Blocks, in dem sie deklariert worden ist. Ein eingeschachtelter Block gehört zum Sichtbarkeitsbereich des umgebenden Blocks.	Die Instanzvariablen eines Objekts sind in einer Methode verfügbar, wenn ... • der Zugriff erlaubt ist (siehe Abschnitt 5.12 zu den Schutzstufen) • eine Referenz zum Objekt vorhanden ist Instanzvariablen werden in klasseneigenen Instanzmethoden durch gleichnamige lokale Variablen überlagert, können in dieser Situation jedoch über das vorgeschaltete Schlüsselwort **this** angesprochen werden (siehe Abschnitt 5.2.4).
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 im Programm vorhanden ist.
Ablage im Speicher	Sie wird auf dem sogenannten **Stack** (deutsch: *Stapel*) abgelegt. Dieses Segment des programmeigenen Speichers dient zur Durchführung 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]



5.2.2 Deklaration mit Modifikatoren für den Zugriffsschutz und für andere Zwecke

Während lokale Variablen in einer Methode (oder Eigenschaft) deklariert werden, erscheinen die Deklarationen der Instanzvariablen in der Klassendefinition *außerhalb* jeder Methoden- oder Eigenschaftsdefinition, z. B. in der Definition der Klasse `Bruch`:

```
using System;
public class Bruch {
   int zaehler, nenner = 1;
   string etikett = "";

   public int Zaehler {
      . . .
   }
   . . .
   public void Kuerze() {
      . . .
   }
   . . .
```

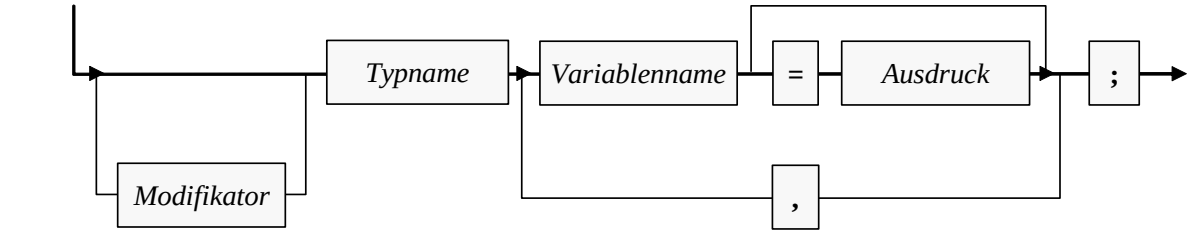
[1] Hier wird aus didaktischen Gründen ein wenig gemogelt: Die beiden Etiketten sind selbst Objekte und liegen „neben“ den `Bruch`-Objekten auf dem Heap. In jedem `Bruch`-Objekt befindet sich eine Referenz-Instanzvariable namens `etikett`, die auf das zugehörige **String**-Objekt zeigt.

```
}
```

Man sollte die Instanzvariablen der Übersichtlichkeit halber am Anfang der Klassendefinition deklarieren, wenngleich der Compiler auch ein späteres Erscheinen akzeptiert.[1]

Für die Deklaration einer lokalen Variablen haben wir **const** als einzigen Modifikator kennengelernt und in einem speziellen Syntaxdiagramm beschrieben (siehe Abschnitt 4.3.9). Dieser Modifikator ist auch bei Instanzvariablen erlaubt (siehe Abschnitt 5.2.5). Außerdem kommen hier weitere Modifikatoren in Frage, die z. B. zur Spezifikation der **Schutzstufe** (Zugriffsberechtigung für andere Klassen) dienen. Insgesamt ist es sinnvoll, in das Syntaxdiagramm zur Deklaration von Instanzvariablen den allgemeinen Begriff des Modifikators aufzunehmen:[2]

Deklaration von Instanzvariablen



In C# besitzen alle Instanzvariablen per Voreinstellung die Schutzstufe **private**, sodass 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 in der Klassendefinition jeweils eine zugehörige Eigenschaft vorhanden.

Auf den ersten Blick scheint die Datenkapselung nur beim Nenner eines Bruchs relevant zu sein, doch auch bei den restlichen Instanzvariablen bringt sie potenziell Vorteile:

- Zugunsten einer übersichtlichen Bildschirmausgabe soll das Etikett auf 40 Zeichen beschränkt bleiben. Mit Hilfe der Eigenschaft `Etikett()` kann dies auf einfache Weise 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. Ein zwingendes Argument für die Eigenschaft `Zaehler` würde aus der Entscheidung resultieren, beim Zähler (und beim Nenner) eines `Bruch`-Objekts negative Werte zu verbieten (vgl. Abschnitt 1.2).

Trotz ihrer überzeugenden Vorteile soll die Datenkapselung nicht zum Dogma erhoben werden. Sie verliert an Bedeutung, wenn ...

---

[1] Anders als bei lokalen Variablen von Methoden hat der Deklarationsort bei Instanzvariablen keinen Einfluss auf die Verfügbarkeit. Zum Sichtbarkeitsbereich der lokalen Variablen existiert bei den Instanzvariablen keine Entsprechung.

[2] 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 aufwändige Darstellungstechnik zu verbieten.

- bei einem Feld Lese- und Schreibzugriffe uneingeschränkt erlaubt sein sollen, wenn es also insbesondere nicht erforderlich ist, die möglichen Werte zu restringieren.
- es nicht von Interesse ist, auf bestimmte Wertzuweisungen zu reagieren, um z. B. bestimmte Objekteigenschaften (man sagt auch: *Invarianten*) sicherzustellen.

Um allen Klassen den Direktzugriff auf ein Feld zu erlauben, wird in seiner Deklaration der Modifikator **public** angegeben, z. B.:

```
public int Zaehler;
```

Im Abschnitt 5.12 finden Sie eine Tabelle mit allen Schutzstufen und zugehörigen Modifikatoren.

Bei Felddeklaration mit Initialisierung spricht man auch von einem *Feldinitialisierer*.

Für die Benennung von Instanzvariablen werden die folgenden Konventionen empfohlen (vgl. Mössenböck 2019, S. 14):

- Für Felder mit den Schutzstufen **private**, **protected**, **internal** oder **file** wird das Camel Casing verwendet (z. B. `currentSpeed`). Oft tritt eine *private* Instanzvariable (z. B. `nenner`), die einen Namen mit *kleinem* Anfangsbuchstaben besitzt, als Hintergrund zu einer *öffentlichen* Eigenschaft (z. B. `Nenner`) auf, die einen Namen mit *großem* Anfangsbuchstaben besitzt.
- Für die Felder mit der Schutzstufe **public** wird das Pascal Casing verwendet (z. B. `MinValue`).

Einige Programmierer verwenden einen Unterstrich als Präfix für die Namen von Feldern mit der Schutzstufe **private** (verfügbar in der eigenen Klasse) oder **internal** (verfügbar im eigenen Assembly) und sorgen so für eine gute Unterscheidbarkeit von lokalen Variablen und Formalparametern (siehe Abschnitt 5.3.1.3), z. B.:

```
private int _nenner;
```

Gelegentlich ist in derselben Situation auch das Präfix **m_** zu sehen, z. B.:

```
private int m_nenner; //das "m" steht für "Member"
```

5.2.3 Automatische Initialisierung auf den Voreinstellungswert

Während bei lokalen Variablen der Programmierer für die Initialisierung verantwortlich ist, erhalten die Instanzvariablen eines neuen Objekts 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 (Zeichen mit der Nummer 0 im Unicode)
bool	**false**
Referenztyp	**null**

In der Klasse `Bruch` wird nur die automatische `zaehler`-Initialisierung unverändert übernommen,

```
int zaehler,
    nenner = 1;
string etikett = "";
```

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* elementarer 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 Initialisierungswert **null**, zeigt also auf nichts. Weil der `etikett`-Wert **null** z. B. beim Aufruf der `Bruch`-Methode `Zeige()` einen Laufzeitfehler (**NullReferenceException**) zur Folge hätte, wird ein **String**-Objekt mit einer leeren Zeichenfolge erstellt und zur `etikett`-Initialisierung verwendet.[1] Das Erzeugen des **String**-Objekts erfolgt *implizit* (ohne **new**-Operator, siehe Abschnitt 6.3.1), 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 zuzugreifen, z. B.:[2]

```
class A {
    int eins = 1;
    int zwei = eins + 1;
```

(Feld) int A.eins

CS0236: Ein Feldinitialisierer kann nicht auf das nicht statische Feld bzw. die nicht statische Methode oder Eigenschaft "A.eins" verweisen.

Im Rahmen eines Konstruktors (siehe Abschnitt 5.4.3) ist eine solche Initialisierung hingegen möglich.

5.2.4 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()` geschieht:

```
Console.WriteLine($" {luecke}{zaehler}\n {etikett}{glz}-----\n {luecke}{nenner}\n");
```

Im Beispiel zeigt sich syntaktisch kein Unterschied zwischen dem Zugriff auf die Instanzvariablen (`zaehler`, `nenner`, `etikett`) und dem Zugriff auf die lokalen Variablen (`luecke`, `glz`).

Gelegentlich kann es sinnvoll oder erforderlich sein, einem Instanzvariablennamen über das Schlüsselwort **this** (vgl. Abschnitt 5.4.4.3) eine Referenz auf das handelnde Objekt voranzustellen, wobei das Schlüsselwort und der Feldname durch den **Punktoperator** zu trennen sind:

---

[1] Seit C# 8 bietet der C# - Compiler eine optionale Unterstützung zur Vermeidung der **NullReferenceException** an, die aus didaktischen Gründen vorläufig ausgespart bleibt (siehe Abschnitt 15.1).

[2] Eine als **const** deklarierte Instanzvariable kann so verwendet werden. Dabei handelt es sich allerdings um ein implizit statisches Feld (siehe Abschnitt 5.2.5).

- Das kann optional der Klarheit halber geschehen, z. B.:

```
Console.WriteLine($" {luecke}{this.zaehler}\n {this.etikett}{glz}-----\n" +
                  $" {luecke}{this.nenner}\n");
```

- Instanzvariablen werden durch gleichnamige lokale Variablen oder Methodenparameter (siehe Abschnitt 5.3) überlagert, können jedoch in dieser (besser zu vermeidenden) Situation über das vorgeschaltete Schlüsselwort **this** weiter angesprochen werden (siehe Abschnitt 5.4.4.3).

Beim Zugriff auf eine Instanzvariable eines *anderen* Objekts 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:

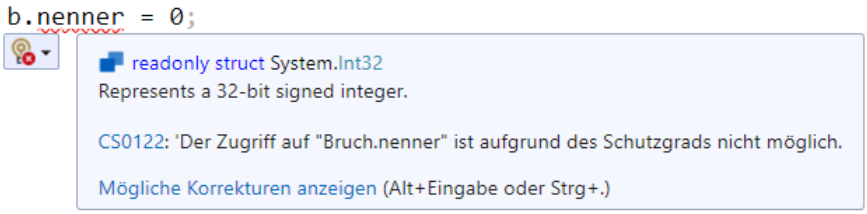
```
zaehler = zaehler * b.nenner + b.zaehler * nenner;
```

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.:



5.2.5 Wertfixierung zur Übersetzungszeit oder nach der Initialisierung per Konstruktor

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 4.3.9) sondern auch Felder einer Klasse als konstant deklariert werden. Der Wert wird zur Übersetzungszeit unveränderlich festgelegt, sodass eine neue Übersetzung erforderlich ist, wenn der Wert doch einmal geändert werden soll. Damit eignet sich ein mit dem Modifikator **const** dekoriertes Feld z. B. zur Speicherung der Kreiszahl π, aber nicht zur Speicherung des Mehrwertsteuersatzes.

Als Datentypen sind für Felder mit **const**-Deklaration ausschließlich die elementaren Datentypen und der Typ **String** erlaubt bzw. sinnvoll.[1]

Besonderheiten bei den konstanten Feldern einer Klasse:

[1] Neben **String** sind zwar auch andere Referenztypen erlaubt, doch muss in diesem Fall bei der obligatorischen Initialisierung das Referenzliteral **null** verwendet werden, sodass in der Regel kein nützliches Feld resultiert.

- Sie 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 5.2 über Instanzvariablen.
- Sie werden *nicht* automatisch (mit dem typspezifischen Nullwert) initialisiert. Bei der somit erforderlichen Deklaration mit expliziter Initialisierung ist ein Ausdruck zu verwenden, der schon *zur Übersetzungszeit* berechnet werden kann.
- Es ist keine Datenkapselung als Schutz gegen irreguläre Wertzuweisungen erforderlich. Wenn fremden Klassen der lesende Zugriff erlaubt sein soll, spricht nichts gegen den Zugriffsmodifikator **public**. Die folgende Deklaration stammt aus der BCL-Klasse **Math** im Namensraum **System**:

```
public const double E = 2.7182818284590452354;
```

Soll eine Instanzvariable *zur Laufzeit* in einem sogenannten Konstruktor (siehe Abschnitt 5.4.3) initialisiert und danach fixiert werden, verwendet man in der Deklaration den Modifikator **readonly**, z. B.:

Quellcode	Ausgabe

`ReadonlyFraction b = new ReadonlyFraction(1, 7);`
`Console.WriteLine(b.Denom);`
`// b.Denom = 13; // verboten`
 	
`class ReadonlyFraction {`
`  public readonly int Num, Denom;`
 	
`  public ReadonlyFraction(int n, int d) { // Konstruktor`
`  Num = n;`
`  Denom = d != 0 ? d : 1;`
`  }`
`}`
	`7`

Bei den Konstruktoren handelt es sich um spezielle Methoden, die den Namen ihres Typs tragen und keinen Rückgabetyp besitzen (auch nicht **void**).

Das Beispiel demonstriert, dass in einer Quellcodedatei auf die am Anfang stehenden Anweisungen auf oberster Ebene noch Typdefinitionen folgen dürfen.

Unterschiede zwischen der **const**- und der **readonly**-Deklaration von Variablen:

	const	**readonly**
Erlaubte bzw. sinnvolle 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

In vielen Büchern über die professionelle Software-Entwicklung wird empfohlen, Klassen nach Möglichkeit als unveränderlich (engl.: *immutable*) zu entwerfen, sodass ihre Objekte nach der Initialisierung nicht mehr geändert werden können. Bloch (2018, S. 82ff) nennt die folgenden Vorteile unveränderlicher Klassen:

- Wird bei der Kreation eines Objekts für einen gültigen Zustand gesorgt, ist die Gültigkeit während der gesamten Lebenszeit garantiert, was die Handhabung von Objekten sicher und einfach macht.
- Ein unveränderliches Objekt kann ohne Synchronisierungsaufwand von mehreren Threads genutzt werden (siehe Kapitel 17 von Baltes-Götz (2021)). Es wird also auf einfache Weise Thread-Sicherheit erzielt.
- Unveränderliche Objekte können an mehreren Stellen einer Anwendung wiederverwendet werden, statt jeweils ein neues Objekt zu erzeugen (z. B. ein **String**-Objekt mit dem Inhalt „`N.N.`“).

Unveränderliche Klassen in der BCL sind z. B.:

- **String**
- **DateTime**

Die Klasse `Bruch` folgt dem Trend hin zu unveränderlichen Objekten noch *nicht*, was vermutlich Programmiereinsteigern entgegenkommt. Sobald die sichere Verwendung der Klasse in einer Multithreading-Umgebung relevant wird, sollte eine unveränderliche Neukonzeption erwogen werden.

Von zentraler Bedeutung sind unveränderliche Klassen im *funktionalen Programmier-Paradigma*, das in C# z. B. durch die Erweiterung um die Lambda-Syntax unterstützt wird (siehe Abschnitte 5.8.1 und10.1.5.2). Dem funktionalen Programmierstil verpflichtete Methoden verändern nicht den Zustand von Objekten (z. B. die Koordinaten von Positionen), sondern produzieren nach Bedarf neue Objekte (z. B. neue Positionen). Man kann sich leicht vorstellen, dass die funktionale Programmierung nicht für alle Aufgabenstellungen eine angemessene Modellierung erlaubt. Sehr viele Objekte zu erstellen, verursacht einen hohen Zeitaufwand und bei großen Objekten auch einen hohen Speicherbedarf.

Traditionelle, veränderliche Klassen sind für viele Aufgaben weiterhin unverzichtbar.[1] Wir werden z. B. im Abschnitt 6.3 …

- einerseits die für unveränderliche Zeichenfolge optimierte Klasse **String** näher betrachten
- und andererseits als Alternative die Klasse **StringBuilder** kennenlernen, die für variable Zeichenfolgen konzipiert ist.

Es wird sich zeigen, dass die unveränderliche Klasse **String** für bestimmte Algorithmen aus Performanzgründen nicht geeignet ist.

5.3 Instanzmethoden

Durch eine Bauplan-Klassendefinition werden Objekte mit einer Anzahl von Verhaltenskompetenzen entworfen, die sich über Methodenaufrufe nutzen lassen. Objekte sind also Dienstleister, die eine Reihe von Nachrichten interpretieren und mit passendem Verhalten beantworten können.

Ihre Instanzvariablen sind bei konsequenter Datenkapselung für fremde 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 5.5).

---

[1] Klassen dienen auch zum Modellieren des Aufgabenbereichs mit realen Akteuren bzw. Objekten, die zum großen Teil einen variablen Zustand besitzen (z. B. Urlaubspläne von Mitarbeitern, Produkte während der Fertigung). Werden variable reale Objekte durch unveränderliche IT-Projekte modelliert, dann muss bei einer Änderung des realen Objekts das modellierende IT-Objekt verworfen und durch ein neues IT-Objekt ersetzt werden. Genau das tut die funktionale Programmierung, und wenn dieses Verfahren (z. B. wegen der erleichterten Multithreading-Programmierung) für einen Algorithmus geeignet ist, dann kann und sollte es auch in C# realisiert werden.

Beim Aufruf einer Methode werden oft durch sogenannte **Parameter** erforderliche Daten und/oder Anweisungen zur Steuerung der Arbeitsweise an die Methode übergeben, und von vielen Methoden wird dem Aufrufer ein **Rückgabewert** geliefert (z. B. mit einer 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, dann kann er es mit einem Aufruf der Methode `Addiere()` veranlassen, einen per Parameter benannten zweiten Bruch zum eigenen Wert zu addieren, wobei das Ergebnis auch noch gekürzt wird:

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

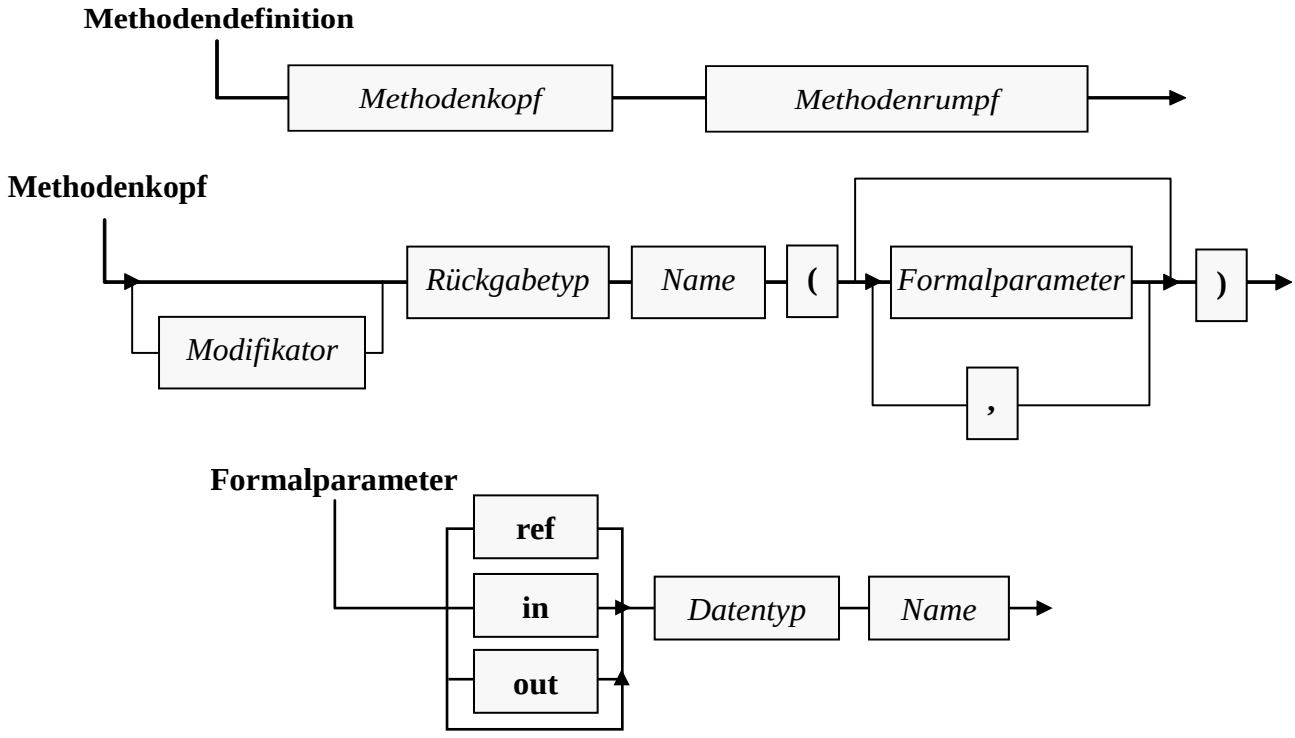
Weil diese Methode auch für fremde Klassen verfügbar sein soll, wird per Modifikator die Schutzstufe **public** gewählt.

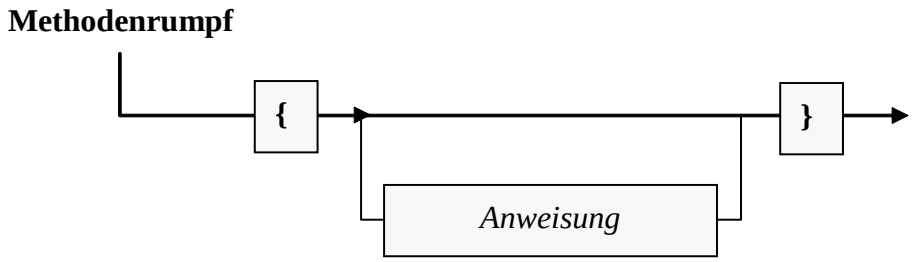
Da es vom Verlauf der Auftragserledigung nichts zu berichten gibt, liefert `Addiere()` keinen Rückgabewert. Folglich wird im Kopf der Methodendefinition der Rückgabetyp **void** angegeben.

Während jedes Objekt einer Klasse seine eigenen Instanzvariablen auf dem Heap besitzt, ist der IL-Code der Instanzmethoden nur *einmal* im Speicher vorhanden und wird von allen Objekten der Klasse verwendet.

5.3.1 Methodendefinition

Die folgende Serie von Syntaxdiagrammen zur Methodendefinition unterscheidet sich von der Variante im Abschnitt 4.1.3.2 durch eine genauere Erklärung der (im Abschnitt 5.3.1.3 behandelten) Formalparameter:





Bei der anschließenden Erläuterung der (mehr oder weniger) neuen Bestandteile dieser Syntaxdiagramme werden die Methodendefinition und der Methodenaufruf keinesfalls so sequentiell und getrennt dargestellt, wie es die Abschnittsüberschriften vermuten lassen. Schließlich ist die Bedeutung mancher Details der Methodendefinition am besten am Effekt auf den Methodenaufruf zu erkennen.

Während sich in C# bei Feldnamen die Groß-/Kleinschreibung des Anfangsbuchstabens nach einer weithin akzeptierten Konvention an der Schutzstufe orientiert (Camel Casing für Felder mit den Schutzstufen **private**, **protected**, **internal** oder **file** sowie Pascal Casing für öffentliche Felder, vgl. Abschnitt 5.2.2), starten die Namen von Methoden in der Regel unabhängig von der Schutzstufe mit einem Großbuchstaben (Pascal Casing).

5.3.1.1 Modifikatoren

Bei einer Methodendefinition kann per Modifikator die voreingestellte **Schutzstufe** verändert werden. In C# gilt für Methoden wie für Instanzvariablen:

- Voreingestellt ist die Schutzstufe **private**, sodass 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 zum API einer Klasse gehören sollen.[1] In unserer Beispielklasse `Bruch` haben *alle* Methoden den Zugriffsmodifikator **public**.

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**).

5.3.1.2 Rückgabewert und return-Anweisung

Für den Informationstransfer von einer Methode an ihren Aufrufer kann neben **ref**- und **out**-Parametern (siehe Abschnitt 5.3.1.3.2) 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 im Abschnitt 4.5.2 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 liefert.

Bei der Definition einer Methode muss festgelegt werden, von welchem Datentyp ihr Rückgabewert ist. Erfolgt *keine* Rückgabe, ist der Ersatztyp **void** anzugeben.

---

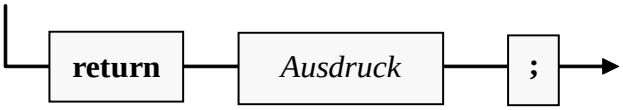
[1] Im Abschnitt 1.2 wurden *Schnittstelle* und *API* (*Application Programming Interface*) als synonyme Bezeichnungen für die öffentlich verfügbaren Methoden und Felder einer Klasse eingeführt. Allerdings wird im Kapitel 9 die Bezeichnung *Schnittstelle* (mit dem Alias *Interface*) zur Bezeichnung einer Sammlung von Methodendefinitionsköpfen (ohne Implementation) verwendet. Die beiden mit *Schnittstelle* bezeichneten Begriffe sind verwandt, aber keinesfalls deckungsgleich.

Als Beispiel betrachten wir die aktuelle Variante der `Bruch`-Methode `Frage()` (mit einem Vorgriff auf die Ausnahmebehandlung per **try-catch** – Anweisung, siehe Abschnitt 5.1.3.1), 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("Zähler: ");
      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 (nicht durch einen unbehandelten Ausnahmefehler abgebrochene) Ausführungspfad mit einer **return**-Anweisung endet, die einen Wert von kompatiblem 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 der Methode mit dem Rückgabewert **false**.

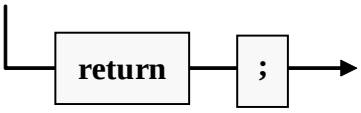
Beim bedenklichen Wunsch des Anwenders, den Nenner auf 0 zu setzen, überlässt es die Methode `Frage()` der Eigenschaft `Nenner`, angemessen darauf zu reagieren.

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 einem beendeten Programmeinsatz unter Windows in einem per **cmd.exe** gestarteten Konsolenfenster befindet sich der Rückgabewert in der Pseudo-Umgebungsvariablen **errorlevel**, die mit dem **echo**-Kommando angezeigt werden kann, z. B.:

```
>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;
```

5.3.1.3 Formalparameter

Parameter wurden bisher leicht vereinfachend als Daten und/oder Informationen beschrieben, die einer Methode beim Aufruf übergeben werden. Tatsächlich kennt C# verschiedene Parameterarten, um den Informationsaustausch zwischen einer rufenden und einer gerufenen Methode in *beide* Richtungen zu unterstützen.

Im Kopf der **Methodendefinition** werden über sogenannte **Formalparameter** Daten von bestimmtem Typ spezifiziert, die entweder den Ablauf der Methode beeinflussen, und/oder durch die Methode verändert werden, um Informationen an den Aufrufer zu übergeben. Beim späteren **Aufruf** der Methode sind korrespondierende **Aktualparameter** zu übergeben (siehe Abschnitt 5.3.2), 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:

- **Transfermodus**
  Ein *Wertparameter* dient dazu, Informationen in die Methode zu kopieren, und dabei wird *kein* Transfermodus angegeben (siehe Abschnitt 5.3.1.3.1). Bei den *Verweisparametern* wird durch die Schlüsselwörter **ref**, **in** und **out** festgelegt, in welche Richtung(en) Informationen übertragen werden können (siehe Abschnitt 5.3.1.3.2).
- **Datentyp**
  Es sind beliebige Typen erlaubt. Man muss den Datentyp eines Formalparameters auch dann explizit angeben, wenn er mit dem Typ des linken Nachbarn übereinstimmt.
- **Name**
  Nach den Empfehlungen aus dem Abschnitt 4.1.6 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, …
  - müssen sich die Parameternamen von den Namen der (anderen) lokalen Variablen unterscheiden,
  - 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 *unbenannten* Aktualparameter gemäß ihrer Reihenfolge den Formalparametern zugeordnet werden.

5.3.1.3.1 Wertparameter

Über einen Wertparameter werden Informationen in die Methode *kopiert*, um diese mit Daten zu versorgen oder ihre Arbeitsweise zu steuern. Als Beispiel betrachten wir die folgende Variante der `Bruch`-Methode `Addiere()`.[1] Das beauftragte Objekt soll den über **int**-Parameter (`zpar`, `npar`)

[1] Ein Visual Studio - Projekt mit der aktuellen Entwicklungsstufe der Klasse `Bruch` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Bruch\b1 Methoden-Überladung

übergebenen Bruch mit dem Hauptnennerverfahren zum eigenen Wert addieren und optional (**bool**-Parameter `autokurz`) das Ergebnis 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 beim Aufruf übergebenen Wert initialisiert worden ist.

Selbstverständlich sind auch methodeninterne Änderungen eines Wertparameters möglich, die *ohne* Effekt auf eine als Aktualparameter fungierende Variable der rufenden Methode bleiben. 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 Klasse `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, etwas umständlich wirkenden Konstruktion wird es vermieden, im aktuellen Abschnitt 5.3.1 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 5.7.3).

Als Wertaktualparameter sind nicht nur Variablen erlaubt, sondern beliebige (z. B. auch aus einem Literal bestehende) Ausdrücke mit einem Typ, der nötigenfalls erweiternd in den Typ des zugehörigen Formalparameters gewandelt werden kann.

5.3.1.3.2 Verweisparameter

C# bietet die Möglichkeit, beim Methodenaufruf das Kopieren eines Aktualparameterwerts zu ersetzen durch die Übergabe der Speicheradresse des Originals, wobei es sich um eine Variable handeln muss. Dieses Verfahren lohnt sich vor allem bei umfangreichen Strukturen, also bei den im Abschnitt 6.1 zu behandelnden klassenähnlichen Werttypen. Es wird dabei auf lokale Variablen von Methoden verwiesen, d. h. die Adressen befinden sich auf dem *Stack*.

Nach der Richtung des möglichen Informationstransfers sind drei Arten von Verweisparametern zu unterscheiden, die jeweils durch ein Schlüsselwort zu kennzeichnen sind:

- **ref**-Parameter
  Sie ermöglichen den Informationstransfer in beide Richtungen (siehe Abschnitt 5.3.1.3.2.1).
- **out**-Parameter
  Sie ermöglichen der aufgerufenen Methode einen *Schreibzugriff* auf eine Variable der rufenden Methode (siehe Abschnitt 5.3.1.3.2.2).
- **in**-Parameter
  Sie ermöglichen der aufgerufenen Methode einen *Lesezugriff* auf eine Variable der rufenden Methode (siehe Abschnitt 5.3.1.3.2.3).

In einer Klasse können keine Methoden koexistieren, deren Köpfe sich lediglich bei der Transferrichtung von Verweisparametern unterscheiden. Das folgende Programm kann also *nicht* übersetzt werden:

```
using System;

class Prog {
    public void Meto(ref int i) { Console.WriteLine("ref-Parameter"); }

    public void Meto(in int i) { Console.WriteLine("in-Parameter"); }

    static void Main() {
        Prog p = new Prog();
        int i = 13;
        p.Meto(ref i);
        p.Meto(in i);
    }
}
```

Die beiden folgenden Methoden dürfen hingegen in einer Klasse koexistieren, weil `i` in der ersten Variante ein Wert- und in der zweiten Variante ein Verweisparameter ist:

```
public void Meto(int i) { Console.WriteLine("Wertparameter"); }
public void Meto(in int i) { Console.WriteLine("in-Parameter"); }
```

Im Abschnitt 5.3.5 zum *Überladen* von Methoden wird erläutert, wann sich die *Signaturen* von zwei Methoden unterscheiden, sodass sie in derselben Klasse definiert werden dürfen.

5.3.1.3.2.1 ref-Parameter

Ein **ref**-Parameter ermöglicht es der aufgerufenen Methode, auf eine als Aktualparameter übergebene Variable lesend und schreibend zuzugreifen.[1] Die Methode erhält beim Aufruf keine *Kopie* des Variableninhalts, sondern die Speicheradresse des Originals. Alle methodenintern über den Formalparameternamen vorgenommenen Modifikationen wirken sich direkt auf das Original aus.

Als **ref**-Aktualparameter sind nur *Variablen* erlaubt (keine Literale), weil eine Übergabe per Verweis stattfindet. Eine als **ref**-Aktualparameter fungierende Variable muss außerdem initialisiert sein und den *exakten* Formalparametertyp besitzen. Es findet also keine implizite Typanpassung statt. Auf den garantiert definierten Wert eines **ref**-Aktualparameters kann die gerufene Methode auch lesend zugreifen. Im Unterschied zu den Wertparametern und den gleich vorzustellenden **in**- bzw. **out**-Parametern ermöglichen die **ref**-Parameter einen Informationsfluss in *beide* Richtungen.

Das kennzeichnende Schlüsselwort **ref** ist in der Methodendefinition *und* beim Methodenaufruf anzugeben.

---

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref

Im folgenden Programm (nach dem aus dem Abschnitt 5.3.1.3.1 bekannten Strickmuster) tauscht eine Instanzmethode die Werte zwischen den als **ref**-Aktualparameter übergebenen Variablen:[1]

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($"Vorher:  x = {x}, y = {y}");`
`  p.Tausche(ref x, ref y);`
`  Console.WriteLine($"Nachher: x = {x}, y = {y}");`
`  }`
`}`
	`Vorher:  x = 1, y = 2`
	`Nachher: x = 2, y = 1`

Im Beispiel werden die lokalen (auf dem Stack befindlichen) Variablen `x` und `y` der Methode **Main()** als **ref**-Aktualparameter der Methode `Tausche()` verändert. Es ist auch möglich, auf dem Heap befindliche Instanzvariablen über **ref**-Parameter zu verändern, z. B.:

Quellcode	Ausgabe

`using System;`
 	
`class Prog {`
`  int a, b;`
 	
`  void Tausche(ref int a, ref int b) {`
`  int temp = a;`
`  a = b;`
`  b = temp;`
`  }`
 	
`  static void Main() {`
`  Prog p = new Prog();`
`  p.a = 1; p.b = 2;`
`  Console.WriteLine($"Vorher:  p.a = {p.a}, p.b = {p.b}");`
`  p.Tausche(ref p.a, ref p.b);`
`  Console.WriteLine($"Nachher: p.a = {p.a}, p.b = {p.b}");`
`  }`
`}`
	`Vorher:  p.a = 1, p.b = 2`
	`Nachher: p.a = 2, p.b = 1`

Es folgt ein Satz für Begriffsakrobaten: Wird eine *Referenzvariable* (hat als Inhalt eine Objektadresse) als **ref**-Aktualparameter übergeben, dann …

[1] Durch die Verwendung einer statischen Methode wäre die Demonstration des **ref**-Parameterverhaltens auch ohne `Prog`-Objekt möglich (siehe Abschnitt 5.7.3). Das Visual Studio macht den sinnvollen Vorschlag, die Methode `Tausche()` statisch zu definieren. Wir folgen dem Vorschlag nicht, weil wir bisher nur die spezielle statische Methode **Main()** kennen.

- kann man methodenintern nicht nur auf das Objekt auf dem Heap zugreifen (z. B. auf seine Instanzvariablen bei entsprechenden Zugriffsrechten),
- sondern kann auch den Inhalt der übergebenen Referenzvariablen ändern, sodass sie z. B. anschließend auf ein anderes Objekt zeigt.

Ein möglicher Einsatzzweck ist eine methodenintern zu verändernde und dem Aufrufer zurück zu liefernde Zeichenfolge. Wie sich im Abschnitt 6.3.1.1 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 aus der Methode hinaus (zum Aufrufer) zu befördern:

Quellcode	Ausgabe

`using System;`
 	
`class Prog {`
`  void RefRefParDemo(ref String spar) {`
`  spar = "NEU";`
`  }`
 	
`  static void Main() {`
`  string s = "ALT";`
`  Prog p = new Prog();`
`  p.RefRefParDemo(ref s);`
`  Console.WriteLine(s);`
`  }`
`}`
	`NEU`

Ein weiteres Beispiel für die sinnvolle Verwendung des **ref**-Modifikators bei einem Parameter mit Referenztyp liefert die im Abschnitt 6.2.2 vorgestellte statische Methode **Resize()** der Klasse **Array**.

5.3.1.3.2.2 out-Parameter

Über einen **out**-Parameter kann man einer Methode die Veränderung einer als Aktualparameter übergebenen Variable ermöglichen.[1] Alle methodenintern über den Formalparameternamen vorgenommenen Modifikationen wirken sich auf das Original aus.

Als **out**-Aktualparameter sind nur *Variablen* erlaubt (keine Literale), weil eine Übergabe per Verweis stattfindet. Eine als **out**-Aktualparameter fungierende Variable muss außerdem den *exakten* Formalparametertyp haben. Es findet also keine implizite Typanpassung statt. Der Compiler interessiert sich *nicht* dafür, ob eine als **out**-Aktualparameter fungierende Variable beim Methodenaufruf initialisiert ist. Stattdessen stellt er sicher, dass jedem **out**-Parameter vor dem Verlassen der Methode ein Wert zugewiesen wird.[2]

Das kennzeichnende Schlüsselwort **out** ist in der Methodendefinition *und* beim Methodenaufruf anzugeben, z. B.:

---

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier

[2] Außerdem ist ein Lesezugriff auf den **out**-Aktualparameter erst nach einer Methoden-internen Wertzuweisung möglich.

Quellcode	Eingabe (**fett**) und Ausgabe

`using System;`
 	
`class Prog {`
`  void Lies(out int x, out int y) {`
`  Console.Write("x = ");`
`  x = Convert.ToInt32(Console.ReadLine());`
`  Console.Write("\ny = ");`
`  y = 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`

Seit C# 7.0 kann ein **out**-Aktualparameter durch eine sogenannte *Ausschussvariable* (engl.: *discard variable*) ersetzt werden, wenn man am Ergebniswert des Parameters nicht interessiert ist. Statt eines Variablennamens ist der Unterstrich zu setzen, z. B. in der folgenden Variante des letzten Beispiels:

Quellcode	Ausgabe (Eingaben **fett**)

`using System;`
 	
`class Prog {`
`  void Lies(out int x, out int y) {`
`  . . .`
`  }`
 	
`  static void Main() {`
`  Prog p = new Prog();`
`  int x;`
`  p.Lies(out x, out _);`
`  Console.WriteLine("\nx = " + x);`
`  }`
`}`
	`x = 4`
	
	`y = 7`
	
	`x = 4`

Eine ebenfalls mit C# 7.0 eingeführte Syntax-Vereinfachung ist die **Inline-Variablendeklaration** (engl.: *inline variable declaration*) bei **out**-Aktualparametern, die in einer weiteren Variante des aktuellen Beispiels demonstriert wird:

```
  static void Main() {
      Prog p = new Prog();
      p.Lies(out int x, out int y);
      Console.WriteLine($"\nx % y = {x % y}");
  }
```

Wenn die als **out**-Aktualparameter fungierenden lokalen Variablen erstmals im Methodenaufruf benötigt werden, dann können sie in der Parameterliste deklariert werden, wobei der Typ anzugeben ist. Ihre Gültigkeit erstreckt sich wie bei anderen lokalen Variablen bis zum Ende des Blocks mit der Deklaration.

Ohne eine vorherige Typdefinition mehrere Werte an den Methodenaufrufer zu übergeben, gelingt auch mit den im Abschnitt 6.6 behandelten Struktur-Tupel – Typen. Dabei hat der Aufrufer die Wahl, …

- die Rückgaben gemeinsam in einer Container-Variablen abzulegen, um später über Felder einzelne Werte anzusprechen,
- oder die Rückgaben auf einzelne Variablen zu verteilen.

5.3.1.3.2.3 in-Parameter

Über einen **in**-Parameter kann man einer Methode den Lesezugriff auf eine als Aktualparameter übergebene Variable ermöglichen.[1] Die Methode erhält beim Aufruf keine *Kopie* des Variableninhalts, sondern die Speicheradresse des Originals. C# unterstützt **in**-Parameter ab Version 7.2.

Als **in**-Aktualparameter sind nur *Variablen* erlaubt (keine Literale), weil eine Übergabe per Verweis stattfindet. Eine als **in**-Aktualparameter fungierende Variable muss außerdem initialisiert sein und den *exakten* Formalparametertyp besitzen. Es findet also keine implizite Typanpassung statt. Der Compiler stellt sicher, dass eine als **in**-Aktualparameter fungierende Variable beim Methodenaufruf initialisiert ist und verhindert einen Methoden-internen Schreibzugriff auf einen **in**-Parameter.

Das kennzeichnende Schlüsselwort **in** *muss* in der Methodendefinition und *sollte* auch beim Methodenaufruf angegeben werden, z. B.:

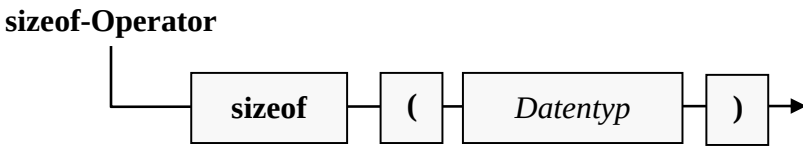
Quellcode	Ausgabe

`using System;`
 	
`class Prog {`
`  public void Meto(int i) {Console.WriteLine("Wertparameter " + i);}`
`  public void Meto(in int i) {Console.WriteLine("in-Parameter " + i);}`
 	
`  static void Main() {`
`  Prog p = new Prog();`
`  int i = 13;`
`  p.Meto(in i);`
`  p.Meto(i);`
`  }`
`}`
	`in-Parameter 13`
	`Wertparameter 13`

Wenn sich zwei Methodenköpfe nur darin unterscheiden, dass für einen Parameter *kein* Transfermodus bzw. der **in**-Transfermodus deklariert ist, dann gibt das Schlüsselwort **in** vor dem Aktualparameter den Ausschlag für die Wahl der auszuführenden Methode.

Durch einen **in**-Parameter spart man im Vergleich zu einem Wertparameter einen Kopiervorgang. Das lohnt sich aber nur, wenn der Datentyp des Aktualparameters (z. B. eine Struktur) mehr Platz beansprucht als eine Speicheradresse.

Den Platzbedarf eines Datentyps ermittelt man über den **sizeof**-Operator, dessen einfache Syntax mit einem Blick zu erfassen ist:



Die Größe einer Speicheradresse in Bytes ist aus der statischen Eigenschaft **Size** der Struktur **IntPtr** abzulesen und beträgt bei einem 64-Bit – Betriebssystem 8 Bytes. Im obigen Beispiel wird offenbar nur die Technik der **in**-Parameter demonstriert, aber kein Nutzen erzielt:

---

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in-parameter-modifier

Quellcode	Ausgabe

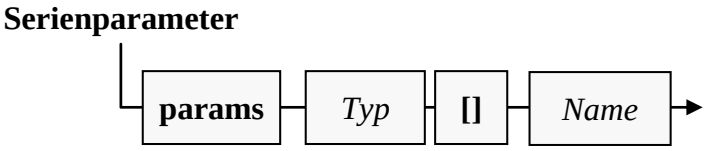
`Console.WriteLine("Größe in Bytes:\n Speicheradresse: {0} "+`
`  "\n int-Variable:  {1}", IntPtr.Size, sizeof(int));`
	`Größe in Bytes:`
	` Speicheradresse: 8`
	` int-Variable:  4`

5.3.1.3.3 Serienparameter

Vielleicht haben Sie sich schon darüber gewundert, dass man beim Aufruf der Methode **Console.WriteLine()** hinter einer geeigneten Formatierungszeichenfolge *unterschiedlich viele* Ausdrücke durch jeweils ein Komma getrennt als Aktualparameter übergeben darf, z. B.:[1]

```
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 der Parameterliste stehen und in der Definition durch das Schlüsselwort **params** gekennzeichnet sein muss. Im Syntaxdiagramm zum Methodenkopf wurde diese Option der Einfachheit halber weggelassen. Hier wird das Syntaxdiagramm zum Serienparameter nachgeliefert:



Als Transfermodus ist beim Serienparameter nur das Kopieren erlaubt. Das Schlüsselwort **params** ist also nicht mit den Verweisparameter-Schlüsselwörtern **ref**, **out** und **in** kombinierbar.

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);`
`  p.PrintSum();`
`  double[] da = {1.1, 2.2, 3.9 };`
`  p.PrintSum(da);`
`  }`
`}`
	`Die Summe ist = 2,2`
	`Die Summe ist = 5,800000000000001`
	`Die Summe ist = 0`
	`Die Summe ist = 7,2`

Beim Methodenaufruf wird das Schlüsselwort **params** *nicht* angegeben. Beim Aufruf darf die Liste der Aktualparameter eine beliebige Länge haben, sogar die Länge null. Es ist erlaubt, aber keineswegs erforderlich, als Aktualparameter eine Variable mit Array-Datentyp (siehe Abschnitt 6.2) zu liefern.

---

[1] Die traditionelle Variante der formatierten Ausgabe unter Verwendung von Platzhaltern (vgl. Abschnitt 4.2.2.1) wird seit Einführung der Zeichenfolgeninterpolation (vgl. Abschnitt 4.2.2.2) seltener verwendet.

5.3.1.4 Methodenrumpf

Über die Blockanweisung, die den Rumpf einer Methode bildet, haben Sie bereits erfahren:

- Hier werden die Formalparameter wie lokale Variablen verwendet.
- Wert-, **ref**- und **in**-Parameter werden von der rufenden Methode initialisiert. Damit kann die aufrufende Methode den Ablauf der gerufenen Methode beeinflussen. Besitzt ein solcher Parameter einen Referenzdatentyp, dann kann die gerufene Methode beim Bestehen entsprechender Zugriffsrechte das referenzierte Heap-Objekt verändern.
- Über **ref**- und **out**-Parameter können lokale Variablen der rufenden Methode verändert werden.
- Die **return**-Anweisung dient zur Rückgabe eines Werts 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 der Methode zu realisieren.

Im letzten Satz war bewusst von *dem Zweck* einer Methode die Rede und nicht von *den Zwecken*. Durch Mehrzweckmethoden verschlechtern sich Lesbarkeit und Wartungsfreundlichkeit, während das Fehlerrisiko steigt, weil z. B. die für eine Teilaufgabe benötigten Variablen auch im Codesegment anderer Teilaufgaben gültig sind und durch Tippfehler unverhofft ins Spiel kommen oder in Mitleidenschaft gezogen werden können. Um diese Nachteile zu vermeiden, sollte für jede Aufgabe bzw. Aktivität eine eigene Methode definiert werden (Bloch 2018, S. 263).

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 5.3.4).

5.3.1.5 Lokale (eingeschachtelte) Methoden

Seit C# 7.0 ist es möglich, eine lokale Methode innerhalb einer ausführbaren Programmeinheit (z. B. Methode oder Eigenschaft) zu definieren. Dies ist eine sinnvolle Option, wenn es sich um eine ausschließlich lokal benötigte Hilfsmethode handelt:

- Im Quellcode kommt die eingeschränkte Verwendung der Hilfsmethode zum Ausdruck.
- Weil sich die Methode nicht auf Klassenebene befindet, ist keine Absprache mit anderen Programmierern erforderlich, die an derselben Klasse arbeiten.
- Weil die lokale Methode auf die lokalen Variablen der umgebenden Methode zugreifen kann, müssen diese Variablen weder auf Klassenebene veröffentlicht noch über Parameter transferiert werden. Seit C# 8 kann eine lokale Methode aber als **static** definiert werden, sodass kein Zugriff auf die lokalen Variablen der umgebenden Methode möglich ist (siehe unten).

Im folgenden Programm wird innerhalb der Methode `M()` die lokale Methode `LM()` definiert und verwendet:[1]

[1] Ein Visual Studio - Projekt mit dem Programm `LMDemo` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Lokale Methoden

Quellcode	Ausgabe

`using System;`
 	
`class LMDemo {`
`  static readonly int staticField = 1;`
`  readonly int field = 2;`
 	
`  int M() {`
`  int local = 3;`
`  int LM() { return staticField + field + local; }`
`  return LM();`
`  }`
 	
`  static void Main() {`
`  LMDemo p = new LMDemo();`
`  Console.WriteLine(p.M());`
`  }`
`}`
	`6`

Das Verschachteln von Methoden lässt sich auf mehreren Ebenen fortsetzen. Es ist also auch in einer lokalen Methode erlaubt, eingeschachtelte Methoden zu definieren.

Als Modifikatoren sind für lokale Methoden ausschließlich erlaubt:

- **static**
  Seit C# 8 kann eine lokale Methode als statisch definiert werden. Dann ist weiterhin ein Zugriff auf statische Felder möglich, aber nicht auf Instanzvariablen des Objekts und nicht auf lokale Variablen der umgebenden Methode, z. B.:

```
using System;

class LMDemo {
    static readonly int staticField = 1;
    readonly int field = 2;

    int M() {
        int local = 3;
        static int SLM() => staticField + field + local;
        return SLM();
    }

    static void Main() {
        LMDemo p = new LMDemo();
        Console.WriteLine(p.M());
    }
}
```

  Eine lokale Methode als **static** zu deklarieren, hat die folgenden Vorteile:
  - Weil kein Zugriff auf die lokalen Variablen der umgebenden Methode möglich ist, müssen diese Variablen beim Aufruf der lokalen Methode nicht in deren Stack-Speicher kopiert werden. Der somit erzielte Performanzvorteil ist aber nur dann spürbar, wenn die umgebende Methode umfangreiche lokale Variablen besitzt.
  - Durch den eingeschränkten Zugriff sind manche Fehler ausgeschlossen, und bei mancher Fehlerursache kommt die die lokale Methode nicht in Frage.

- **async**
  Durch den Modifikator **async** wird eine Methode als *asynchron* deklariert. Mit asynchronen Methoden beschäftigt sich der Abschnitt 17.5 in Baltes-Götz (2021).

- **unsafe**
  Durch den Modifikator **unsafe** muss unsicherer Code (z. B. unter Verwendung von Zeigeroperationen) markiert werden. Mit dieser selten benötigten Programmiertechnik werden wir uns im Manuskript nicht beschäftigen.

5.3.2 Methodenaufruf und Aktualparameter

Beim Aufruf einer Instanzmethode, z. B.:

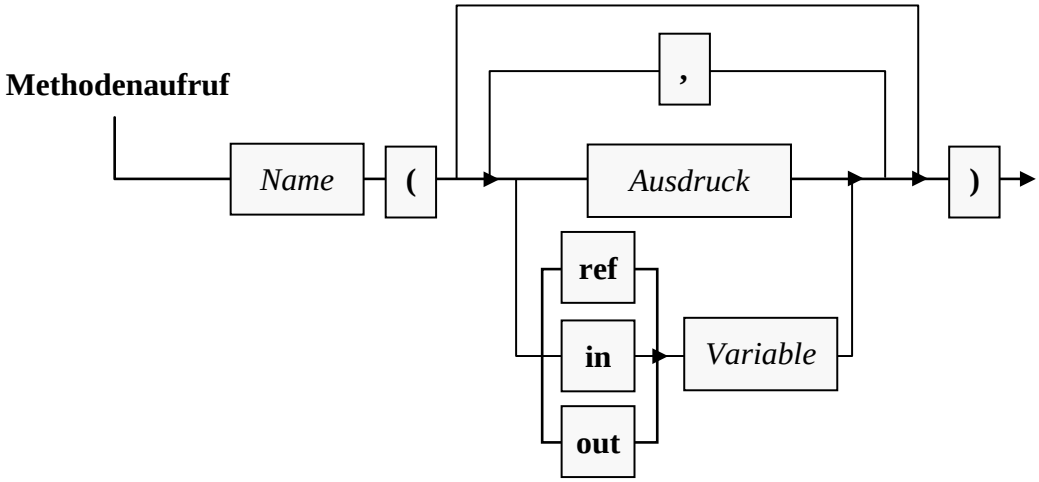
```
b1.Zeige();
```

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 im Normalfall (bei Verzicht auf benannte und optionale Parameter, siehe Abschnitt 5.3.3) um eine analog zur Formalparameterliste geordnete Serie von Ausdrücken bzw. Variablen kompatiblen Typs handeln muss.



Es ist grundsätzlich eine Parameterliste anzugeben, ggf. eine leere.

Als Beispiel betrachten wir einen Aufruf der im Abschnitt 5.3.1.3.1 vorgestellten Variante der `Bruch`-Methode `Addiere()`:

```
b1.Addiere(1, 3, true);
```

Einem **ref**- oder **out**-Parameter muss auch beim Aufruf das Schlüsselwort **ref** bzw. **out** vorangestellt werden, z. B.:

```
p.Tausche(ref x, ref y);
```

Es wird empfohlen, auch bei einem **in**-Parameter analog zu verfahren.

Liefert eine Methode einen Wert zurück, dann kann der aus ihrem Aufruf bestehende **Ausdruck** als Argument in komplexeren Ausdrücken verwenden werden, z. B.:

```
do
  Console.WriteLine("Welchen Bruch möchten Sie kürzen?");
while (!b1.Frage());
```

Durch ein angehängtes Semikolon entsteht aus einem Methodenaufruf eine vollständige **Anweisung**, wobei ein Rückgabewert ggf. ignoriert wird, z. B.:

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

Soll in einer Methodenimplementierung das aktuell handelnde Objekt eine andere Instanzmethode ausführen, dann muss beim Aufruf dieser vom selben Objekt auszuführenden Methode *keine* Objektbezeichnung angegeben werden. In beiden Varianten der `Bruch`-Methode `Addiere()` soll das beauftragte Objekt den via Parameterliste übergebenen Bruch zum eigenen Wert addieren und das Resultat (bei der Variante aus dem Abschnitt 5.3.1.3.1 parametergesteuert) 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.:

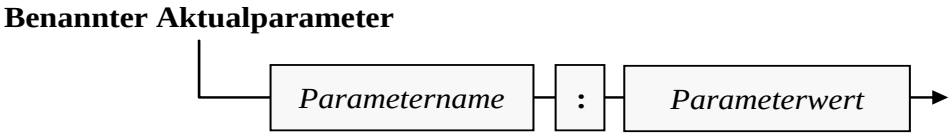
```
this.Kuerze();
```

5.3.3 Benannte und optionale Parameter

In diesem Abschnitt werden Varianten zur Definition und Verwendung von Parametern nachgeliefert, die bisher der Übersichtlichkeit halber weggelassen wurden. Mit der nachträglichen Beschreibung ist aber keinesfalls eine Herabstufung der durchaus nützlichen Optionen verbunden.

5.3.3.1 Benannte Aktualparameter

Statt beim Aufruf einer Methode mit zwei oder mehr Parametern eine analog zur Formalparameterliste geordnete Serie von Ausdrücken bzw. Variablen passenden Typs zu liefern, kann man *benannte* Aktualparameter verwenden (engl. Bezeichnung: *named arguments*):



Der folgende Aufruf der im Abschnitt 5.3.1.3.1 vorgestellten Variante der `Bruch`-Methode `Addiere()`

```
b1.Addiere(1, 3, true);
```

lässt sich äquivalent so formulieren:

```
b1.Addiere(npar: 3, zpar: 1, autokurz: true);
```

Als Vorteile der benannten Aktualparameter sind zu nennen:

- Die Lesbarkeit des Quellcodes wird verbessert.
- In Kombination mit den im Abschnitt 5.3.3.2 behandelten optionalen Parametern ist es möglich, für manche optionale Parameter einen Wert anzugeben und bei anderen die Voreinstellung beizubehalten.
- Man muss sich nicht an die Definitionsreihenfolge der Parameter halten.
- Die benannten Parameter erleichtern zusammen mit den im Abschnitt 5.3.3.2 behandelten optionalen Parametern die COM-Interoperabilität.[1]

Es ist erlaubt, auf eine Serie von positionsbezogenen Parametern (ab jetzt bezeichnet als *Positionsparameter*) benannte Parameter (ab jetzt bezeichnet als *Namensparameter*) folgen zu lassen, z. B.:

```
b1.Addiere(1, 3, autokurz: true);
```

Seit C# 7.2 dürfen Positionsparameter auf Namensparameter folgen, sofern die Namensparameter an ihrer korrekten Position stehen, z. B.:

```
b1.Addiere(zpar: 1, 2, true);
```

5.3.3.2 Optionale Parameter

In einer Methodendefinition kann seit C# 4.0 für einen Formalparameter ein Voreinstellungswert angegeben werden, sodass 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 im Abschnitt 5.3.3.1 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 die folgenden Regeln zu beachten:

- Als Voreinstellungswert ist ein konstanter Ausdruck erlaubt, dessen Wert schon zur Übersetzungszeit feststeht, z. B.:
  ```
  void IncreaseRes(int i = Int32.MaxValue - 1)  {
    . . .
  }
  ```
- Auf einen optionalen Formalparameter darf kein obligatorischer mehr folgen.

Optionale Parameter sind auch bei Konstruktoren und Indexern erlaubt (siehe Abschnitte 5.4.3.1 bzw. 5.11).

Beim Aufruf können optionale Parameter weggelassen werden. Um eine Teilmenge der optionalen Parameter mit Werten zu versorgen, arbeitet man mit benannten Aktualparametern (siehe Abschnitt 5.3.3.1), z. B.:

---

[1] Unter Windows ist es gelegentlich 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.

Quellcode	Ausgabe

`using System;`
 	
`class Prog {`
`  void PrintSum(int a = 1, int b = 2, int c = 3) {`
`  Console.WriteLine("a = " + a);`
`  Console.WriteLine("b = " + b);`
`  Console.WriteLine("c = " + c + '\n');`
`  }`
`  static void Main() {`
`  Prog p = new Prog();`
`  p.PrintSum(13, 4711);`
`  p.PrintSum(7);`
`  p.PrintSum();`
`  p.PrintSum(c: 999);`
`  }`
`}`
	`a = 13`
	`b = 4711`
	`c = 3`
	
	`a = 7`
	`b = 2`
	`c = 3`
	
	`a = 1`
	`b = 2`
	`c = 3`
	
	`a = 1`
	`b = 2`
	`c = 999`

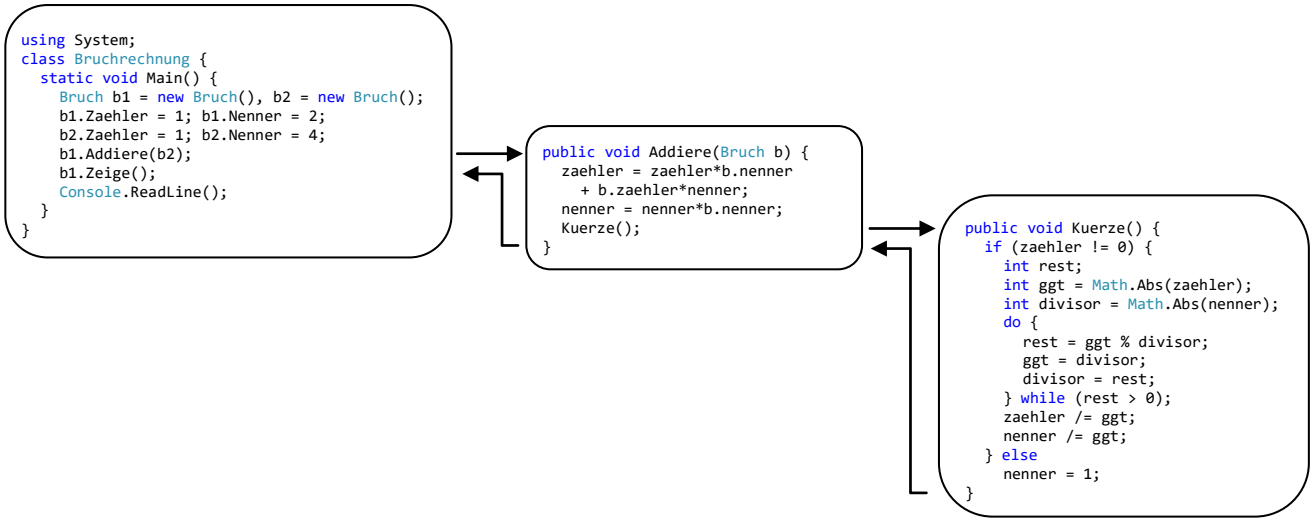
5.3.4 Debug-Einsichten zu (verschachtelten) Methodenaufrufen

Verschachtelte Methodenaufrufe stellen keine Besonderheit dar, sondern den selbstverständlichen Normalfall. Anhand der folgenden Bruchrechnungsstartklasse

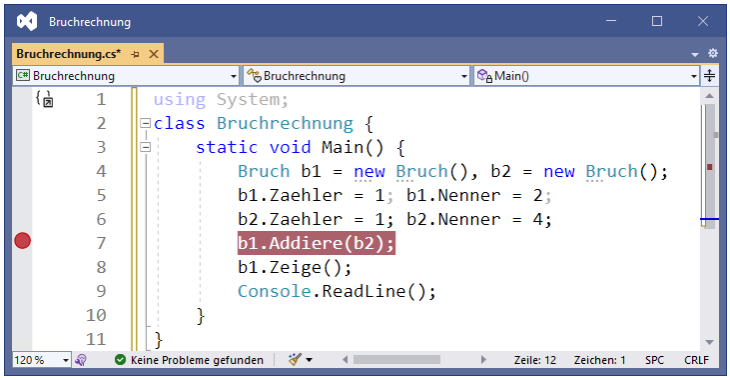
```
using System;

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 der Entwicklungsumgebung Visual Studio untersucht werden, was bei der folgenden Aufrufverschachtelung geschieht:

- Die statische Methode **Main()** der Klasse `Bruchrechnung` ruft die `Bruch`-Instanzmethode `Addiere()` auf.
- Die `Bruch`-Instanzmethode `Addiere()` ruft die `Bruch`-Instanzmethode `Kuerze()` auf.



Wir verwenden dabei die zur Fehlersuche konzipierte Debug-Technik der Entwicklungsumgebung. Das Bruchrechnungsprogramm soll an mehreren Stellen durch einen sogenannten **Halte-** bzw. **Unterbrechungspunkt** (engl. *breakpoint*) gestoppt werden, sodass wir jeweils die Lage im Hauptspeicher inspizieren können. Um einen Haltepunkt zu setzen oder wieder zu entfernen, setzt man im Quellcodeeditor 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

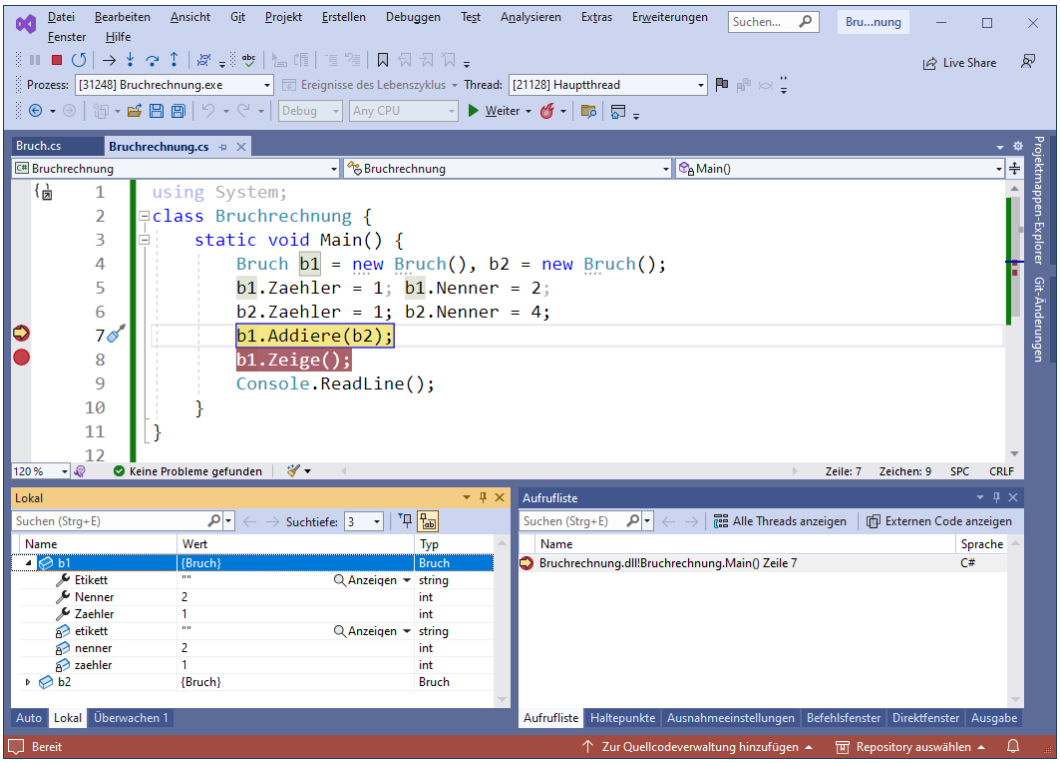
  ```
  ggt = divisor;
  ```

  im Block der **do-while** - Schleife.

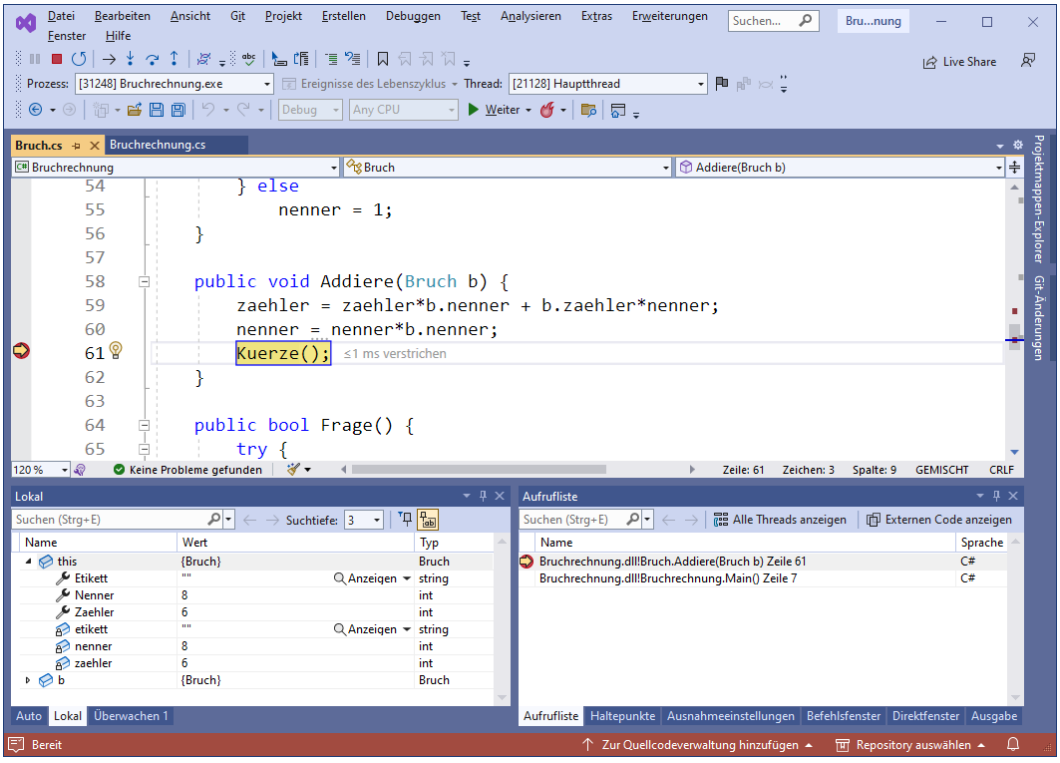
Starten Sie das Programm über den Schalter ▶, die Funktionstaste **F5** oder den Menübefehl

Debuggen > Debugging starten

Die Entwicklungsumgebung stoppt das Programm beim ersten Unterbrechungspunkt und zeigt im Quellcode-Editor den erreichten Programmfortschritt an. Unten links zeigt die mit **Lokal** betitelte Registerkarte die beiden lokalen Referenzvariablen der **Main()** - Methode (`b1`, `b2`) und auf Wunsch (nach einem Mausklick auf den ▷ - Schalter neben einer Referenzvariablen) das Innenleben des referenzierten Objekts. Das mit **Aufrufliste** betitelte Fenster unten rechts zeigt, dass aktuell nur die Startmethode **Main()** aktiv ist (mit Daten im Stack-Bereich des Speichers):



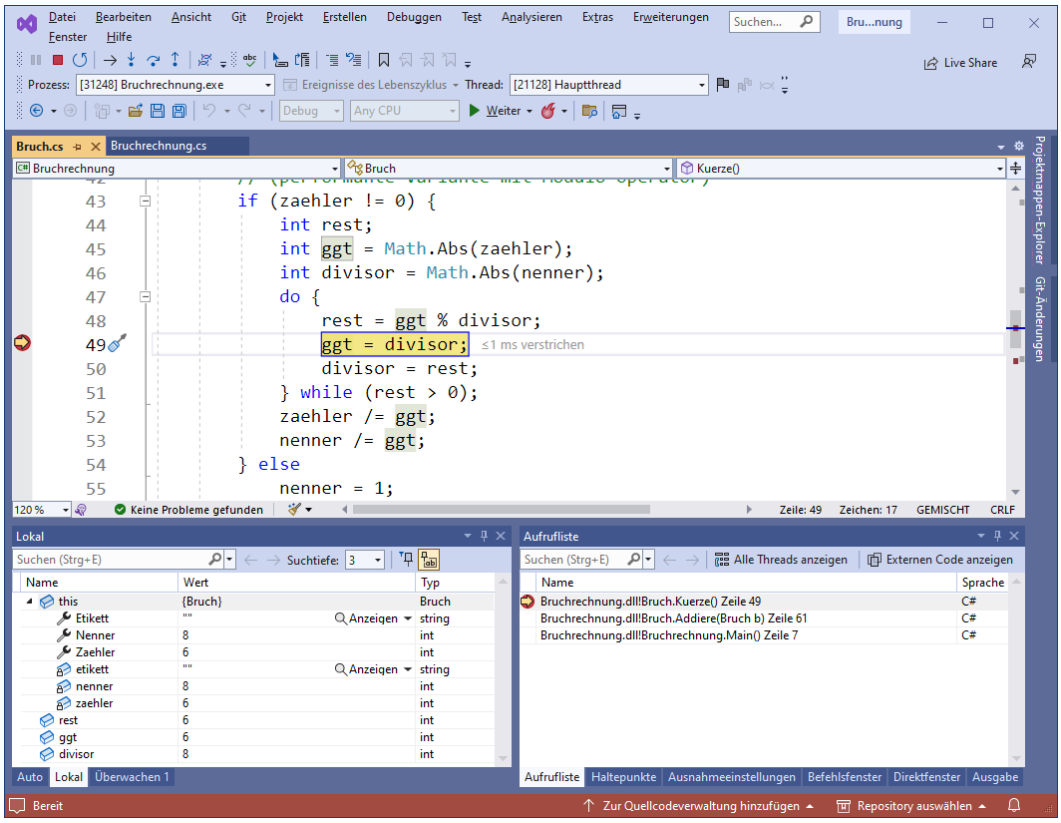
Veranlassen Sie mit dem Schalter ▶ oder mit der Funktionstaste **F5** die Fortsetzung des Programms. Beim Erreichen des zweiten Haltepunkts (Anweisung „`Kuerze();`“ in der Methode `Addiere()`) liegen auf dem Stack die Daten und Verwaltungsinformationen (die *Stack Frames*) der Methoden `Addiere()` und **Main()** übereinander:



Die Registerkarte **Lokal** zeigt als lokale Variablen der Methode `Addiere()`:

- **this** (Referenz auf das handelnde `Bruch`-Objekt)
  Erwartungsgemäß ist der Bruch noch nicht gekürzt.
- Parameter `b` (Referenz auf das zu addierende `Bruch`-Objekt)

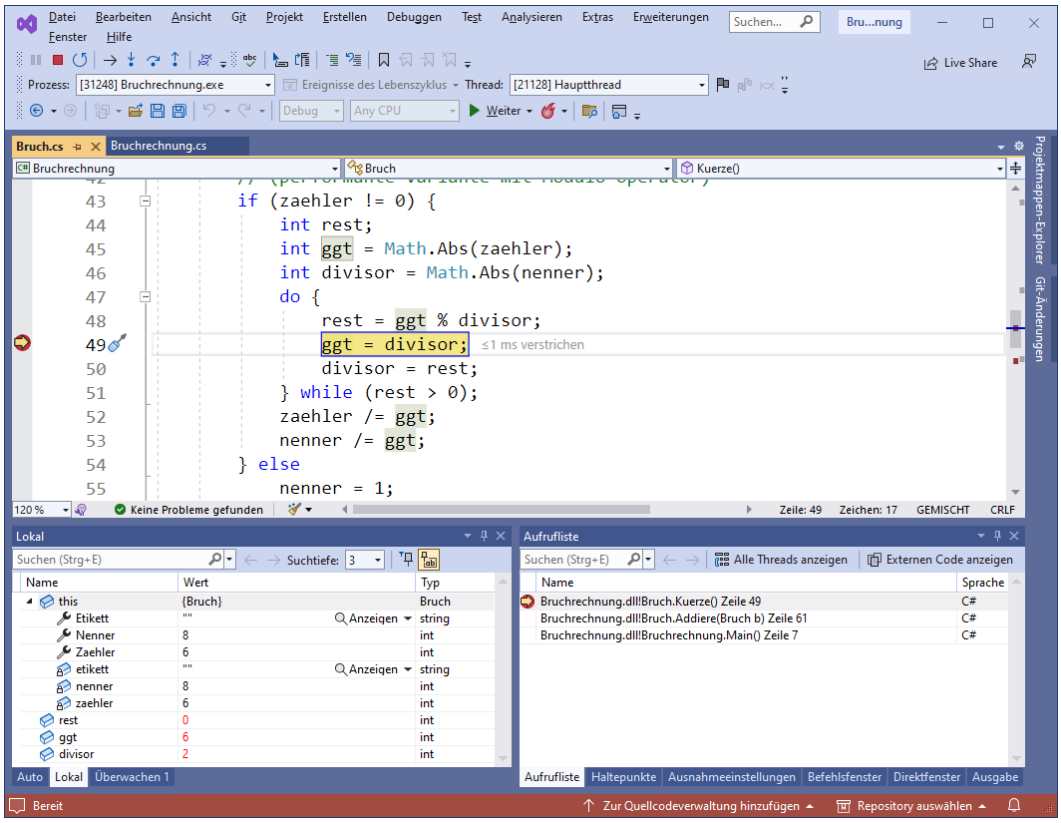
Beim Erreichen des dritten Haltepunkts (Anweisung „`ggt = divisor;`“ in der Methode `Kuerze()`) liegen die Stack Frames der Methoden `Kuerze()`, `Addiere()` und **Main()** übereinander:



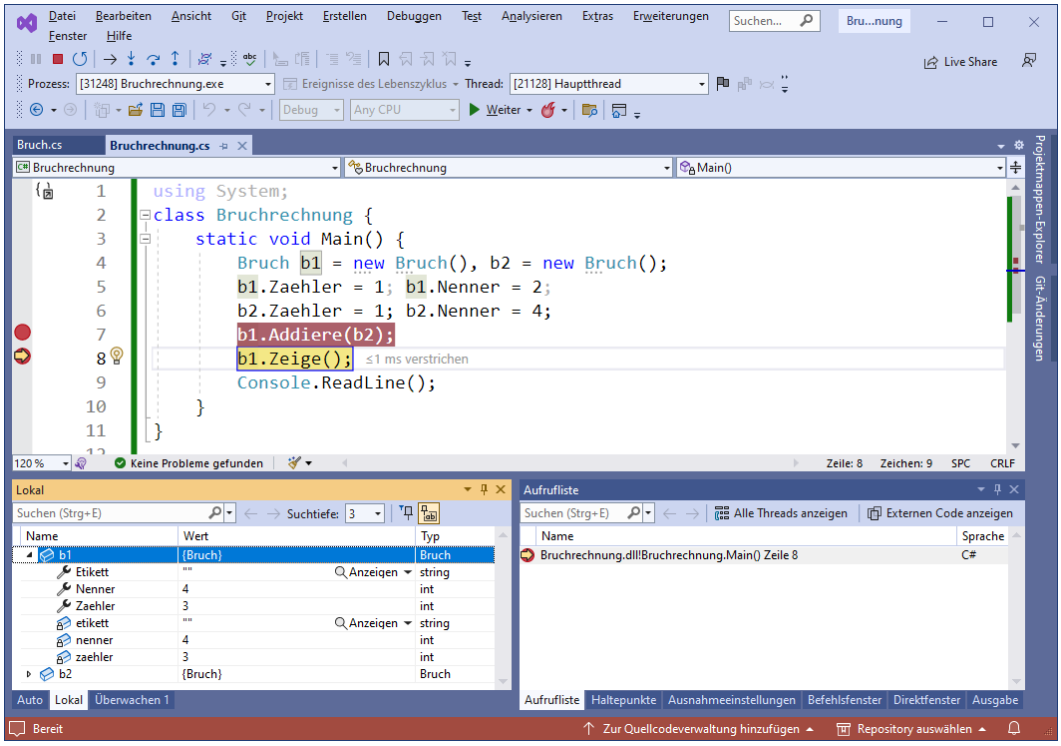
Das Fenster **Lokal** zeigt als lokale Variablen der Methode `Kuerze()`:

- **this** (Referenz auf das handelnde `Bruch`-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 drei Fortsetzungsbefehle bis zum Verlassen der Methode `Kuerze()` erforderlich, wobei die Werte der lokalen Variablen den Verarbeitungsfortschritt erkennen lassen, z. B.:

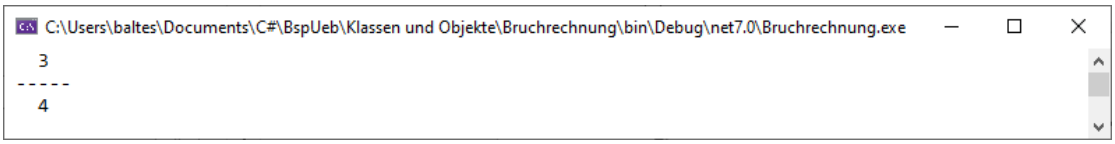


Beim Erreichen des letzten Haltepunkts (Anweisung „`b1.Zeige();`“ in **Main()**) ist nur noch der Stack Frame der Methode **Main()** vorhanden:



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

Ferner lassen sich per **Debuggen**-Menü alle Haltepunkte **deaktivieren** bzw. **aktivieren**.

Weil der verfügbare Stack-Speicher endlich ist, kann es bei einer Aufrufverschachtelung und der damit verbundenen Stapelung von Stack Frames zu einem Laufzeitfehler vom Typ **StackOverflowException** kommen. Das wird aber nur bei einem schlecht entworfenen bzw. fehlerhaften Algorithmus passieren.

5.3.5 Methoden überladen

Die im Abschnitt 5.3.1.3.1 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 sogenannte *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 alle erfüllt sind:[1]

- Die Namen der Methoden sind identisch.
- Die Formalparameterlisten sind gleich lang.
- Positionsgleiche Parameter haben denselben Datentyp.
- Positionsgleiche Parameter haben denselben Transfermodus, d.h. …
  - beide sind Wertparameter,
  - oder beide sind Verweisparameter.

  Bei zwei Verweisparametern sorgen unterschiedliche Richtungsangaben (**ref**, **in**, **out**) *nicht* für abweichende Signaturen.[2]

Für die Signatur einer Methode sind irrelevant:

---

[1] Bei den später zu behandelnden *generischen* Methoden (siehe Abschnitt 8.5) muss die Liste mit den Kriterien für die Identität von Signaturen erweitert werden.

[2] Hinsichtlich der Definition von Überladungsfamilien gehören die Parametermodifikatoren **ref**, **in** und **out** also nicht zur Signatur. Bei anderen, von den Methodensignaturen abhängigen Entscheidungen des Compilers (z. B. Überschreiben von Methoden bei der Vererbung, vgl. Abschnitt 7.9) gehören die Parametermodifikatoren aber doch zur Signatur).

- Der Rückgabetyp
  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 für den Compiler entscheidbar sein, welche Methode aus einer Überladungsfamilie zu verwenden ist.
- Modifikatoren
- Die *Namen* der Formalparameter
- Das beim letzten Formalparameter erlaubte **params**-Schlüsselwort (siehe ECMA 2022, S. 54ff).

Ist bei einem Methodenaufruf die passende Überladung nicht eindeutig zu bestimmen, dann meldet der Compiler einen Fehler, was bei einem sinnvollen Entwurf von überladenen Methoden nur sehr selten passiert.

Von einer Methode unterschiedlich parametrisierte Varianten in eine Klassendefinition aufzunehmen, lohnt sich z. B. in den 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 decimal 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]
- Für eine Methode sollen unterschiedlich 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 5.3.3).

Zum Schluss noch ein Beispiel aus dem Kuriositäten- bzw. Gruselkabinett. Wenn zu einer Methode mit optionalen Parametern (siehe Abschnitt 5.3.3.2) eine Überladung mit einer kürzeren Parameterliste existiert, wobei ein optionaler Parameter fehlt, dann wird bei einem Aufruf mit der kürzeren Aktualparameterliste die Methode *ohne* den optionalen Parameter aufgerufen. In dieser Konstellation ist die Definition von Voreinstellungswerten also wirkungslos, z. B.:[2]

---

[1] So tauscht z. B. die folgende statische Methode die Inhalte von zwei **ref**-Parametern mit beliebigem (natürlich identischem) Datentyp:

```
static void Tausche<T>(ref T a, ref T b) {
  T temp = a;
  a = b;
  b = temp;
}
```

[2] Diese Konstellation wurde aufgeklärt von Jens Weber (Universität Trier).

Quellcode	Ausgabe

`using System;`
 	
`class Prog {`
`  void Test(int i = 13) {`
`  Console.WriteLine("Opt. Par. = " + i);`
`  }`
 	
`  void Test() {`
`  Console.WriteLine("Ohne Parameter");`
`  }`
 	
`  static void Main() {`
`  Prog p = new Prog();`
`  p.Test();`
`  }`
`}`
	`Ohne Parameter`

Wird bei einem Einsatz der Klasse aus der Überladungsfamilie die kurze Parameterliste im Vertrauen auf den Voreinstellungswert des weggelassenen Parameters gewählt, dann sind ernste Folgen möglich (z. B. ein Raketenabsturz). Zu der unglücklichen Konstellation kann es z. B. kommen, wenn mehrere Programmierer an einer Klassendefinition arbeiten, oder wenn ein Programmierer mit größeren zeitlichen Unterbrechungen an einer Klassendefinition arbeitet.

5.4 Objekte

Im Abschnitt 5.4 geht es darum, wie Objekte erzeugt und im obsoleten Zustand wieder aus dem Speicher entfernt werden.

5.4.1 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, dann können deren Objekte ebenfalls über `Bruch`-Referenzvariablen verwaltet werden. Von der Vererbungstechnik der objektorientierten Programmierung haben Sie schon einiges gehört, doch steht die gründliche Behandlung im Kapitel 7 noch aus.

- **null**
  Wird einer Variablen dieses Referenzliteral zugewiesen, dann ist die Variable nicht undefiniert, sondern zeigt explizit auf nichts.[1]

Wir nehmen nunmehr offiziell zur Kenntnis, dass *Klassen als Datentypen* zugelassen sind und haben damit bislang die folgenden Datentypen zur Verfügung (vgl. Abschnitt 4.3.2):

- Elementare Typen (**bool**, **char**, **byte**, **double**, ...)
  Hier handelt es sich um Werttypen.
- Klassen (Referenztypen)
  Ist eine Variable vom Typ einer Klasse, dann 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 Feldern ausgestattet werden können.

5.4.2 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. im folgenden Ausdruck:

```
new Bruch()
```

Als Operanden erwartet der **new**-Operator einen Klassennamen, dem eine Parameterliste zu folgen hat, weil der **new**-Operand als Name eines *Konstruktors* fungiert (siehe Abschnitt 5.4.3). Die involvierte Klasse legt den Typ des Ausdrucks fest. Als Wert resultiert eine Referenz, die (im Rahmen bestehender Rechte) einen Zugriff auf das neue Objekt (seine Methoden, Eigenschaften, etc.) ermöglicht.

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 Arbeitsspeicher:[2]

---

[1] Seit C# 8 bietet der C# - Compiler eine optionale Unterstützung zur Vermeidung der **NullReferenceException** an, die aus didaktischen Gründen vorläufig ausgespart bleibt (siehe Abschnitt 15.1). Ist diese Einstellung (der sogenannte *Nullable-Kontext*) aktiv, dann reagiert der Compiler mit einer Warnung auf den Versuch, einer **Bruch**-Referenzvariablen der Wert **null** zuzuweisen, z. B.:

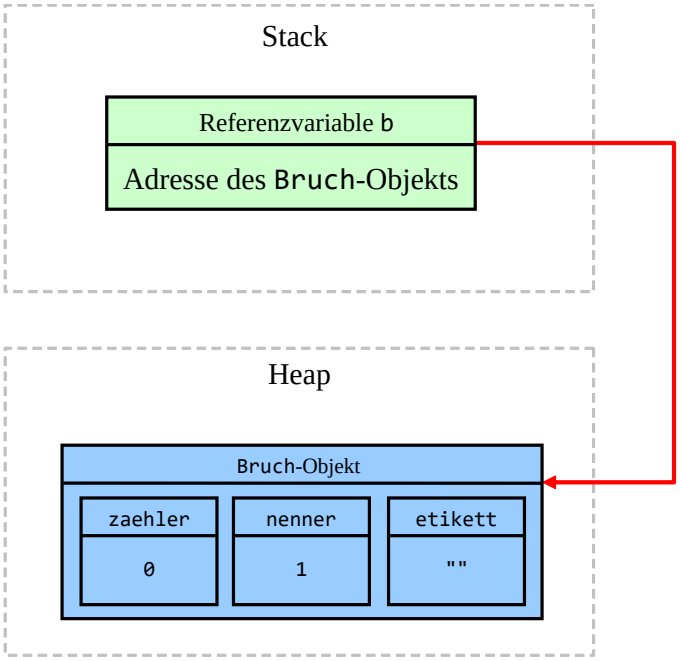
```
Bruch b = null;
```

Die Zuweisung wird aber *nicht* verhindert. Ein an die Typbezeichnung angehängtes Fragezeichen signalisiert dem Compiler explizit, dass bei einer `Bruch`-Variablen der Wert **null** zulässig ist, z. B.:

```
Bruch? b = null;
```

Seit .NET 6 ist der Nullable-Kontext bei neuen Projekten voreingestellt.

[2] 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 dieses **String**-Objekts.



Während lokale Variablen (während der Methodenausführung) im **Stack**-Bereich des programmeigenen Arbeitsspeichers abgelegt werden, entstehen Objekte mit ihren Instanzvariablen im **Heap**-Bereich.

In einem Programm können *mehrere* Referenzvariablen auf *dasselbe* Objekt zeigen, z. B.:

Quellcode	Ausgabe

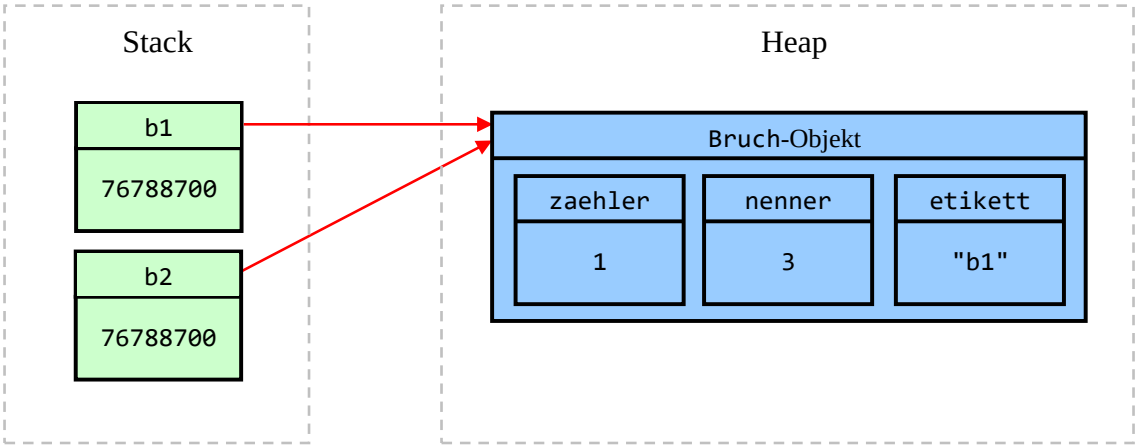
```
using System;
class Bruchrechnung {
  static void Main() {
    Bruch b1 = new Bruch();
    b1.Zaehler = 1;
    b1.Nenner = 3;
    b1.Etikett = "b1";
    Bruch b2 = b1;
    b2.Etikett = "b2";
    b1.Zeige();
  }
}
```

```
        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.

5.4.3 Objekte initialisieren

C# bietet verschiedene Möglichkeiten, um ein neues Objekt auf seinen Einsatz vorzubereiten.

5.4.3.1 Konstruktoren

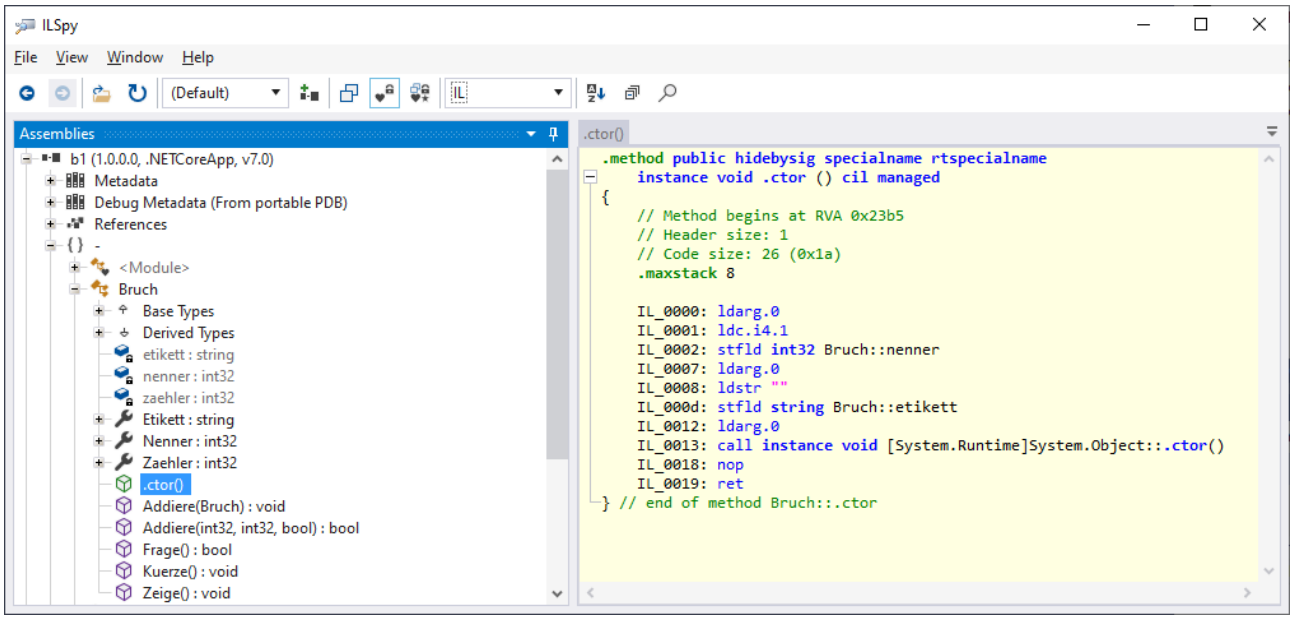
In diesem Abschnitt werden mit den sogenannten *Konstruktoren* spezielle Methoden behandelt, die beim Erzeugen von neuen Objekten zum Einsatz kommen, um deren Instanzvariablen zu initialisieren und/oder andere Arbeiten zu verrichten (z. B. Öffnen einer Datei). Ziel der Konstruktor-Tätigkeit ist es, ein neues Objekt in einen 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.

5.4.3.1.1 Standardkonstruktor

Ist beim Klassen-Design kein Konstruktor definiert worden, dann erhält die Klasse automatisch einen **Standardkonstruktor** (engl.: *default constructor*). Weil dieser Konstruktor keine Parameter besitzt, ergibt sich sein Aufruf aus dem Klassennamen durch Anhängen einer leeren Parameterliste, z. B.:

```
Bruch b = new Bruch();
```

Inspiziert man die aktuelle Entwicklungsstufe der Klasse `Bruch` mit dem Hilfsprogramm ILSpy (vgl. Abschnitt 3.4.1), dann entdeckt man den Standardkonstruktor unter dem Namen **.ctor** (Abkürzung für *constructor*):



In diese Methode fügt der Compiler automatisch IL-Code für die im Rahmen ihrer Deklaration initialisierten Instanzvariablen ein (betroffen: `nenner` und `etikett`):[1] Für eine automatische Null-Initialisierung (vgl. Abschnitt 5.2.3) ist hingegen kein IL-Code erforderlich.

Am Ende des IL-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.[2]

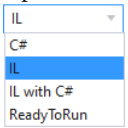
5.4.3.1.2 Explizite Konstruktoren

Beim Klassendesign ist es oft erforderlich, Konstruktoren *explizit* zu definieren, um das individuelle Initialisieren der Instanzvariablen von neuen Objekten unter Beachtung von Konsistenzbedingungen zu ermöglichen und/oder sonstige Vorbereitungen durchzuführen. Dabei sind die folgenden Regeln zu beachten:

- Ein Konstruktor trägt denselben Namen wie die Klasse.
- In der Definition wird *kein* Rückgabetyp angegeben.
- Es sind nur Zugriffsmodifikatoren erlaubt (z. B. **public**, **private**).
- Während der Standardkonstruktor bei konkreten Klassen die Schutzstufe **public** und bei abstrakten Klassen die Schutzstufe **protected** besitzt, haben explizit definierte Konstruktoren wie gewöhnliche Methoden die voreingestellte Schutzstufe **private**. Wenn sie für beliebige fremde Klassen (in beliebigen Assemblies) zur Objektkreation verfügbar sein sollen, dann ist in der Definition der Modifikator **public** zu verwenden.

---

[1] In ILSpy wählt man die Anzeige von IL-Code per Drop-Down – Menü:



[2] Abstrakte Klassen, mit denen wir uns im Abschnitt 7.13 im Zusammenhang mit der Vererbung beschäftigen werden, haben einen Standardkonstruktor mit der Zugriffsebene **protected** (ECMA 2022, S. 413).

- Wie bei einer gewöhnlichen Methodendefinition ist eine Parameterliste anzugeben, ggf. eine leere. Parameter erlauben das individuelle Initialisieren der Instanzvariablen von neuen Objekten.
- Sobald man einen expliziten Konstruktor definiert hat, steht der Standardkonstruktor *nicht* mehr zur Verfügung. Ist weiterhin ein parameterfreier Konstruktor gewünscht, dann muss dieser *zusätzlich* explizit definiert werden.
- Der Compiler fügt bei *jedem* Konstruktor automatisch IL-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 (vgl. Abschnitt 5.3.5) ist also auch bei Konstruktoren erlaubt.
- Konstruktoren können (bis auf zwei Ausnahmen, siehe unten) 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 individuellen Initialisierung aller Instanzvariablen:[1]

```
public Bruch(int zpar, int npar, string epar) {
  Zaehler = zpar;
  Nenner  = npar;
  Etikett = epar;
}
```

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 andere Methode einer Klasse muss 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.[2]

Wenn weiterhin auch ein parameterfreier Konstruktor verfügbar sein soll, dann muss dieser explizit definiert werden, z. B. mit einem leeren Anweisungsteil:

```
public Bruch() {}
```

Im folgenden Programm werden beide Konstruktoren eingesetzt:

---

[1] Ein Visual Studio - Projekt mit der aktuellen Entwicklungsstufe der Klasse `Bruch` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Bruch\b2 Konstruktoren

[2] Eine sinnvollere Reaktion auf den Versuch, ein defektes Objekt zu erstellen, besteht darin, im Konstruktor eine sogenannte *Ausnahme* zu werfen und dadurch den Aufrufer über das Scheitern seiner Absicht zu informieren. Mit der Kommunikation über Ausnahmeobjekte werden wir uns im Kapitel 13 beschäftigen.

Quellcode	Ausgabe

`var b1 = new Bruch(1, 2, "b1");`
`var b2 = new Bruch();`
`b1.Zeige();`
`b2.Zeige();`
	`1`
	`b1 = -----`
	`2`
	
	`0`
	`-----`
	`1`

Als Ausnahme von der Regel, dass Konstruktoren nur über den **new**-Operator zu verwenden sind, darf man zwischen der Parameterliste und dem Anweisungsblock eines Konstruktors einen anderen Konstruktor derselben Klasse über das Schlüsselwort **this** aufrufen, z. B.:

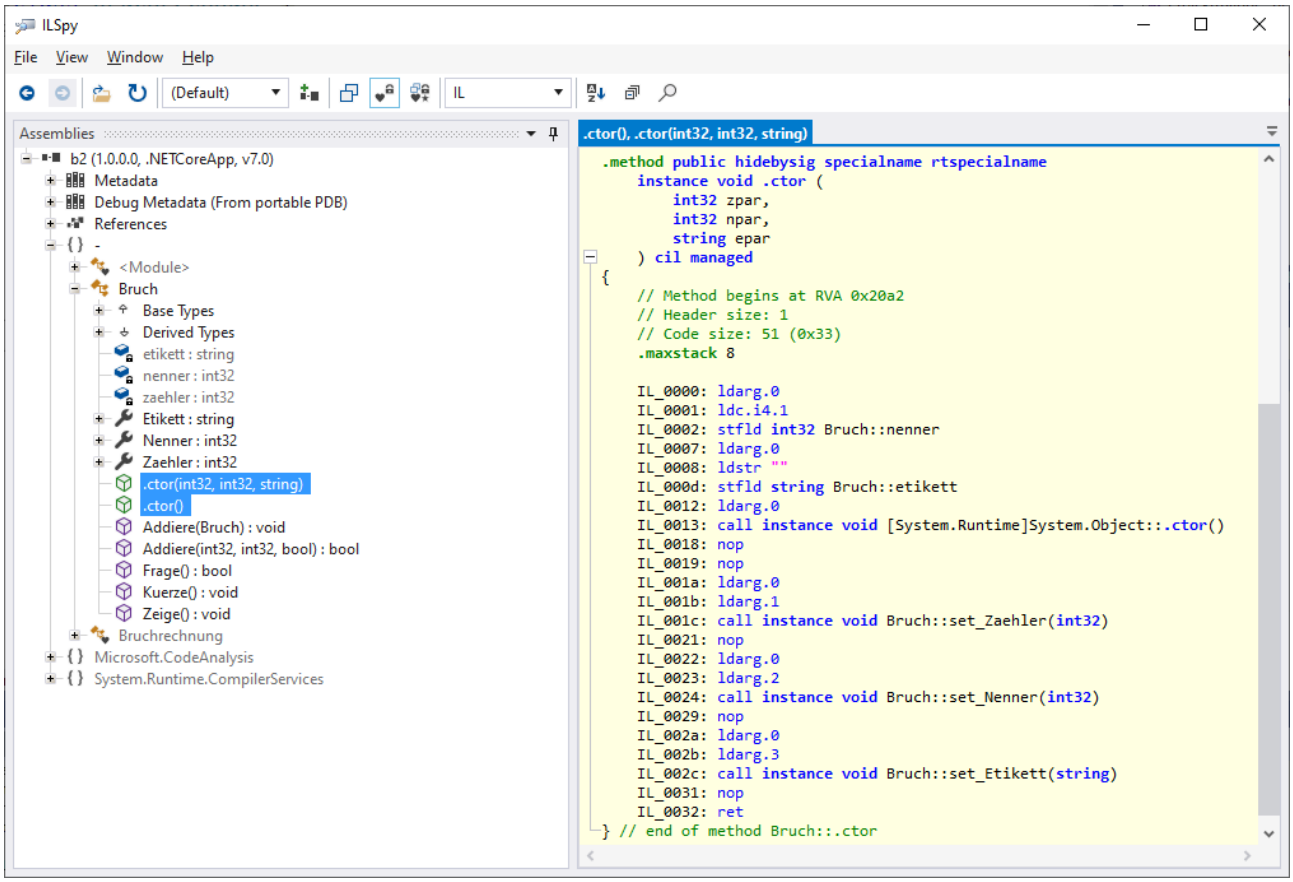
```
public Bruch() : this(0, 1, "unbekannt") {}
```

Die Anweisungen des gerufenen Konstruktors werden zuerst ausgeführt. Im Beispiel beschränkt sich der rufende Konstruktor darauf, dem (aufwändiger parametrisierten) Kollegen die Arbeit zu überlassen.

Im Zusammenhang mit der Vererbung werden wir noch die Möglichkeit kennenlernen, in einem Konstruktor über das Schlüsselwort **base** einen Konstruktor der Basisklasse aufzurufen (siehe Abschnitt 7.4), z. B.:

```
public Kreis(double x, double y, double rad) : base(x, y) { ... }
```

Wie das Hilfsprogramm ILSpy für die aktuelle Entwicklungsstufe des Bruchrechnungs-Assemblies zeigt, führen die Konstruktoren einer Klasse unter dem Namen **.ctor** die Liste der Instanzmethoden an:



5.4.3.1.3 Vereinfachte Deklaration mit Initialisierung

Wie Sie bereits aus dem Abschnitt 4.3.7 wissen, kann man bei der initialisierenden Deklaration von Referenzvariablen die doppelte Nennung des Klassennamens vermeiden. Für *lokale* Referenzvariablen von Methoden gelingt das schon seit C# 3 über das Schlüsselwort **var** und die implizite Typisierung, z. B.:

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

Während die implizite Typisierung für Felder verboten ist, steht der mit C# 9 eingeführte zieltypisierte **new**-Ausdruck für lokale Variablen *und* für Felder zur Verfügung, z. B.:

```
Bruch b = new();
```

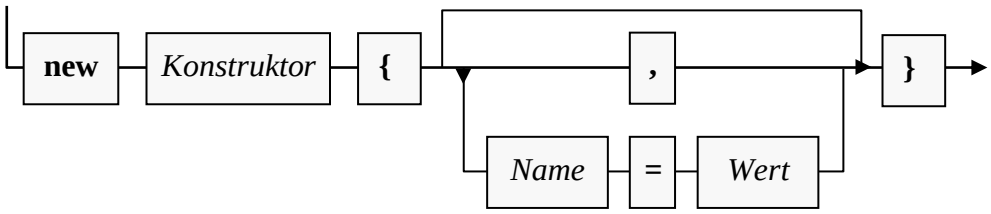
Dass der Name des Konstruktors weggelassen werden darf, gilt nicht nur für den parameterfreien Standardkonstruktor, sondern für *jede* Konstruktorüberladung, z. B.:

```
Bruch b = new(1, 2, "1/2");
```

5.4.3.2 Objektinitialisierer

Öffentliche Felder und Eigenschaften (siehe Abschnitt 5.5) 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, die man als *Objektinitialisierer* bezeichnet.

Objektinitialisierer mit Konstruktor



In der `Bruch`-Klassendefinition könnte man auf den parametrisierten Konstruktor,

```
public Bruch(int zpar, int npar, string epar) {
   Zaehler = zpar;
   Nenner  = npar;
   Etikett = epar;
}
```

der sich auf das Initialisieren der Felder beschränkt, verzichten und stattdessen Objektinitialisierer verwenden, wobei allerdings der Schreibaufwand beim Erstellen von `Bruch`-Objekten steigen würde, z. B.:

Quellcode	Ausgabe

`var b = new Bruch() { Zaehler = 3,`
`  Nenner = 7,`
`  Etikett = "b" };`
`b.Zeige();`
	3
	b = -----
	7

Auch bei Verwendung eines Objektinitialisierers kommt ein Konstruktor zum Einsatz, wobei meist der *parameterfreie* Konstruktor verwendet wird. Dessen leere Parameterliste darf weggelassen werden, z. B.:

```
var b = new Bruch { Zaehler = 3, Nenner = 7, Etikett = "b" };
```

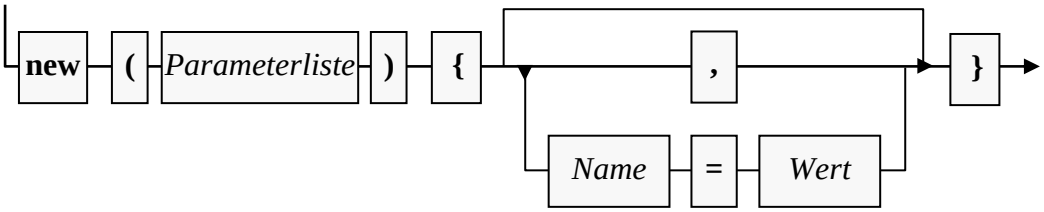
Um den Wert einer Eigenschaft (bzw. eines Felds) in einem Objektinitialisierer festzulegen, darf man statt eines Literals auch einen Ausdruck verwenden, z. B.:

```
Bruch b1 = new Bruch(1, 2, "b1");
Bruch b2 = new Bruch { Nenner = 7, Zaehler = b1.Zaehler, Etikett = "b2" };
```

Außerdem ist bei der *namensorientierten* Objektinitialisierung die Reihenfolge der Felder beliebig, während im *positionsorientierten* Konstruktoraufruf die Reihenfolge aus der Konstruktordefinition einzuhalten ist (engl.: *positional* vs. *nominal creation*).

Ein Objektinitialisierer lässt sich auch mit einem zieltypisierten **new**-Ausdruck kombinieren:

Objektinitialisierer mit zieltypisiertem new-Ausdruck



Im folgenden Beispiel wird die Kombination aus einem zieltypisiertem **new**-Ausdruck und einem Objektinitialisierer für eine lokale Variable verwendet:

Quellcode	Ausgabe

`Bruch b = new() { Zaehler = 3, Nenner = 7, Etikett = "b" };`
`b.Zeige();`
	`3`
	`b = -----`
	`7`

Durch die Objektinitialisierer werden Konstruktoren keinesfalls überflüssig, denn Konstruktoren können ...

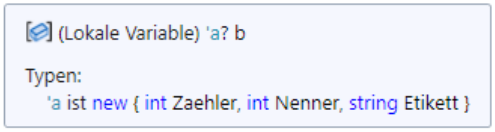
- beliebige Initialisierungsarbeiten ausführen und dazu Methoden aufrufen,
- Konsistenzbedingungen sicherstellen.

Nachdem bisher Schreibvarianten zu sehen waren, die alle zu einem `Bruch`-Objekt führten, folgt nun eine „Schreibvariante“, die ein Objekt *einer anderen* Klasse erzeugt. Gibt man …

- bei der Deklaration einer lokalen Referenzvariablen
- mit einer durch das Schlüsselwort **var** angeforderten impliziten Typisierung
- nach dem Schlüsselwort **new** weder den Namen, noch die Parameterliste eines Konstruktors an,

dann resultiert ein Objekt einer sogenannten *anonymen Klasse* (siehe Abschnitt 6.5), z. B.

```
var b = new { Zaehler = 3, Nenner = 7, Etikett = "b" };
```



Das im letzten Beispiel erzeugte Objekt namens `b` …

- besitzt die **int**-Eigenschaften `Zaehler` und `Nenner` sowie die **String**-Eigenschaft `Etikett`,
- wobei für alle Eigenschaften ein lesender Zugriff erlaubt ist, aber kein schreibender.

Außerdem besitzt die anonyme Klasse nur die von der Urahnklasse **Object** geerbten Methoden.

5.4.4 Objektreferenzen verwenden

In diesem Abschnitt geht es um Wertparameter und Methodenrückgaben mit Referenztyp sowie um das Schlüsselwort **this**, mit dem man in einer Methode das aktuell handelnde Objekt ansprechen kann.

5.4.4.1 Objektreferenzen als Wertparameter

Wie Sie bereits aus dem Abschnitt 5.3.1.3.1 wissen, funktionieren die formalen Wertparameter einer Methode wie *lokale Variablen*, die beim Methodenaufruf mit den Werten der Aktualparameter initialisiert werden. Methodeninterne Änderungen dieser lokalen Variablen wirken sich *nicht* auf die eventuell als Aktualparameter 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, sodass die gerufene Methode über ihre lokale Referenzvariable (über den Wertparameter mit Referenztyp) 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 bekanntgegebenen Bruch zum eigenen Wert zu addieren und das Resultat gleich zu kürzen.

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 weder „zirkulär“ noch ungewöhnlich. Es ist vielmehr unvermeidlich, wenn `Bruch`-Objekte miteinander kooperieren sollen.

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 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 Wertparameter vom Typ `Bruch` bestimmt wird:[2]

```
public void DuplWerte(Bruch bc) {
   bc.zaehler = zaehler;
   bc.nenner = nenner;
}
```

Im folgenden Programm wird das `Bruch`-Objekt `b1` beauftragt, die `DuplWerte()` - Methode auszuführen, wobei als Parameter eine Referenz auf das Objekt `b2` übergeben wird:

---

[1] Wertparameter mit Referenztyp arbeiten also analog zu den Referenzparametern (vgl. Abschnitt 5.3.1.3.2), insofern als beide einer gerufenen 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 den Instanzen von Werttypen zu ermöglichen. Wir werden mit den sogenannten Strukturen noch Werttypen kennenlernen, die ähnlich umfangreich sein können wie Klassen.

[2] Ein Visual Studio - Projekt mit der aktuellen Entwicklungsstufe der Klasse `Bruch` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Bruch\b3 DuplWerte

Quellcode	Ausgabe

`var b1 = new(1, 2, "b1");`
`var b2 = new(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`

5.4.4.2 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);
}
```

Im folgenden Beispiel wird das durch `b2` referenzierte `Bruch`-Objekt in der vom Objekt `b1` ausgeführten Methode `Klone()` erstellt:

Quellcode	Ausgabe

`var b1 = new(1, 2, "b1");`
`b1.Zeige();`
`var b2 = b1.Klone();`
`b2.Zeige();`
	`  1`
	`b1 = -----`
	`  2`
	
	`  1`
	`b1 = -----`
	`  2`

5.4.4.3 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. Das ist mit dem Schlüsselwort **this** möglich, das innerhalb einer Instanzmethode wie eine Referenzvariable funktioniert. Im folgenden Beispiel ermöglicht die **this**-Referenz die Verwendung von Formalparameternamen, die mit den Namen von Instanzvariablen übereinstimmen:

---

[1] Bei einer für die breitere Öffentlichkeit gedachten Klasse sollte auch eine die Schnittstelle **ICloneable** (siehe Kapitel 9 über Schnittstellen bzw. Interfaces) implementierende Vervielfältigungsmethode angeboten werden, obwohl diese Schnittstelle durch semantische Unklarheit von begrenztem Wert ist, was im Kapitel 9 näher erläutert wird.
Ein Visual Studio - Projekt mit der aktuellen Entwicklungsstufe der Klasse `Bruch` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Bruch\b4 Klone

```
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 kennenlernen.

5.4.5 Überflüssige Objekte abräumen

Am Ende von Abschnitt 5.4 geht es darum, was mit Objekten nach dem erfolgreichen Einsatz geschehen soll. Das Bemühen um nachvollziehbare Erklärungen hat zu vielen Details geführt, die möglicherweise die Motivation zur Beschäftigung mit Altlasten überstrapazieren. Eilige Leser können sich vorläufig auf die folgende Kurzform beschränken:

- Im Abschnitt 5.4.5.1 wird beschrieben, dass in C# der von obsolet gewordenen Objekten belegte Speicher vom sogenannten *Garbage Collector* automatisch freigegeben wird. Mit diesem Thema müssen sich Einsteiger noch nicht beschäftigen.
- Im Abschnitt 5.4.5.2 geht es um die Freigabe von sogenannten *unverwalteten Ressourcen* (z. B. Datei-, Netzwerk- oder Datenbankverbindungen). Werden in einer Klasse solche Ressourcen über Instanzvariablen repräsentiert, dann sind Methoden zur Freigabe der Ressourcen erforderlich. Mit diesem Thema muss man sich erst dann beschäftigen, wenn man unverwaltete Ressourcen auf die beschriebene Weise verwendet.

5.4.5.1 Speicherfreigabe per Garbage Collector

Ist ein Objekt nicht mehr über eine Referenz bzw. eine Sequenz von Referenzen erreichbar, dann wird es bei passender Gelegenheit (z. B. bei Speichermangel) vom **Garbage Collector** (**GC**) (dt.: *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 möglich (und normal), dass ein Objekt die erzeugende Methode überlebt, weil eine Referenz nach Außen transportiert worden ist (z. B. per Rückgabewert, vgl. Abschnitt 5.4.4.2).

Erreichbar ist ein Objekt genau dann, wenn von einer sogenannten *Wurzel* (engl.: *root*) der Anwendung eine Serie von Referenzen startet, die zum Objekt führt. Zu den Wurzeln einer Anwendung gehören:

- Statische Felder
- Lokale Variablen auf dem Stack
- CPU-Register (ein Speicher im Prozessor)
- FReachable-Warteschlange des Garbage Collectors (siehe Abschnitt 5.4.5.2)

Wenn sich z. B. zwei Objekte gegenseitig referenzieren, die aber nicht über eine Wurzel erreichbar sind, dann gelten die beiden Objekte als unerreichbar und damit obsolet.

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 freigegeben, sobald die Ausführung der Methode endet. Demgegenüber muss der von überflüssig gewordenen *Objekten* belegte Speicher bei älteren Programmiersprachen (z. B. C++) nach Gebrauch explizit freigegeben werden. In Anbetracht der Objektmassen, die manche Programme (z. B. ein Grafikeditor) benötigen, ist einiger Aufwand erforderlich, um eine Verschwendung von Speicherplatz zu verhindern. Mit seinem automatischen Garbage Collector vermeidet C# lästigen Aufwand und zwei kritische Fehler:

- Weil der Programmierer keine Verpflichtung (und Berechtigung) zum Entsorgen von Objekten hat, kann es nicht zu Programmabstürzen durch den Zugriff auf voreilig entsorgte Objekte kommen.
- Es entstehen keine **Speicherlöcher** (engl.: *memory leaks*) durch versäumte Freigaben des Speichers, den überflüssig gewordene Objekte belegen. Durch Speicherlöcher wird das Erzeugen neuer Objekte erschwert und schließlich verhindert.

Über vergessene Referenzen kann trotz Garbage Collector auch in einem C# - Programm eine Speicherverschwendung auftreten (siehe Albahari 2022, S. 576ff), z. B.

- Ein Objekt hat eine Behandlungsmethode bei einem Ereignis registriert (siehe Abschnitt 10.2).
- Ein Objekt hat eine regelmäßig auszuführende Methode bei einem Timer registriert (siehe Abschnitt 17.3 in Baltes-Götz (2021)).

Solche Objekte sind von einer Wurzel aus erreichbar, sodass der Garbage Collector sie nicht abräumen kann.

Der Garbage Collector wird spontan anlässlich einer Objektkreation per **new**-Operator tätig, wenn eine von den folgenden Bedingungen erfüllt ist:

- Das kumulative Volumen von neuen Objekten (aus der Generation 0, siehe unten) übersteigt eine festgelegte Grenze.
- Das Betriebssystem meldet einen allgemeinen Speichermangel.

Es ist für Entwickler nicht vorhersehbar, wann und in welcher Reihenfolge obsolete Objekte im Rahmen einer spontanen Tätigkeit des Garbage Collectors entsorgt werden. Es ist noch nicht einmal garantiert, dass der Garbage Collector überhaupt spontan tätig wird.[1]

Man kann die spontane Tätigkeit des Garbage Collectors ergänzen durch explizit angeforderte Einsätze, die mit Hilfe der statischen Methode **GC.Collect()** angefordert werden. So lässt sich z. B. eine Phase mit bekanntermaßen geringer Programmauslastung dazu verwenden, um den von obsoleten Objekten belegen Speicher zurückzugewinnen. Dieser Aufruf sollte wegen des erheblichen Zeitaufwands mit Bedacht eingesetzt werden.

Der Garbage Collector arbeitet per Voreinstellung in einem eigenen Thread (parallelen Ausführungspfad), beeinträchtigt aber die Leistung und vor allem die Reaktivität einer Anwendung. Während seiner Tätigkeit müssen die anderen Ausführungspfade der Anwendung angehalten werden. Daher werden die anschließend (vereinfachend) beschriebenen Maßnahmen zur Leistungsoptimierung ergriffen.[2]

Es werden drei **Generationen von Objekten** differenziert behandelt:

---

[1] https://ericlippert.com/2015/05/18/when-everything-you-know-is-wrong-part-one/

[2] https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

- Generation 0
  Dazu gehören *neue* Objekte, die noch keinen GC-Einsatz erlebt haben. Das kumulative Volumen der Gen0-Objekte wird unter einer laufend optimierten Schwelle gehalten (ca. zwischen 256 KB and 4 MB).[1] Ist diese Grenze überschritten, findet eine Gen0-Prüfung statt, sobald die Anwendung ein weiteres Objekt anlegen möchte. Eine Gen0-Prüfung benötigt wenig Zeit und besitzt eine hohe Erfolgsquote, weil sich unter den jungen Objekten viele obsolete befinden. Mit steigender Überlebensrate bei der Gen0-Prüfung wird das zulässige kumulative Volumen der Gen0-Objekte erhöht.
- Generation 1
  Dazu gehören Objekte, die eine Gen0-Prüfung überstanden haben.
- Generation 2
  Dazu gehören Objekte, die eine Gen1-Prüfung überstanden haben oder im Heap-Bereich für *große* Objekte abgelegt wurden (siehe unten).

Mit dem Ziel, durch möglichst wenig Aufwand (Belastung für die Anwendung) möglichst viel Speicher zurückzugewinnen, konzentriert sich der Garbage Collector auf die jüngeren Objekte. Wenn durch eine Gen0-Prüfung nicht genügend Speicher zurückgewonnen wurde, dann führt der GC eine Gen1-Prüfung durch. Steht anschließend immer noch nicht genügend Speicher für neue Objekte zur Verfügung, dann findet eine zeitaufwändige Gen2-Prüfung statt.

Durch das Löschen obsoleter Objekte entsteht ein fragmentierter Speicher, der das Erzeugen neuer Objekte behindern würde. Daher werden die lebendigen Objekte zusammengeschoben, um einen **kompaktifizierten Speicher** zu erzielen.

Objekte mit einer Größe von mindestens 85.000 Bytes werden …

- auf dem sogenannten **Large Object Heap** (**LOH**) abgelegt,
- grundsätzlich in die Generation 2 aufgenommen
- und wegen des zu großen Aufwands beim Kompaktifizieren *nicht* verschoben.

Die kleinen Objekte werden auf dem **Small Object Heap** (**SOH**) abgelegt.

5.4.5.2 Finalisierer und Ressourcen-Freigabe

Sollen die Objekte einer Klasse vor dem Entfernen aus dem Speicher noch Aufräumaktionen durchführen, dann ist als Gegenstück zu den Konstruktoren ein sogenannter **Finalisierer** zu definieren, der ggf. vom Garbage Collector aufgerufen wird. Er trägt denselben Namen wie die Klasse, wobei das Tilde-Zeichen (~) voranzustellen ist.

Die Existenz eines Finalisierers steigert den Aufwand des Garbage Collectors für die Objekte einer Klasse. Er muss jedes Objekt einer solchen Klasse in seine Finalisierungs-Warteschlange aufnehmen. Wird bei einer GC-Prüfung ein obsoletes und zu finalisierendes Objekt festgestellt, dann wird es nicht aus dem Speicher entfernt, sondern von der Finalisierungs-Warteschlange verschoben in die sogenannte *FReachable-Warteschlange*, die von einem speziellen Thread des Garbage Collectors abgearbeitet wird (Marshall & Bruno 2009, S. 152). Nach einer erfolgreichen Finalisierung wird das obsolete Objekt aus der FReachable-Warteschlange entfernt, sodass es bei der nächsten GC-Prüfung mit Beteiligung seiner Generation abgeräumt werden kann.

---

[1] Die Angaben zur Größe des Gen0-Heaps stammen von der Webseite:
https://www.methodpark.de/blog/garbage-collection-in-net-part-1-4/
Ein KB enthält 1000 Bytes, und ein MB enthält 1000 KB. Manchmal wird ein KB mit 1024 (= 2^{10}) Bytes und ein MB mit 1024 KB gleichgesetzt.

Die wichtigste Tätigkeit eines Finalisierers ist die Freigabe von Ressourcen, die *nicht* von der CLR verwaltet werden (z. B. Datei-, Netzwerk- oder Datenbankverbindungen). Weil wir im Manuskript solche Ressourcen noch nicht verwendet haben, beschränken wir uns auf ein inhaltsfreies Beispiel, das die Tätigkeit des Garbage Collectors dokumentiert. In der **Main()** - Methode der Startklasse `FinalDemo` wird ein Objekt der Klasse `K2` erzeugt, die von der Klasse `K1` abstammt (zur Vererbung siehe Kapitel 7). Die Objekte bestehen im Wesentlichen aus einem **int**-Array mit 15.000 Elementen, der 60 KB belegt. Die Gesamtgröße eines Objekts bleibt unter 85.000 Bytes, sodass es vom Garbage Collector der zuerst überprüften Generation 0 zugeordnet wird (vgl. Abschnitt 5.4.5.1). Auf einem Testrechner mit .NET 7 waren 33 Objekte erforderlich (zusammen ca. 2 MB), um eine Gen0-Prüfung zuverlässig auszulösen:[1]

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
using System;

class K1 {
    int[] ia = new int[15_000];
    ~K1() { Console.WriteLine("K1-Finalisierer"); }
}

class K2 : K1 {
    ~K2() { Console.Write("K2-Finalisierer, "); }
}

class FinalDemo {
    void UseK2() { K2 k2 = new(); }

    static void Main() {
        FinalDemo kd = new();
        Console.WriteLine("Vorher  " +
          GC.GetTotalMemory(false));
        for (int i = 0; i < 33; i++)
            kd.UseK2();
        Console.WriteLine("Nachher " +
          GC.GetTotalMemory(false));
        Console.ReadLine();
    }
}
```

</td><td>

```
Vorher  119600
Nachher 2120400
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
K2-Finalisierer, K1-Finalisierer
```

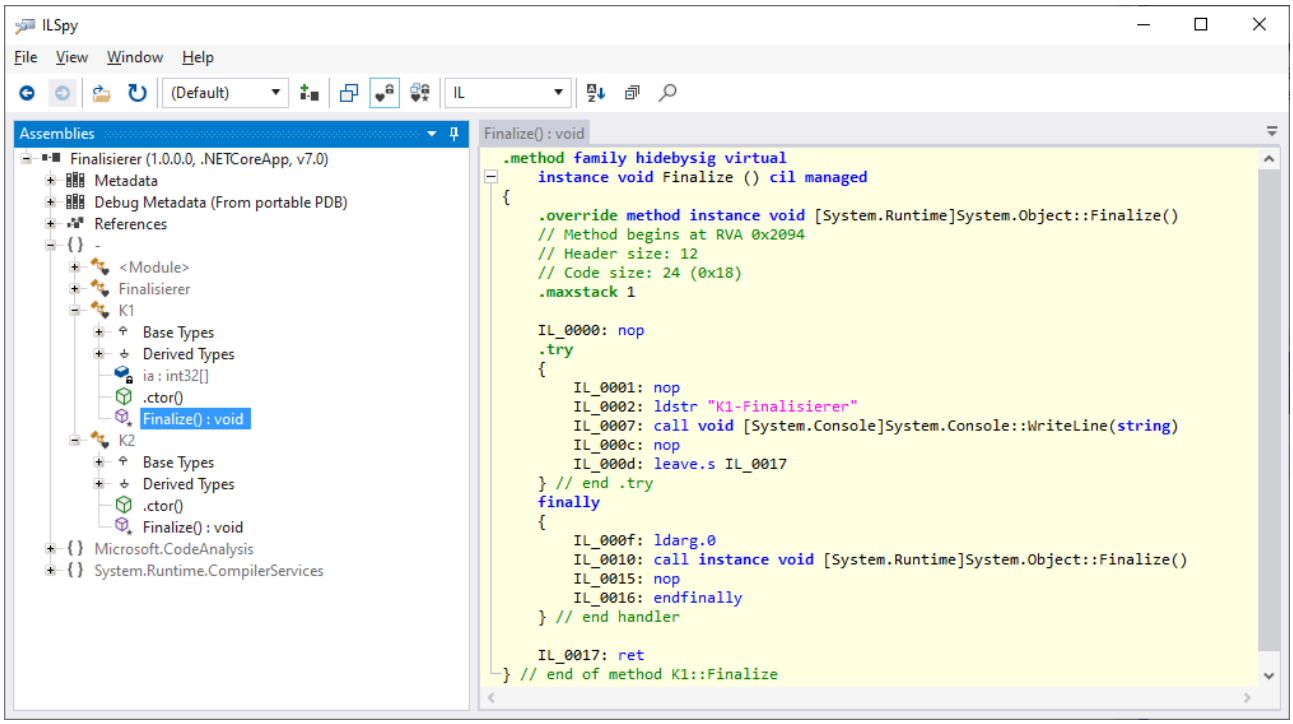
</td></tr>
</table>

In .NET Core und .NET mit einer Version ab 5 findet (anders als im .NET Framework) bei der Beendigung einer Anwendung für die noch nicht vom Garbage Collector abgeräumten Objekte *keine Finalisierung statt*. Daher enthält die **Main()** - Methode am Ende einen **ReadLine()** - Aufruf, um die Anwendung am Leben zu halten, damit der Garbage Collector tätig werden und die Finalisierer der obsoleten Objekte aufrufen kann.

Für ein Objekt werden alle Finalisierer in seinem Stammbaum aufgerufen. Vom Finalisierer der Urahnklasse **Object** ist in der Ausgabe des Beispielprogramms nichts zu sehen, weil er keine Ausgabe vornimmt (und auch sonst nichts tut).

Bei einer Assembly-Inspektion mit dem Hilfsprogramm ILSpy stellt man fest, dass der Finalisierer den Namen **Finalize()** trägt:

[1] Ein Visual Studio - Projekt mit dem Projekt `Finalisierer` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Finalisierer



Aus der vom Programmierer definierten Destruktionsmethode, die den Namen der Klasse mit einem vorangestellten Tilde-Zeichen (~) trägt, erstellt der Compiler die Methode **Finalize()** und fügt dabei den Aufruf der Basisklassenvariante ein.[1]

Solange eine Klasse nur Speicher belegt, benötigt sie keinen Finalisierer. 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. Für die weiteren Erläuterungen ist ein Vorgriff auf das Thema *Interfaces* erforderlich, der aber keine großen Schwierigkeiten bereiten dürfte (siehe Kapitel 9). Man verwendet eine unverwaltete Ressource (z. B. eine Datei) in aller Regel über ein Objekt aus einer BCL-Klasse (z. B. **FileStream**, siehe Abschnitt 16.1 in Baltes-Götz (2021)), die das Interface **IDisposable** erfüllt, also eine Methode namens **Dispose()** anbietet, um die unverwaltete Ressource freizugeben. Solange man das BCL-Objekt in einer Methode verwendet und anschließend die **Dispose()** – Methode aufruft, muss man in der eigenen Klasse das Interface **IDisposable** *nicht* implementieren. Über die **using**-Anweisung kann der **Dispose()** – Aufruf implizit und ohne Aufwand erfolgen, z. B.:

```
using (StreamReader sr = new(
                         new FileStream("text.txt", FileMode.Open, FileAccess.Read)))
    for (int i = 0; sr.Peek() >= 0; i++)
        Console.WriteLine($"{i}:\t{sr.ReadLine()}");
```

Im Manuskript wird der korrekte Aufruf der **Dispose()** - Methoden von BCL-Klassen noch mehrfach ein wichtiges Thema sein, während sich keine Notwendigkeit zur Definition eines Finalisierers zu einer eigenen Klasse ergeben wird.

Ein sicheres Indiz für die Notwendigkeit, in einer eigenen Klasse das Interface **IDisposable** zu implementieren, liegt dann vor, wenn ein Feld auf ein Objekt aus einer Klasse zeigt, die das Interface **IDisposable** implementiert (Albahari 2022, S. 578).

Das Interface **IDisposable** verlangt von einer implementierenden Klasse eine Methode namens **Dispose()**, die alle von einem Objekt verwendeten unverwalteten Ressourcen freigibt. Nutzer der Klasse sollten die **Dispose()** - Methode aufrufen, wenn ein Objekt nicht mehr benötigt wird und so

[1] https://www.oreilly.com/library/view/programming-net-components/0596102070/ch04s04.html

für eine sofortige und zuverlässige Freigabe der Ressourcen sorgen (ohne Abhängigkeit vom Garbage Collector).

Um für den Fall vorzusorgen, dass es ein Benutzer der Klasse den **Dispose()** – Ausruf unterlässt, ist eine Finalisierungsmethode erforderlich, die auf Veranlassung des Garbage Collectors aufgerufen wird. Die Definition einer Finalisierungsmethode ist u. a. deshalb komplex und fehleranfällig, weil der Garbage Collector nicht-deterministisch arbeitet und selbständig über Zeitpunkt und Reihenfolge seiner Aufräumaktionen entscheidet. Bei der von Microsoft empfohlenen Vorgehensweise kommt die Definition eines *eigenen* Finalisierers nur als letzter Ausweg vor:[1]

- Nach Möglichkeit sollten unverwaltete Ressourcen in ein Objekt aus einer Ableitung der Klasse **SafeHandle** im Namensraum **System.Runtime.InteropServices** verpackt werden, weil diese Klassen über eine robuste, auch unter ungünstigen Bedingungen korrekt arbeitende Finalisierungsmethode verfügen.
- Nur wenn keine **SafeHandle** - Verpackung für unverwaltete Ressourcen möglich ist, sollte ein Klassendesigner eine eigene Finalisierungsmethode implementieren und darin **Dispose()** aufrufen.

Ist die Definition eines Finalisierers nicht zu vermeiden, dann sind die folgenden Regeln zu beachten:[2]

- Ein Finalisierer trägt denselben Namen wie die Klasse, wobei das Tilde-Zeichen (~) voranzustellen ist.
- Ein Finalisierer liefert grundsätzlich *keinen* Rückgabewert, und es wird bei der Definition *kein* Typ angegeben.
- Pro Klasse ist nur *ein* Finalisierer erlaubt. Dieser muss eine leere Parameterliste haben.
- Finalisierer können nicht direkt aufgerufen werden. Es ist ausschließlich der automatische Aufruf durch den Garbage Collector vorgesehen.
- Es ist nicht garantiert, dass der Garbage Collector tätig wird und einen Finalizer aufruft.
- Bei der Definition eines Finalisierers sind Modifikatoren überflüssig und verboten.
- Ein Finalisierer sollte eine möglichst kurze Laufzeit besitzen und darf auf keinen Fall blockierende Methodenaufrufe vornehmen (z. B. durch das Warten auf einen anderen Thread oder einen Dateizugriff), weil in dieser Lage in der Anwendung keine Finalisierungen mehr möglich sind.
- Finalisierer werden nicht vererbt.

Der C# - Insider Eric Lippert warnt auf einer Stackoverflow-Seite:[3]

> Implementing a destructor correctly is one of the hardest things to do in C# in all but the most trivial cases.

5.5 Eigenschaften

Die Eigenschaften von C# bieten zur Verwaltung des Zustands von Objekten und Klassen eine gelungene Kombination von Feldern und Methoden. Während die Benutzer einer Klasse bei der Wertabfrage oder -zuweisung dieselbe Syntax wie bei Feldern nutzen können, haben die Designer einer Klasse die Flexibilität von Methoden zur Verfügung, um z. B. …

---

1 https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/unmanaged

2 https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/finalizers

3 https://stackoverflow.com/questions/8174347/inheritance-and-destructors-in-c-sharp

- einen abgefragten Wert aus einer Datenbank oder von einem Webdienst zu beschaffen,
- vor der Ausführung einer Wertzuweisung den beantragten Wert zu validieren,
- aufgrund einer Wertzuweisung ein Ereignis auszulösen.

5.5.1 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 entsteht 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 sind ein **get**- und ein **set**-Block mit naheliegender Syntax zu implementieren (siehe Syntaxdiagramm im Abschnitt 4.1.3.3).[1] In der Regel steckt hinter einer öffentlich zugänglichen, also zum API der Klasse gehörenden, Eigenschaft ein privates Feld, das in der englischen Literatur als *backing field* bezeichnet wird.

Erwähnenswert ist, dass im **set**-Block der gewünschte neue Wert ohne Formalparameterdefinition über das Schlüsselwort **value** angesprochen wird.

Für den Klassen*anwender* ändert sich mehr, weil er eine Eigenschaft intuitiver verwenden kann als korrespondierende Zugriffsmethoden, z. B.:

```
b1.Nenner = 2;
Console.WriteLine(b1.Nenner);
```

---

[1] Das Syntaxdiagramm im Abschnitt 4.1.3.3 verschweigt die Möglichkeit, in einem **get**-Block neben der obligatorischen **return**-Anweisung noch weitere Anweisungen unterzubringen, z. B.:
`public double X { get { nacc++; return x; } set { x = value; } }`
Der Zustand des befragten Objekts sollte dabei aber nicht geändert werden, siehe:
https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/using-properties

Sogar die Aktualisierungsoperatoren (vgl. Abschnitt 4.5.8) werden unterstützt, z. B.:

```
b1.Nenner += 2;
```

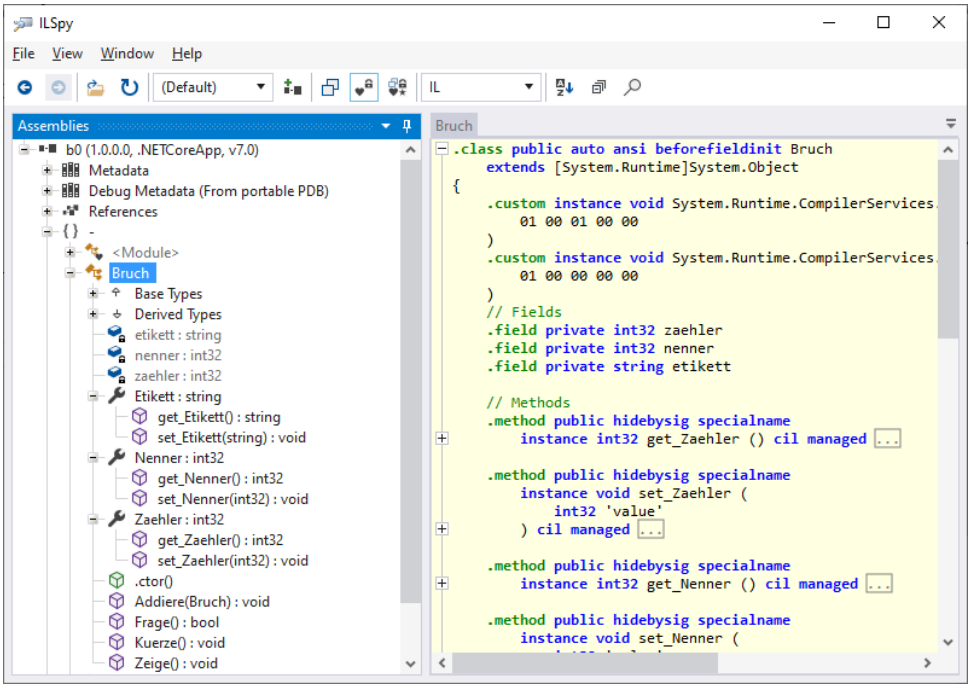
Um eine **get-only** - oder **set-only** - **Eigenschaft** zu realisieren, verzichtet man auf die **set**- bzw. **get**-Implementation. Durch die folgende Variante der `Nenner`-Definition aus der Klasse `Bruch` entsteht eine Eigenschaft, die nur den lesenden Zugriff erlaubt, wobei die klasseneigenen Methoden nach wie vor auf die private Instanzvariable `nenner` schreibend zugreifen können:

```
public int Nenner {
    get {
        return nenner;
    }
}
```

Es ist möglich, den für eine Eigenschaft gültigen Zugriffsschutz entweder für den **get**- oder für den **set**-Zugriff zu verschärfen, aber nicht für beide gleichzeitig. 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 Hilfsprogramm ILSpy zeigt, erstellt der Compiler zu den Eigenschaften unserer Klasse `Bruch` jeweils ein Paar von Zugriffsmethoden:



Wenngleich die Eigenschaften sehr praktisch sind, handelt es sich doch um *syntactic sugar* (Mössenböck 2019, S. 3). Eine äquivalente Funktionalität ist mit akzeptablem Aufwand auch in Programmiersprachen ohne Eigenschaften realisierbar (z. B. in Java oder C++).[1]

5.5.2 Automatisch implementierte Eigenschaften

5.5.2.1 Routinearbeit an den Compiler delegieren

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, den Datentyp, den 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` könnten also die Deklaration

```
int zaehler;
```

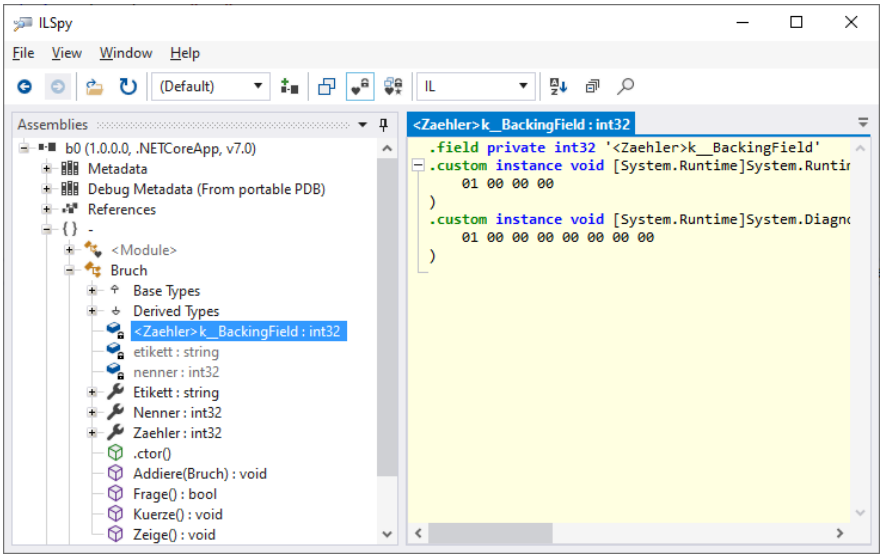
und die Definition

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

äquivalent ersetzt werden durch die Deklaration:

```
public int Zaehler {get; set;}
```

Wie das Hilfsprogramm ILSpy 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 traditionelle **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 eine Ausnahme geworfen werden (siehe Kapitel 13), was natürlich auch in der **set**-Implementation einer Eigenschaft möglich ist.

Dieses Feld ist ausschließlich über die Eigenschaft ansprechbar, sodass bei einem Fehler gute Voraussetzungen für die Ursachensuche bestehen.

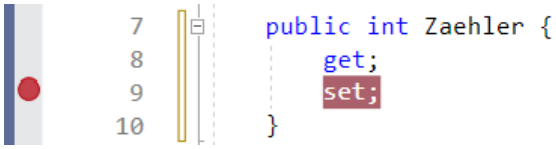
Mit der C# - Version 6.0 sind für automatisch implementierte Eigenschaften (engl. Bezeichnung *Auto-Implemented Properties*) zwei Verbesserungen eingeführt worden:

- 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 (per *Eigenschafts-Initialisierer*, siehe Beispiel) oder in einem Konstruktor, z. B.:
  ```
  public Bsp() {
      Auto = 4711;
  }
  ```
  Bei einer Eigenschaft ohne **set** ist nach der Initialisierung kein weiterer Schreibzugriff mehr möglich.

Im `Bruch`-Einstiegsbeispiel wird der Klarheit halber auf automatisch implementierte Eigenschaften verzichtet.

5.5.2.2 Automatisch implementierte Eigenschaft im Vergleich zu Feldern

Auf den ersten Blick unterscheidet sich eine automatisch implementierte öffentliche Eigenschaft mit **get**- *und* **set**-Methode nur durch den leicht erhöhten Schreibaufwand von einem öffentlichen Feld. Eine genauere Analyse fördert aber doch Vorteile der Eigenschaft zu Tage:[1]

- Eine automatisch implementierte Eigenschaft erleichtert die Fehlersuche. Weil auf ihr backing field nicht direkt zugegriffen werden kann, genügt ein einziger Haltepunkt, um alle Schreibzugriffe beobachten zu können, z. B.:
  

  Wenn hingegen mehrere Methoden auf ein Feld schreibend zugreifen, sind entsprechend viele Haltepunkte erforderlich.
  Im Visual Studio kann man seit der Version 2019 allerdings im Debug-Modus **Datenhaltepunkte** (engl.: *data break points*) setzen, die das Programm stoppen, sobald bei einem bestimmten Objekt ein bestimmtes Feld geändert wird. Dazu setzt man einen Haltepunkt vor die Deklaration des Felds und startet das Debugging. Nach dem Erreichen des Haltepunkts wählt man im **Auto**- oder **Lokal**-Fenster des Debuggers aus dem Kontextmenü zu dem interessierenden Feld eines interessierenden Objekts das Item **Bei Werteänderungen unterbrechen**. Man hat zwei Vorteile gegenüber einem Code-Haltepunkt:
  - Der Stopp ist nicht an eine Quellcode-Stelle gebunden, sondern an eine Instanzvariable. Es ist also gleichgültig, welche Anweisung die Wertveränderung auslöst.
  - Sind mehrere Objekte einer Klasse vorhanden, dann stoppt ein Code-Haltepunkt (und natürlich auch ein **set**-Haltepunkt) bei *jedem* Objekt, während per Datenhaltepunkt ein konkretes Objekt beobachtet wird.

---

[1] Die aufgelisteten und weitere Vorteile von Eigenschaften werden hier beschrieben: https://stackoverflow.com/questions/1180860/public-fields-versus-automatic-properties

Allerdings verursacht ein Datenhaltepunkt beim Debugging einen höheren Zeitaufwand als ein Code-Haltepunkt.

- In einer Schnittstelle (siehe Kapitel 9) kann von einer implementierenden Klasse die Existenz von öffentlichen Eigenschaften verlangt werden, aber nicht die Existenz von öffentlichen Instanzvariablen.

Mit dem Zeitaufwand bei Eigenschafts- bzw. Feldzugriffen beschäftigt sich der Abschnitt 5.5.4.

5.5.3 Objektinitialisierer und init - Setter

Für Eigenschaften mit **set**-Methode ist kein Konstruktor-Parameter zur Initialisierung bei der Objektkreation erforderlich, weil der Objektinitialisierer dazu verwendet werden kann (siehe Abschnitt 5.4.3.2), z. B.:

```
namespace KlassenUndObjekte;
using System;
public class Prog {
    public int Auto { get; set; }
    static void Main() {
        Prog p = new() { Auto = 13 };
        Console.WriteLine(p.Auto);
    }
}
```

Soll eine Eigenschaft nach der Initialisierung *unveränderlich* sein, scheidet die Konstruktor-freie Initialisierung aber aus, weil der Objektinitialisierer nur veränderliche Eigenschaften versorgen kann.

```
public int Auto { get; }
static void Main() {
    Prog p = new() { Auto = 13 };
```

int Prog.Auto { get; }

CS0200: Für die Eigenschaft oder den Indexer "Prog.Auto" ist eine Zuweisung nicht möglich. Sie sind schreibgeschützt.

Seit der C# - Version 9.0 ist dieses Problem behoben durch die **init-only** - Eigenschaften, wobei das Schlüsselwort **set** durch **init** ersetzt wird. Im Beispiel klappt nun die Objektkreation mit Initialisierung, während eine spätere Wertänderung scheitert:

```
public int Auto { get; init; }
static void Main() {
    Prog p = new() { Auto = 13 };

    p.Auto = 4711;
```

Das Schlüsselwort **init** ist nicht nur zur automatischen, sondern auch zur expliziten Eigenschaftsimplementation verwendbar, wobei der **set**-Block durch den **init**-Block ersetzt wird. Im nächsten Beispiel werden die Eigenschaften `FirstName` und `LastName` explizit implementiert unter Verwendung eines **get**- und eines **init**-Blocks:

```
public class Person {
    readonly string firstName = "unbekannt";
    readonly string lastName = "unbekannt";
    readonly bool nameInit;
```

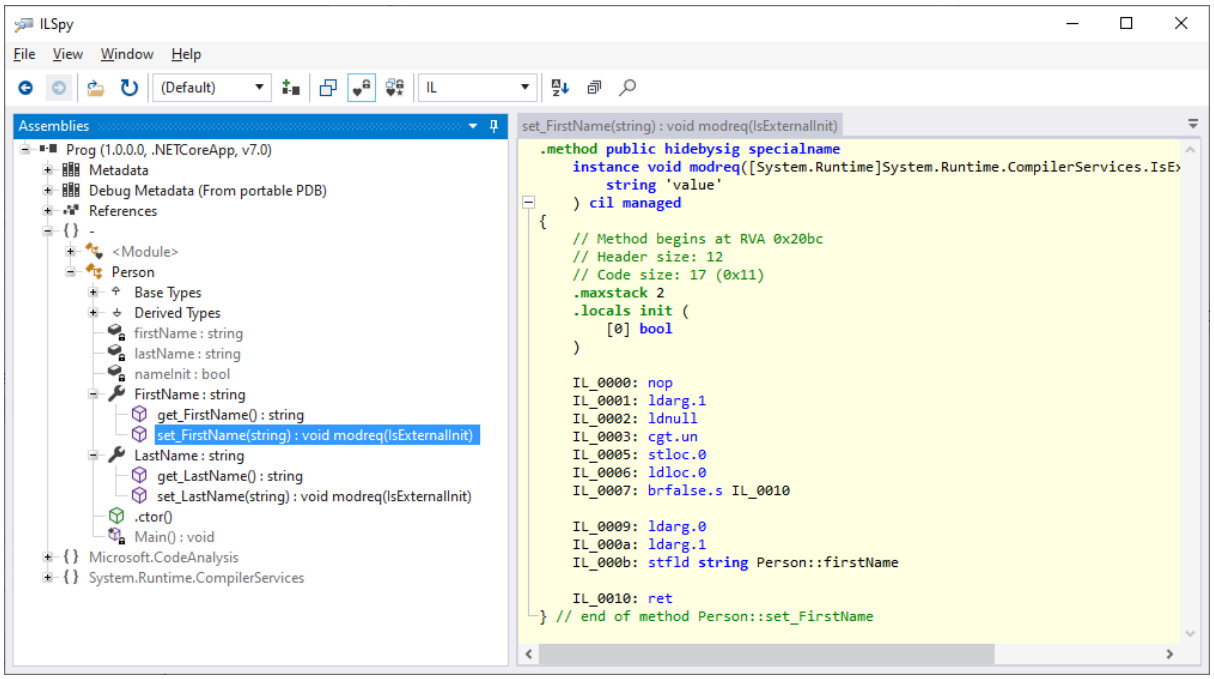
```
        public string FirstName {
            get { return firstName; }
            init {
                if (value != null)
                    firstName = value;
            }
        }

        public string LastName {
            get { return lastName; }
            init {
                if (value != null)
                    lastName = value;
                nameInit = true;
            }
        }
    }
```

Das Beispiel zeigt, dass …

- im **init**-Block der gewünschte Wert (analog zum **set**-Block) über das Schlüsselwort **value** angesprochen wird,
- im **init**-Block (analog zum **get**- und zum **set**-Block) mehrere Anweisungen stehen dürfen.

Weil der **init**-Block nur im Rahmen eines Konstruktors oder eines Objektinitialisierers zum Einsatz kommt, sind hier Zugriffe auf (beliebige) **readonly**-Instanzvariablen erlaubt.

Wie das Hilfsprogramm ILSpy zeigt, wird ein **init**-Block durch eine spezielle **set**-Methode realisiert:



5.5.4 Zeitaufwand bei Eigenschafts- und Feldzugriffen

Eigenschaftszugriffe sind in der Regel *nicht* aufwändiger als Feldzugriffe. 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**. Im Fall einer virtuellen (von abgeleiteten Klassen

überschreibbaren) Eigenschaft (siehe Kapitel 7 zur Vererbung) ist allerdings kein Inlining möglich, weil zur Übersetzungszeit die tatsächlich auszuführende Methode nicht bekannt ist.

Um die Annahme über die erfolgreiche Optimierung des Compilers bei Eigenschaftszugriffen zu verifizieren, wurden im Bruchrechnungsbeispiel für den lesenden und schreibenden Eigenschafts- bzw. Feldzugriff auf den Zähler bzw. Nenner eines Bruchs Vergleichsmessungen unter den folgenden Bedingungen ausgeführt:

- .NET 7
- Windows 10 (64 Bit)
- Rechner mit Intel-CPU Core i3 550
- Erstellungskonfiguration: Release
  Zum Unterschied zwischen Release- und Debug-Konfiguration siehe Abschnitt 3.3.8.3.

Mit einer Ausnahme dauert der direkte Feldzugriff sogar länger als der Eigenschaftszugriff. Von der Ausnahme ist das Schreiben des Nenners betroffen, das beim Eigenschaftszugriff mit Validierung erfolgt, während der Feldzugriff den neuen Wert ungeprüft setzt.

a) Eigenschafts- bzw. Feldzugriff auf den Zähler eines `Bruch`-Objekts:

Zugriffsart	Zeitaufwand für 100 Millionen Zugriffe in Millisekunden	
	via Eigenschaft	direkter Feldzugriff
Lesen	64,0635	64,3607
Schreiben	64,1873	66,1297

b) Eigenschafts- bzw. Feldzugriff auf den Nenner eines `Bruch`-Objekts:

Zugriffsart	Zeitaufwand für 100 Millionen Zugriffe in Millisekunden	
	via Eigenschaft	direkter Feldzugriff
Lesen	64,9500	66,8787
Schreiben	69,2242	67,2364

Ähnliche Ergebnisse finden auch andere Personen.[1]

Schreibende Feldzugriffe auch in klasseneigenen Methoden durch Eigenschaftszugriffe zu ersetzen, bringt ohne Performanz-Beeinträchtigung (in der Release-Konfiguration) einige Vorteile:[2]

- Maßnahmen zur Sicherung der Objektkonsistenz (z. B. Validierung) und zur Reaktion auf Wertveränderungen (z. B. durch das Auslösen von Ereignissen) sind leichter realisierbar.
- Im Multithreading - Betrieb ist die Koordination von mehreren, schreibend zugreifenden Threads leichter realisierbar (siehe Abschnitt 17.1 in Baltes-Götz (2021)).
- Beim Debugging ist man nicht auf einen Datenhaltepunkt angewiesen, sondern kann einen (weniger aufwändigen) Code-Haltepunkt verwenden.
- Dass die Eigenschaften im Unterschied zu den Feldern nicht als **ref**- oder **out**-Parameter verwendbar sind (vgl. Abschnitt 5.3.1.3.2), spielt keine große Rolle. Die Eigenschaften gehören zu Objekten, und die befinden sich auf dem Heap. Die **ref**- oder **out**-Parameter sind hingegen primär für die lokalen Variablen der rufenden Methode gedacht.

Es wird daher explizit empfohlen, schreibende Feldzugriffe auch in klasseneigenen Methoden durch Eigenschaftszugriffe zu ersetzen. Leider ist nur im Fall der automatisch implementierten

---

[1] Siehe z. B.: https://www.jacksondunstan.com/articles/2968

[2] https://stackoverflow.com/questions/10366904/within-the-containing-class-use-property-or-field

Eigenschaften sichergestellt, dass klasseneigene Methoden tatsächlich nicht direkt auf das private backing field zugreifen. Eine konsistente Umsetzung der Empfehlung setzt also Selbstdisziplin voraus.[1]

In mehreren Methoden der Klasse `Bruch` sind aktuell noch schreibende Feldzugriffe zu finden, z. B.:

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

In der folgenden Variante der `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();
}
```

5.6 Initialisierungsverpflichtung für Felder und Eigenschaften

Um für ein Feld oder eine Eigenschaft in einer Klasse oder in einer Struktur (siehe Abschnitt 6.1) dafür zu sorgen, dass bei jeder Instanzkreation eine explizite Initialisierung stattfindet, kann man passend parametrisierte Konstruktoren anbieten und auf einen öffentlichen parameterfreien Konstruktor verzichten. Seit C# 11 steht für denselben Zweck der auf Felder und Eigenschaften anwendbare Modifikator **required** zur Verfügung.

Eine so angeordnete Initialisierungspflicht wirkt sich auf Konstruktoren und Objektinitialisierer aus, wobei die Konsequenzen für Konstruktoren überraschend bis irritierend sind. Der Compiler kann nämlich nicht verifizieren, ob ein Konstruktor die **required**-Member tatsächlich initialisiert und geht generell von einer *fehlenden* Initialisierung aus, sodass der folgende Quellcode *nicht* übersetzt wird:

```
public class RequiredDemo {
    public required string RequiredProperty { get; init; }

    public RequiredDemo(string reqProp) {
        RequiredProperty = reqProp;
    }

    public RequiredDemo() { }

    static void Main() {
        RequiredDemo rd1 = new("Required");
    }
}
```

Um den Fehler zu beseitigen kann der parametrisierte Konstruktor durch einen Objektinitialisierer ergänzt werden, was im Beispiel allerdings den parametrisierten Konstruktor überflüssig macht:

```
RequiredDemo rd1 = new("Required") { RequiredProperty = "Required" };
```

---

[1] Es kann leider nicht garantiert werden, dass im Manuskript ab jetzt keine schreibenden Feldzugriffe mehr zu sehen sind.

Eine zweite „Lösungsmöglichkeit“ besteht darin, den Konstruktor mit dem Attribut **SetsRequiredMembers** aus dem Namensraum **System.Diagnostics.CodeAnalysis** zu dekorieren (zu Attributen siehe Kapitel 14), z. B.:

```
[SetsRequiredMembers]
public RequiredDemo(string reqProp) {
    RequiredProperty = reqProp;
}
```

Damit versichert man dem Compiler, dass sich der Konstruktor um die **required**-Deklarationen kümmert, und die folgende Objektkreation mit Konstruktoraufruf wird übersetzt

```
RequiredDemo rd2 = new("Required");
```

Der Compiler vertraut der Versicherung durch das Attribut, sodass die Initialisierungspflicht aufgehoben wird. Nachdem auch der parameterfreie Konstruktor das Attribut **SetsRequiredMembers** erhalten hat,

```
[SetsRequiredMembers]
public RequiredDemo() { }
```

kann trotz *fehlender* Initialisierung auch die folgende Anweisung übersetzt werden:

```
RequiredDemo rd3 = new();
```

Es folgen weitere Regeln zum Modifikator **required**:[1]

- In Schnittstellen (vgl. Kapitel 9) ist der Modifikator **required** *nicht* erlaubt.
- Für Member mit **required**-Modifikator darf der Zugriffsschutz gegenüber dem enthaltenden Typ *nicht* verschärft werden.
- Einer Referenzvariablen darf im Rahmen einer Pflichtinitialisierung auch das Literal **null** zugewiesen werden.

Der Modifikator **required** erfüllt seinen Zweck in einem Typ ohne explizite Konstruktoren, dessen Instanzen per Standardkonstruktor plus Objektinitialisierer erstellt werden. In einem Typ mit expliziten Konstruktoren führt das Schlüsselwort **required** zu lästigem Mehraufwand bei der Instanzkreation. Kommt dann das Konstruktor-Attribut **SetsRequiredMembers** zum Einsatz, sorgt das Schlüsselwort **required** vor allem für falsche Erwartungen und ein dadurch gesteigertes Fehlerrisiko.

5.7 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 von *statischen* Feldern, Methoden, Eigenschaften etc. Ansonsten gibt es bei der Deklaration bzw. Definition kaum Unterschiede zwischen einem Instanz-Mitglied und dem analogen statischen Mitglied.

Auch bei den statischen Mitgliedern gilt für den Zugriffsschutz:

- Voreingestellt ist die Schutzstufe **private**, sodass eine Verwendung nur in klasseneigenen Methoden erlaubt ist.
- Durch Modifikatoren kann eine alternative Schutzstufe festgelegt werden (z. B. **public**).

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/required

5.7.1 Statische Felder und Eigenschaften

In unserem Bruchrechnungsbeispiel soll ein statisches Feld die Anzahl der bisher im aktuellen Programmlauf erzeugten `Bruch`-Objekte aufnehmen:[1]

```
using System;
public class Bruch {
  int zaehler,
      nenner = 1;
  string etikett = "";

  static int anzahl;

  . . .

  public static int Anzahl {
    get {
      return anzahl;
    }
    private set {
      anzahl = value;
    }
  }

  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. Sofern Methoden *fremder* Klassen (durch den Modifikator **public**) der direkte Zugriff auf eine Klassenvariable gewährt wird, müssen diese dem Variablennamen einen 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, sodass der direkte Zugriff klasseneigenen Methoden vorbehalten bleibt.

Damit im erweiterten Bruchrechnungsbeispiel fremde Klassen trotz Datenkapselung die Anzahl der bisher erzeugten `Bruch`-Objekte in Erfahrung bringen können, wird eine statische Eigenschaft namens `Anzahl` mit **public**-Zugriff ergänzt. Weil die **set**-Funktionalität als **private** deklariert ist, können fremde Klassen den `anzahl`-Wert zwar ermitteln, aber nicht verändern.

Es sprechen einige Argumente dafür, auch in den Methoden der Klasse `Bruch` den privaten **setter** der Eigenschaft `Anzahl` statt direkter Feldzugriffe zu verwenden (siehe Abschnitt 5.5.4). Im

---

[1] Ein Visual Studio - Projekt mit der aktuellen Entwicklungsstufe der Klasse `Bruch` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Bruch\b5 Statische Member

Beispiel wird die (automatisch auf 0 initialisierte) Klassenvariable `anzahl` in den beiden Instanzkonstruktoren über die Eigenschaft `Anzahl` inkrementiert.

Weil im **get**- und im **set**-Block der statischen Eigenschaft `Anzahl` nur die Standardzugriffe auf das zugrunde liegende private Feld (backing field) stattfinden, ist eine automatisch implementierte statische Eigenschaft die bequemste und beste Lösung (vgl. Abschnitt 5.5.2):

```
public static int Anzahl { get; private set; }
```

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 5.2.3). Alternative Initialisierungen können in der Variablendeklaration oder im statischen Konstruktor (siehe Abschnitt 5.7.4) 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 erreichbar ist.	Klassenvariablen werden beim Laden der Klasse angelegt und initialisiert.[1]

Einige Programmierer verwenden das Präfix **s_** für die Namen von statischen Feldern mit der Schutzstufe **private** (verfügbar in der eigenen Klasse) oder **internal** (verfügbar im eigenen Assembly), z. B.:[2]

```
private static int s_anzahl;
```

5.7.2 Wiederholung zur Kategorisierung von Variablen

Mittlerweile haben wir verschiedene Variablensorten kennengelernt, 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 oder Eigenschaften deklariert,
  - landen auf dem Stack,
  - werden *nicht* automatisch initialisiert,
  - sind gültig von der Deklaration bis zum Ende des Blocks, der die Deklaration enthält.

---

[1] Entladen kann man eine Klasse nur zusammen mit ihrem sogenannten *AssemblyLoadContext*. Das wird im Manuskript nicht behandelt.

[2] https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

- **Instanzvariablen** ...
  - werden außerhalb jeder Methode bzw. Eigenschaft 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 bzw. Eigenschaft 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 bzw. Eigenschaft (lokale Variable) gehören. Damit sind weitere Eigenschaften wie Verfügbarkeit im Quellcode, Lebensdauer, Ablage im Hauptspeicher und Initialisierung festgelegt.

5.7.3 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. Sofern statische Methoden (alias: *Klassenmethoden*) vorhanden sind, kann man eine Klasse als *Akteur* auf der objektorientierten Bühne betrachten.

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 Konsolenprogramm 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 (impliziten bzw. expliziten) **using**-Direktiven.

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, "Startwert");
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	Eingabe (**fett**) und Ausgabe

`var b = Bruch.BenDef("Startwert");`
`if (b != null)`
`  b.Zeige();`
`else`
`  Console.WriteLine("b zeigt auf null.");`
	`Zähler: 26`
	`Nenner: 39`
	`  2`
	`Startwert = -----`
	`  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 5.3.5) kann es in einer Klasse keine zwei Methoden geben, die sich nur hinsichtlich der An- bzw. Abwesenheit des Modifikators **static** unterscheiden.

In früheren Abschnitten waren mit *Methoden* stets *objektbezogene* Methoden (*Instanzmethoden*) gemeint. Dies soll auch weiterhin so gelten.

5.7.4 Statische Konstruktoren

Analog zu den Instanzkonstruktoren (siehe Abschnitt 5.4.3), die beim Erzeugen eines Objekts ausgeführt werden und das Objekt auf den Einsatz vorbereiten (z. B. durch die Initialisierung von Instanzvariablen), kann für jede Klasse ein statischer Konstruktor zur Initialisierung von Klassenvariablen und andere Arbeiten 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.

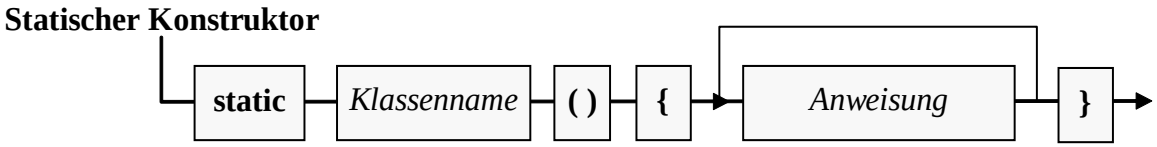
Eine statische Variable (ohne **const**-Modifikator, siehe Abschnitt 5.2.5) durchläuft die folgenden Initialisierungen:

- Zunächst erhält sie den typspezifischen Nullwert (vgl. Abschnitt 5.2.3).
- Ggf. wird anschließend die Initialisierung aus der Deklarationsanweisung vorgenommen.
- Schließlich wird der statische Konstruktor ausgeführt.[1]

Naheliegenderweise 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 von der CLR aufgerufen wird. Insgesamt erhalten wir das folgende Syntaxdiagramm:

---

[1] Genaugenommen führt die vorhandene Initialisierung einer statischen Variablen im Rahmen der Deklarationsanweisung bereits zur automatischen Erstellung eines statischen Konstruktors, der die Initialisierung vornimmt.



In einer etwas künstlichen Erweiterung des `Bruch`-Beispiels soll der parameterfreie Instanzkonstruktor zufallsabhängige, aber pro Programmeinsatz identische Werte zur Initialisierung der Felder `zaehler` und `nenner` verwenden:

```
public Bruch() {
  Zaehler = zaehlerVoreinst;
  Nenner = nennerVoreinst;
  Anzahl++;
}
```

Dazu erhält die `Bruch`-Klasse private und 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 5.2.2). Im statischen Konstruktor wird ein Objekt der Klasse **Random** aus dem Namensraum **System** erzeugt und dann durch **Next()** - Methodenaufrufe mit der Produktion von **int**-Zufallswerten beauftragt:

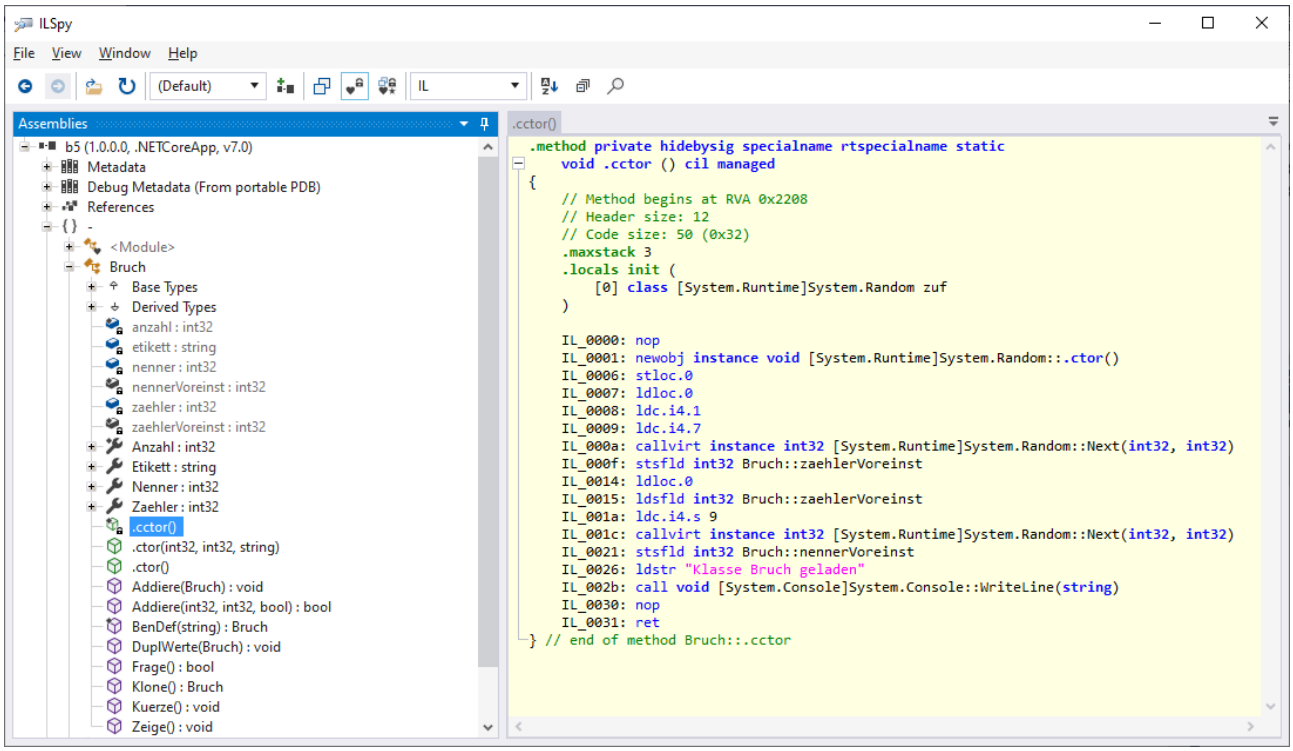
```
static Bruch() {
   var zuf = new Random();
   zaehlerVoreinst = zuf.Next(1,7);
   nennerVoreinst = zuf.Next(zaehlerVoreinst, 9);
   Console.WriteLine("Klasse Bruch geladen");
}
```

Außerdem protokolliert der statische Konstruktor noch das Laden der Klasse, z. B.:

Quellcode	Ausgabe

`var b1 = new Bruch();`
`var b2 = new Bruch();`
`b1.Zeige();`
`b2.Zeige();`
	`Klasse Bruch geladen`
	`  1`
	`-----`
	`  5`
	
	`  1`
	`-----`
	`  5`

Eine Inspektion mit dem Werkzeug ILSpy zeigt, dass der statische Konstruktor einer Klasse im Assembly den Namen **.cctor** trägt, z. B.:



5.7.5 Statische Klassen

Besitzt eine Klasse *ausschließlich* statische Member, ist das Erzeugen von Objekten nicht sinnvoll. Man kann es mit dem Modifikator **static** im Kopf der Klassendefinition verhindern, z. B.

```
public static class Service {
  . . .
}
```

Außerdem lässt sich eine statische Klasse nicht beerben.

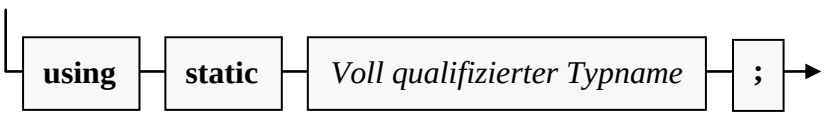
Auch die BCL enthält etliche Klassen, die ausschließlich statische Member enthalten und damit nicht zum Erzeugen von Objekten konzipiert sind. Mit der Klasse **Math** aus dem Namensraum **System** haben wir ein wichtiges Beispiel bereits kennengelernt.

```
public static partial class Math {
    public const double E = 2.7182818284590452354;
    public const double PI = 3.14159265358979323846;
    . . .
}
```

5.7.6 Statische Member eines Typs in eine Quellcodedatei importieren

Die im Abschnitt 2.6.3 beschriebenen und in praktisch jedem Programme implizit und/oder explizit vorhandenen **using**-Direktiven dienen dazu, Namensräume zu importieren, sodass 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, sodass im Programm beim Zugriff weder der Namensraum noch die Klasse anzugeben ist:

Import der statischen Member eines Typs



Im folgenden Beispielprogramm können aufgrund einer Verwendung der vertrauten **using**-Variante die beiden Klassen **Console** und **Math** ohne Namensraumpräfix angesprochen werden:

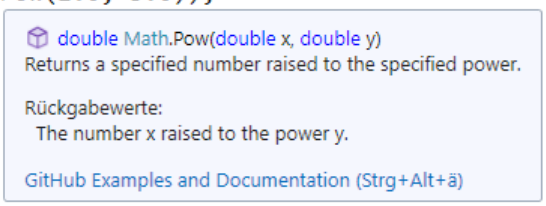
```
using System;
class Prog{
  static void Main(){
    Console.WriteLine(Math.Pow(2.0, 3.0));
  }
}
```

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, z. B.:

```
WriteLine(Pow(2.0, 3.0));
```



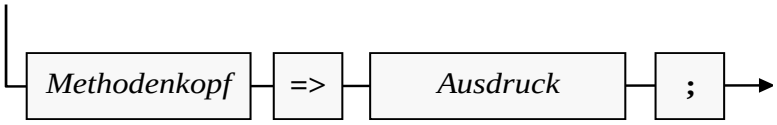
5.8 Vertiefungen zum Thema Methoden

5.8.1 Definition per Lambda-Symbol und Ausdruck

Die mit C# 3.0 zur Unterstützung der LINQ-Technik eingeführte *Lambda-Syntax* dient primär dazu, ein Objekt eines sogenannten *Delegaten* zu realisieren. Mit diesem Datentyp werden wir uns im Kapitel 10 ausführlich beschäftigen. Seit C# 6.0 lässt sich das Lambda-Symbol (=>) auch zu einer Methoden- oder Eigenschaftsdefinition verwenden, sofern die Funktionalität durch einen *einzelnen Ausdruck* realisierbar ist (engl. Bezeichnung: *expression bodied members*). Obwohl die Leser im aktuellen Kapitel 5 schon mit zahlreichen Details belastet worden sind, hat es die Methoden- bzw. Eigenschaftsdefinition durch einen mit Lambda-Symbol eingeleiteten Ausdruck ins Manuskript geschafft, weil diese Syntax in der C# - Dokumentation mit zunehmender Frequenz zum Einsatz kommt. Das Verständnis dieser Dokumentation sollte nicht am Lambda-Symbol scheitern.

Das folgende Syntaxdiagramm zeigt die Definition einer Methode durch einen mit Lambda-Symbol eingeleiteten Ausdruck:

Methodendefinition per Lambda-Symbol und Ausdruck



Auf das Lambda-Symbol (=>) 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 auch der Ausdruck diesen Typ haben, z. B.:[1]

```
static void WriteModulo(int a, int b) =>
  Console.WriteLine($"{a} modulo {b} = {a % b}");
```

Das folgende Programm zeigt die beiden per Lambda-Symbol plus Ausdruck definierten Methoden in Aktion:

Quellcode	Ausgabe

`Console.WriteLine(Max(3, 4));`
`WriteModulo(17, 3);`
 	
`static int Max(int a, int b) => a >= b ? a : b;`
 	
`static void WriteModulo(int a, int b) =>`
`  Console.WriteLine($"{a} modulo {b} = {a % b}");`
	`4`
	`17 modulo 3 = 2`

Mittlerweile können neben Methoden auch weitere ausführbare Klassenmitglieder per Lambda-Symbol plus Ausdruck definiert werden:[2]

Mitglied	**Möglich ab C# - Version**
Methode	6.0
Get-Block einer Eigenschaft	6.0
Set-Block einer Eigenschaft	7.0
Indexer[3]	7.0
Konstruktor	7.0
Finalisierer	7.0

Wir beschränken uns darauf, die Ausdrucksdefinition für eine Eigenschaft zu demonstrieren. In unserer Klasse `Bruch` kommen bei der Eigenschaft `Zaehler` der **get**- und der **set**-Block mit *einem* Ausdruck aus:

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

Folglich ist für die Eigenschaft `Zaehler` auch eine Ausdrucksdefinition möglich, wobei die Anzahl der benötigten Zeilen sinkt, und die Übersichtlichkeit steigt:

---

1 Weil gemäß Abschnitt 4.5.2 auch ein Methodenaufruf einen Ausdruck darstellt, ist das Beispiel eine korrekte Konkretisierung des Syntaxdiagramms zur Methodendefinition per Ausdruck.

2 https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/expression-bodied-members

3 Indexer werden im Abschnitt 5.11 behandelt.

```
public int Zaehler {
   get => zaehler;
   set => zaehler = value;
}
```

Allerdings ist im Beispiel eine automatisch implementierte Eigenschaft (vgl. Abschnitt 5.5.2)

```
public int Zaehler { get; set; }
```

eine noch bessere Lösung, zumal das backing field zur Eigenschaft …

- automatisch angelegt wird
- und vor jedem Zugriff (auch durch klasseneigene Methoden) geschützt ist (vgl. Abschnitt 5.5.4).

5.8.2 Rekursive Methoden

Innerhalb einer Methode darf man selbstverständlich nach Belieben *andere* Methoden aufrufen. Es ist aber auch zulässig, dass eine Methode *sich selbst* aufruft. Solche *rekursive* Aufrufe erlauben eine elegante Lösung für ein Problem, das sich sukzessive 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 arbeitende) Realisierungen des Euklidischen Lösungsverfahrens kennengelernt: Im Kapitel 1 wurde ein sehr einfacher Algorithmus benutzt, den Sie später in einer Übungsaufgabe (siehe Abschnitt 4.7.4) durch einen effizienteren Algorithmus (unter Verwendung des Modulo-Operators) 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:[1]

```
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:

[1] Ein Visual Studio - Projekt mit der aktuellen Entwicklungsstufe der Klasse `Bruch` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Bruch\b6 Rekursion

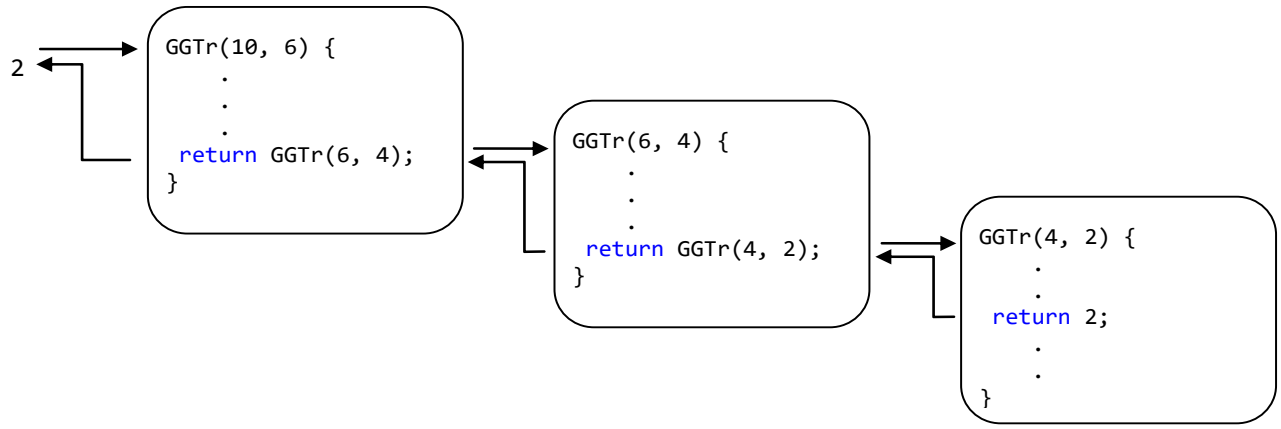
```
int GGTr(int a, int b) {
   int rest = a % b;
   if (rest == 0)
      return b;
   else
      return GGTr(b, rest);
}
```

Statt eine Schleife zu benutzen, arbeitet die rekursive Methode nach der folgenden 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 bestimmen, auf das einfachere Problem zurückgeführt, den GGT von `b` und `(a % b)` zu bestimmen, und die Methode `GGTr()` ruft sich selbst mit neuen Aktualparametern auf. Dies geschieht im Ausdruck der folgenden **return**-Anweisung:
  ```
  return GGTr(b, rest);
  ```

Im iterativen Algorithmus wird derselbe Trick zur Reduktion des Problems verwendet, und den zugrunde liegenden Satz der mathematischen Zahlentheorie kennen Sie schon aus der oben erwähnten Übungsaufgabe im Abschnitt 4.7.4.

Wird die Methode `GGTr()` z. B. mit den Argumenten 10 und 6 aufgerufen, dann kommt es zur folgenden Aufrufverschachtelung:



Generell läuft eine rekursive Methode mit Lösungsübermittlung per Rückgabewert nach der im folgenden **Struktogramm** beschriebenen Logik ab:

Ist das Problem direkt lösbar?	
Ja	**Nein**
Lösung ermitteln und an den Aufrufer melden	**Rekursiver Aufruf mit einem einfacheren Problem** **Lösung des einfacheren Problems zur Lösung des Ausgangsproblems verwenden** **Lösung an den Aufrufer melden**

Im Beispiel ist die Lösung des einfacheren Problems 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 irgendwann die Stack-Kapazität, und es kommt zu einem Ausnahmefehler. Um einen Stapelüberlauf zu provozieren, hat die folgende Variante der Methode `GGTr()` eine fehlerhafte Parameterliste im Selbstaufruf erhalten:

```
int GGTr(int a, int b) {
    int rest = a % b;
    if (rest == 0)
        return b;
    else
        return GGTr(a, b);
}
```

Nach ca. 20.000 Selbstaufrufen kommt es zum Stapelüberlauf:

```
Stack overflow.
Repeat 19273 times:
--------------------------------
   at Bruch.GGTr(Int32, Int32)
--------------------------------
   at Bruch.Kuerze()
   at Program.<Main>$(System.String[])
```

Zu einem rekursiven Algorithmus (per Selbstaufruf einer Methode) existiert immer 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 und mehr Speicher.

5.8.3 Operatoren überladen

In C# können nicht nur Methoden überladen werden (siehe Abschnitt 5.3.5), sondern auch die *Operatoren* (**+, *** etc.), was z. B. in der Klasse **String** mit dem „+“ - Operator geschehen ist, sodass wir Zeichenfolgen bequem per „+“ - Operator 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.:

```
Bruch b3 = b1.Klone();
b3.Addiere(b2);
b3.Etikett = $"{b1.Etikett} + {b2.Etikett}";
```

Mit reichlich Schreibaufwand wird das Ziel erreicht:

Quellcode	Ausgabe

```
var b1 = new Bruch(1, 2, "b1");
b1.Zeige();
var b2 = new Bruch(1, 4, "b2");
b2.Zeige();

Bruch b3 = b1.Klone();
b3.Addiere(b2);
b3.Etikett = $"{b1.Etikett} + {b2.Etikett}";

b3.Zeige();
```

```
        1
b1 =  -----
        2

        1
b2 =  -----
        4

            3
b1 + b2 =  -----
            4
```

Eleganter wäre es, denselben Zweck mit der von elementaren Datentypen her gewohnten Syntax zu realisieren:

```
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:[1]

```
public static Bruch operator +(Bruch b1, Bruch b2) {
    Bruch tmp = b1.Klone();
    tmp.Addiere(b2);
    tmp.Etikett = $"{b1.Etikett} + {b2.Etikett}";
    return tmp;
}
```

Beim Überladen von Operatoren sind u. a. die folgenden Regeln zu beachten:

- Es ist grundsätzliche eine *statische* Definition erforderlich.
- Als Namen verwendet man das Schlüsselwort **operator** mit dem jeweiligen Operationszeichen als Suffix.
- Die Bindungskraft und die Assoziativität des überladenen Operators bleiben erhalten.

Nähere Hinweise finden sich z. B. bei Mössenböck (2019, S. 78ff).

Mit dem überladenen „+“ - Operator lassen sich `Bruch`-Additionen nun wunschgemäß formulieren:

Quellcode	Ausgabe

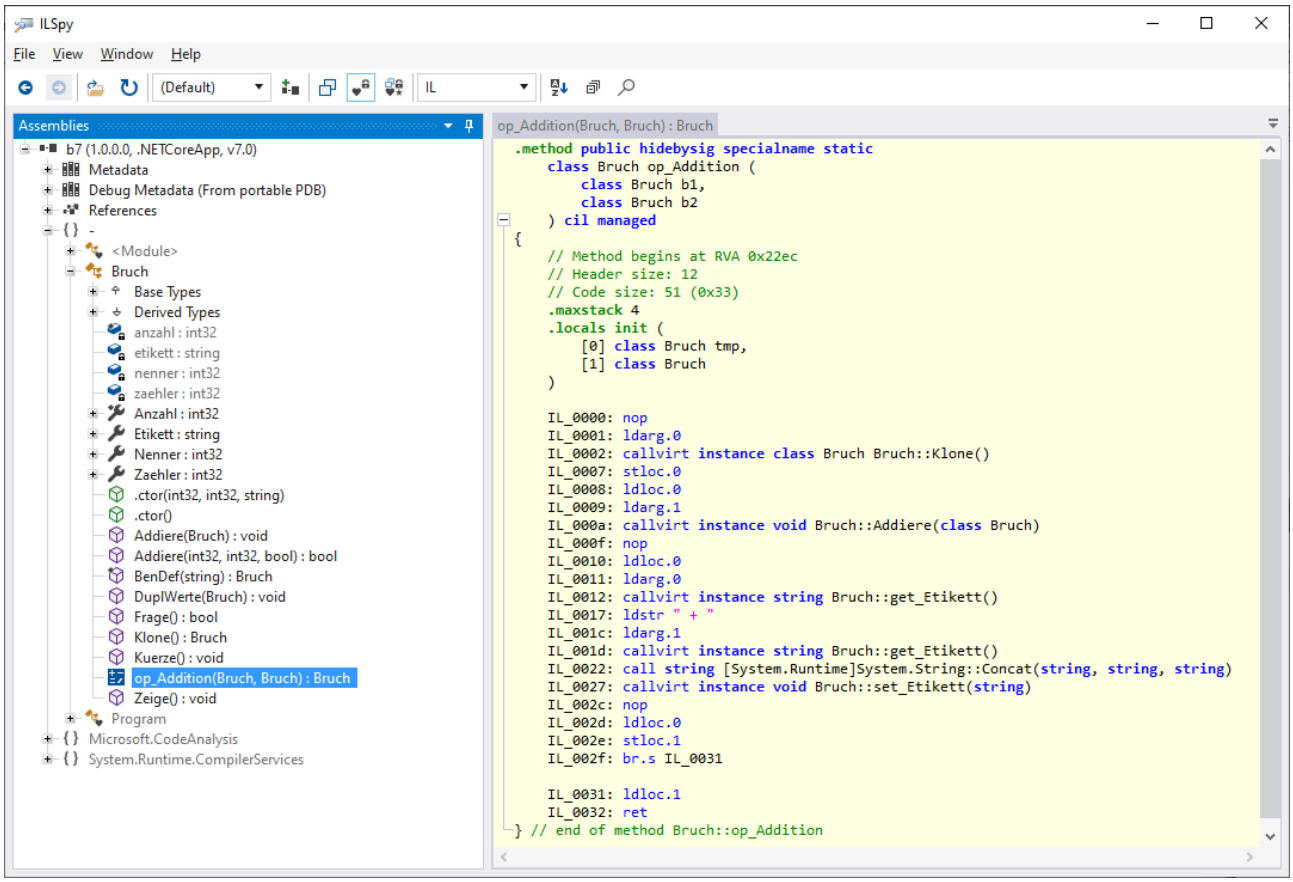
```
var b1 = new Bruch(1, 2, "b1");
b1.Zeige();
var b2 = new Bruch(1, 4, "b2");
b2.Zeige();
var b3 = b1 + b2;
b3.Zeige();
```

```
        1
b1 =  -----
        2

        1
b2 =  -----
        4

            3
b1 + b2 =  -----
            4
```

Wie das Hilfsprogramm ILSpy (vgl. Abschnitt 3.4.1) zeigt, resultiert aus unserer Operatorenüberladung im Assembly die Methode **op_Addition()**:

---

[1] Ein Visual Studio - Projekt mit der aktuellen Entwicklungsstufe der Klasse `Bruch` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Bruch\b7 Operatoren-Überladung



Weitere Kandidaten für Operatorüberladungen in der Klasse `Bruch` sind == und **!=**, wobei dann aber auch die von **Object** geerbte und die Speicheradressen der Objekte vergleichende Methode **Equals()** durch eine die Feldinhalte vergleichende Alternative überschrieben werden sollte. Weil dabei zu viele Vorgriffe auf das Kapitel 7 über die Vererbung erforderlich wären, verzichten wir auf das Überladen der Operatoren == und **!=**.

5.9 Aggregation bzw. 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 Vererbung und der Polymorphie ist diese *Aggregation* bzw. *Komposition* eine weitere 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.

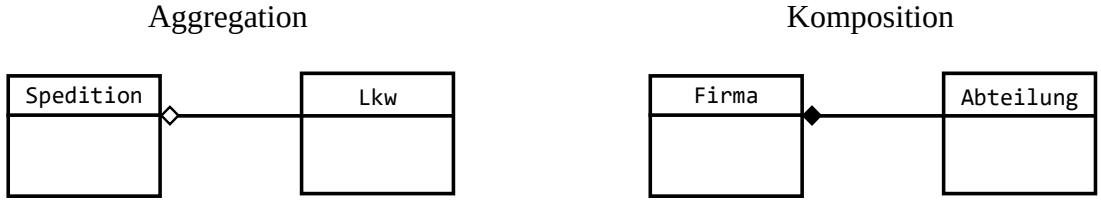
In der Literatur zur objektorientierten Programmierung wird den beiden Begriffe *Aggregation* und *Komposition* in der Regel eine unterschiedliche Bedeutung beigemessen, wobei kein strenges Kriterium für den Begriffsgebrauch existiert, und die Differenzierung in konkreten Fällen oft schwierig ist:

[1] Die betroffenen Personen mögen den Fachterminus *Objekt* nicht persönlich nehmen.

- Aggregation
  Die aggregierende bzw. „besitzende“ Klasse A (z. B. Spedition) enthält Objekte der Klasse B (z. B. Lkw), doch können die Objekte der Klasse B auch ohne das Objekt der Klasse A existieren.
- Komposition
  Die das Ganze repräsentierende Klasse A (z. B. Firma) enthält Objekte der Klasse B (z. B. Abteilung), und ein Objekt der Klasse B kann nicht unabhängig vom „besitzenden“ Objekt der Klasse A existieren. Wenn eine Klasse B nur im Rahmen einer Klasse A zu gebrauchen ist, dann kommt die Definition als innere Klasse von A in Betracht (siehe Abschnitt 5.10).

Die Komposition („Hat - Beziehung“) wurde schon im Abschnitt 1.2 erwähnt mit dem Beispiel „Ein Auto hat einen Motor“. Ob es sich bei diesem Beispiel tatsächlich um eine Komposition oder eher um eine Aggregation handelt, soll hier nicht diskutiert werden.

Obwohl die Aggregations- bzw. Kompositionsbeziehung nicht zwischen Klassen besteht, sondern zwischen Objekten, werden auch UML-Klassendiagramme zur Darstellung verwendet:



Die Beziehung zwischen den Klassen wird durch eine Linie dargestellt, wobei an der Seite der besitzenden Klasse unterschiedliche Symbole verwendet werden:

- Eine offene Raute bei der Aggregation
- Eine gefüllte Raute bei der Komposition

Man kann den Standpunkt einnehmen, dass die Aggregation bzw. 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 unkritisch, wenn Sie den Rest des aktuellen Abschnitts mit dem recht länglichen Beispiel überspringen.

Wir konstruieren eine Klasse namens `Aufgabe` zur Verwendung in einem Bruchrechnungs-Training. In der `Aufgabe`-Klassendefinition tauchen vier Instanzvariablen vom Typ `Bruch` auf:[1]

```
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, "Resultat");
      antwort = new Bruch();
      Init();
   }
```

[1] Ein Visual Studio - Projekt mit der aktuellen Entwicklungsstufe der Klasse `Bruch` ist hier zu finden:
…\BspUeb\Klassen und Objekte\Bruch\b8 Aggregation

```
    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;
       }
    }

    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("\n Leider falsch!\n");
          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 den 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.

Ob es sich bei der Beziehung zwischen den Klassen `Aufgabe` und `Bruch` um eine Aggregation oder um eine Komposition handelt, ist eine überflüssige Frage.

In der Klasse `Bruch` wird die Instanzmethode `Multipliziere()` nachgerüstet, die analog zur Methode `Addiere()` arbeitet:

```
public void Multipliziere(Bruch b) {
   Zaehler = zaehler * b.zaehler;
   Nenner = nenner * b.nenner;
   Kuerze();
}
```

Im folgenden Programm wird die Klasse `Aufgabe` für ein Bruchrechnungstraining verwendet:

```
using System;
class Bruchrechnung {
   static void Main() {
      Aufgabe 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

 Leider falsch!
              9
 Resultat = -----
              10
```

5.10 Geschachtelte Klassen

Eine Klasse darf neben Feldern, Methoden etc. auch Klassendefinitionen enthalten, wobei *geschachtelte Klassen* (engl. *nested classes*) entstehen. Eine Klasse innerhalb einer umgebenden Klasse zu definieren, ist z. B. dann sinnvoll, wenn die geschachtelte Klasse nicht allgemein benötigt wird, sondern nur zur Modellierung von speziellem Zubehör der umgebenden Klasse dient. Bei der voreingestellten Schutzstufe **private** ist die geschachtelte Klasse im restlichen Programm nicht sichtbar, kann also dort nicht (fehlerhaft) verwendet werden.

Im folgenden Beispiel werden innerhalb der Klasse `Familie` die Klassen `Tochter` und `Sohn` definiert:

```
Familie f = new("Müller", "Lea", 7, "Theo", 4);
f.Informiere();

public class Familie {
  readonly string name;
  readonly Tochter t;
  readonly 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 Informiere() {
    Console.WriteLine($"Die Kinder von Familie {name}:\n");
    t.Informiere();
    s.Informiere();
  }

  class Tochter {
    readonly Familie f;
    readonly string name;
    readonly int alter;

     public Tochter(Familie f_, string name_, int alt_) {
       f = f_;
       name = name_;
       alter = alt_;
    }

    public void Informiere() {
     Console.WriteLine($"  Ich bin die {alter}-jährige Tochter {name} von Familie {f.name}.");
    }
  }

  public class Sohn {
    readonly Familie f;
    readonly string name;
    readonly int alter;

    public Sohn(Familie f_, string name_, int alt_) {
      f = f_;
      name = name_;
      alter = alt_;
    }

    public void Informiere() {
     Console.WriteLine($"  Ich bin der {alter}-jährige Sohn {name} von Familie {f.name}.");
    }
  }
}
```

Die am Anfang der Quellcodedatei erlaubten Anweisungen auf oberster Ebene

```
var f = new Familie("Müller", "Lea", 7, "Leo", 4);
f.Informiere();
```

liefern die 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. die folgende Regeln:[1]

- Die Methoden einer geschachtelten Klasse dürfen auf die als **private** oder **protected** deklarierten Member der umgebenden Klasse zugreifen, d. h.:[2]
  - auf die Member eines Objekts der umgebenden Klasse, sofern eine Referenz vorhanden ist, die meist per Konstruktor bekannt gemacht wird (wie in den inneren Klassen `Tochter` und `Sohn`)
  - auf die statischen Member der umgebenden Klasse

  In der folgenden Anweisung aus der `Tochter`-Methode `Informiere()` erfolgt ein Zugriff auf die private Instanzvariable `name` aus der Klasse `Familie`):

  ```
  Console.WriteLine("  Ich bin die {0}-jährige Tochter {1} von Familie {2}",
                    alter, name, f.name);
  ```
- Umgekehrt hat die umgebende Klasse *keine* Zugriffsrechte für die privaten Member einer geschachtelten Klasse, weshalb im Beispiel die Konstruktoren und die `Informiere()` - Methoden der inneren Klassen als **public** definiert wurden.
- Geschachtelte Klassen besitzen wie die sonstigen Klassen-Member (Felder, Methoden, Eigenschaften etc.) die voreingestellte Schutzstufe **private**.
- Erhält eine geschachtelte 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(f, "Leo", 4);
  ```

5.11 Indexer

Bei Arrays und vielen Klassen (z. B. **String**) hat sich der Indexzugriff auf die Elemente eines Objekts per **[ ]** - Operator als sehr nützlich bis unverzichtbar erwiesen (siehe z. B. Abschnitt 4.7.2.3.2). Um denselben Komfort für eine eigene Klasse (oder Struktur, siehe Abschnitt 6.1) zu realisieren, die zur Verwaltung einer Kollektion (z. B. einer Liste) von Elementen desselben Typs dient, muss man dem Typ einen sogenannten **Indexer** spendieren. Analog zur Situation bei einer Eigenschaft steht hinter einem Indexer ein *Paar von Methoden* für den lesenden bzw. schreibenden Zugriff auf ein Element per Indexsyntax.

5.11.1 Definition am Beispiel einer Klasse zur Verwaltung einer verketteten Liste

Als Beispiel betrachten wir eine Klasse namens `PersList`, die eine Liste von Objekten der folgenden Klasse `Person` verwaltet:

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/nested-types

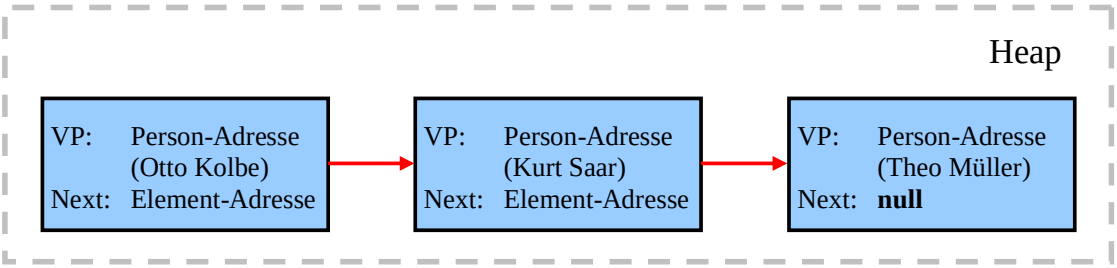
[2] Zum Zugriffsmodifikator **protected** siehe Abschnitt 5.12.

```
public class Person {
   public string Vorname { get; set; }
   public string Name { get; set; }
   public Person(string vorname, string nachname) {
      Vorname = vorname;
      Name = nachname;
   }
}
```

Die Klasse `PersList` verwendet eine geschachtelte Klasse namens `Element` (vgl. Abschnitt 5.10):

```
public class PersList {
   Element first, last;

   class Element {
      public Person VP { get; set; }
      public Element Next { get; set; }

      public Element(Person p) {
         VP = p;
      }
   }

   public int Count {
      get;
      private set;
   }

   public void Add(Person neu) {
      if (neu == null)
         return;
      Element e = new(neu);
      if (Count == 0)
         first = last = e;
      else {
         last.Next = e;
         last = e;
      }
      Count++;
   }

   Element FindElement(int i) {
      Element sel = first;
      for (int j = 0; j < i; j++)
         sel = sel.Next;
      return sel;
   }

   public Person this[int i] {
      get {
         if (i >= 0 && i < Count)
            return FindElement(i).VP;
         else
            return null;
      }
      set {
         if (i >= 0 && i < Count && value != null)
            FindElement(i).VP = value;
      }
   }
}
```

Ein Objekt der geschachtelten Klasse `Element` verwaltet ein Objekt der Klasse `Person` auf und besitzt dazu die Eigenschaft `VP`. In der Eigenschaft `Next` merkt sich ein `Element`-Objekt seinen Nachfolger, das auf die nächste Person zeigt. Die `Element`-Objekte werden von der Container-Klasse `PersList` als sogenannte *verkettete Liste* verwaltet. Weil ein `PersList`-Objekt im Feld `first` eine Referenz auf das erste Element besitzt, lässt sich durch das Verfolgen der `Next`-Referenzen jedes Serienelement erreichen. In der folgenden Abbildung ist eine Kette aus drei `Element`-Objekten zu sehen, die jeweils ein `Person`-Objekt verwalten:



Nun kommen wir endlich zum Thema des Abschnitts 5.11. Für den lesenden oder schreibenden Zugriff auf das *i*-te Listenelement stellt `PersList` 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**).

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 mit dem Datentyp und dem Namen deklariert.
- Dem **set**-Block wird wie bei einer Eigenschaft ein impliziter Parameter namens **value** übergeben.

Im Beispiel liefert der **get**-Block des Indexers bei einem ungeeigneten Indexwert die Rückgabe **null**. Ansonsten arbeitet er sich mit dem `first`-Element startend über die `Next`-Werte vor, um schließlich die Adresse des gewünschten `Person`-Objekts zu liefern.

Weil der **set**-Block dieselbe `Next`-Nachverfolgung benötigt, wird dazu die private Methode `FindeElement()` definiert.

Der **set**-Block des Indexers wird im Beispiel aktiv, wenn die zu ersetzende Indexposition gültig und **value** von **null** verschieden ist:

- Das betroffene `Element`-Objekt wird mit Hilfe von `FindeElement()` ermittelt,
- und seine `VP`-Eigenschaft erhält die Adresse des Neulings.

Es bleibt noch die `PersList`-Methode `Add()` zu erläutern, die nur bei einem von **null** verschiedenen Aktualparameter tätig wird:

- Es wird ein neues `Element`-Objekt mit der Adresse des aufzunehmenden `Person`-Objekts als `VP`-Wert angelegt.
- Ist die Liste noch leer, dann wird die Adresse des neuen `Element`-Objekts in die `PersList`-Felder `first` und `last` eingetragen.

- Anderenfalls wird der Neuling am Ende angehängt:
  - Das bislang letzte `Element`-Objekt erhält die Adresse des neuen `Element`-Objekts als `Next`-Wert.
  - Das `PersList`-Feld `last` erhält die Adresse des neuen `Element`-Objekts.
- Schließlich wird die Anzahl der Listenelemente inkrementiert.

Das folgende Programm verwendet ein `PersList`-Objekt zur Verwaltung einer Personenliste und demonstriert den lesenden sowie den schreibenden Indexzugriff:

```
class PersListDemo {
  static void Main() {
    var pl = new PersList();
    pl.Add(new Person("Otto", "Kolbe"));
    pl.Add(new Person("Kurt", "Saar"));
    pl.Add(new Person("Theo", "Müller"));
    for (int i = 0; i < pl.Count; i++)
      Console.WriteLine($"Nummer {i}: {pl[i].Vorname} {pl[i].Name}");
    Console.WriteLine();
    pl[1] = new Person("Ilse", "Golter");
    for (int i = 0; i < pl.Count; i++)
      Console.WriteLine($"Nummer {i}: {pl[i].Vorname} {pl[i].Name}");
  }
}
```

Es resultiert die 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
```

Statt eine eigene Klasse `PersList` mit Listenkonstruktion und Indexer zu entwerfen, wird man in der Praxis eine solche Aufgabe weit ökonomischer unter Verwendung einer generischen Kollektionsklasse aus dem Namensraum **System.Collections.Generic** realisieren (siehe Abschnitt 11.3). Allerdings gehört der Eigenbau einer verketteten Liste zur Programmierer-Grundausbildung, sodass der Aufwand des `PersList`-Beispiels gerechtfertigt ist.

Die Definition eines Indexers hat zur Folge, dass der Compiler automatisch eine Eigenschaft mit dem Namen **Item** anlegt.[1] Auf diese Eigenschaft ist im Quellcode kein Zugriff möglich, und man könnte sie ignorieren, wenn nicht die Definition eines Indexers scheitern würde, sobald man selbst eine Eigenschaft mit dem Namen `Item` erstellt. Wäre z. B. in der Klasse `PersList` eine explizit definierte Eigenschaft namens `Item` vorhanden,

```
public class PersList {
  Element first, last;

  int Item { get; set; }
  . . .
}
```

dann würde eine Indexer-Definition scheitern:

[1] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/using-indexers

```
public Person this[int i] {
```

Person PersList.this[int i] { get; set; }

CS0102: Der Typ "PersList" enthält bereits eine Definition für "Item".

Wenn die explizit definierte Eigenschaft namens `Item` unbedingt benötigt wird, dann muss der interne Name des Indexers durch ein angeheftetes **IndexerNameAttribute** (Namensraum **System.Runtime.CompilerServices**) geändert werden, z. B.:[1]

```
[System.Runtime.CompilerServices.IndexerName("MyItem")]
public Person this[int i] {
  . . .
}
```

Ein Visual Studio – Projekt mit dem Indexer-Beispiel befindet sich im Ordner:

…\BspUeb\Klassen und Objekte\Indexer

5.11.2 Indexer überladen

Wenn die Elemente einer Kollektion alternativ 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 aus dem Abschnitt 5.11.1 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 **string**-Parameter zu sehen:

```
String s = "Ilse";
Console.WriteLine($"\nName der ersten Person mit dem Vornamen \"{s}\": {pl[s].Name}");
```

5.11.3 Mehrdimensionale Indexer

C# erlaubt auch *mehrdimensionale* Indexer, sodass z. B. eine Klasse namens `Grundbuch` zur Verwaltung von Grundstücken einen Indexer zur Ansprache ihrer Elemente über einen ersten Parameter vom Typ **String** und einen zweiten Parameter vom Typ **int** anbieten könnte:

```
public class Grundbuch {

    public GBEintrag this[string flur, int nr] {
        . . .
    }
     . . .
}
```

Allerdings rät Microsoft in der Codeanalyse-Regel CA1023 von dieser Option ab:[2]

---

[1] Mit Attributen werden wir uns im Kapitel 14 beschäftigen.

[2] https://learn.microsoft.com/en-us/visualstudio/code-quality/ca1023?view=vs-2022&tabs=csharp

> Indexers, that is, indexed properties, should use a single index. Multi-dimensional indexers can significantly reduce the usability of the library.
> . . .
> To fix a violation of this rule, change the design to use a lone integer or string index, or use a method instead of the indexer.

5.12 Schutzstufen für Klassen und ihre Mitglieder

Nachdem die Datenkapselung im Manuskript mehrfach als wesentlicher Vorzug bzw. als Kernidee der objektorientierten Programmierung herausgestellt wurde, und wiederholt Angaben zur Verfügbarkeit von Klassen bzw. Klassenmitgliedern gemacht wurden, sollen die Regeln zum Zugriffsschutz nun zusammengestellt werden, obwohl dabei noch ein Vorgriff auf das Thema *Vererbung* nötig ist.

Im Kapitel 1 haben wir uns bei der Darstellung der Datenkapselung aus didaktischen Gründen auf die beiden Schutzstufen **private** und **public** beschränkt. Es sprechen aber gute Gründe dafür, …

- andere Typen im eigenen Assembly
- oder andere Typen in der eigenen Quellcodedatei
- oder abgeleitete Klassen

gegenüber sonstigen Typen zu privilegieren.

In der C# - Version 11 wurden die Optionen zur Vergabe von Zugriffsrechten um das (kontextabhängige) Schlüsselwort **file** erweitert, das als Modifikator für nicht geschachtelte Typen zugelassen ist.[1] Ein so dekorierter Typ ist nur für andere Typen in derselben Quellcodedatei verfügbar.

Befindet sich ein Typ `F` mit der Schutzstufe **file** in derselben Quellcodedatei wie ein Typ `A` mit einer liberaleren Schutzstufe (**internal** oder **public**), dann ist der Typ `F` auch im Typ `A` nur eingeschränkt nutzbar:

- Eine Methode des Typs `A` darf den Typ `F` weder als Rückgabetyp noch als Parametertyp verwenden. Eine lokale Variable vom Typ `F` ist jedoch möglich.
- In der Klasse `A` sind keine Felder vom Typ `F` erlaubt.

Im folgenden Quellcode sind eine gelungene und eine gescheiterte Verwendung der Klasse `F` in der Klasse `A` zu sehen:

```
file class Pfeil {
    public void Hallo() {
        Console.WriteLine("Hallo, ich bin ein Pfeil");
    }
}

public class Klasse {
    Pfeil pf = new();

    static public void Meth() {
        Pfeil pl = new();
        pl.Hallo();
    }
}
```

Für eine nicht geschachtelte Klasse ist die Verfügbarkeit in anderen Klassen folgendermaßen geregelt:

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/file

Modifikator	Verfügbarkeit für *alle* Klassen …		
	in der eigenen cs-Datei	im eigenen Assembly	in anderen Assemblies
ohne oder **internal**	ja	ja	nein
file	ja (eingeschränkt)	nein	nein
public	ja	ja	ja

Unsere als **public** definierte Beispielklasse `Bruch` kann in beliebigen .NET - Programmen mit Zugang zur Assembly-Datei genutzt werden, was gleich im Abschnitt 5.13 demonstriert werden soll.

Für Klassen-Member (Felder, Methoden, Eigenschaften, Indexer, geschachtelte Klassen etc.) unterstützt C# die folgenden Schutzstufen:[1]

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 Member	geerbte Member	nein
protected internal	ja	ja	ja	geerbte Member	nein
private protected	ja	nein	geerbte Member	nein	nein
public	ja	ja	ja	ja	ja

Die Zugriffsmodifikatorenkombination **private protected** ist seit C# 7.2 verfügbar.

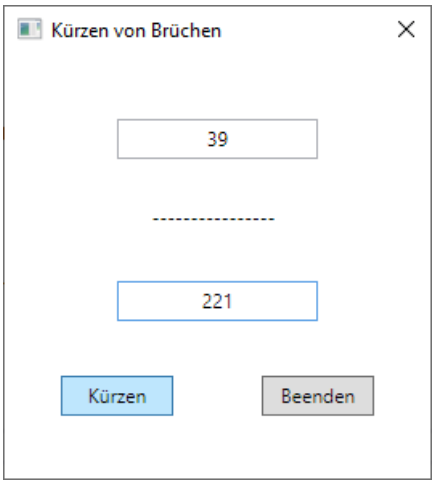
Wir haben die Methoden und Eigenschaften unserer Beispielklasse `Bruch` als **public** definiert und bei den Feldern die voreingestellte Schutzstufe **private** beibehalten.

Die beschriebenen Zugriffsregeln für Klassen und ihre Mitglieder gelten analog auch bei den später vorzustellenden Typen (Strukturen, Enumerationen, Schnittstellen und Delegaten).

5.13 Bruchkürzungsprogramm mit WPF-Bedienoberfläche

Nachdem Sie nun wesentliche Teile der objektorientierten Programmierung mit C# kennengelernt haben, ist vielleicht ein weiterer Ausblick auf die Entwicklung von Windows-Programmen mit grafischer Bedienoberfläche als Belohnung und Motivationsquelle angemessen. Schließlich soll im Manuskript auch die Erfahrung vermittelt werden, dass man beim Programmieren Erfolg und damit Spaß haben kann. Wir erstellen nun das schon im Abschnitt 1.6 präsentierte Bruchkürzungsprogramm mit grafischer Bedienoberfläche:

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/access-modifiers
https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers



Bei dieser Gelegenheit erweitern wir unser Wissen über die Erstellung einer WPF-Anwendung (*Windows Presentation Foundation*), um bei der „offiziellen“ Behandlung des Themas im Kapitel 12 schon einige Erfahrungen einbringen zu können. WPF wird im Manuskript als GUI-Technik bevorzugt gegenüber den Alternativen:

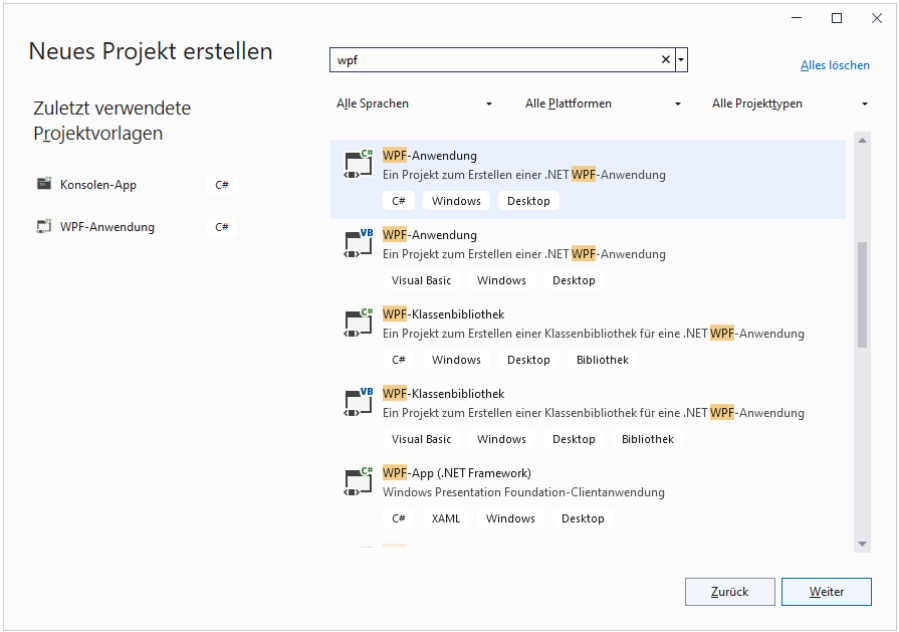
- **WinForms**
  Diese Technik ist veraltet.
- **UWP** (*Universal Windows Platform*)
  Diese Technik hat sich trotz vieler Bemühungen der Firma Microsoft nicht durchgesetzt.
- **MAUI** (*Multi-platform App UI*)
  Für reine Windows-Anwendungen besteht bei dieser Technik momentan noch eine ungünstige Kosten/Nutzen – Bilanz.

5.13.1 Projekt anlegen

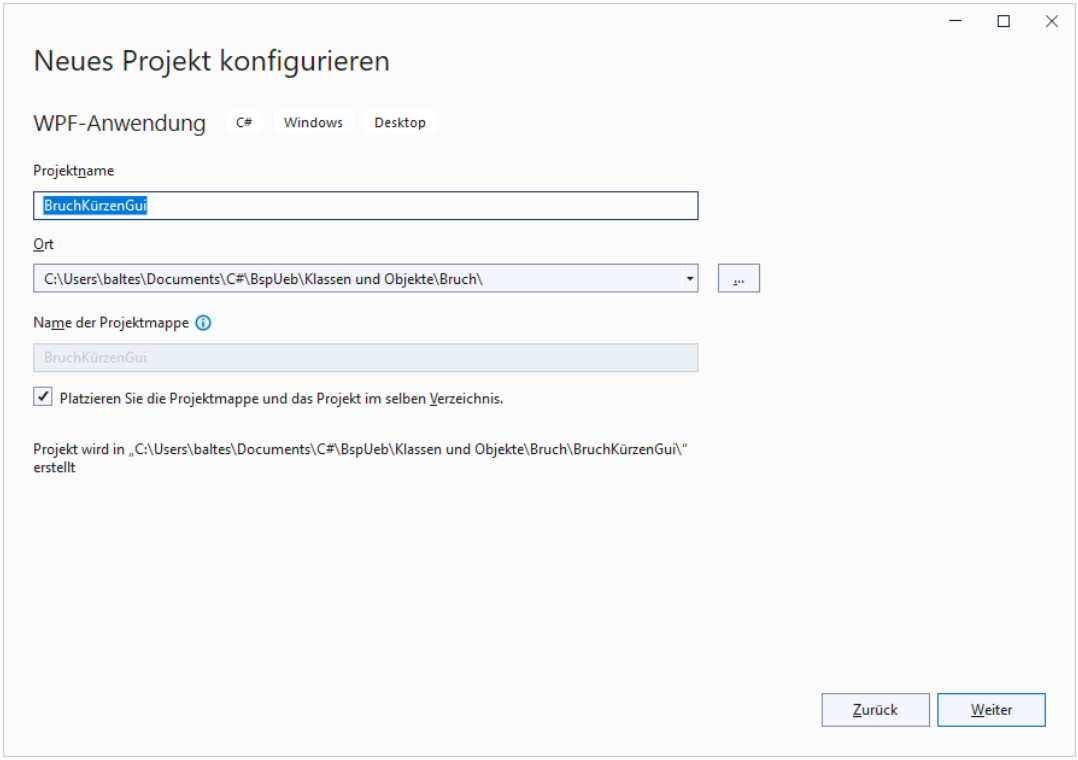
Verwenden Sie im Visual Studio nach

Datei > Neu > Projekt

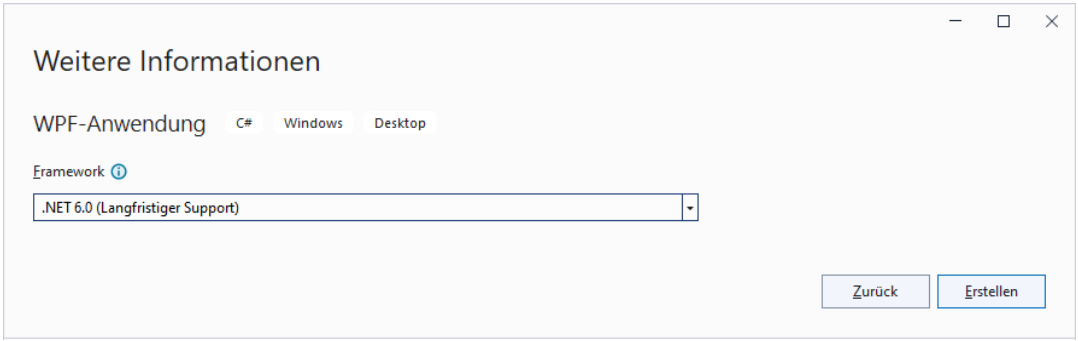
für ein neues Projekt mit dem Namen `BruchKürzenGui` die Vorlage **WPF-Anwendung**:



Tragen Sie den **Projektnamen** ein, und wählen Sie einen passsenden **Ort**, z. B.:

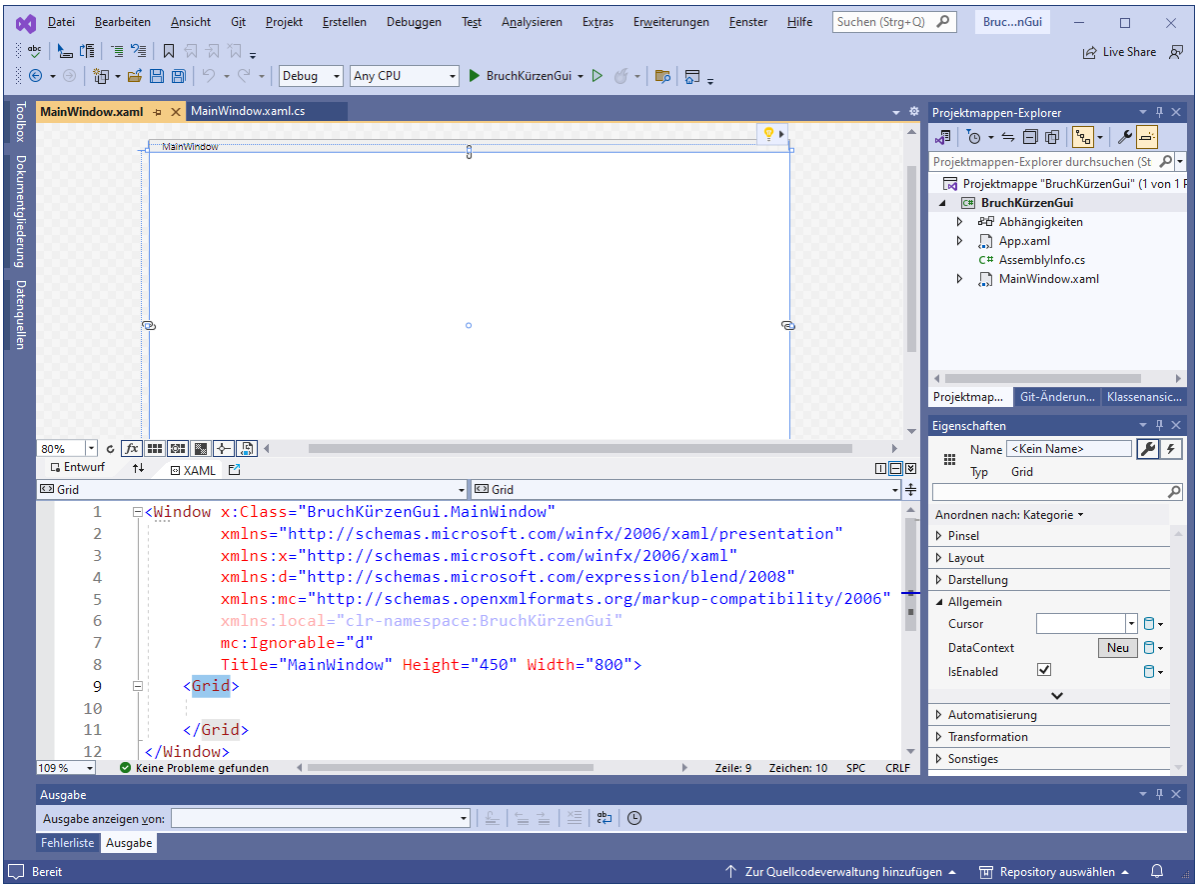


Es bietet sich an, eine .NET – Version mit langfristiger Unterstützung zu wählen:



Weil wir ein Assembly mit der Klasse `Bruch` einbinden wollen, darf das im dortigen Projekt eingestellte Kompatibilitätsniveau (.NET 6.0, siehe Abschnitt 5.13.7) nicht unterschritten werden.

Nach einem Mausklick auf **Erstellen** präsentiert die Entwicklungsumgebung im **WPF-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 konfigurierbare Komponenten (Steuerelemente) aus der **Toolbox** (siehe Abschnitt 5.13.3) übernehmen. Wie gleich zu sehen sein wird, definieren wir dabei die neue Klasse `MainWindow` im Namensraum `BruchKürzenGui`.

5.13.2 Deklaration der Bedienoberfläche per XAML

Ein zentrales Merkmal der WPF-Technologie besteht darin, das GUI-Design einer Anwendung durch eine XML-Spezialisierung (*eXtensible Markup Language*) namens XAML (*eXtensible Application Markup Language*) zu deklarieren. Zu unserem Anwendungsfenster gehört eine XAML-Datei, die im unteren Teil des WPF-Designers 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="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Das **Window**-Element mit dem Start-Tag

```
<Window  . . . >
```

und dem End-Tag

```
</Window>
```

definiert ein Fenster, also das Erscheinungsbild eines Objekts aus einer anwendungseigenen Klasse, die von der BCL-Klasse **Window** abstammt. Im Start-Tag des **Window**-Elements befinden sich etliche XML-Attribute:

- Die **xmlns**-Attribute deklarieren die XML-Namensräume, aus denen die verwendeten XAML-Bestandteile stammen. Von den 5 Namensräumen erhalten 4 ein Präfix, das den aus einem Namensraum stammenden XAML-Bestandteilen vorangestellt werden muss (z. B. **x:Class**).
- Das Attribut **x:Class** nennt die zum **Window**-Element gehörende Klasse samt .NET - Namensraum, in unserem Fall also `BruchKürzenGui.MainWindow`.
- Das Attribut **mc:Ignorable** mit dem Wert "**d**" ist nur bei Verwendung des GUI-Designers *Blend for Visual Studio* relevant.[1]
- Durch das Attribut **Title** wird die Fensterbeschriftung festgelegt.
- Die Attribute **Height** und **Width** sind für die initiale Fenstergröße zuständig.

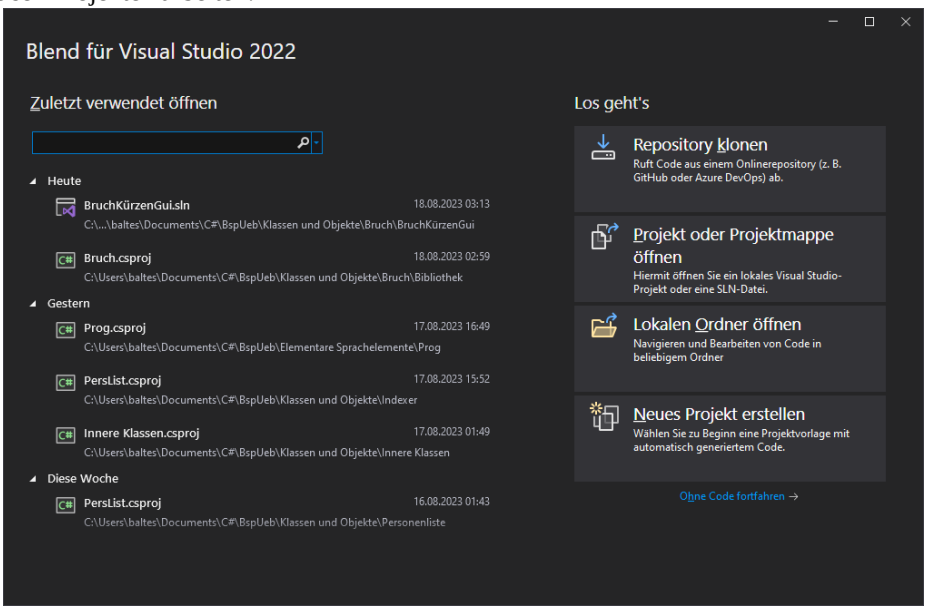
Außerdem enthält das **Window**-Element initial ein **Grid**-Kindelement, das als Container für Steuerelemente dient und später noch ausführlich besprochen wird (siehe Kapitel 12).

Der grafische WPF-Designer ist lediglich ein (sehr praktisches!) Werkzeug zur Bearbeitung der XAML-Datei.

Die XAML-Datei zu unserem Anwendungsfenster heißt **MainWindow.xaml** und beschreibt die Oberfläche von Objekten der Klasse `MainWindow` hinsichtlich Optik und Verhalten („Look and Feel“). Zu dieser **Fensterklasse** gehören auch zwei Quellcodedateien mit den Deklarationen bzw. Definitionen der üblichen Klassen-Mitglieder (Felder, Methoden, Eigenschaften etc.):

---

[1] Es sorgt dafür, dass alle Elemente und Attribute mit dem Namensraumpräfix **d** zur Laufzeit ignoriert werden. Um solche Bestandteile kümmert sich das für Animationen und andere aufwändige GUI-Techniken zuständige Programm *Blend for Visual Studio* (frühere Bezeichnung: *Expression Blend*). Dieses zusammen mit dem Visual Studio installierte Programm (siehe Startmenü) ist primär für Grafikdesigner gedacht, die zusammen mit Software-Entwicklern an großen Projekten arbeiten:

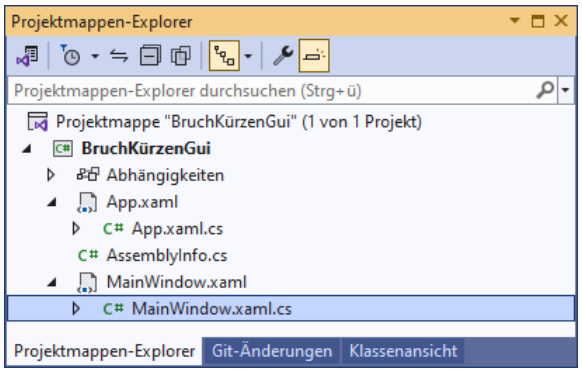


Im Manuskript wird Blend for Visual Studio nicht verwendet.

- **MainWindow.xaml.cs**
  Hier kommen unsere Beiträge zur Funktionalität der Klasse `MainWindow` unter.
- **MainWindow.g.cs**
  Diese Datei wird automatisch durch das Übersetzen der XAML - Deklarationen in C# - Quellcode generiert und vor unseren Blicken verborgen, weil direkte Änderungen durch Programmierer *nicht* vorgesehen sind.

Weitere Details zu den beiden C# - Dateien folgen im Abschnitt 5.13.6.

Der Projektmappen-Explorer zeigt zum Anwendungsfenster die XAML-Datei und die für Programmierer relevante C# - Datei:



Neben der Fensterklasse benötigt eine WPF-Anwendung auch eine **Anwendungsklasse**, die von der BCL-Klasse **Application** abstammt. Erneut sind eine XAML-Deklarationsdatei und zwei C# - Quellcodedateien beteiligt (siehe Abschnitt 5.13.6 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 im Attribut **StartupUri** das beim Programmstart anzuzeigende Fenster (über seine XAML-Datei) festgelegt wird:

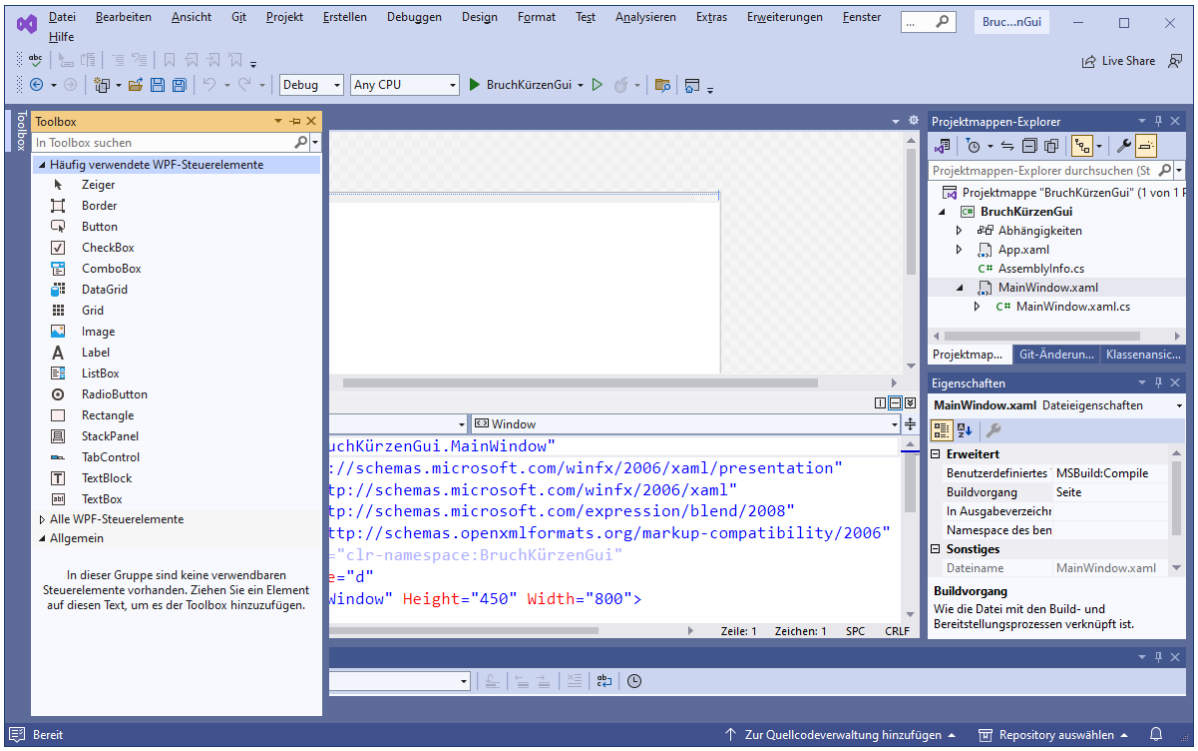
```
<Application x:Class="BruchKürzenGui.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:BruchKürzenGui"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

5.13.3 Steuerelemente aus der Toolbox übernehmen

Öffnen Sie das **Toolbox**-Fenster mit der Tastenkombination **Strg-Alt-X**, mit dem Menübefehl

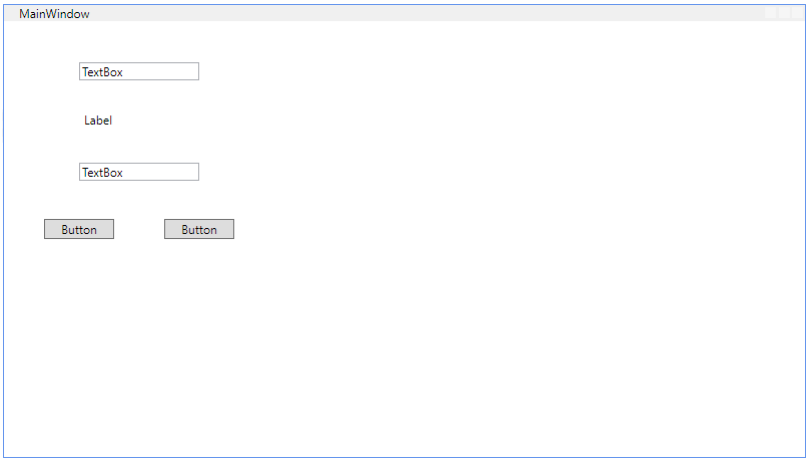
Ansicht > Toolbox

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 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.

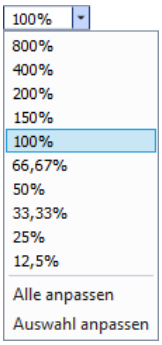
In dieses Ergebnis ist einige Mausarbeit eingeflossen (siehe Abschnitt 3.3.7.2):



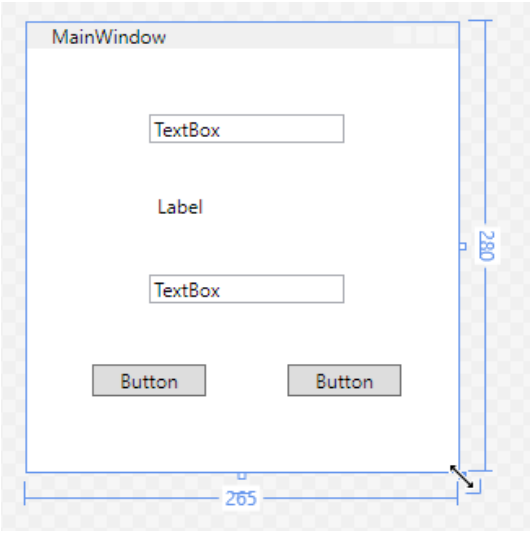
Gleich folgen weitere Hinweise zur Gestaltung der Bedienoberfläche.

5.13.4 Positionen und Größen der Steuerelemente gestalten

Mit dem WPF-Designer kann man die Positionen und Größen der Fensterbestandteile ändern. Zur Erleichterung der Arbeit lässt sich mit dem **Zoom-Werkzeug** des WPF-Designers (unter der Fenster-Vorschau, am linken Rand) eine passende Ansicht einstellen:

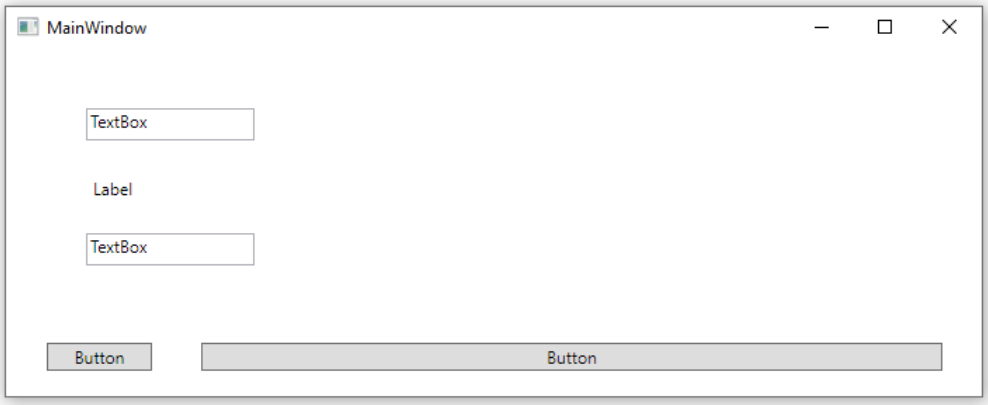


Wählen Sie zunächst für das Hauptfenster eine Größe über den Anfasser unten rechts:

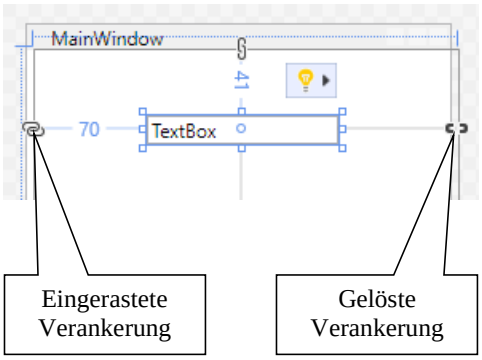


Die aktuelle Breite und Höhe in geräteunabhängigen Pixeln (1/96 Zoll pro Einheit) wird numerisch angezeigt. Achten Sie darauf, dass Sie wirklich das Hauptfenster erwischen (ein Objekt der von **Window** abstammenden Klasse `MainWindow`) und nicht etwa den darin enthaltenen und aus didaktischen Gründen 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.:

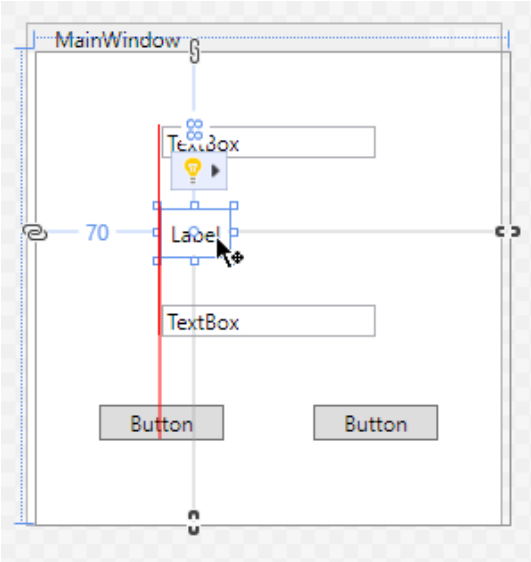


Um eine Verankerung vorzunehmen oder aufzuheben, klickt man auf den zugehörigen Verankerungspunkt. Ist beim Lösen einer Verankerung die gegenüberliegende Seite gerade frei, dann springt die Verankerung dorthin.

Soll ein Steuerelement z. B. vom horizontalen Größenwachstum des Anwendungsfensters profitieren, auf vertikale Änderungen aber nicht reagieren,

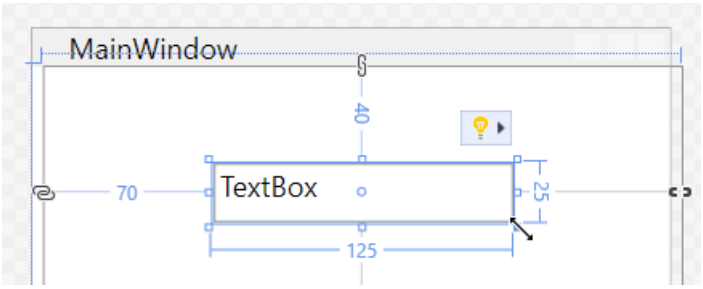
dann verankert man es oben, links und rechts.

Bei der Positions- und Größenanpassung von Steuerelementen helfen **Ausrichtungs- bzw. Führungslinien**, die auftauchen und dabei anziehend wirken, sobald die horizontale oder vertikale Position von potenziellen Ausrichtungskandidaten erreicht wird. Im folgenden Beispiel wurde das **Label**-Objekt bewegt:

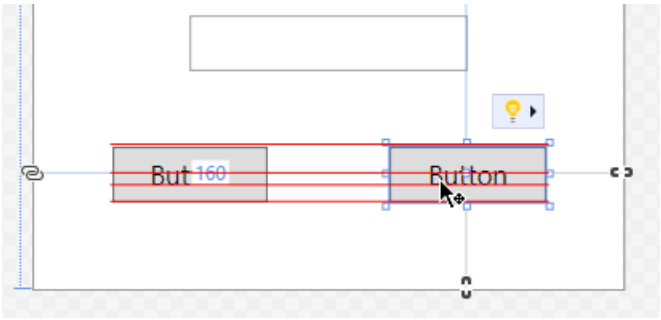


Man kann z. B. so vorgehen, um die Steuerelemente für das Beispielprogramm anzuordnen:

- 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,
  - dieselbe Breite einstellen, wobei die Führungslinien helfen
  - und die Höhe des oberen Textfelds übernehmen.
- Bei den **Button**-Objekten helfen die Führungslinien dabei, eine geeignete Position zu finden:

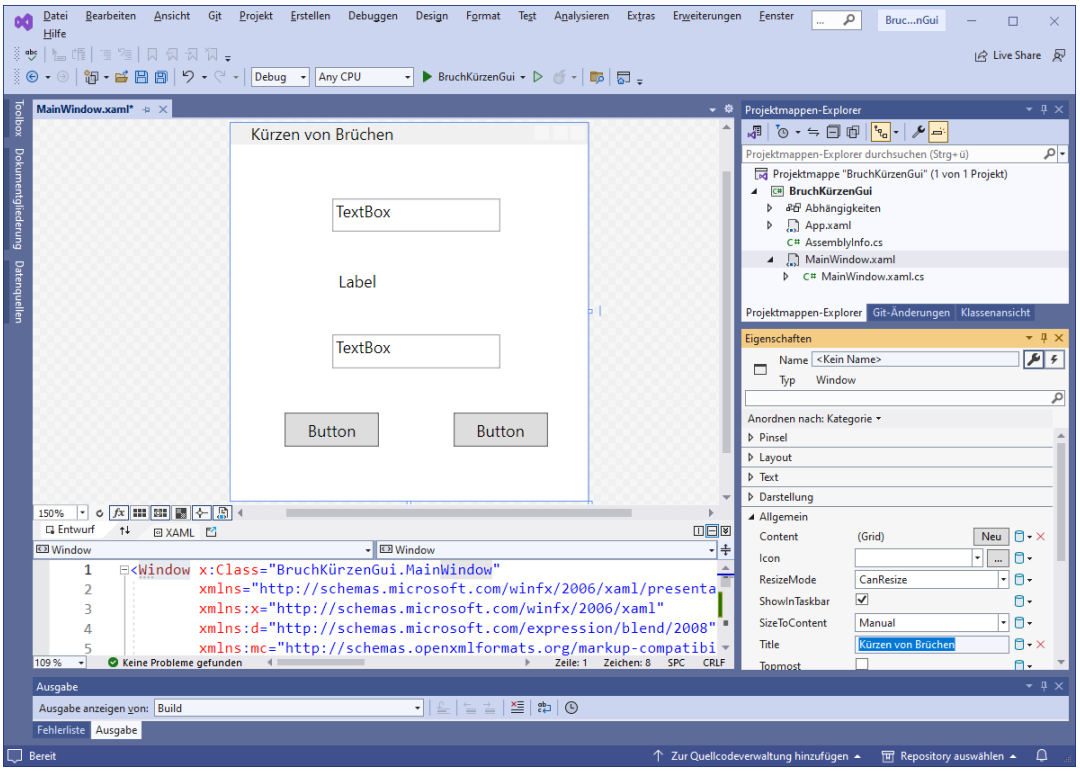


5.13.5 Eigenschaften der Steuerelemente ändern

Im Eigenschaftenfenster der Entwicklungsumgebung, das bei Bedarf mit der Funktionstaste **F4** oder mit dem Menübefehl

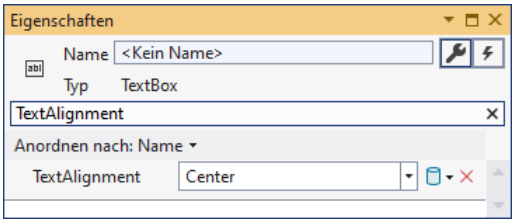
Ansicht > Eigenschaftenfenster

zu öffnen ist, lassen sich diverse Eigenschaften (im Sinne von Abschnitt 5.5) der markierten Objekte festlegen, z. B. der Titel des Anwendungsfensters:

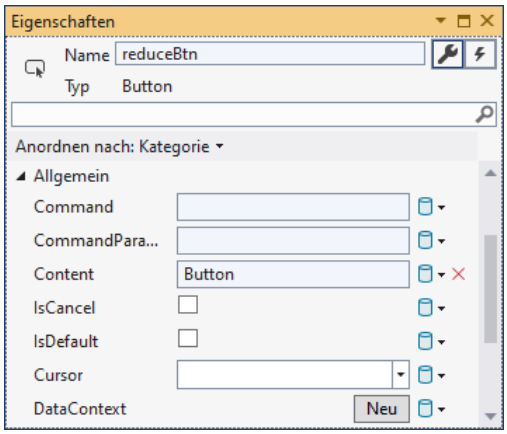


Wenn Sie eine Eigenschaft mit bekanntem Namen bei der voreingestellten **Anordnung nach Kategorie** nicht finden, können Sie ...

- die **Anordnung nach Namen** wählen
- oder das Suchfeld des Eigenschaftsfensters benutzen, z. B.:



Neben den Eigenschaftsmodifikationen ist insbesondere die Möglichkeit von Relevanz, den Instanzvariablennamen eines Steuerelements zu ändern, z. B.:



Wir ändern …

- die Beschriftung (**Content**-Eigenschaft) des linken **Button**-Objekts auf `Kürzen` und die Beschriftung des rechten **Button**-Objekts auf `Beenden`,
- die Namen der Instanzvariablen zu den **TextBox**- und den **Button**-Objekten auf `numTb`, `denomTb`, `reduceBtn` und `closeBtn`,
- die **Content**-Eigenschaft des Labels, um durch eine Serie von Bindestrichen einen behelfsmäßigen Bruchstrich zu erstellen, und setzen die **HorizontalContentAlignment** des Labels auf den Wert **Center**,
- für beide **TextBox**-Objekte die Eigenschaft **Text** auf eine leere Zeichenfolge sowie die Eigenschaften **TextAlignment** und **VerticalContentAlignment** auf den Wert **Center**,
- die **Title**-Eigenschaft des **Window**-Objekts (z. B. auf `Kürzen von Brüchen`),
- die Eigenschaft **ResizeMode** des **Window**-Objekts auf den Wert **NoResize**, sodass sich die Fenstergröße im laufenden Programm nicht verändern lässt,
- die **IsDefault**-Eigenschaft der linken Schaltfläche auf den Wert **True**, sodass diese Schaltfläche im laufenden Programm per **Enter**-Taste angesprochen werden kann.[1]

Beim Verschieben eines Steuerelements im WPF-Designer (siehe Abschnitt 5.13.4) wird übrigens seine **Margin**-Eigenschaft verändert und so für jede Seite ein Randabstand festgelegt.

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, dann kann man die Reihe der Steuerelemente per Tabulator-Taste vorwärts und bei zusätzlich gedrückter **Umschalt**-Taste rückwärts durchlaufen.

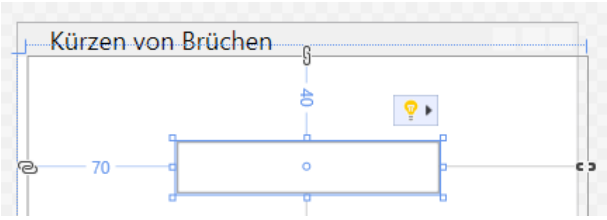
Um *mehrere* Steuerelemente zu markieren, kann man im WPF-Designer ...

- ein Markierungsrechteck um die gewünschten Teilnehmer ziehen,
- bei gedrückter **Strg**-Taste die Steuerelemente nacheinander anklicken.

Die Eigenschaften von mehreren, gleichzeitig markierten Steuerelementen lassen sich in *einem* Arbeitsgang ändern.

Zur *Entwurfszeit* ist eine Eigenschaftsmodifikation bequem per Eigenschaftenfenster zu bewerkstelligen. Gelegentlich ist es erforderlich, eine Steuerelementeigenschaft zur *Laufzeit* (dynamisch) per C# - Anweisung zu ändern.

Über die Glühbirne, die im WPF-Designer zum markierten Steuerelement erscheint,

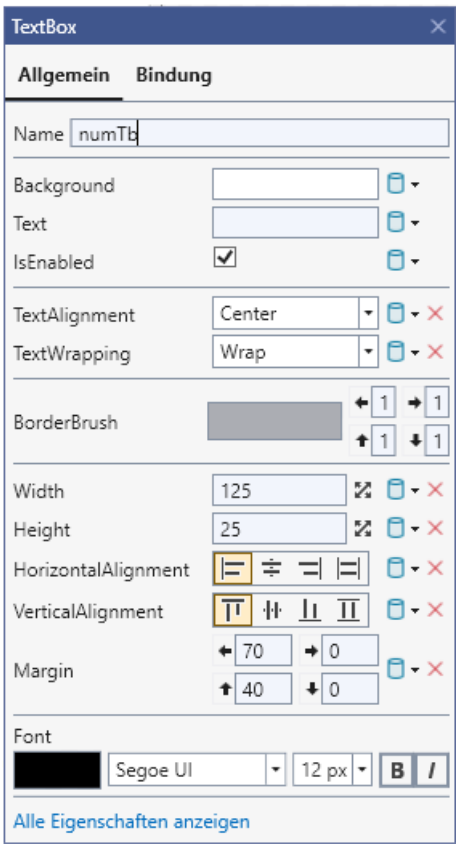


ist auch eine Kontextmenüversion des **Eigenschaften**-Fensters verfügbar, z. B.:

---

[1] Anders als in C# wird das boolesche Literal **True** in XAML großgeschrieben, z. B.:

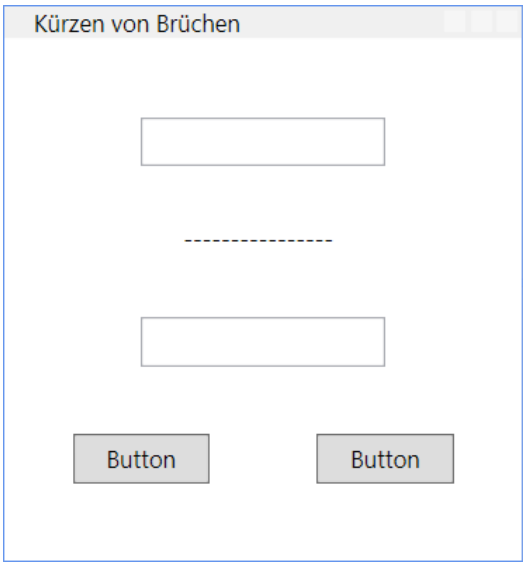
```
<Button x:Name="reduceBtn" Content="Kürzen" ... IsDefault="True"/>
```



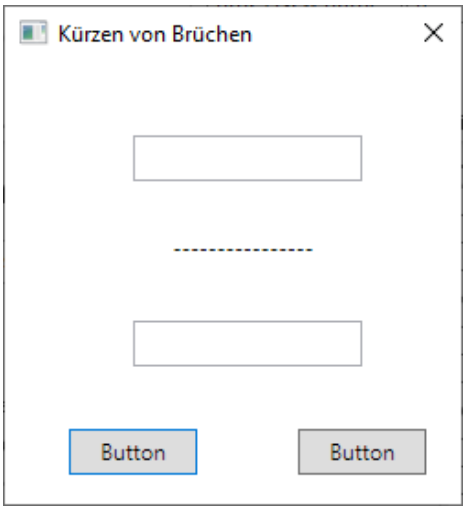
Hier sind die wichtigsten Eigenschaften zu erreichen.

Bei einem Probelauf zur Layout-Kontrolle zeigt sich im Beispiel leider ein am rechten und am unteren Rand beschnittenes Anwendungsfenster:

Vorschau im Visual Studio



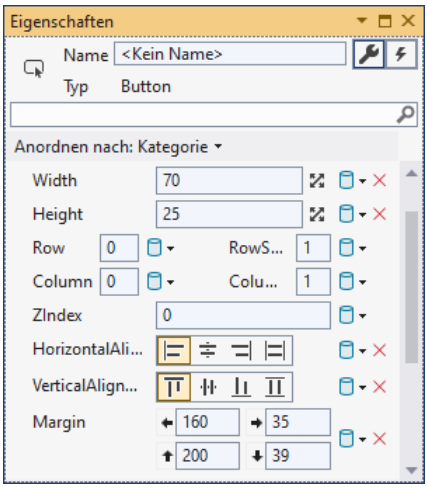
Fenster zur Laufzeit



Aufgrund einer Internet-Recherche ist zu befürchten, dass der Fehler von der Bildschirmauflösung abhängt, sodass die Veränderung der Fenstergröße im XAML-Code vermutlich keine auflösungsunabhängige Lösung ist.[1] Stattdessen werden die folgenden Maßnahmen vorgeschlagen:

[1] https://stackoverflow.com/questions/67972372/why-are-window-height-and-window-width-not-exact-c-wpf

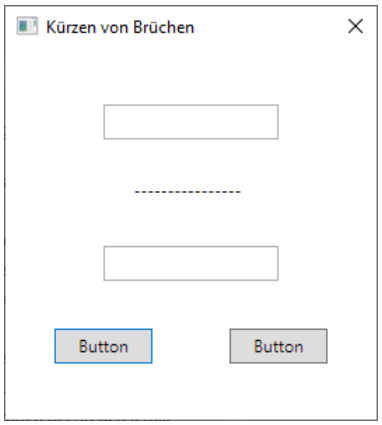
- Der rechte und der untere Rand des rechten **Button**-Objekts erhalten die gewünschten **Margin**-Werte, z. B.:



- Im Fensterklassenkonstruktor (siehe Abschnitt 5.13.6) wird die Eigenschaft **SizeToContent** auf den Wert **SizeToContent.WidthAndHeight** gesetzt, sodass die korrekte Fenstergröße zur Laufzeit passend berechnet wird:

```
public MainWindow() {
    InitializeComponent();
    SizeToContent = SizeToContent.WidthAndHeight;
}
```

Damit ist das Problem behoben (hoffentlich für jede Bildschirmauflösung):

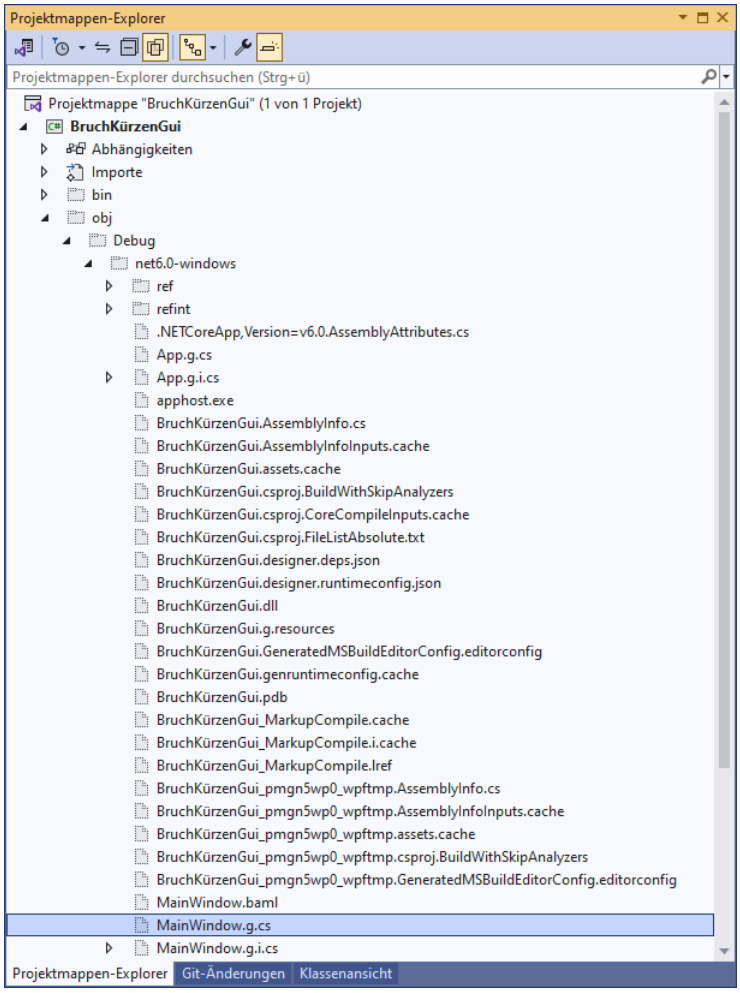


5.13.6 Automatisch erstellter und gepflegter Quellcode

Aufgrund Ihrer kreativen Tätigkeit beim GUI-Design erzeugt die Entwicklungsumgebung im Hintergrund Quellcode zur Fensterklasse `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, die bereits erwähnt worden sind:

- **MainWindow.xaml.cs** im Projektordner
  Hier landen die von Ihnen erstellten Methoden (siehe unten).
- **MainWindow.g.cs** im Projektunterordner **...\obj\Debug\net6.0-windows**[1]
  Hier landet der beim Interpretieren der XAML-Datei **MainWindow.xaml** entstehende C# - Quellcode. In dieser Datei sollten Sie keine Änderungen vornehmen, damit das Visual Studio nicht aus dem Tritt kommt. Daran erinnert der Namensbestandteil *g* für *generated*.[2]

Wer die Datei **MainWindow.g.cs** im Visual Studio 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**:

---

1 Das gilt bei Verwendung der Debug-Konfiguration (vgl. Abschnitt 3.3.8.3).

2 Neben der Datei **MainWindow.g.cs** gibt es im selben Ordner noch eine Datei mit dem Namen **MainWindow.g.i.cs**. Während die Datei **MainWindow.g.cs** beim Erstellen des Programms entsteht, wird die Datei **MainWindow.g.i.cs** schon *vor* dem Erstellen des Programms angelegt und sukzessiv an gespeicherte Änderungen im XAML-Code angepasst.

```
using System;
using System.Collections.Generic;
   .  .  .
using System.Windows.Shapes;

namespace BruchKürzenGui {
   /// <summary>
   /// Interaction logic for MainWindow.xaml
   /// </summary>
   public partial class MainWindow : Window {
      public MainWindow() {
         InitializeComponent();
      }
   }
}
```

Der `MainWindow`-Quellcode in der Datei **MainWindow.xaml.cs** enthält einen Konstruktor, der die Methode **InitializeComponent()** aufruft. Diese Methode wird in der Datei **MainWindow.g.cs** vom Visual Studio implementiert.

Aufgrund Ihrer Tätigkeit im WPF-Designer enthält die Fensterklasse `MainWindow` im Sinne der im Abschnitt 5.9 behandelten Komposition mehrere Objekte anderer Klassen, die Steuerelemente der grafischen Bedienoberfläche repräsentieren. In der Datei **MainWindow.g.cs** finden sich die Deklarationen der zugehörigen Instanzvariablen, nachdem das Projekt erstellt worden ist:

```
internal System.Windows.Controls.TextBox numTb;
internal System.Windows.Controls.TextBox denomTb;
internal System.Windows.Controls.Button reduceBtn;
internal System.Windows.Controls.Button closeBtn;
```

Über den Modifikator **internal** wird der Zugriff durch alle Klassen im eigenen Assembly erlaubt (vgl. Abschnitt 5.12).

Neben der Klasse `MainWindow` mit der XAML-Datei **MainWindow.xaml** und dem C# - Quellcode-Duo **MainWindow.xaml.cs** und **MainWindow.g.cs** enthält das Projekt noch die Klasse `App` mit ...

- der Basisklasse **Application** aus der BCL,
- der XAML-Datei **App.xaml**,
- den Quellcodedateien **App.xaml.cs** und **App.g.cs**.

Wer die **Main()** - Methode zu unserer WPF-Anwendung vermisst, findet sie in der Datei **App.g.cs**:

```
public static void Main() {
    BruchKürzenGui.App app = new BruchKürzenGui.App();
    app.InitializeComponent();
    app.Run();
}
```

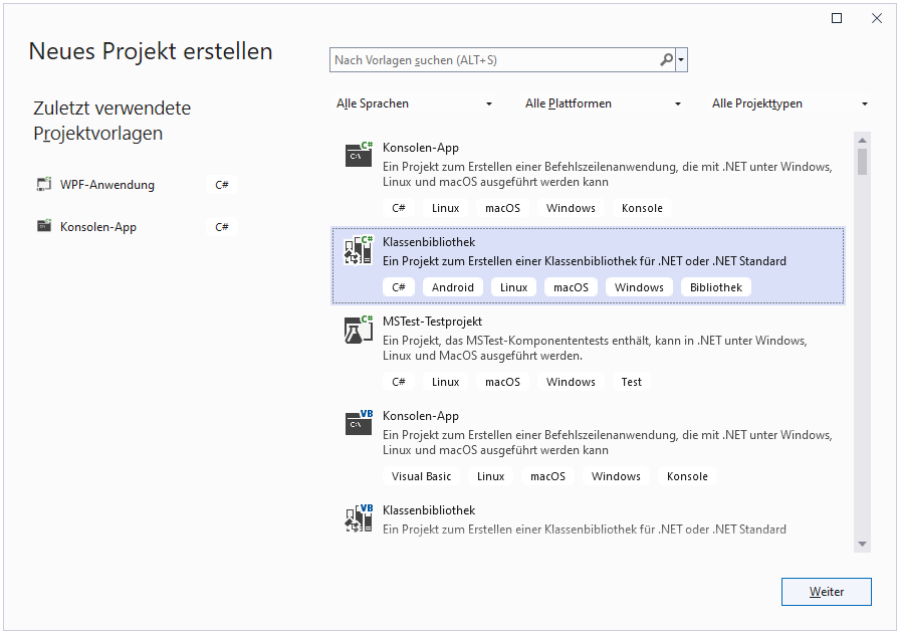
Weitere Details zu den vom Visual Studio bei WPF-Projekten gepflegten Quellcodedateien folgen im Kapitel 12.

5.13.7 Bibliotheks-Assembly mit der Bruch-Klasse einbinden

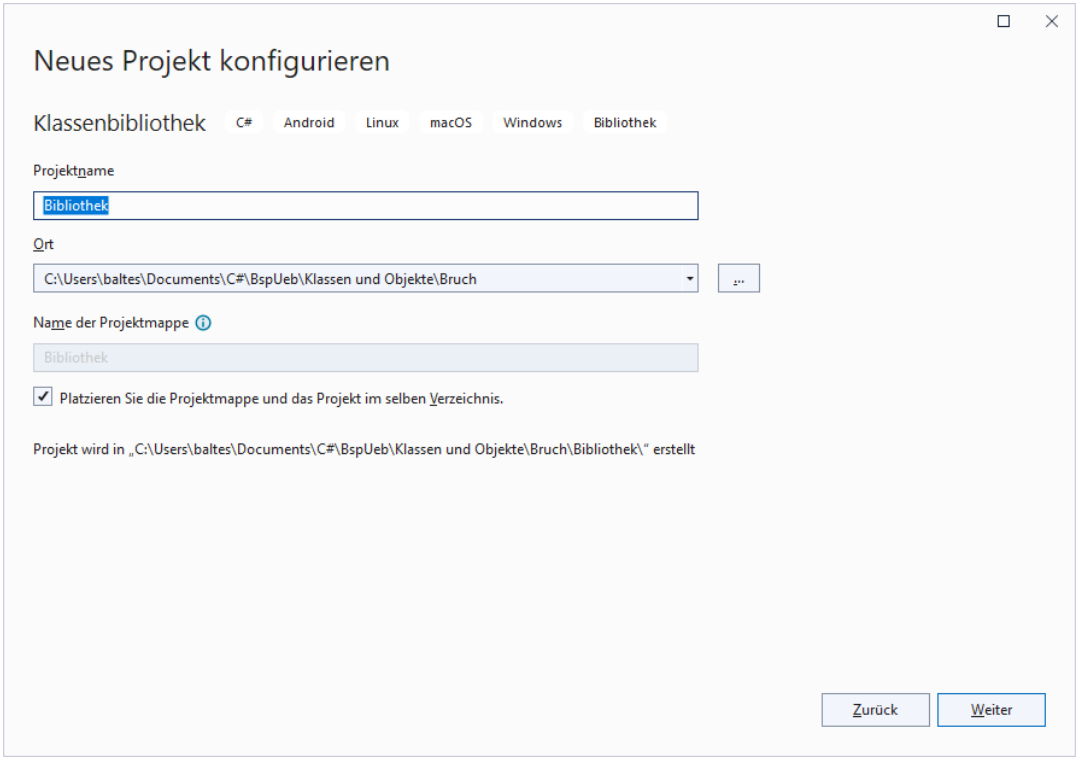
Zur Erstellung eines Bibliotheks-Assemblies aus der `Bruch`-Klassen – Entwicklungsstufe mit statischen Elementen starten wir im Visual Studio mit dem Menübefehl

Datei > Neu > Projekt

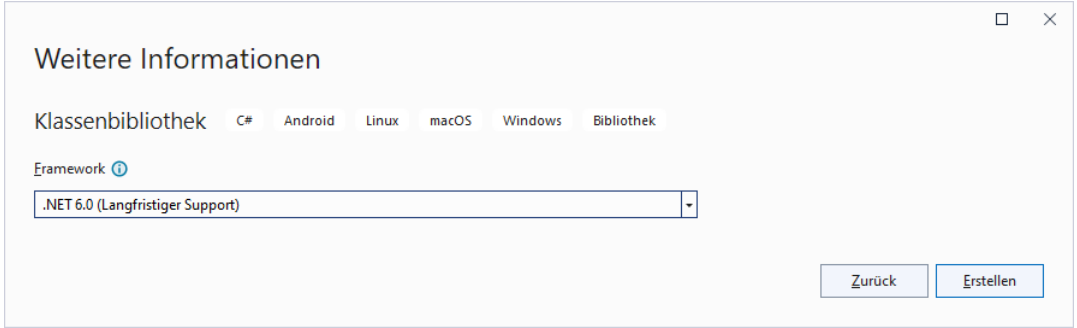
ein neues Projekt unter Verwendung der Vorlage **Klassenbibliothek**:



Wie wählen einen **Projektnamen** und einen **Ort**



sowie eine .NET – Version mit langfristiger Unterstützung:



Nach dem Menübefehl

Projekt > Eigenschaften > Anwendung > Allgemein

ändern wir den **Assemblynamen** und den **Standardnamespace**, z. B.:

Wir kopieren die folgende Datei

…\BspUeb\Klassen und Objekte\Bruch\b5 Statische Member\Bruch.cs

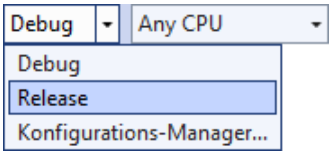
in den Projektordner und fügen eine **namespace**-Direktive am Anfang der Quellcodedatei ein, z. B.

```
namespace de.bebagoe.cs;

public class Bruch {
  . . .
}
```

Dabei verzichten wir auf einen durch geschweifte Klammern begrenzten Anwendungsbereich der **namespace**-Direktive, was seit C# 10 erlaubt ist (siehe Abschnitt 2.6.1). Die vom Assistenten angelegte Quellcodedatei **Class1.cs** ist überflüssig und sollte gelöscht werden.

Durch die Verwendung eines Namensraums wird die Klasse `Bruch` aus dem globalen Namensraum herausgehalten, sodass nicht mit Namenskollisionen zu rechnen ist. Die im Beispiel gewählte Namensraumbezeichnung startet mit dem Namen einer Internetdomäne (mit den Namenssegmenten in umgekehrter Reihenfolge).

Weil die Klasse `Bruch` als bewährt gelten kann, wählen wir die **Release**-Konfiguration über das folgende Bedienelement in der Visual Studio - Symbolleiste:



Zum Erstellen des Bibliotheks-Assemblies, das keine Startklasse besitzt, verwenden wir den Menübefehl

Erstellen > Projektmappe erstellen

Gehen Sie folgendermaßen vor, um Objekte der Klasse `Bruch` im Kürzungsprogramm mit WPF-Bedienoberfläche nutzen zu können:

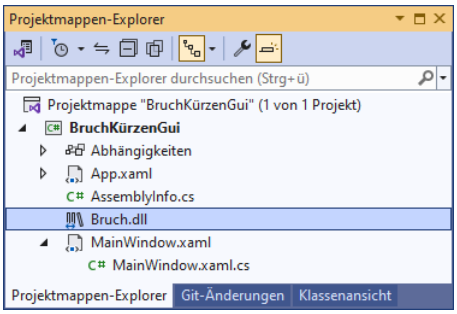
- Kopieren Sie das Bibliotheks-Assembly mit der Klasse `Bruch`

  …\BspUeb\Klassen und Objekte\Bruch\Bibliothek\bin\Release\net6.0\Bruch.dll

  in den Ordner des WPF-Projekts, also z. B. nach

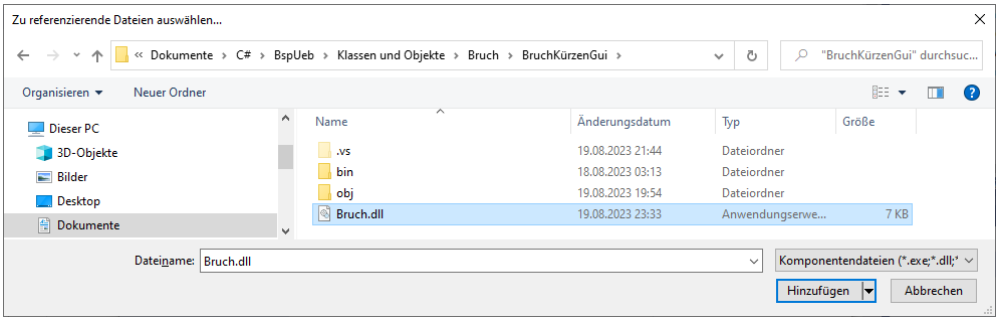
  …\BspUeb\Klassen und Objekte\Bruch\BruchKürzenGui

  Der **Projektmappen-Explorer** zeigt das Bibliotheks-Assembly an:



Beim Erstellen der Anwendung landet die Datei **Bruch.dll** im selben Ordner wie das Anwendungs-Assembly **BruchKürzenGui.exe**.

- Damit die Datei **Bruch.dll** beim Erstellen des Bruchkürzungsprogramms gefunden wird, muss sie in die Liste mit den Abhängigkeiten des Projekts aufgenommen werden:
  - Wählen Sie aus dem Kontextmenü zum Projekteintrag mit den **Abhängigkeiten** das Item **Projektverweis hinzufügen**.
  - Klicken Sie im erscheinenden **Verweis-Manager** auf den Schalter **Durchsuchen**.
  - Wählen Sie die Datei **Bruch.dll**, z. B.:



- Ergänzen Sie am Anfang der Quellcodedatei **MainWindow.xaml.cs** mit der partiellen Definition der Fensterklasse eine **using**-Direktive zum Namensraum, in den Sie die Klasse `Bruch` eingeordnet haben, z. B.:

```
using de.bebagoe.cs;

namespace BruchKürzenGui {
    public partial class MainWindow : Window {
      . . .
    }
}
```

5.13.8 Ereignisbehandlungsmethoden anlegen

Zunächst erstellen wir zu den beiden Befehlsschaltern jeweils eine Methode, die durch das Betätigen (z. B. per Mausklick) gestartet wird. Setzen Sie im Formulardesigner einen Doppelklick auf den Befehlsschalter `reduceBtn` (mit der Beschriftung *Kürzen*), sodass die Entwicklungsumgebung in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `reduceBtn_Click()` der Klasse `MainWindow` mit leerem Rumpf anlegt

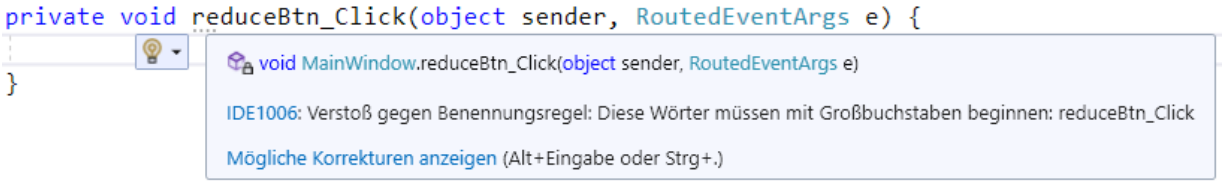
```
private void reduceBtn_Click(object sender, RoutedEventArgs e) {

}
```

und die Datei im Editor öffnet.

Außerdem wird die Methode in der XAML-Datei zum Anwendungsfenster dem **Button**-Ereignis **Click** zugeordnet:

```
<Button x:Name="reduceBtn" Content="Kürzen" HorizontalAlignment="Left"
   Margin="35,199,0,0" Width="70" Height="25" VerticalAlignment="Top"
   IsDefault="True" Click="reduceBtn_Click"/>
```

Lässt man den Mauszeiger auf der punktierten Unterstreichung des Methoden-Namensanfangs verharren, dann äußert das Visual Studio eine Kritik an der eigenen Produktion:



Für die Namen von Methoden sollte in C# grundsätzlich das Pascal Casing (mit großem Anfangsbuchstaben) benutzt werden (vgl. Abschnitte 4.1.6 und 5.3.1). Weil es sich nur um eine Konvention handelt, verzichten wir auf eine Korrektur der Assistentenproduktion, die uns bei konsequenter Umsetzung zu ständigen Eingriffen zwingen würde. Man könnte das Problem vermeiden durch die Entscheidung, bei Namen von Feldern mit der Zugriffebene **internal** (abweichend von den Empfehlungen im Abschnitt 5.2.2) das Pascal Casing zu verwenden.

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 im Abschnitt 5.14).

Tritt im **try**-Block der **try-catch** - Anweisung (vgl. Abschnitt 13.2.1) eine Ausnahme auf (z. B. beim Versuch, eine irreguläre Benutzereingabe zu konvertieren), dann wird der **catch**-Block ausgeführt, und es erscheint eine Fehlermeldung. Im Aufruf der **MessageBox**-Methode **Show**() sorgen Werte der Enumerationen (siehe Abschnitt 6.4) **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 Klick-Ereignisbehandlungsmethode, und ergänzen Sie einen Aufruf der **Window**-Methode **Close()**, die das Hauptfenster schließt und damit das Programm beendet, z. B.:

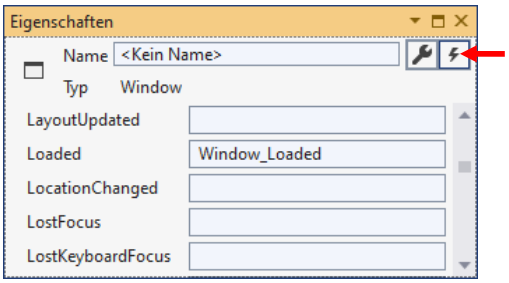
```
private void closeBtn_Click(object sender, RoutedEventArgs e) {
    Close();
}
```

Wie man dafür sorgt, dass nach dem Start des Programms das Zähler-Textfeld den Tastaturfokus erhält, sodass der Benutzer unmittelbar mit der Werteingabe beginnen kann, ohne zuvor den Tastaturfokus (z. B. per Maus) setzen zu müssen, wurde schon im Abschnitt 4.5.11 beschrieben. Man fordert das zu privilegierende Steuerelement über die Methode **Focus()** der Klasse **UIElement** auf, den Fokus zu übernehmen, z. B.:

```
numTb.Focus();
```

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. Um sicher das Fenster (ein **Window**-Objekt) und nicht den darin enthaltenen Container aus der Klasse **Grid** zu erwischen, klickt man auf das Start-Tag zum **Window**-Element im XAML-Code.
- Wechseln Sie im Eigenschaftenfenster zur Liste mit den Ereignissen, die von der Klasse **Window** angeboten werden.



- Setzen Sie einen Doppelklick auf das Texteingabefeld zum Ereignis **Loaded**. Wenn das Anwendungsfenster das Ereignis **Loaded** „feuert“, ist der richtige Moment für Initialisierungsarbeiten gekommen.
- Daraufhin wird in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `Window_Loaded()` zur Fensterklasse `MainWindow` angelegt:

```
private void Window_Loaded(object sender, RoutedEventArgs e) {

}
```

  Außerdem wird die Methode in der XAML-Datei zum Hauptfenster dem Ereignis **Loaded** zugeordnet:

```
<Window x:Class="BruchKürzenGui.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   . . .
    Title="Kürzen von Brüchen" Height="280" Width="265" ResizeMode="NoResize"
    Loaded="Window_Loaded">
```

- Ergänzen Sie im Rumpf der Methode `Window_Loaded()` den oben beschriebenen **Focus()** – Aufruf:

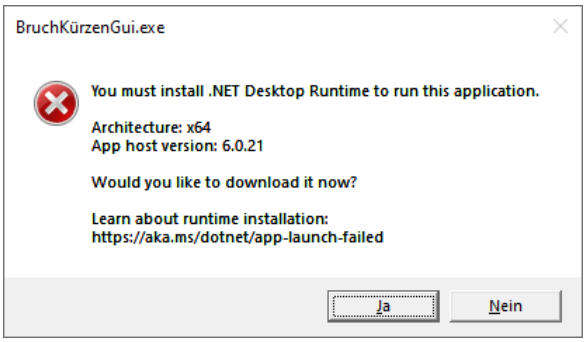
```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    numTb.Focus();
}
```

Veranlassen Sie mit der Tastenkombination **Strg+F5** das Erstellen und die Ausführung der fertigen Anwendung.

Weil wir die **Debug**-Erstellungskonfiguration verwendet haben (vgl. Abschnitt 3.3.8.3), befinden sich die ausführbare Assembly-Datei **BruchKürzenGui.dll** und der Windows-Starthelfer **BruchKürzenGui.exe** im folgenden Projektunterordner:

…\bin\Debug\net6.0-windows

Zum „Installieren“ des Programms genügt es, diesen Ordner zum Zielrechner zu befördern. Dabei darf die beim Erstellen automatisch in den Programmordner kopierte Assembly-Datei **Bruch.dll** mit der Klasse `Bruch` nicht vergessen werden. In der Datei **BruchKürzenGui.deps.json** wird die Abhängigkeit des Programms von **Bruch.dll** beschrieben. Die Abhängigkeitskonfigurationsdatei wird aber beim Programmstart nicht benötigt, um das im Ordner des ausführbaren Programms abgelegte Bibliotheks-Assembly zu finden. Ohne die Datei **BruchKürzenGui.runtimeconfig.json** mit einer Beschreibung der benötigten Laufzeitumgebung erscheint beim Startversuch nur die folgende Fehlermeldung:



Ein Visual Studio – Projekt mit dem WPF-Bruchkürzungsprogramm befindet sich im Ordner:

…\BspUeb\Einleitungsbeispiel Bruchrechnen\GUI

5.14 Übungsaufgaben zum Kapitel 5

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**- Feld und einer Eigenschaft ohne **set** - Implementierung, die man auch als *get-only* - Eigenschaft bezeichnet.

3) Im folgenden Programm soll die statische `Bruch`-Eigenschaft `Anzahl` ausgelesen werden:

```
using System;
class Bruchrechnung {
  static void Main() {
    var b1 = new Bruch();
    var b2 = new Bruch();
    Console.WriteLine("Jetzt sind wir " + Bruch.Anzahl);
  }
}
```

Es liegt die folgende die 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:

```
Stack overflow.
Repeat 19272 times:
--------------------------------
   at Bruch.get_Anzahl()
--------------------------------
   at Bruch..ctor()
   at Bruchrechnung.Main()
```

Offenbar hat sich ein Logikfehler in die Eigenschaftsdefinition eingeschlichen, den der Compiler nicht reklamiert.

4) Für diese Aufgabe werden Grundbegriffe der linearen Algebra benötigt, wobei die beteiligten Formeln in der Aufgabenbeschreibung enthalten sind: Erstellen Sie eine Klasse für Vektoren im $\mathbb{R}^2$ (in der der reellen Zahlenebene), die mindestens über Methoden bzw. Eigenschaften mit den 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 1 **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:

  $$|\tilde{x}| = \sqrt{\tilde{x}_1^2 + \tilde{x}_2^2} = \sqrt{\left(\frac{x_1}{\sqrt{x_1^2 + x_2^2}}\right)^2 + \left(\frac{x_2}{\sqrt{x_1^2 + x_2^2}}\right)^2} = \sqrt{\frac{x_1^2}{x_1^2 + x_2^2} + \frac{x_2^2}{x_1^2 + x_2^2}} = 1$$

  Bei einem Vektor mit einer Länge nahe 0 (z. B. $< 10^{-14}$, vgl. Abschnitt 4.5.4) sollte das Normieren unterbleiben.

- Vektoren **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:

  $$\begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

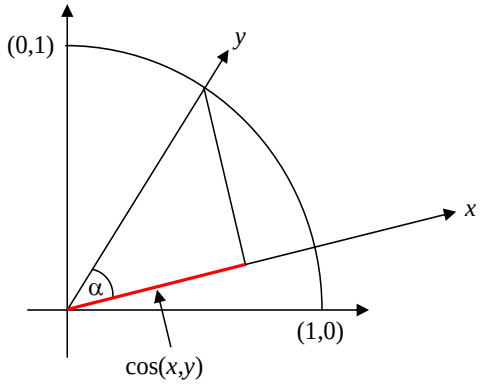
- **Skalarprodukt** zweier Vektoren ermitteln

  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 (linksherum) 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 der folgenden Formel unter Verwendung der Kreiszahl π in Grad umrechnen (*deg*):

$$deg = \frac{rad}{2\pi} \cdot 360$$

- **Rotation** eines Vektors um einen bestimmten Winkel
  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 der folgenden Formel unter Verwendung der Kreiszahl π 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:

[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.

Wer Spaß an der Programmierpraxis zur Lösung mathematischer Aufgaben gefunden hat, kann auch noch ein rekursives Verfahren zur Berechnung der Fibonacci-Zahlen entwerfen. Die Fibonacci-Folge entsteht, wenn ausgehend von den Zahlen 0 und 1 alle weiteren Folgenglieder als Summe der beiden Vorgänger berechnet werden:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

6) Ersetzen Sie beim GUI-Bruchkürzungsprogramm im Abschnitt 5.13 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 (2019).

6 Weitere .NETte Typen

Nachdem wir uns ausführlich mit elementaren Datentypen und mit Klassen beschäftigt haben, wird in diesem Kapitel 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)
- Tupel als syntaktisch bequeme Strukturen zur Verwaltung von Elementen mit unterschiedlichen Datentypen
- Record-Datentyp für Objekte, die hauptsächlich zur Verwaltung unveränderlicher Daten dienen und nur einfache Handlungskompetenzen benötigen

6.1 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 C# mit den *Strukturen* auch *Werttypen* zur Verfügung, die in manchen Situationen bei geringerem Ressourcenverbrauch „echte“ Klassen ersetzen und somit die Performanz der Software steigern können. Im Abschnitt 6.1.4 werden wir in einem Beispielprogramm den Zeit- und Speicheraufwand für Objekte bzw. Strukturinstanzen untersuchen.

Den sehr flexiblen Klassen, mit denen z. B. die Programmiersprache Java (neben den elementaren Datentypen) auskommt, die ressourcen-schonenden Strukturen mit abweichender Syntax und Semantik zur Seite zu stellen, hilft bei der Entwicklung leistungskritischer Programme, aber nicht beim Einstieg in das Programmieren. Das stark von der CPU-Technik abstrahierende objektorientierte Paradigma wird hier ergänzt (man könnte auch sagen: *belastet*) durch eine Berücksichtigung leistungsrelevanter Merkmale der Computer-Technik (unterschiedliche Kosten für Strukturinstanzen bzw. Objekte). Zum Glück sind wir …

- mit dem Heap (zur Aufbewahrung der Methoden-übergreifend persistierenden Objekte)
- und dem Stack (zur Aufbewahrung von lokalen Variablen während der Methodenausführung)

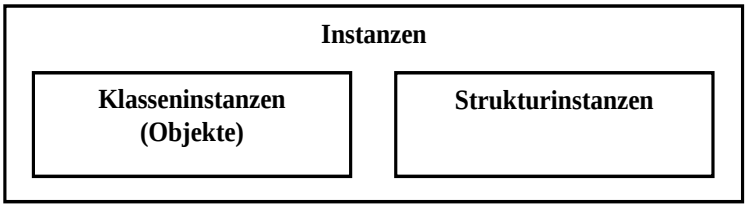
schon vertraut. Als Bestandteile eines Objekts sind Strukturen (und elementare Datentypen) aber auch auf dem Heap anzutreffen. Wird z. B. ein Array (siehe Abschnitt 6.2) mit 1000 Elementen angelegt, dann …

- führt die Verwendung eines Strukturtyps zu einem Array-Objekt mit 1000 hintereinander abgelegten, leicht auffindbaren Strukturinstanzen,
- führt die Verwendung eines Klassentyps zu einem Array-Objekt und 1000 weiteren Objekten, die einzeln auf dem Heap abgelegt werden, während der Array lediglich die Adressen der Elementobjekte enthält.

Der hier zu erwartende Einspareffekt durch die Verwendung eines Strukturtyps (z. B. beim Erstellen und Abräumen der Instanzen) motiviert vermutlich Programmiereinsteiger, den zusätzlichen Lernaufwand in Kauf zu nehmen.

Beim Design eines Strukturtyps können mit Ausnahme des Finalisierers dieselben Member eingesetzt werden wie bei einer Klassendefinition (Felder beliebigen Typs, Methoden, Eigenschaften, Konstruktoren, ein statischer Konstruktor, usw.). Eine Variable vom Typ einer Struktur enthält jedoch keine *Referenz* auf ein Heap-Objekt, sondern alle Feldinhalte ihres Typs. Wie bei Variablen mit einem elementaren Datentyp liegt keine Referenz-, sondern eine **Wertsemantik** vor.

Für ein Individuum nach dem Bauplan einer Struktur wird im Manuskript *nicht* der Begriff *Objekt*, sondern der allgemeinere Begriff *Instanz* verwendet:



Von *Instanzvariablen* und *-methoden* sprechen wir bei Objekten *und* bei Strukturinstanzen.

Eine Struktur eignet sich als Datentyp in der folgenden Konstellation:

- Kleine Instanzen
  Der Typ ist relativ einfach aufgebaut, hat also nur wenige Instanzvariablen. Microsoft empfiehlt für Strukturinstanzen eine Gesamtgröße von weniger als 16 Bytes.[1]
- Wertsemantik erwünscht
  Bei einem Methodenaufruf wird eine als Wertaktualparameter verwendete Strukturinstanz komplett kopiert, während von einem Parameter*objekt* nur die Adresse übergeben wird. Das Original der kopierten Strukturinstanz wird durch den Methodenaufruf nicht verändert, was beim Parameterobjekt hingegen möglich ist.
- Unveränderliche Instanzen
  Es wird empfohlen, Strukturinstanzen als **unveränderlich** (engl. *immutable*) zu konzipieren, so z. B. auf einer Microsoft Webseite mit .NET - Designrichtlinien:[2]

  DO NOT define mutable value types.

  Aufgrund der Wertsemantik kann es bei einer Struktur nämlich leicht passieren, dass man lediglich eine *Kopie* modifiziert an Stelle der eigentlich zu ändernden Instanz. Um die Unveränderlichkeit eines Typs zu erreichen, muss man die Datenkapselung realisieren (also die Instanzvariablen schützen) und außerdem auf Methoden und Eigenschaften verzichten, die Instanzvariablen ändern. Seit C# 7.2 kann man in einer Strukturdefinition dem Compiler durch den Modifikator **readonly** mitteilen, dass die Unveränderlichkeit der Strukturinstanzen realisiert werden soll (siehe Abschnitt 6.1.1).

  Die Unveränderlichkeit von Strukturinstanzen ist allerdings kein Dogma, und Microsoft hält sich auch nicht immer an die eigene Empfehlung (siehe unten).
- Nullinitialisierung akzeptabel
  Bei einer Struktur muss sichergestellt sein, dass die Nullinitialisierung aller Felder zu einer regulären Instanz führt (siehe Abschnitt 6.1.1).
- Keine Vererbung erforderlich
  Den Strukturen fehlt die Möglichkeit, per Vererbung (siehe Kapitel 7) eine Hierarchie spezialisierter Typen aufzubauen. Das Implementieren von Schnittstellen (siehe Kapitel 9) ist aber möglich.

Werden von einem Typ viele Instanzen benötigt, kann sich die Vermeidung von Objektkreationen durch die Verwendung eines Strukturtyps lohnen. Typische Anwendungsbeispiele für Strukturen:

---

1 https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/choosing-between-class-and-struct

2 https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/struct

- Punkte in einem zwei- oder dreidimensionalen Raum
  Ein Beispiel ist die von Microsoft definierte BCL-Struktur **Point** im Namensraum **System.Windows**. Sie besitzt allerdings abweichend von einer obigen Empfehlung öffentliche Eigenschaften mit **set**-Methode für die Koordinaten und ist folglich veränderlich.
- Komplexe Zahlen[1]
  Bei der Struktur **Complex** im Namensraum **System.Numerics** hat sich Microsoft an die eigene Empfehlung gehalten, Strukturen als unveränderlich zu konzipieren. 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.
- Elementare Datentypen
  Im Abschnitt 6.1.5 wird sich herausstellen, dass es sich auch bei den elementaren Datentypen (z. B. **int**, **double**, **char**) um Strukturen handelt.

Anschließend werden einige Empfehlungen für die Entscheidung zwischen einer Klasse und einer Struktur beim Entwurf eines neuen Datentyps zusammengetragen. Microsoft sagt dazu:[2]

- Typically, you use structure types to design small data-centric types that provide little or no behavior.
- If you're focused on the behavior of a type, consider defining a class.

Griffiths (2013, S. 97) empfiehlt:

- 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 6.3.1), ist bei einem Typ mit unveränderlichen Instanzen noch lange nicht entschieden, dass eine Struktur definiert werden sollte. Beim Datentyp **String** hat wohl u.a. das außerordentlich umfangreiche API (mit einer dreistelligen Zahl von Methoden) die Entscheidung für eine Klasse motiviert.

Mit dem von Albahari (2022, S. 129) genannten zentralen Kriterium für die Wahl einer Struktur beschäftigt sich der Abschnitt 6.1.1:

> A struct is appropriate when value-type semantics are desirable.

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 6.3.1).

Seit C# 7.2 kann man bei der Definition einer Struktur durch den Modifikator **ref** verhindern, dass Instanzen auf den Heap gelangen (z. B. per Boxing oder als Array-Elemente). Von dieser Option profitieren vor allem die zur Leistungsoptimierung dienenden generischen BCL-Strukturen **Span<T>** und **ReadOnlySpan<T>**, die im Abschnitt 15.3.2 zusammen mit dem **ref**-Modifikator vorgestellt werden (zu generischen Typen siehe Kapitel 8).

---

[1] Dieser mathematische Begriff meint Paare aus reellen Zahlen, für die spezielle Rechenregeln gelten. Wer nicht mathematisch vorbelastet ist, kann das Beispiel ignorieren.

[2] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct

6.1.1 Implikationen der Wertsemantik von Strukturen

In diesem Abschnitt wird der Unterscheid zwischen der *Referenzsemantik* der Klassen und der *Wertsemantik* der Strukturen demonstriert. Es wird sich zeigen, warum aus der Wertsemantik der Strukturen die Empfehlung resultiert, Strukturinstanzen als unveränderlich zu entwerfen. Wir definieren 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]

```
public struct Punkt {
   double x, y;

   public Punkt(double x_, double y_) {
      x = x_;
      y = y_;
   }

// Nutzlose bzw. gefährlich irreführende statische Methode mit Punkt-Parameter:
   public static void Bewegen(Punkt p, double hor, double vert) {
      p.x += hor;
      p.y += vert;
   }

   public void Bewegen(double hor, double vert) {
      x += hor;
      y += vert;
   }

   public string Pos() {
      return $"({x}; {y})";
   }
}
```

Wir ignorieren die Empfehlung, 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 den ersten Parameter zum Referenzparameter machen (vgl. Abschnitt 5.3.1.3.2).
- Die Instanzmethode `Bewegen()` erfüllt ihren Zweck.

Wie die Objekte von Klassen können auch Strukturinstanzen per **new**-Operator unter Verwendung eines Konstruktors initialisiert werden. Beim folgenden Einsatz der `Punkt`-Struktur wird die Variable `p1` über den expliziten Konstruktor mit den Koordinaten (1; 2) initialisiert. Der Punkt `p2` erhält (vom *nicht* verloren gegangenen) Standardkonstruktor eine Initialisierung mit den Koordinaten (0; 0). Der Punkt `p3` erhält eine *Kopie* von `p1` (mit allen Instanzvariablen):

---

[1] Leser mit C# - Erfahrung werden eventuell vorschlagen, die Methode `Pos()` durch eine **ToString()** – Überschreibung zu ersetzen. Das würde aber einen Vorgriff auf das Kapitel 7 über die Vererbung erzwingen. Im Manuskript werden Vorgriffe nach Möglichkeit vermieden.

Quellcode (mit `Punkt` als Struktur)	Ausgabe

`var p1 = new Punkt(1, 2);`
`var p2 = new Punkt();`
`var p3 = p1;`
`Punkt.Bewegen(p2, 1, 1); // nutzlos`
`p1.Bewegen(1, 0); // klappt, aber ohne Effekt auf p3`
`Console.WriteLine("p1 = " + p1.Pos()+`
`"\np2 = " + p2.Pos() +`
`"\np3 = " + p3.Pos());`
	`p1 = (2;2)`
	`p2 = (0;0)`
	`p3 = (1;2)`

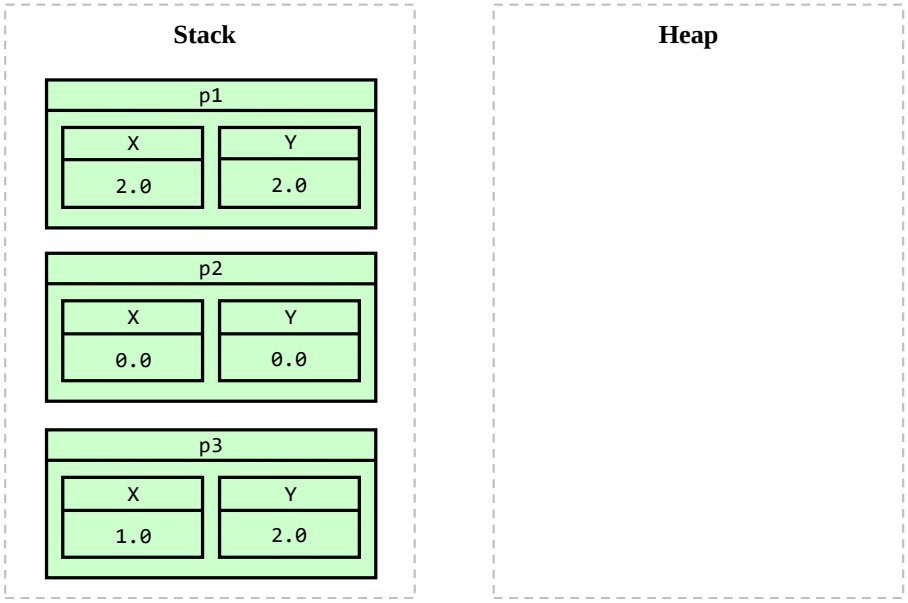
Die Wertsemantik von Strukturen ist im Verhalten des Programms vor allem an zwei Stellen zu beobachten:

- 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`, weil durch die Anweisung
  ```
  var p3 = p1;
  ```
  nicht die Adresse von `p1` in `p3` kopiert wird, sondern eine vollständige Strukturinstanz. Eine spätere Veränderung des Originals hat keinen Effekt auf die Kopie.

Nach der Anweisung

```
p1.Bewegen(1, 0);
```

haben wir die 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 definiert wird (vgl. Abschnitt 5.4.3), hat die Klasse `Punkt` auch einen parameterlosen expliziten Konstruktor erhalten.

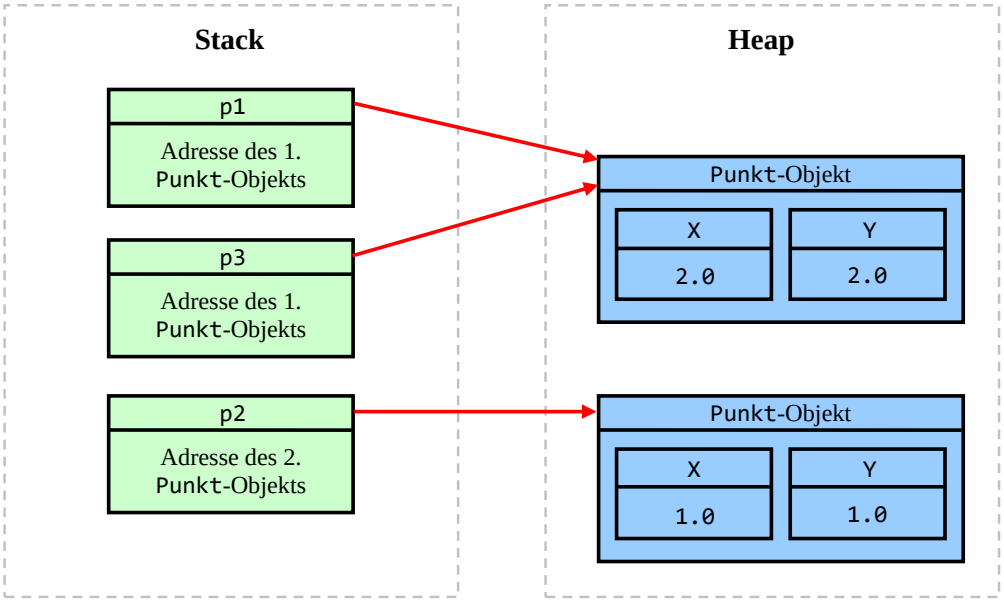
Das obige Programm kann unverändert statt mit der `Punkt`-Struktur auch mit der `Punkt`-Klasse arbeiten und zeigt dabei ein anderes Verhalten:

Quellcode (mit `Punkt` als Klasse)	Ausgabe

`var p1 = new Punkt(1, 2);`
`var p2 = new Punkt();`
`var p3 = p1;`
`Punkt.Bewegen(p2, 1, 1);`
`p1.Bewegen(1, 0);`
`Console.WriteLine("p1 = " + p1.Pos()+`
`"\np2 = " + p2.Pos() +`
`"\np3 = " + p3.Pos());`
	`p1 = (2;2)`
	`p2 = (1;1)`
	`p3 = (2;2)`

Die in der statischen Überladung der Methode `Bewegen()` vorgenommene Ortsveränderung wirkt sich auf das *Objekt* mit der im ersten Aktualparameter übergebenen Adresse aus.

`p1`, `p2` und `p3` 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, solange es im Programm per Referenz erreichbar ist (im Sinn von Abschnitt 5.4.5.1), kann eine Strukturinstanz nur als Objekt-Member (z. B. als Array-Element) 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 Beenden der Methode gelöscht. Bei entsprechender Methodendefinition kann dem Aufrufer via Rückgabewert 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 besitzt die Klasse `PunktDemo` eine (automatisch implementierte und initialisierte) Eigenschaft namens `Position` vom Typ `Punkt`:

```
using System;
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, dann liefert der Lesezugriff auf die Eigenschaft `Position` des `PunktDemo`-Objekts die *Adresse* des `Punkt`-Instanzobjekts. Dieses Objekt wird durch die Methode `Bewegen()` erfolgreich zu einer Ortsveränderung veranlasst, und es resultiert die Ausgabe:

```
Neue Position: (1;2)
```

Ist `Punkt` hingegen eine Struktur, wird keine Positionsveränderung gemeldet:

```
Neue Position: (0;0)
```

Die **get**-Methode der Eigenschaft `Position` liefert eine *Kopie* der `Punkt`-Struktur, und diese Kopie wird anschließend per `Bewegen()` - Aufruf verschoben. Das Original (die hinter der Eigenschaft stehende Strukturinstanz, vgl. Abschnitt 5.5.2) wird dabei *nicht* verändert. Die „Scheinbewegung“ tritt diesmal nicht mit der statischen `Bewegen()` – Methode der Struktur `Punkt` auf, sondern mit der gleichnamigen Instanzmethode. Um solche Missverständnisse mit beliebig gravierenden Konsequenzen auszuschließen, sollten Strukturen als unveränderlich konzipiert werden.

Ein Visual Studio – Projekt mit dem Beispielprogramm zur Demonstration der Wertsemantik von Strukturen befindet sich hier:

…\BspUeb\Weitere .NETte Typen\Strukturen\Punkt

6.1.2 Kreation und Initialisierung von Strukturinstanzen

Während eine Variable vom Typ einer Klasse als Wert eine Speicheradresse aufnimmt und in einem 64-Bit – Betriebssystem unabhängig von der Ausstattung der Klasse immer 8 Byte (= 64 Bit) belegt, nimmt eine Variable vom Typ einer Struktur die gesamte Instanz auf, sodass ihre Speichergröße von der Ausstattung der Struktur abhängt. Um den von einem benutzerdefinierten Datentyp belegten Speicherplatz in Bytes per **sizeof**-Operator zu ermitteln, muss unsicherer Code in einem **unsafe**-Block ausgeführt werden, z. B.:[1]

---

[1] Den **unsafe**-Block muss man zudem in der Projektdatei durch ein **AllowUnsafeBlocks**-Element ankündigen:

```
<PropertyGroup>
    <OutputType>Exe</OutputType>
    . . .
   <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

Quellcode	Ausgabe

`unsafe {`
`    Console.WriteLine(sizeof(Struktur));`
`    Console.WriteLine(sizeof(Klasse));`
`}`
 	
`public struct Struktur {`
`    public long A, B, C, D;`
`}`
 	
`public class Klasse {`
`    public long A, B, C, D;`
`}`
	`40`
	`8`

Durch die Deklaration einer Variablen mit Strukturtyp (z. B. in einer Methode) wird bereits der gesamte Speicherplatz der Strukturinstanz belegt. Solange nicht alle Felder initialisiert sind, kann man aber mit einer Strukturinstanz nichts anfangen, z. B. keine Methoden aufrufen.

Dass man einer Variablen mit Strukturtyp das Referenzliteral **null** *nicht* zuweisen darf, versteht sich von selbst.

Sofern Zugriffsrechte bestehen, ist eine Initialisierung der Struktur-Instanzvariablen über Wertzuweisungen möglich, z. B.:

Quellcode	Ausgabe

`Punkt punkt;`
`punkt.X = 1;`
`punkt.Y = 2;`
`punkt.Position();`
 	
`public struct Punkt {`
`    public int X;`
`    public int Y;`
 	
`    public void Position() {`
`        Console.WriteLine($"({X}; {Y})");`
`    }`
`}`
	`(1; 2)`

Der Compiler besteht darauf, dass vor dem Aufruf einer Instanzmethode alle Felder initialisiert worden sind. Im Beispiel können die Felder *nicht* durch automatisch implementierte Eigenschaften ersetzt werden, weil dann zum Initialisieren ein **set**-Block ausgeführt werden müsste, was aber eine abgeschlossene Initialisierung voraussetzt, z. B.:

```
Punkt punkt;
punkt.X = 1; // verboten

public struct Punkt {
    public int X { get; set; }
    public int Y { get; set; }

    public void Position() {
        Console.WriteLine($"({X}; {Y})");
    }
}
```

In der Regel verwendet man bei den Strukturinstanzen (wie bei den Objekten von Klassen) einen Konstruktur zum Initialisieren. Eine Struktur besitzt einen parameterlosen Standardkonstruktor, der alle Instanzvariablen auf den typspezifischen Nullwert setzt, z. B.:

Quellcode	Ausgabe

`var punkt = new Punkt();`
`punkt.Position();`
 	
`public struct Punkt {`
`  public int X { get; set; }`
`  public int Y { get; set; }`
 	
`  public void Position() {`
`  Console.WriteLine($"({X}; {Y})");`
`  }`
`}`
	`(0; 0)`

In einer Strukturdefinition sind beliebig viele explizite Konstruktorüberladungen erlaubt, wobei der Standardkonstruktor *nicht* verlorengeht, z. B.:

Quellcode	Ausgabe

`var p1 = new Punkt(1, 2);`
`p1.Position();`
`var p2 = new Punkt();`
`p2.Position();`
 	
`public struct Punkt {`
`  public int X { get; set; }`
`  public int Y { get; set; }`
 	
`  public Punkt(int x, int y) {`
`  X = x;`
`  Y = y;`
`  }`
 	
`  public void Position() {`
`  Console.WriteLine($"({X}; {Y})");`
`  }`
`}`
	`(1; 2)`
	`(0; 0)`

Bis C# 10 musste ein expliziter Strukturkonstruktor *alle* Instanzvariablen initialisieren. Seit C# 11 ist das nicht mehr erforderlich, wobei die nicht-initialisierten Felder automatisch den typspezifischen Nullwert erhalten, z. B.:

Quellcode	Ausgabe

`var punkt = new Punkt(1);`
`punkt.Position();`
 	
`public struct Punkt {`
`  public int X { get; set; }`
`  public int Y { get; set; }`
 	
`  public Punkt(int x) {`
`  X = x;`
`  }`
 	
`  public void Position() {`
`  Console.WriteLine($"({X}; {Y})");`
`  }`
`}`
	`(1; 0)`

Seit C# 10 ist die Initialisierung von Struktur-Instanzvariablen im Rahmen der Deklaration erlaubt, wobei man (bei Strukturen und auch bei Klassen) von einem *Feldinitialisierer* spricht. Allerdings benötigt eine Struktur mit Feldinitialisierer einen *expliziten* Konstruktor, damit die Initialisierungen realisiert werden. Der Compiler fügt den einem Feldinitialisierer entsprechenden IL-Code in jeden expliziten Konstruktor ein, aber nicht in den Standardkonstruktor, z. B:[1]

Quellcode	Ausgabe

`var punkt = new Punkt();`
`punkt.Position();`
 	
`public struct Punkt {`
`  public int X = 1;`
`  public int Y = 2;`
 	
`  public Punkt() {`
`  }`
 	
`  public void Position() {`
`  Console.WriteLine($"({X}; {Y})");`
`  }`
`}`
	`(1; 2)`

Erst seit C# 10 ist in einer Struktur ein expliziter *parameterfreier* Konstruktor erlaubt, wobei der Standardkonstruktor auch durch die Definition eines expliziten parameterfreien Konstruktors *nicht* verlorengeht.

Weil der im Abschnitt 8.6 zu beschreibende **default**-Operator bei Werttypen dasselbe Ergebnis liefert wie der Standardkonstruktor (ECMA 2022, Abschnitt 8.3.3), lässt sich das Produkt des Standardkonstruktors weiter anfordern. Wird der Typ einer zu initialisierenden Variablen explizit angegeben, dann darf der **default**-Operator sogar ohne Argument verwendet werden, z. B.:

[1] Bei einer Klasse geht der aus einem Feldinitialisierer resultierende IL-Code auch in den Standardkonstruktor ein.

Quellcode	Ausgabe

`var p1 = default(Punkt);`
`p1.Position();`
`Punkt p2 = default;`
`P2.Position();`
`var p3 = new Punkt();`
`p3.Position();`
 	
`public struct Punkt {`
`  public int X { get; set; }`
`  public int Y { get; set; }`
 	
`  public Punkt() {`
`  X = 1;`
`  Y = 2;`
`  }`
 	
`  public void Position() {`
`  Console.WriteLine($"({X}; {Y})");`
`  }`
`}`
	`(0; 0)`
	`(0; 0)`
	`(1; 2)`

In der englischen Literatur wird der Standardkonstruktor von Werttypen als *default-Konstruktor* bezeichnet.

Der Standard- bzw. default-Konstruktor einer Struktur übersteht nicht nur die Definition eines expliziten parameterfreien Konstruktors, sondern dominiert diesen auch, was nicht ignoriert werden darf. Befindet sich in einer Klassendefinition eine Instanzvariable mit Strukturtyp, dann kommt bei einer Objektkreation im Rahmen der automatischen Nullwert-Initialisierung der default-Konstruktor zum Einsatz. Das passiert insbesondere bei einem Array (vgl. Abschnitt 6.2) mit Elementen vom Typ einer Struktur, z. B.:

Quellcode	Ausgabe

`var pa = new Punkt[3];`
`pa[0].Position();`
`var punkt = new Punkt();`
`punkt.Position();`
 	
`public struct Punkt {`
`  public int X { get; set; }`
`  public int Y { get; set; }`
 	
`  public Punkt() {`
`  X = 1;`
`  Y = 2;`
`  }`
 	
`  public void Position() {`
`  Console.WriteLine($"({X}; {Y})");`
`  }`
`}`
	`(0; 0)`
	`(1; 2)`

In einer Strukturdefinition sollte man daher möglichst auf einen expliziten parameterfreien Konstruktor verzichten.

6.1.3 Detailvergleich von Klassen und Strukturen

Nachdem wir uns mit den Implikationen der Wertsemantik von Strukturen und der (ziemlich variantenreichen) Strukturinitialisierung beschäftigt haben, betrachten wir in diesem Abschnitt die Strukturdefinition im Überblick, wobei die Klassendefinition als Referenz dient.[1]

Die Struktur- und die Klassendefinition haben viele Gemeinsamkeiten, doch gibt es auch Unterschiede:

- Das Schlüsselwort **class** wird durch das Schlüsselwort **struct** ersetzt.
- Bis auf den Finalizer (siehe Abschnitt 5.4.5) sind in einer Strukturdefinition dieselben Member erlaubt wie in einer Klassendefinition.
- In einer Struktur sind auch statische Member (z. B. Felder und Methoden) erlaubt, aber selten erforderlich.
- Wie bei Klassen wird die Schutzstufe eines Strukturtyps über Modifikatoren geregelt (Voreinstellung: **internal**, Alternativen **public** und **file**, vgl. Abschnitt 5.12).
- Seit C# 7.2 kann man in einer Strukturdefinition dem Compiler durch den Modifikator **readonly** mitteilen, dass die generell empfohlene Unveränderlichkeit von Strukturinstanzen durchgesetzt werden soll, z. B.:

```
readonly public struct Strecke {
    public int Li { get; }
    public int Re { get; }
    public Strecke(int li, int re) {
        Li = li; Re = re;
    }
}
```

  Dann besteht der Compiler darauf, dass ...
  - alle Felder den Modifikator **readonly** erhalten
  - und alle Eigenschaften schreibgeschützt sind:
    - Vor C# 9 war nur ein **get**-Block zulässig.
    - Seit C# 9 ist zusätzlich ein **init**-Block erlaubt (siehe Abschnitt 5.5.3).

  Instanzen einer **readonly**-Struktur sollten aus Performanzgründen als Aktualparameter mit dem **in**-Schlüsselwort gekennzeichnet werden (siehe Abschnitt 5.3.1.3.2.3), wenn sie größer sind als eine Speicheradresse.[2]

  Mit dem Typmodifikator **readonly** besitzen die Strukturen eine Option, die den Klassen fehlt.
- Wie bei Klassen können auch bei Strukturen die Felder den Modifikator **readonly** erhalten, sodass nach der Initialisierung keine Wertveränderung mehr möglich ist.
- Seit C# 8 kann bei Strukturen auch eine instanzbezogene Methode oder Eigenschaft den Modifikator **readonly** erhalten. Daraufhin stellt der Compiler sicher, dass diese Methode bzw. Eigenschaft den Zustand der *agierenden Instanz* nicht ändert. Eine *andere* Instanz derselben Struktur (z. B. per **ref**-Parameter übergeben) ist aber *nicht* vor Veränderungen geschützt, z. B.:

```
struct Struktur {
    public int Temp;
    public readonly void RoMo(ref Struktur p) {
        Temp = 13; // verboten
        p.Temp = 13; // zulässig
    }
}
```

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct

[2] Die Größe einer Speicheradresse ist aus der statischen Eigenschaft **IntPtr.Size** abzulesen.

- Den Strukturen fehlt die Vererbungstechnik, die wir im Kapitel 7 gründlich behandeln werden. Jede Struktur stammt direkt von der Klasse **ValueType** im Namensraum **System** ab, die wiederum direkt von der Urahnklasse **Object** erbt (siehe Abbildung im Abschnitt 6.1.5). Die Verwendung einer Struktur als Ableitungsbasis ist ausgeschlossen. Somit können keine Strukturhierarchien entstehen, und mit der Vererbung im Zusammenhang stehende Modifikatoren (z. B. **protected**) sind verboten (siehe ECMA 2022, Abschnitt 15.4.3, S. 428).
- Strukturen erben die Methode **Equals()**

  public bool Equals(Object *obj***)**

  von der Basisklasse **ValueType**. Die **Equals()** - Methode der Klasse **ValueType** überschreibt die **Equals()** - Methode der Urahnklasse **Object** und nimmt im Unterschied zur überschriebenen Methode keinen Adress- sondern einen Inhaltsvergleich vor (zum Überschreiben von Methoden siehe Abschnitt 7.9). Im folgenden Beispiel

  ```
  Demo p1 = new() { A = 1}, p2 = new() { A = 1};
  Console.WriteLine(p1.Equals(p2));
  struct Demo { public int A; };
  ```

  resultiert die Ausgabe **true**. Wird `Demo` als Klasse definiert, dann resultiert die Ausgabe **false**, weil die **Equals()** - Methode bei Objekten auf den Speicheradressen basiert.

  Besitzt beim Aufruf der **ValueType**-Methode **Equals()** der Parameter denselben Typ wie die angesprochene Instanz, dann wird für jedes Feld der Struktur die typspezifische **Equals()** - Methode mit einem angepassten Parameter aufgerufen, und das Gesamtergebnis hat den Wert **true**, wenn alle Einzelvergleiche so enden. Die inhaltsorientierte **ValueType**-Implementation von **Equals()** ist respektabel, hat aber zwei gravierende Probleme:[1]

  - Die **ValueType**-Implementation muss in der Regel die zu vergleichenden Felder per Reflexion ermitteln (siehe Kapitel 14), was eine schlechte Performanz zur Folge hat.
  - Wegen des Parametertyps **object** kommt es zum Boxing:

    ```
    public override bool Equals(object obj) {
     . . .
    }
    ```

  Durch die Definition einer **Equals()** – *Überladung* mit der definierten Struktur als Parametertyp werden die beiden Probleme gelöst. Diese **Equals()** – Überladung wird beim Aufruf mit passendem Aktualparameter bevorzugt, sodass kein Boxing stattfindet.[2] Das etwas lästige Codieren der Einzelvergleiche spart Rechenzeit im Vergleich zur Lösung per Reflexion und bietet eine gelegentlich vorteilhafte Flexibilität. Nach Albahari (2022, S. 331) bewirken die Ersetzung der auf Reflexion beruhenden **ValueType**-Implementation und die Vermeidung der Boxing-Operationen bei der **Equals()** – Ausführung jeweils eine Beschleunigung um den Faktor fünf.

  Für den Fall, dass beim **Equals()** – Aufruf die zu vergleichende Strukturinstanz in verpacktem Zustand angeliefert wird, definiert man zusätzlich eine **Equals()** – *Überschreibung*, die ggf. die **Equals()** – Überladung aufruft (siehe Abschnitt 6.1.6). Die ähnlich klingenden Begriffe *Überladung* und *Überschreibung* haben eine erheblich divergierende Bedeutung, und im Fall der **Equals()** – Methode in einer selbst definierten Struktur ist eine Überschreibung *und* eine Überladung zu empfehlen. Während wir die Überladung von Methoden aus dem Abschnitt 5.3.5 kennen, steht die Behandlung der Überschreibung noch aus (siehe Abschnitt 7.9).

  Im Beispiel wird die Regel befolgt, zusammen mit der **Equals()** – Methode stets auch die (bei Strukturen ebenfalls von **ValueType** geerbte) Methode **GetHashCode()** zu

---

[1] https://devblogs.microsoft.com/premier-developer/box-or-not-to-box-that-is-the-question/

[2] Aufgrund der Existenz der beschriebenen **Equals()** – Überladung kann ein Typ von sich behaupten, die Schnittstelle **IEquatable<T>** zu erfüllen (siehe Kapitel 9).

überschreiben (siehe z. B. Bloch 2018, S. 50). Das ist wichtig für den nie auszuschließenden Fall, dass Strukturinstanzen durch Kollektionsklassen wie **HashSet<T>** oder **Dictionary<K, V>** verwaltet werden (siehe Kapitel 11).

Wenn auch die Operatoren == und **!=** für einen Strukturtyp verwendbar sein sollen, dann werden Operatorüberladungen fällig, die aber mit Hilfe einer **Equals()** – Überschreibung leicht zu realisieren sind (vgl. Abschnitt 5.8.3).

- Auch bei Strukturen ist eine Datenkapselung möglich und sinnvoll. 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 Modifikator **protected** verboten.
- Am Ende einer Struktur- oder Klassendefinition darf optional ein Semikolon stehen.

Seit C# 10 lässt sich der ursprünglich für Record-Typen (siehe Abschnitt 6.7) eingeführte **with**-Operator auch bei Strukturen dazu verwenden, aus einer Instanz eine Kopie mit partiell geänderten Werten zu erstellen, z. B.:

Quellcode	Ausgabe

`Punkt p1 = new() { X = 1, Y = 2 };`
`Punkt p2 = p1 with { X = 3 };`
`Console.WriteLine(p2.X);`
 	
`public struct Punkt {`
`  public double X { get; init; }`
`  public double Y { get; init; }`
`}`
	`3`

6.1.4 Zeit- und Speicheraufwand für Objekte und Strukturinstanzen

In diesem Abschnitt untersuchen wir mit einem Testprogramm den Zeit- und Speicheraufwand für Strukturinstanzen bzw. Objekte. Es wird jeweils in einer Methode ein Array (vgl. Abschnitt 6.2) mit 1.000.000 Strukturinstanzen bzw. Objekten unter Verwendung der folgenden Typdefinitionen angelegt.

```
public struct SPunkt {
    public double X { get; init; }
    public double Y { get; init; }

    public SPunkt(double x, double y) {
        X = x; Y = y;
    }
}

public class CPunkt {
    public double X { get; init; }
    public double Y { get; init; }

    public CPunkt(double x, double y) {
        X = x; Y = y;
    }
}
```

Zur Ermittlung des vom Programm belegten Speichers wird die statische Methode **GetTotalMemory()** der Klasse **GC** verwendet:

public static long GetTotalMemory(bool *forceFullCollection***)**

Mit dem Parameter *forceFullCollection* legt man fest, ob die Methode vor der Rückkehr einen Einsatz des Garbage Collectors abwarten soll (Aktualparameterwert **true**). Für die Zeitmessungen wird ein Objekt der Klasse **Stopwatch** verwendet (vgl. Abschnitt 4.7.4).

```
using System;
using System.Diagnostics;

class Aufwandsvergleich {
    const int anzahl = 1_000_000;
    static Stopwatch stopwatch = new();
    static string format = "{0,-55}{1,5}";

    static void MakeStructArray() {
        Console.WriteLine(format, "Speicher in MB vor struct-Array - Erstellung:",
                          GC.GetTotalMemory(true) / 1048576);
        stopwatch.Restart();
        SPunkt[] arc = new SPunkt[anzahl];
        for (int i = 0; i < anzahl; i++)
            arc[i] = new SPunkt(i, i);
        stopwatch.Stop();
        Console.WriteLine(format, "Speicher in MB nach struct-Array - Erstellung:",
                          GC.GetTotalMemory(false) / 1048576, 20);
        Console.WriteLine(format, "Benötigte Zeit in Millisek. für den struct-Array:",
                          stopwatch.ElapsedMilliseconds);
    }

    static void MakeClassArray() {
        Console.WriteLine("\n"+ format, "Speicher in MB vor class-Array - Erstellung:",
                          GC.GetTotalMemory(true) / 1048576);
        stopwatch.Restart();
        CPunkt[] arc = new CPunkt[anzahl];
        for (int i = 0; i < anzahl; i++)
            arc[i] = new CPunkt(i, i);
        stopwatch.Stop();
        Console.WriteLine(format, "Speicher in MB nach class-Array - Erstellung:",
                          GC.GetTotalMemory(false) / 1048576);
        Console.WriteLine(format, "Benötigte Zeit in Millisek. für den class-Array:",
                          stopwatch.ElapsedMilliseconds);
    }

    static void Main() {
        MakeStructArray();
        MakeClassArray();
    }
}
```

Die folgenden Messergebnisse wurden mit der Release-Konfiguration ermittelt (vgl. Abschnitt 3.3.8.3):

```
Speicher in MB vor struct-Array - Erstellung:              0
Speicher in MB nach struct-Array - Erstellung:            15
Benötigte Zeit in Millisek. für den struct-Array:         33

Speicher in MB vor class-Array - Erstellung:               0
Speicher in MB nach class-Array - Erstellung:             38
Benötigte Zeit in Millisek. für den class-Array:         119
```

Der höhere Speicherbedarf für die Objekte überrascht nicht, 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. Das Array enthält nur die Adressen der 1.000.000 separaten `Punkt`-Objekte.

Der Zeitaufwand für das Erstellen ist bei den Objekten ca. 4 Mal höher als bei den Strukturinstanzen, und der (im Programm nicht gemessene) Aufwand für das Abräumen durch den Garbage

Collector dürfte bei einem Array mit Objekten ebenfalls erheblich höher ausfallen. Während alle Strukturinstanzen zusammen mit dem Array untergehen, müssen die Objekte einzeln gelöscht werden.

Für die seit dem Kapitel 1 wiederholt in Beispielen verwendete Klasse `Bruch` kommt die Ersetzung durch eine Struktur aus mehreren Gründen *nicht* in Frage:

- Weil bei Strukturen der Standardkonstruktor nicht deaktiviert werden kann, wären `Bruch`-Instanzen mit dem Nenner 0 möglich. Bei einer Struktur muss sichergestellt sein, dass die Nullinitialisierung aller Felder zu einer regulären Instanz führt. Das wäre bei `Bruch`-Strukturinstanzen nicht gewährleistet.
- Es ist nicht damit zu rechnen, dass `Bruch`-Objekte derart massenhaft auftreten, dass ein Spareffekt durch die Verwendung von Strukturen spürbar wird.
- Man würde die Möglichkeit verlieren, per Vererbung aus dem Typ `Bruch` spezialisierte Varianten abzuleiten.

6.1.5 Strukturen im allgemeinen .NET - Typsystem

Aus den bisherigen Ausführungen zu folgern, dass Strukturen nur für leistungskritische Anwendungen interessant seien, wäre schon deshalb grundverkehrt, weil es sich bei allen elementaren Datentypen um Strukturen handelt. Die als Typbezeichner eingeführten C# - Schlüsselwörter sind lediglich Aliasnamen für Strukturen aus dem BCL-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.

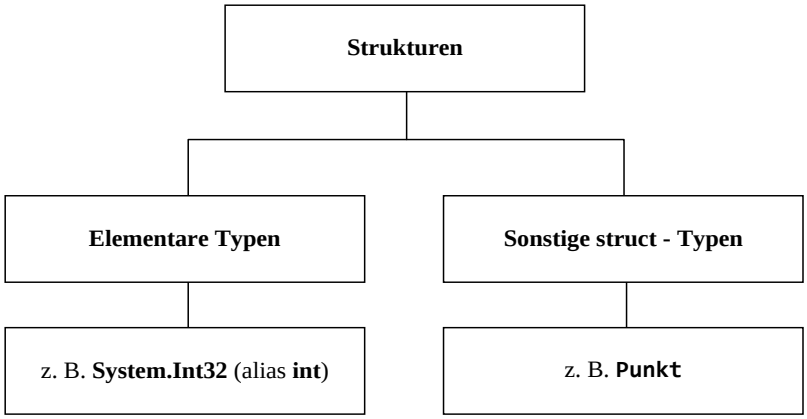
Für die elementaren Datentypen bietet C# im Vergleich zu den sonstigen Strukturtypen neben den reservierten Wörtern als Typaliasnamen noch weitere Vorzugsbehandlungen, z. B. die Erzeugung von Werten über Literale (vgl. ECMA 2022, S. 74f). Wie z. B. die Definition des BCL-Typs **Int32** zeigt,[1]

```
public readonly struct Int32 : IComparable, IConvertible { ... }
```

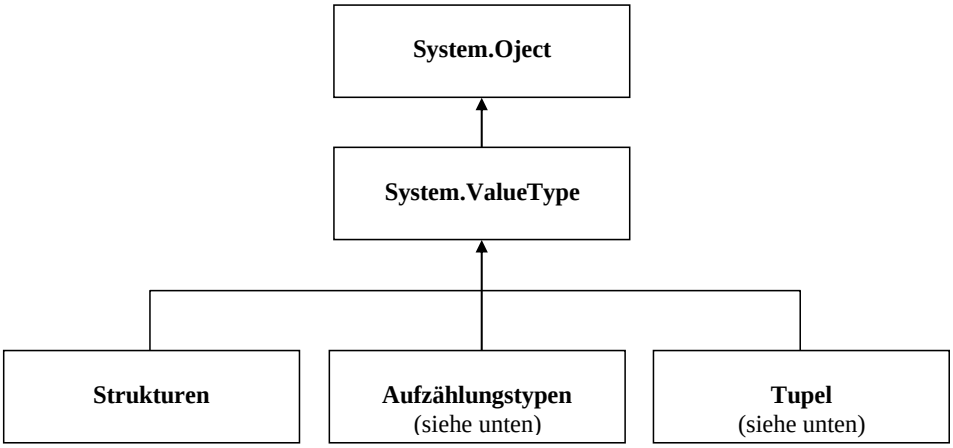
sind die elementaren Datentypen ansonsten reguläre Strukturen.

Bei den Strukturen im CTS (Common Type System) können wir zwischen den elementaren Datentypen und sonstigen Strukturtypen differenzieren:

[1] Hinter dem Doppelpunkt im Definitionskopf sind *Schnittstellen* aufgelistet (siehe Kapitel 9).



Jeder Strukturtyp stammt von der Klasse (!) **ValueType** im Namensraum **System** ab, die wiederum direkt von der .NET - Urahnklasse **Object** erbt:

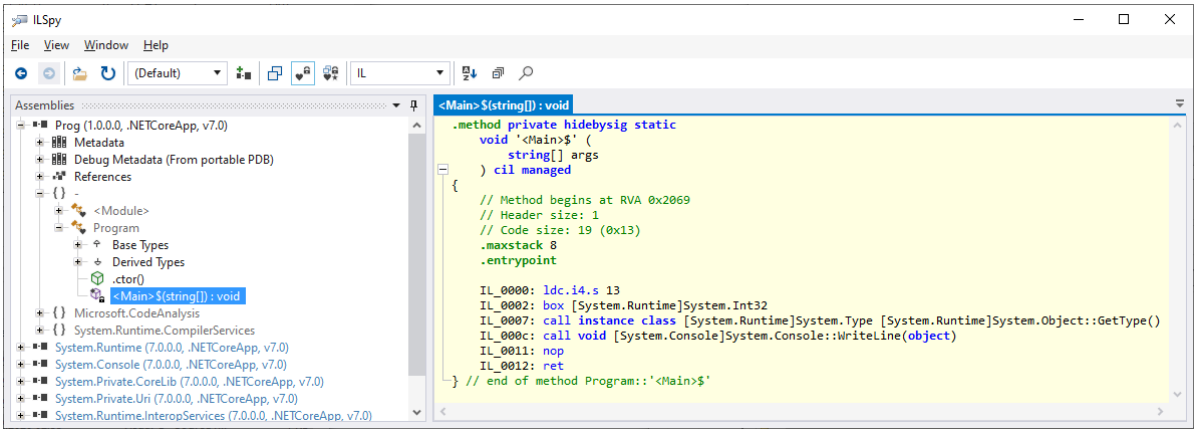


Im folgenden Beispiel wird beim Ganzzahlliteral 13 (vom Strukturtyp **Int32**) über die von **System.Object** geerbte Methode **GetType()** erfolgreich der Datentyp erfragt:

Quellcode	Ausgabe
`Console.WriteLine(13.GetType());`	`System.Int32`

Trotz des obigen Stammbaums *ist* eine Strukturinstanz kein Objekt, aber aufgrund einer im nächsten Abschnitt zu beschreibenden Raffinesse kann eine Strukturinstanz jederzeit so behandelt werden, als *wäre* sie ein Objekt.

Im Beispiel ist zur Ausführung der von **Object** geerbten Methode **GetType()** ein „echtes“ Objekt erforderlich, und das wird über die im nächsten Abschnitt behandelte Boxing-Operation hergestellt, wie ein Blick auf den IL-Code mit Hilfe des Analyseprogramms ILSpy zeigt:



In der Methode **Main()** (genauer: **<Main>$()**, vgl. Abschnitt 4.1.2) befindet sich die IL-Anweisung (der *operation code*, kurz *OpCode*) **box**.

6.1.6 Boxing

Wie bald im Kapitel 7 über die Vererbung 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 aufnehmen. Was soll nun aber geschehen, wenn einer solchen Referenzvariablen z. B. ein Wert vom elementaren Typ **int** zugewiesen wird?

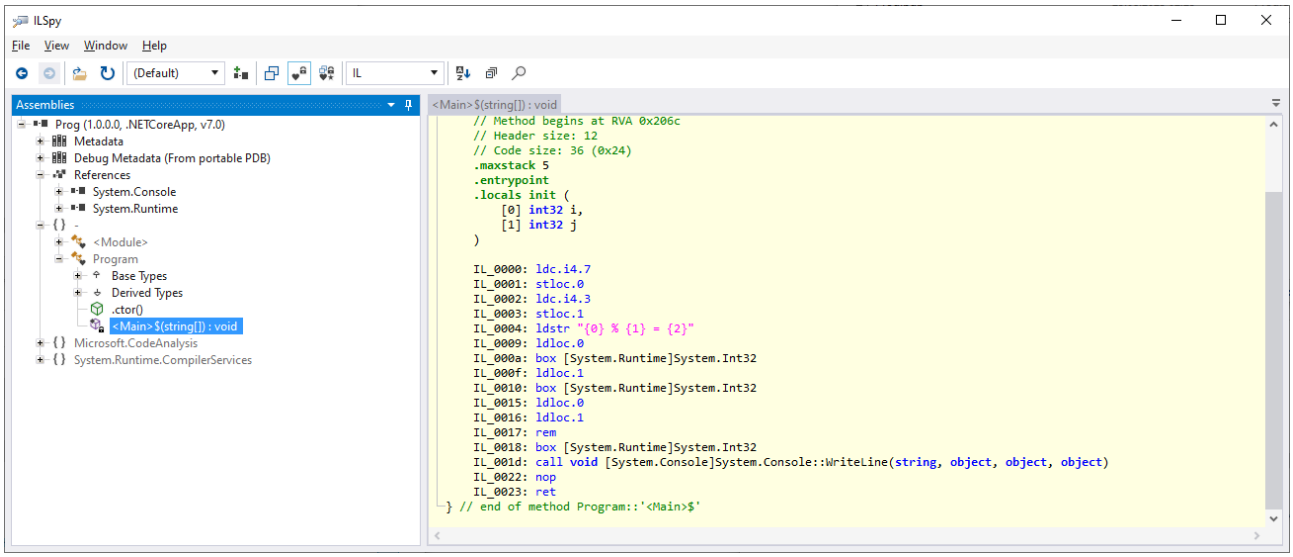
Eine analoge Situation liegt vor, wenn bei einem Methodenaufruf eine Strukturinstanz als Aktualparameter auftritt, obwohl gemäß Methodendefinition eine Objektreferenz (die Adresse eines Heap-Objekts) benötigt wird. Wir haben uns längst daran gewöhnt, dass der Compiler Werte von beliebigem Typ als **Object**-Instanzen akzeptiert. Z. B. werden im folgenden Beispiel

```
int i = 7, j = 3;
Console.WriteLine("{0} % {1} = {2}", i, j, i % j);
```

der Methode **WriteLine()** Aktualparameter vom Werttyp **int** an Positionen mit dem Formalparametertyp **Object** übergeben:

public static void WriteLine(String *format***, params Object[]** *arg***)**

Diese Zuweisungskompatibilität wird möglich durch eine als **Boxing** bezeichnete Technik: Sie sorgt dafür, dass ein Wert bei Bedarf automatisch in ein Objekt einer passenden Hilfs- bzw. Hüllenklasse verpackt wird. Somit existiert ein Heap-Objekt und der **Console**-Methode **WriteLine()** kann die benötigte Adresse übergeben werden. Ein Blick mit dem Assembly-Analyseprogramm ILSpy auf den IL-Code des Beispiels zeigt drei Boxing-Operationen:



Die Objektkreation zur Verpackung eines **System.Int32** - Werts wird mit dem *OpCode* (*operation code*) **box** veranlasst.

Im folgenden Beispiel 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]

```
int i = 4711;
object iBox = i;
int j = (int) iBox;
```

Wie die letzte Anweisung des Beispiels zeigt, benötigt der Compiler beim **Unboxing**, also beim Auspacken eines Wertes aus einem Hüllenobjekt, eine explizite Typumwandlung mit Angabe des Zieltyps:

```
int j = (int)iBox;
```

Ein oft unerwartetes Boxing findet statt, wenn …

- ein Strukturtyp eine Schnittstelle (alias: *ein Interface*) implementiert (siehe Kapitel 9),
- und Strukturinstanzen über Referenzvariablen vom Typ der implementierten Schnittstelle angesprochen werden.

Im folgenden Beispiel wird eine Instanz der Struktur **Int32** einer Referenzvariablen vom Typ **IComparable** zugewiesen. Das ist möglich, weil der Typ **Int32** diese Schnittstelle implementiert, und wird per Boxing realisiert.

```
int i = 4711;
IComparable ic = i;
```

Weil das Boxing durch die damit verbundene Objektkreation aufwändig ist, sollte man die Verwendung dieser Technik auf das notwendige Maß beschränken. Im Abschnitt 6.1.3 war am Rand erwähnt worden, dass die Vermeidung von Boxing-Operationen für die Definition einer **Equals()** – Überladung in einer Struktur spricht. Im folgenden Beispiel besitzt die Struktur `Punkt` noch *keine* solche Überladung:

```
class Prog {
    static Punkt p1 = new() { X = 1, Y = 2 };
    static Punkt p2 = new() { X = 1, Y = 2 };
    static void Main() {
        Console.WriteLine(p1.Equals(p2));
    }
}

public struct Punkt {
    public double X { get; init; }
    public double Y { get; init; }
}
```

Aufgrund der sinnvollen Überschreibung der **Object**-Methode **Equals()** in der `Punkt`-Basisklasse **ValueType** werden `Punkt`-Instanzen inhaltsorientiert verglichen, und im Beispiel resultiert die Ausgabe:

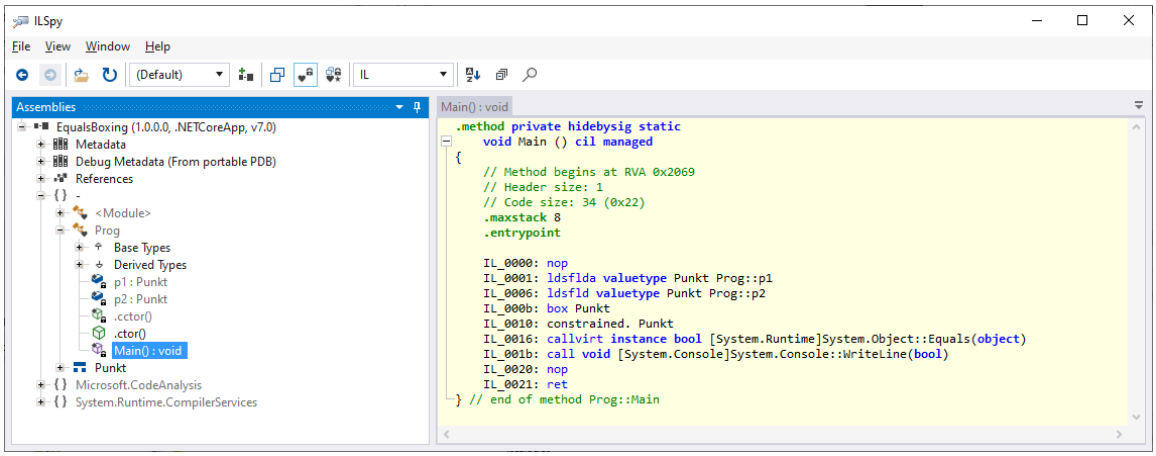
```
True
```

Eine Assembly-Inspektion mit ILSpy ergibt, dass der **Equals()** – Aufruf ein Boxing verursacht:

---

[1] Neben dem *impliziten* Boxing, das auch als **Autoboxing** bezeichnet wird, ist auch ein *explizites* Boxing per Typumwandlung möglich, aber nie erforderlich, z. B.

```
int i = 4711;
object iBox = (object) i;
```



Diesmal werden sogar *zwei* Strukturinstanzen verpackt:

- Das Argument `p2` im **Equals()** – Aufruf wird über den OpCode **box** verpackt, weil in der ausgeführten **Object**-Methode der Parameter vom Typ **Object** ist.
- Weil zur Ausführung der Methode ein Objekt erforderlich ist, wird auch die Strukturinstanz `p1` verpackt. Das passiert im IL-Code über die OpCodes **constrained** und **callvirt**.[1]

Während das Boxing in den bisherigen Beispielen durch die Verwendung einer Variablen oder eines Parameters vom Typ **Object** (oder von einem Interface-Typ) verursacht wurde, ist der Grund für die zweite Verpackung im letzten Beispiel weniger offensichtlich:

- **Equals()** ist in der Klasse **Object** als virtuelle Methode definiert (vgl. Kapitel 7), sodass beim Aufruf der Typ der handelnden Instanz ermittelt und die Methodentabelle des Typs konsultiert werden muss.
- Das setzt ein Objekt voraus, sodass ein Boxing stattfindet.

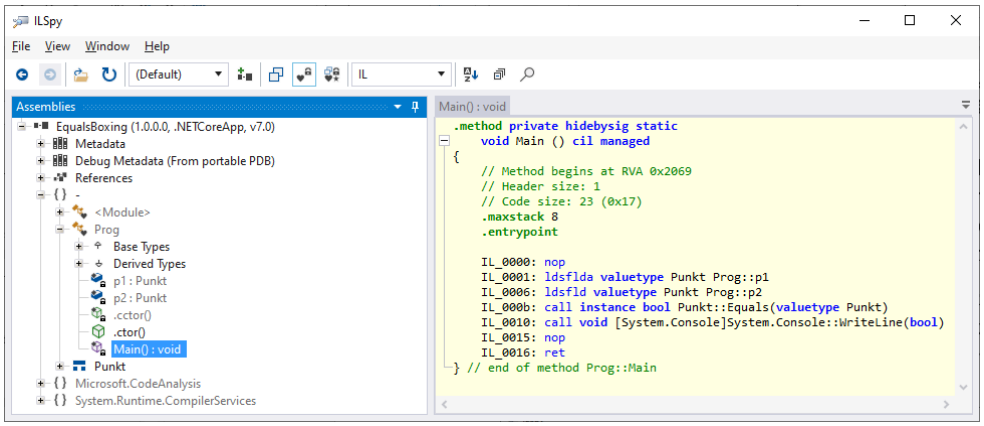
Es gibt noch weitere Gründe dafür, dass zur Ausführung einer Methode ein Objekt erforderlich ist. Wie im Abschnitt 6.1.5 zu sehen war, verursacht auch die von **Object** geerbte, *nicht virtuelle* Methode **GetType()** beim Aufruf ein Boxing (zur Erklärung siehe Richter 2006, S. 163).

Um beim Vergleich von zwei `Punkt`-Strukturinstanzen *ohne* Boxing auszukommen, muss man lediglich eine **Equals()** – *Überladung* in die Definition der Struktur aufnehmen:

```
public struct Punkt {
    public double X { get; init; }
    public double Y { get; init; }

    public bool Equals(Punkt p) => X == p.X && Y == p.Y;
}
```

Erhält eine angesprochene `Punkt`-Instanz einen `Punkt`-Aktualparameter, dann hat der **Equals()** – Aufruf *kein* Boxing zur Folge:

[1] https://learn.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.constrained



In der Struktur `Punkt` ist aufgrund der obigen Definition auch die von **ValueType** geerbte **Equals()** – Methode mit **Object**-Parameter verfügbar. Deren durch Reflexion verursachte schlechte Performanz macht sich aufgrund der Existenz einer **Equals()** – Überladung nur noch selten bemerkbar, weil eine `Punkt`-Instanz in der Regel mit einer anderen `Punkt`-Instanz verglichen wird. Trotzdem ist es sinnvoll, in der Struktur `Punkt` die **Equals()** – Methode mit **Object**-Parameter zu überschreiben, z. B. per Lambda-Symbol und Ausdruck (siehe Abschnitt 5.8.1):[1]

```
public struct Punkt {
    public double X { get; init; }
    public double Y { get; init; }

    public override bool Equals(object obj) => obj is Punkt oap && Equals(oap);
    public bool Equals(Punkt p) => X == p.X && Y == p.Y;

    public override int GetHashCode() => 31 * X.GetHashCode() + Y.GetHashCode();
}
```

Wenn die (mit **override** dekorierte) überschreibende **Equals()** - Methode per Parameter eine verpackte `Punkt`-Instanz erhält, dann ruft sie die **Equals()** – Überladung mit `Punkt`-Parameter auf. Der **is**-Operator (siehe Abschnitt 7.7) erledigt die Typprüfung und liefert bei positivem Ausgang eine lokale Variable mit dem passenden Typ.

Im Beispiel wird die Regel befolgt, zusammen mit der **Equals()** – Methode stets auch die Methode **GetHashCode()** zu überschreiben. Eine Struktur erbt diese Methode (wie **Equals()**) von der Klasse **ValueType**. Die Idee zur Kombination der beiden von der Struktur **Double** gelieferten **GetHashCode()** - Werte zu einem `Punkt`-Hashcode stammt von Bloch (2018, S. 52).

Der Blog-Beitrag

https://theburningmonk.com/2015/07/beware-of-implicit-boxing-of-value-types/

leitet aus einer Analyse von Boxing-Performanz-Risiken die folgenden Empfehlungen für das Design von Strukturtypen ab:

- make them immutable
- override **Equals()** (the one that takes an object as argument)
- overload **Equals()** to take another instance of the same value type (implement **IEquatable<T>**)
- overload operators == and **!=**
- override **GetHashCode()**

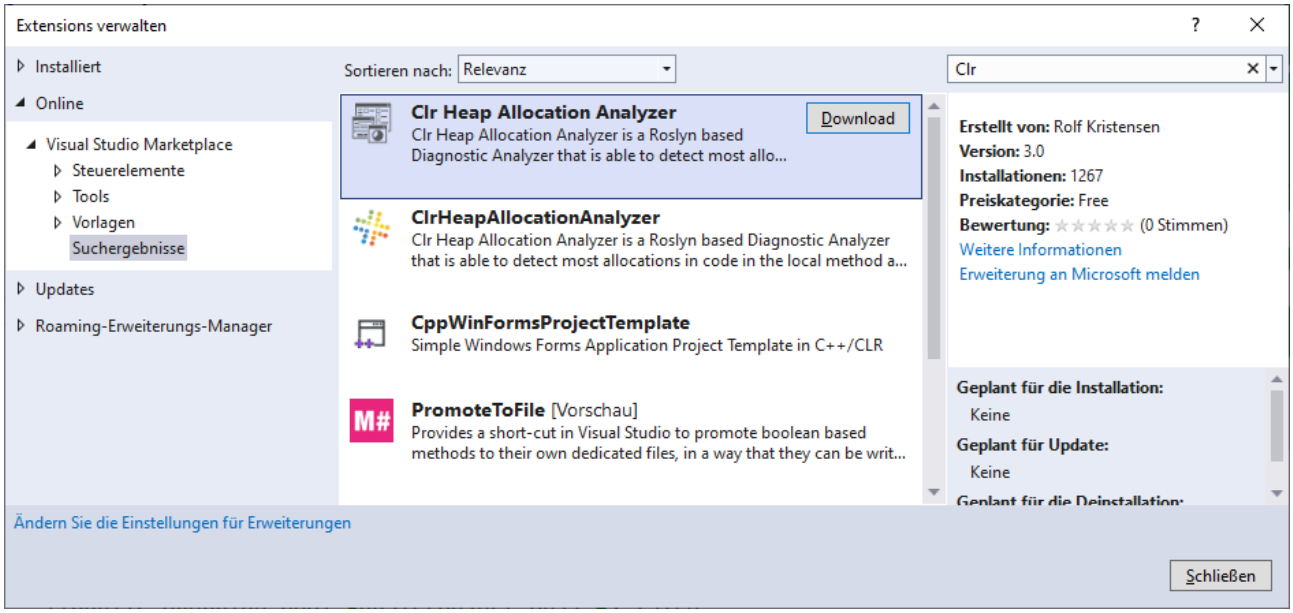
---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/how-to-define-value-equality-for-a-type

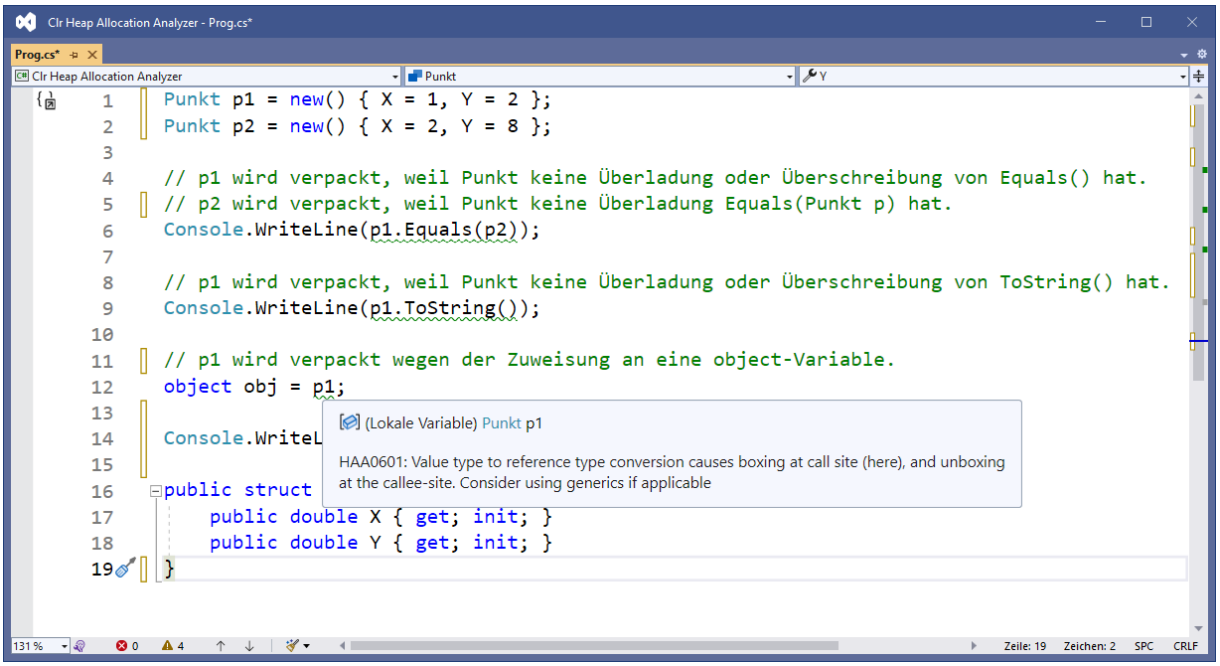
Zur Fahndung nach Performanz-schädlichen Boxing-Operationen eignet sich das kostenlose Visual Studio – Plugin **Clr Heap Allocation Analyzer**, das über den Menübefehl

Erweiterungen > Erweiterungen verwalten

zu installieren ist (siehe Abschnitt 3.4.1):



Im folgenden Quellcode sind drei oben besprochene Boxing-Anlässe enthalten, die vom Plugin erkannt und mit einer kommentierten Warnung versehen werden:

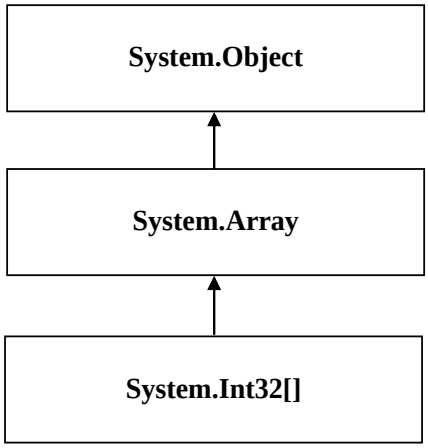


6.2 Arrays

Ein Array ist ein Objekt, das eine feste Anzahl von Elementen desselben Datentyps als Instanzvariablen enthält, die in einem zusammenhängenden Speicherbereich hintereinander abgelegt werden. Man kann den kompletten Array ansprechen (z. B. als Aktualparameter an eine Methode übergeben), oder auf einzelne Elemente über einen durch eckige Klammern begrenzten Index zugreifen.

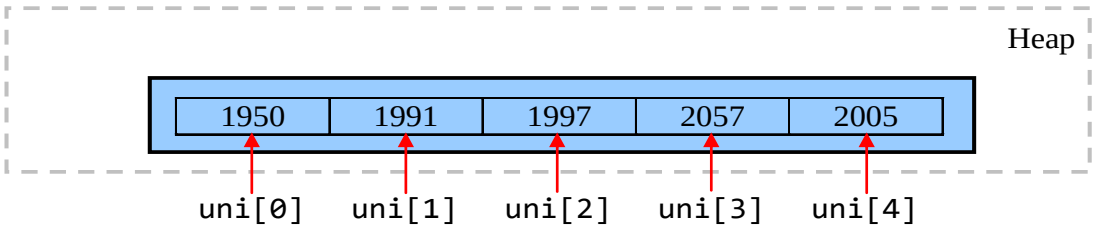
Arrays werden in vielen Programmiersprachen auch *Felder* genannt. In C# bezeichnet man jedoch einheitlich die (Instanz-)variablen einer Klasse oder Struktur als *Felder*, sodass der Name hier nicht mehr zur Verfügung steht.

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**):



Weil alle Arrays von der Klasse **System.Array** abstammen, implementieren sie wichtige Schnittstellen (vgl. Kapitel 9), z. B. **ICollection**, **IEnumerable**, **IList**. Dieser Satz wird sich später als wichtig erweisen, kann und muss aber jetzt noch nicht verstanden werden.

Hier ist als konkretes Objekt aus der Klasse **int[]** ein Array namens `uni` mit fünf 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 Array-Namen den durch eckige Klammern begrenzten Index an, wobei die Nummerierung mit 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 ermöglichen Arrays eine gravierende Vereinfachung der Programmierung:

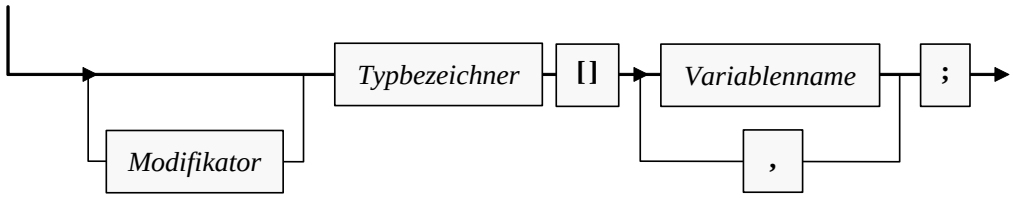
- Weil der Index auch durch einen *Ausdruck* (z. B. durch eine Variable) geliefert werden kann, sind Arrays im Zusammenhang mit den Wiederholungsanweisungen äußerst praktisch.
- Man kann die *gemeinsame* Verarbeitung *aller* Elemente (z. B. bei der Ausgabe in eine Datei) per Methodenaufruf mit Array-Aktualparameter 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.

6.2.1 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 sind, hängt davon, ob die Variable zu einer Methode, zu einer Klasse oder zu einer Instanz gehört. Die Array-Variable `uni` aus dem zu Beginn des Abschnitts 6.2 vorgestellten und im weiteren Verlauf des Abschnitts noch mehrfach betrachteten Beispiel gehört zu einer Methode und wird folgendermaßen deklariert:

```
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** (Zeiger auf nichts) zugewiesen werden.

6.2.2 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`:

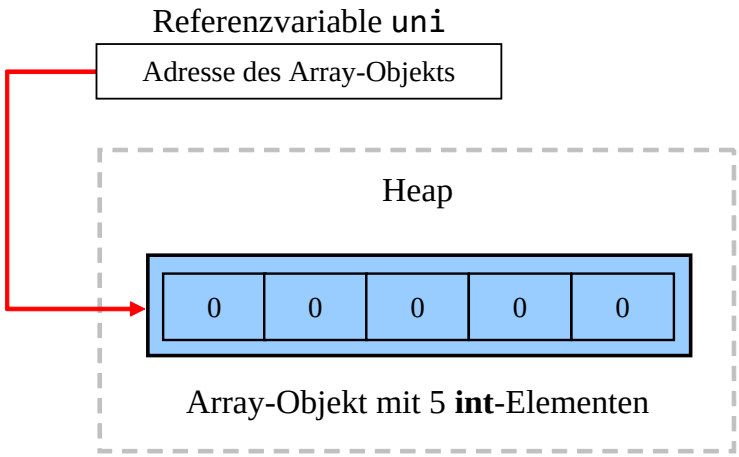
```
uni = new int[max+1];
```

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 lassen sich 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 Null-Initialisierung nach den Regeln von Abschnitt 5.2.3. Die **int**-Elemente im Beispiel erhalten folglich den Startwert 0.

Als Objekt wird ein Array vom Garbage Collector entsorgt, wenn es nicht mehr per Referenz erreichbar ist (vgl. Abschnitt 5.4.5). Um eine Referenzvariable aktiv von einem Array-Objekt zu „entkoppeln“, kann man ihr z. B. das Referenzliteral **null** oder aber ein alternatives Referenzziel zuweisen. Es ist ohne weiteres möglich, dass mehrere Referenzvariablen auf dasselbe Array-Objekt zeigen, z. B.:

Quellcode	Ausgabe

`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. Die statische Methode **Resize()** der Klasse **System.Array** erlaubt 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 dorthin kopieren, was einen erheblichen Aufwand verursacht. Solche Aktionen werden von Kollektionstypen bei Bedarf im Hintergrund automatisch ausgeführt.[1]

Weil die Methode **Resize()** ein neues Array-Objekt anlegen und dessen Adresse an den Aufrufer übergeben muss, hat der zuständige erste Parameter den Modifikator **ref** erhalten, der beim Aufruf anzugeben ist.

Eine Tücke beim Einsatz der Methode **Resize()** besteht darin, dass andere Referenzen auf den Parameter-Array von der Neukreation nichts bemerken und weiterhin auf das ursprüngliche Array-Objekt zeigen, z. B.:

---

[1] Im Abschnitt 6.2.11 wird die nicht-generische Kollektionsklasse **ArrayList** vorgestellt. Nachdem wir uns im Kapitel 8 mit dem typgenerischen Programmieren beschäftigt haben, können im Kapitel 11 die meist zu bevorzugenden generischen Kollektionsklassen behandelt werden.

Quellcode	Ausgabe

`int[] a1 = new int[3];`
`a1[0] = 1; a1[1] = 2; a1[2] = 3;`
`int[] a2 = a1;`
`Array.Resize(ref a1, 5);`
`a1[3] = 4; a1[4] = 5;`
`foreach (int i in a1)`
`  Console.Write(i + " ");`
`Console.WriteLine();`
`foreach (int i in a2)`
`  Console.Write(i + " ");`
	`1 2 3 4 5`
	`1 2 3`

Kollektionsklassen sind von diesem Referenzproblem nicht betroffen und erledigen außerdem eine erforderliche Größenveränderungen autonom. Allerdings ist auch hier die „Vergrößerung“ durch die Neukreation des intern verwendeten Arrays und das Kopieren der Elemente ein zeitaufwändiges Verfahren.

6.2.3 Arrays benutzen

Der Zugriff auf die Elemente eines Array-Objekts geschieht über eine zugehörige Referenzvariable, an deren Namen zwischen eckigen Klammern ein passender Index anzuhängen ist. Dabei ist ein beliebiger Ausdruck mit einem nicht-negativen 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,

```
for (int i = 0; i < drl; i++)
  uni[zzg.Next(5)]++;
```

auf den die Referenzvariable `uni` aufgrund der im Abschnitt 6.2.2 beschriebenen Deklaration und Initialisierung zeigt:

```
int[] uni = new int[5];
```

Den Indexwert liefert die von einem **Random**-Objekt ausgeführte Methode **Next()** mit dem Rückgabetyp **int** (siehe unten). Die **for**-Anweisung stammt aus einer Methode, die im Abschnitt 6.2.4 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. Somit existiert z. B. nach der Anweisung

```
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 13), dann wird es vom Laufzeitsystem beendet.

Die (z. B. durch eine Benutzerentscheidung zur Laufzeit festgelegte) Länge eines Array-Objekts lässt sich über seine (get-only -) Eigenschaft **Length** (vom Typ **int)** jederzeit ermitteln, z. B.:

Quellcode	Eingabe (**fett**) und Ausgabe

`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`

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.

6.2.4 Beispiel: Beurteilung des Pseudozufallszahlengenerators in der Klasse Random

Im bisherigen Verlauf des Abschnitts 6.2 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 **Pseudozufallszahlengenerators** in der Klasse **Random** aus dem Namensraum **System** überprüft. Dieser Generator produziert Folgen von Zahlen mit einem bestimmten Verteilungsverhalten. Obwohl eine Serie perfekt vom Initialisierungswert des Pseudozufallszahlengenerators abhängt, kann sie in der Regel echte Zufallszahlen ersetzen. Manchmal ist es sogar von Vorteil, eine Serie über einen festen Initialisierungswert reproduzieren zu können. Meist verwendet man aber variable Initialisierungen, z. B. abgeleitet aus einer Zeitangabe. Der Einfachheit halber spricht man oft von *Zufallszahlen* und lässt den Zusatz *Pseudo* weg.

Man kann mit moderner EDV-Technik unter Verwendung von physischen 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**, das als Pseudozufallszahlengenerator taugt:

```
var zzg = new Random();
```

Durch die Verwendung des parameterfreien **Random**-Konstruktors entscheidet man sich für einen aus der Systemzeit abgeleiteten Initialisierungswert für den Pseudozufall.

Das angekündigte Programm zieht 10.000 Zufallszahlen aus der Menge {0, 1, ..., 4} und überprüft die empirische Verteilung dieser Stichprobe:

```
const int drl = 10_000;
int[] uni = new int[5];
var zzg = new Random();
for (int i = 0; i < drl; i++)
   uni[zzg.Next(5)]++;
Console.WriteLine("Absolute Häufigkeiten:");
for (int i = 0; i < 5; i++)
   Console.Write(uni[i] + " ");
Console.WriteLine("\n\nRelative Häufigkeiten:");
for (int 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 statische Methode **Next()** der Klasse **Random** liefert die BCL-Dokumentation ausführliche Informationen, die vom Visual Studio aus z. B. so zu erreichen sind:

- Einfügemarke auf den Methodennamen setzen
- Funktionstaste **F1** drücken

Der voreingestellte Browser zeigt ein HTML-Dokument mit den Überladungen der Methode:

Überlädt

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 zufällige ganze Zahl zurück, die in einem angegebenen Bereich liegt.

Über die im Beispiel verwendete mittlere Überladung erhält man die folgenden Informationen:

Next(Int32)

Gibt eine nicht negative Zufallsganzzahl zurück, die kleiner als das angegebene Maximum ist.

C# Kopieren

```
public virtual int Next (int maxValue);
```

Parameter

`maxValue` Int32

Die exklusive obere Grenze der Zufallszahl, die generiert werden soll. `maxValue` muss größer oder gleich 0 sein.

Gibt zurück

Int32

Eine ganze 32-Bit-Zahl mit Vorzeichen, die größer oder gleich 0 (null) und kleiner als `maxValue` ist, d.h., der Bereich der Rückgabewerte umfasst in der Regel 0 (null), aber nicht `maxValue`. Wenn `maxValue` jedoch gleich 0 ist, wird 0 zurückgegeben.

Ausnahmen

ArgumentOutOfRangeException

`maxValue` ist kleiner als 0.

Eine weitere Anleitung zur Nutzung der BCL-Dokumentation ist in diesem Manuskript sicher nicht mehr erforderlich.

6.2.5 Initialisierungslisten

Bei einem Array mit wenigen Elementen ist die Möglichkeit von Interesse, beim Deklarieren der Referenzvariablen eine Initialisierungsliste mit den Werten für die Elementvariablen anzugeben und das Array-Objekt dabei implizit (ohne Verwendung des **new**-Operators) zu erzeugen, z. B.:

Quellcode	Ausgabe
`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, sodass keine erweiternde Typanpassung erforderlich ist, dann 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};
```

Weil der Compiler den Datentyp **int[]** des **new**-Ausdrucks erkennt, ist auch die folgende Deklaration mit Initialisierung möglich:

```
var wecktor = new[] { 1, 2, 3 };
```

Die implizite Objektkreation (*ohne* Schlüsselwort **new**) akzeptiert der Compiler ausschließlich bei der Array-Deklaration mit Initialisierung, im folgenden Beispiel also *nicht*:

```
wecktor = {1, 2, 3}; // Nicht erlaubt
```

6.2.6 Suchen und Sortieren

Die Klasse **Array** (Namensraum **System**) bietet mehrere statische Methoden, die ein 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

`const int len = 100_000;`
`const int mxr = 50_000;`
`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.

Um ein Array statt in der von **IndexOf()** verwendeten aufsteigenden Suchreihenfolge vom Ende her zu durchsuchen, nutzt man die Methode **LastIndexOf().**

Während die beiden eben genannten Methoden nach einem festen Wert suchen, fahnden die Methoden **FindIndex()** und **FindLastIndex()** nach einem Wert mit einer bestimmten Eigenschaft. Im folgenden Beispiel wird der Index des ersten restfrei durch 3 teilbaren Elements in einem per Initialisierungsliste (vgl. Abschnitt 6.2.5) erstellten **int**-Array ermittelt:

---

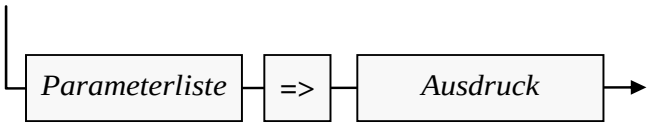
[1] Im aktuellen Abschnitt wird (die Fähigkeit des Compilers zur Typinferenz ausnutzend) die generische Natur der Methoden **IndexOf<T>()**, **LastIndexOf<T>()**, **FindIndex<T>()** und **FindLastIndex<T>()** aus didaktischen Gründen unterschlagen.

Quellcode	Ausgabe

`int[] iar = { 2, 13, 12, 4, 7 };`
`Console.WriteLine("Index: " +`
`  Array.FindIndex(iar, (int i) => i % 3 == 0));`
	`Index: 2`

Als zweiter Parameter wird im Beispiel an die Methode **FindIndex()** ein per **Ausdrucks-Lambda** realisiertes **Delegatenobjekt** (siehe Abschnitt 10.1.5.2) übergeben. Es führt eine Methode zur Beurteilung der Array-Elemente aus (Rückgabetyp **bool**). Die Syntax für ein per Ausdrucks-Lambda realisiertes Delegatenobjekt zeigt eine starke Ähnlichkeit zu der im Abschnitt 5.8.1 beschriebenen Methodendefinition per Lambda-Symbol und Ausdruck:

Delegatenobjekt per Ausdrucks-Lambda



Wird die Methode **FindIndex()** nicht fündig, liefert sie die Rückgabe -1.

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 statischen **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,[1]
- oder ein die Schnittstelle **IComparer<T>** implementierendes Objekt engagiert wird, das eine beliebig definierte Anordnung von zwei Elementen liefert.

Das folgende Programm sortiert **int**-Werte auf Basis der natürlichen Ordnung:

Quellcode	Ausgabe

`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 in sortiertem Zustand, dann kann die Suche nach einem Element durch das in der statischen **Array**-Methode **BinarySearch()** implementierte **binäre Suchverfahren** im Vergleich zu den oben beschriebenen Suchmethoden erheblich beschleunigt werden:

[1] Eine mit **CompareTo()** beauftragte Instanz vergleicht sich mit dem Aktualparameter und liefert die Rückgaben -1, 0, oder 1, wenn sie kleiner, gleich oder größer als der Aktualparameter ist. Die Methode **CompareTo()** wird später (z. B. im Abschnitt 6.3.1.2.2) noch ausführlich behandelt.

- 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, dann wird die Suche in der oberen Array-Hälfte fortgesetzt.
  - Ist der gesuchte Wert kleiner als das mittlere Array-Element, dann wird die Suche in der unteren Array-Hälfte fortgesetzt.
- Im zweiten Schritt wird in der relevanten Array-Hälfte das mittlere Element aufgesucht usw.

Das folgende Beispielprogramm zeigt allerdings, dass der Zeitgewinn beim binären Suchen im Vergleich zum einfachen Suchen nur bei einer hohen Anzahl von Suchvorgängen den Aufwand des vorherigen Sortierens rechtfertigt:

```
using System.Diagnostics;

const int len = 1_000_000;
const int nCand = 1_000;
const string format = "\n{0,-30}{1,5} ms";
Stopwatch stopwatch = new();

int[] iar = new int[len];
Random zzg = new();
for (int i = 0; i < len; i++)
    iar[i] = zzg.Next(len);

stopwatch.Start();
for (int i = 0; i < nCand; i++)
    Array.IndexOf(iar, zzg.Next(len));
stopwatch.Stop();
Console.WriteLine(format, "Zeit für die einfache Suche:",
                  stopwatch.ElapsedMilliseconds);

stopwatch.Restart();
Array.Sort(iar);
stopwatch.Stop();
Console.WriteLine(format, "Zeit für das Sortieren:",
                  stopwatch.ElapsedMilliseconds);

stopwatch.Restart();
for (int i = 0; i < nCand; i++)
    Array.BinarySearch(iar, zzg.Next(len));
stopwatch.Stop();
Console.WriteLine(format, "Zeit für die binäre Suche:",
                  stopwatch.ElapsedMilliseconds);
```

In einem Array mit 1 Milliarde **int**-Elementen mit Zufallswerten aus dem Bereich von 0 bis 1.000.000-1 werden `nCand` Werte gesucht:

- Vor dem Sortieren mit der **Array**-Methode **IndexOf()**
- Nach dem Sortieren mit der **Array**-Methode **BinarySearch()**

Bei **BinarySearch()** signalisiert eine negative Rückgabe eine gescheiterte Suche.[1] Die folgenden Messergebnisse wurden mit der Release-Konfiguration des Projekts ermittelt:

[1] Zur genauen Bedeutung der negativen Rückgabewerte siehe:
https://learn.microsoft.com/en-us/dotnet/api/system.array.binarysearch

nCand = 10	nCand = 1.000
Zeit für die einfache Suche: 8 ms Zeit für das Sortieren: 87 ms Zeit für die binäre Suche: 0 ms	Zeit für die einfache Suche: 345 ms Zeit für das Sortieren: 88 ms Zeit für die binäre Suche: 0 ms

Die binäre Suche ist also nur dann sinnvoll, wenn ein Array bereits sortiert vorliegt, oder wenn viele Suchvorgänge erforderlich sind.

Ein Visual Studio - Projekt mit dem Benchmark-Programm ist hier zu finden:

…\BspUeb\Weitere .NETte Typen\Arrays\SortSearch

6.2.7 Objekte als Array-Elemente

Für die Elemente eines Arrays ist auch ein Referenztyp erlaubt. Im folgenden Beispiel wird ein Array mit `Bruch`-Objekten erzeugt:

Quellcode

```
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();
  }
}
```

Ausgabe

```
        5
b2 =  -----
        6
```

6.2.8 Indizes und Bereiche

Für Arrays und andere Typen mit Indexzugriff (z. B. **String**) bietet C# seit der Version 8.0 Verbesserungen bei der Ansprache von einzelnen Elementen und von zusammenhängenden Bereichen.

Mit Hilfe des ^ - Operators lässt sich ein Element mit Bezug auf das *Ende* des Arrays ansprechen, wobei ^1 das letzte Element adressiert, ^2 das vorletzte usw. Im folgenden Beispiel werden Elemente in einem **char**-Array angesprochen:

Quellcode

```
char[] ca = { 'a', 'b', 'c', 'd' };
Console.WriteLine($"{ca[^1]} {ca[^2]} {ca[^3]}");
```

Ausgabe

```
d c b
```

Weil z. B. beim Array `ca` der Ausdruck

```
ca[^i]
```

für das Element

```
ca[ca.Length - i]
```

steht, existiert das Element `ca[^0]` nicht, sodass ein Zugriffsversuch zu einer **IndexOutOfRangeException** führt:

Quellcode

```
Console.WriteLine($"{ca[^0]}");
```

Ausgabe

```
Unhandled exception.
System.IndexOutOfRangeException
```

Mit dem Bereichsoperator **..** lässt sich der Bereich von einer Startposition (inklusive) bis zu einer Endposition (exklusive) ansprechen, z. B.:

```
var bc = ca[1 .. ^1]; // {'b', 'c'}
```

Bei einer fehlenden Startposition werden alle Elemente bis zur angegebenen Endposition (exklusive) angesprochen, z. B.:

```
var abc = ca[.. ^1]; // {'a', 'b', 'c'}, äquivalent: var abc = ca[0 .. ^1]
```

Bei einer fehlenden Endposition werden alle Elemente ab der angegebenen Startposition (inklusive) angesprochen, z. B.:

```
var cd = ca[2 ..]; // {'c', 'd' }, äquivalent: var cd = ca[2 .. ^0]
```

6.2.9 Mehrdimensionale Arrays

In der linearen Algebra und in vielen anderen Anwendungsbereichen werden auch *mehrdimensionale* Arrays benötigt, z. B.:

Quellcode	Ausgabe

`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, wobei sich die Zellen per Doppelindizierung ansprechen lassen. Bei der Erzeugung bzw. Verwendung eines mehrdimensionalen Arrays sind die in eckigen Klammern eingeschlossenen Dimensionsangaben bzw. Indexwerte jeweils durch ein Komma zu trennen.

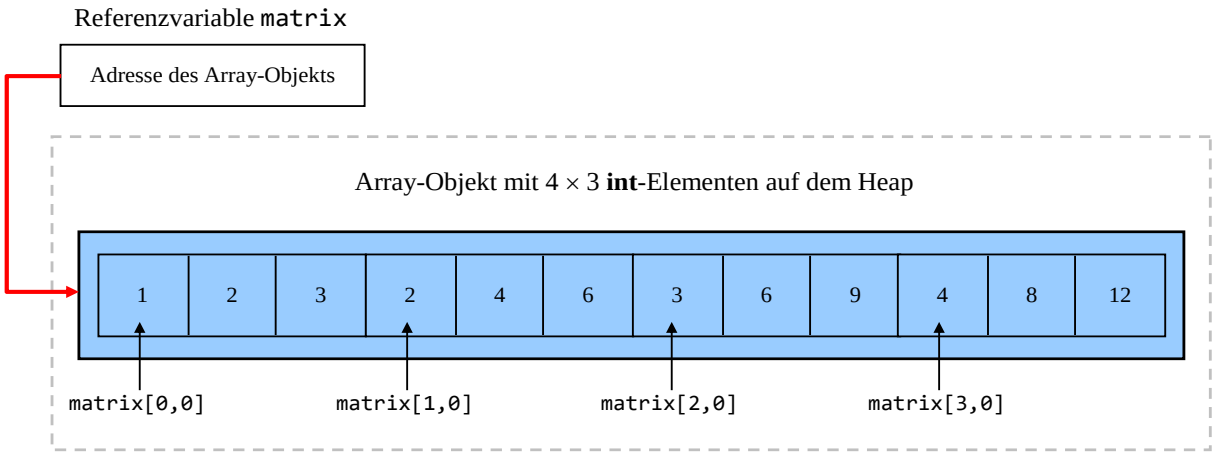
Auch im mehrdimensionalen Fall sind Initialisierungslisten erlaubt, wobei z. B. die Elemente einer (zweidimensionalen) Matrix zeilendominant anzugeben sind (als Liste von Zeilenvektoren):

```
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 BCL-Dokumentation), z. B.:

- Die Eigenschaft **Rank** enthält die Anzahl der Dimensionen. Diese Eigenschaft hat wenig Bezug zum mathematischen Begriff des Rangs einer Matrix (Anzahl der linear unabhängigen Zeilen- oder Spaltenvektoren).
- Über die Methode **GetLength()** erfährt man von einem Array-Objekt die Anzahl der Elemente in der per Parameter angegebenen Dimension.
- Die Eigenschaft **Length** mit dem Datentyp **int** liefert die Gesamtzahl der Array-Elemente, im Beispiel also $4 \cdot 3 = 12$.
- Hat ein mehrdimensionaler Array mehr als 2^{31}-1 Elemente, dann verwendet man statt **Length** die Eigenschaft **LongLength** mit dem Datentyp **long**, um die Gesamtzahl der Elemente zu ermitteln.

Auch bei einem mehrdimensionalen Array liegen die Elemente im Speicher unmittelbar hintereinander, so dass ein schneller Indexzugriff möglich ist, z. B.:



Es ist eine beliebige Anzahl von Dimensionen erlaubt, wobei aber Arrays mit mehr als zwei Dimensionen kaum benötigt werden.

6.2.10 Array aus Arrays

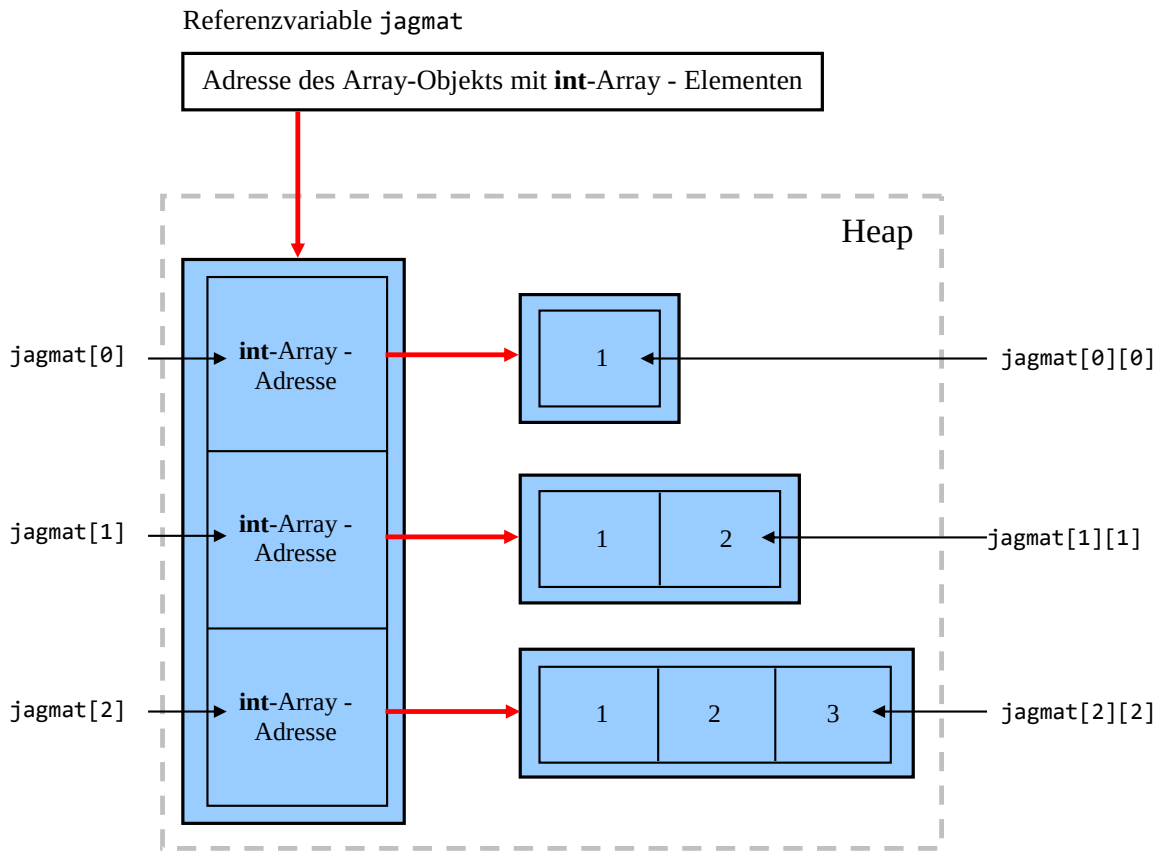
C# unterstützt auch Arrays, die als Elemente wiederum Arrays enthalten. So lässt sich etwa eine zweidimensionale Matrix mit unterschiedlich langen Zeilen realisieren:

Quellcode	Ausgabe

`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`

Man spricht hier auch von „ausgesägten“ Arrays (engl.: *jagged arrays*).

Im Unterschied zum **int[,]** - Objekt `matrix` aus dem Beispiel im Abschnitt 6.2.9, das doppelt indizierte **int**-Elemente enthält, handelt es sich bei den Elementen des **int[][]** - Objekts `jagmat` um 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 **int[]** - Objekte und legt ihre Adressen in den `jagmat`-Elementvariablen ab, z. B.:

```
jagmat[2] = new int[3] {1, 2, 3};
```

Mit Hilfe einer im Abschnitt 6.2.5 erläuterten Initialisierungsliste lässt sich etwas Schreibarbeit sparen:

```
jagmat[2] = new[] { 1, 2, 3 };
```

Aus den Initialisierungsausdrücken zu den inneren Arrays lässt sich eine Initialisierungsliste für den äußeren Array bilden:

```
int[][] jagmat = {new[] {1}, new[] {1, 2}, new[] {1, 2, 3}};
```

Im Unterschied zur Initialisierungsliste für einen mehrdimensionalen Array (vgl. Abschnitt 6.2.9) muss der **new**-Operator für jeden inneren Array wiederholt werden. Es entsteht nicht (wie im mehrdimensionalen Fall) *ein* Array, sondern es entstehen ...

- ein äußerer Array
- und mehrere selbständige innere Arrays.

---

[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. Vermutlich hat man sich daran orientiert, dass die von `jagmat`-Elementen indizierten **int[]** - Arrays als *Zeilenvektoren* aufgefasst werden. In der mathematischen Matrix-Notation wird der Zeilenindex zuerst angegeben (Zeilendominanz).

6.2.11 Kollektionsklasse ArrayList

Im Namensraum **System.Collections** bietet die BCL etliche Klassen zur flexiblen Verwaltung von Datenbeständen *variablen* Umfangs, mit denen wir uns im Kapitel 11 detailliert beschäftigen werden. Ein Objekt der Kollektionsklasse **ArrayList** kann analog zu eindimensionalen Arrays genutzt werden. Man muss jedoch beim Erzeugen eines **ArrayList**-Objekts keinen Umfang festlegen, sondern kann z. B. mit der Methode **Add()** nach Bedarf neue Elemente einfügen:

Quellcode	Eingabe (**fett**) und Ausgabe

```
using System.Collections;
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** lässt sich die momentane Kapazität ermitteln und auch einstellen. 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 5.11), kann man per Indexsyntax (über einen durch rechteckige Klammern begrenzten ganzzahligen Ausdruck, siehe Beispiel) 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 .NET - Typsystems. Folglich darf man in einen **ArrayList**-Container Daten beliebigen Typs einfüllen, wobei dank Boxing-Technik (siehe Abschnitt 6.1.6) auch die Werttypen erlaubt sind, z. B.:

Quellcode	Ausgabe

```
using System.Collections;
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 die Wahl des Element-Datentyps **Object** auch mit einem einfachen Array zu realisieren.

Im Kapitel 8 über das typgenerische Programmieren wird sich herausstellen, dass dem Gemischtwarenladen **ArrayList** in 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.

6.3 Klassen für Zeichenketten

C# bietet für die Verwaltung von Zeichenfolgen, die grundsätzlich aus Unicode-Zeichen bestehen, zwei Klassen an, die für unterschiedliche Einsatzzwecke optimiert wurden:

- **String** (im Namensraum **System**)
  String-Objekte können nach dem Erzeugen nicht mehr geändert werden. Momentan erscheinen Ihnen unveränderliche Objekte eventuell noch als eingeschränkt brauchbar. Im weiteren Verlauf des Manuskripts wird aber immer öfter von den Vorteilen unveränderlicher Objekte zu hören sein (z. B. im Zusammenhang mit dem Multithreading).
- **StringBuilder** (im Namensraum **System.Text**)
  Für *variable* Zeichenketten sollte allerdings unbedingt die Klasse **StringBuilder** verwendet werden, weil deren Objekte nach dem Erzeugen noch modifiziert werden können.

6.3.1 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:

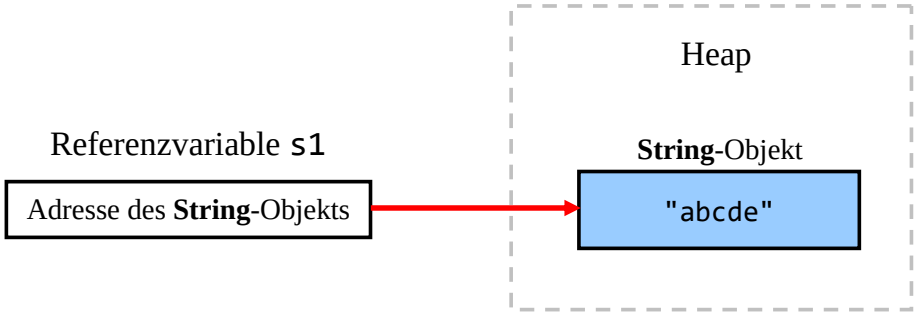
- eine **String**-Referenzvariable namens `s1` angelegt,
- ein neues **String**-Objekt mit dem Inhalt „abcde“ auf dem Heap erzeugt,
- und die Adresse des neuen Heap-Objekts in der Referenzvariablen abgelegt.

So viel 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, sodass 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 die folgende Situation:



6.3.1.1 String als WORM - Klasse

Nachdem ein **String**-Objekt auf dem Heap erzeugt wurde, ist es **unveränderlich** (engl.: *immutable*). In der Abschnittsüberschrift wird für diesen Sachverhalt eine Abkürzung aus dem Bereich der Datenspeicherungs-Hardware ausgeliehen: WORM (***W**rite **O**nce **R**ead **M**any*). Es könnte jemand die Unveränderlichkeit von **String**-Objekten in Zweifel ziehen und ein Gegenbeispiel der folgenden Art vorbringen:

Quellcode	Ausgabe

`String testr = "abc";`
`Console.WriteLine("testr = " + testr);`
`testr += "def";`
`Console.WriteLine("testr = " + testr);`
	`testr = abc`
	`testr = abcdef`

In der Zeile

```
testr += "def";
```

wird aber das per `testr` ansprechbare **String**-Objekt (mit dem Text „abc“) nicht geändert, sondern durch ein neues **String**-Objekt (mit dem Inhalt „abcdef“) ersetzt. Das alte Objekt ist noch vorhanden, aber nicht mehr referenziert. Sobald die CLR bei einer Heap-Anforderung zu wenig freien Platz feststellt, wird das obsolete **String**-Objekt vom Garbage Collector eliminiert (siehe Abschnitt 5.4.5.1).

6.3.1.2 Methoden für String-Objekte

Von den zahlreichen Instanzmethoden und -eigenschaften der Klasse **String** werden in diesem Abschnitt nur die wichtigsten angesprochen. Für spezielle Anwendungen lohnt sich also ein Blick in die BCL-Dokumentation.

6.3.1.2.1 Verketten von Strings

Weil die Klasse **String** den „+“ - Operator geeignet überladen hat (vgl. Abschnitt 5.8.3), taugt er zum Verketten von **String**-Objekten, wobei Operanden beliebiger Datentypen bei Bedarf automatisch in **String**-Objekte konvertiert werden. Wie Sie aus dem Abschnitt 6.3.1.1 wissen, entsteht beim Verketten von zwei Zeichenfolgen ein *neues* **String**-Objekt. Im ersten **WriteLine()** - Aufruf des folgenden Beispiels wird mit Klammern dafür gesorgt, dass der Compiler die „+“ - Operatoren jeweils sinnvoll interpretiert (Verketten von **String**-Objekten bzw. Addieren von Zahlen):

Quellcode	Ausgabe
`Console.WriteLine("4 + 3 = " + (4 + 3));` `Console.WriteLine((15 + 2) + " ist eine Primzahl.");`	`4 + 3 = 7` `17 ist eine Primzahl.`

Wie der zweite **WriteLine()** - Aufruf zeigt, hängt die Interpretation des „+“ - Zeichens als Operator-Überladung der Klasse **String** nicht davon ab, ob das Zeichenfolgen-Argument links oder rechts vom Operatorsymbol steht.

6.3.1.2.2 Vergleichen von Strings

Angewendet auf **String**-Variablen vergleichen die auf **Identität** prüfenden Operatoren „==“ und „**!=**“ *nicht* (wie z. B. in Java) die in den Variablen abgelegten *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 im Abschnitt 6.3.1.3 demonstriert wird, ist bei einer großen Anzahl von **String**-Vergleichen durch das sogenannte Internalisieren und die Verwendung von Adressvergleichen eine Leistungssteigerung zu erzielen.

Zum Testen auf **lexikographische Priorität** (z. B. beim Sortieren) kann die **String**-Methode **CompareTo()**

public int CompareTo (string *verglString***)**

dienen, z. B.:

Quellcode	Ausgabe

`String a = "Müller, Anja", b = "Müller, Kurt", c = "Müller, Anja";`
`Console.WriteLine("<  : " + a.CompareTo(b));`
`Console.WriteLine("== : " + a.CompareTo(c));`
`Console.WriteLine(">  : " + b.CompareTo(a));`
	`<  : -1`
	`== : 0`
	`>  : 1`

CompareTo() liefert die folgenden **int**-Rückgabewerte:

		CompareTo() - Ergebnis
Die lexikographische Priorität des angesprochenen **String**-Objekts ist im Vergleich zum Parameterobjekt:	kleiner	-1
	gleich	0
	größer	1

6.3.1.2.3 Länge einer Zeichenkette

Über die Länge einer Zeichenkette informiert die **String**-Eigenschaft **Length**, z. B.:

Quellcode	Ausgabe
`Console.WriteLine("abc".Length);`	`3`

6.3.1.2.4 Zeichen(folgen) extrahieren, suchen oder ersetzen

Auf einzelne Zeichen eines Strings kann man per Indexsyntax zugreifen, z. B.:

Quellcode	Ausgabe

`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** *n***)**

erhält man von einem **String**-Objekt die *n* Zeichen ab Position *start* (inklusive) als Kopie.

Mit der Methode

public int IndexOf(string *gesucht***)**

befragt man einen **String** nach der Existenz einer anderen Zeichenfolge. Als Rückgabewert erhält man ...

- nach einer erfolgreichen Suche: die (0-basierte) Startposition der ersten Trefferstelle
- nach einer vergeblichen 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.

6.3.1.2.5 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

`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, sodass eine kurze Erklärung angemessen ist:

- Das von der Variablen `a` referenzierte **String**-Objekt führt die Methode **ToUpper()** aus. Dabei entsteht ein neues **String**-Objekt, und der Ausdruck `a.ToUpper()` ist eine Referenz auf dieses Objekt.
- Diese Referenz ermöglicht es, dem neuen Objekt Botschaften zu übermitteln, was durch den Methodenaufruf `IndexOf("T")` geschieht.

6.3.1.3 Interner String-Pool

Ein Zeichenkettenliteral ist ein Ausdruck vom Typ **String** mit der Speicheradresse eines Objekts als Wert. Für wiederholt im Quellcode auftretende identische Zeichenkettenliterale lässt dieses Prinzip eine Verschwendung von Speicherplatz durch eine Vielzahl von Objekten mit identischem Inhalt befürchten. Um das zu verhindern, verwaltet die CLR eine als **interner String-Pool** bezeichnete Tabelle ...

- für die auf Literalen basierenden **String**-Objekte
- und für explizit internalisierte **String**-Objekte (siehe unten).

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();`
`  Console.WriteLine(Object.ReferenceEquals(ls, p.sf));`
`  }`
`}`
	`True`

Dass die beiden Referenzvariablen tatsächlich auf dasselbe Objekt zeigen, wird durch die statische **Object**-Methode **ReferenceEquals()** nachgewiesen.

Über die statische **String**-Methode **Intern()** kann man den internen **String**-Pool zusätzlich bevölkern:

public static string Intern (string *str***)**

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 **String**-Objekte (vgl. Abschnitt 6.3.1.2.2) beschleunigen, weil bei Referenzvariablen zu internalisierten **String**-Objekten aus der Gleichheit der Adressen bereits die Inhaltsgleichheit folgt. Im folgenden Programm werden `anz` Zufallszeichenfolgen der Länge `len` jeweils `vergl` mal mit einem zufällig gewählten Partner verglichen. Dies geschieht zunächst per Inhaltsvergleich und dann nach dem zwischenzeitlichen Internalisieren per Adressvergleich:

```
using System;
using System.Diagnostics;
using System.Text;

class StringIntern {
    public static void Main() {
        const int anz = 50_000, len = 50, vergl = 5;
        var sb = new StringBuilder();
        var ran = new Random();
        var stopwatch = new Stopwatch();
        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);
        }

        int hits = 0;
        // anz * vergl Inhaltsvergleiche
        stopwatch.Start();
        for (int i = 0; i < anz; i++)
            for (int n = 1; n <= vergl; n++)
                if (sar[i] == sar[ran.Next(anz)])
                    hits++;
        stopwatch.Stop();
        Console.WriteLine((anz * vergl) + " Inhaltsvergleiche (" + hits +
                " hits) benötigen " + stopwatch.ElapsedMilliseconds +
                " ms");

        hits = 0;
        stopwatch.Restart();
        // Internalisieren
        for (int j = 1; j < anz; j++)
            sar[j] = String.Intern(sar[j]);
        stopwatch.Stop();
        Console.WriteLine("Zeit für das Internalisieren: " +
                stopwatch.ElapsedMilliseconds + " ms");

        // anz * vergl Adressvergleiche
        stopwatch.Start();
        for (int i = 0; i < anz; i++)
            for (int n = 1; n <= vergl; n++)
                if ((Object)sar[i] == sar[ran.Next(anz)])
                    hits++;
        stopwatch.Stop();
        Console.WriteLine((anz * vergl) + " Adressvergleiche (" + hits +
                          " hits) benötigen (inkl. Internalisieren) " +
                          stopwatch.ElapsedMilliseconds + " ms");
    }
}
```

Beim Erzeugen der Zufallszeichenfolgen kommt ein Objekt der Klasse **StringBuilder** zum Einsatz (siehe Abschnitt 6.3.2).

Um den Identitätsoperator zu einem Adressvergleich 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 `vergl` ab, welche Vergleichsmethode überlegen ist:[1]

[1] Die Ergebnisse wurden unter Verwendung der Visual Studio - Release-Konfiguration auf einem PC mit Intel - CPU Core i3 550 (3,2 GHz) unter Windows 10 (64 Bit) ermittelt.

	Laufzeit in Millisekunden	
	Inhaltsvergleiche	Internalisieren + Adressvergl.
`anz` = 50000, `len` = 50, `vergl` = 5	25	34
`anz` = 50000, `len` = 50, `vergl` = 50	189	62

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 **String**-Parameterobjekts, falls es sich im internen Pool befindet:

public static string IsInterned (string *str***)**

Anderenfalls wird jedoch *kein* Pool-String erzeugt, sondern der Wert **null** abgeliefert.

Wird bei der Verwendung des internen **String**-Pools vor allem der Zweck verfolgt, Speicherplatz zu sparen, dann sollten auch die folgenden Punkte beachtet werden:[1]

- Im internen Pool befindliche **String**-Objekte persistieren in der Regel auch dann, wenn keine Referenzen mehr im Programm vorhanden sind. Sie sind zwar nicht grundsätzlich aus dem Aufgabenbereich des Garbage Collectors ausgeschlossen, existieren aber meist bis zum Prozessende.
- Bevor ein **String**-Objekt in den Pool gelangt, muss es zunächst als normales Heap-Objekt erstellt werden. Eine obsolet gewordene Heap-Instanz belegt Speicher, bis sie vom Garbage Collector abgeräumt wird.

Ein Visual Studio - Projekt mit dem Programm zur Beurteilung des Internierungs-Effekts auf **String**-Vergleiche ist hier zu finden:

…\BspUeb\Weitere .NETte Typen\Arrays\SortSearch

6.3.2 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 das Erstellen eines neuen Objekts und die Entsorgung des obsolet gewordenen Objekts entfallen.

Ein **StringBuilder**-Objekt kann nicht *implizit* erzeugt werden, jedoch stehen bequeme Konstruktoren zur Verfügung, z. B.:

- **public StringBuilder**()
- **public StringBuilder**(**String** *str*)

Im folgenden Programm wird eine Zeichenkette 10.0000 Mal verlängert, zunächst mit Hilfe der Klasse **String**, dann mit Hilfe der Klasse **StringBuilder**:

[1] https://docs.microsoft.com/en-us/dotnet/api/system.string.intern

```
using System;
using System.Diagnostics;
using System.Text;

class StrBldrDemo {
    static void Main() {
        const int n = 500_000;
        string s = "*";
        var stopwatch = new Stopwatch();
        const string format = "{0,-40}{1,8} ms";

        stopwatch.Start();
        for (int i = 0; i < n; i++)
            s += '*';
        stopwatch.Stop();
        Console.WriteLine(format, "Zeit für String-Manipulation:",
                          stopwatch.ElapsedMilliseconds);

        var t = new StringBuilder('*');
        stopwatch.Restart();
        for (int i = 0; i < n; i++)
            t.Append('*');
        stopwatch.Stop();
        Console.WriteLine(format, "Zeit für StringBuilder-Manipulation:",
                          stopwatch.ElapsedMilliseconds);
    }
}
```

Die (in Millisekunden gemessenen) Laufzeiten unterscheiden sich erheblich:[1]

```
Zeit für String-Manipulation:             109881 ms
Zeit für StringBuilder-Manipulation:           2 ms
```

Ein **StringBuilder**-Objekt kennt u. a. die folgenden Methoden und Eigenschaften (alle **public**):

[1] Die Ergebnisse wurden unter Verwendung der Visual Studio - Release-Konfiguration auf einem PC mit Intel - CPU Core i3 550 (3,2 GHz) unter Windows 10 (64 Bit) ermittelt.

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 der vom ersten Parameter bestimmten Position eingefügt, z. B.: `sb.Insert(4, 3.14);`
Remove()	Ab einer Startposition wird eine Anzahl von Zeichen entfernt, z. B.: `sb.Remove(2, 5);`
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());`

6.4 Aufzählungstypen

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, sanguinisch, 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 Codierungsvorschrift 0 = cholerisch, 1 = melancholisch etc.
  Es wird wenig Speicher benötigt, allerdings ist der Quellcode nur für Eingeweihte zu verstehen, z. B.:

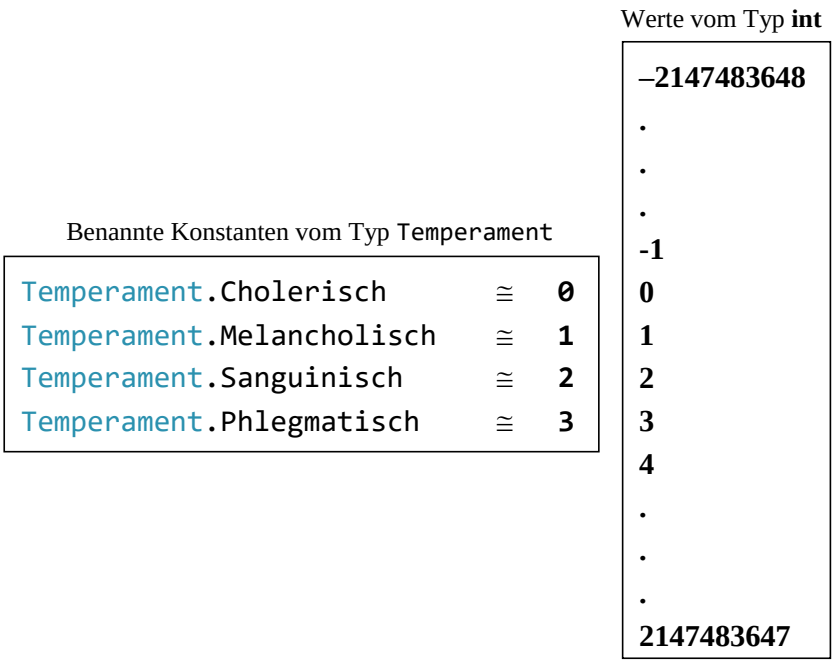
```
public int Temp;
  .  .  .
if (otto.Temp == 3) ...
```

Fehlerhafte Zuweisungen könnte und sollte man bei beiden Lösungsansätzen durch eine sorgfältige Eigenschaftsdefinition verhindern (Abweisen ungeeigneter Werte im **set**-Block, siehe Abschnitt 5.5).

C# bietet mit den **Aufzählungstypen (Enumerationen)** eine Lösung, die alle Anforderungen erfüllt:

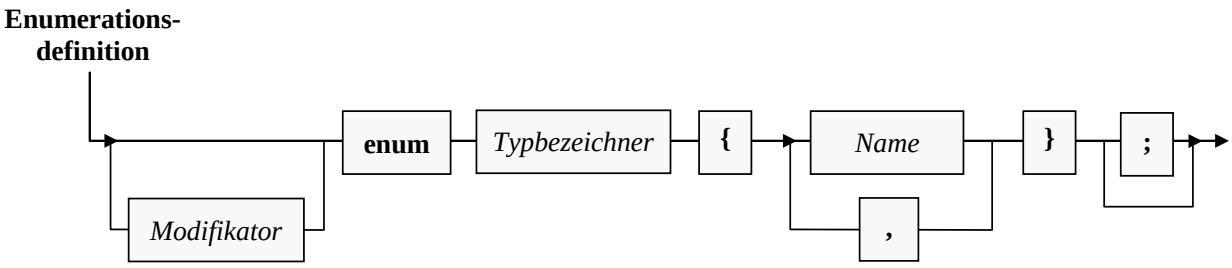
- Gut lesbarer Quellcode durch Klartextnamen für die Merkmalsausprägungen
- Abweisung von falsch geschriebenen Klartextnamen durch den Compiler
- Geringer Speicherbedarf

Eine Enumeration basiert auf einem zugrunde liegenden integralen Typ (meist **int**) und enthält eine (meist kleine) Menge von benannten Konstanten dieses Typs, z. B. die Enumeration `Temperament`:



Sofern man auf eine explizite Typumwandlung verzichtet (siehe unten), können einer Variablen des Enumerationstyps nur die definierten Konstanten zugewiesen werden. Dabei sind nicht die zugrunde liegenden Werte (per Voreinstellung 0, 1, 2, …) zu verwenden, sondern die vereinbarten Namen (z. B. `Temperament.Sanguinisch`).

Bei der Definition eines Aufzählungstyps folgt auf das Schlüsselwort **enum** und den Typbezeichner eine geschweift eingeklammerte Liste mit den Namen für die Konstanten:



Analog zu den Klassen …

- wird die Schutzstufe einer Enumeration über Modifikatoren geregelt (Voreinstellung: **internal**, Alternativen **public** und **file**, vgl. Abschnitt 5.12),
- ist neben einer Top-Level - Definition auch eine eingeschachtelte Definition (innerhalb eines anderen Typs) möglich,
- kann optional hinter der schließenden Klammer der Enumerationsdefinition ein Semikolon stehen.

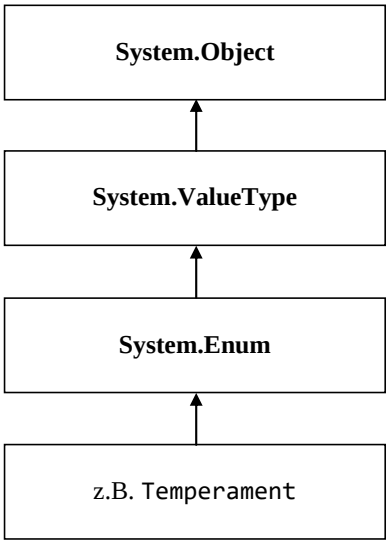
Als Beispiel betrachten wir die von Hippokrates inspirierte Enumeration zur Erfassung des Charakters von Personen:

```
public enum Temperament {Cholerisch, Melancholisch, Sanguinisch, Phlegmatisch}
```

Per Voreinstellung stehen die Namen in der Enumerationsdefinition für eine 0-basierte 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.:

```
public enum Temperament { Cholerisch = 1, Melancholisch, Sanguinisch, Phlegmatisch }
```

Die Enumerationen sind **Werttypen** (wie die Strukturen) und folgendermaßen in das CTS (Common Type System) eingeordnet:



Alternative Abstammungen sind bei Enumerationen *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.Sanguinisch);
      if (otto.Temp == Temperament.Sanguinisch)
         System.Console.WriteLine("Lustiger Typ!");
   }
}
```

Einer Variablen mit einem Enumerationstyp können über eine explizite Typumwandlung neben den benannten Konstanten auch beliebige andere Werte des zugrunde liegenden Typs zugewiesen werden, z. B.:

---

[1] Wir verzichten der Kürze halber bei der Klasse `Person` auf die Datenkapselung.

```
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.:

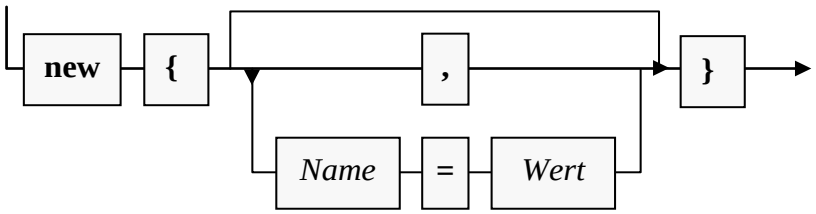
```
Temperament temp;

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.[1]

6.5 Anonyme Klassen

Zu dem im Abschnitt 5.4.3.2 beschriebenen Objektinitialisierer existiert eine Variante *ohne* Klassenname, wobei ein Objekt aus einer anonymen Klasse entsteht:

Objekt einer anonymen Klasse



Im folgenden Beispiel wird es zur Initialisierung einer über das Schlüsselwort **var** implizit typisierten lokalen Variablen verwendet:

Quellcode	Ausgabe
`var a = new {Name = "Knut", Alter = 53 };` `Console.WriteLine(a.GetType());`	`<>f__AnonymousType0`2[System.String,System.Int32]`

Wie das Beispiel zeigt, hat die „anonyme“ Klasse durchaus einen (vom Compiler vergebenen) Namen, doch darf dieser Name im Quellcode nicht verwendet werden. Daher ist das Schlüsselwort **var**, das wir bisher als Schreiberleichterung kennengelernt haben, hier erforderlich, um eine Variable mit dem anonymen Typ deklarieren zu können.

Das Schlüsselwort **var** ist allerdings nur methodenintern erlaubt, sodass man die anonymen Klassen nicht …

- als Parameter- oder Rückgabetyp in einer Methodendefinition
- oder als Typ für ein Feld

verwenden kann.

---

[1] Alternativ zum Operator **typeof** hätte die **Object**-Methode **GetType()** verwendet werden können:
`value.GetType()`

Als Innenausstattung enthält die anonyme Klasse eine öffentliche, automatisch implementierte get-only - Eigenschaft (mit privatem backing field) für jedes Element in der Liste des Objektinitialisierers, sodass im Beispiel eine Klasse mit den Eigenschaften `Name` und `Alter` resultiert, wobei der Compiler die Datentypen (**String** und **Int32**) aus den zugewiesenen Werten ermittelt.

Weil ein Objekt einer anonymen Klasse nur private Felder mit zugehörigen get-only - Eigenschaften besitzt, ist es *unveränderlich*, wie der folgende Fehlversuch demonstriert:

```
var a = new { Name = "Knut", Alter = 53 };
a.Alter = 54;
```

Eine anonyme Klasse erbt die Methoden ihrer Basisklasse **Object**. Aufgrund ihrer Entstehungsgeschichte können anonyme Klassen über das **Object**-Erbgut hinaus keine Handlungskompetenzen besitzen. Immerhin wird die **Object**-Methode **Equals()** *inhaltsorientiert* überschrieben, z. B.:

Quellcode	Ausgabe

`var ano1 = new { i = 13, d = 3.1 };`
`var ano2 = new { i = 13, d = 3.1 };`
`Console.WriteLine(ano1.Equals(ano2));`
	`True`

An der Stelle eines Name-Wert - Paars kann im Objektinitialisierer auch ein Variablen- oder Parametername der Umgebung stehen, der als Eigenschaftsname für die anonyme Klasse übernommen wird, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  string zf = "s";`
`  static void Main() {`
`  Prog p = new();`
`  int alter = 53;`
`  var a = new { p.zf, alter };`
`  Console.WriteLine(a);`
`  }`
`}`
	`{ zf = s, alter = 53 }`

Um den Wert zu einer Eigenschaft festzulegen, ist im Objektinitialisierer auch ein Ausdruck zugelassen, z. B.:

Quellcode	Ausgabe

`int alter = 53;`
`var ano = new { Name = "Otto",`
`  Gen50plus = alter > 50 };`
`Console.WriteLine(ano);`
	`{ Name = Otto, Gen50plus = True }`

Zwei durch anonyme Objektinitialisierer entstandene Objekte gehören nur dann zur selben Klasse, wenn bei den Initialisierungslisten die Namen und Typen der Eigenschaften sowie die Reihenfolgen übereinstimmen. In diesem Fall können die Referenzvariablen einander zugewiesen werden, z. B.:

```
var a = new {Name = "Knut", Alter = 45};
var b = new {Name = "Otto", Alter = 54};
  . . .
b = a;
```

Eine anonyme Klasse ist auch als Elementtyp für ein Array zugelassen, z. B.:

```
var duo = new[] {
    new { Name = "Knut", Alter = 45 },
    new { Name = "Otto", Alter = 54 }
};
```

Seit C# 10 lässt sich der ursprünglich für Record-Typen (siehe Abschnitt 6.7) eingeführte **with**-Operator auch bei anonymen Klassen dazu verwenden, aus einem Objekt eine Kopie mit partiell geänderten Werten zu erstellen, z. B.:

Quellcode	Ausgabe

`var a = new { Verein = "Gesangverein", Vorname = "Otto", Name = "Mock" };`
`var b = a with { Vorname = "Kurt" };`
`Console.WriteLine(b.Vorname);`
	`Kurt`

Die anonymen Klassen wurden in C# 3.0 zur Unterstützung der LINQ-Technik (*Language Integrated Query* eingeführt (siehe Kapitel 19 in Baltes-Götz (2021)). Hier dienen sie als spontan, ohne den Aufwand einer expliziten Definition erstellte Typen meist zur sogenannten *Projektion*. Wegen des geringen syntaktischen Aufwands blieb ihr Einsatz aber nicht auf die LINQ-Technik beschränkt, und aus diesem Grund werden sie schon im aktuellen Kapitel 6 vorgestellt.

Mittlerweile ist C# um Datentypen erweitert worden, die ebenfalls ohne explizite Typdefinition auskommen und in manchen Situationen gegenüber den anonymen Klasen zu bevorzugen sind. So bieten z. B. die im Abschnitt 6.6 vorgestellten Struktur-Tupel eine Wertsemantik (ohne Objektkreation). In der LINQ-Technik sind anonyme Klassen weiterhin angemessen (z. B. wegen der Unterstützung von sogenannten *Ausdrucksbäumen*).

6.6 Tupel

Mehrere Werte bzw. Variablen in einem Datencontainer zusammenzufassen und gemeinsam behandeln zu können (z. B. als Parameter oder Rückgabe einer Methode), ist eine unverzichtbare Option im Alltag der Software-Entwicklung. Selbst die älteren, strukturierten Programmiersprachen (wie C oder Pascal) bieten zu diesem Zweck Arrays und benutzerdefinierte Datenstrukturen an (siehe Abschnitt 5.1.2). In C# können Datencontainer über Arrays (vgl. Abschnitt 6.2), Klassen (siehe Kapitel 5) oder Strukturen (vgl. Abschnitt 6.1) realisiert werden. Seit C# 7.0 ist mit den als *Strukturen realisierten Tupeln* eine neue Option mit Vorteilen gegenüber den traditionellen Lösungen verfügbar.[1]

Arrays haben den Nachteil, dass alle Elemente vom selben Typ sein müssen. Klassen und Arrays haben als Referenztypen den Nachteil, dass bei ihrer Verwendung Objekte auf dem Heap entstehen, sodass bei einer großen Anzahl von Objekten Performanzprobleme entstehen können. Strukturen verursachen keinen Heap-Aufwand, doch haben sie (wie Klassen) die folgenden Nachteile:

- Es entsteht Aufwand durch die Notwendigkeit einer Definition.
- Strukturen und Klassen sind dazu vorgesehen, Daten *und* Handlungskompetenzen zu kombinieren). Ihre Verwendung als pure Datencontainer erschwert ein schnelles Verständnis der Programmlogik.

Die mit C# 7.0 eingeführten Struktur-Tupel ermöglichen die syntaktisch einfache Kreation einer Datensammlung mit Wertsemantik. Dies ist besonders nützlich bei Methoden, die mehr als einen Wert zurückliefern. Wir betrachten als Beispiel eine Methode namens `AnalyzeName()`, die bei ihrem **String**-Parameter (in einer nicht praxistauglichen Vereinfachung) davon ausgeht, dass ein Vor-

[1] In der BCL gibt es seit längerer Zeit als *Klassen* realisierte Tupel, die aber mittlerweile nur noch selten empfohlen werden (siehe Abschnitt 6.6.1). Wenn im Abschnitt 6.6 von *Tupeln* die Rede ist, dann sind die neuen *Struktur-Tupel* gemeint, sofern es nicht ausdrücklich um *Objekt-Tupel* geht.

und ein Nachname durch ein Leerzeichen getrennt enthalten sind. Als Rückgabe erhält der Aufrufer:

- die beiden Namensbestandteile,
- jeweils eine Beurteilung, ob es sich um ein Palindrom handelt.[1]
  Ein Palindrom besitzt in beiden Leserichtungen dieselbe Buchstabensequenz, z. B. Reittier.

Die Methode liefert die Bestandteile einzeln ab, sodass sie vom Aufrufer bequem verarbeitet werden können (siehe Abschnitt 6.6.4):

```
static (String First, String Last, bool FistPalin, bool LastPalin) AnalyzeName(String name) {
    int posSpace = name.IndexOf(" ");
    int lenLast = name.Length - (posSpace + 1);
    string fn = name.Substring(0, posSpace);
    string ln = name.Substring(posSpace + 1, lenLast);
    bool CheckPalin(string ins) {
        int len = ins.Length;
        var sb = new StringBuilder(len);
        for (int i = 0; i < len; i++)
            sb.Append(ins[len - i - 1]);
        return sb.ToString().ToUpper() == ins.ToUpper();
    }
    return (fn, ln, CheckPalin(fn), CheckPalin(ln));
}
```

Hier wird ohne nennenswerten syntaktischen Aufwand für die Rückgabe der Tupeltyp

```
(String First, String Last, bool FirstPalin, bool LastPalin)
```

definiert, der vier öffentlich zugängliche Felder mit unterschiedlichen Datentypen enthält.

Tupel eignen sich als leichtgewichtige Datencontainer:

- Es entsteht kein Heap-Aufwand, weil die Tupel als Werttypen realisiert sind.
- Die Tupeltypen können mit einer bequemen Syntax definiert werden.
- Tupeltypen besitzen nur rudimentäre, nicht erweiterbare Handlungskompetenzen, z. B.:
  - die Methoden **CompareTo()**, **Equals()** und **ToString()**)
  - die Eigenschaft **Length**

  Seit C# 7.3 sind Identitätsvergleiche mit Hilfe der Operatoren == und **!=** möglich.

Weil ein Tupeltyp seine Daten in Feldern mit der Schutzstufe **public** verwaltet, sind seine Instanzen veränderbar.

Bequemlichkeit, Typsicherheit und Performanz sprechen für die Verwendung der Tupeltypen, sodass sie für den Klassen- oder Struktur-internen Einsatz sehr gut geeignet sind. Als Rückgabetypen für *öffentliche* Methoden werden die Tupeltypen von Microsoft jedoch *nicht* empfohlen:[2]

> In public APIs, consider defining a class or a structure type.

Während die Tupel für den erfahrenen C# - Entwickler eine willkommene Detailverbesserung darstellen, kann die erneute Erweiterung des Typenarsenals von Einsteigern als Informationsüberladung empfunden werden. Einsteigern wird daher empfohlen, sich vorläufig auf die im Abschnitt 6.6.4 beschriebene Verwendung von Tupeln als Rückgabetypen von Methoden zu beschränken. Diese Einsatzart ist der primäre Zweck von Tupeln (Albahari & Johannsen 2020, S. 197) und wird spontan als Verbesserung gegenüber alternativen Optionen (explizite Definition eines zusammengesetzten Rückgabetyps, **out**-Parameter) empfunden.

---

1 https://de.wikipedia.org/wiki/Palindrom

2 https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples

6.6.1 Tupel im CTS (Common Type System)

Das CTS (Common Type System) enthält zur Unterstützung der Tupeltypen acht generische Strukturen im Namensraum **System**:

- **public struct ValueTuple<T1>**
- **public struct ValueTuple<T1, T2>**
  . . .
- **public struct ValueTuple<T1, T2, T3, T4, T5, T6, T7>**
- **public struct ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>**

Da wir uns noch nicht mit generischen Typen beschäftigt haben (siehe Kapitel 8), sollten Sie sich mit den durch spitze Klammern begrenzten Typformalparametern jetzt noch nicht beschäftigen. Eine Konkretisierung für den vier Felder enthaltenden Typ **ValueTuple<T1, T2, T3, T4>** ist der zu Beginn des Abschnitts 6.6 als Rückgabe der Methode `AnalyzeName()` verwendete Typ:

```
(String First, String Last, bool FirstPalin, bool LastPalin)
```

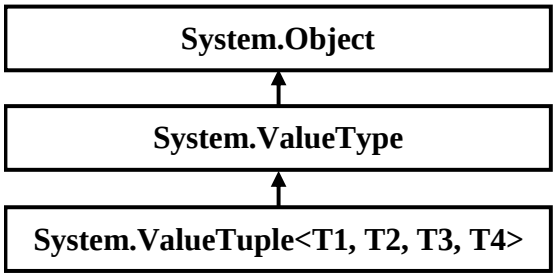
Die durch das Streichen der Feldnamen entstehende Tupel-Typbezeichnung

(String, String, bool, bool)

ist eigentlich nur ein Aliasname für:

ValueTupel<String, String, bool, bool>

Hier ist der Stammbaum von **ValueTuple<T1, T2, T3, T4>** zu sehen:[1]



Die Realisation der Tupel als Strukturen hat wichtige Konsequenzen:

- Wird einer Tupel-Instanz `tia` eine andere Tupel-Instanz `tib` zugewiesen, dann erhält `tia` nicht die Adresse von `tib` , sondern eine vollständige Kopie aller Werte.
- Beim Vergleich von zwei Tupel-Instanzen durch die **Equals()** - Methode findet kein Adressvergleich statt, sondern ein Inhaltsvergleich.

Weitere Eigenschaften der Struktur-Tupel - Typen:[2]

- Die Schutzstufe der Typen ist **public**.
- Alle Felder der Struktur-Tupel - Typen haben ebenfalls die Schutzstufe **public**.
- Die Instanzen der Struktur-Tupel - Typen sind veränderlich, während z. B. die Objekte der anonymen Klassen und auch die Objekte der im Anschluss erwähnten Tupel-Klassen unveränderlich sind.
- Die Felder können über ihre optional bei der Definition vergebenen Ergänzungsnamen oder über die Standardnamen **Item1**, **Item2**, usw. angesprochen werden.
- Wie bei anderen Strukturen ist *keine* Vererbung möglich (siehe Abschnitt 6.1).

Es muss noch erwähnt werden, dass in der BCL auch Tupel-*Klassen* enthalten sind (Namensraum **System**):

---

[1] https://learn.microsoft.com/en-us/dotnet/api/system.valuetuple-4

[2] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples

- **public class Tuple<T1>**
- **public class Tuple <T1, T2>**
  . . .
- **public class Tuple <T1, T2, T3, T4, T5, T6, T7>**
- **public class Tuple <T1, T2, T3, T4, T5, T6, T7, TRest>**

Diese haben im Vergleich zu den Struktur-Tupeln folgende Nachteile:[1]

- Es handelt sich um Referenztypen, was sich zumindest bei Verwendung einer größeren Anzahl negativ auf die Performanz auswirken kann.
- Ihre Member können nur über die Standardnamen **Item1**, **Item2**, etc. angesprochen werden, was die Lesbarkeit des Quellcodes erschwert.
- Es fehlt die in C# 7.0 für die Wert-Tupel eingeführte bequeme Syntax.

Die Objekt-Tupel sind unveränderbar, was *nicht* als Nachteil zu werten ist. Microsoft hat sich bei der Erneuerung des Tupel-Konzepts aber für *veränderbare* Strukturen entschieden.

6.6.2 Variablen mit Tupeltyp deklarieren

Im folgenden Beispiel deklarieren wir Variablen mit dem Tupeltyp (**String**, **String**, **int**) und verzichten dabei auf die Benennung der Felder. Die Fähigkeiten des Compilers zur Typinferenz ausnutzend kann man in einer Methode die Typangabe durch das Schlüsselwort **var** ersetzen:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  (string, string, int) p1 = ("Elena", "Müller", 57);`
`  var p2 = ("Otto", "Meyer", 34);`
`  Console.WriteLine(p2.Item1);`
`  }`
`}`
	`Otto`

Um beim Feldzugriff nicht auf die Standardnamen **Item1**, **Item2**, usw. angewiesen zu sein, sollten für Tupeltypen explizite Feldnamen vereinbart werden, z. B.:

Quellcode	Ausgabe

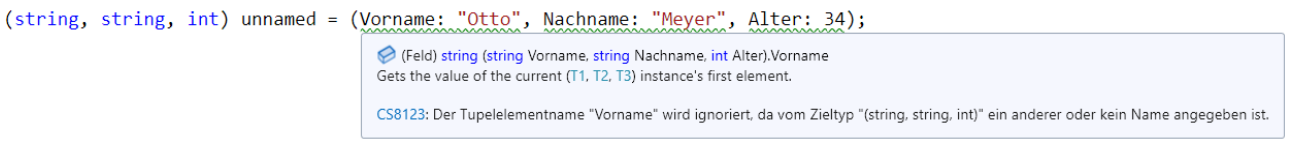
`(string Vorname, string Nachname, int Alter) named = ("Elena", "Müller", 57);`
`(string, string, int) unnamed = (Vorname: "Otto", Nachname: "Meyer", Alter:34);`
 	
`Console.WriteLine(named.Vorname);`
`Console.WriteLine(unnamed.Item2);`
	`Elena`
	`Meyer`

Für die Variable `named` wird ein Tupeltyp mit benannten Feldern explizit vereinbart. Auf der rechten Seite der Zuweisung steht ein Tupeltyp mit unbenannten Feldern und Werten mit passenden Typen, sodass die Zuweisung gelingt. Es resultiert eine Variable mit den deklarierten Feldnamen und den zugewiesenen Werten.

Für die Variable `unnamed` wird ein Tupeltyp vereinbart mit Feldern, die einen expliziten Typ erhalten, aber keinen Namen. Durch die im Tupel-Literal auf der rechten Seite der Zuweisung vorhandenen Feldnamen wird der unbenannte Zustand aus der Variablendeklaration von `unnamed` *nicht* geändert:

---

[1] https://learn.microsoft.com/en-us/dotnet/api/system.tuple



Ersetzt man die Tupel-Definition mit expliziten Typangaben durch das Schlüsselwort **var** (möglich innerhalb von Methoden), sodass der Compiler zur Typinferenz gezwungen wird, dann kommt es zur Übernahme der Feldnamen, z. B.:

```
var named2 = (Vorname: "Otto", Nachname: "Meyer", Alter: 34);
Console.WriteLine(named2.Vorname);
```

Weil die Felder eines Struktur-Tupel – Typs die Schutzstufe **public** besitzen, sollte für ihre optionalen Ergänzungsnamen das Pascal Casing verwendet werden (mit einem Großbuchstaben am Anfang, vgl. Abschnitt 5.2.2). Die Standardnamen **Item1**, **Item2**, usw. folgen dieser Konvention.

Wird die seit C# 7.1 bestehende Option genutzt, in einer Methode einen Ergänzungsnamen für eine Tupeltyp-Instanz mit einer durch das Schlüsselwort **var** vertretenen Typangabe von einer bereits deklarierten Variablen elementaren Typs zu übernehmen, dann endet die Benennungskonsistenz. Der mit einem kleinen Anfangsbuchstaben startende Name der lokalen Variablen wird dann zum Namen eines Felds mit Schutzstufe **public**, z. B.:[1]

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int alter = 34;`
`  var named = (Vorname: "Otto", Nachname: "Meyer", alter);`
`  Console.WriteLine(named.alter);`
`  var named2 = (Vorname: "Otto", Nachname: "Meyer", Alter: alter);`
`  Console.WriteLine(named2.Alter);`
`  named2 = named;`
`  }`
`}`
	`34`
	`34`

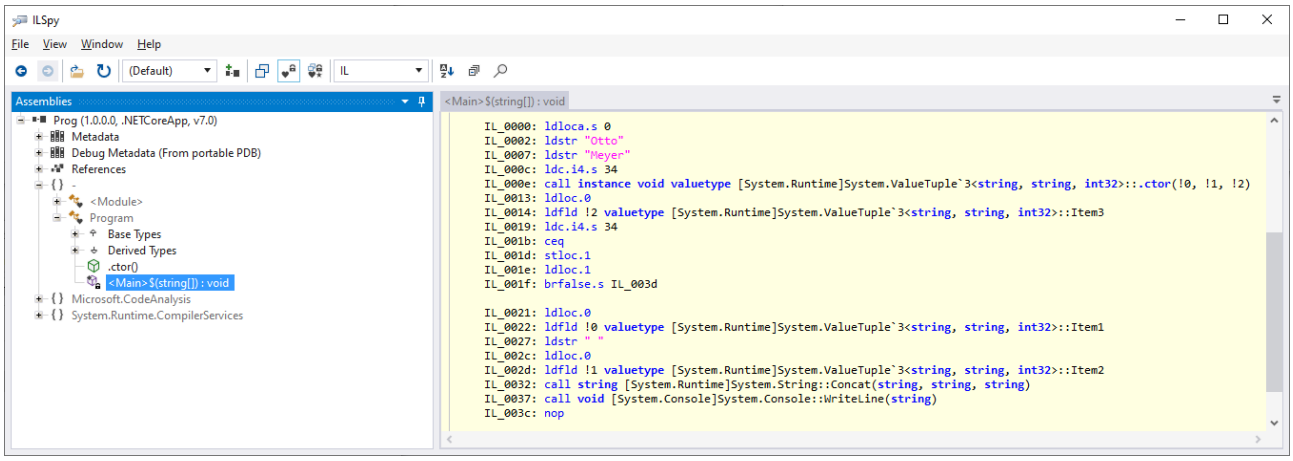
Es ist allerdings möglich, den übernehmbaren Variablennamen durch einen explizit definierten Feldnamen zu dominieren (siehe Variable `named2`).

Die beiden Tupel-Instanzen bleiben trotz der abweichenden Feldnamen zuweisungskompatibel (siehe Abschnitt 6.6.3), und im IL-Code existieren ohnehin nur die Standardnamen der Felder. Um die zweite Aussage zu verifizieren, betrachten wir zum folgenden Beispiel

```
var person = (Vorname: "Otto", Nachname: "Meyer", Alter: 34);
if (person.Alter == 34)
    Console.WriteLine(person.Vorname + " " + person.Nachname);
```

den IL-Code mit ILSpy:

[1] Das Bemühen um einen dank konsistenter Benennungen gut lesbaren Quellcode droht im konkreten Fall, in übertriebene Bürokratie abzudriften.



Am Ende des Abschnitts stehen Vergleiche der Struktur-Tupel mit den anonymen Klassen, die ebenfalls spontan (ohne explizite Typdefinition) als Container für Daten mit unterschiedlichen Typen verwendet werden können (siehe Abschnitt 6.5). Leider sind dabei Vorgriffe auf das Kapitel 8 über das typgenerische Programmieren erforderlich, sodass Einsteiger beim ersten Lesen vermutlich eher Ärger erleben als Lerngewinn:

- Die höhere Funktionalität der Struktur-Tupel zeigt sich u. a. in der Möglichkeit zur Konkretisierung von Typformalparametern, sodass die zahlreichen generischen Typen in der BCL und in anderen Bibliotheken nutzbar sind, z. B. für eine Liste mit Struktur-Tupel - Instanzen:

  ```
  List<(string Name, int Alter)> mitglieder;
  ```

- Im Quellcode vorhandene anonyme Klassen werden vom C# - Compiler in neu definierte IL-Klassen übersetzt, wobei die Eigenschafts- bzw. Feldnamen aus dem Quellcode Verwendung finden. Demgegenüber resultieren aus den Tupeltypen konkretisierte generische Strukturen, und diese enthalten im IL-Code nur die Standardfeldnamen **Item1**, **Item2**, usw. Solange keine speziellen Programmiertechniken wie die Reflexion zum Einsatz kommen (siehe Kapitel 14), ist der Verlust der Ergänzungsfeldnamen für die Software-Entwicklung aber irrelevant.

6.6.3 Anweisungen mit Tupel-Dekonstruktion

Eine Tupeltyp-Instanz kann einer Tupeltyp-Variablen zugewiesen werden, wenn ...

- gleich viele Felder vorhanden sind,
- und alle Feldtypen auf der rechten Seite der Zuweisung erweiternd in den zugehörigen Typ auf der linken Seite konvertiert werden können.

Die optionalen Feldnamen spielen bei der Zuweisungskompatibilität keine Rolle.

Im folgenden Beispiel übernimmt die Variable `sint` vom rechten Teil der Zuweisung den Tupeltyp mit benannten Feldern, wobei das zweite Feld den erschlossenen Typ **int** hat:

```
var sint = (Vorname: "Otto", Alter: 34);
```

Im explizit definierten Tupeltyp der Variablen `slong` hat das zweite Feld den Typ **long**,

```
(string Vorname, long Alter) slong = ("Otto", 34);
```

sodass die folgende Zuweisung

```
sint = slong;
```

scheitert, während gegen die umgekehrte Zuweisung

```
slong = sint;
```

nichts einzuwenden ist.

Im nächsten Beispiel werden die beiden lokalen Variablen `ganz` und `gk` mit den elementaren Datentypen **int** bzw. **double** per Tupel-Syntax deklariert und initialisiert. Dasselbe passiert auch mit den Variablen `ganzTi` und `gkTi`, wobei zusätzlich über das Schlüsselwort **var** die Typinferenz genutzt wird:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  (int ganz, double gk) = (13, 47.11);`
`  Console.WriteLine(ganz);`
`  (var ganzTi, var gkTi) = (13, 47.11);`
`  Console.WriteLine(ganzTi);`
 	
`  int ivar;`
`  double dvar;`
`  (ivar, dvar) = (13, 47.11);`
`  Console.WriteLine(dvar);`
 	
`  // Seit C# 10 erlaubt: Wertzuweisung und Deklaration mit Initial.:`
`  (ivar, double ndvar) = (13, 47.11);`
`  }`
`}`
	`13`
	`13`
	`47,11`

Anschließend erhalten die bereits deklarierten Variablen `ivar` und `dvar` per Tupel-Syntax einen neuen Wert.

In allen Zuweisungen werden die beteiligten Tupel-Literale (z. B.: `(13, 47.11)`) zerlegt, und ihre Feldinhalte landen in einfachen lokalen Variablen. Man spricht hier von einer **Dekonstruktion**.

Bis C# 9 konnte in einer Anweisung mit Dekonstruktion für alle beteiligten Variablen *einheitlich* …

- entweder eine Deklaration mit Initialisierung für neue Variablen
- oder eine Wertzuweisung für vorhandene Variablen

stattfinden. Seit C# 10 ist auch ein kombiniertes Auftreten erlaubt (siehe letzte Anweisung im Beispiel).

Gelegentlich sind bei der Zerlegung einer Tupeltyp-Instanz nicht alle Felder von Interesse. Den Verzicht auf ein Feld signalisiert man durch die seit C# 7.0 zugelassene Ausschussvariable (engl.: *discard variable*), die wir im Zusammenhang mit **out**-Parametern kennengelernt haben (vgl. Abschnitt 5.3.1.3.2.2), z. B.:

```
(string first, string last, _) = ("Otto", "Meyer", 34);
```

6.6.4 Tupel als Rückgabetypen von Methoden

Wie zu Beginn von Abschnitt 6.6 erwähnt, ist die Verwendung von Tupeltypen für die Rückgabe von Methoden besonders attraktiv. Die bereits vorgestellte Methode `AnalyzeName()` geht bei ihrem **String**-Parameter (in einer nicht praxistauglichen Vereinfachung) davon aus, dass ein Vor- und ein Nachname durch ein Leerzeichen getrennt enthalten sind. Der Aufrufer erhält in einer Struktur-Tupel – Instanz vom Typ:[1]

(String, String, bool, bool)

[1] Das ist ein Aliasname für den Typ **ValueTupel<String, String, bool, bool>**, wobei es sich um eine Konkretisierung der generischen Struktur **ValueTuple<T1,T2,T3,T4>** handelt (siehe Kapitel 8).

für die beiden Namensbestandteile jeweils eine Zeichenfolge und die Information darüber, ob ein Palindrom vorliegt, z. B.:

Quellcode	Ausgabe

`using System;`
 	
`class ReturnTuple {`
`  static (string First, string Last, bool FirstPalin, bool LastPalin)`
`  AnalyzeName(string name) {`
`  int posSpace = name.IndexOf(" ");`
`  string fn = name[..posSpace];`
`  string ln = name[++posSpace..];`
 	
`  static bool CheckPalin(string ins) {`
`  char[] rev = ins.ToCharArray();`
`  Array.Reverse(rev);`
`  return new string(rev) == ins;`
`  }`
 	
`  return (fn, ln, CheckPalin(fn.ToUpper()), CheckPalin(ln.ToUpper()));`
`  }`
 	
`  static void Main() {`
`  var an = AnalyzeName("Otto Rentner");`
`  Console.WriteLine(an.First);`
`  var (_, _, _, lastPal) = AnalyzeName("Otto Rentner");`
`  Console.WriteLine(lastPal);`
`  }`
`}`
	`Otto`
	`True`

Für die lokale Methode `CheckPalin()`, die den Palindromtest erledigt, wird über den Modifikator **static** ein Zugriff auf lokale Variablen der umgebenden Methode (und auf Instanzvariablen) ausgeschlossen. Das verbessert die Performanz, reduziert das Fehlerrisiko und erleichtert ggf. die Fehlersuche (siehe Abschnitt 5.3.1.5).

In `CheckPalin()` wird mit Hilfe der statischen Methode **Reverse()** der Klasse **Array** eine invertierte Version der übergebenen Zeichenfolge erstellt.

Beim ersten Aufruf von `AnalyzeName()` wird die Rückgabe in der lokalen Variablen `an` mit dem Tupeltyp **(string, string, bool, bool)** abgelegt. Anschließend kann z. B. der Vorname über das öffentliche Feld `First` angesprochen werden. Im zweiten Aufruf von `AnalyzeName()` wird die Rückgabe durch die im Abschnitt 6.6.3 beschriebene Dekonstruktion auf einzelne lokale Variablen verteilt.

Verzichtet man in einer Methodendefinition mit Tupel-Rückgabetyp auf eine Benennung der Felder, dann müssen die Felder später über die Standardnamen **Item1**, **Item2**, usw. angesprochen werden, z. B.:

Quellcode	Ausgabe

`using System;`
`using System.Text;`
 	
`class ReturnTuple {`
`  static (string, string, bool, bool) AnalyzeName(string name) {`
`  . . .`
`  }`
 	
`  static void Main() {`
`  var an = AnalyzeName("Otto Rentner");`
`  Console.WriteLine(an.Item1);`
`  Console.WriteLine(AnalyzeName("Otto Rentner").Item3);`
`  }`
`}`
	`Otto`
	`True`

Man kann zwar auch mit **out**-Parametern (siehe Abschnitt 5.3.1.3.2.2) mehrere Werte an den Aufrufer übertragen, verliert dabei aber z. B. ...

- die Möglichkeit, die erhaltenen Werte als Felder einer Instanz gemeinsam zu verarbeiten,
- syntaktische Eleganz.

6.6.5 Dekonstruktion für selbst definierte Typen

Eine Möglichkeit zur bequemen Übertragung von Instanzeigenschaften und -feldern in einzelne Variablen nach dem Muster der im Abschnitt 6.6.3 beschriebenen Dekonstruktion kann man auch für selbst definierte Typen realisieren. Man definiert eine öffentliche Methode namens **Deconstruct()** (bei Bedarf auch in mehreren Überladungen) und verwendet dabei **out**-Parameter für die auszuliefernden Instanzeigenschaften und -felder. Im folgenden Beispiel wird eine datenzentrierte Klasse namens `Person` definiert:[1]

```
using System;

public class Person {
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string first, string last) {
        FirstName = first;
        LastName = last;
    }

    public void Deconstruct(out string firstName, out string lastName) {
        firstName = FirstName;
        lastName = LastName;
    }
}

class Prog {
    static void Main() {
        Person p = new("Otto", "Meyer");
        var (first, last) = p;
        Console.WriteLine(first);
    }
}
```

[1] Vorbild: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/functional/deconstruct

In der Startmethode des Programms werden die lokalen Variablen `first` und `last` in einer Anweisung mit implizierter Dekonstruktion deklariert und initialisiert. Als Ausgabe resultiert:

```
Otto
```

Der Finalisierer einer Klasse, der ggf. kurz vor der Beseitigung eines obsolet gewordenen Objekts vom Garbage Collector aufgerufen wird (siehe Abschnitt 5.4.5), wurde früher als *Destruktor* bezeichnet.[1] Trotz der Namensähnlichkeit zur **Deconstruct**-Methode besteht nicht die geringste begriffliche Verwandtschaft.

6.6.6 Überladene (Un)gleichheitsoperatoren

Seit C# 7.3 sind für Tupeltyp-Instanzen die grundsätzlich inhaltsorientierten Gleichheitsprüfungen nicht nur über die Methode **Equals()**, sondern auch mit Hilfe der überladenen Operatoren == und **!=** möglich, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var tti1 = (G: 13, D: 13.0);`
`  var tti2 = (13, 13.0);`
`  var tti3 = (13, 13);`
`  Console.WriteLine(tti1 == tti2);`
`  Console.WriteLine(tti2 == tti3);`
`  }`
`}`
	`True`
	`True`

Regeln:

- Die optionalen Ergänzungsfeldnamen spielen beim Vergleich keine Rolle.
- Bei Bedarf werden erweiternde Typanpassungen vorgenommen.

6.7 Record-Datentypen

Die in C# 9 eingeführten Record-Datentypen ...[2]

- vereinfachen die Definition von *unveränderlichen* Klassen,
- bieten eine *inhaltsorientierte* Gleichheitsprüfung durch synthetisierte (vom Compiler erstellte) Methoden
- und unterstützen die *Vererbung*.

In der Bezeichnung *Record* (dt.: *Datensatz*) steckt eine klare Anwendungsempfehlung für den neuen Referenzdatentyp im Kontrast zur traditionellen Klasse. Wir haben uns seit dem Kapitel 1 mit der Vorstellung vertraut gemacht, dass Objekte über Kompetenzen, einen in ständiger Wandlung begriffenen Zustand und einen Verantwortungsbereich besitzen. Dies ist eine grundlegende Vorstellung im objektorientierten Programmier-Paradigma. Allerdings befinden sich im Aufgabenbereich mancher Programme auch Objekte, die eher als Datensatz denn als aktives Individuum zu beschreiben sind. Wenn dann ....

---

1 https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/finalizers

2 https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record

- Objekte (mit Heap-orientierter Speicherverwaltung und Vererbung) gefragt sind,
- die Unveränderlichkeit der Objekte auf einfache Weise sichergestellt werden soll,
- und außerdem eine inhaltsorientierte Gleichheitsprüfung gewünscht ist,

dann sind Record-Datentypen eine gute Lösung. Unveränderliche Objekte sind z. B. in Programmen mit mehreren Threads (Ausführungsfäden) und gemeinsam genutzten Daten von Vorteil, weil die Thread-Synchronisation entfällt.

Die mit C# 9 einführten Record-*Klassen* wurden schon in C# 10 durch Record-*Strukturen* ergänzt:

- Dass bei Strukturen keine Vererbung möglich ist, spielt für viele Programme keine Rolle.
- Als Member von Objekten oder Elemente von Arrays lassen sich Strukturinstanzen auch auf dem Heap stationieren.

Zugunsten einer übersichtlichen Darstellung werden im weiteren Verlauf des Abschnitts 6.7 zunächst die Record-Klassen vorgestellt. Mit den Besonderheiten der Record-Strukturen beschäftigt sich der Abschnitt 6.7.5.

6.7.1 Definition

Ein Record-Datentyp besteht im Wesentlichen aus get-only – und/oder init-only – Eigenschaften, die nach der Initialisierung per Konstruktor oder Objektinitialisierer nicht mehr geändert werden können. Wenn diese Eigenschaften auf traditionelle Weise deklariert werden, dann unterscheidet sich eine Datensatzdefinition von einer analogen Klassendefinition nur durch die Verwendung des Schlüsselworts **record** statt **class**. Im folgenden Beispiel kommen get-only – Eigenschaften zum Einsatz, sodass ein Konstruktur definiert werden muss, um eine Initialisierung der Objekte zu ermöglichen:

```
public record Person {
   public string LastName { get; }
   public string FirstName { get; }

   public Person(string first, string last) {
      FirstName = first;
      LastName = last;
   }
}
```

Seit C# 10 kann statt des Schlüsselworts **record** das Schlüsselwortpaar **record class** verwendet werden zwecks konsistenter und deutlicher Abgrenzung vom nunmehr zugelassenen Schlüsselwortpaar **record struct** (siehe Abschnitt 6.7.5), z. B.:

```
public record class Person {
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
}
```

Bei der im letzten Beispiel demonstrierten Verwendung von init-only – Eigenschaften kann auf einen Konstruktor verzichtet und ein Objektinitialisierer verwendet werden, z. B.:

```
var p = new Person() { FirstName = "Otto", LastName = "Brgl" };
```

Im nächsten Beispiel sind zwei get-only – Eigenschaften und eine init-only – Eigenschaft vorhanden:

```
public record Person {
    public string LastName { get; }
    public string FirstName { get; }
    public int YearOfBirth { get; init; }

    public Person(string first, string last) {
        FirstName = first;
        LastName = last;
    }
}
```

Für zwei mit bzw. ohne Initialisierer erstellte Record-Objekte

```
var p1 = new Person("Luise", "Schmit") { YearOfBirth = 2002 };
var p2 = new Person("Otto", "Rempremerding");
Console.WriteLine(p1);
Console.WriteLine(p2);
```

ist anschließend das Ergebnis der vom Compiler für den Record-Typ `Person` automatisch erstellten **ToString()** - Überschreibung zu sehen, die von **WriteLine()** aufgerufen wird:

```
Person { LastName = Schmit, FirstName = Luise, YearOfBirth = 2002 }
Person { LastName = Rempremerding, FirstName = Otto, YearOfBirth = 0 }
```

Gegenüber den bisher vorgeführten Record-Definitionsvarianten mit expliziten Eigenschaftsdefinitionen lässt sich mit der *positionalen* Syntax zur Eigenschaftsdefinition viel Aufwand sparen. Eine solche Record-Definition ähnelt dem Definitionskopf eines parametrisierten Konstruktors, z. B.:

```
public record Person(string FirstName, string LastName);
```

Aufgrund einer Record-Definition mit Konstruktor-Syntax erstellt der Compiler automatisch …

- eine init-only - Eigenschaft (Bezeichnung: *positionale Eigenschaft*) für jeden Parameter in der Definition,
- den sogenannten *primären Konstruktor* (mit einem Parameter für jede positionale Eigenschaft),
- eine **Deconstruct()** – Methode (vgl. Abschnitt 6.6.5).

Im Abschnitt 6.7.2 werden weitere Methoden und Operator-Überladungen beschrieben, die der Compiler für *alle* Record-Typen erstellt.

Durch den Verzicht auf die explizite Eigenschaftsdefinition geht die Validierungsmöglichkeit in den **set**-Blöcken verloren. Andere Änderungen bzw. Ergänzungen der Klassendefinition sind aber möglich. Im folgenden Beispiel wird die (von **Object** geerbte) **ToString()** – Definition überschrieben, sodass der Compiler auf seine Kreation verzichtet:[1]

```
public record Person(string FirstName, string LastName) {
    public override string ToString() => $"Name: {FirstName} {LastName}";
}
```

Es ist möglich, positionale mit explizit definierten Eigenschaften zu kombinieren, z. B.:

```
public record Student(string Id, string FirstName, string LastName) {
    public string Id { get; } = Id;
}
```

Im Beispiel werden zwei implizite init-only – Eigenschaften ergänzt durch eine explizite get-only – Eigenschaft, die auf diese Weise von der nicht-destruktiven Mutation ausgenommen wird (siehe Abschnitt 6.7.3). In der expliziten Eigenschaftsdefinition ist eine Initialisierung unter Verwendung

---

[1] Wer die Definition der **ToString()** – Methode merkwürdig findet, möge sich im Abschnitt 5.8.1 über die Methodendefinition per Lambda-Symbol und Ausdruck sowie nötigenfalls im Abschnitt 4.2.2.2 über die Zeichenfolgeninterpolation informieren.

des zugehörigen Parameters aus dem primären Konstruktor (im Beispiel: `Id`) erforderlich (Albahari 2022, S. 220).

Ein Record-Typ lässt sich auch mit *änderbaren* Eigenschaften ausstatten, z. B.:

```
public record Person {
    public string LastName { get; }
    public string FirstName { get; }
    public int NoOfVisits { get; set; }

    public Person(string first, string last, int nov) {
        FirstName = first;
        LastName = last;
        NoOfVisits = nov;
    }
}
```

Da Records eingeführt wurden, um die Erstellung von *unveränderlichen* Objekten zu erleichtern, sind änderbare Eigenschaften sowie Methoden zur Änderung von Record-Objekten allerdings nur in begründeten Ausnahmefällen sinnvoll.

6.7.2 Vom Compiler erstellte Methoden

Der Compiler erstellt zu einem Record-Typ mehrere Methoden, die zur inhaltsbasierten Gleichheitsprüfung und für andere Zwecke geeignet sind, wobei im Quellcode vorhandene, Signatur-gleiche Methoden *nicht* ersetzt werden:

- Eine Überschreibung der **Object**-Methode **Equals()**:

  public override bool Equals(object *obj***)**

  Zwei Variablen mit demselben Record-Typ werden von der Methode **Equals()** und auch vom überladenen Operator == (siehe unten) als gleich beurteilt, wenn alle Felder der referenzierten Objekte (inkl. der backing fields zu den automatisch implementierten Eigenschaften) denselben Wert besitzen. Für zwei Objekte von einem Record-Typ wird also eine *inhaltsbasierte* Gleichheitsprüfung durchgeführt.

- Eine **Equals()** - Überladung mit einem Parameter vom eigenen Typ, die z. B. beim Record-Typ `Person` den folgenden Definitionskopf besitzt:

  public virtual bool Equals(`Person` *other***)**

  Weil ein Record-Typ eine solche Methode besitzt, kann er von sich behaupten, die generische Schnittstelle **System.IEquatable<T>** zu implementieren (siehe Kapitel 8 zu generischen Typen und Kapitel 9 zu Schnittstellen).

- Eine Überschreibung der **Object**-Methode **GetHashCode()**:

  public override int GetHashCode()

  Die Methode **GetHashCode()** ist z. B. relevant beim Befüllen eines Kollektionsobjekts vom Typ **HashSet<T>** (siehe Abschnitt 11.4.2). Sie muss mit der **Equals()** - Methode kompatibel sein. Sind z. B. zwei Objekte gleich im Sinne der **Equals()** - Methode, dann müssen sie bei einem **GetHashCode()** - Aufruf denselben Wert liefern.

- Auf der **Equals()** - Methode basierende Überladungen für die Operatoren == und **!=** (vgl. Abschnitt 5.8.3).

- Eine Überschreibung der **Object**-Methode **ToString()**, die für eine informative Selbstdarstellung sorgt (siehe Beispiele im Abschnitt 6.7.1):

  public override String ToString()

Aufgrund einer Record-Definition mit Konstruktor-Syntax (mit positionalen Eigenschaften) wie im folgenden Beispiel (vgl. Abschnitt 6.7.1)

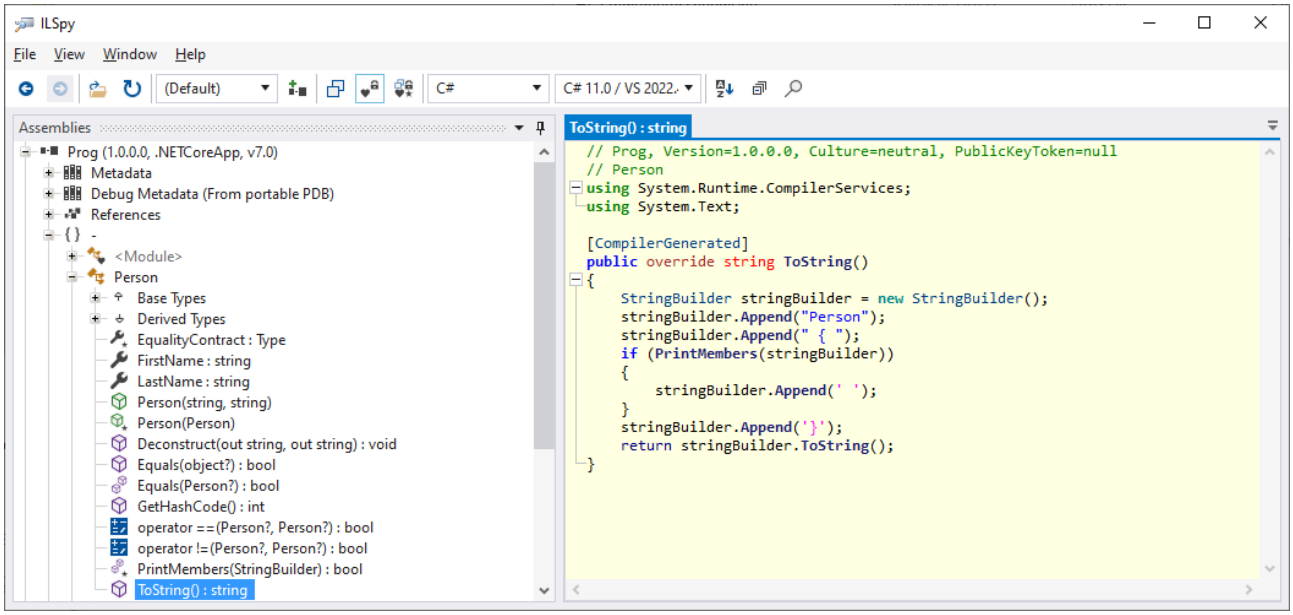
```
public record Person(string FirstName, string LastName);
```

erstellt der Compiler über die oben erwähnten Methoden hinaus:

- den primären Konstruktor (mit einem Parameter für jede Eigenschaft)
- eine **Deconstruct()** - Methode (vgl. Abschnitt 6.6.5), die einen **out**-Parameter für jede öffentliche Eigenschaft des Record-Typs besitzt, z. B.:

```
Person p = new Person("Otto", "Rempremerding");
var (first, last) = p;
```

Wie eine Inspektion mit ILSpy zeigt, entsteht im Beispiel aus der einzeiligen Record-Definition eine Klasse mit zahlreichen Mitgliedern:



6.7.3 Kopierkonstruktoren und with-Ausdrücke

Mit den Record-Typen ist es möglich, unveränderliche, datenzentrierte Objekte mit geringem Aufwand zu erstellen. Wenn für ein unveränderliches Record-Objekt aber trotzdem einzelne Eigenschaften modifiziert werden müssen, dann bleibt nur der Weg, …

- ein neues Objekt als Kopie des „zu verändernden“ Originals zu erstellen,
- und die benötigten Wertanpassungen vorzunehmen.

Zur Vereinfachung dieses Vorgangs hat Microsoft die sogenannte *nicht-destruktive Mutation* ersonnen. Der Compiler erstellt zu jeder Record-Klasse einen sogenannten *Kopierkonstruktor* (engl. *copy constructor*), der einen Parameter vom eigenen Typ besitzt und eine sogenannte *flache Kopie* des Parameterobjekts liefert. Die Kopie übernimmt vom Parameterobjekt die Werte aller Felder (öffentlich und privat, inklusive der backing fields zu automatisch implementierten Eigenschaften). Man bezeichnet die Kopie als *flach*, weil bei Eigenschaften mit Referenztyp *keine* Kopie des referenzierten Objekts entsteht. Solange die Eigenschaften einer Record-Klasse einen Werttyp oder einen unveränderlichen Referenztyp (z. B. **String**) besitzen, ist die flache Kopie mit der tiefen Kopie äquivalent.

Im Fall einer versiegelten (nicht beerbbaren) Record-Klasse (siehe Kapitel 7) wie im folgenden Beispiel

```
public sealed record Person(string FirstName, string LastName)
```

entsteht ein Kopierkonstruktor mit der Schutzstufe **private**:

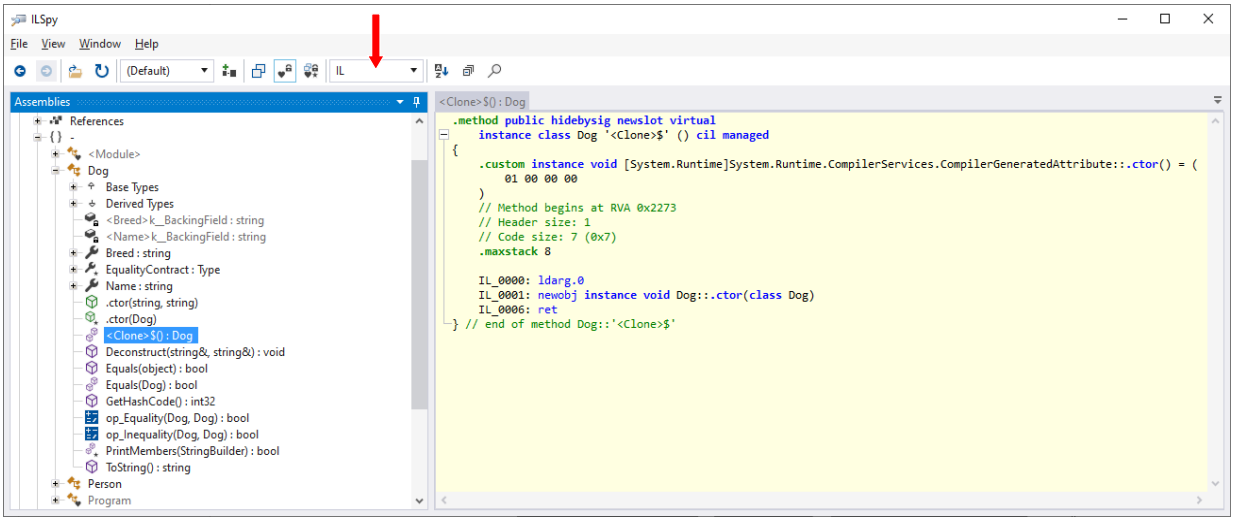
```
private Person(Person original) {
    FirstName = original.FirstName;
    LastName = original.LastName;
}
```

Bei nicht-versiegelten Klassen wie in den folgenden Beispielen

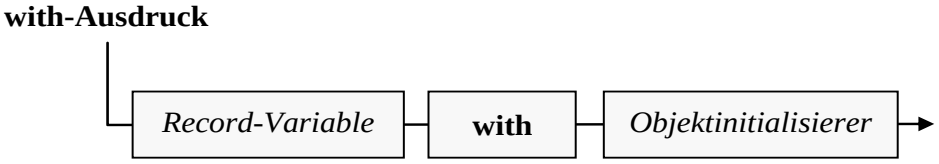
```
public record Dog(string Name, string Breed);
public record Person(string FirstName, string LastName, Dog Dog);
```

haben die automatisch erstellten Kopierkonstruktoren die Schutzstufe **protected**.

Zur Verwendung des Kopierkonstruktors durch fremde Klassen besitzt eine Record-Klasse eine vom Compiler automatisch erstellte öffentliche Methode mit dem ungewöhnlichen Namen **<Clone>$()**, die den Kopierkonstruktor aufruft. Weil der Methodenname in C# verboten, in der IL aber erlaubt ist, kann die Methode vom Compiler benutzt, im Quellcode aber *nicht* verwendet werden. In ILSpy ist die Methode nur dann zu sehen, wenn IL als Sprache eingestellt ist, z. B.:



Um einer fremden Klasse die nicht-destruktive Mutation unter Verwendung des Kopierkonstruktors und der Methode **<Clone>$()** zu erlauben, bietet C# seit der Version 9 den **with**-Ausdruck:



Es wird per Kopierkonstruktor ein neues Record-Objekt erstellt und anschließend per Objektinitialisierer (siehe Abschnitt 5.4.3.2) modifiziert. Im folgenden Beispiel wird vom **Record**-Objekt `p1` eine Kopie erstellt, wobei mit Ausnahme des Nachnamens alle Eigenschaftsausprägungen erhalten bleiben:

```
var p1 = new Person("Otto", "Rempremerding", new Dog("Emma", "Mix"));
var p2 = p1 with { LastName = "Brgl" };
```

Im Objektinitialisierer können auch *mehrere* Eigenschaften einen neuen Wert erhalten.

Es ist erlaubt und oft sinnvoll, *keine* zu verändernde Eigenschaft anzugeben, z. B.:

```
var p3 = p1 with { };
```

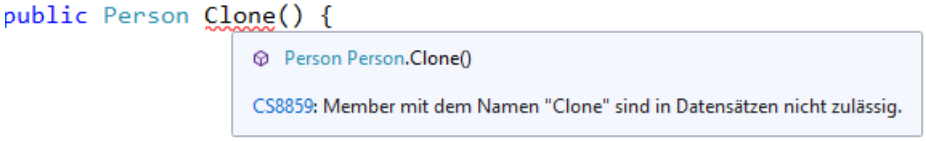
Dies ermöglicht die implizite Nutzung des vom Compiler automatisch erstellten Kopierkonstruktors, der aufgrund seines Zugriffsmodifikators **protected** oder **private** ansonsten in fremden Klassen nicht verfügbar ist.

Soll für eine Eigenschaft die Änderung per nicht-destruktiver Mutation ausgeschlossen sein, dann definiert man sie explizit als get-only (vgl. Abschnitt 6.7.1), z. B.:

```
var s1 = new Student(123, "Otto", "Rempremerding");
var s2 = s1 with { Id = 12 };

record Student(int Id, string FirstName, string LastName) {
    public int Id { get; } = Id;
}
```

Der Kopierkonstruktor übernimmt die get-only – Eigenschaftsausprägung in das neue Record-Objekt, weil er *alle* Felder unabhängig von der Schutzstufe verarbeitet. Der Objektinitialisierer hält sich jedoch an die deklarierte Schutzstufe, z. B.:

Während in einer Record-Klasse keine Methode namens `Clone()` erlaubt ist,



darf man den Kopierkonstruktor selbst implementieren, z. B.:

```
public record Person(string FirstName, string LastName, Dog Dog) {
    public Person(Person p) {
        FirstName = p.FirstName;
        LastName = p.LastName;
        Dog = p.Dog;
    }
}
```

Microsoft empfiehlt, analog zum Verhalten des Compilers bei der automatischen Erstellung eines Kopierkonstruktors die Schutzstufe **protected** bzw. **private** zu verwenden:[1]

> If you need different copying behavior, you can write your own copy constructor in a record class. If you do that, the compiler doesn't synthesize one. Make your constructor **private** if the record is **sealed**, otherwise make it **protected**.

Eine Begründung für diese Empfehlung bleibt Microsoft schuldig. Wer sich vorsichtshalber an die Vorgabe hält, hat dadurch in der Regel keine Nachteile. Wenn aber doch ein öffentlich verfügbarer Kopierkonstruktor benötigt wird, dann muss die Empfehlung nicht befolgt werden, denn:

- Offenbar stammt die Empfehlung aus der Programmiersprache C++, wird aber nicht von allen Vertretern dieser Sprache als sinnvoll erachtet:[2]

  > Though this inhibition of copy construction is probably the earliest example of a private constructor that a programmer will meet, it is also probably the least useful when used in isolation.

---

1 https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record#nondestructive-mutation

2 https://accu.org/journals/overload/1/3/glassborow_1390/

- Microsoft sieht *kein* generelles Problem darin, für eine C# - Klasse einen öffentlichen Kopierkonstruktor zu definieren, sondern leitet sogar dazu an.[1] Eine Record-Klasse unterscheidet sich von einer normalen Klasse aber nur durch die besondere Unterstützung durch den Compiler.
- Wie oben am Beispiel
  ```
  var p3 = p1 with { };
  ```
  zu sehen war, ist ein geschützter Kopierkonstruktor per **with**-Ausdruck durch fremde Klassen leicht zu verwenden.

6.7.4 Vererbung

Bei der Beschreibung der Besonderheiten von Record-Typen im Zusammenhang mit der Vererbung werden einige, vermutlich leicht verdauliche Vorgriffen auf das Kapitel 7 in Kauf genommen:

- Wird in einer Record-Definition kein Basistyp angegeben, dann stammt der neue Typ von der Urahnklasse **Object** ab.
- Alternativ kann man eine Record-Klasse aus einer vorhandenen Record-Klasse ableiten und zusätzliche Eigenschaften ergänzen, z. B.:
  ```
  public record Person(string FirstName, string LastName);
  public record Lehrer(string FirstName, string LastName, string Subject) :
      Person(FirstName, LastName);
  ```
  Als Basisklasse für eine Record-Klasse sind nur **Object** oder eine andere Record-Klasse erlaubt.
- Aus einer Record-Klasse können nur andere Record-Klassen abgeleitet werden.
- Mit dem Modifikator **sealed** lässt sich für einen Record-Typ (und auch für jede andere Klasse) die Definition von Ableitungen verhindern, z. B.:
  ```
  public sealed record Student : Person { ... }
  ```
- Seit C# 10 kann man für die in einer Record-Klasse überschriebene **ToString()** – Methode mit dem Modifikator **sealed** das Überschreiben in einer abgeleiteten Record-Klasse verhindern, z. B.:
  ```
  public record Person {
    public string LastName { get; }
    public string FirstName { get; }

    public Person(string first, string last) {
        FirstName = first; LastName = last;
    }

    public sealed override string ToString() => $"Name: {FirstName} {LastName}";
  }
  ```
  An das Verbot hält sich auch der Compiler, indem er in abgeleiteten Klassen auf die automatische Erstellung einer **ToString()** – Überladung verzichtet.
- Sind zwei Ableitungen einer Record-Basisklasse im Spiel, dann werden Objekte als gleich beurteilt, wenn …
  - die Laufzeittypen identisch sind,
  - und alle Felder (inkl. der backing fields zu den automatisch implementierten Eigenschaften) denselben Wert besitzen.

Das Implementieren von Schnittstellen ist einem Record-Typ uneingeschränkt erlaubt.

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/how-to-write-a-copy-constructor

6.7.5 Record-Typen mit Wertsemantik

In C# 10 wurden die Record-Klassen durch analog arbeitende Record-Strukturen ergänzt. Trotz der gemeinsamen Datensatz-Logik erzwingt der Übergang von einer Klasse zu einer Struktur einige Unterschiede, die in diesem Abschnitt erläutert werden.

Eine Variable mit Record-Strukturtyp nimmt keine Objektadresse auf, sondern eine komplette Strukturinstanz mit allen Feldern.

Im Definitionskopf wird das Schlüsselwort **record** ergänzt durch das Schlüsselwort **struct**. Die Eigenschaften einer Record-Struktur können wie die Eigenschaften einer Record-Klasse positional (in einer Definition mit Konstruktor-Syntax) oder explizit deklariert werden.

Aus der positionalen Deklaration resultieren bei einer Record-Struktur aber *veränderliche* Instanzen (get-set), z. B.:

```
Punkt p1 = new(1, 2);
Punkt p2 = new();
p1.X = 13;

public record struct Punkt(double X, double Y);
```

Unveränderliche Instanzen (init-only) erzielt man bei der positionalen Syntax mit Hilfe des zusätzlichen Schlüsselworts **readonly**, z. B.:

```
Punkt p = new(1, 2);
p.X = 13;

public readonly record struct Punkt(double X, double Y);
```

Im nächsten Beispiel werden veränderliche Eigenschaften explizit definiert, was z. B. eine Validierung erlaubt:

```
Punkt p = new(1, 2);
p.X = 13;

public record struct Punkt {
    public double X { get; set; }
    public double Y { get; set; }

    public Punkt(double x, double y) {
        X = x;
        Y = y;
    }
}
```

Im Unterschied zu einer Record-Klasse besitzt eine Record-Struktur (unabhängig von der Definitionssyntax) einen parameterfreien Standardkonstruktor, der alle Felder auf den typspezifischen Nullwert setzt.

Weil bei der Zuweisung einer Strukturinstanz generell ein Kopiervorgang stattfindet, verzichtet der Compiler bei Record-Strukturen auf den Kopierkonstruktor, der bei Record-Klassen für die nichtdestruktive Mutation relevant ist. Der **with**-Operator ist aber auch bei Record-Typen mit Wertsemantik (sowie generell bei Strukturen) verwendbar, z. B.:

Quellcode	Ausgabe

`Punkt p1 = new(1, 2);`
`Punkt p2 = p1 with { X = 3 };`
`Console.WriteLine(p2.X);`
 	
`public record struct Punkt(double X, double Y);`
	`3`

Bei Record-Strukturen ist keine Vererbung möglich:

- Sie dürfen in ihrer Definition keinen Basistyp angeben,
- und sie sind in keiner Typdefinition als Basistyp zulässig.

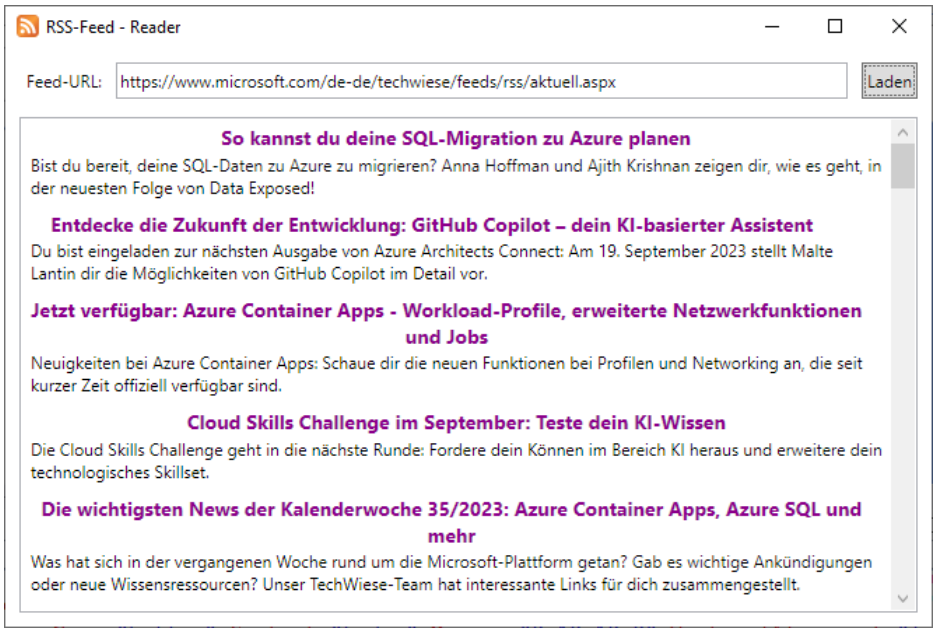
Allerdings darf eine Record-Struktur Schnittstellen implementieren, und eine implementierte Schnittstelle kann manchmal eine ähnliche Rolle spielen wie eine Basisklasse (z. B. bei der Polymorphie, siehe Abschnitt 9.3).

Abschließend werden die Gemeinsamkeiten von Record-Klassen und Record-Strukturen aufgelistet:

- Eigenschaften können positional und/oder explizit definiert werden.
- Bei einer Record-Definition mit Konstruktor-Syntax erstellt der Compiler einen primären Konstruktor und eine **Deconstruct()** – Methode.
- Bei allen Record-Typen erstellt der Compiler weitere Methoden und Operatorüberladungen (siehe Abschnitt 6.7.2).
- Es findet eine inhaltsorientierte Gleichheitsprüfung statt.
- Über einen **with**-Ausdruck lässt sich zu einer Instanz eine Kopie mit partiell veränderten Werten erstellen.

6.8 Ein RSS-Feed - Reader zur Motivationsstärkung

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 und per URL (*Uniform Resource Locator*) ansprechbare Datei, die neue Beiträge zu einem Thema kurz beschreibt und jeweils einen Link zur Vollinformation bietet.[1] Das RSS-Dateiformat ist XML-basiert (*eXtensible Markup Language*) und hat (seit 2002) die aktuelle Version 2.0.

[1] Laut Wikipedia (https://de.wikipedia.org/wiki/Content-Syndication) ist mit *Content-Syndication* im Internet das Zusammenführen der Informationen von verschiedenen Webseiten gemeint.

Eine RSS-Datei der Version 2.0 hat den folgenden Aufbau:[1]

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
  <channel>
    <title>Titel des Feeds</title>
    <link>URL (z. B. "https://www.microsoft.com/de-de/techwiese/default.aspx") </link>
    <description>Kurze Beschreibung des Feeds</description>
    <language>Sprache des Feeds (z. B. "de-de")</language>
    <copyright>Urheberrechtsangabe</copyright>
    <pubDate>Erstellungsdatum (z. B. "Tue, 05 Sep 2023 07:53:06 GMT")</pubDate>

    <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 (z. B.: "3efbeb4f-0de8-4214-a5ef-ff76a1c2b79e")</guid>
      <pubDate>Erstellungsdatum (z. B. "Tue, 05 Sep 2023 07:53:06 GMT")</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.

Zwar haben z. B. der Google-Browser Chrome und der Mozilla-Browser Firefox die direkte (nicht von Erweiterungen abhängige) Unterstützung für RSS-Feeds eingestellt, doch werden im Internet nach wie vor zahlreiche attraktive RSS-Feeds angeboten, sodass sich die Entwicklung eines RSS-Feed - Readers lohnt.[2]

6.8.1 Projekt anlegen mit Vorlage *WPF - Anwendung*

Wir legen im aktiven Visual Studio über den Menübefehl

Datei > Neu > Projekt

ein neues Projekt basierend auf der Vorlage **WPF-Anwendung in C#** mit dem Namen `RssFeedReader` an:[3]

---

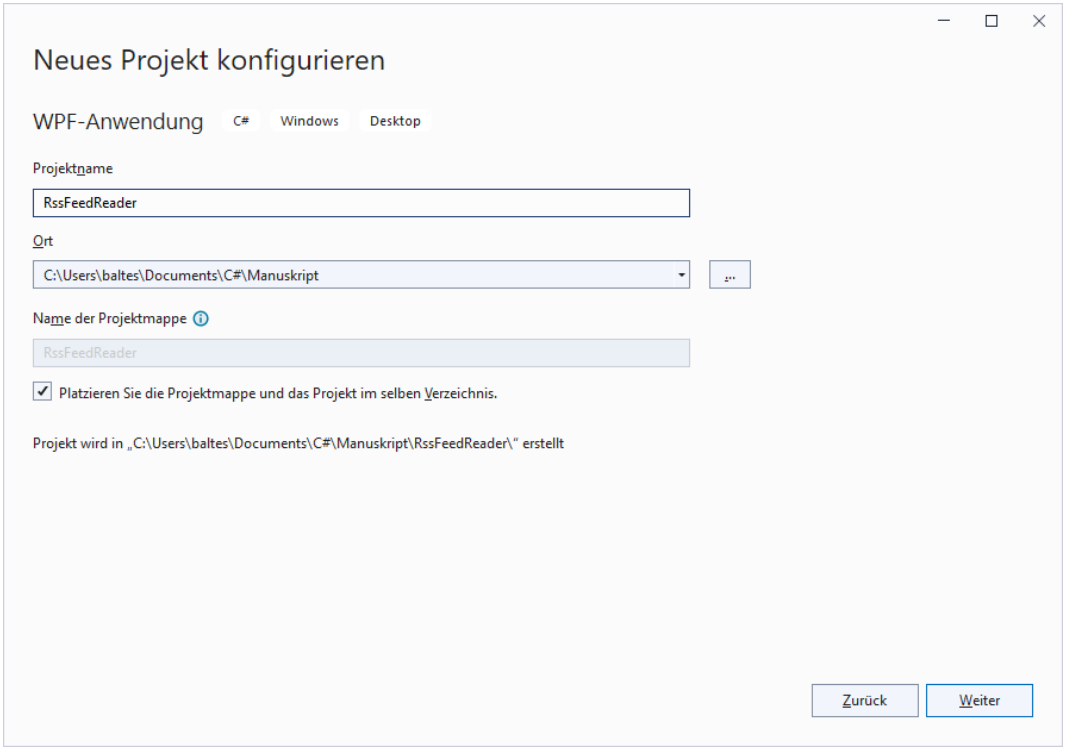
[1] https://de.wikipedia.org/wiki/RSS_(Web-Feed)

[2] https://support.mozilla.org/de/kb/feed-reader-ersatz-firefox

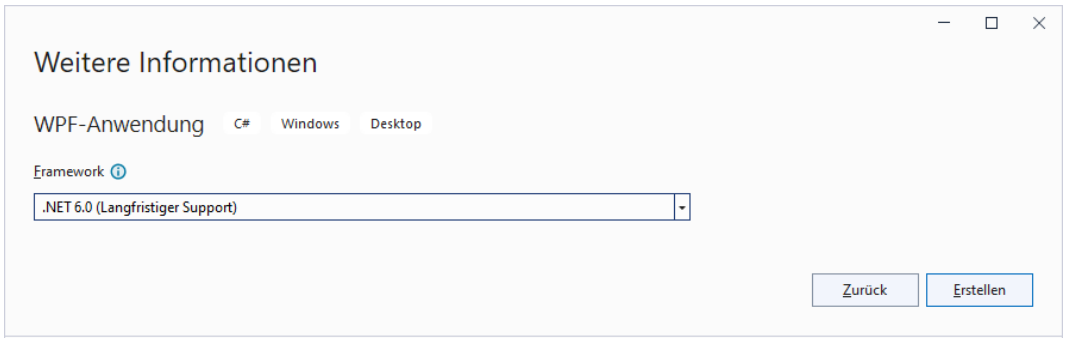
[3] Im Manuskript wird das GUI-Design (*Graphical User Interface*) per WPF (*Windows Presentation Foundation*) bevorzugt gegenüber:

- **WinForms** (veraltet)
- **UWP** (*Universal Windows Platform*, hat sich trotz vieler Bemühungen der Firma Microsoft nicht durchgesetzt)
- **MAUI** (*Multi-platform App UI*, noch ungünstige Kosten/Nutzen – Bilanz für reine Windows-Anwendungen)

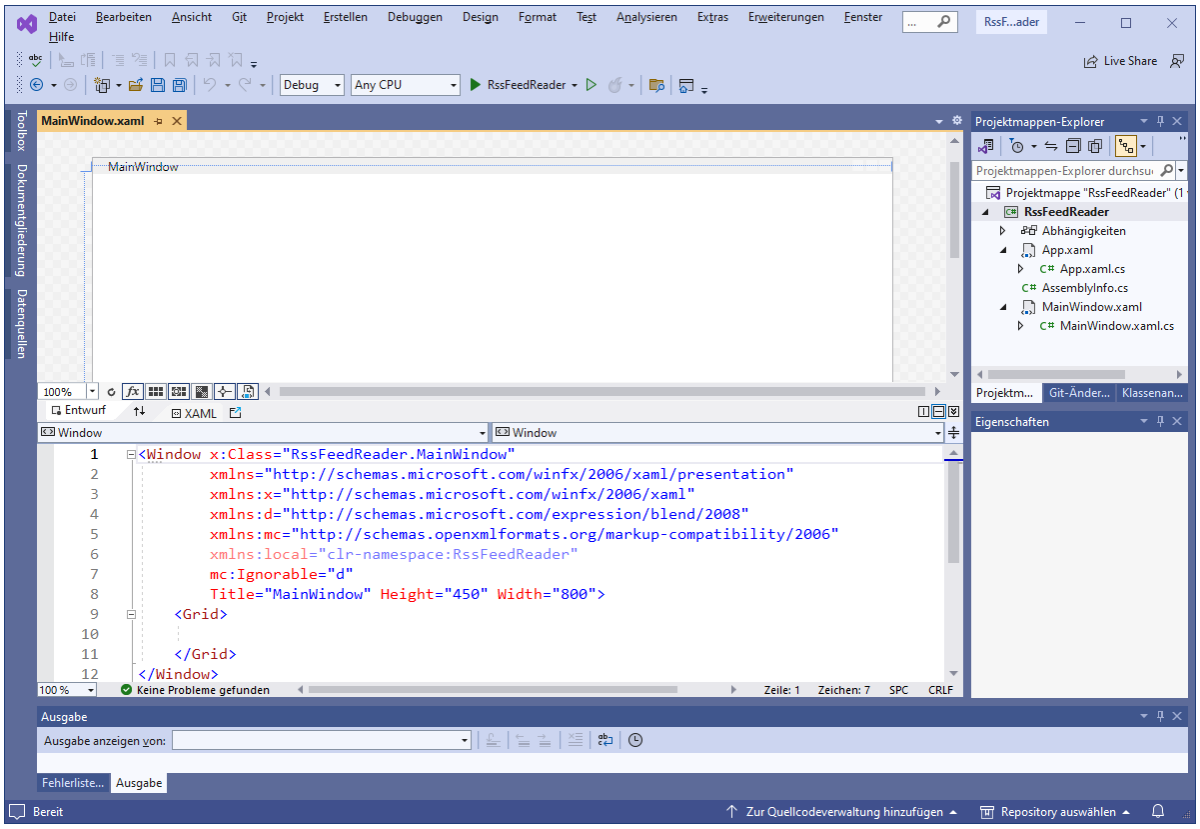
Nähere Erläuterungen zu den GUI-Bibliotheken sind im Abschnitt 12.1 zu finden.



Wir machen **weiter** und wählen die Laufzeitumgebung **.NET 6.0** wegen der langfristigen Unterstützung:



Nach einem Mausklick auf **Erstellen** präsentiert das Visual Studio im WPF- bzw. XAML-Designer einen Rohling für das Fenster der entstehenden Anwendung:



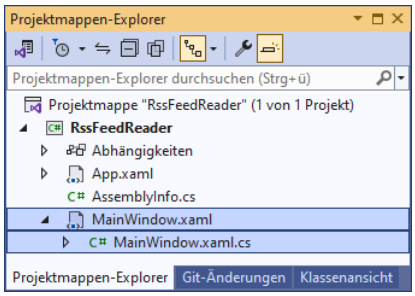
Mittlerweile haben wir bereits einige Routine darin, mit dem WPF-Designer die Bedienoberfläche eines Programms unter Verwendung von konfigurierbaren Steuerelementen aus der **Toolbox** zu gestalten (siehe Abschnitt 6.8.2). Dabei definieren wir die neue Klasse `MainWindow`, die von der BCL-Klasse **Window** im Namensraum **System.Windows** abstammt.

Wie Sie bereits aus dem Abschnitt 5.13 wissen, besteht ein zentrales Merkmal der WPF-Technologie darin, das GUI-Design einer Anwendung durch eine XML-Spezialisierung namens XAML (*eXtensible Application Markup Language*) zu deklarieren. Zu unserem Anwendungsfenster gehört die XAML-Datei **MainWindow.xaml**, 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="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Das **Window**-Wurzelelement definiert ein Anwendungsfenster, also das Erscheinungsbild eines Objekts aus der Klasse `MainWindow` (siehe Wert zum XML-Attribut **x:Class** im Start-Tag). Die initialen Bestandteile des **Window**-Elements wurden schon im Abschnitt 5.13.2 erläutert. Der grafische Fenster-Designer ist ein Werkzeug zur bequemen Bearbeitung der XAML-Datei. Manchmal erweist es sich als praktischer, den XAML-Code direkt zu editieren.

Zur Definition der Fensterklasse `MainWindow` trägt auch eine C# - Quellcode-Datei namens **MainWindow.xaml.cs** bei, für die wir als Entwickler verantwortlich sind. Der **Projektmappen-Explorer** zeigt die XAML- und die Quellcodedatei:



Aus der GUI-Deklaration in der Datei **MainWindow.xaml** entsteht durch Assistententätigkeit die Quellcodedatei **MainWindow.g.cs** mit einer partiellen Definition der Klasse `MainWindow`, die unsere (in **MainWindow.xaml.cs** enthaltene und ebenfalls partielle) Definition dieser Fensterklasse ergänzt.

Über die Fensterklasse `MainWindow` hinaus benötigt eine WPF-Anwendung auch noch eine Anwendungsklasse. Diese stammt von der BCL-Klasse **Application** im Namensraum **System.Windows** ab und trägt im Beispiel den (vom Visual Studio gewählten) Namen `App`. Analog zur Fensterklasse sind eine XAML-Deklarationsdatei (**App.xaml**) und eine C# - Quellcodedatei mit einer partiellen Klassendefinition (**App.xaml.cs**) beteiligt. Beim geplanten Beispielprogramm müssen wir uns um diese beiden Dateien *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**-Wurzelelement u. a. die die folgenden Attribute:

- **x:Class**
  Es nennt die zugehörige Klasse `App` samt Namensraum `RssFeedReader`.
- **StartupUri**
  Es nennt die XAML-Datei zum Fenster, das beim Programmstart angezeigt werden soll.

6.8.2 Steuerelemente aus der Toolbox übernehmen

Holen Sie nötigenfalls im Editor die Datei **MainWindow.xaml** (und damit den WPF-Designer) in den Vordergrund, und 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 Formular die folgenden Steuerelemente:

- ein **TextBox**-Objekt
  Hier können die Benutzer die Feed-Adresse eintragen.
- ein **Label**-Objekt
  Es soll den im Texteingabefeld benötigten Inhalt beschreiben („Feed-URL: “).

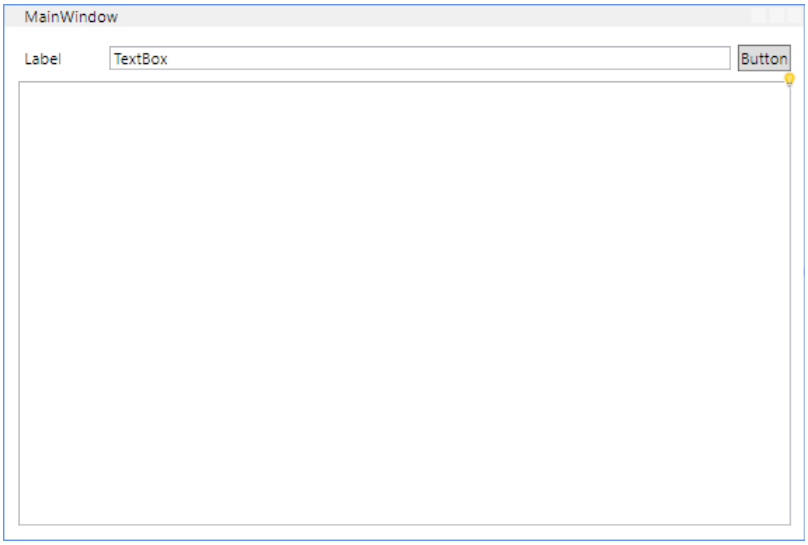
- ein **Button**-Objekt
  Damit fordern die Benutzer das Laden einer RSS-Datei an.
- ein **ListBox**-Objekt
  Hier werden die RSS-Items formatiert angezeigt.

Zur Übernahme eines Steuerelements aus der **Toolbox** ...

- setzt man einen Doppelklick auf den **Toolbox**-Eintrag
- oder arbeitet mit Drag & Drop (Ziehen und Ablegen)

6.8.3 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 Ausdehnung von 1/96 Zoll, sodass 96 DIP gerade einem Zoll (= 2,54 cm) entsprechen. Bei einer Bildschirmauflösung von 96 DPI (*Dots Per Inch*) hat ein physisches Pixel gerade eine Breite und eine Höhe von einem DIP. Positionen und Größen werden in Variablen vom Typ **double** gespeichert.

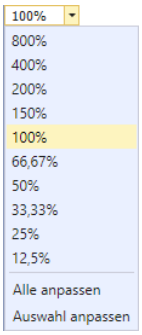
6.8.3.1 Arbeitshilfen

Wir erweitern unser Wissen über die vom WPF-Designer angebotenen Arbeitshilfen im Vergleich zum Abschnitt 5.13, ignorieren aber weiterhin die Bedienhilfen mit Bezug zu der noch nicht behandelten WPF-Containertechnik.

6.8.3.1.1 Zoom-Stufe

Über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

lässt sich eine Zoom-Stufe für die Entwurfsansicht per Drop-Down – Menü wählen:



Bei gedrückter **Strg**-Taste kann man die Zoom-Stufe ohne die vorherige Öffnung des Drop-Down – Menüs per Mausrad ändern.

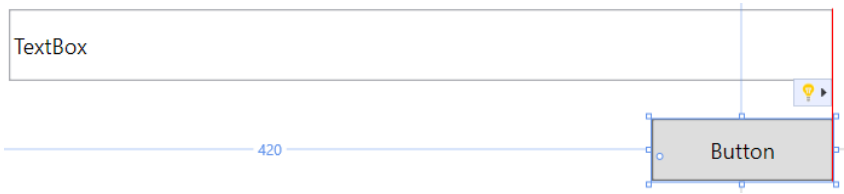
6.8.3.1.2 Andocken an Ausrichtungslinien

Über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

lässt sich das **Andocken an den Ausrichtungslinien** aktivieren. Im aktivierten Zustand, der an einem leicht verstärkten Rand zu erkennen ist, erscheint eine rote Linie, ...[1]

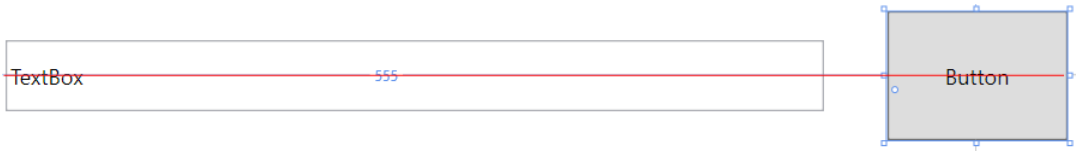
- wenn die *Ränder* von zwei Steuerelementen vertikal

oder horizontal



ausgerichtet sind,

- oder wenn die *Beschriftungen* von zwei Steuerelementen vertikal ausgerichtet sind:



6.8.3.1.3 Rasterpositionen und -linien

Wenn über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

---

[1] Die beiden beteiligten Steuerelemente …
- sind links und oben angedockt,
- haben bei der **Control**-Eigenschaft **VerticalContentAlignment** den Wert **Center**.

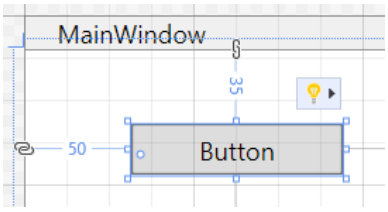
das **Ausrichten an Rasterlinien** aktiviert ist, dann wird ...

- 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

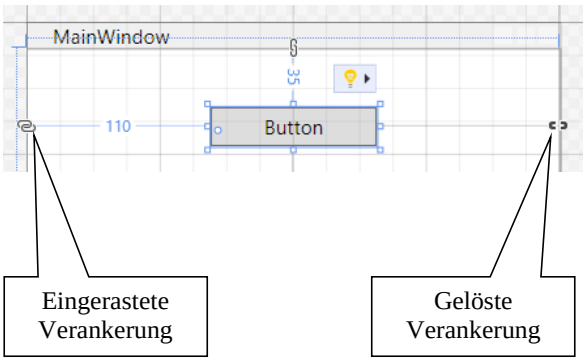
kann man eine (je nach Zoom-Stufe) ausgedünnte Version des Rasterliniengitters ein- bzw. ausblenden:



Im Beispiel beträgt der Abstand zwischen den sichtbaren Rasterlinien 20 DIP.

6.8.3.1.4 Verankerung und Abstände

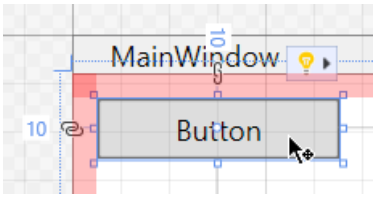
Per Voreinstellung sind die Steuerelemente am linken und am oberen Rand des umgebenden Containers angedockt, was bei einem markierten Steuerelement durch Verankerungssymbole angezeigt wird, z. B.:



Um eine Verankerung vorzunehmen oder aufzuheben, klickt man auf den zugehörigen Verankerungspunkt. Ist beim Öffnen einer Verankerung die gegenüberliegende Verankerung gerade offen, dann wird sie eingerastet.

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, z. B.:

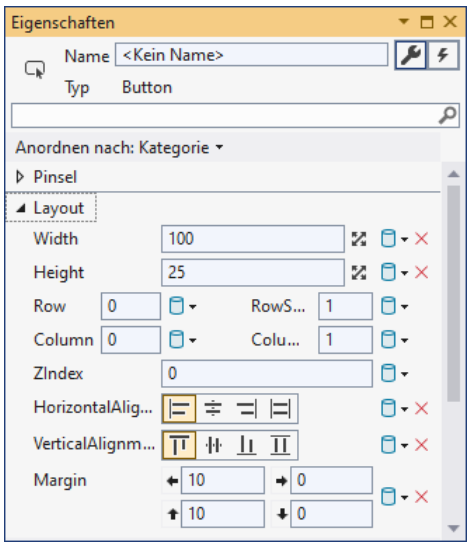


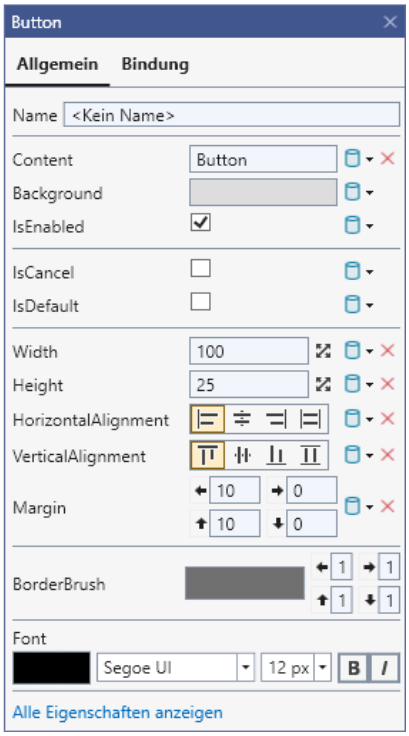
Um bestimmte Randabstände für ein Steuerelement zu realisieren, muss man es nicht unbedingt mit Mausgeschicklichkeit zu einer horizontalen bzw. vertikalen Rasterposition bewegen, sondern kann

im XAML-Attribut **Margin** die gewünschten Werte für den linken, oberen, rechten und unteren Randabstand (in dieser Reihenfolge) eintragen, z. B.:

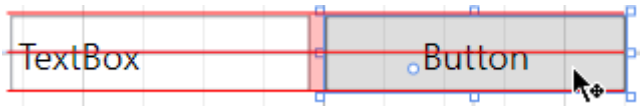
```
<Button Content="Button" Margin="10,10,0,0"
        HorizontalAlignment="Left" VerticalAlignment="Top" Height="25" Width="100"/>
```

Weitere Möglichkeiten zur numerischen Spezifikation der Randabstände bieten die Kategorie **Layout** im **Eigenschaften**-Fenster (links) bzw. das per -Schalter zu öffnende Kontextmenü zum Steuerelement (rechts):





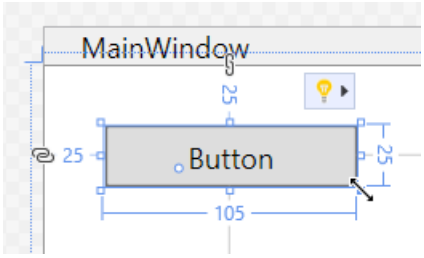
Die vom WPF-Designer vorgeschlagenen Mindestabstände zwischen zwei Steuerelementen sind etwas knapp bemessen, z. B.:



Oft dient es der Übersichtlichkeit, mehr Platz zwischen den Bedienelementen zu lassen.

6.8.3.1.5 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.:



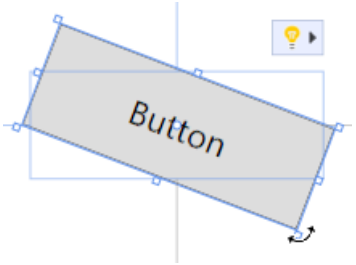
Um eine bestimmte Breite und Höhe für ein Steuerelement zu realisieren, muss man nicht unbedingt die Anfasser mit Mausgeschicklichkeit zu einer horizontalen bzw. vertikalen Rasterposition bewegen, sondern kann die XAML-Attribute **Width** und **Height** auf die gewünschten Werte setzen, z. B.:

```
<Button Content="Button" Width="105" Height="25" Margin="25,25,0,0"
    HorizontalAlignment="Left" VerticalAlignment="Top" />
```

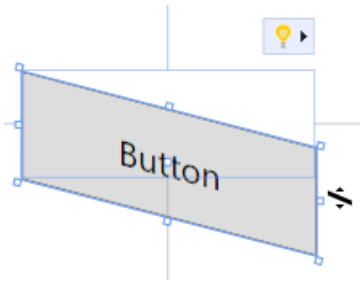
Weitere Möglichkeiten zur numerischen Spezifikation der Ausdehnung bieten das **Eigenschaften**-Fenster (Kategorie **Layout**) sowie das Kontextmenü zu einem Steuerelement (siehe obige Bildschirmfotos).

Bei Verzicht auf die XAML-Attribute **Width** und **Height** findet eine am Inhalt des Steuerelements orientierte automatische Größenberechnung statt.

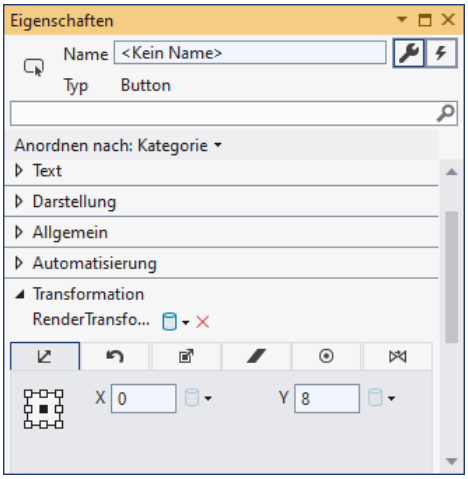
Es ist auch möglich, ein Steuerelement zu drehen



oder zu neigen:



Im **Eigenschaften**-Fenster finden sich diese Einstellungen in der Kategorie **Transformation**, z. B.:



6.8.3.2 Arbeitsablauf

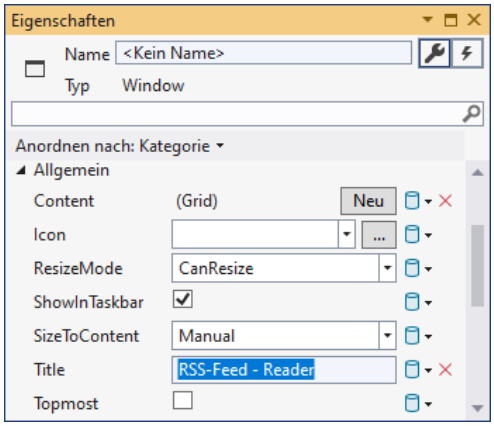
In diesem Abschnitt wird die Erstellung der Bedienoberfläche für den RSS-Feed - Reader Schritt für Schritt beschrieben.

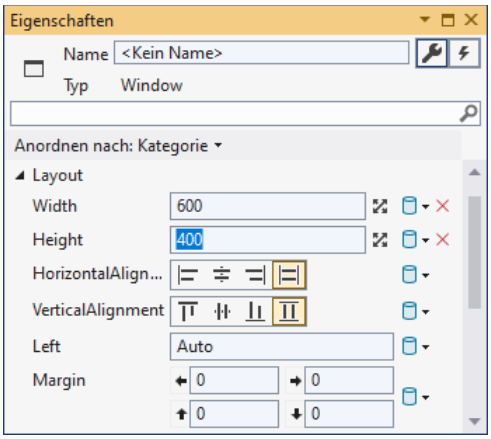
6.8.3.2.1 Anwendungsfenster

Markieren Sie das Anwendungsfenster, was am einfachsten über das **Window**-Wurzelelement in der XAML-Zone gelingt. Achten Sie darauf, dass Sie wirklich das Anwendungsfenster erwischen

und nicht etwa den darin enthaltenen und aus didaktischen Gründen vorläufig ignorierten Container (ein Objekt aus der Klasse **Grid**).

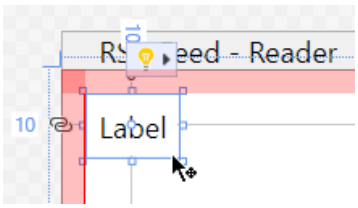
Legen Sie dann per **Eigenschaften**-Fenster eine Titelzeilenbeschriftung und die initiale Fenstergröße fest:





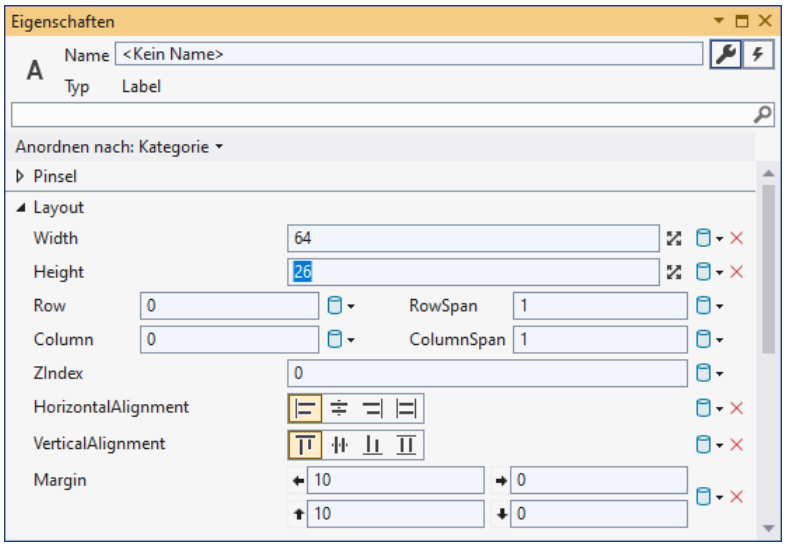
6.8.3.2.2 Label-Objekt

Setzen Sie ein **Label**-Objekt, das den Zweck des Texteingabefelds beschreibt, in die linke obere 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**-Objekts kann die Beschriftung vor Ort geändert werden, z. B. auf „Feed-URL:“.

Als Werte für die Breite und die Höhe des **Label**-Objekts eignen sich 64 und 26 (Kategorie **Layout** im **Eigenschaften**-Fenster:

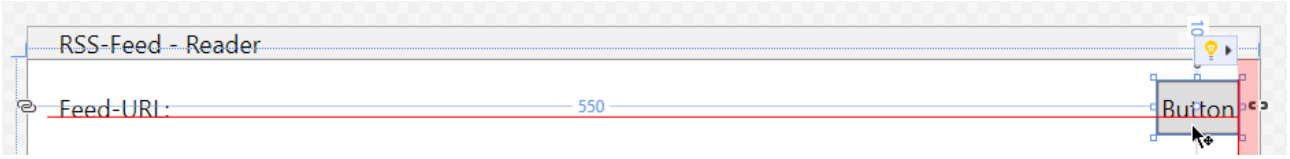


Über den Symbolschalter ⛶ wählt man eine am Inhalt des Steuerelements orientierte automatische Größenberechnung.

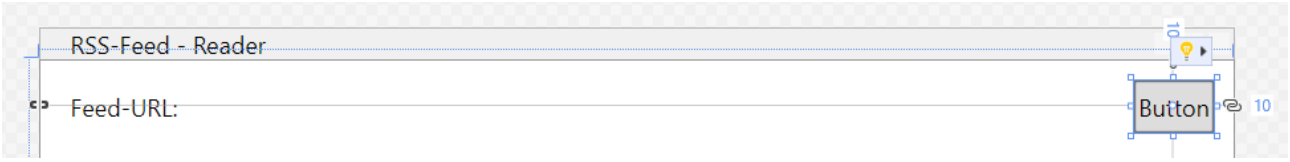
6.8.3.2.3 Button-Objekt

Ergänzen Sie ein **Button**-Objekt, mit dem das Laden einer Feed-Datei angefordert werden soll. Vergeben Sie für die Breite und die Höhe des Schalters die Werte 40 und 26. Setzen Sie den Schalter so in die rechte obere Fensterecke, dass er ...

- in vertikaler Richtung die Textbasislinie des **Label**-Objekts übernimmt
- und zum rechten Fensterrand den Standardabstand von 10 DIP einhält.

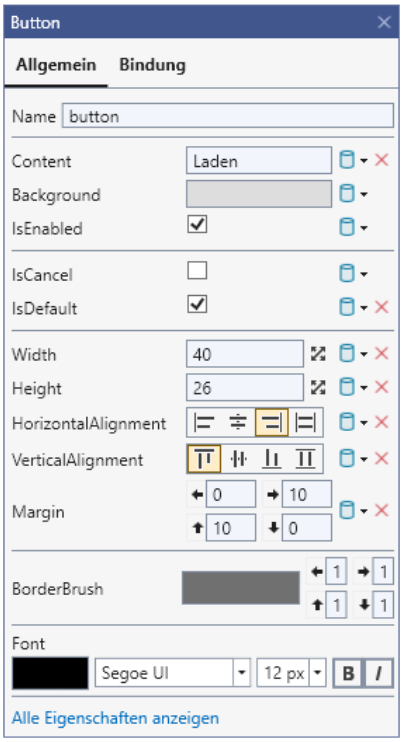


Das **Button**-Objekt sollte oben und *rechts* verankert sein, *nicht* jedoch an der linken oder unteren Fensterkante, damit es bei einer Vergrößerung des Fensters nicht mitwächst, sondern am rechten Rand verbleibt:



Nach einem *einfachen* Mausklick auf das bereits markierte **Button**-Objekt kann es vor Ort beschriftet werden, z. B. durch „Laden“.

Markieren Sie die **IsDefault**-Eigenschaft der Schaltfläche, damit sie im laufenden Programm per **Enter**-Taste angesprochen werden kann. Ein Kontrollkästchen zu dieser oft relevanten Eigenschaft ist im Kontextmenü zum Steuerelement zu finden:

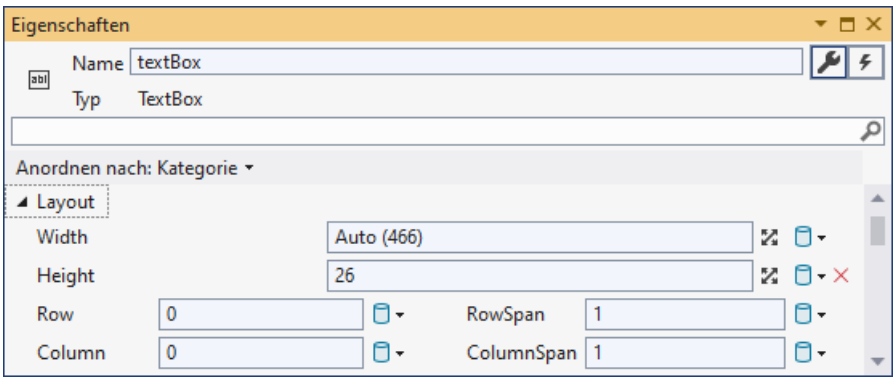


Als **Namen** für die Instanzvariable zum **Button**-Steuerelement, der im Kontextmenü, im **Eigenschaften**-Fenster oder im XAML-Code (als Wert zum Attribut **x:Name**) eingetragen werden kann, wählen wir `button`.

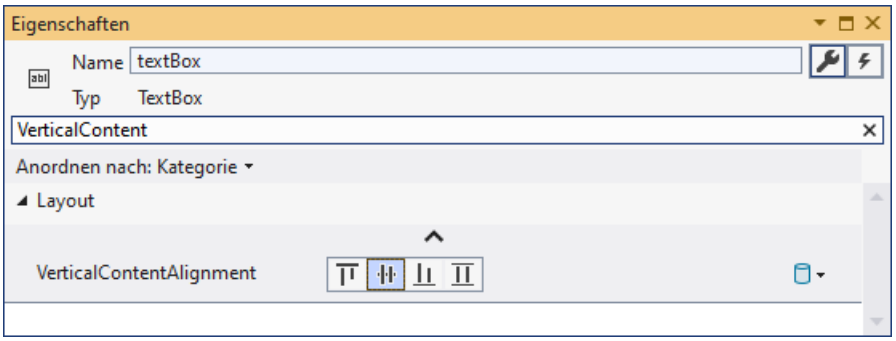
6.8.3.2.4 TextBox-Objekt

Ergänzen Sie ein **TextBox**-Objekt zur Aufnahme der vom Benutzer gewünschten Feed-Adresse.

Übernehmen Sie als Höhe den Wert 26 von den beiden vorhandenen Steuerelementen, um die gemeinsame vertikale Ausrichtung zu erleichtern,

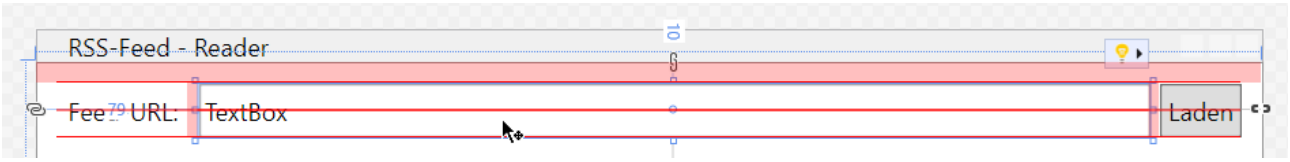


und setzen Sie die Eigenschaft **VerticalContentAlignment** auf den Wert **Center**:

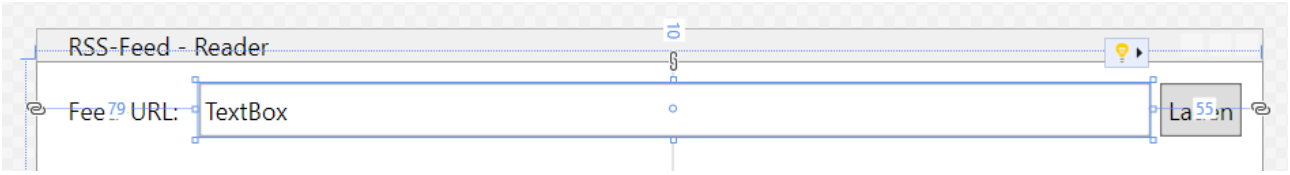


Das **TextBox**-Objekt sollte so zwischen das **Label**- und das **Button**-Objekt gesetzt werden, dass es sich ...

- in vertikaler Richtung an der gemeinsamen Textposition der Nachbarn orientiert
- und horizontal zu beiden Nachbarn den vom Designer vorgeschlagenen Mindestabstand einhält.

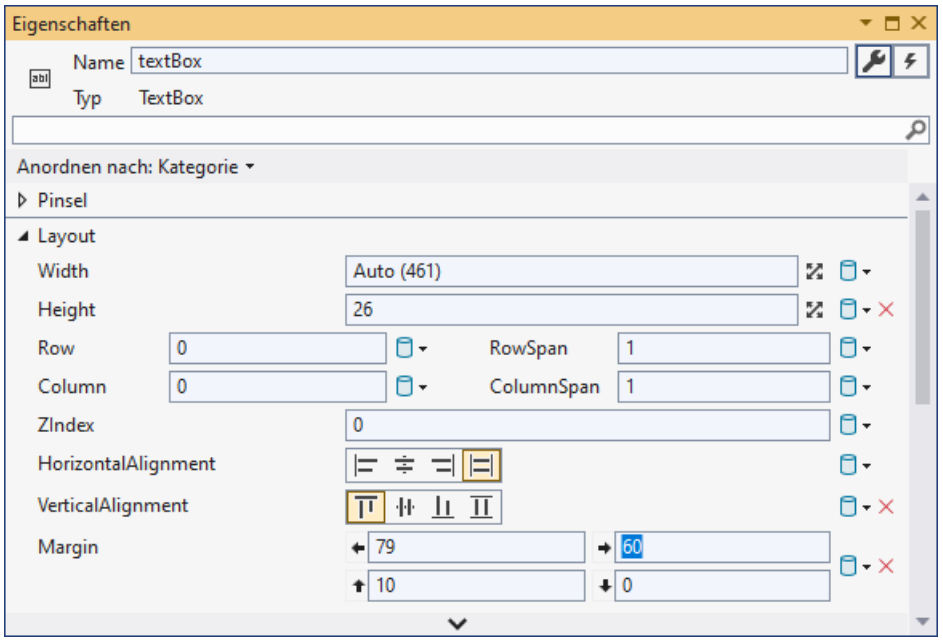


Das **TextBox**-Objekt sollte links, oben und rechts verankert sein, damit es bei einer horizontalen (nicht aber bei einer vertikalen) Vergrößerung des Fensters mitwächst:



Sobald ein Steuerelement an zwei *gegenüberliegenden* Seiten verankert ist, wird für seine Ausdehnung in der zugehörigen Richtung (also für seine Breite bzw. Höhe) vom WPF-Designer die automatische Wertvergabe eingeschaltet, und im XAML-Code fehlt das zugehörige Attribut (**Width** bzw. **Height**).

Wenn Ihnen der Abstand zwischen dem **TextBox**- und dem **Button**-Objekt zu klein erscheint, dann können Sie per **Eigenschaften**-Fenster den rechten Randabstand des **TextBox**-Objekts fein dosiert vergrößern (Kategorie **Layout**), z. B.:



Sorgen Sie über den Wert **NoWrap** für die Eigenschaft **TextWrapping** (Kategorie **Text**) dafür, dass trotz Platznot die Feed-Adresse nicht umgebrochen wird.

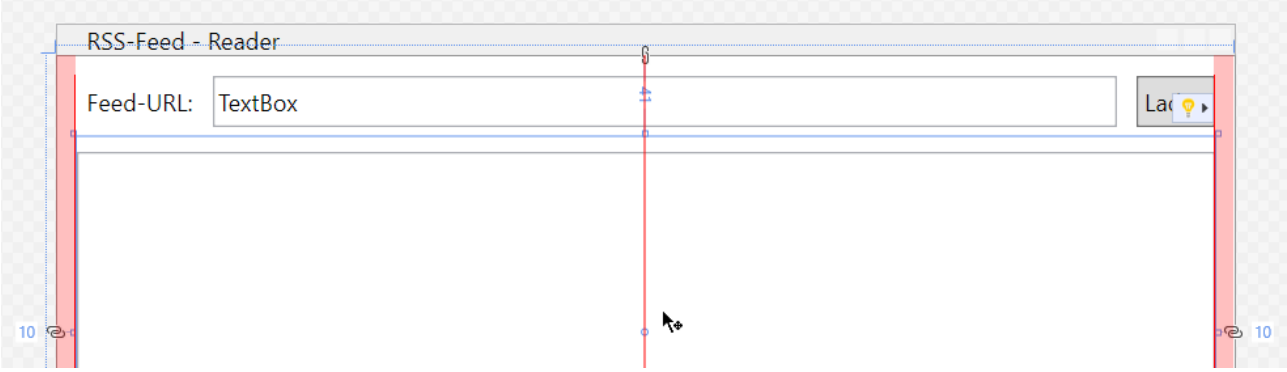
Den initialen Inhalt des **TextBox**-Objekts werden wir per Programm festlegen, sodass wir uns jetzt nicht darum kümmern müssen.

Als **Namen** für die Instanzvariable zum **TextBox**-Steuerelement, der im **Eigenschaften**-Fenster, im Kontextmenü oder im XAML-Code (als Wert zum Attribut **x:Name**) eingetragen werden kann, wählen wir `textBox`.

6.8.3.2.5 ListBox-Objekt

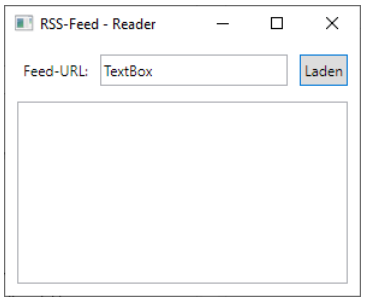
Ergänzen Sie ein **ListBox**-Objekt, das zur formatierten Anzeige der Feed-Items dienen soll. Das Objekt sollte so positioniert werden, dass es ...

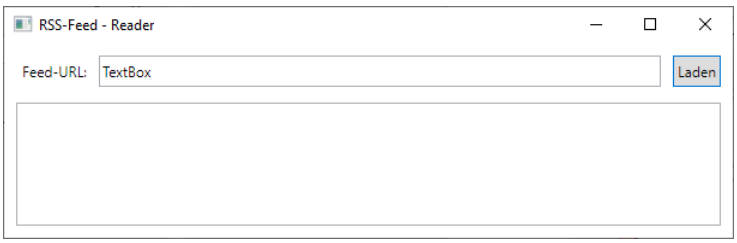
- zum linken, rechten und zum unteren Fensterrand den vom Designer vorgeschlagenen Abstand von 10 DIP einhält
- und nicht allzu dicht unter den drei anderen Bedienelementen sitzt.



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 das Verhalten der Steuerelemente bei variabler Fenstergröße leicht überprüfen, z. B.:





Als **Namen** für die Instanzvariable zum **ListBox**-Steuerelement, der im **Eigenschaften**-Fenster, im Kontextmenü oder im XAML-Code (als Wert zum Attribut **x:Name**) eingetragen werden kann, wählen wir `listBox`.

6.8.4 Fensterklasse `MainWindow`

Wie Sie bereits aus dem Abschnitt 5.13.6 wissen, erzeugt das Visual Studio aufgrund der GUI-Gestaltung 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.cs**
  Der Quellcode in dieser Datei wird bei jedem Erstellen des Programms aufgrund der Datei **MainWindow.xaml** automatisch neu erzeugt, sodass eine Änderung durch den Entwickler sinnlos ist.

Dem C# - Compiler wird durch das Schlüsselwort **partial** im Kopf der Klassendefinition signalisiert, dass der Quellcode auf mehrere Dateien verteilt ist.

Der `MainWindow`-Quellcode in der Datei **MainWindow.xaml.cs** enthält einen Konstruktor, der die Methode **InitializeComponent()** aufruft:

```
using System;
  . . .
using System.Windows.Shapes;

namespace RssFeedReader {
   /// <summary>
   /// Interaktionslogik für MainWindow.xaml
   /// </summary>
   public partial class MainWindow : Window {
      public MainWindow() {
         InitializeComponent();
      }
   }
}
```

Diese Methode wird in der Datei **MainWindow.g.cs** implementiert.

Aufgrund unserer Tätigkeit im WPF-Designer enthält die Fensterklasse Member-Objekte aus den Klassen **TextBox**, **Button** und **ListBox** im Namensraum **System.Windows.Controls**, die Steuerelemente der grafischen Bedienoberfläche repräsentieren. In der Datei **MainWindow.g.cs** befinden sich die Deklarationen der zugehörigen Instanzvariablen mit den von uns festgelegten Namen (`textBox`, `button` und `listBox`):[1]

[1] Wie man die Datei **MainWindow.g.cs** anzeigen lassen kann, wird im Abschnitt 5.13.6 erläutert.

```
using System;
  . . .
using System.Windows.Shapes;

namespace RssFeedReader {
   /// <summary>
   /// MainWindow
   /// </summary>
   public partial class MainWindow : System.Windows.Window,
                                     System.Windows.Markup.IComponentConnector {
  . . .
      internal System.Windows.Controls.TextBox textBox;
  . . .
      internal System.Windows.Controls.Button button;
  . . .
      internal System.Windows.Controls.ListBox listBox;
     . . .
      public void InitializeComponent() {
        . . .
      }
   }
}
```

Das Visual Studio hat sich dafür entschieden, die Datenkapselung zu lockern und die voreingestellte Schutzstufe **private** durch **internal** zu ersetzen, sodass alle Klassen im Assembly direkt zugreifen können. Sicherheit und bequeme Programmierung sind nicht immer leicht zu vereinbaren.

Damit beim Programmstart im Texteingabefeld die Adresse des Microsoft-Feeds mit aktuellen Informationen für Entwickler erscheint, nehmen wir im `MainWindow`-Konstruktor für die **Text**-Eigenschaft des **TextBox**-Objekts eine entsprechende Initialisierung vor:

```
public MainWindow() {
   InitializeComponent();
   textBox.Text = "https://www.microsoft.com/de-de/techwiese/feeds/rss/aktuell.aspx";
}
```

6.8.5 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 (falls vorhanden).
- Die Items im RSS-Feed werden an das **ListBox**-Objekt zur formatierten Anzeige übergeben.

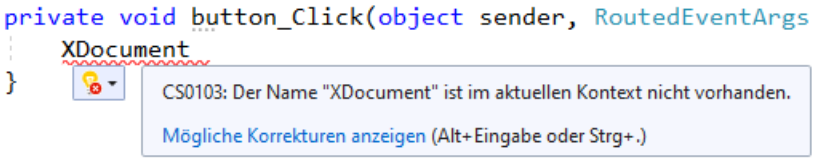
Setzen Sie im WPF-Designer einen Doppelklick auf den Befehlsschalter, sodass die Entwicklungsumgebung in der Datei **MainWindow.xaml.cs** die Instanzmethode `button_Click()` der Klasse `MainWindow` mit leerem Rumpf anlegt

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

}
```

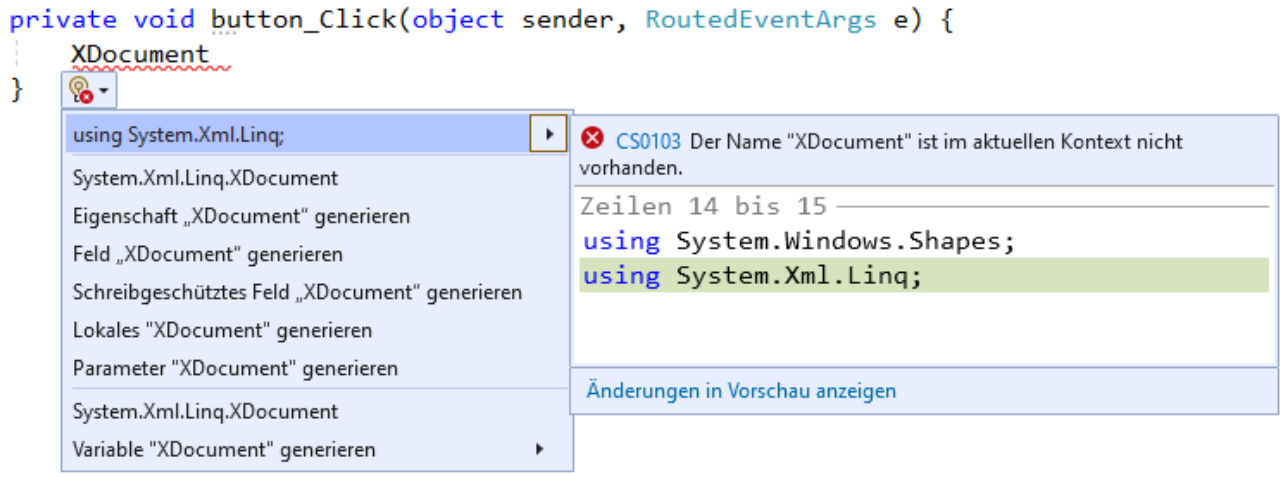
und die Quellcodedatei im Editor öffnet.

Wir initiieren die Übertragung der vom Benutzer benannten RSS-Datei in ein Objekt der BCL-Klasse **XDocument**, die ein XML-Dokument verwalten kann. Für das nicht triviale, mit einem Internetzugriff verbundene Laden der RSS-Datei verwenden wir die statische **XDocument**-Methode **Load()** und übergeben die **Text**-Eigenschaft des **TextBox**-Bedienelements als Aktualparameter.

Der optimistisch in der Methode `button_Click()` als Datentyp für eine lokale Variable eingetippte Klassenname **XDocument** wird von der Entwicklungsumgebung rot unterschlängelt:



Wenn der Mauszeiger über der Unterschlängelung verharrt, erscheinen neben einer Fehlerbeschreibung auch Unterstützungsangebote. Nach einem Mausklick auf die Glühbirne oder auf den Link **Mögliche Korrekturen anzeigen** wird u. a. vorgeschlagen, den Namensraum **System.Xml.Linq**, in dem sich die Klasse **XDocument** befindet, per **using**-Direktive zu importieren:[1]



Wir übernehmen per Mausklick auf **using System.Xml.Linq;** den Vorschlag zum Namensraumimport.

Nun geben wird der **XDocument**-Referenzvariablen den Namen `feed` und weisen als Wert die Adresse des **XDocument**-Objekts zu, das im Erfolgsfall von der eben beschriebenen statischen **XDocument**-Methode **Load()** basierend auf der Internetadresse in `textBox.Text` erstellt wird:

```
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, sodass z. B. in einer **foreach**-Schleife ein Iterieren über die Elemente möglich ist. Wir werden uns im Kapitel 9 mit Schnittstellen im Allgemeinen und im Abschnitt 9.5 mit der Schnittstelle **IEnumerable** im Besonderen 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, beim Erzeugen des Containers festzulegenden Typ, darstellt. In unserem Fall treten RSS-Items als Elemente

---

[1] In Konsolanwendungen waren **using**-Direktiven selten erforderlich, weil die entsprechende Projektvorlage im Visual Studio über das folgende Element in der Projektdatei

```
<ImplicitUsings>enable</ImplicitUsings>
```

dafür gesorgt hat, dass alle wichtigen Namensräume automatisch importiert wurden. Von der Projektvorlage für WPF-Anwendungen wird *kein* **ImplicitUsing**-Element in die Projektdatei befördert.

auf, und die modellierende Klasse mit dem Namen `RssItem` muss erst noch definiert werden. Mit generischen Typen und mit Kollektionen werden wir uns im Kapitel 8 bzw. 11 beschäftigen, sodass es motivationspsychologisch zu begrüßen ist, wenn Ihnen jetzt relevante Beispiele für das typgenerische Programmieren 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`:

```
private void button_Click(object sender, RoutedEventArgs e) {
    XDocument feed = XDocument.Load(textBox.Text);
    List<RssItem> items = new();
}
```

CS0246: Der Typ- oder Namespacename "RssItem" wurde nicht gefunden (möglicherweise fehlt eine using-Direktive oder ein Assemblyverweis).

Mögliche Korrekturen anzeigen (Alt+Eingabe oder Strg+.)

Zur erwarteten Fehlermeldung fordern wir die Korrekturvorschläge an

```
private void button_Click(object sender, RoutedEventArgs e) {
    XDocument feed = XDocument.Load(textBox.Text);
    List<RssItem> items = new();
}
```

class-Objekt "RssItem" in neuer Datei generieren
class-Objekt "RssItem" generieren
Geschachteltes class-Objekt "RssItem" generieren
Neuen Typ generieren...
Typ "RssItem" generieren

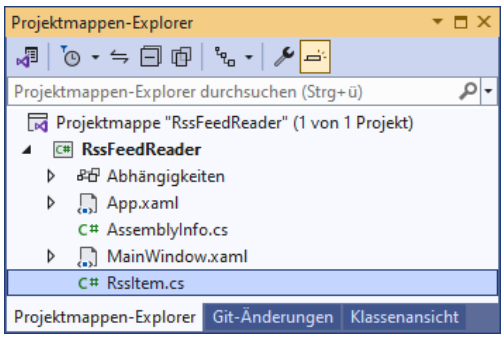
CS0246 Der Typ- oder Namespacename "RssItem" wurde nicht gefunden (möglicherweise fehlt eine using-Direktive oder ein Assemblyverweis).

"RssItem.cs" wird zu "RssFeedReader" hinzugefügt mit dem Inhalt:

```
namespace RssFeedReader {
    internal class RssItem {
    }
}
```

Änderungen in Vorschau anzeigen

und wählen den ersten. Daraufhin erstellt das Visual Studio eine Klasse mit dem gewünschten Namen `RssItem` in einer eigenen Quellcodedatei, die im **Projektmappen-Explorer** auftaucht:



In der automatisch erstellten Klassendefinition

```
namespace RssFeedReader {
  internal class RssItem {
  }
}
```

werden wir noch automatisch implementierte Eigenschaften (vgl. Abschnitt 5.5.2) 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. 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:

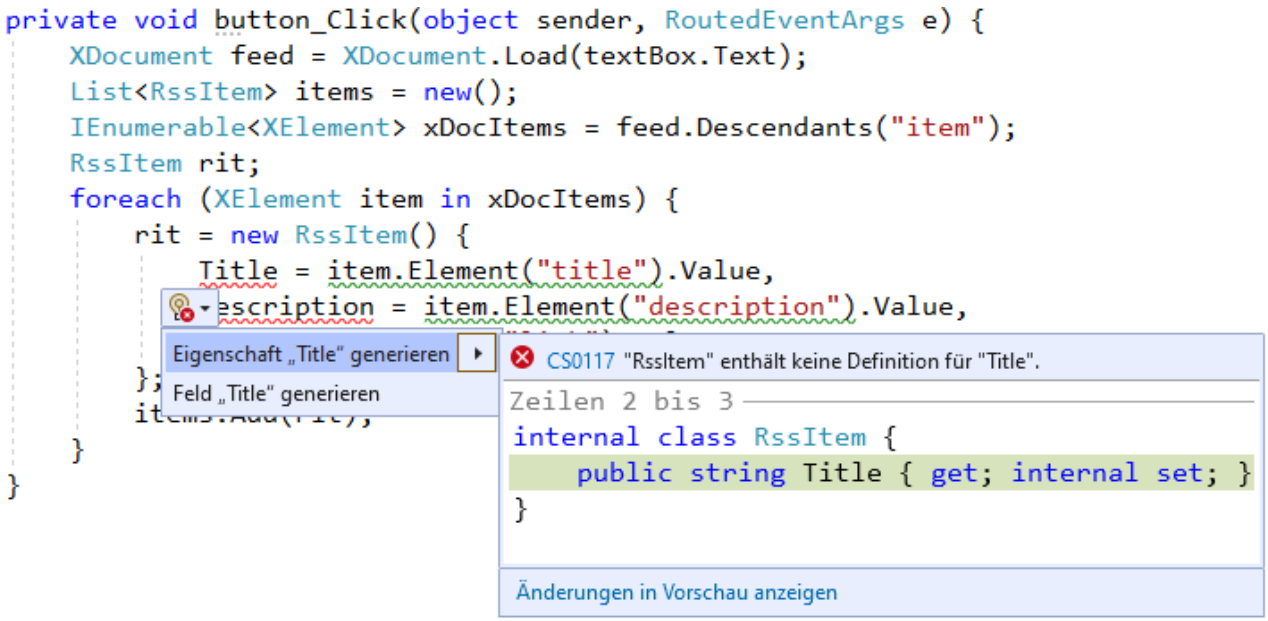
```
IEnumerable<XElement> xDocItems = feed.Descendants("item");
```

In einer **foreach**-Schleife erstellen wir zu jedem **XElement**-Objekt ein `RssItem`-Objekt und nehmen dieses per **Add()** - Methode in die Kollektion `items` vom Typ **List**<`RssItem`> auf:

```
private void button_Click(object sender, RoutedEventArgs e) {
    XDocument feed = XDocument.Load(textBox.Text);
    List<RssItem> items = new();
    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);
    }
}
```

Statt eines explizit definierten initialisierenden `RssItem`-Konstruktors wird die im Abschnitt 5.4.3.2 beschriebene Objektinitialisierung verwendet.

In der **foreach**-Schleife werden Subelemente eines `<item>` - Elements sowie korrespondierende Eigenschaften der Klasse `RssItem` verwendet, die dort noch nicht definiert sind (`Title`, `Description` und `Url`). Unsere Entwicklungsumgebung erkennt das Problem und schlägt geeignete Maßnahmen vor, z. B.:



Wir übernehmen den ersten Vorschlag, und die Entwicklungsumgebung erweitert die Definition der Klasse `RssItem` um die automatisch implementierte Eigenschaft `Title` (siehe Abschnitt 5.5.2):

```
internal class RssItem {
   public string Title { get; internal set; }
}
```

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);
   List<RssItem> items = new();
   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:

6.8.6 Formatierung der Listenelemente per DataTemplate-Objekt

Bislang zeigt das **ListBox**-Objekt zu jedem RSS-Item lediglich den Datentyp an (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="10,49,10,10">
  <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 12.3.2.3) 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 12.6.3) mit vertikaler Orientierung, um zwei **TextBlock**-Objekte übereinander zu präsentieren.

Ein **TextBlock**-Objekt erhält seine Daten über die **Datenbindungstechnologie**. Durch die folgende Attributsyntax mit einer sogenannten *Markup-Erweiterung* (vgl. Abschnitt 12.3.2.3.6)

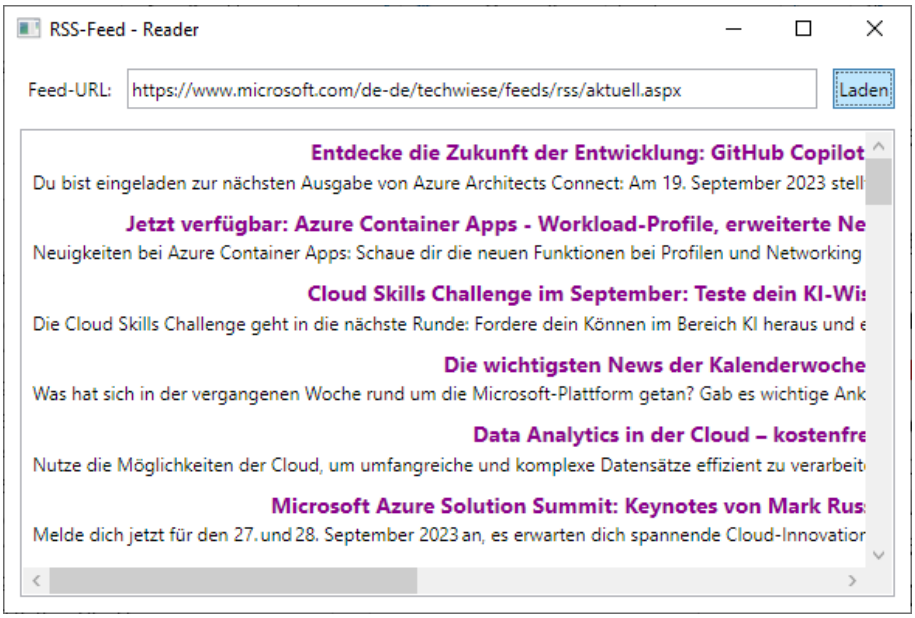
```
Text="{Binding Path=Title}"
```

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 erzeugt und der **ListBox**-Eigenschaft **ItemsSource** zugewiesen (siehe Abschnitt 6.8.5):

```
listBox.ItemsSource = items;
```

Die weiteren **TextBlock** - XAML-Attribute dienen der Formatierung. Wenn ihnen z. B. die Farbe **DarkMagenta** für den Titel missfällt, können Sie z. B. einen alternativen Farbnamen aus der Enumeration **ConsoleColor** (Namensraum **System**) verwenden.

Der bei **ListBox**-Steuerelementen per Voreinstellung vorhandene *horizontale* Rollbalken ist bei unserem nun schon deutlich verbesserten RSS-Feed - Reader für eine unpraktische Textpräsentation verantwortlich



und wird daher über das folgende Attribut zum **ListBox**-Element in der XAML-Datei **MainWindow.xaml** abgeschaltet:

```
<ListBox . . . ScrollViewer.HorizontalScrollBarVisibility="Disabled">
   . . .
</ListBox>
```

HorizontalScrollBarVisibility ist eine sogenannte *Abhängigkeitseigenschaft* der Klasse **ScrollViewer**. Warum man mit ihrer Hilfe für ein Objekt der Klasse **ListBox** den horizontalen Rollbalken beeinflussen kann, werden Sie im Abschnitt 12.5 erfahren.

Nun sieht die Anzeige des Feed-Readers akzeptabel aus:

Bei manchen RSS-Feeds fällt ein weiterer Schönheitsfehler auf: Unser Reader kann die in `<description>` - Elementen der RSS-Datei enthaltenen HTML-Elemente nicht interpretieren, z. B.:

Daher sollten die HTML-Elemente entfernt werden. Eine bei **StackOverflow**, einem professionellen Forum zu Fragen der Software-Entwicklung, von *Jossef Harush Kadouri* veröffentlichte Lösung erledigt diese Aufgabe mit Hilfe der statischen Methode **Replace()** aus der Klasse **Regex** im Namensraum **System.Text.RegularExpressions**.[1] Das Bereinigen der `<description>` - Elemente in den RSS-Items findet im Rahmen der Objektinitialisierung statt, die wir in der Methode `button_Click()` bei der Erstellung eines `RssItem`-Objekts verwenden (siehe Abschnitt 6.8.5):

```
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
};
```

Die zu suchende und durch **String.Empty** zu ersetzende Zeichenfolge wird über einen sogenannten *regulären Ausdruck* definiert:[2]

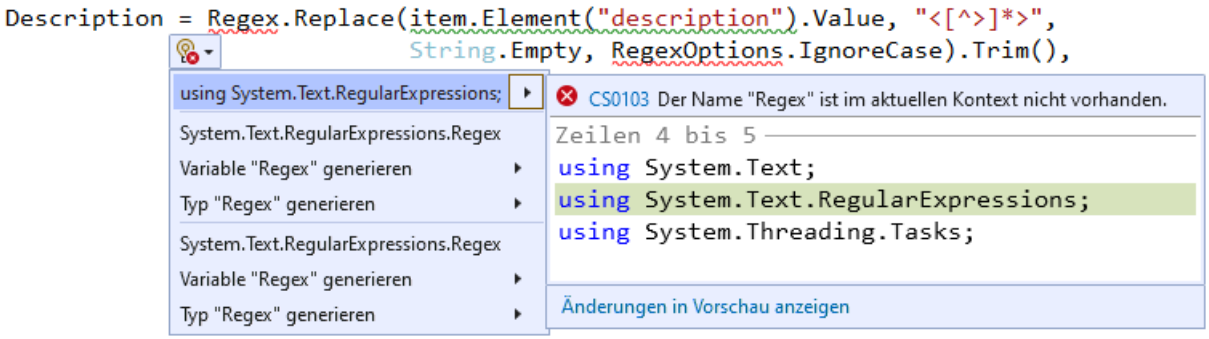
---

[1] http://stackoverflow.com/questions/23040422/delete-img-path-from-description

[2] Siehe z. B. https://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck

- <
  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.

Damit die Klasse **Regex** und die Enumeration **RegexOptions** ohne Namensraumpräfix angesprochen werden können, importieren wir den Namensraum **System.Text.RegularExpressions**:

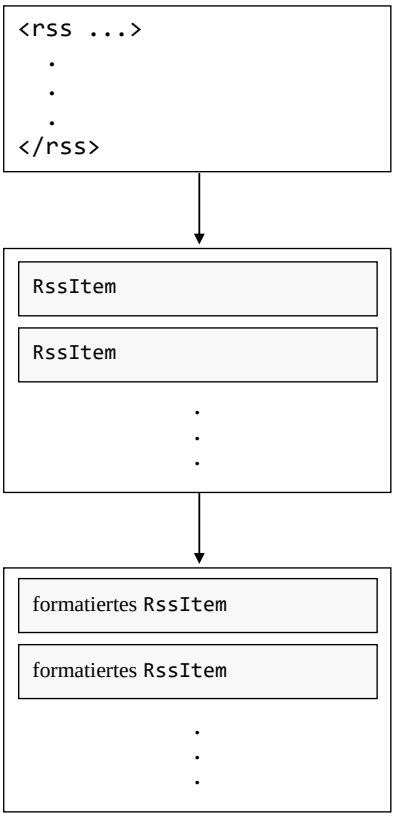


Die HTML-Inkompetenz unseres Feed-Readers ist damit zwar nicht behoben, fällt aber weniger auf:

In der folgenden Abbildung wird skizziert, wie die in den Abschnitten 6.8.5 und 6.8.6 beschriebenen Typen

- **XDocument**
- `RssItem`
- **List**`<RssItem>`
- **ListBox**
- **DataTemplate**

zusammenarbeiten, um aus einer RSS-Datei ein **ListBox**-Steuerelement mit formatierten RSS-Items zu erstellen:



Typ: **XDocument**
Variable: `feed`

Typ: **List<RssItem>**
Variable: `items`

Typ: **ListBox**
Variable: `listBox`
Eigenschaften:
- **ItemsSource** = `items`
- **ItemTemplate** = **new DataTemplate() {...}**

6.8.7 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 das Kapitel 13 und zeigen ggf. im **catch**-Block eine Fehlermeldung an:

```
private void button_Click(object sender, RoutedEventArgs e) {
   Cursor oldCursor = this.Cursor;
   Cursor = Cursors.Wait;
   try {
      XDocument feed = XDocument.Load(textBox.Text);
      List<RssItem> items = new();
      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.Message, "Es ist ein Fehler aufgetreten.",
                      MessageBoxButton.OK, MessageBoxImage.Error);
   } finally {
      Cursor = oldCursor;
   }
}
```

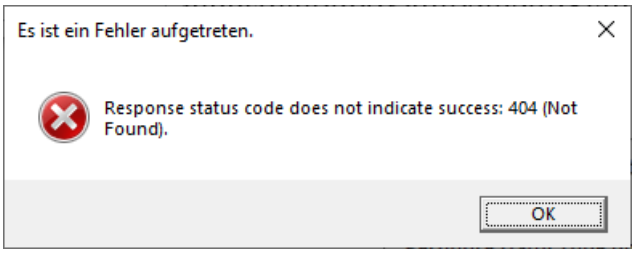
Von der statischen Methode **Show()** der Klasse **MessageBox** im Namensraum **System.Windows**

public static MessageBox Show(Window *owner***, string** *messageBoxText***, string** *caption***, MessageBoxButton** *button***, MessageBoxImage** *icon***)**

wird eine Überladung mit den folgenden Parametern benutzt:[1]

- *owner*
  Das Meldungsfenster orientiert seinen Erscheinungsort am übergeordneten Fenster. Weil das Anwendungsfenster unseres Programms die Methode `button_Click()` ausführt, wird mit der Referenz **this** das Anwendungsfenster als Besitzer benannt.
- *messageBoxText*
  Dieser Parameter ist für die Fehlerbeschreibung gedacht. Wir geben den Text in der **Message**-Eigenschaft des bei einem Fehler übermittelten **Exception**-Objekts aus.
- *caption*
  Mit diesem Parameter wird die Titelzeile des Meldungsfensters festgelegt.
- *button*
  Ein Wert der Enumerationen **MessageBoxButton** sorgt für die gewünschte Ausstattung der Meldungsdialogbox mit Schaltflächen.
- *icon*
  Ein Wert der Enumerationen **MessageBoxImage** sorgt für die gewünschte Ausstattung der Meldungsdialogbox mit einem Symbol.

Existiert z. B. die vom Benutzer angegebenen Adresse nicht, dann erscheint die folgende Fehlermeldung:



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:

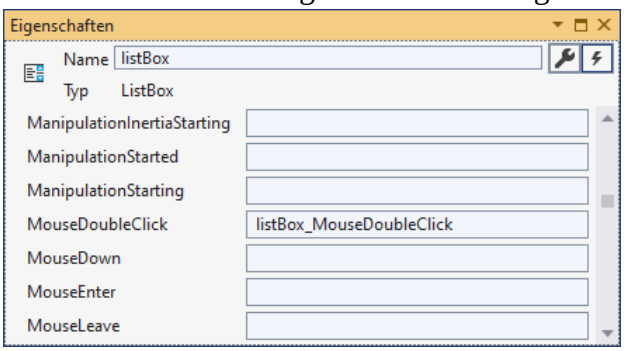
```
finally {
   Cursor = oldCursor;
}
```

[1] https://learn.microsoft.com/en-us/dotnet/api/system.windows.messagebox.show

6.8.8 Doppelklick-Ereignisbehandlung zum ListBox-Steuerelement

Nach einem Doppelklick auf ein RSS-Item soll unser RSS-Feed – Reader dafür sorgen, dass die zugehörige Vollinformation vom bevorzugten Browser angezeigt wird. 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 Texteingabefeld zum Ereignis **MouseDoubleClick**:



- Daraufhin erscheint der Eintrag `listBox_MouseDoubleClick` im Textfeld, und in der Quellcodedatei **MainWindow.xaml.cs** wird für unsere Fensterklasse `MainWindow` die Instanzmethode `listBox_MouseDoubleClick()`angelegt.

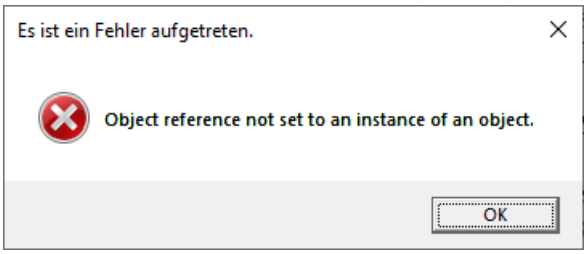
Im Rumpf dieser Methode verwenden wir die statische Methode **Start()** der Klasse **Process** im Namensraum **System.Diagnostics** dazu, um das Betriebssystem zu beauftragen, mit einem geeigneten Programm den Link in demjenigen `RssItem`-Objekt zu öffnen, das aktuell im **ListBox**-Objekt ausgewählt ist. Die **ListBox**-Eigenschaft **SelectedItem** liefert das gewählte Item, besitzt aber den Typ **Object**, sodass 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 das Kapitel 13 eine **try-catch** - Anweisung und zeigen ggf. im **catch**-Block eine Fehlermeldung an, welche die **Message**-Eigenschaft des **Exception**-Objekts verwendet:

```
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 {
         ProcessStartInfo psi = new() { FileName = item.Url,
                                        UseShellExecute = true };
         Process.Start(psi);
      } catch (Exception ex) {
         MessageBox.Show(this, ex.Message, "Es ist ein Fehler aufgetreten.",
                         MessageBoxButton.OK, MessageBoxImage.Error);
      }
}
```

Wir unternehmen aber nur dann einen Startversuch, wenn die **Url**-Eigenschaft des gewählten Items auf ein nicht-leeres **String**-Objekt zeigt. Ansonsten hätte z. B. vor dem Laden eines RSS-Feeds ein Doppelklick auf das **ListBox**-Steuerelement den folgenden Effekt:

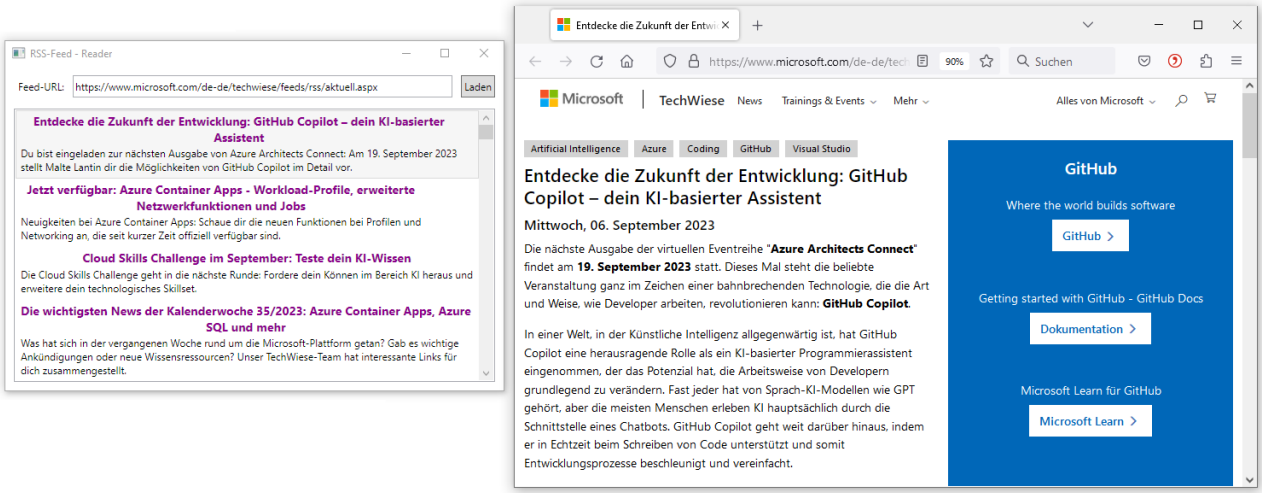


Der zu startende Prozess wird über ein Objekt der Klasse **ProcessStartInfo** aus dem Namensraum **System.Diagnostics** beschrieben, das wir mit Hilfe eines Objektinitialisierers erstellen:[1]

- In der **String**-Eigenschaft **FileName** wird der URL (Uniform Resource Locator) abgelegt.
- Die **bool**-Eigenschaft **UseShellExecute** muss auf den Wert **true** gesetzt werden, weil ab .NET ab Version 5 der für unsere Zwecke ungeeignete Wert **false** voreingestellt ist.

Das **ProcessStartInfo**-Objekt sorgt als Parameter in einem Aufruf der statischen Methode **Start()** der Klasse **Process** dafür, dass die im RSS-Item enthaltene Webseite durch den voreingestellten Browser geöffnet wird.

Nun ist unser Feed-Reader in einem brauchbaren Zustand. Nach einem Doppelklick auf ein Item öffnet der bevorzugte Browser den zugehörigen Link, z. B.:



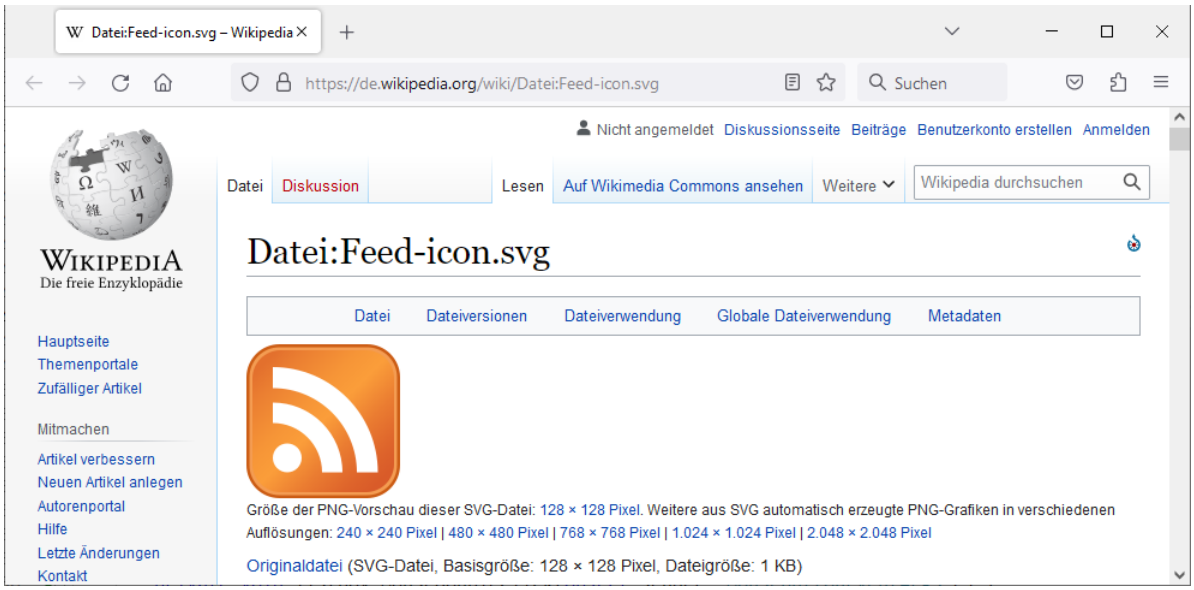
6.8.9 Symbol für das Programm und sein Fenster

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-Symbol im **PNG-Format** (***P**ortable **N**etwork **G**raphics*) mit einer 128 × 128 Pixelmatrix:

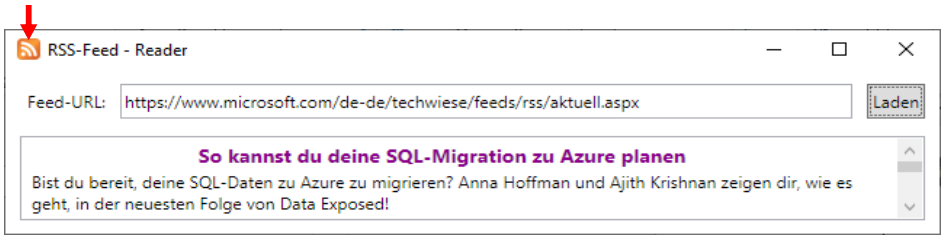
[1] https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.filename



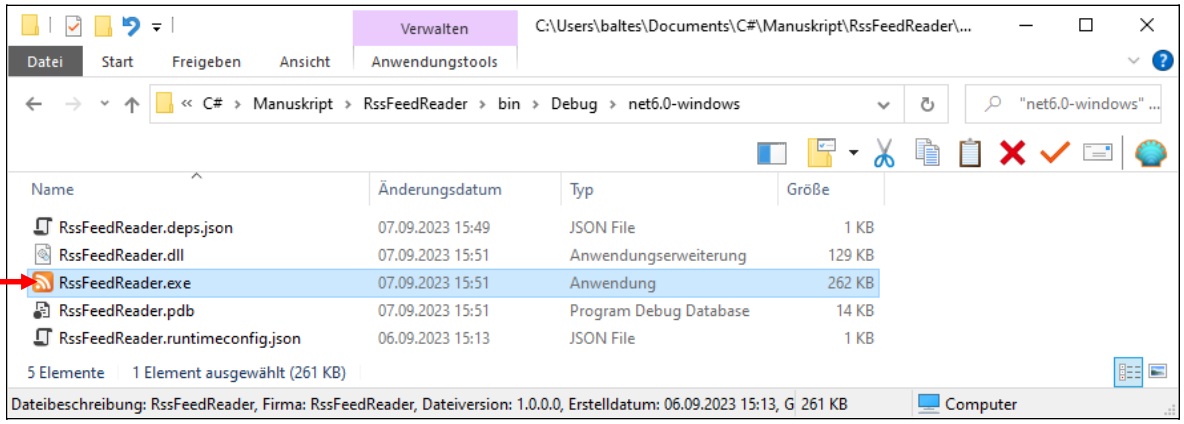
Daraus lässt sich z. B. mit Hilfe der folgenden Webseite

https://convertico.com/

eine Windows-Symboldatei (z. B. mit dem Namen **RssFeed.ico**) erstellen, die sich für das Fenstersymbol



und für das Anwendungssymbol



eignet.

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.
- Öffnen Sie das Fenster mit den Projekteigenschaften über den Menübefehl

  Projekt > Eigenschaften

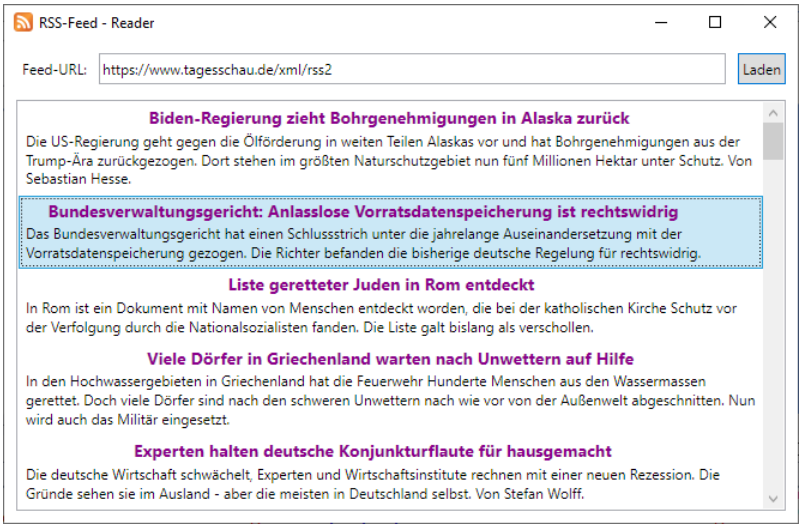
  und wählen Sie im Bereich **Anwendung** das **Symbol**:

Das Symbol wird in die Projektdatei eingetragen:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net6.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWPF>true</UseWPF>
     <ApplicationIcon>RssFeed.ico</ApplicationIcon>
  </PropertyGroup>

  <ItemGroup>
    <Content Include="RssFeed.ico" />
  </ItemGroup>

</Project>
```

6.8.10 Selbstkritik und Ausblick

Unser Programm präsentiert RSS-Dateien recht ansehnlich, z. B.:



Allerdings lässt sich das Anwendungsfenster nicht verschieben, während eine RSS-Datei geladen oder vergeblich nach einer fehlerhaften Adresse gesucht wird. Der für die Darstellung des Fensters zuständige UI-Thread (**U**ser **I**nterface – Thread) ist ausgelastet und kann nicht auf Anweisungen zur Änderung der Fensterposition reagieren. Mit der im Kapitel 17 von Baltes-Götz (2021) behandelten Multithreading-Programmierung lassen sich solche Probleme vermeiden (siehe speziell Abschnitt 17.2.3).

Den Projektordner mit dem RSS-Feed - Reader auf dem aktuellen Entwicklungsstand finden Sie hier:

...\BspUeb\WPF\RssFeedReader\BlockedUI

6.9 Übungsaufgaben zum Kapitel 6

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

`object o1 = 1;`
`object o2 = 1;`
`Console.WriteLine(o1 == o2);`
	`False`

2) Erstellen Sie ein Programm, das 6 Lottozahlen (von 1 bis 49) zufällig zieht und aufsteigend sortiert ausgibt.

Man kann das Problem, eine zufällige Teilmenge aus einem Array mit den Elementen 1 bis 49 zu ziehen, elegant mit dem folgenden Verfahren lösen:

- Bringe die Elemente des Arrays in eine zufällige Reihenfolge, wähle also eine zufällige Permutation.
- Wähle aus der Permutation die ersten 6 Elemente.

Eine zufällige Anordnung von Array-Elementen lässt sich mit dem **Fisher-Yates - Algorithmus** herstellen.[1] Dieser Algorithmus verwirbelt ein Array mit *n* Elementen in einer Schleife mit *n* – 1 Durchgängen:

- Im ersten Durchgang wird ein zufälliger Array-Index *k* aus dem Bereich von 0 bis *n* – 1 bestimmt. Dann werden die Array-Elemente *k* und *n* – 1getauscht. Im Ergebnis steht ein zufällig gewähltes Array-Element an der Position *n* – 1 (also an der letzten Position).
- Im zweiten Durchgang wird ein zufälliger Array-Index *k* aus dem Bereich von 0 bis *n* – 2 bestimmt. Dann werden die Array-Elemente *k* und *n* – 2 getauscht. Im Ergebnis steht ein zufällig aus den Elementen an den Positionen 0 bis *n* – 2 gewähltes Element an der Position *n* – 2.
- usw.

Das Sortieren eines Arrays ist mit der statischen Methode **Sort()** aus der Klasse **Array** im Namensraum **System** schnell erledigt. Um das Arbeiten mit Arrays zu üben und einen Eindruck von Sortieralgorithmen zu gewinnen, sollten Sie zum Sortieren jedoch das **Auswahlverfahren** (engl.: *Selection Sort*) verwenden, das bei einem Array mit *n* geordneten Elementen folgendermaßen abläuft:

- Für das Array mit den Elementen 0, …, *n* - 1 wird das Minimum gesucht und an den linken Rand befördert. Dann werden die Elemente 1, …, *n* - 1 analog behandelt, usw.
- Bei jeder Teilaufgabe muss man das kleinste Element eines Vektors an seinen linken Rand befördern, was auf die folgende Weise geschehen kann:
  - Man geht davon aus, das Element am linken Rand sei das kleinste (genauer: *ein* Minimum).
  - Es wird sukzessive mit seinen rechten Nachbarn verglichen. Ist das Element an der Position *i* kleiner, so tauscht es mit dem „Linksaußen“ seinen Platz.
  - Nun steht am linken Rand ein Element, das die anderen Elemente mit Positionen kleiner oder gleich *i* nicht übertrifft. Es wird nun sukzessive mit den Elementen an den Positionen ab *i* + 1 verglichen.
  - Nachdem auch das Element an der letzten Position mit dem Element am linken Rand verglichen worden ist, steht mit Sicherheit am linken Rand ein Element, zu dem sich kein kleineres findet.

---

[1] https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

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 ab 2 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 endet, 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 $n \le K$ mit echtem Teiler das beschriebene Streichverfahren überstanden. Mit zwei positiven Zahlen u, v würde dann gelten:

$$n = u \cdot v \text{ und } u \le \sqrt{K} \text{ oder } v \le \sqrt{K} \text{ (wegen } n \le K)$$

Wir nehmen ohne Beschränkung der Allgemeinheit $u \le \sqrt{K}$ 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 gezeigt, dass die Kandidatenrestmenge nur noch Primzahlen enthält.

Sollen z. B. alle Primzahlen kleiner oder gleich 18 bestimmt werden, so startet man mit der folgenden 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

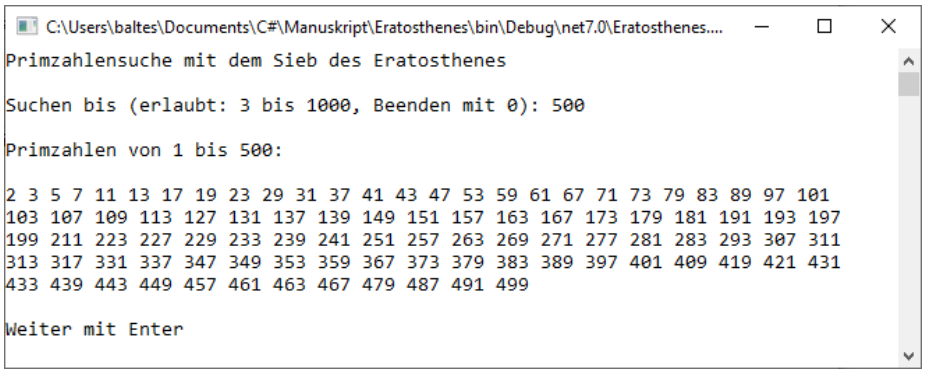
Als neue Basis wird die Zahl 3 gewählt (> 2, nicht gestrichen). Ihre echten Vielfachen werden gestrichen:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

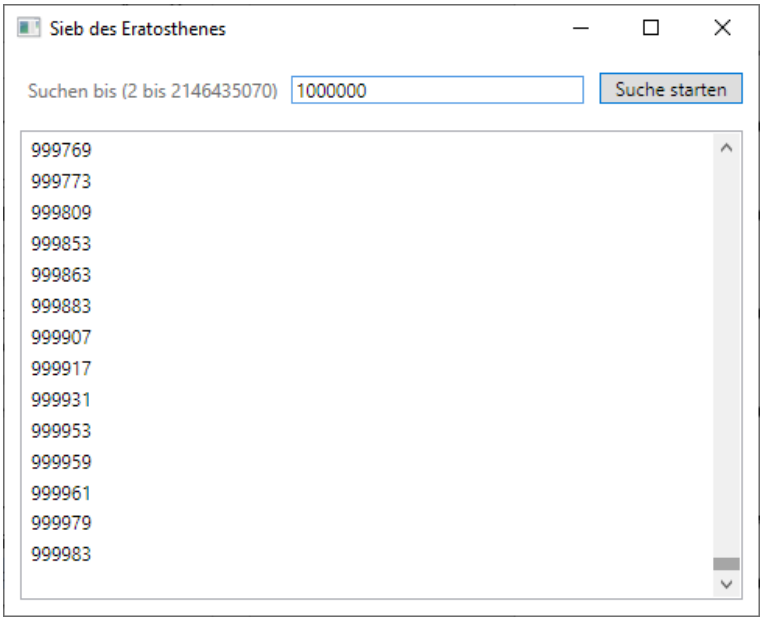
Als neue Basis wird die Zahl 5 gewählt (> 3, nicht gestrichen). Allerdings ist 5 größer als $\sqrt{18}$ ($\approx$ 4,24) und der Algorithmus daher bereits beendet. Als Primzahlen kleiner oder gleich 18 erhalten wir also:

2, 3, 5, 7, 11, 13 und 17

Alternativ können Sie ein Programm für die Konsole



oder eine WPF-Anwendung erstellen:



Der Lösungsvorschlag für die WPF-Anwendung zeigt eine starke Ähnlichkeit mit dem im Abschnitt 6.8 entwickelten RSS-Feed - Reader. Es werden vier Steuerelemente verwendet:

- ein **Label**-Objekt
  Es beschreibt die unterstützten Suchaufträge.
- ein **TextBox**-Objekt
  Hier können die Benutzer das Maximum für die Suche eintragen.
- ein **Button**-Objekt
  Damit wird die Suche gestartet.
- ein **ListBox**-Objekt
  Hier werden die gefundenen Primzahlen ausgegeben.

Hinweise zum Befüllen des **ListBox**-Steuerelements mit den gefundenen Primzahlen:

- Erstellen Sie ein Objekt der konkretisierten generischen Kollektionsklasse **List<int>**:
  ```
  var items = new List<int>();
  ```
- Zur Aufnahme einer Primzahl in die Liste eignet sich die Methode **Add()**, z. B.:
  ```
  items.Add(i);
  ```
- Weisen Sie die gefüllte **List<int>** - Kollektion der **ListBox**-Eigenschaft **ItemsSource** zu, z. B.:
  ```
  listBox.ItemsSource = items;
  ```

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 Initialisierungswert 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[]** (vgl. Abschnitt 4.7.2.3.2):
  ```
  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 eine Zeichenfolge (ein **String**-Objekt) auf die Konsole schreibt und dabei einen korrekten Zeilenumbruch vornimmt. Die Anwender Ihrer Methode sollen darüber entscheiden, ...

- wie breit eine Zeile werden darf,
- ob ein Bindestrich zwischen zwei Wörtern (z. B. „EU-Parlament“) als optionaler Trennstrich am Zeilenende zulässig ist.

Weitere Anforderungen an die Methode:

- Aufeinanderfolgende Leerzeichen sollen wie ein einzelnes Leerzeichen wirken.
- Ist ein Wort breiter als die Ausgabezeile, dann ist ein Umbruch *innerhalb* des Worts unvermeidlich.

Durch die folgenden Anweisungen auf oberster Ebene wird die Verwendung der Methode demonstriert:

```
string s = "Dieser Satz - bitte beachten - passt nicht in eine Schmal-Zeile, " +
           "die nur wenige Spalten umfasst.";
StringUtil.WrapLine(s, 30, true);
Console.WriteLine();
StringUtil.WrapLine(s, 50, true);
Console.WriteLine();
StringUtil.WrapLine(s, 40);
Console.WriteLine();
StringUtil.WrapLine(s);
```

Der erste Methodenaufruf sollte die folgende Ausgabe erzeugen:

```
Dieser Satz - bitte beachten -
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. Im folgenden Beispiel wird die Arbeitsweise von **Split()** demonstriert:

Quellcode	Ausgabe

`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, sodass z. B. ein als Trennzeichen definierter Bindestrich verloren geht. Das sollte nach Möglichkeit in Ihrem Programm verhindert werden.

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.

7 Vererbung und Polymorphie

7.1 Einführung

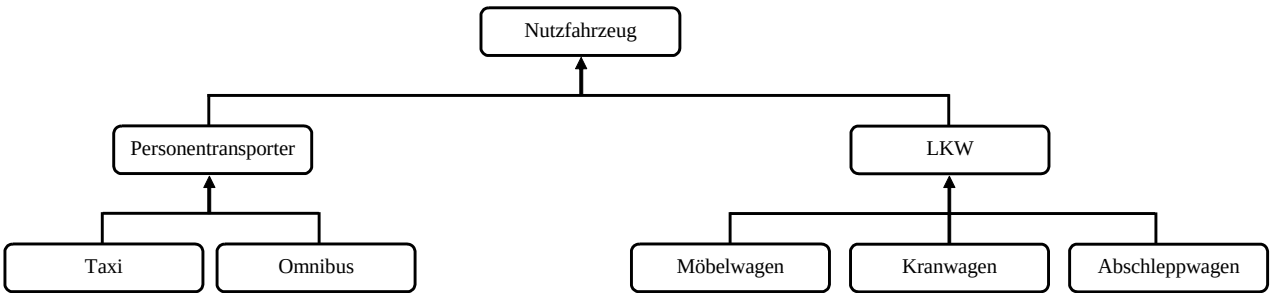
Im Manuskript war schon mehrfach davon die Rede, dass die .NET - Datentypen in eine strenge Abstammungshierarchie eingeordnet sind. Nun betrachten wir die Vererbungsbeziehung zwischen den Klassen im Detail. Die Vererbung gilt als eine von drei Säulen der objektorientierten Programmierung. Sie wird nicht nur in der BCL intensiv und erfolgreich eingesetzt, sodass eine gründliche Behandlung angemessen ist.

Von den drei Säulen der objektorientierten Programmierung wird eine weitere, nämlich die Polymorphie, wegen ihrer starken Bezüge zur Vererbung ebenfalls in diesem Kapitel beschrieben.

Obwohl die Vererbungstechnik in der OOP eine tragende Rolle spielt und viele Vorteile bringt (z. B. bei der Modellierung des Gegenstandsbereichs, beim Software-Recycling) muss auch von einigen Komplikationen und Tücken berichtet werden. Im weiteren Verlauf des Manuskripts werden Alternativen zur Vererbung vorgestellt. Z. B. kann die Funktionalität einer Klasse modifiziert werden, indem einem Feld mit Delegatentyp eine kompatible Methode zugewiesen wird (siehe Kapitel 10). Wenn die Klasse ihr Delegatenobjekt aufruft, kommt die „injizierte“ Methode zum Einsatz. Diese Option zur Verhaltenskonfiguration ist oft flexibler als die Definition von abgeleiteten Klassen.

7.1.1 Modellierung realer Klassenhierarchien

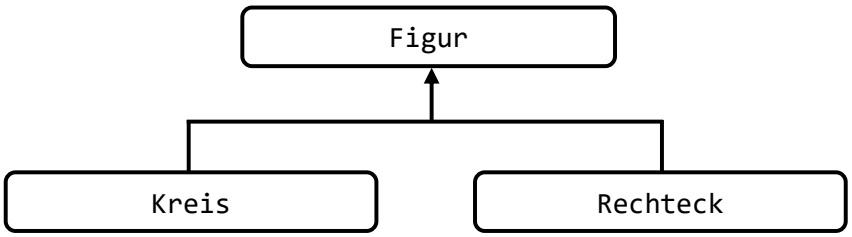
Beim Modellieren eines Gegenstandbereichs durch Klassen, die durch Merkmale (Instanz- und Klassenvariablen) sowie Handlungskompetenzen (Instanz- und Klassenmethoden) gekennzeichnet sind, müssen auch die Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen real existierenden Klassen abgebildet 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 Verwaltung von Fahrzeugen und Einsätzen sollte diese reale Klassenhierarchie abbilden.

7.1.2 Übungsbeispiel

Bei unseren Beispielprogrammen bewegen wir uns in einem bescheideneren Rahmen und betrachten meist eine einfache Hierarchie mit Klassen für geometrische Figuren:



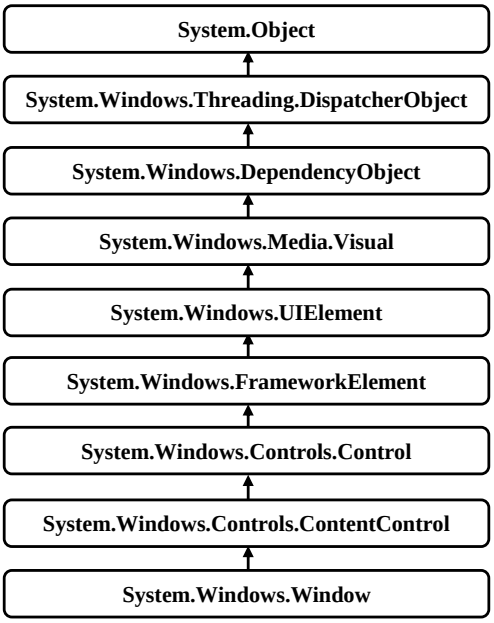
Vielleicht haben manche Leser als Gegenstück zum Rechteck (auf derselben Hierarchieebene) die *Ellipse* erwartet, die ebenfalls zwei ungleiche lange „Seiten“ besitzt. Weiterhin liegt es auf den ersten Blick nahe, den Kreis als Spezialisierung der Ellipse und das Quadrat als Spezialisierung des Rechtecks zu betrachten. Wir werden aber im Abschnitt 7.14 über das *Liskovsche Substitutionsprinzip* genau diese Ableitungen (von Kreis aus Ellipse bzw. von Quadrat aus Rechteck) kritisieren. Man spricht hier vom *Kreis-Ellipse* - oder *Quadrat-Rechteck - Problem*.[1] Es ist wohl akzeptabel, an Stelle der Ellipse den Kreis neben das Rechteck zu stellen, um das Erlernen der neuen Konzepte durch ein möglichst einfaches Beispiel *ohne* Verstoß gegen das Liskovsche Substitutionsprinzip zu erleichtern.

7.1.3 Vererbung in der OOP

In objektorientierten Programmiersprachen ist es weder sinnvoll noch erforderlich, jede Klasse einer Hierarchie komplett neu zu definieren. Man geht stattdessen 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 kann nach Bedarf Anpassungen bzw. Erweiterungen zur Lösung spezieller Aufgaben vornehmen, z. B.:

- zusätzliche Felder deklarieren
- zusätzliche Methoden, Eigenschaften oder Indexer definieren
- geerbte Methoden ersetzen, d.h. unter Beibehaltung der Signatur umgestalten

Die BCL 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:



[1] Siehe z. B. https://en.wikipedia.org/wiki/Circle-ellipse_problem

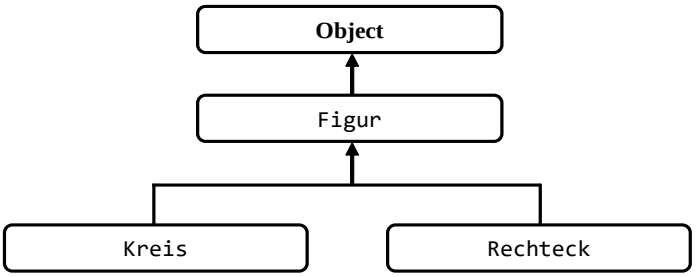
Wir haben bereits mehrere WPF-Anwendungen erstellt und dabei die Klasse **Window** als Basisklasse für die Ableitung einer projektspezifischen Klasse namens `MainWindow` verwendet (siehe z. B. Abschnitt 6.8.4).

7.1.4 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 Software-Systeme entstehen, die gleichzeitig stabil und innovationsoffen sind (vgl. Abschnitt 5.1.1.3 zum Open-Closed - Prinzip). Die nicht selten anzutreffende Praxis, vorhandenen Code per *Copy & Paste* in neue Projekte bzw. Klassen zu übernehmen, hat gegenüber der Nutzung und Erweiterung einer sorgfältig geplanten Klassenhierarchie offensichtliche Nachteile. Natürlich kann C# nicht garantieren, dass jede Klassenhierarchie exzellent entworfen ist und langfristig von einer stetig wachsenden Entwicklergemeinde eingesetzt wird.

7.2 Allgemeines .NET - Typsystem

Alle .NET - Klassen stammen von der Klasse **Object** im Namensraum **System** ab. Das gilt sowohl für die in der BCL enthaltenen als auch für die von Anwendungsentwicklern definierten Klassen. 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 6.1) 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 (Enumerationen), die von der Klasse **System.Enum** (mit der Basisklasse **System.ValueType**) abstammen (siehe Abschnitt 6.4)
- sowie für die Tupel, die ebenfalls auf **System.ValueType** basieren (siehe Abschnitt 6.6).

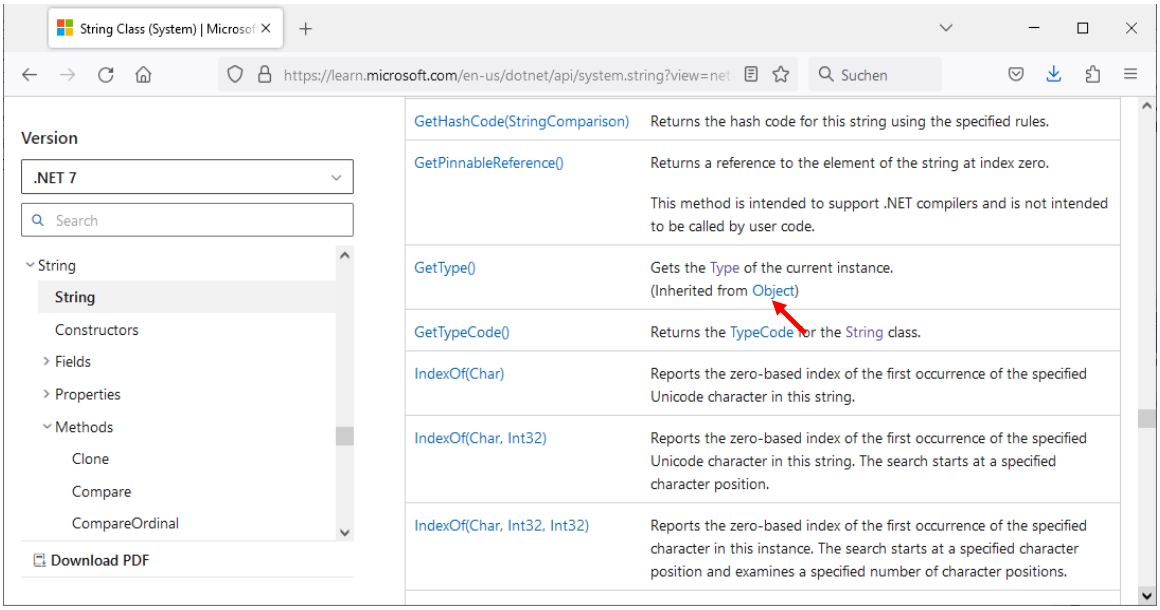
Jeder Typ erbt alle Merkmale und Handlungskompetenzen aus der eigenen Abstammungslinie von der Urahnklasse **Object** beginnend. Folglich kann z. B. jedes Objekt und jede Strukturinstanz die in der Urahnklasse definierte Methode **GetType()** ausführen, die ein auskunftsfreudiges **Type**-Objekt liefert. Im folgenden **WriteLine()** - Aufruf verraten drei **Type**-Objekte (über die implizit aufgerufene Methode **ToString()**) die Typbezeichnung samt Namensraum:

Quellcode	Ausgabe

`var o = new Object();`
`var s = "abc";`
`var i = 13;`
`Console.WriteLine(o.GetType() + "\n" +`
`s.GetType() + "\n" +`
`i.GetType());`
	`System.Object`
	`System.String`
	`System.Int32`

Genaugenommen kann die Strukturinstanz `i` die Methode **GetType()** nicht ausführen, weil dazu ein „echtes“ Objekt erforderlich ist. Die als *Boxing* bezeichnete Technik sorgt dafür, dass eine Strukturinstanz bei Bedarf automatisch in ein Objekt einer passenden Hilfs- bzw. Hüllenklasse verpackt wird (vgl. Abschnitt 6.1.6). Das geschieht zwar transparent, doch darf wegen des erheblichen Aufwands die Anzahl der Boxing-Operationen nicht zu groß werden.

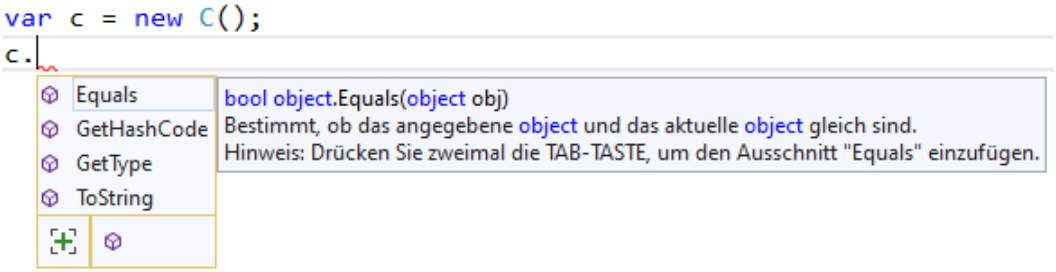
In der BCL-Dokumentation zu einem Datentyp sind seine Erbstücke gekennzeichnet, z. B. bei der Klasse **String**:



Auch die Objekte der folgenden Klasse

```
class C { }
```

beherrschen immerhin vier, von **Object** geerbte Methoden:



Die Methoden **Equals()**, **GetHashCode()** und **ToString()** sind allerdings in der Klasse **Object** nur rudimentär implementiert, sodass oft das Überschreiben durch eine eigene Implementation sinnvoll ist (siehe Abschnitt 7.9):

- Die **Object**-Methode **ToString()** liefert lediglich den Typnamen, kann also einzelne Instanzen *nicht* charakterisieren.
- Die Methode **Equals()** beurteilt die Gleichheit von zwei Objekten über deren Speicheradressen. Oft ist eine Gleichheitsbeurteilung anhand des Inhalts erwünscht (wie z. B. bei der Klasse **String**).
- Die Methode **GetHashCode()**, die zu einem Objekt einen möglichst eindeutigen **int**-Wert liefern soll, orientiert sich an der Speicheradresse.

Wenn Objekte eines Typs durch Kollektionsklassen wie **HashSet<T>** oder **Dictionary<K, V>** verwaltet werden sollen, dann sind Überschreibungen der **Object**-Methoden **Equals()** und **GetHashCode()** erforderlich (siehe Kapitel 11).

In der Definition der Klasse **String** sind die **Object**-Methoden **Equals()**, **GetHashCode()** und **ToString()** überschrieben, sodass diese Methoden in der BCL-Dokumentation zur Klasse **String** *nicht* als Erbstücke ausgewiesen sind.

Nur für Klassen ist es in C# möglich, ...

- in der eigenen Definition eine Basisklasse anzugeben,
- bei der Definition anderer Klassen als Basistyp benannt zu werden.

Als Basisklasse für eine Record-Klasse sind nur **Object** oder eine andere Record-Klasse erlaubt, und aus einer Record-Klasse kann nur eine andere Record-Klasse abgeleitet werden (siehe Abschnitt 6.7.4).

Strukturen, Enumerationen, Tupel und Record-Strukturen ...

- haben eine implizite, fest vorgegebene Basisklasse
- und können nicht beerbt werden.

Strukturen und Record-Strukturen dürfen allerdings Schnittstellen implementieren, und eine implementierte Schnittstelle kann manchmal eine ähnliche Rolle spielen wie eine Basisklasse (z. B. bei der Polymorphie, siehe Abschnitt 9.3).

7.3 Definition einer abgeleiteten Klasse

Wir definieren im angekündigten Beispiel zunächst die Basisklasse `Figur`, die private Instanzvariablen für die X- und die Y-Position der linken oberen Ecke einer zweidimensionalen Figur, zwei Konstruktoren sowie eine Methode `Wo()` zur Positionsmeldung besitzt:[1]

```
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 im Definitionskopf hinter dem Klassennamen durch Doppelpunkt getrennt den Namen der Basisklasse angeben:

---

[1] Die folgende Parametervalidierung im parametrisierten `Figur`-Konstruktor

```
if (x >= 0.0 && y >= 0.0) {
      xpos = x;
      ypos = y;
   }
```

Ist durch das verwendete Bildschirmkoordinatensystem begründet:

- In der linken oberen Ecke befindet sich die Position (0,0; 0,0).
- Die X-Koordinaten wachsen von links nach rechts.
- Die Y-Koordinaten wachsen von oben nach unten.

```
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 Positionsfelder 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 C# werden auch **private Member** vererbt, doch hat eine abgeleitete Top-Level - Klasse keinen direkten Zugriff auf diese Member.[1] Sie kann also z. B. ...

- weder direkt auf private Felder der Basisklasse zugreifen,
- noch private Methoden der Basisklasse aufrufen.

In der `Kreis`-Klasse wird (wie in der Basisklasse `Figur`) auch ein parameter*freier* Konstruktor definiert. Vielleicht hat jemand erwartet, die `Kreis`-Klasse würde den parameterfreien Konstruktor ihrer Basisklasse (bei automatischer Anpassung des Namens) übernehmen. Konstruktoren werden jedoch grundsätzlich *nicht* vererbt. Ein Konstruktor hat die Aufgabe, ein Objekt für seinen Einsatz vorzubereiten, indem er vor allem die Felder initialisiert. Weil ein Basisklassenkonstruktor zusätzliche Felder von abgeleiteten Klassen nicht kennt, ist ein Vererben von Konstruktoren an abgeleitete Klassen nicht sinnvoll. Ihre Konstruktoren muss eine abgeleitete Klasse also neu definieren, wobei es aber möglich ist, über das Schlüsselwort **base** einen Basisklassenkonstruktor (z. B. zur Initialisierung von geerbten Instanzvariablen) einzuspannen (siehe Beispiel und Abschnitt 7.4).

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 (zum sogenannten *Deadly Diamond of Death* siehe Kreft & Langer 2014).

Einen gewissen Ersatz bieten die im Kapitel 9 behandelten Schnittstellen (Interfaces), weil ...

- eine Klasse mehrere Schnittstellen implementieren darf,
- und außerdem eine Schnittstelle mehrere vorhandene Schnittstellen beerben (besser: erweitern) darf.

---

[1] Ist eine abgeleitete Klasse in ihre Basisklasse *eingeschachtelt* (siehe Abschnitt 5.10), dann kann sie auf die privaten Member der Basisklasse zugreifen. Diese Zugriffsrechte basieren aber nicht auf der Vererbungsbeziehung, sondern sind bei jeder eingeschachtelten Klasse vorhanden (siehe Abschnitt 5.10).

Statische Member werden in C# genauso vererbt wie instanzbezogene.

7.4 base-Konstruktoren und Initialisierungs-Sequenzen

Zwar werden Konstruktoren nicht vererbt, doch ist bei der Entstehung einer Instanz eines abgeleiteten Typs aus *jeder* Basisklasse entlang der Ahnenreihe ein Konstruktor durch einen impliziten oder expliziten Aufruf beteiligt. Im folgenden Beispiel entsteht ein Objekt der Klasse `Figur` und ein Objekt der von `Figur` abgeleiteten Klasse `Kreis`, wobei die beteiligten Konstruktoren ihre Tätigkeit melden:

Quellcode	Ausgabe

`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 BCL-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 im Abschnitt 7.3 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 automatisch den *parameterfreien* 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 erfor-
dert 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 parameterfreie 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**-Zugriffsmodifikator).

Beim Erzeugen eines Unterklassenobjekts laufen die folgenden 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 IL-Code zu einem Feldinitialisierer 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 gemäß Deklaration initialisieren, Aufruf eines Basisklassenkonstruktors, Anweisungsteil des Konstruktors).
- Das Verfahren wird bis zur obersten Hierarchieebene fortgesetzt, wobei natürlich der **Object**-Konstruktor keinen Basisklassenkonstruktor aufruft.

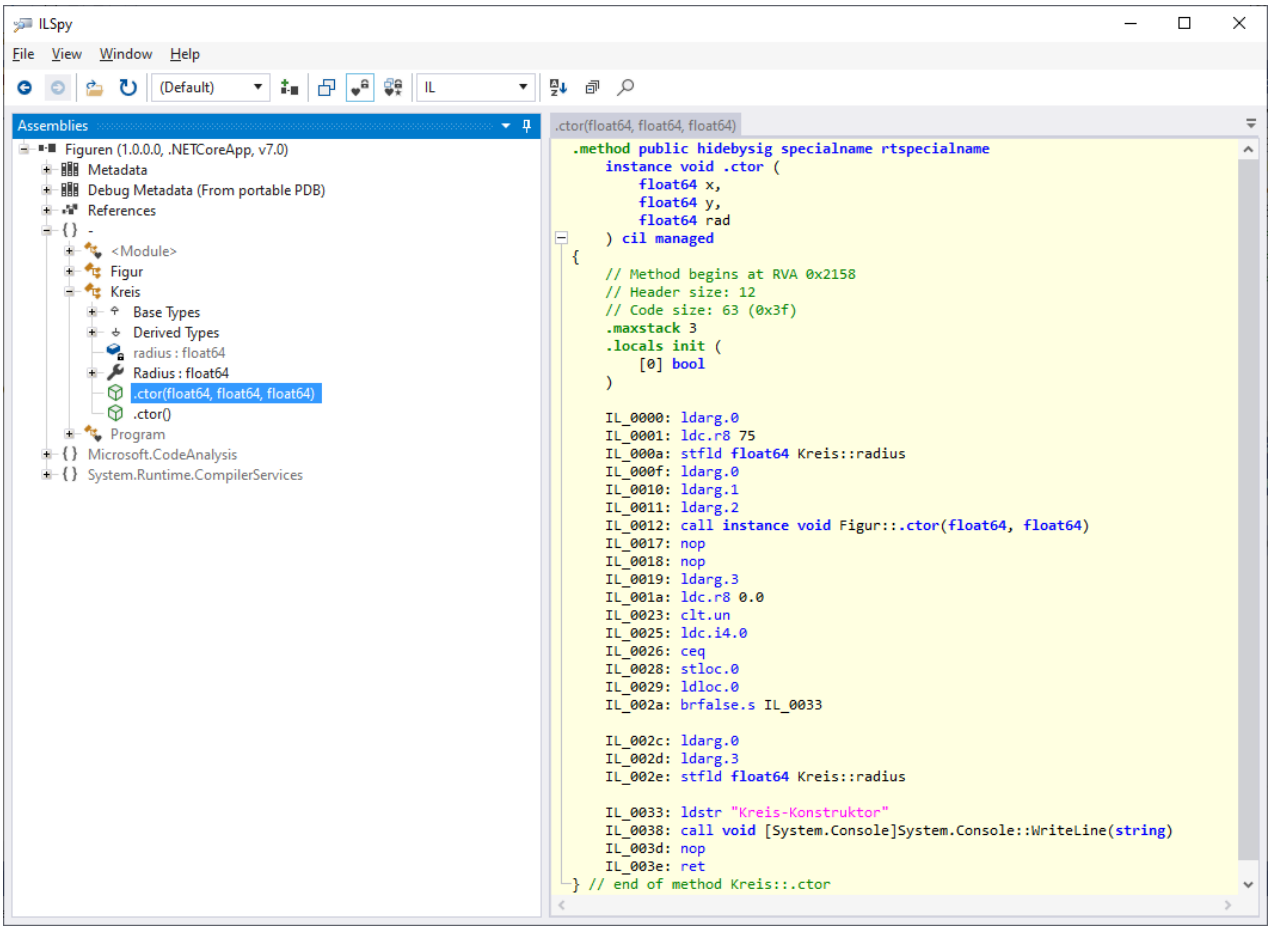
Betrachten wir z. B., was beim Erzeugen eines `Kreis`-Objekts mit dem Konstruktor-Aufruf

```
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. Den dazu erforderlichen IL-Code hat der Compiler automatisch erstellt.
- 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. Den dazu erforderlichen IL-Code hat der Compiler automatisch erstellt.
- Der parameterlose **Object**-Konstruktor startet, tut aber nichts:
  ```
  public Object() { }
  ```
  Es gibt auch keine Instanzvariablen in der **Object**-Klassendefinition, die einen Initialisierungswert erhalten könnten.
- 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.

Wie eine Inspektion der Klasse `Kreis` mit dem Programm ILSpy zeigt, haben die beiden Konstruktoren den Namen **.ctor**:



Im IL-Code des parametrisierten `Kreis`-Konstruktors ist die oben beschriebene Sequenz zu sehen:

- Instanzvariable `radius` gemäß Deklaration initialisieren
- Aufruf des parametrisierten `Figur`-Konstruktors
- Anweisungsteil des `Kreis`-Konstruktors

Für die automatische Null-Initialisierung (vgl. Abschnitt 5.2.3) sorgt die CLR, sodass dazu kein IL-Code erforderlich ist.

Neben den Instanzkonstruktoren werden auch die folgenden Klassenmitglieder *nicht* vererbt:

- statische Konstruktoren (siehe Abschnitt 5.7.4)
- Finalisierer (siehe Abschnitt 5.4.5)

Der Visual Studio - Projektordner mit dem Figurenbeispiel auf dem aktuellen Entwicklungsstand ist hier zu finden:

...\BspUeb\Vererbung und Polymorphie\Figuren

7.5 Zugriffsmodifikator protected

Auf **private**-Member einer Basisklasse haben die Methoden einer abgeleiteten Klasse *keinen* direkten Zugriff. Hier besteht kein Unterschied zu den Methoden einer beliebigen anderen Klasse. Um den abgeleiteten Klassen besondere Rechte einzuräumen, bietet C# den Zugriffsmodifikator **protected**. Auf Member mit diesem Zugriffsschutz dürfen neben den Methoden der eigenen Klasse auch die Methoden von (direkt oder indirekt) *abgeleiteten* Klassen direkt zugreifen, 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()` verändert werden. Diese Methode verschiebt die obere linke Ecke der Figur in das aktuelle Zentrum:

```
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* direkten Zugriff.

Für die Methoden *fremder* Klassen sind die Member mit dem Zugriffsschutz **protected** ebenso unerreichbar wie die Member mit dem Zugriffsschutz **private**, z. B.:

```
using System;
class Prog {
   static void Main() {
      Kreis krs = new (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 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!");
}
```

Der Visual Studio - Projektordner mit dem Figurenbeispiel auf dem aktuellen Entwicklungsstand ist hier zu finden:

...\BspUeb\Vererbung und Polymorphie\protected

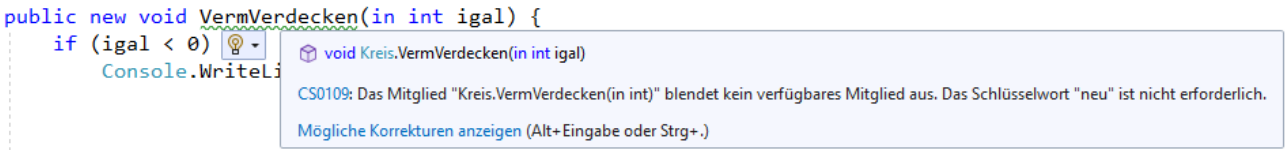
7.6 Erbstücke durch spezialisierte Varianten verdecken

Das schon mehrfach erwähnte, für die Polymorphie relevante *Überschreiben* von Methoden ist noch nicht dran (siehe Abschnitt 7.9). Während eine überschriebene Methode auch dann zum Einsatz kommt, wenn ein Objekt einer abgeleiteten Klasse durch eine Referenzvariable vom Typ einer Basisklasse angesprochen wird, wird eine verdeckende Methode nur bei Verwendung einer Referenz vom eigenen Typ ausgeführt.

7.6.1 Methoden und andere ausführbare Member verdecken

Eine geerbte Basisklassenmethode kann in einer abgeleiteten Klasse durch eine Methode mit gleicher Signatur verdeckt (ausgeblendet) werden. Zwei Methoden haben genau dann dieselbe Signatur, wenn die Namen und die Parameterlisten (hinsichtlich Datentyp sowie Transfermodus und -richtung aller Formalparameter) übereinstimmen, während z. B. die Rückgabetypen keine Rolle spielen (vgl. Abschnitte 5.3.5 und 8.5).

Während abweichende Richtungsmodifikatoren zu einem Verweisparameter beim Überladen von Methoden nicht signaturrelevant sind (siehe Abschnitt 5.3.5), sorgen sie beim Verdecken (und beim Überschreiben) von Methoden für abweichende Signaturen.[1] Aufgrund der Erfahrung mit dem Überladen könnte man vermuten, dass eine Unterklassenmethode eine Basisklassenmethode trotz abweichender Richtungsmodifikatoren verdecken kann. Das ist aber nicht der Fall, und der Compiler macht durch eine Warnung auf den Fehlversuch aufmerksam, z. B.:



Weil beim Methodenaufruf die Richtungsmodifikatoren **ref** und **out** ggf. angegeben werden müssen, sollten sich aus dem Missverständnis keine gefährlichen Konsequenzen ergeben. Es bleibt aber ein Risiko, wenn in der abgeleiteten Klasse z. B. neben der vermeintlich verdeckenden Methode

```
public new void VermVerdecken(in int igal)
```

auch die Basisklassenmethode existiert:

```
public void VermVerdecken(ref int igal)
```

Wenn ein Programmierer von einer erfolgreichen Verdeckung ausgeht, ein Objekt aus der Unterklasse erzeugt und beim Methodenaufruf den Richtungsmodifikator aus der Basisklasse verwendet, dann wird unerwartet die Basisklassenmethode ausgeführt.

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:[2]

---

[1] Um die Missverständnisse im Zusammenhang mit dem Begriff *Signatur* einzudämmen, sollte zwischen der *Überladungssignatur* und der *Ersetzungssignatur* unterschieden werden.

[2] 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; 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.

```
using System;
public class Kreis : Figur {
  . . .
  public new void Wo() {
    base.Wo();
    Console.WriteLine($"Unten Rechts:\t({xpos + 2.0*radius}; {ypos + 2.0*radius})");
  }
}
```

Ist die Deklaration einer *verdeckenden* Methode tatsächlich intendiert, sollte mit dem Modifikator **new** im Definitionskopf die folgende Warnung des Compilers vermieden werden:[1]

```
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 dieser Warnung will der Compiler ein ungeplantes Verdecken verhindern.

Im Anweisungsteil der neuen Methode kann man sich oft durch Rückgriff auf die verdeckte Basisklassenmethode die Arbeit erleichtern, wobei erneut das Schlüsselwort **base** zum Einsatz kommt, z. B.:

```
base.Wo();
```

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* (vgl. Abschnitt 5.3.5).

Im folgenden Beispiel erhalten ein `Figur`- und ein `Kreis`-Objekt jeweils die Nachricht `Wo()`, und die beiden Objekte zeigen jeweils ihr artspezifisches Verhalten:

Quellcode	Ausgabe

`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)`

Im nächsten Beispiel wird ein `Kreis`-Objekt zweimal aufgefordert, die Methode `Wo()` auszuführen. Im ersten Aufruf wird eine Referenzvariable vom Typ `Kreis` verwendet und im zweiten Aufruf eine Referenzvariable vom Typ `Figur`. Das Objekt führt beim ersten Aufruf die `Kreis`-Methode aus und im zweiten Aufruf die `Figur`-Methode:

Quellcode	Ausgabe

`var k = new Kreis(50.0, 50.0, 10.0);`
`k.Wo();`
`Figur f = k;`
`f.Wo();`
	`Oben Links:  (50, 50)`
	`Unten Rechts:  (70, 70)`
	
	`Oben Links:  (50, 50)`

Wenn in einem Programm für dasselbe Objekt, das z. B. einen Raketenmotor modelliert, Referenzvariablen unterschiedlichen Typs existieren, dann sorgen verdeckende Methoden für ein Fehlerrisiko, weil das Verhalten des Objekts von der verwendeten Referenzvariablen abhängt.

Der Visual Studio - Projektordner mit dem Figurenbeispiel auf dem aktuellen Entwicklungsstand ist hier zu finden:

...\BspUeb\Vererbung und Polymorphie\Methoden verdecken

---

[1] Dieser Modifikator darf nicht mit dem *Operator* **new** verwechselt werden.

Wird eine öffentliche Basisklassenmethode durch eine *private* Unterklassenmethode verdeckt, dann wird die verdeckende Methode nur *klassenintern* verwendet, und im API der abgeleiteten Klasse bleibt das signaturgleiche Erbstück mit **public**-Zugriff 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 a = new Spezialklasse();
        a.NenneTyp();
        a.DeinTyp();
    }
}
```

```
Basisklasse
Spezialklasse
```

Im Beispiel fordert die **Main()** - Methode der Klasse `Prog` ein Objekt der `Spezialklasse` auf, die Methode `NenneTyp()` auszuführen. Dabei wird die Methode aus der `Basisklasse` ausgeführt, weil die Methode `NenneTyp()` in der Spezialklasse als **private** deklariert ist. Wenn hingegen in der öffentlichen `Spezialklassen`-Methode `DeinTyp()` die Methode `NenneTyp()` aufgerufen wird, dann kommt die Version der `Spezialklasse` zum Einsatz. Dabei wurde in beiden Fällen eine Referenzvariable vom Typ der `Spezialklasse` verwendet.

Neben objektbezogenen können auch *statische* Methoden verdeckt werden, wobei die verdeckte Basisklassenvariante durch Voranstellen des Klassennamens ansprechbar bleibt, 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
```

Neben Methoden können auch andere ausführbare Member verdeckt werden: Eigenschaften, Indexer und die im Abschnitt 10.2 behandelten Ereignisse. Generell sollte das Verdecken von ausführbaren Membern vermieden werden, weil es die Lesbarkeit des Quellcodes erschwert.

Das im Abschnitt 7.9 erläuterte *Überschreiben* von Methoden ist insofern unproblematisch, als ein Objekt aus einer abgeleiteten Klasse unabhängig vom deklarierten Typ der zur Kommunikation verwendeten Referenzvariablen immer das in der eigenen Klasse definierte Verhalten zeigt. Allerdings ist das Überschreiben mit Risiken verbunden, wenn der Designer der abgeleiteten Klasse …

- die Basisklasse nicht genau kennt
- oder über eine Änderung der Basisklasse nicht informiert wird.

7.6.2 Felder verdecken

Auch geerbte Instanz- und Klassenvariablen lassen sich verdecken (ausblenden), was aber im Sinne eines gut nachvollziehbaren Quellcodes 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

```
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();
      ako.BM();
      ako.AM();
   }
}
```

Ausgabe

```
x in BK-Methode:        Bast
x in AK-Methode:        333
base-x in AK-Methode:   Bast
```

Ist die Deklaration einer verdeckenden Variablen tatsächlich intendiert, sollte der Modifikator **new** angeben werden, um die folgende Warnung des Compilers zu vermeiden:[1]

[1] Dieser Modifikator darf nicht mit dem *Operator* **new** verwechselt werden.

```
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 Hinweis will der Compiler ein ungeplantes Verdecken verhindern. In der Regel sollte der Hinweis zum Anlass genommen werden, die Namenskollision durch eine Umbenennung zu beseitigen, um die Lesbarkeit des Quellcodes zu verbessern.

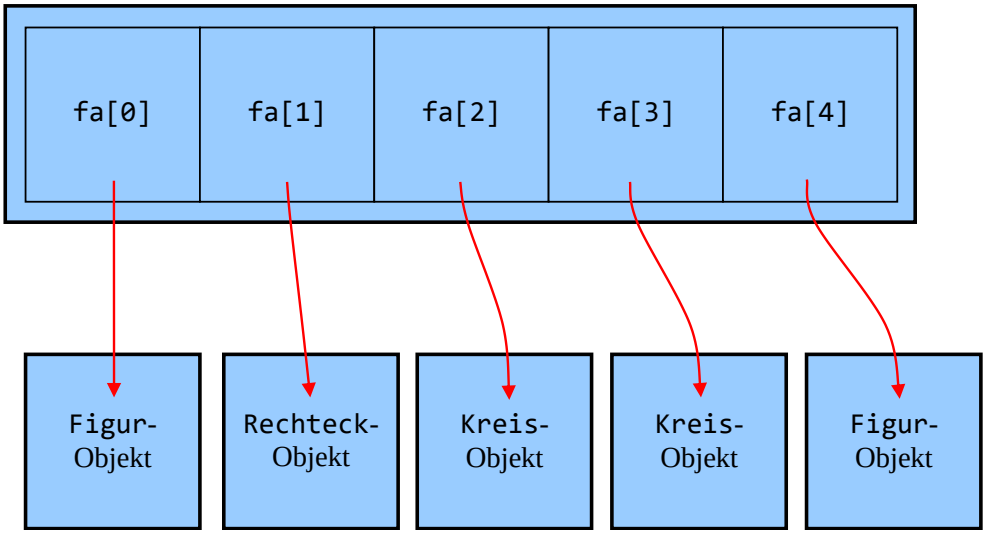
7.7 Verwaltung von Objekten über Basisklassenreferenzen

Eine Basisklassen-Referenzvariable darf die Adresse eines beliebigen Unterklassenobjekts aufnehmen. Schließlich besitzt Letzteres die komplette Ausstattung der Basisklasse und kann z. B. dort definierte Methoden ausführen. Ein Objekt steht nicht nur zur eigenen Klasse in der „ist-ein“ - Beziehung, sondern erfüllt diese Relation auch in Bezug auf die direkte Basisklasse sowie in Bezug auf alle indirekten Basisklassen in der Ahnenreihe. Angewendet auf das Beispiel im Abschnitt 7.3 kann man sagen, dass jeder `Kreis` auch eine `Figur` und ein **Object** 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 einer Unterklassen-Referenzvariablen zuzuweisen.

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`



Im folgenden Beispiel wird die abgebildete Konstellation realisiert:

```
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();
```

Die Elemente im Array `fa` werden aufgefordert, die in der Klasse `Figur` definierte Methode `Wo()` auszuführen (vgl. Abschnitte 7.3 und 7.6.1). Bei Ansprache per *Basisklassenreferenz* führen die Objekte aus den Klassen `Kreis` und `Rechteck` die Basisklassenvariante der `Wo()` - Methode aus, obwohl eine *verdeckende*, artspezifische `Wo()` - Methode vorhanden ist:

```
Oben Links:     (10, 10)

Oben Links:     (20, 20)

Oben Links:     (30, 30)

Oben Links:     (40, 40)

Oben Links:     (50, 50)
```

Der Vollständigkeit halber wird die Definition der Klasse `Rechteck` nachgeliefert:

```
public class Rechteck : Figur {
   double breite = 50.0, hoehe = 50.0;

   public Rechteck(double x, double y, double b, double h)  : base(x, y) {
      if (breite >= 0.0 && hoehe >= 0.0) {
         breite = b;
         hoehe = h;
      }
   }
   public Rechteck() { }

   public new void Wo() {
      base.Wo();
      Console.WriteLine($"Unten Rechts:\t({xpos + breite}; {ypos + hoehe})");
   }
}
```

Im Abschnitt 7.9 über die *Polymorphie* werden Sie erfahren, dass man in der Basisklasse eine *virtuelle* Methode definieren und diese in den abgeleiteten Klassen *überschreiben* muss, damit Objekte aus abgeleiteten Klassen auch bei der Ansprache per Basisklassenreferenz ein artspezifisches Verhalten zeigen.

Der Visual Studio - Projektordner mit dem Figurenbeispiel auf dem aktuellen Entwicklungsstand ist hier zu finden:

...\BspUeb\Vererbung und Polymorphie\Basisklassenreferenzen

7.8 Typtest-Operatoren

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, dann tritt ein Ausnahmefehler vom Typ **InvalidCastException** 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 meist auf Werttypen angewendet, sie spielt aber auch bei Referenztypen eine wichtige Rolle. Welche expliziten Konvertierungen erlaubt sind, ist der C# - Sprachspezifikation (ECMA 2022, Abschnitt 10.3) zu entnehmen. Im konkreten Fall wird der deklarierte Typ `Figur` durch die Spezialisierung bzw. Ableitung `Kreis` ersetzt. Der Compiler erlaubt die Konvertierung, übernimmt jedoch keine Verantwortung dafür.

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);
}
```

Weil der **is**-Operator nicht den deklarierten (statischen) Typ prüft, sondern den Laufzeittyp (den dynamischen Typ), resultiert bei einer Referenzvariablen mit dem Wert **null** immer das Ergebnis **false**, z. B.:

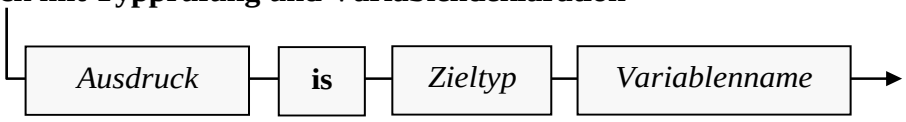
Quellcode	Ausgabe
`Kreis krs = null;` `Console.WriteLine(krs is Kreis);`	`False`

Seit C# 7.0 kann man in einem logischen Ausdruck mit **is**-Operator eine lokale Variable von dem zu verifizierenden Typ deklarieren und nach einer erfolgreichen Prüfung mit dem zu prüfenden Ausdruck initialisieren, sodass die explizite Typumwandlung überflüssig wird, z. B.:

```
foreach (Figur e in fa) {
   e.Wo();
   if (e is Kreis kr)
      Console.WriteLine("Radius:         " + kr.Radius);
}
```

Im folgenden Syntaxdiagramm wird der **is**-Ausdruck mit der Deklaration und Initialisierung einer lokalen Variablen nach erfolgreicher Typprüfung auf dem Stand von C# 7.0 beschrieben:

is-Ausdruck mit Typprüfung und Variablendeklaration



Das Prüfergebnis des **is**-Operators ist positiv, wenn der *Laufzeittyp* des Ausdrucks ...

- exakt den Zieltyp hat,
- oder vom Zieltyp abgeleitet ist,
- oder das als Zieltyp angegebene Interface implementiert.

In C# 8 und 9 profitiert der **is**-Ausdruck von den erweiterten Musterabgleich-Optionen, die im Abschnitt 15.2 behandelt werden. Im folgenden Beispiel wird für ein Objekt geprüft, ob es sich um eine `Kreis`-Instanz mit dem `Radius` 5 handelt:

```
Console.WriteLine(krs is Kreis { Radius: 5.0 });
```

Eine weitere Möglichkeit zur Vermeidung der expliziten Typumwandlung

```
((Kreis)e).Radius
```

bietet der **as**-Operator:

```
(e as Kreis).Radius
```

Die wichtigsten Regeln für den **as**-Operator sind:

- Der Zieltyp im rechten Operanden muss ein Referenztyp oder ein **null**-fähiger Werttyp sein (siehe Abschnitt 8.3). 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 linken Operanden verlangt der Compiler einen Ausdruck, dessen Wert potenziell 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 (ECMA 2022, Abschnitt 11.11.12).
- Stellt sich zur Laufzeit eine Konvertierung als unmöglich heraus, dann liefert der **as**-Operator im Unterschied zum gewohnten Typumwandlungsoperator *keinen* Ausnahmefehler vom Typ **InvalidCastException**, 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
  ```

  Im obigen Beispiel

  ```
  (e as Kreis).Radius
  ```

  kommt es zum Laufzeitfehler vom Typ **NullReferenceException**, wenn die Variable `e` *nicht* auf einen `Kreis` zeigt, weil die `Radius`-Eigenschaftsanfrage nicht an **null** gerichtet werden kann.

Weil der **is**-Operator einen impliziten **null**-Test vornimmt, ist er meist gegenüber dem **as**-Operator zu bevorzugen. Um die obige **if**-Anweisung

```
if (e is Kreis kr)
    Console.WriteLine("Radius:         " + kr.Radius);
```

äquivalent mit Hilfe des **as**-Operators zu formulieren, ist mehr Schreibaufwand erforderlich:

```
if (e != null)
    Console.WriteLine("Radius:         " + (e as Kreis).Radius);
```

Lahres & Rayman (2009, Abschnitt 5.1.5) bewerten das explizite Konvertieren einer Referenzvariablen in einen spezielleren Typ als Notlösung, die durch ein gutes Programmdesign vermieden werden sollte. Sofern man nicht den Quellcode aller Klassen unter Kontrolle hat, ist die Notlösung kaum perfekt zu vermeiden.

Konsistent mit dem **is**-Operator, der nicht den deklarierten (statischen) Typ prüft, sondern den Laufzeittyp (den dynamischen Typ), liefert die schon in der Urahnklasse **Object** definierte Methode **GetType()** den Laufzeittyp ihres Parameters. Im folgenden Beispiel

```
object[] oar = new object[3] { new object(), "123", 13 };
foreach(var o in oar)
    Console.WriteLine($"{o, 15} hat den Laufzeittyp {o.GetType()}");
```

produziert die Ausgabe:

```
System.Object hat den Laufzeittyp System.Object
          123 hat den Laufzeittyp System.String
           13 hat den Laufzeittyp System.Int32
```

7.9 Polymorphie (Überschreiben von Methoden)

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, dann zeigt es bei überschreibenden *und* bei verdeckenden Methoden das Unterklassenverhalten. Bei Ansprache über eine Referenz vom *Basisklassentyp* gilt hingegen:

- Bei einer Verdeckung kommt die *Basisklassenvariante* zum Einsatz.
- Bei einer Überschreibung wird die *Unterklassenvariante* benutzt.

Während bei einer *Verdeckung* die auszuführende Methode schon beim Übersetzen festliegt, wird im Fall der *Überschreibung* erst zur Laufzeit 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 ist.[1]

Werden Objekte aus verschiedenen Klassen über Referenzvariablen eines gemeinsamen Basistyps verwaltet, dann sind nur Methoden nutzbar, die schon in der Basisklasse definiert sind. Bei überschreibenden Methoden reagieren die Objekte aber jeweils unterklassenspezifisch 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 der objektorientierten Programmierung (siehe Abschnitt 5.1.1).

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})");
 }
}
```

In der abgeleiteten Klasse `Kreis` wird mit dem Schlüsselwort **override** das Überschreiben der geerbten `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() { }
```

[1] Siehe Einschätzungen in https://stackoverflow.com/questions/9937150/performance-impact-of-virtual-methods und Messungen (bei C++) in: https://stackoverflow.com/questions/449827/virtual-functions-and-performance-c

```
    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 im Abschnitt 7.7 Referenzen auf Figuren und Kreise aufnehmen, z. B.:

```
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) des angesprochenen Objekts fest und wählt die passende Methode:

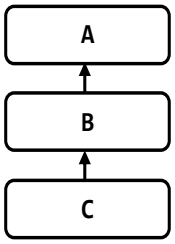
```
Oben Links:     (100, 100)

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.

Der Visual Studio - Projektordner mit dem Figurenbeispiel auf dem aktuellen Entwicklungsstand ist hier zu finden:

...\BspUeb\Vererbung und Polymorphie\Polymorphie

Bei komplizierten Erbschaftsverhältnissen kann es passieren, dass beim Aufruf einer überschreibenden Methode durch eine Basisklassenreferenz *keine* Polymorphie stattfindet. Im folgenden Beispielprogramm (nach Mössenböck, 2019, S. 96) mit den Vererbungsbeziehungen



überschreibt die Klasse `C` die ursprünglich in der Klasse `A` virtuell definierte Methode `M()`. Die von `A` abgeleitete Klasse `B` verdeckt aber die Methode `M()` und deklariert die Ersetzung als virtuell:

```
public new virtual void M() { ... }
```

Wird in dieser Konstellation ein `C`-Objekt per `A`-Referenz beauftragt, die Methode `M()` auszuführen, dann kommt die Variante der Klasse `A` zum Einsatz (keine Polymorphie). Wird dasselbe Objekt per `B`-Referenz beauftragt, die Methode `M()` auszuführen, dann kommt polymorph die überschreibende Variante der Klasse `C` zum Einsatz:

Quellcode	Ausgabe

`using System;`
`class A {`
`  public virtual void M() {`
`  Console.WriteLine("M in A");`
`  }`
`}`
 	
`class B : A {`
`  public new virtual void M() {`
`  Console.WriteLine("M in B");`
`  }`
`}`
 	
`class C : B {`
`  public override void M() {`
`  Console.WriteLine("M in C");`
`  }`
`}`
 	
`class Prog {`
`  static void Main() {`
`  C c = new();`
`  A a = c;`
`  B b = c;`
`  a.M();`
`  b.M();`
`  }`
`}`
	`M in A`
	`M in C`

In Bezug auf den Zugriffsschutz von virtuellen bzw. überschreibenden Methoden gelten die folgenden Regeln:

- Bei virtuellen Methoden ist der Zugriffsschutz **private** verboten.
- Beim Überschreiben darf der Zugriffsschutz nicht geändert werden. Im aktuellen Beispiel ist die Methode `Wo()` in der Klasse `Figur` als **public** deklariert, sodass der folgende Versuch scheitert, die überschreibende Methode der Klasse `Kreis` als **protected** zu deklarieren:

```
protected override void Wo() {
}
```

Das Überschreiben ist nicht nur bei Methoden erlaubt, sondern auch bei Eigenschaften und Indexern.

Bei *statischen* Methoden sind die Modifikatoren **virtual** und **override** sinnlos und verboten, weil es hier nicht zu einer späten Bindung kommen kann. Das per **new**-Modifikator signalisierte Verdecken einer geerbten statischen Methode ist jedoch sinnvoll und erlaubt (siehe Abschnitt 7.6.1).

In Bezug auf das Überladen von Methoden sorgen unterschiedliche Richtungsmodifikatoren (**ref**, **in**, **out**) zu einem Verweisparameter *nicht* für abweichende Signaturen (siehe Abschnitt 5.3.5). Folglich können z. B. die folgenden Methoden wegen identischer Signaturen *nicht* in einer Klasse koexistieren:

```
class Prog {
    public virtual void Meth(ref int par) {
        Console.WriteLine("Meth mit ref-Parameter");
    }

    public virtual void Meth(in int par) {
    }
}
```

void Prog.Meth(in int par)

CS0663: "Prog" kann kein überladenes Methode-Element definieren, das sich nur in den Parametermodifizierern "in" und "ref" unterscheidet.

In Bezug auf das Überschreiben von Methoden sorgen unterschiedliche Richtungsmodifikatoren aber doch für abweichende Signaturen, sodass z. B. in der folgenden Situation wegen abweichender Signaturen kein Überschreiben möglich ist:[1]

```
class A {
    public virtual void Meth(ref int par) {
        Console.WriteLine("Meth in A");
    }
}

class B : A {
    public override void Meth(in int par) {
        Console.WriteLine(
    }
}
```

void B.Meth(in int par)

CS0115: "B.Meth(in int)": Es wurde keine passende Methode zum Überschreiben gefunden.

7.10 Versiegelte Methoden und Klassen

Gelegentlich möchte man das Überschreiben einer Methode *verhindern*, damit 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 ein per `B`-Referenz angesprochenes `C`-Objekt garantiert die `B`-Methode `Passwd()` aus.

Meist verhindert man in C# das Überschreiben einer Methode schlicht dadurch, dass man die Methode *nicht* als **virtual** deklariert. Wenn allerdings eine Methode den Modifikator **override** verwendet, um eine virtuelle Basisklassenvariante zu überschreiben, dann ist sie per Voreinstellung ihrerseits virtuell. Um eine solche Methode vor dem Überschreiben zu bewahren, muss sie auch den Modifikator **sealed** (dt.: *versiegelt*) erhalten. Der Modifikator **sealed** ist also nur erforderlich bei Methoden, die auch den Modifikator **override** besitzen, und er ist auch nur in dieser Situation erlaubt.

---

[1] Um die Missverständnisse im Zusammenhang mit dem Begriff *Signatur* einzudämmen, sollte zwischen der *Überladungssignatur* und der (beim Ersetzen bzw. Überschreiben von Methoden relevanten) *Ersetzungssignatur* unterschieden werden.

Um den Modifikator **sealed** in einem Beispiel mit der Methode `Passwd()` vorführen zu können, muss also neben den oben vorgestellten Klassen `B` und `C` noch die dritte Klasse `A` ins Spiel kommen:

```
class A {
  public virtual void Passwd() { }
}

class B : A {
  public sealed override void Passwd() { }
}
```

Eine Betrachtung unterschiedlicher Einsatzfälle für die Modifikatoren **virtual**, **new**, **override** und **sealed** bestätigt, dass der Modifikator **sealed** ausschließlich in Kombination mit dem Modifikator **override** erforderlich ist:

- 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 den 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 per Voreinstellung 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 versiegeln, sodass sie zwar verwendet, aber nicht beerbt werden kann. Microsoft nennt als möglichen Grund die Existenz von sicherheitsrelevanten Geheimnissen, die eine Klasse geerbt hat. Die Erbstücke haben die Schutzstufe **protected**, und diese Schutzstufe kann generell nicht eingeschränkt werden.[1] Durch das Versiegeln wird verhindert, dass die Geheimnisse in einer weiteren Vererbungsgeneration bekannt werden.

Es gibt noch weitere Gründe für das Versiegeln von Klassen:

- Wenn für eine Klasse die Unveränderlichkeit ihrer Objekte erreicht werden soll (z. B. wegen der Vorteile beim Multithreading), dann muss auch die Vererbung unterbunden werden. Anderenfalls könnte eine abgeleitete Klasse die Unveränderlichkeit unterlaufen, denn ihre Objekte werden vom Compiler überall akzeptiert, wo die Basisklassenreferenz vorgeschrieben ist (z. B. bei einem Parameterdatentyp).
- Der JIT-Compiler kann bei versiegelten Klassen Optimierungsmaßnahmen anwenden und die Performanz leicht verbessern.[2]

Von den versiegelten BCL-Klassen ist **String** das bekannteste Beispiel:[3]

---

1 https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/sealing

2 https://www.meziantou.net/performance-benefits-of-sealed-class.htm

3 Aus dem Quellcode der BCL zu .NET 7.

```
public sealed partial 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 6.3.1.3) und außerdem von vielen (Bibliotheks)klassen vorausgesetzt wird.

Von der Vererbung ausgeschlossen sind neben den versiegelten Klassen auch die statischen Klassen (vgl. Abschnitt 5.7.5).

7.11 Fragilität und Komplexität

In C# sind beim Ersetzen einer Basisklassenmethode durch eine signaturgleiche Unterklassenmethode (Verdecken oder Überschreiben) mehrere Modifikatoren (**virtual**, **override**, **new**, **sealed**) und Regeln im Spiel, die in der folgenden Tabelle zusammengestellt sind:

Ersetzungsart	Unterstützung der Polymorphie	Syntax und Verhalten
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. Eine verdeckende Methode ist nicht virtuell, kann also in einer abgeleiteten Klasse nicht überschrieben, aber erneut verdeckt werden. 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, wenn die Methode nicht in der Vererbungshistorie verdeckt und erneut virtualisiert worden ist (siehe Abschnitt 7.9). Eine überschreibende Methode darf den Zugriffsschutz der Basisklassenmethode nicht ändern. Eine überschreibende Methode ist virtuell, wenn nicht mit dem Modifikator **sealed** das Überschreiben verhindert wird. Statische Methoden können *nicht* überschrieben werden.

Eine virtuelle Basisklassenmethode kann also verdeckt oder überschrieben werden:

- **Überschreibt** eine abgeleitete Klasse die Methode (Modifikator **override**), dann ist die überschreibende Methode per Voreinstellung virtuell (mit Bedeutung für die nächste Ableitungsgeneration). Den Modifikator **virtual** zusammen mit dem Modifikator **override** anzugeben, ist überflüssig und verboten. Ausschließlich in Kombination mit **override** erlaubt ist der Modifikator **sealed**, der das Überschreiben in abgeleiteten Klassen verhindert.
- **Verdeckt** eine abgeleitete Klasse die Methode (Modifikator **new**), dann ist die Methode in der abgeleiteten Klasse nicht mehr virtuell, sofern nicht gleichzeitig der Modifikator **virtual** vergeben wird.

Bei einer *nicht*-virtuellen Basisklassenmethode ist nur das Verdecken möglich.

Software-Entwickler haben die Mittel und die Pflicht, eine riskante bzw. fehlerhafte Anwendung der Vererbung zu verhindern, z. B.:

- Durch das Überschreiben einer Methode ermöglicht man die Polymorphie, übernimmt aber auch mehr Verantwortung, denn:
  - Eine verdeckende Methode wird nur bei Verwendung einer Unterklassenreferenz aufgerufen (direkt oder indirekt durch andere Methoden in der Unterklasse).
  - Eine überschreibende Methode wird bei Verwendung einer Unterklassen- und bei Verwendung einer Basisklassenreferenz aufgerufen (direkt oder indirekt durch andere Methoden in der Unterklasse oder in einer Basisklasse).

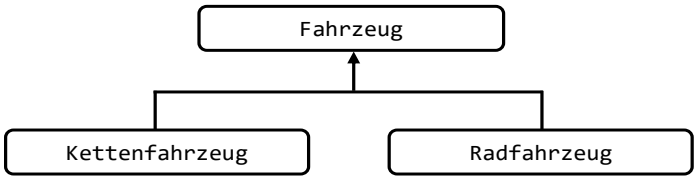
  Vorsicht ist vor allem beim Überschreiben einer Methode aus einer *fremden* Basisklasse geboten. Wenn man den Quellcode der Basisklasse kennt, führt das Überschreiben aber kaum zu einem erhöhten Risiko für Programmierfehler.
- Im Konstruktor einer potenziellen Basisklasse dürfen keine überschreibbaren Methoden aufgerufen werden, weil der Konstruktor der Basisklasse *vor* dem Konstruktor der abgeleiteten Klasse ausgeführt wird (siehe Abschnitt 7.4). Folglich würde die überschreibende Methode der abgeleiteten Klasse *vor* dem Konstruktor der abgeleiteten Klasse aufgerufen. Ein Objekt der abgeleiteten Klasse befindet sich aber erst nach dem Konstruktoraufruf in einsatzfähigem Zustand (Bloch 2018, S. 95f).

Das Verdecken von Methoden hat zur Folge, dass ein Objekt je nach dem deklarierten Typ der zur Ansprache verwendeten Referenzvariablen auf denselben Methodenaufruf unterschiedlich reagiert. Diese komplexe und potenziell verwirrende Konstellation sollte nach Möglichkeit durch einen eigenständigen Namen für die Unterklassenmethode vermieden werden (Mössenböck 2019, S. 97).

In C# wurde für das Ersetzen von Basisklassenmethoden eine komplexe Kombination von Überschreibung und Verdeckung entwickelt, um das Problem der sogenannten *fragilen Basisklassen* zu beheben (siehe z. B. Mössenböck 2019, S. 96f).[1] In der Programmiersprache Java gibt es beim Ersetzen von Instanzmethoden in abgeleiteten Klassen nur die Überschreibung. Das sorgt für angenehm einfache Verhältnisse, aber leider auch für Gefahren, weil eine Änderung der Basisklasse schädliche Effekte auf abgeleitete Klassen haben kann (siehe z. B. Baltes-Götz & Götz 2023, Abschnitt 7.10). Für Java wird die Vererbung mittlerweile von vielen ernst zu nehmenden Autoren als kritische Technik eingeschätzt, die besondere Sorgfalt erfordert (Bloch 2018, S. 87ff).

Nach den vielen Warnungen sind zwei Hinweise angemessen, damit die Vererbung nicht am Ende als entmutigend komplex erlebt wird:

- Wenn man die Basisklasse und die abgeleitete Klasse selbst erstellt, also den Quellcode beider Klassen kennt, dann bewegt man sich auch beim Einsatz der Vererbung im normalen Risikobereich von Programmierfehlern.
- Bei einer konkreten Aufgabenstellung ist die Lage viel übersichtlicher als in den allgemeinen Erörterungen des aktuellen Kapitels. Wenn etwa Fahrzeuge mit Rad- und Kettenantrieb, die sich auf einem gemeinsamen Gelände bewegen, durch ein Programm zu modellieren und zu steuern sind, dann bietet sich die folgende Vererbungshierarchie an:

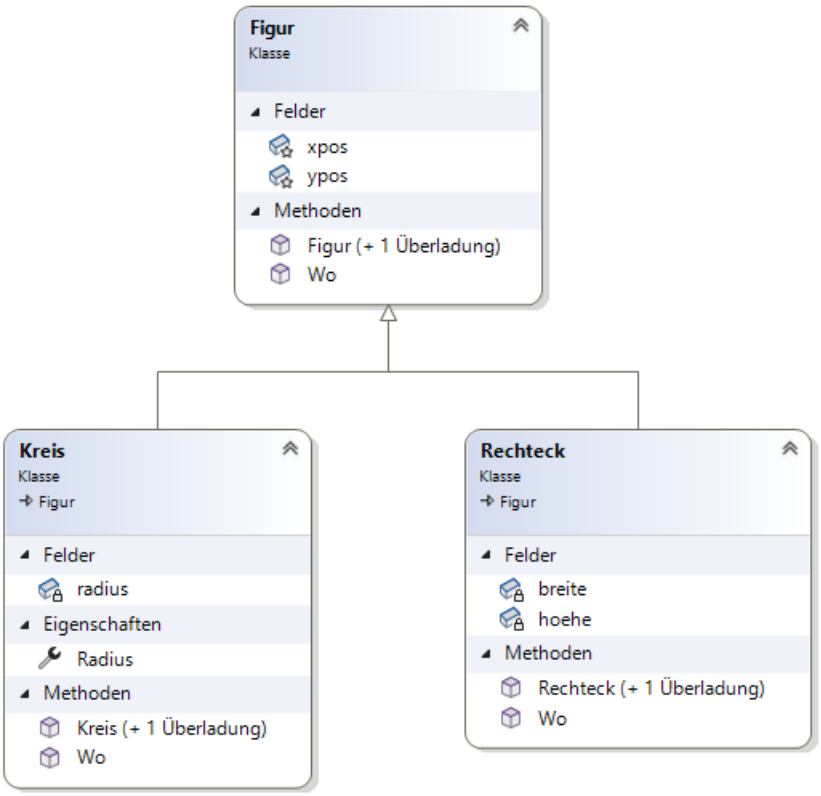


[1] https://stackoverflow.com/questions/2921397/what-is-the-fragile-base-class-problem

Bei der für alle Fahrzeugklassen benötigten Methode `ZielAnfahren()` ist eine polymorphe Lösung sinnvoll. Die Vererbungstechnik spart Entwicklungszeit, und man muss sich über fragile Basisklassen oder verdeckende Methoden keine Gedanken machen.

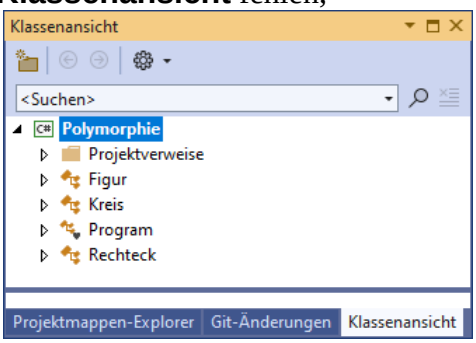
7.12 Klassendiagramme mit Vererbungsbeziehung

Unsere Entwicklungsumgebung Visual Studio Community 2022 kann mit wenigen Mausaktionen zur Erstellung des folgenden Klassendiagramms für das Figurenbeispiel veranlasst werden, wenn die im Abschnitt 3.3.6 beschriebene Erweiterung der Installation um den **Klassen-Designer** durchgeführt worden ist. Die Vererbungs- bzw. Spezialisierungsbeziehungen zwischen den Klassen werden nach den Regeln der UML-Notation (*Unified Modeling Language*) dargestellt:



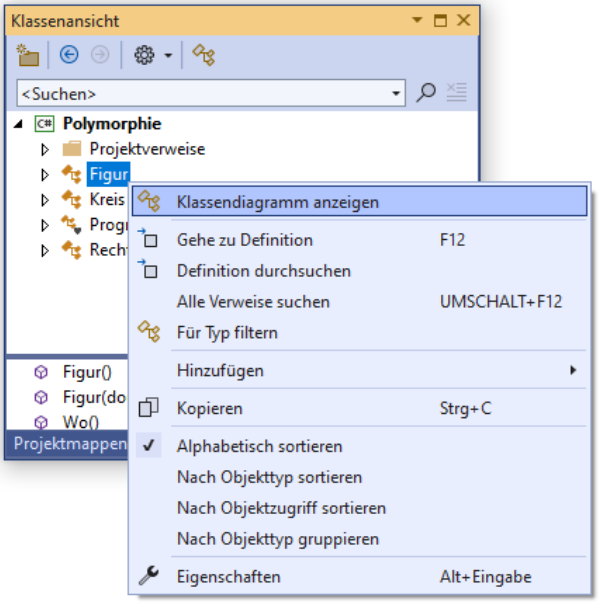
So lässt sich ein Klassendiagramm erstellen:

- Projekt öffnen und nötigenfalls das Programm erstellen
- Sollte das Registerblatt der **Klassenansicht** fehlen,

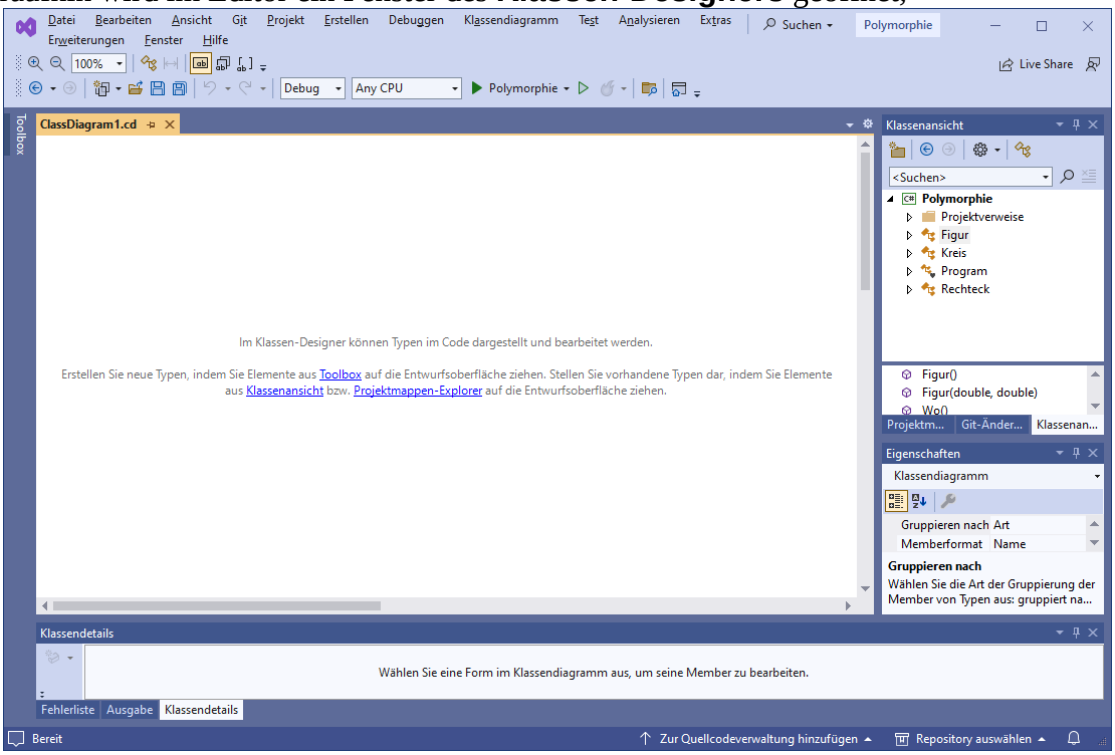


öffnet man es mit dem Menübefehl:
Ansicht > Klassenansicht

- Auf dem Registerblatt der **Klassenansicht** wählt man aus dem Kontextmenü zu einer Klasse das Item **Klassendiagramm anzeigen**, z. B.:

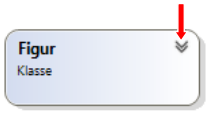


- Daraufhin wird im Editor ein Fenster des **Klassen-Designers** geöffnet,

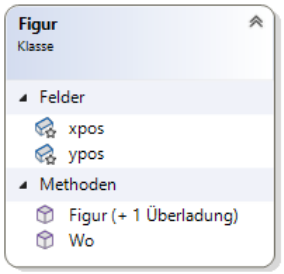


und aus der **Klassenansicht** können per Drag & Drop Klassen auf die Designerzone befördert werden.

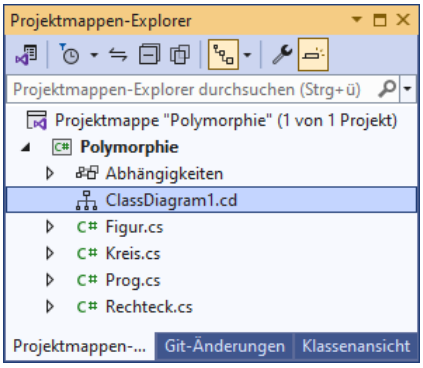
- Die initial komprimierte Darstellung einer Klasse



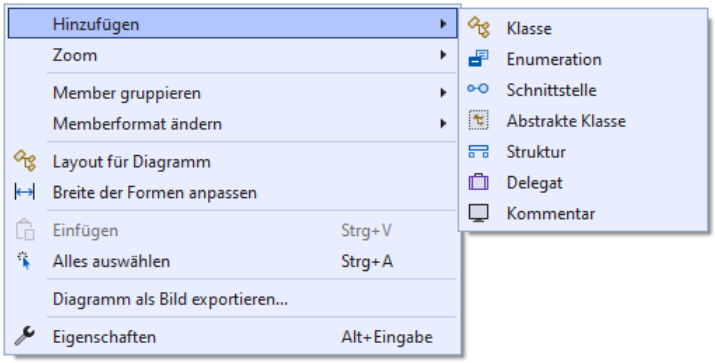
lässt sich per Mausklick erweitern:



- Die Spezialisierungsbeziehungen zwischen den Klassen werden automatisch ergänzt, sodass man nur noch die Positionen der Klassen zu wählen hat, um das oben dargestellte Ergebnis zu erzielen.
- Das Ergebnis wird als Datei mit der Namenserweiterung **cd** im Projektordner abgelegt. Diese Datei erscheint im **Projektmappen-Explorer** und ermöglicht das Öffnen das Diagramms per Doppelklick:



Das Kontextmenü zu einem geöffneten Klassendiagramm erlaubt einige Modifikationen und den Export des Diagramms in eine Datei mit wählbarem Format (z. B. EMF und PNG):



7.13 Abstrakte Methoden und Klassen

Um die 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 in der Basisklasse zu einer Methode keine sinnvolle Implementierung gibt, erstellt man dort eine **abstrakte** Methode:

- Man beschränkt sich auf den Methodenkopf, der den Modifikator **abstract** erhält.
- 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 dort 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.[1]

Abstrakte Methoden sind grundsätzlich virtuell (vgl. Abschnitt 7.9), 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, dann lassen sich Objekte von dieser Klasse herstellen; anderenfalls ist sie ebenfalls abstrakt.[2]

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) kommt bei einer abstrakten Basisklassenmethode nur das Überschreiben in Frage, wobei das zugehörige Schlüsselwort **override** anzugeben ist (siehe Abschnitt 7.9).

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

---

[1] Während der (vom Compiler spendierte) Standardkonstruktor einer konkreten (instanziierbaren) Klasse die (nicht änderbare) Schutzstufe **public** besitzt, hat der Standardkonstruktor einer abstrakten Klasse die (nicht änderbare) Schutzstufe **protected**.

[2] Wenn im Beispiel `Figur`-Objekte erforderlich wären, dann könnte man in der Klasse `Figur` eine `Skaliere()`-Methode mit einem leeren Rumpf definieren und auf den **abstract**-Modifikator verzichten. Das passive, vom Parameterwert unbeeindruckte Verhalten der `Figur`-Objekte nach der Botschaft `Skaliere()` wäre kein allzu großes Problem, aber auch kein ideales Design.

schon in der Basisklasse `Figur` vorhanden sein. Dort ist aber keine sinnvolle Flächenberechnung möglich, sodass die Eigenschaft abstrakt definiert wird:

```
public abstract double Inhalt {
   get;
}
```

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 sogar unverzichtbar, wenn Objekte diverser Unterklassen gemeinsam verwaltet werden sollen, z. B.:

Quellcode	Ausgabe

`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($"Fläche Figur {i}:`
`{fa[i].Inhalt,10:f2}");`
`  ges += fa[i].Inhalt;`
`}`
`Console.WriteLine($"\nGesamtfläche:  {ges,10:f2}");`
	`Fläche Figur 0:  314,16`
	`Fläche Figur 1:  100,00`
	
	`Gesamtfläche:  414,16`

Mit Hilfe der im Kapitel 9 vorzustellenden *Schnittstellen* werden wir noch mehr Flexibilität gewinnen und polymorphe Methodenaufrufe auch für Typen *ohne* gemeinsame Basisklasse realisieren.

7.14 Das Liskovsche Substitutionsprinzip

In diesem Abschnitt geht es um eine auf den ersten Blick theoretisch wirkende, aber durchaus praxisrelevante Klärung zur objektorientierten Vererbungsbeziehung. 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 übernehmen, d.h. u. a.:

- Das „vertraglich“ zugesicherte Verhalten der Basisklassenmethoden wird auch von den verdeckenden oder überschreibenden Unterklassenvarianten eingehalten.
- Unterklassenobjekte werden bei Verwendung in der Rolle von Basisklassenobjekten nicht beschädigt.

Zu einer Verletzung der Ersetzbarkeitsregel kommt es z. B., weil 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 Sicht 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. 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 zu Problemen kommen, weil nach einer Verdopplung der X-Ausdehnung der Flächeninhalt entgegen der Erwartung nicht auf das Doppelte, sondern auf das Vierfache wächst.

Um die Einhaltung des Substitutionsprinzips beurteilen zu können, bedarf es einer sorgfältigen Analyse. Wenn etwa Objekte der Klasse `Rechteck` *unveränderlich* wären, wenn also die Methoden `SkaliereX()` und `SkaliereY()` in der Klassendefinition von `Rechteck` fehlen würden, dann könnte die Klasse `Quadrat` sehr wohl als Spezialisierung von `Rechteck` definiert werden.

C# bietet gute Voraussetzungen für eine erfolgreiche objektorientierte Programmierung, kann aber z. B. eine Verletzung des Substitutionsprinzips nicht verhindern.

7.15 Erweiterungsmethoden

Soll eine Klasse um zusätzliche Handlungskompetenzen erweitert werden, dann 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 bei versiegelten Klassen).

Ihre wichtigste Anwendung finden Erweiterungsmethoden bei den LINQ-Abfragen (*Language Integrated Query*, siehe Kapitel 19 in Baltes-Götz (2021)), doch sie werden auch in anderen Bereichen erfolgreich eingesetzt.

7.15.1 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.

Den Instanzen des zu erweiternden Typs können die neuen Botschaften syntaktisch genauso zugestellt werden wie die typeigenen. Der Compiler erstellt jedoch statische Methodenaufrufe, und der Zugriffsschutz wird nicht verletzt, weil eine Erweiterungsmethode keinen direkten Zugriff auf private Member (z. B. Methoden, Instanzvariablen) des erweiterten Typs hat.

Durch die im folgenden Beispiel definierte Erweiterungsmethode `Empty()` für die Klasse **String** wird festgestellt, ob ein **String**-Objekt vorhanden ist, aber keine Zeichen enthält:

Quellcodesegment	Ausgabe

`string s = null;`
`Console.WriteLine(s.Empty());`
`Console.WriteLine("m".Empty());`
`Console.WriteLine("".Empty());`
 	
`public static class StringExt {`
`  public static bool Empty(this String arg) {`
`  return arg != null && arg.Length == 0;`
`  }`
`}`
	`False`
	`False`
	`True`

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:

Quellcodesegment	Ausgabe

`double pi = 3.14;`
`Console.WriteLine(pi.H(2));`
 	
`public static class DMath {`
`  public static double H(this double arg, double expo) {`
`  return Math.Pow(arg, expo);`
`  }`
`}`
	`9,8596`

Als zu erweiternde Typen sind nicht nur Klassen und Strukturen zulässig, sondern auch Schnittstellen (vgl. Kapitel 9).

Visual Studio - Projektordner mit den Beispielen des aktuellen Abschnitts sind hier zu finden:

...\BspUeb\Vererbung und Polymorphie\Erweiterungsmethoden

7.15.2 Anwendungsempfehlung

Um die Funktionalität einer Klasse zu erweitern, sollte nach einer Empfehlung von Microsoft nach Möglichkeit eine abgeleitete Klasse definiert werden.[1]

> While it's still considered preferable to add functionality by modifying an object's code or deriving a new type whenever it's reasonable and possible to do so, extension methods have become a crucial option for creating reusable functionality throughout the .NET ecosystem.

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.

7.16 Übungsaufgaben zum Kapitel 7

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();
   }
}
```

2) Im folgenden Beispiel wird die Klasse `Kreis` aus der Klasse `Figur` abgeleitet:

```
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;
        }
    }
    public Figur() { }
}
```

[1] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

```
public class Kreis : Figur {
    double radius = 75.0;

    public Kreis(double x, double y, double rad) {
        if (x >= 0.0 && y >= 0.0 && rad >= 0.0) {
            xpos = x;
            ypos = y;
            radius = rad;
        }
    }
    public Kreis() { }
}
```

Trotzdem erlaubt der Compiler im initialisierenden `Kreis`-Konstruktor den `Kreis`-Objekten keinen 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?

4) Überschreiben Sie in der Klasse `Bruch` die von der Basisklasse **Object** geerbte Methode **ToString()**, die in der Klasse **Object** den folgenden Definitionskopf hat:

public virtual string ToString()

Die verbesserte **ToString()** - Rückgabe kann z. B. so aussehen:

Quellcode	Ausgabe ohne **ToString()** - Überschreibung	Ausgabe mit **ToString()** - Überschreibung
`var b = new Bruch(7, 8, "");` `Console.WriteLine(b);`	`Bruch`	`7/8`

8 Typgenerisches Programmieren

In C# haben die Felder von Klassen und Strukturen sowie die Parameter von Methoden einen festen Datentyp, sodass der Compiler für Typsicherheit sorgen, d. h. die Zuweisung ungeeigneter Werte bzw. Objekte verhindern kann. Oft werden aber 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, und es entsteht eine sogenannte *geschlossene konstruierte Klasse* (engl.: *closed constructed class*).[1] Im Ergebnis erhält man durch *eine* Definition zahlreiche konkrete Klassen, wobei die Typsicherheit erhalten bleibt.

Durch das typgenerische Programmieren werden die folgenden Ziele erreicht:

- Wiederverwendung von Code
  Code mit generischem Design kann für unterschiedliche Typkonkretisierungen verwendet werden.
- Typsicherheit und Lesbarkeit des Codes
  Im Vergleich zur veralteten Verwendung eines allgemeinen Referenzdatentyps (wie z. B. **Object**) für die Elemente in einer Liste oder einer anderen Kollektion spart man sich lästige und fehleranfällige Typanpassungen.
- Performanz
  Bei der Verwendung eines allgemeinen Referenzdatentyps (wie z. B. **Object**) für Kollektionselemente werden aufwändige (Un)boxing - Operationen in großer Zahl fällig, wenn Elemente mit Werttyp zu verwalten sind. Im Vergleich zu dieser veralteten Technik bringt die generische Programmierung eine erhebliche Verbesserung der Performanz.

Ein besonders erfolgreiches Anwendungsfeld für Typgenerizität sind die Klassen zur Verwaltung von Listen, Mengen, (Schlüssel-Wert) - Tabellen und sonstigen Kollektionen. 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 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, Delegaten und Ereignisse.

8.1 Motive für generische Typen

Im Abschnitt 6.2.11 haben wir die Klasse **ArrayList** aus dem Namensraum **System.Collections** als Container für Objekte beliebigen Typs verwendet:[2]

```
ArrayList al = new();
al.Add("Text");
al.Add(3.14);
al.Add(13);
```

Im Unterschied zu einem Array (siehe Abschnitt 6.2) bietet die Klasse **ArrayList**:

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-classes

[2] Wer sich noch nicht an die seit C# 9 möglichen **new**-Ausdrücke ohne Konstruktor gewöhnt hat, findet im Abschnitt 5.4.3.1.3 eine Erläuterung zu den zieltypisierten **new**-Ausdrücken.

- eine automatische Größenanpassung
- Typflexibilität (durch die Verwendung des maximal allgemeinen 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 die folgenden Nachteile der Typbeliebigkeit:

- Fehlende Typsicherheit
  Wenn beliebige Objekte zugelassen sind, die intern über Referenzvariablen vom Typ **Object** verwaltet werden, dann kann der Compiler *nicht* sicherstellen, dass ausschließlich Objekte des gewünschten Typs in den Container eingefüllt werden. Viele Programmierfehler werden erst zur Laufzeit (womöglich vom Kunden) entdeckt.
- Notwendigkeit von expliziten Typumwandlungen
  Entnommene Objekte können erst nach einer expliziten Typumwandlung die Methoden ihres Typs ausführen. Die häufig benötigten Typanpassungen sind lästig und fehleranfällig.
- Leistungsschädliche (Un)boxing-Operationen
  Wenn Variablen mit Werttyp zu verwalten sind, dann resultieren leistungsschädliche (Un)boxing-Operationen, weil intern ein Referenzdatentyp (wie z. B. **Object**) verwendet wird.

Im folgenden Beispielprogramm sollen **String**-Objekte in einem **ArrayList**-Container verwaltet werden:[1]

```
using System.Collections;
ArrayList al = new();
al.Add("Otto");
al.Add("Rempremerding");
al.Add('.');
for(int i = 0; i < al.Count; i++)
    Console.WriteLine($"Länge von Element {i}: {((String)al[i]).Length}");
```

Bevor ein aus der Liste entnommenes **String**-Element nach seiner Länge befragt werden kann, ist eine lästige Typanpassung fällig, weil der Compiler nur den deklarierten Typ **Object** kennt:

```
((String)al[i]).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 an einem Ausnahmefehler vom Typ **InvalidCastException**:

```
Unhandled exception. System.InvalidCastException: Unable to cast object of type
'System.Char' to type 'System.String'.
   at Program.<Main>$(String[] args) in Prog.cs:line 7
```

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 analog funktionierende Behälter aber auch für alternative Elementtypen benötigt, und entsprechend viele Klassen zu definieren, die sich nur durch den Inhaltstyp

---

[1] Ausnahmsweise soll noch einmal daran erinnert werden, dass es sich tatsächlich um ein vollständiges Programm handelt, wobei die folgenden Neuerungen von C# 10 ausgenutzt werden:
- Aufgrund der Nutzung von impliziten **using**-Direktiven ist der Namensraum **System** automatisch importiert (siehe Abschnitt 2.6.3.2).
- Aufgrund der Nutzung von Anweisungen auf oberster Ebene wird die Startklasse samt **Main()** – Methode vom Compiler definiert (siehe Abschnitt 4.1.2).

unterscheiden, wäre nicht rationell. Für eine solche Aufgabenstellung bietet C# seit der Version 2.0 die generischen Typen. Durch die Verwendung von Typformalparametern bei der Definition wird die gesamte Handlungskompetenz typunabhängig formuliert. Bei jedem Instanziieren (also beim Erstellen einer Container-Instanz) ist jedoch ein konkreter Elementtyp anzugeben. Weil der Compiler den konkreten Typ kennt, kann er beim Befüllen des Containers für Typsicherheit sorgen, und bei der Verwendung von entnommenen Elementen sind keine lästigen Typumwandlungen erforderlich.

Mit der generischen Klasse **List<T>** im Namensraum **System.Collections.Generic** enthält die BCL eine perfekte **ArrayList**-Alternative zur Verwaltung einer dynamischen Liste von Elementen mit identischem Typ. Das obige Beispielprogramm ist schnell auf die neue Technik umgestellt:

```
List<String> gl = new();
gl.Add("Otto");
gl.Add("Rempremerding");
gl.Add(".");
for (int i = 0; i < gl.Count; i++)
   Console.WriteLine($"Länge von Element {i}: {gl[i].Length}");
```

Bei der Deklaration einer **List<T>** - Referenzvariablen ist an Stelle des Typformalparameters **T** ein konkreter Datentyp anzugeben, z. B.:

```
List<String> gl = new();
```

Jeder Versuch, ein Element mit abweichendem Typ in den **List<String>** - Container einzufügen, wird vom Compiler verhindert, z. B.:

```
gl.Add(',');
```

Die Elemente des auf **String**-Objekte spezialisierten Containers beherrschen ohne Typanpassung die Methoden ihrer Klasse, z. B.:

```
for (int i = 0; i < gl.Count; i++)
   Console.WriteLine($"Länge von Element {i}: {gl[i].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.

Wenn ausnahmsweise für die Elemente in einer Liste *unterschiedliche* (nicht in Vererbungsrelation stehende) Datentypen erlaubt sein sollen, dann ist die Klasse **ArrayList** noch als akzeptable Lösung zu betrachten. Allerdings erlaubt die generische Klasse **List<T>** auch für solche Fälle eine attraktivere Lösung, indem der Typformalparameter durch die Klasse **Object** konkretisiert wird. Für die Klasse **List<Object>** ist im Vergleich zur Klasse **ArrayList** mit einer besseren Kompatibilität zu rechnen, weil moderne Software-Entwicklung bevorzugt generische Datentypen verwendet.

8.2 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 BCL-Namensraum **System.Collections.Generic**), aber keinesfalls auf Container mit ihrer weitgehend inhaltstyp*un*abhängigen Verwaltungslogik beschränkt.

Analog zu den generischen Klassen bietet C# auch generische Strukturen, Schnittstellen, Delegaten und Ereignisse. Wir befassen uns im Kapitel 8 hauptsächlich mit generischen Klassen und Strukturen, machen aber auch schon erste Erfahrungen mit generischen Schnittstellen, die wir im Kapitel 9 vertiefen werden.

8.2.1 Definition

Bei einer generischen Klassendefinition verwendet man **Typformalparameter**, die im Definitionskopf 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 einen Typformalparameter für den beliebig wählbaren Elementtyp verwendet. In der Praxis wird man für eine solche Standardaufgabe allerdings die Klasse **Stack<T>** aus dem BCL-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 void Push(T element) {
      if (size < capacity)
         data[size++] = element;
      else
         throw new System.InvalidOperationException("Kapazität erschöpft");
   }

   public T Pop() {
      if (size > 0)
         return data[--size];
      else
         throw new System.InvalidOperationException("Stapel leer");
   }
}
```

Mit der Methode `Push()` legt man ein neues Element auf den Stapel, sofern das noch möglich ist. Wenn beim Aufruf die Kapazität des Stapels erschöpft ist, dann wirft die Methode `Push()` 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 13 über die Ausnahmebehandlung schon mehrfach die Rolle des potenziellen *Empfängers* von Ausnahmeobjekten beobachten konnten, erleben Sie nun einen Ausblick auf die Rolle des *Senders* von Ausnahmeobjekten.

Mit der Methode `Pop()` wird das oberste Element vom Stapel abgehoben und dem Aufrufer übergeben. Ist der Stapel leer, wird der Aufrufer durch ein Ausnahmeobjekt aus der Klasse **InvalidOperationException** informiert.

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()`

In einer Konstruktordefinition ist der Klassenname *ohne* Typformalparameter zu schreiben, z. B.:

```
public SimpleStack() {
   data = new T[capacity];
}
```

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 (siehe Beschreibung der **Array**-Methode **Resize()** im Abschnitt 6.2.2). Im Beispiel wird der Einfachheit halber auf die Größendynamik verzichtet.

Im folgenden Programm wird aus der generischen Klasse `SimpleStack<T>` eine konkrete Klasse (man sagt auch: *eine geschlossene konstruierte Klasse*) zur Verwaltung eines **double**-Stapels erzeugt:

Quellcode	Ausgabe

`var ds = new SimpleStack<double>(10);`
`ds.Push(3.141);`
`ds.Push(2.718);`
`while (ds.Count > 0)`
`  Console.WriteLine("Oben lag: " + ds.Pop());`
	`Oben lag: 2,718`
	`Oben lag: 3,141`

Im Abschnitt 11.5 werden Sie mit der BCL-Klasse **Dictionary<K, V>** ein Beispiel für eine generische Klasse mit *zwei* Typformalparametern kennenlernen.

Für einen einzelnen Typformalparameter sollte in der Regel der Großbuchstabe **T** verwendet werden. Wenn mehrere Typformalparameter auftreten, oder wenn ein einzelner Buchstabe nicht ausreicht, um die Bedeutung eines Typformalparameters zu klären, dann rät Microsoft zu einem aussagekräftigen, mit dem Buchstaben **T** eingeleiteten Namen, z. B.:[1]

Dictionary<TKey, TValue>

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.

Im Namen der Datei mit dem Quellcode einer generischen Klasse tauchen *keine* Typformalparameter auf. Z. B. steckt die Klasse **Dictionary<TKey, TValue>** aus der BCL zu .NET 7.0 in einer Datei mit dem Namen **Dictionary.cs**.

8.2.2 Restringierte Typformalparameter

Häufig muss eine generische Definition bei den Typen, die einen Typformalparameter konkretisieren dürfen, gewisse Handlungskompetenzen voraussetzen. Muss z. B. ein generischer Container seine Elemente sortieren, dann 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 (beschrieben unter Verwendung des Typformalparameters **T**):

public int CompareTo(T *instanz***)**

Im Abschnitt 6.3.1.2.2 haben Sie erfahren, dass z. B. die Klasse **String** eine solche Methode besitzt, und wie **CompareTo()** das Prüfergebnis über den Rückgabewert signalisiert:

---

[1] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-type-parameters#type-parameter-naming-guidelines

		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 im Kapitel 9 ausführlich beschäftigen.

Wir erstellen nun eine generische Listenverwaltungsklasse, die neue Elemente automatisch an der richtigen Sortierposition einfügt 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 void Add(T element) {
      if (size == data.Length)
         throw new System.InvalidOperationException("Kapazität erschöpft");
      int i;
      for (i = size - 1; i >= 0 && element.CompareTo(data[i]) < 0; i--)
         data[i + 1] = data[i];
      data[i + 1] = element;
      size++;
   }

   public int Count {
      get => size;
   }

   public T this[int index] {
      get {
         if (index < 0 || index >= size)
            throw new System.ArgumentOutOfRangeException();
         return data[index];
      }
   }
}
```

In der Methode **Add()** und im Indexer (vgl. Abschnitt 5.11) erlauben wir uns erneut einen Vorgriff auf das Kapitel über Ausnahmefehler und informieren den Aufrufer über eine erschöpfte Kapazität bzw. über einen ungültigen Index, indem wir eine **InvalidOperationException** bzw. eine **ArgumentOutOfRangeException** werfen.

Die get-only – Eigenschaft `Count` ist per Lambda-Symbol und Ausdruck realisiert, weil diese mittlerweile populäre Technik im Manuskript noch nicht allzu oft zu sehen war (vgl. Abschnitt 5.8.1).

Bei der Formulierung von Einschränkungen für einen Typformalparameter wird das Schlüsselwort **where** verwendet, wobei u. a. die folgenden Regeln gelten:[1]

- Man kann eine Basisklasse vorschreiben.
  Seit C# 7.3 sind als Basisklassen auch **System.Enum** (siehe Abschnitt 6.4) sowie **System.Delegate** und **System.MulticastDelegate** (siehe Kapitel 10) erlaubt, sodass man auch einen Aufzählungs- bzw. einen Delegatentyp vorschreiben kann.
- Mit dem Schlüsselwort **class** wird vereinbart, dass nur Referenztypen erlaubt sind.
- Mit dem Schlüsselwort **struct** wird vereinbart, dass nur Werttypen erlaubt sind (siehe Beispiel im Abschnitt 8.3). Dieser Anforderung genügen die elementaren Datentypen, die Strukturen und die Enumerationen.
- 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 9) 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, dann ist für jeden Parameter eine eigene **where**-Klausel anzugeben, und die **where**-Klauseln sind durch Kommata zu trennen.

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 (man sagt auch: *eine geschlossene konstruierte Klasse*) zur Verwaltung einer sortierten **int**-Liste erzeugt:

Quellcode	Ausgabe

`var si = new SimpleSortedList<int>(5);`
`si.Add(11);`
`si.Add(2);`
`si.Add(1);`
`si.Add(4);`
`for (int i = 0; i < si.Count; i++)`
`  System.Console.WriteLine(si[i]);`
	`1`
	`2`
	`4`
	`11`

8.2.3 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 geschlossene konstruierte Klasse
  ```
  class GenDerived<T> : GenBase<int, double> {
   . . .
  }
  ```

[1] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters

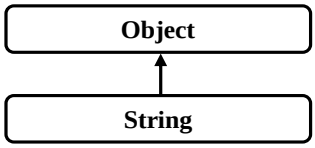
- eine sogenannte *offene konstruierte Klasse*, die noch Typformalparameter aus der neu definierten generischen Klasse enthält, wobei Typrestriktionen der Basisklasse ggf. zu wiederholen sind, z. B.:

```
class GenBase<T1, T2> where T2 : IComparable<T2> {
 . . .
}
class GenDerived<T> : GenBase<int, T> where T : IComparable<T> {
 . . .
}
```

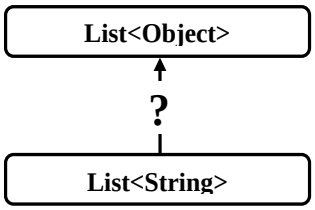
  Wird im Beispiel aus der generischen Klasse `GenDerived<T>` die geschlossene konstruierte Klasse `GenDerived<int>` erstellt, dann ist `GenBase<int, int>` deren Basisklasse.

Offenbar ist die nachträgliche Integration der generischen Typen in das *Common Type System* (CTS) der .NET - Plattform gut gelungen.

Im Zusammenhang mit der Vererbung und der Generizität bleibt die wichtige Frage zu klären, ob sich bei zwei geschlossenen Konstruktionen aus einer generischen Klasse mit dem Typformalparameter **T** (z. B. **List<T>**) eine Spezialisierungsbeziehung zwischen den beiden Aktualparametertypen, z. B.:



auf die konstruierten Klassen 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 essenzielle Prinzip der Sortenreinheit verletzt, und bei nächster Gelegenheit würde der Versuch, dem eingeschmuggelten Objekt eine **String**-Kompetenz abzuverlangen, zu einem Laufzeitfehler führen.

Um das beschriebene Desaster zu vermeiden, wird in C# eine Spezialisierungsbeziehung zwischen zwei Aktualparametertypen **NICHT** auf die zugehörigen Konkretisierungen derselben generischen Klasse übertragen, z. B.:

```
var list = new List<string>();
List<object> lob = list;
```

(Lokale Variable) List<string>? list

"list" ist hier nicht NULL.

CS0029: Der Typ "System.Collections.Generic.List<string>" kann nicht implizit in "System.Collections.Generic.List<object>" konvertiert werden.

Man sagt, dass in C# die Typformalparameter bei generischen Klassen **invariant** sind.

Wenn aus der Spezialisierungsbeziehung von Elementdatentypen eine entsprechende Spezialisierungsbeziehung für Container-Datentypen folgt, dann spricht man von einem **kovarianten** Verhalten, und dieses Verhalten zeigt C# bei Arrays. Infolgedessen drohen beim Einsatz von Arrays schwerwiegende Programmierfehler, die 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:

```
Unhandled exception. System.ArrayTypeMismatchException: Attempted to access
an element as a type incompatible with the array.
```

Obwohl bisher wenig Positives über die Kovarianz gesagt wurde, kann sie unter bestimmten Voraussetzungen doch sinnvoll eingesetzt werden. Bei generischen Schnittstellen und Delegaten ist es möglich, einen Typformalparameter explizit als kovariant zu deklarieren (siehe Abschnitte 9.1.4.1 bzw. 10.1.6).

8.3 Nullable<T> als Beispiel für generische Strukturen

In diesem Abschnitt wird die generische Struktur **Nullable<T>** aus der BCL vorgestellt:[1]

```
public partial struct Nullable<T> where T : struct {
   private readonly bool hasValue;
   internal T value;

   public Nullable(T value) {
      this.value = value;
      this.hasValue = true;
   }
   public readonly bool HasValue {
      get => hasValue;
   }
   public readonly T Value {
      get { ... }
   }
   . . .
}
```

Die Struktur ist als **partial** gekennzeichnet, weil ihre Implementation auf zwei Dateien verteilt ist, was für uns aktuell ohne Belang ist (vgl. Abschnitt 5.13.6).

In der **where**-Klausel wird dafür gesorgt, dass der Typformalparameter nur durch eine Struktur konkretisiert werden darf.

Eine Instanz einer Konkretisierung der generischen Struktur **Nullable<T>** kann einen Wert vom Strukturtyp **T** (z. B. einen Wert eines elementaren Datentyps) verpacken, aber auch den Ausnahmewert **null** annehmen. Z. B. kann eine Variable vom Typ **Nullable<bool>** neben den **bool**-Werten **true** und **false** auch den dritten Wert **null** annehmen, der als *unbekannt* zu interpretieren ist.

In der folgenden Anweisung wird eine Variable vom Typ **Nullable<bool>** deklariert:

```
Nullable<bool> status;
```

---

[1] Wie man den BCL-Quellcode einsehen kann, erläutert der Abschnitt 2.6.4.

Man darf einer **Nullable**-Variablen 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 bezeichnen, z. B.:

```
bool? status;
```

Eine **Nullable**-Instanz informiert in der booleschen get-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 Verpackung 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); // Ausgabe 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 immer **false** ist, z. B. beim Vergleich:

```
0 == null
```

Ein (statisches) Feld mit **Nullable**-Typ hat den automatischen Initialisierungswert **null**, z. B.:

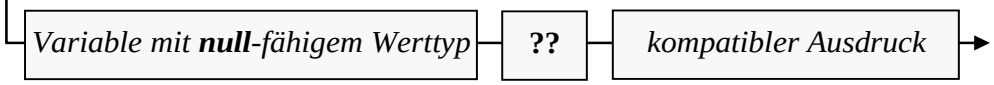
Quellcode	Ausgabe

`class Prog {`
`  int i;`
`  int? ni;`
`  static void Main() {`
`  var p = new Prog();`
`  Console.WriteLine(p.i + "\n" + p.ni.HasValue);`
`  }`
`}`
	`0`
	`False`

Von dieser Regel sind auch die Elemente eines Arrays betroffen, weil sie als Instanzvariablen aufgefasst werden können.

Mit dem sogenannten **null-Koaleszenzoperator** (engl.: *null-coalescing operator*), 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:[1]

null-Koaleszenzoperator



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. Beispiele zur Verwendung des **??** - Operators:

Quellcodesegment	Ausgabe

`int? ni = 4;`
`int i = ni ?? -1;`
`Console.WriteLine(i);`
`ni = null;`
`i = ni ?? -1;`
`Console.WriteLine(i);`
	`4`
	`-1`

Die Anweisung

```
int i = ni ?? -1;
```

ist äquivalent zu:

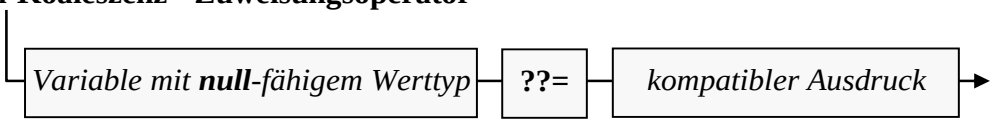
```
int i = (ni != null) ? (int)ni : -1;
```

Der Null-Koaleszenzoperator kann auch auf Referenztypen angewendet werden, z. B.:

```
string s = null;
bool shortOrNull = (s ?? "").Length < 50;
```

Seit C# 8 wird mit dem sogenannten *null-Koaleszenz - Zuweisungsoperator* (engl.: *null-coalescing assignment operator*) das Zusammenwachsen (Bedeutung von *Koaleszenz*) der gewöhnlichen und der **null**-fähigen Werttypen zusätzlich gefördert:[2]

null-Koaleszenz - Zuweisungsoperator



Der Variablen mit **null**-fähigem Werttyp wird genau dann der Wert des kompatiblen rechten Ausdrucks zugewiesen, wenn sie aktuell den Wert **null** besitzt.
Beispiele:

---

[1] Microsoft verwendet unterschiedliche Bezeichnungen:
- *Null-Sammeloperator*
  https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/builtin-types/nullable-value-types
- *NULL-Zusammenfügungsoperator*
  https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/operators/null-coalescing-operator

Weil *Koaleszenz* für das *Zusammenwachsen* (von Grundtyp und Nullable-Variante) steht, ist die Bezeichnung *Null-Koaleszenzoperator* zu bevorzugen.

[2] Microsoft schlägt *Null-Sammelzuweisungsoperator* als deutsche Bezeichnung vor.
https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/operators/null-coalescing-operator.

Quellcode	Ausgabe

`int? ni = null;`
`ni ??= -1;`
`Console.WriteLine(ni);`
`ni ??= 0;`
`Console.WriteLine(ni);`
	`-1`
	`-1`

Die Anweisung

```
ni ??= -1;
```

ist äquivalent zu:

```
if (ni == null) ni = -1;
```

Der Null-Koaleszenz - Zuweisungsoperator kann auch auf Referenztypen angewendet werden, z. B.:

```
string s = null;
s ??= "";
```

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.:

Quellcode	Ausgabe
`double? d = 3.0;` `Console.WriteLine(d.GetType());`	`System.Double`

Weitere Beispiele für generische Strukturen (mit spezieller syntaktischer Unterstützung durch den Compiler) haben wir übrigens im Zusammenhang mit den Tupeltypen kennengelernt (siehe Abschnitt 6.6.1).

8.4 Generische Typen zur Laufzeit

Verwendet ein Programm einen generischen Typ mit einem Typformalparameter (z. B. **List<T>**), dann verhält sich die Laufzeitumgebung folgendermaßen:[1]

- Beim ersten Auftreten einer geschlossenen Konstruktion mit einem *Werttyp* als Aktualparameter (z. B. **List<int>**) wird im Speicher ein eigener Typ mit passender Ersetzung des Typformalparameters erstellt. Für jeden anderen Werttyp als Aktualparameter muss eine zusätzliche Konkretisierung mit passender Ersetzung des Typformalparameters erstellt werden.
- Beim ersten Auftreten einer geschlossenen Konstruktion mit einem *Referenztyp* als Aktualparameter (z. B. **List<String>**) wird ein Typ mit Ersetzung des Typformalparameters durch die Klasse **Object** erstellt. Diese Variante kann aber für *jede* geschlossene Konstruktion mit einem anderen *Referenz*-Typaktualparameter verwendet werden.

Über einen konkretisierten generischen Typ sind alle Laufzeitinformationen vorhanden, sodass z. B. eine Typprüfung per **is**-Operator möglich ist:

[1] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generics-in-the-run-time

Quellcode	Ausgabe
`public class SimpleStack<T> {` `  . . .` `}` `class Prog {` `  public static void Main() {` `  var sist= new SimpleStack<string>();` `  Console.WriteLine(sist.GetType());` `  Console.WriteLine(sist is SimpleStack<string>);` `  }` `}`	`SimpleStack`1[System.String]` `True`

8.5 Generische Methoden

Im Vergleich zu mehreren überladenen Methoden (vgl. Abschnitt 5.3.5), 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 **IComparable<T>** (vgl. Abschnitt 8.2.2) erfüllen, also eine Methode mit dem Definitionskopf

public int CompareTo(T *element***)**

besitzen muss:

Quellcode	Ausgabe

`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"));`
 	
`static T Max<T>(T x, T y) where T : IComparable<T> {`
`  return x.CompareTo(y) >= 0 ? x : y;`
`}`
	`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 Typformalparameter werden durch Kommata getrennt. Ein Typformalparameter ist als Datentyp erlaubt für:

- den Rückgabewert
- Parameter
- lokale Variablen

Zur Formulierung von Restriktionen für Typformalparameter dient wie bei generischen Typen das Schlüsselwort **where**.

Verwendet eine Methode einer *generischen* Klasse einen Typformalparameter der Klasse (als Parametertyp, Rückgabetyp oder Datentyp für eine lokale Variable), dann spricht man *nicht* von einer generischen Methode, weil keine *eigenen* Typformalparameter definiert werden, z. B. bei der Methode `Push()` der im Abschnitt 8.2.1 beschriebenen Klasse `SimpleStack<T>`:

```
public void Push(T element) {
  . . .
}
```

Die verwendeten Typformalparameter sind folglich *nicht* hinter dem Methodennamen zwischen spitzen Klammern anzugeben.

Beim Aufruf einer generischen Methode kann der Compiler fast immer aus den Datentypen der Aktualparameter die passende Konkretisierung ermitteln (Typinferenz). Daher konnte im letzten 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 11.6.3 beschrieben (ECMA 2022).

Der im Zusammenhang mit dem Überladen, Ersetzen und Überschreiben von Methoden relevante Begriff der *Methodensignatur* muss zur Berücksichtigung der Generizität angepasst werden (vgl. Abschnitt 5.3.5). Zwei Methoden besitzen genau dann dieselbe *Überladungssignatur*, wenn die folgenden Bedingungen erfüllt sind:

- Die Namen der Methoden sind identisch.
- Die Typformalparameterlisten sind gleich lang.
- Die Formalparameterlisten sind gleich lang.
- Positionsgleiche Formalparameter stimmen hinsichtlich Datentyp und Transfermodus (Wert- oder Verweisparameter) überein.

Irrelevant für die Überladungssignatur einer Methode sind (vgl. ECMA 2022, Abschnitt 7.6):

- Der Rückgabetyp
- Die Modifikatoren
- Die *Namen* der Formal- bzw. Typformalparameter
- Das beim letzten Formalparameter erlaubte **params**-Schlüsselwort.

Für die beim Ersetzen bzw. Überschreiben von Basisklassenmethoden in abgeleiteten Klassen relevante *Ersetzungssignatur* sind auch die Richtungsmodifikatoren (**ref**, **in**, **out**) von Formalparametern signifikant (siehe Abschnitte 7.6.1 bzw. 7.9).

8.6 default-Operator

In einer Methode eines generischen Typs oder in einer generischen Methode ist es gelegentlich erforderlich, den typspezifischen Nullwert 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] Der Ausdruck **default(T)** liefert ...

- den Wert **null**, wenn beim Konkretisieren des generischen Typs **T** ein Referenztyp oder ein **Nullable**-Strukturtyp (vgl. Abschnitt 8.3) angegeben wurde,
- den typspezifischen Nullwert, wenn für **T** ein elementarer Datentyp angegeben wurde,

[1] Das Schlüsselwort **default** wird in anderer Bedeutung (zur Bezeichnung von beliebigen sonstigen Werten des steuernden Ausdrucks) auch in der **switch**-Anweisung verwendet (siehe Abschnitt 4.7.2.3.1).

- eine Strukturinstanz mit typspezifischen Nullwert-Initialisierungen für alle Felder, wenn für **T** ein Strukturtyp angegeben wurde:
  - Felder mit elementarem Datentyp erhalten den typspezifischen Nullwert.
  - Felder mit einem Referenztyp oder mit einem **Nullable**-Strukturtyp erhalten den Wert **null**.

  Für Werttypen liefert der **default**-Operator dasselbe Ergebnis wie der Standard- bzw. default-Konstruktor (ECMA 2022. Abschnitt 8.3.3). Das gilt auch für Strukturen, die einen expliziten parameterfreien Konstruktor erhalten haben (siehe Abschnitt 6.1.2).

Als Argument des **default**-Operators sind erlaubt:

- der Name eines Typs, z. B.:

  ```
  var c = default(System.Numerics.Complex);
  Console.WriteLine(c); // Ausgabe: <0; 0>
  ```

  Die BCL-Struktur **Complex** im Namensraum **System.Numerics** stellt komplexe Zahlen im Sinne der mathematischen Analysis dar, die aus einem Real- und einem Imaginärteil bestehen. Mathematisch nicht vorbelastete Leser dürfen sich unter einer komplexen Zahl ein Paar aus zwei reellen Zahlen vorstellen. Zur Erstellung einer nullinitialisierten Instanz müsste man statt des **default**-Operators einen Konstruktor verwenden. Zur **default**-Lösung wird gleich noch eine Vereinfachung vorgestellt.

  Wird für eine zu initialisierende Variable der Datentyp explizit angegeben, dann ist für den **default**-Parameter kein Argument erforderlich, z. B.:

  ```
  System.Numerics.Complex c = default;
  ```

- ein Typformalparameter

Der zweite Fall ist der eindeutig wichtigere, weil das Ergebnis des **default**-Ausdrucks in Abhängigkeit von der Konkretisierung des Typformalparameters variiert. Das folgenden Programm mit der generischen Methode `WriteDef<T>()`

```
struct MitNullInt {
  int i;
  int? inu;
  public MitNullInt(int i, int? inu) {
    this.i = i; this.inu = inu;
  }
  public override string ToString() {
    return $"({i}, {(inu == null ? "null" : inu.ToString())})";
  }
}

class DefaultDemo {
  static void WriteDef<T>() {
    var def = default(T);
    Console.WriteLine(
      $"default of {typeof(T),-40} {(def == null ? "null" : def.ToString()),-20}");
  }

  static void Main() {
    WriteDef<int>();
    WriteDef<System.Numerics.Complex>();
    WriteDef<string>();
    WriteDef<int?>();
    WriteDef<MitNullInt>();
  }
}
```

gibt den **default**-Wert aus für:

- den numerischen Werttyp **int**
- den Strukturtyp **System.Numerics.Complex** für komplexe Zahlen
- den Referenztyp **String**
- den konkretisierten generischen Strukturtyp **Nullable<int>** (alias: **int?**)
- den selbst definierten Strukturtyp `MitNullInt`

Es resultiert die Ausgabe:

```
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)
```

Seit C# 7.1 kann im Ausdruck **default(T)** die Typangabe entfallen, wenn sie vom Compiler eindeutig zu ermitteln ist. Damit lässt sich z. B. die obige Null-Initialisierung eines **Complex**-Werts vereinfachen:

```
System.Numerics.Complex c = default;
```

Microsoft spricht hier vom **default**-Literal und empfiehlt seine Verwendung bei optionalen Parametern, z. B.:[1]

```
T[] InitializeArray<T>(int length, T initialValue = default) {
 . . .
}
```

8.7 Übungsaufgaben zum Kapitel 8

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. In C# können Typformalparameter auch durch Werttypen konkretisiert werden.
2. Wenn eine Methode einer generischen Klasse einen Typformalparameter aus dem Klassendefinitionskopf als Rückgabetyp verwendet, liegt eine generische Methode vor.
3. Weil **Object** eine Basisklasse von **String** ist, kann ein Objekt vom Typ **List<String>** durch eine Variable vom Typ **List<Object>** verwaltet werden.
4. Eine generische Klasse darf von einer nicht-generischen Klasse abstammen.

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** zeitaufwändige (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.

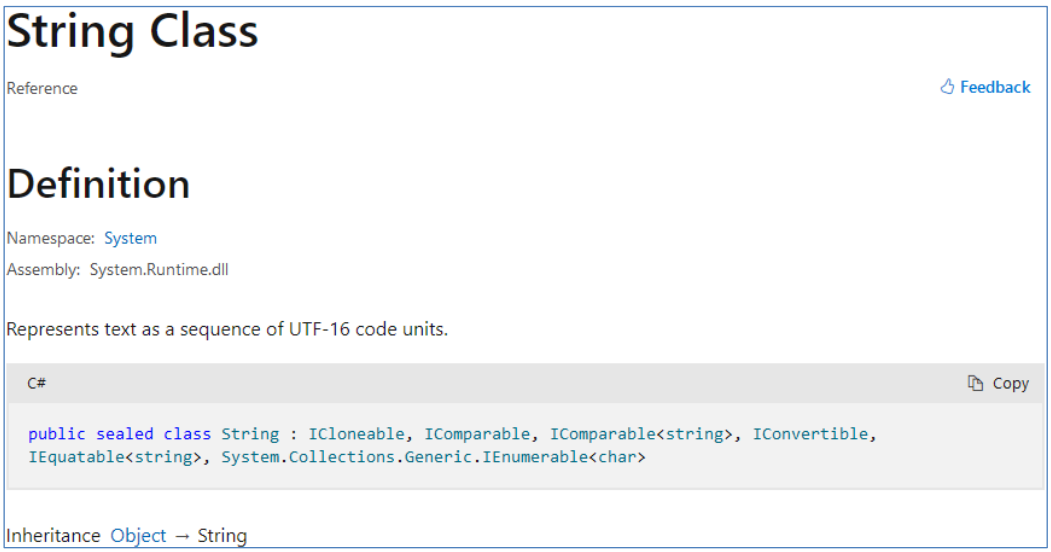
3) Erstellen Sie eine verbesserte Version der generischen Methode `Max<T>()` aus dem Abschnitt 8.5, die einen generischen Serienparameter (siehe Abschnitt 5.3.1.3.3) akzeptiert und das maximale Element liefert.

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/default

9 Interfaces

Zu vielen Klassen führt die BCL-Dokumentation hinter dem Namen und einem Doppelpunkt *mehrere* Typen auf, z. B. bei der Klasse **String**:[1]


String Class

Reference Feedback

Definition

Namespace: System

Assembly: System.Runtime.dll

Represents text as a sequence of UTF-16 code units.

C# Copy

```
public sealed class String : ICloneable, IComparable, IComparable<string>, IConvertible,
IEquatable<string>, System.Collections.Generic.IEnumerable<char>
```

Inheritance Object → String

Bei den sechs aufgelisteten Typen kann es sich nicht um Basisklassen handeln, weil C# keine Mehrfachvererbung unterstützt. Außerdem ist der BCL-Dokumentation zu entnehmen, dass die Klasse **String** direkt von der Urahnklasse **Object** abstammt. Am Anfangsbuchstaben *I* sind in der BCL-Dokumentation zuverlässig die von einer Klasse oder Struktur implementierten *Interfaces* (dt.: *Schnittstellen*) zu erkennen. Hierbei handelt es sich um **Verpflichtungserklärungen** gegenüber dem Compiler. Ein Interface definiert eine Reihe von Handlungskompetenzen abstrakt (ohne Implementierung) über Definitionsköpfe von Methoden, Eigenschaften, Indexern und Ereignissen.[2] 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 dieses Interface als Datentyp vorgeschrieben ist.

Die Liste der von einem Typ implementierten Schnittstellen liefert 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 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, sodass die Typsicherheit gewährleistet ist, und lästige Typumwandlungen entfallen.
  Weil **String** die Schnittstelle **IComparable** implementiert, muss eine Methode
  public int CompareTo(Object *obj***)**
  vorhanden sein.[3] Um den Vertrag **IComparable<string>** zu erfüllen, wird eine Methode mit dem folgenden Definitionskopf benötigt:

---

1 https://learn.microsoft.com/en-us/dotnet/api/system.string

2 Ereignisse werden im Kapitel 10 behandelt.

3 Aus Kompatibilitätsgründen hält die Klasse **String** am Bekenntnis zur nicht-generischen Schnittstelle **IComparable** fest, sodass 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**:

```
Unhandled exception. System.ArgumentException: Object must be of type String.
```

public int CompareTo(string *str*)

Aus dem Abschnitt 6.3.1.2.2 ist bekannt, dass die Klasse **String** eine solche Methode besitzt, und wie **CompareTo()** das Prüfergebnis über den Rückgabewert signalisiert.

Weil **String**-Objekte die Fähigkeit zum Vergleich mit Artgenossen besitzen, kann z. B. ein Array mit Elementen dieses Typs über die statische Methode **Array.Sort()** sortiert werden:

Quellcode	Ausgabe

`string[] star = { "eins", "zwei", "drei" };`
`Array.Sort(star);`
`foreach (string s in star)`
`  Console.WriteLine(s);`
	`drei`
	`eins`
	`zwei`

Erfüllt der Elementtyp eines Arrays weder die Schnittstelle **IComparable** noch die generische Schnittstelle **IComparable<T>**, dann endet ein Sortierversuch mit der Fehlermeldung:

```
Unhandled exception. System.InvalidOperationException: Failed to compare two
elements in the array.
```

Das passiert auch dann, wenn ein Elementtyp zwar eine geeignete **CompareTo()** – Methode besitzt, aber in seinem Definitionskopf die erfüllte Schnittstelle nicht bekanntgibt.

- **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

`string s1 = "eins";`
`string s2 = (string)s1.Clone();`
`Console.WriteLine(s2);`
	`eins`

  Zum Interface **ICloneable** gibt es keine generische Alternative. Die **Clone()** - Methode und damit das **ICloneable**-Interface besitzen wegen einer Implementierungsunklarheit keinen guten Ruf. Es geht um den Unterschied zwischen der *tiefen* Kopie (engl.: *deep copy*), die auch alle direkten und indirekten Member-Objekte dupliziert, und der *flachen* Kopie (engl.: *shallow copy*), die Member-Objekte des Originals weiterverwendet. Die **IClonable**-Schnittstelle lässt das Verfahren offen:[1]

  > It does not specify whether the cloning operation performs a deep copy, a shallow copy, or something in between. Nor does it require all property values of the original instance to be copied to the new instance.

  Um **Clone()** zur Erstellung einer flachen Kopie zu implementieren, kann man auf die von der Klasse **Object** geerbte Methode **MemberwiseClone()** (mit Sichtbarkeit **protected**) zurückgreifen, z. B.:

```
public object Clone() {
  return MemberwiseClone();
}
```

- **IEquatable<string>**

  Diese Schnittstelle schreibt eine Methode namens **Equals()** und einem Parameter vom eigenen Typ vor, sodass zwei Objekte eines implementierenden Typs auf Gleichheit beurteilt werden können:

  public bool Equals(string *str*)

[1] https://learn.microsoft.com/en-us/dotnet/api/system.icloneable

Die Schnittstelle **IEquatable<T>** bzw. die Methode **Equals()** sind relevant für alle generische Kollektionsklassen, die Gleichheitsprüfungen für ihre Elemente vornehmen (z. B. **HashSet<T>**, **Dictionary<TKey, TValue>**, siehe Kapitel 11). Im folgenden Beispiel wird für eine Zeichenfolge geprüft, ob sie in einer Menge (einer Kollektion vom Typ **HashSet<string>**) vorhanden ist:[1]

Quellcode	Ausgabe
`var hs = new HashSet<string>() { "one", "too", "three" };` `Console.WriteLine(hs.Contains("two"));`	`False`

- **IEnumerable<char>**
  Mit dieser Schnittstelle werden wir uns im Abschnitt 9.5 näher beschäftigen, sodass hier kurze Hinweise genügen. Weil die Klasse **String** die konkretisierte (geschlossene konstruierte) Schnittstelle **IEnumerable<char>** implementiert, kann man in einer **foreach**-Schleife über die Zeichen in einem **String**-Objekt iterieren (vgl. Abschnitt 4.7.3.2), z. B.:

Quellcode	Ausgabe

`string abc = "abc";`
`foreach (char c in abc)`
`  Console.WriteLine((int)c);`
	`97`
	`98`
	`99`

Seit C# 8 sind in Schnittstellen neben den *abstrakt* definierten Verhaltenskompetenzen, die ein implementierender Typ selbst realisieren *muss*, auch *konkrete* Methoden mit einem vollständigen Rumpf (Anweisungsblock) erlaubt (siehe Abschnitt 9.1.1.2). Ein implementierender Typ kann eine konkrete Methode unverändert nutzen oder durch eine eigene Lösung überschreiben. Durch die in C# 8 eingeführten konkreten Methoden soll verhindert werden, dass durch die Ergänzung neuer Schnittstellenmethoden die bisher abgegebenen Verpflichtungserklärungen implementierender Typen ungültig werden.

Damit besteht neben den Erweiterungsmethoden (vgl. Abschnitt 7.15) eine zweite Möglichkeit, die eine Schnittstelle implementierenden Typen mit zusätzlicher Funktionalität auszustatten. Primär dienen Schnittstellen aber weiterhin dem Zweck, Verhaltenskompetenzen von Typen abstrakt zu definieren.

Eine Schnittstelle ist ein **Datentyp**. Es lassen sich zwar keine *Instanzen* von diesem Typ erzeugen, aber *Referenzvariablen* sind erlaubt und als Abstraktionsmittel sehr nützlich. Sie dürfen auf Instanzen beliebiger Typen zeigen, die die Schnittstelle implementieren. Somit können Instanzen unabhängig von den Vererbungsbeziehungen ihrer Typen gemeinsam verwaltet werden (z. B. in einem Array). Die von einer Schnittstelle über abstrakte (und damit virtuelle) Methoden vorgeschriebenen Handlungskompetenzen werden von den implementierenden Typen individuell realisiert. Methodenaufrufe über Schnittstellenvariablen erfolgen **polymorph** (mit später bzw. dynamischer Bindung, siehe Abschnitt 7.9).

Implementiert eine Klasse oder eine Struktur ein Interface, dann …

- muss sie die im Interface abstrakt enthaltenen Methoden, Eigenschaften, Indexer und Ereignisse konkret realisieren,
- 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.:

---

[1] Im Vorgriff auf Abschnitt 11.3.1 wird ein Kollektionsinitialisierer verwendet, was wegen der plausiblen Syntax vor dem Hintergrund der Erfahrung mit dem analogen Array-Initialisierer didaktisch akzeptabel ist.

- **Implementierung von vorhandenen Schnittstellen in einer eigenen Typdefinition**
  Wenn z. B. unsere Klasse `Bruch` das Interface **IComparable<`Bruch`>** implementiert, dann können wir die bequeme Methode **Array.Sort()** verwenden, um ein Array mit `Bruch`-Objekten zu sortieren.
- **Schnittstellen als Datentypen in eigenen Methoden- oder Typdefinitionen verwenden**
  In einer eigenen Methodendefinition ist es oft sinnvoll, Parameterdatentypen und/oder den Rückgabetyp über Schnittstellen festzulegen. In den Anweisungen der Methode können Verhaltenskompetenzen von Parameterinstanzen genutzt werden, die durch Schnittstellenverpflichtungen garantiert sind. Bei einer Schnittstelle als Rückgabetyp weiß der Aufrufer, dass er ein Objekt mit bestimmten Verhaltenskompetenzen erhält; die genaue Klassenzugehörigkeit muss der Aufrufer nicht kennen. Damit wird die Typsicherheit ohne überflüssige Einengung erreicht. Ein Beispiel für eine Methode mit einer generischen Schnittstelle als Parametertyp ist die folgende Überladung der Methode **Sort()** in der generischen BCL-Klasse **List<T>**:

  public void Sort(IComparer<T> *comparer***)**

  Ein die Schnittstelle **IComparer<T>** erfüllendes Objekt kann durch die folgende Methode aufgefordert werden, die Anordnung von zwei Elementen vom Typ **T** über den Rückgabewert zu signalisieren:

  public int Compare (T *x***, T** *y***)**

  Auch bei Instanzvariablen oder statischen Variablen steigern Interface-Datentypen die Flexibilität.

  Sind bei der Definition eines generischen Typs für einen beschränkten Typformalparameter bestimmte Verhaltenskompetenzen zu fordern, dann gelingt das oft am besten per Schnittstellendatentyp (siehe Abschnitt 8.2.2).
- **Schnittstellen definieren**
  Zur Verwendung als Datentypen in Methoden- oder Typdefinitionen kommen natürlich auch selbst definierte Schnittstellen in Frage.

9.1 Interfaces definieren

Wir behandeln zuerst das im Programmieralltag vergleichsweise seltene Definieren einer Schnittstelle, weil man dabei einen guten Eindruck von den Bestandteilen einer Schnittstelle und von ihrer Rolle bei der objektorientierten Programmierung gewinnt. Allerdings verzichten wir auf ein eigenes Beispiel und betrachten stattdessen die angenehm einfach aufgebaute und außerordentlich wichtige Schnittstelle **IComparable<T>** aus der BCL:[1]

```
public interface IComparable<in T> {
  // Interface does not need to be marked with the serializable attribute
  /// <summary>Compares the current instance with another object of the same type and
returns an integer that indicates whether the current instance precedes, follows, or
occurs in the same position in the sort order as the other object.</summary>
  int CompareTo(T other);
}
```

Wie der (Dokumentations-)Kommentar im obigen Quellcode zeigt, sind bei einer Schnittstellen-Definition neben den syntaktischen Forderungen meist auch Verhaltenserwartungen im Spiel. Der Compiler kann aber z. B. aufgrund der obigen **IComparable<T>** - Definition sinnlose **CompareTo()** - Implementierungen *nicht* verhindern.

---

[1] Wenn der Name einer Schnittstelle mit einem kovarianten bzw. kontravarianten Typformalparameter (siehe Abschnitt 9.1.4) im Text auftaucht (z. B. **IComparable<T>**), dann wird das in der Definition (als sogenannter *Varianzmodifikator*) erforderliche Schlüsselwort **out** bzw. **in** meist weggelassen.

Bei Schnittstellendefinitionen sind einige Regeln zu beachten:

- Zugriffsmodifikatoren
  Bei einem Top-Level - Interface sind die Zugriffsmodifikatoren **public**, **internal** und **file** erlaubt. Die voreingestellte Schutzstufe **internal** schränkt die Verwendung auf das eigene Assembly ein. 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 **interface** dient zur Unterscheidung von Klassen- oder Strukturdefinitionen.
- Schnittstellenname
  Per Konvention beginnt der Interfacename mit einem großen *I*.

  Bei generischen Schnittstellen folgen auf den Namen die Typformalparameter zwischen spitzen Klammern und durch Kommata getrennt. Wie bei generischen Klassen und Strukturen können auch bei generischen Schnittstellen Restriktionen für die Typformalparameter formuliert werden (vgl. Abschnitt 8.2.2).

  Mit der Kennzeichnung eines Typformalparameters als kovariant bzw. kontravariant über die sogenannte *Varianzmodifikatoren* **out** bzw. **in** beschäftigen wir uns im Abschnitt 9.1.4.
- Erlaubte Interface-Member
  Als Interface-Member sind *instanzbezogene* Methoden, Eigenschaften, Indexer und Ereignisse erlaubt. Verboten sind instanzbezogene Konstruktoren und Felder. Seit C# 8 sind hinzugekommen:
  - statische Methoden, Eigenschaften und Ereignisse
  - statische Felder
  - ein statischer Konstruktor
  - geschachtelte Typen

  Trotz mancher Änderungen am Schnittstellenkonzept von C# bleibt es dabei, dass das Verhalten von Instanzen vorgeschrieben werden soll, nicht deren Zustand.
- Generell erlaubte Modifikatoren für Interface-Member
  Für alle Interface-Member ist die Schutzstufe **public** voreingestellt. Der Modifikator ist meist überflüssig, aber zulässig. Seit C# 8 sind außerdem die Zugriffsmodifikatoren **protected** und **internal** generell erlaubt. Informationen über die zulässigen Modifikatoren für *spezielle* Member folgen in den Abschnitten 9.1.1, 9.1.2 und 9.1.3.[1]

Die Liste der Regeln zur Schnittstellen-Definition ist im Verlauf der C# - Entwicklung sukzessive angewachsen. Für den primären Zweck von Schnittstellen sind aber nur wenige Regeln relevant (siehe das obige Beispiel **IComparable<in T>**).

[1] Microsoft gibt auf der folgenden Webseite einige Hinweise zu den Modifikatoren für Interface-Member: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods

9.1.1 Instanzmethoden

Die anschließenden Erläuterungen gelten analog auch für Eigenschaften, Indexer und Ereignisse.

9.1.1.1 Abstrakte Instanzmethoden

Durch abstrakte Instanzmethoden werden Verhaltenskompetenzen beschrieben, die implementierende Typen besitzen müssen. Auf den Methodendefinitionskopf folgt an Stelle des durch geschweifte Klammern begrenzten Rumpfs ein Semikolon, z. B.:

```
int CompareTo(T other);
```

Eine abstrakte Instanzmethode ist implizit **public** und **abstract**. In der Regel kann man auf explizite Modifikatoren verzichten und muss dementsprechend die folgenden Regeln nicht beachten:

- Die Modifikatoren **abstract** und **public** sind überflüssig, aber erlaubt.
- Als Zugriffsmodifikatoren sind alternativ zu **public** auch **protected** und **internal** möglich.

Bei der Implementierung einer abstrakten Schnittstellenmethode (siehe Abschnitt 9.2) ist zu beachten:

- Die Schutzstufe **public** muss verwendet und explizit deklariert werden.
- Der Modifikator **override** muss und darf *nicht* angegeben werden.

Beim ersten Lesen ist es vertretbar, an dieser Stelle zum Abschnitt 9.2 zu springen. Die teilweise komplexen Details in den ausgelassenen Abschnitten sollten Sie sich erst dann zumuten, wenn es zum Verständnis von wichtigen Diskussionen oder Beispielen erforderlich ist.

9.1.1.2 Konkrete Instanzmethoden

Seit C# 8 ist es möglich, Instanzmethoden *mit* Implementation (also mit einem ausführbaren Methodenrumpf) in eine Schnittstelle aufzunehmen. Wir werden anschließend meist von *konkreten Schnittstellenmethoden* sprechen. Implementierende Typen können eine konkrete Interface-Methode unverändert verwenden oder überschreiben.

Ein wesentliches Motiv für die Einführung der konkreten Schnittstellenmethoden bestand darin, die nachträgliche Erweiterung von Schnittstellen um neue Methoden ohne Nachteil für vorhandene implementierende Typen zu ermöglichen. Bis zur Version 7.x wurde in C# eine Möglichkeit vermisst, vorhandene Interfaces um neue Instanzmethoden zu erweitern, ohne die Kompatibilität mit implementierenden Typen zu verlieren. Seit C# 8 ist das Problem durch die Möglichkeit zur Erweiterung von Schnittstellen um konkrete Methoden gelöst. Vorhandene implementierende Typen erfüllen auch ein um konkrete Methoden erweitertes Interface, weil es ihnen keine zusätzlichen Pflichten aufbürdet.

Die in der Programmiersprache Java seit der Version 8 (eingeführt im März 2014) unterstützten **default**-Schnittstellenmethoden (siehe z. B. Baltes-Götz & Götz 2023) waren eventuell ein Vorbild für die Einführung der konkreten Schnittstellenmethoden in C# 8 (im September 2019).[1] In der Literatur zu C# ist jedenfalls oft von *default interface methods* statt von *concrete interface methods* die Rede.

Wir betrachten ein einfaches Beispiel mit einer Schnittstelle namens `IWinterface`

[1] Durch die Erweiterung um implementierte Schnittstellenmethoden greifen Java und C# das seit längerer Zeit in der Informatik diskutierte *Trait*-Konzept auf (Schärli et al. 2003). Dabei geht es um die Wiederverwendung von Methoden durch Typen, die in keiner Vererbungsbeziehung stehen müssen.

```
interface IWinterface {
   void SagA();
}
```

und einer implementierenden Klasse namens `Cimple`:

```
class Cimple : IWinterface {
   public void SagA() {
      Console.WriteLine("A");
   }

   static void Main() {
      var cimpl = new Cimple();
      cimpl.SagA();
   }
}
```

Eine Klasse oder Struktur muss das Bekenntnis zu einer Schnittstelle im Definitionskopf ankündigen (siehe Abschnitt 9.2):

- Auf den Namen der Klasse oder Struktur folgt ein Doppelpunkt.
- Dann folgt der Name der Schnittstelle.

Im Beispiel soll die Schnittstelle `IWinterface` um eine Instanzmethode `SagB()` erweitert werden, ohne alte Klassen (z. B. `Cimple`) ändern zu müssen. Dazu ergänzt man `SagB()` als *konkrete* Instanzmethode:

```
interface IWinterface {
   void SagA();

   void SagB() {
      SagA();
      Console.WriteLine("B");
   }
}
```

In einer konkreten Schnittstellenmethode dürfen andere Schnittstellenmethoden verwendet werden (abstrakte und konkrete, instanzbezogene und statische, öffentliche und private).

Implementiert eine bestehende Klasse eine neuerdings um eine konkrete Methode erweiterte Schnittstelle, dann bleibt die Klasse kompatibel zum erweiterten Interface, d. h. die Klasse muss weder verändert noch neu übersetzt werden.

Wenn ein Objekt der bestehenden Klasse unter Verwendung einer Referenzvariablen vom Typ der konkret erweiterten Schnittstelle die neue und fremde Methode ausführt, dann muss das nicht unbedingt harmlos enden.[1] Ein vergleichbares Risiko besteht aber auch …

- für eine abgeleitete Klasse bei der Erweiterung der Basisklasse um eine zusätzliche Methode
- und bei der Definition von Erweiterungsmethoden für eine Klasse, Struktur oder Schnittstelle (siehe Abschnitt 9.1.1.3).

Eine neue bzw. aktualisierte Klasse, die das Interface implementiert, kann die konkrete Methode unverändert nutzen oder durch eine eigene Implementation überschreiben. Im folgenden Beispiel wird die *erste* Option verwendet. Es ist zu beachten, dass ein Objekt der Klasse `Cimple` über eine Referenz vom Typ der Schnittstelle `IWinterface` angesprochen werden muss, um die konkrete Schnittstellenmethode ausführen zu können:

---

[1] Zu den ähnlich realisierten **default**-Schnittstellenmethoden in Java sagt J. Bloch (2018, Item 21):
But adding new methods to existing interfaces is fraught with risk.

Quellcode	Ausgabe

`class Cimple : IWinterface {`
`  public void SagA() {`
`  Console.WriteLine("A");`
`  }`
 	
`  static void Main() {`
`  var cimple = new Cimple();`
`  ((IWinterface)cimple).SagB();`
`  }`
`}`
	`A`
	`B`

Eine implementierende Klasse **erbt keine konkreten Schnittstellenmethoden**, sodass der folgende Aufruf scheitert:

```
cimple.SagB();
```

CS1061: "Cimple" enthält keine Definition für "SagB", und es konnte keine zugängliche SagB-Erweiterungsmethode gefunden werden, die ein erstes Argument vom Typ "Cimple" akzeptiert (möglicherweise fehlt eine using-Direktive oder ein Assemblyverweis).

Mögliche Korrekturen anzeigen (Alt+Eingabe oder Strg+.)

Wenn eine Klasse eine konkrete Schnittstellenmethode implementiert und dabei den **public**-Zugriff beibehält, dann wird die eigene Implementation auch beim Aufruf über eine Interface-Referenz verwendet (Polymorphie):

Quellcode	Ausgabe

`class Cimple : IWinterface {`
`  public void SagA() {`
`  Console.WriteLine("A");`
`  }`
 	
`  public void SagB() {`
`  Console.WriteLine("B von Cimple");`
`  }`
 	
`  static void Main() {`
`  IWinterface cimple = new Cimple();`
`  cimple.SagB();`
`  }`
`}`
	`B von Cimple`

Daher ist es vertretbar, von einer *Überschreibung* zu sprechen, obwohl dieser Begriff für eine abgeleitete Klasse und eine Basisklassenmethode geprägt wurde, während es aktuell um eine implementierende Klasse und eine Schnittstellenmethode geht. Implementiert ein Typ eine konkrete, öffentliche und nicht versiegelte Schnittstellenmethode, dann muss und darf der Modifikator **override** *nicht* angegeben werden. Dieser Modifikator ist auch beim Implementieren von abstrakten Schnittstellenmethoden verboten (siehe Abschnitt 9.1.1.1).

Die Dominanz der eigenen Implementation beim Methodenaufruf per Schnittstellenreferenz gilt auch dann, wenn die eigene Implementation von der Basisklasse geerbt wurde.

Implementiert ein Typ *mehrere* Schnittstellen mit signaturgleichen konkreten Methoden, und verwendet er diese Methoden ohne eigene Realisierung, dann entscheidet der deklarierte Referenzvariablentyp darüber, welche Methode von einer Instanz ausgeführt wird, z. B.:

Quellcode	Ausgabe

`interface IWinterface {`
`  void SagWas() {`
`  Console.WriteLine("Winter");`
`  }`
`}`
 	
`interface IOtherface {`
`  void SagWas() {`
`  Console.WriteLine("Other");`
`  }`
`}`
 	
`class Cimple : IWinterface, IOtherface {`
`  static void Main() {`
`  Cimple cimple = new();`
`  (cimple as IWinterface).SagWas();`
`  (cimple as IOtherface).SagWas();`
`  }`
`}`
	`Winter`
	`Other`

Wenn ein implementierender Typ eine konkrete Schnittstellenmethode selbst realisiert, dann darf die Sichtbarkeit **public** durch Vergabe eines Zugriffsmodifikators reduziert werden, weil ja im Unterschied zu einer abstrakten Schnittstellenmethode beim Aufruf über eine Schnittstellenreferenz eine Implementation verfügbar ist. Allerdings kann (wie beim Verdecken von Methoden in abgeleiteten Klassen, vgl. Abschnitt 7.6.1) ein schlecht nachvollziehbares Programmverhalten resultieren. Lässt der implementierende Typ z. B. den Modifikator **public** weg, dann entsteht eine private Methode. In anderen eigenen Methoden wird …

- über eine Referenz vom eigenen Typ die private Implementation angesprochen,
- bei Verwendung einer Schnittstellenreferenz die konkrete Schnittstellenmethode angesprochen.

Diese potenziell verwirrende Konstellation ist im folgenden Beispiel zu sehen:

Quellcode	Ausgabe

`interface IWinterface {`
`  void SagWas() {`
`  Console.WriteLine("Winter");`
`  }`
`}`
 	
`class Cimple : IWinterface {`
`  void SagWas() {`
`  Console.WriteLine("Cimple");`
`  }`
 	
`  static void Main() {`
`  Cimple cimple = new();`
`  cimple.SagWas();`
`  (cimple as IWinterface).SagWas();`
`  }`
`}`
	`Cimple`
	`Winter`

Weil ein Typ seit C# 8 die Möglichkeit hat, *mehrere* Schnittstellen mit konkreten Methoden zu implementieren, scheint die beim Design von C# bewusst ausgeschlossene Mehrfachvererbung durch eine Hintertür eingedrungen zu sein. Die mit der Mehrfachvererbung verbundenen Risiken bleiben aber ausgeschlossen, denn:

- Ein Typ *erbt* keine konkreten Schnittstellenmethoden. Diese können ohne eigenen Implementation nur über Referenzen vom jeweiligen Schnittstellentyp genutzt werden.
- In Schnittstellen sind Felder generell statisch. Folglich können *Instanzvariablen* nur von der Basisklasse übernommen werden, und der sogenannte *Deadly Diamond of Death* ist ausgeschlossen (siehe Kreft & Langer 2014).

Eine konkrete Instanzmethode einer Schnittstelle ist implizit **public** und **virtual**. Die Modifikatoren **public** und **virtual** sind in der Schnittstellendefinition überflüssig, aber erlaubt.

Als Zugriffsmodifikatoren sind alternativ zu **public** auch **internal**, **protected** und **private** erlaubt.

Eine konkrete Methode mit der Schutzstufe **protected** ist nicht in implementierenden Typen verfügbar, sondern nur in erweiternden Schnittstellen (siehe Abschnitt 9.1.3), z. B.:

```
interface IWinterface {
    protected void SagWas() {
        Console.Write("Winter");
    }
}

interface IWinterfaceExt : IWinterface{
    void SagWasExt() {
        SagWas();
        Console.WriteLine("face");
    }
}

class Cimple : IWinterface {
    static void Main() {
        IWinterface winter = new Cimple();
        winter.SagWas(); // Kein Zugriff erlaubt
    }
}
```

Der Zugriffsmodifikator **private** sorgt dafür, dass eine konkrete Methode nur Schnittstellen-intern (von anderen konkreten Methoden) aufgerufen werden kann und *nicht* **virtual** ist.

Eine mit dem Modifikator **sealed** dekorierte konkrete Methode ist nicht **virtual**, kann also von einem implementierenden Typ *nicht* überschrieben werden, z. B.:

```
interface IWinterface {
    void SagA() {
        Console.WriteLine("A von IWinterface");
    }
    sealed void SagB() {
        Console.WriteLine("B von IWinterface");
    }
}

class Cimple : IWinterface {
    public void SagA() {
        Console.WriteLine("A von Cimple");
    }
    public void SagB() {
        Console.WriteLine("B von Cimple");
    }
    static void Main() {
        IWinterface winter = new Cimple();
        winter.SagA(); // Ausgabe: A von Cimple
        winter.SagB(); // Ausgabe: B von IWinterface
    }
}
```

Bei einer als **private** deklarierten konkreten Methode ist der Modifikator **sealed** überflüssig und verboten.

9.1.1.3 Konkrete ausführbare Schnittstellen-Member im Vergleich zu Erweiterungsmethoden

Mit den konkreten Schnittstellenmethoden besteht neben den Erweiterungsmethoden (vgl. Abschnitt 7.15) eine zweite Möglichkeit, die eine Schnittstelle implementierenden Typen mit zusätzlicher Funktionalität auszustatten. Einige Merkmale sind beiden Techniken gemeinsam:

- Ein profitierender Typ muss keine eigene Leistung erbringen.
- Ein betroffener Typ kann zwischen der unveränderten Übernahme und einer eigenen Realisation entscheiden.

Bei den Erweiterungsmethoden bestehen u. a. die folgenden Besonderheiten:

- Zur Nutzung einer unverändert übernommenen Erweiterungsmethode kann und muss eine begünstigte Instanz über eine Referenz vom eigenen Typ aufgerufen werden.
- Sind mehrere implementierte Schnittstellen durch signaturgleiche Methoden erweitert, dann wählt der Compiler die *erste* angetroffene Erweiterungsmethode.
- Im Fall einer selbst realisierten Erweiterungsmethode liegt eine *Verdeckung* vor, sodass keine Polymorphie möglich ist. Bei einer Ansprache per Schnittstellenreferenz führen alle Instanzen die Methode aus der Erweiterungsklasse aus.

Bei den konkreten, ausführbaren Schnittstellen-Mitgliedern bestehen u. a. die folgenden Besonderheiten:

- Es können nicht nur Methoden mit einer voreingestellten Implementation ausgestattet werden, sondern auch Eigenschaften, Indexer und Ereignisse. Aus Gründen der sprachlichen Bequemlichkeit ist anschließend nur von *konkreten Methoden* die Rede.
- Eine konkrete Methode kann andere Schnittstellenmethoden aufrufen (abstrakte und konkrete, instanzbezogene und statische, öffentliche und private).
- Eine unverändert übernommene konkrete Methode kann nur über eine Schnittstellenreferenz angesprochen werden. Das schafft Transparenz im Fall von mehreren implementierten Schnittstellen mit signaturgleichen konkreten Methoden.
- Im Fall einer selbst realisierten konkreten Methode liegt eine *Überschreibung* vor, sodass Polymorphie möglich ist. Bei einer Ansprache per Schnittstellenreferenz führen alle Instanzen die Methode aus der eigenen Typdefinition aus.

Obwohl die Listen mit den relevanten Merkmalen der beiden Techniken sicher nicht vollständig sind, ergibt sich doch eine klare Empfehlung zur Bevorzugung der konkreten Schnittstellenmethoden gegenüber den Erweiterungsmethoden für Schnittstellen. Das folgende Beispiel demonstriert, dass im Fall der Selbstrealisation durch Klassen bei konkreten Methoden ein Überschreiben (also Polymorphie) resultiert, während bei Erweiterungsmethoden ein Verdecken stattfindet:

```
interface IWinterface {
    void SagA();

    void SagB() {
        Console.WriteLine("B");
    }
}

static class Extensions {
    public static void SagC(this IWinterface obj) {
        Console.WriteLine("C");
    }
}
```

```
class Cimple : IWinterface {
    public void SagA() {
        Console.WriteLine("A von Cimple");
    }

    public void SagB() {
        Console.WriteLine("B von Cimple");
    }

    public void SagC() {
        Console.WriteLine("C von Cimple");
    }

    static void Main() {
        var c1 = new Cimple();
        (c1 as IWinterface).SagB(); // Ausgabe: B von Cimple
        c1.SagC();                  // Ausgabe: C von Cimple
        (c1 as IWinterface).SagC(); // Ausgabe: C
    }
}
```

Ein Visual Studio - Projektordner zum Vergleich von konkreten Schnittstellenmethoden und Erweiterungsmethoden für Schnittstellen ist hier zu finden:

...\BspUeb\Interfaces\Konkrete Implementation vs. Erweiterung

9.1.2 Statische Mitglieder

9.1.2.1 Felder und ausführbare Mitglieder mit Implementation

Seit C# 8 sind in Schnittstellen statische Felder sowie konkrete statische Methoden, Eigenschaften, Indexer und Ereignisse (ausführbare Mitglieder mit Implementation) erlaubt. Diese Bestandteile sind von Nutzen bei der Realisation der ebenfalls in C# 8 eingeführten konkreten ausführbaren Instanzmitglieder. Nachdem die letzten Abschnitte stark mit Details und Regeln befrachtet waren, soll in diesem Abschnitt ein anwendungsnahes Beispiel zur Vermittlung der wichtigsten Informationen dienen.

Die folgende Schnittstelle erwartet von implementierenden Typen öffentliche Eigenschaften mit Lesezugriff für den Namen und den Geburtstag von Personen:

```
interface IPerson {
    string Name { get; }
    DateTime BirthDay { get; }
    const int allowedBirthDayDelay = 10;
    private static int maxBirthDayDelay = allowedBirthDayDelay;

    static int MaxBirthDayDelay {
        get { return maxBirthDayDelay; }
        set {
            if (value <= allowedBirthDayDelay  &&  value >= 0)
                maxBirthDayDelay = value;
        }
    }
```

```
    void SendBirthDayGreetings() {
        var birthDayThisYear =
                new DateTime(DateTime.Today.Year, BirthDay.Month, BirthDay.Day);
        var birthDayLastYear =
                new DateTime(DateTime.Today.Year-1, BirthDay.Month, BirthDay.Day);
        double elapsedDays;
        if (DateTime.Today < birthDayThisYear)
            elapsedDays = (DateTime.Today - birthDayLastYear).TotalDays;
        else
            elapsedDays = (DateTime.Today - birthDayThisYear).TotalDays;
        if (elapsedDays >= 0 && elapsedDays <= maxBirthDayDelay)
            Console.WriteLine($"Liebe*r {Name}, herzlichen Glückwunsch zum Geburtstag!");
    }
}
```

Die Schnittstelle bietet eine konkrete Instanzmethode namens `SendBirthDayGreetings()`, die gratuliert, wenn der Geburtstag erst wenige Tage zurückliegt. Per Voreinstellung wird gratuliert, wenn der Geburtstag höchstens 10 Tage zurückliegt:

```
const int allowedBirthDayDelay = 10;
private static int maxBirthDayDelay = allowedBirthDayDelay;
```

Die Konstante `allowedBirthDayDelay` (ein Feld mit dem Modifikator **const**) ist implizit öffentlich und statisch (vgl. Abschnitt 5.2.5). Sie dient dazu, ein statisches Feld mit der Schutzstufe **private** zu initialisieren.

Die erlaubte Distanz zum Geburtstag kann von implementierenden Typen über eine konkrete statische Eigenschaft ermittelt und (in Grenzen) verändert werden:

```
static int MaxBirthDayDelay {
  get { return maxBirthDayDelay; }
  set {
    if (value <= allowedBirthDayDelay  &&  value >= 0)
       maxBirthDayDelay = value;
  }
}
```

In der folgenden implementierenden Klasse `Person` wird die statische und konkrete Schnittstellen-Eigenschaft `MaxBirthDayDelay` genutzt, um die maximale Geburtstagsverzögerung festzulegen. Anschließend wird die konkrete Schnittstellen-Instanzmethode `SendBirthDayGreetings()` aufgerufen:

```
class Person : IPerson {
  public string Name { get; private set;}
  public DateTime BirthDay { get; private set; }

  static void Main() {
    var p = new Person() {Name = "Franz", BirthDay = new DateTime(1988, 12, 22)};
    IPerson.MaxBirthDayDelay = 8;
    (p as IPerson).SendBirthDayGreetings();
  }
}
```

Die maximale Verzögerung könnte auch per Methodenparameter festgelegt werden, müsste dann aber bei jedem Gruß wiederholt werden.

Ein Visual Studio - Projektordner mit dem Geburtstagsgruß-Beispiel ist hier zu finden:

...\BspUeb\Interfaces\Statische Mitglieder\Felder und konkrete ausführbare Mitglieder

Eine konkrete und statische Schnittstellenmethode mit dem voreingestellten Zugriffsschutz **public** kann übrigens von einer Klasse auch ohne Implementierung der Schnittstelle genutzt werden, z. B.:

```
interface IWinterface {
    static void SagWas() {
        Console.WriteLine("Winter");
    }
}

class Cimple {
    static void Main() {
        IWinterface.SagWas(); // Ausgabe: Winter
    }
}
```

Eine konkrete statische Methode mit dem Zugriffsmodifikator **protected** kann von einer erweiternden Schnittstelle und von einem implementierenden Typ genutzt werden.

9.1.2.2 Abstrakte und virtuelle ausführbare Mitglieder

Seit C# 11 sind in Schnittstellen auch abstrakte und virtuelle statische Mitglieder erlaubt:

- Bei den abstrakten statischen Mitgliedern fehlt eine Implementation (wie bei den abstrakten Instanzmethoden).
- Bei den konkreten statischen Mitgliedern ist das Schlüsselwort **virtual** erlaubt, sodass ein implementierender Typ die voreingestellte Implementation überschreiben kann.

Der Unterschied zwischen einer abstrakten und einer virtuellen statischen Interface-Methode besteht in der Ab- bzw. Anwesenheit einer voreingestellten Implementation.

9.1.2.2.1 Generische Mathematik

Dank der abstrakten bzw. virtuellen statischen Mitglieder können überladene Operatoren, die in C# durch statische Methoden zu realisieren sind, in Schnittstellen deklariert bzw. definiert werden. Damit lässt sich das Überladen von Operatoren per Interface vorschreiben, was in Anbetracht der zahlreichen numerischen Datentypen *generisch* geschehen muss. Die BCL enthält seit C# 11 bzw. .NET 7 zahlreiche, hierarchisch aufgebaute generische Schnittstellen zur Unterstützung von mathematischen Algorithmen, die von allen numerischen BCL-Typen (z. B. **Int64**, **Double**, **Decimal**) implementiert werden.[1] Damit kann man nun zur Beschränkung von Typformalparametern in generischen Typen oder Methoden vorschreiben, dass ein konkretisierender Typ von „numerischer Natur“ sein muss.

Besonders nützlich ist in dieser Hinsicht die Schnittstelle **INumber<TSelf>** im Namensraum **System.Numerics**, die eine reelle Zahl modelliert, den verwendeten C# - Datentyp aber offenlässt. **INumber<TSelf>** erweitert u. a. die Schnittstelle **IAdditionOperators<TSelf, TOther, TResult>**, die einen überladenen + - Operator vorschreibt. Weil seit .NET 7 alle eingebauten numerischen Typen die Schnittstelle **INumber<TSelf>** implementieren, stehen der überschriebene + - Operator, viele weitere Operatorüberladungen und sonstige Methoden generell zur Verfügung.

Operatorüberladungen für die eingebauten numerischen Typen sind alles andere als neu. Dass nun Algorithmen generisch formuliert und dann z. B. wahlweise mit den Typen **double** oder **decimal** ausgeführt werden können, eröffnet aber neue Möglichkeiten. Insbesondere können oft umfangreiche Überladungsfamilien durch *eine* generische Lösung ersetzt werden, was die Wartung des Quellcodes erheblich erleichtert.

Im Abschnitt 4.3.5.2 wurden die Ergebnisse eines Programms vorgestellt, das bei der Berechnung der Differenz

---

[1] https://learn.microsoft.com/en-us/dotnet/standard/generics/math

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

erhebliche Genauigkeits- und Laufzeitunterschiede zwischen den Datentypen **double** und **decimal** festgestellt hat. Wegen der damaligen frühen Lernphase wurde auf die Darstellung des Quellcodes verzichtet:[1]

```
using System.Diagnostics;

int anz = 1_000_000_000;
double exDouble = 1_300_000_000;
decimal exDecimal = 1_300_000_000m;
Stopwatch stopwatch = new();

stopwatch.Start();
double d = 0.0;
for (int i = 0; i < anz; i++)
    d += 1.3;
stopwatch.Stop();
Console.WriteLine($"double:\n  Abweichung:\t {(d - exDouble)}");
Console.WriteLine($"  Benöt. Zeit:\t {stopwatch.ElapsedMilliseconds} Millisek.");

stopwatch.Restart();
decimal dec = 0.0m;
for (int i = 0; i < anz; i++)
    dec += 1.3m;
stopwatch.Stop();
Console.WriteLine($"decimal:\n  Abweichung:\t {(dec - exDecimal)}");
Console.WriteLine($"  Benöt. Zeit:\t {stopwatch.ElapsedMilliseconds} Millisek.");
```

Dass wesentliche Teile des Quellcodes doppelt vorhanden sind, verstößt gegen das DRY-Prinzip (*Don't Repeat Yourself*).

Mit Hilfe der in C# 11 bzw. .NET 7 ergänzten generischen Schnittstellen zur Unterstützung mathematischer Algorithmen wird das Programm übersichtlicher. Vor allem ist der Algorithmus nur noch *einmal* vorhanden:

```
using System.Diagnostics;
using System.Numerics;

class Prog {
  public static void Teste<T>() where T : INumber<T> {
    T exakt = T.CreateChecked(1_300_000_000);
    T summand = T.CreateChecked(1.3);
    int anz = 1_000_000_000;
    Stopwatch stopwatch = new();

    stopwatch.Start();
    T summe = T.Zero;
    for (int i = 0; i < anz; i++)
      summe += summand;
    stopwatch.Stop();
    Console.WriteLine($"{typeof(T)}:\n  Abweichung:\t {(summe - exakt)}");
    Console.WriteLine($"  Benöt. Zeit:\t {stopwatch.ElapsedMilliseconds} Millisek.");
  }
```

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
…\BspUeb\Elementare Sprachelemente\DoubleDecimal

```
    static void Main() {
      Teste<double>();
      Teste<decimal>();
    }
  }
```

Die Ergebnisse der beiden Programme sind „kompatibel“:[1]

„Überladungslösung“	Generische Lösung

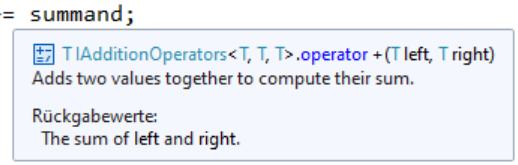
`double:`
`  Abweichung:  -24,516295194625854`
`  Benöt. Zeit:  4107 Millisek.`
`decimal:`
`  Abweichung:  0,0`
`  Benöt. Zeit:  23524 Millisek.`
	`System.Double:`
	`  Abweichung:  -24,516295194625854`
	`  Benöt. Zeit:  4811 Millisek.`
	`System.Decimal:`
	`  Abweichung:  0,0`
	`  Benöt. Zeit:  14833 Millisek.`

Die generische Methode `Teste<T>` …

- verwendet die Schnittstelle **INumber<T>** dazu, um den Typformalparameter **T** auf numerische Typen zu restringieren:
  ```
  public static void Teste<T>() where T : INumber<T> {
   . . .
  }
  ```
- benutzt die virtuelle Methode **CreateChecked<TOther>()** aus der von **INumber<TSelf>** erweiterten Schnittstelle **INumberBase<TSelf>** dazu, um Instanzen der **T**-Konkretisierung zu erstellen, wobei im Fall eines Überlaufs (im Beispiel nur möglich beim Typ **decimal**) ein Ausnahmefehler resultiert:
  ```
  static virtual TSelf CreateChecked<TOther>(TOther value)
   . . .
  }
  ```
- erstellt die zur **T**-Konkretisierung passende Null über die in **INumberBase<TSelf>** abstrakt deklarierte
  ```
  static abstract TSelf Zero { get; }
  ```
  und in **Double** überschriebene read-only – Eigenschaft **Zero**:
  ```
  internal const double Zero = 0.0;
  static double INumberBase<double>.Zero => Zero;
  ```
  Im Unterschied zu abstrakten Instanzmethoden ist der Modifikator **abstract** bei statischen Mitgliedern ohne Implementation nicht nur erlaubt, sondern sogar vorgeschrieben.

Der in `Teste<T>` verwendete Aktualisierungsoperator ist in die `+ -` Operatorüberladung einbezogen:

```
summe += summand;
```

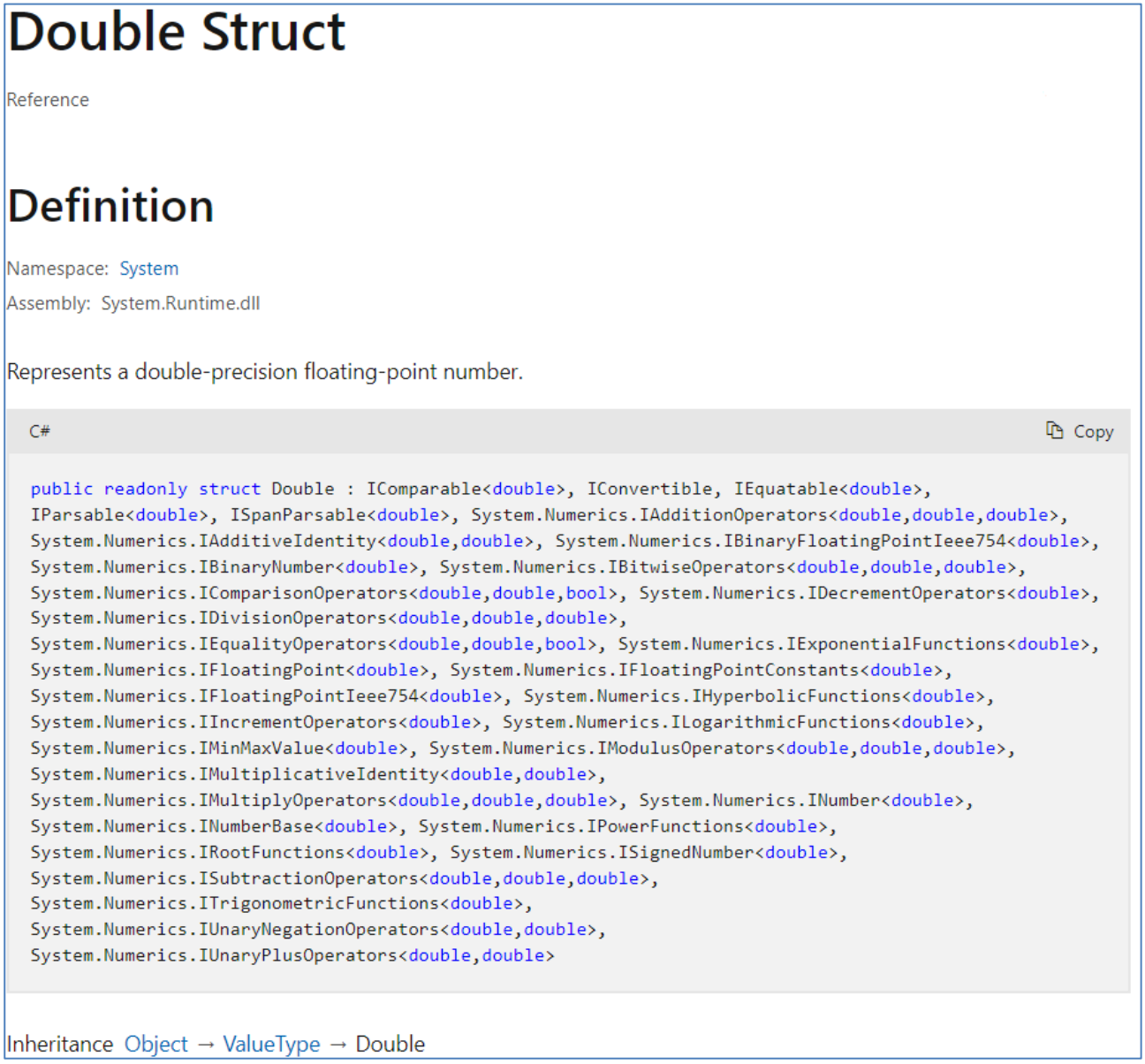


[1] Vermutlich bestehen systematische Unterschiede, wobei die Variabilität bei wiederholten Programmeinsätzen mit *derselben* Technik einen Vergleich erschwert. Das NuGet-Paket **BenchmarkDotNet** liefert in solchen Fällen eine aussagenkräftige Beurteilung durch zahlreiche Wiederholungen und eine statistische Auswertung. Wir verzichten auf eine Untersuchung, um nicht vom anspruchsvollen Thema des aktuellen Abschnitts abgelenkt zu werden.

Ein Visual Studio - Projekt mit dem modernisierten **double**/**decimal** – Vergleich ist hier zu finden:

…\BspUeb\Interfaces\Statische Mitglieder\Generische Mathematik

Wir haben eben die Schnittstelle **INumber<TSelf>** erfolgreich eingesetzt und davon profitiert, dass diese Schnittstelle von den BCL-Strukturen **Double** und **Decimal** implementiert wird. Welche beeindruckende Arbeit in .NET 7 im Zusammenhang mit den Schnittstellen zur generischen Unterstützung mathematischer Algorithmen geleistet wurde, zeigt z. B. ein Blick auf die Implementierungsliste der Struktur **Double**:[1]



Double Struct

Reference

Definition

Namespace: System

Assembly: System.Runtime.dll

Represents a double-precision floating-point number.

C# Copy

```
public readonly struct Double : IComparable<double>, IConvertible, IEquatable<double>,
IParsable<double>, ISpanParsable<double>, System.Numerics.IAdditionOperators<double,double,double>,
System.Numerics.IAdditiveIdentity<double,double>, System.Numerics.IBinaryFloatingPointIeee754<double>,
System.Numerics.IBinaryNumber<double>, System.Numerics.IBitwiseOperators<double,double,double>,
System.Numerics.IComparisonOperators<double,double,bool>, System.Numerics.IDecrementOperators<double>,
System.Numerics.IDivisionOperators<double,double,double>,
System.Numerics.IEqualityOperators<double,double,bool>, System.Numerics.IExponentialFunctions<double>,
System.Numerics.IFloatingPoint<double>, System.Numerics.IFloatingPointConstants<double>,
System.Numerics.IFloatingPointIeee754<double>, System.Numerics.IHyperbolicFunctions<double>,
System.Numerics.IIncrementOperators<double>, System.Numerics.ILogarithmicFunctions<double>,
System.Numerics.IMinMaxValue<double>, System.Numerics.IModulusOperators<double,double,double>,
System.Numerics.IMultiplicativeIdentity<double,double>,
System.Numerics.IMultiplyOperators<double,double,double>, System.Numerics.INumber<double>,
System.Numerics.INumberBase<double>, System.Numerics.IPowerFunctions<double>,
System.Numerics.IRootFunctions<double>, System.Numerics.ISignedNumber<double>,
System.Numerics.ISubtractionOperators<double,double,double>,
System.Numerics.ITrigonometricFunctions<double>,
System.Numerics.IUnaryNegationOperators<double,double>,
System.Numerics.IUnaryPlusOperators<double,double>
```

Inheritance Object → ValueType → Double

Damit eigene Typen ebenfalls an der generischen Mathematik teilnehmen können, müssen auch dort die zugehörigen Schnittstellen implementiert werden.[2] Ein geeigneter Kandidat ist die Klasse `Bruch`, die uns in den ersten Kapiteln oft als Beispiel gedient hat. Anschließend erweitern wir diese Klasse um eine Implementation der Schnittstelle **IAdditionOperators**<`Bruch`, `Bruch`, `Bruch`>:

```
public class Bruch : IAdditionOperators<Bruch, Bruch, Bruch> {
  . . .
}
```

[1] https://learn.microsoft.com/en-us/dotnet/api/system.double

[2] Siehe Überblick in: https://learn.microsoft.com/en-us/dotnet/standard/generics/math

Die + - Operatorüberladung ist als *abstrakte* statische Methode deklariert:

```
public interface IAdditionOperators<TSelf, TOther, TResult>
  where TSelf : IAdditionOperators<TSelf, TOther, TResult>? {
  static abstract TResult operator +(TSelf left, TOther right);

  static virtual TResult operator checked +(TSelf left, TOther right)
     => left + right;
}
```

Erfreulicherweise haben wir den + - Operator in der Klasse `Bruch` bereits in akzeptabler Form überladen (siehe Abschnitt 5.8.3):

```
public static Bruch operator +(Bruch b1, Bruch b2) {
    Bruch tmp = b1.Klone();
    tmp.Addiere(b2);
    tmp.Etikett = $"{b1.Etikett} + {b2.Etikett}";
    return tmp;
}
```

In C# 11 wurden auch **checked**-Varianten zu manchen Operatoren eingeführt, die eine Behandlung des Überlaufproblems erlauben. In der Schnittstelle **IAdditionOperators<TSelf, TOther, TResult>** ist die **checked** + - Operatorüberladung als virtuelle statische Methode definiert, wobei sich die vorgegebene Implementation aber *nicht* um das Überlaufproblem kümmert (siehe oben). Wir überschreiben diese Methode in der Klasse `Bruch` und führen die kritischen Berechnungen in einem Kontext mit Überlaufdiagnose aus (vgl. Abschnitt 4.6.1):

```
public static Bruch operator checked +(Bruch b1, Bruch b2) {
    var temp = new Bruch(checked(b1.zaehler * b2.nenner + b1.nenner * b2.zaehler),
                         checked(b1.nenner * b2.nenner),
                         b1.Etikett + " + " + b2.Etikett);
    temp.Kuerze();
    return temp;
}
```

Im folgenden Beispiel

```
var b1 = new Bruch(2147483647, 1, "");
var b2 = new Bruch(10, 1, "");
var b = b1 + b2;
```

liefert der + - Operator ein sinnloses Ergebnis:

```
-2147483639/1
```

Die **checked**-Variante verhindert das Fehlverhalten des Programms und fordert per Ausnahmefehlermeldung zur Behandlung des Problems auf:

```
Unhandled exception. System.OverflowException: Arithmetic operation resulted in an
overflow.
   at Bruch.op_CheckedAddition(Bruch b1, Bruch b2)
```

Ob man das Ersetzen der als **static** und **virtual** dekorierten Schnittstellenmethode

```
static virtual TResult operator checked +(TSelf left, TOther right)
   => left + right;
```

in einem implementierenden Typ als *Überschreibung* bezeichnen sollte, ist eine terminologische Frage. Wir haben den Begriff der *Überschreibung* im Zusammenhang mit der Ableitung von Klassen kennengelernt (siehe Abschnitte 7.9 und 7.11). Dabei wurde …

- die *späte Bindung* als wesentliches Bestimmungsstück genannt,
- und folglich bei statischen Methoden explizit *nicht* vom *Überschreiben* gesprochen.

Wir beschäftigen uns aktuell aber nicht mit dem Ableiten von Klassen, sondern mit dem Implementieren von Schnittstellen, und beim Ersetzen der Implementation einer virtuellen und statischen

Schnittstellenmethode in einem implementierenden Typ wird in der Regel vom *Überschreiben* gesprochen, obwohl eine *frühe Bindung* stattfindet.[1]

Ein Visual Studio - Projekt mit der aktuellen Entwicklungsstufe der Klasse `Bruch` ist hier zu finden:

…\BspUeb\Klassen und Objekte\Bruch\b10 IAdditionOperators

9.1.2.2.2 Kopfschmerzen durch eine unvollkommene Lösung

Viele für die generische Mathematik essenzielle Schnittstellen im Namensraum **System.Numerics** weisen eine Besonderheit auf, die unter der Bezeichnung *Curiously Recurring Template Pattern* (*CRTP*) bekannt und berüchtigt ist. Es stammt aus der Programmiersprache C++ und spielt bei der dortigen Template-Technik eine wichtige Rolle. In den betroffenen Definitionen generischer Schnittstellen wird für einen Typformalparameter verlangt, dass er just die definierte generische Schnittstelle implementiert. Hier ist die Situation bei der Schnittstelle **INumber<TSelf>** zu sehen, wobei das Fragezeichen im aktuellen Kontext keine Rolle spielt:

```
public interface INumber<TSelf>
    : IComparable,
      IComparable<TSelf>,
      IComparisonOperators<TSelf, TSelf, bool>,
      IModulusOperators<TSelf, TSelf, TSelf>,
      INumberBase<TSelf>
    where TSelf : INumber<TSelf>? {
    . . .
}
```

Die Typformalparameterrestriktion unter Verwendung des zu definierenden Typs stört viele Leute, und Eric Lippert, einer der profiliertesten Kenner von C# und .NET, äußert sich kritisch zum CRTP:[2]

> But this really hurts my brain.

Wir untersuchen das CRTP-Phänomen anhand eines einfacheren Beispiels. Es stammt von Anu Viswan und behandelt die ebenfalls praxisrelevente Aufgabenstellung, per Schnittstelle von implementierenden Typen die Existenz einer Fabrikmethode zu fordern, die Instanzen vom eigenen Typ liefert.[3] In der folgenden Definition der generischen Schnittstelle `IFab<T>` soll mit einer CRTP-Restriktion für den Typformalparameter `T` die Existenz einer Eigentyp-Fabrikmethode sichergestellt werden:

```
interface IFab<T> where T : IFab<T> {
    static abstract T CreateInstance();
}
```

Ein implementierender Typ versichert, eine statische Methode namens `CreateInstance()` zu besitzen, die eine Instanz von einem Typ liefert, der eine statische Methode namens `CreateInstance()` besitzt, die eine … usw. Viele menschliche Leser empfinden diese Rekursion

---

[1] Dieser Sprachgebrauch findet sich z. B. auf einer Webseite der Firma JetBrains, die u.a. Entwicklungswerkzeuge für C# herstellt:
https://blog.jetbrains.com/dotnet/2023/03/14/static-interface-members-generic-attributes-auto-default-structs-using-csharp-11-in-rider-and-resharper/

[2] https://ericlippert.com/2011/02/02/curiouser-and-curiouser/

[3] https://www.c-sharpcorner.com/article/static-abstract-interface-members-in-c-sharp-11-and-curiously-recurring-template-patt/

ohne Terminierungsbedingung als verdächtig. Was ein Compiler mit diesem Quellcode anfängt, können nur die Designer des Übersetzers beurteilen.[1]

Ein Typ mit korrekter Fabrikmethode, die eine Instanz vom eigenen Typ liefert, zeigt übrigens die endlose Rekursion, ohne Verdacht zu erregen: Die von der Fabrikmethode abgelieferte Instanz beherrscht eine Fabrikmethode, die eine … usw.

Die folgende Klasse `Apple` implementiert die Schnittstelle `IFab<Apple>`, indem sie den Typformalparameter mit dem eigenen Typ konkretisiert:

```
class Apple : IFab<Apple> {
    public static Apple CreateInstance() => new Apple();
    public void SayHello() => Console.WriteLine("Hello, I'm an apple.");
}
```

Genau das bezweckt die CRTP-Restriktion für den Typformalparameter. Ein Produkt der Fabrikmethode führt erwartungsgemäß die `Apple`-Instanzmethode `SayHello()` aus:

```
Apple.CreateInstance().SayHello(); // Ausgabe: Hello, I'm an apple.
```

Allerdings erfüllt auch die folgende Klasse `Pear` (dt.: *Birne*) die Schnittstelle `IFab<T>`, ohne eine korrekte Fabrikmethode zu besitzen:

```
class Pear : IFab<Apple> {
    public static Apple CreateInstance() => new Apple();
    public void SayHello() => Console.WriteLine("Hello, I'm a pear.");
}
```

Statt der erwarteten Birnen werden Äpfel hergestellt:

```
Pear.CreateInstance().SayHello(); // Ausgabe: Hello, I'm an apple.
```

Es ist zu hoffen, dass man in C# die Forderung an einen Typ nach der Existenz einer korrekten Fabrikmethode bald so formulieren kann, dass …

- die Intention aus dem Quellcode ohne Grübeln zu erkennen ist,
- eine lediglich *scheinbare* Erfüllung ausgeschlossen ist.

Man braucht eine Self-Type – Technik, die sich nicht auf die CRTP-Restriktion und den Namen für den Typformalparameter (z. B. **INumber<TSelf>**) beschränkt.

9.1.3 Vererbung bzw. Erweiterung von Schnittstellen

Ein Interface kann andere Interfaces beerben bzw. erweitern, wobei dieselbe Syntax wie beim Ableiten von Klassen zu verwenden ist. Nachdem wir uns im Abschnitt 9.1.2 mit *statischen* Schnittstellenmethoden beschäftigt haben, geht es im aktuellen Abschnitt ausschließlich um Instanzmethoden. Weil …

- abstrakte und konkrete Schnittstellenmethoden existieren,
- es beim Erweitern von Schnittstellen zum Verdecken oder Überschreiben von Methoden kommen kann,
- die Schnittstellen von Klassen bzw. Strukturen implementiert werden,

ist auch bei Beschränkung auf Instanzmethoden noch viel Variabilität im Spiel.

---

[1] Angeblich haben auch einige C++ - Insider anfangs darüber gestaunt, dass der Compiler die CRTP-Konstruktion übersetzt, siehe:
https://www.sandordargo.com/blog/2019/03/13/the-curiously-recurring-templatep-pattern-CRTP

9.1.3.1 ICollection<T> : IEnumerable<T> : IEnumerable

In der BCL wird 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);
}
```

Manchmal 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 verdeckt 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 7.6.1).

Bei der Implementation einer erweiternden Schnittstelle durch eine Klasse oder Struktur sind auch die Handlungskompetenzen der Basisschnittstellen zu realisieren. Im Fall der Schnittstelle **IEnumerable<T>** kommt es dabei zu einer Namenskollision, weil zwei parameterfreie Methoden namens **GetEnumerator()** zu implementieren sind. Das Problem wird durch die sogenannte *explizite Schnittstellenimplementierung* gelöst (siehe Abschnitt 9.4 und Beispiel im Abschnitt 9.5.1).

Die Schnittstellenhierarchie ist unabhängig von der Klassenhierarchie, und ein Interface kann von beliebigen Klassen bzw. Strukturen implementiert werden.

9.1.3.2 Verdecken und Überschreiben von geerbten Schnittstellenmethoden

Seit C# 8 sind die neu hinzugekommenen *konkreten* (implementierten) Methoden, Eigenschaften, Indexer und Ereignisse bei der Erweiterung von Schnittstellen zu berücksichtigen. In einer abgeleiteten (erweiternden) Schnittstelle `IB` kann eine von der Schnittstelle `IA` geerbte *konkrete* Methode (oder ein sonstiges ausführbares Mitglied) durch eine eigene konkrete Variante …

- **verdeckt** werden, wobei der Modifikator **new** erlaubt und empfohlen ist. Über eine Referenz …
  - vom Typ `IB` wird die verdeckende Methode angesprochen.
  - vom Typ `IA` bleibt die verdeckte Methode ansprechbar.
- **überschrieben** werden, wobei der Modifikator **override** weder erforderlich noch zulässig ist. Wie bei der expliziten Schnittstellenimplementierung (siehe Abschnitt 9.4) ist der Name der Basisschnittstelle anzugeben. Sowohl über eine Referenz vom Typ `IB` also auch über eine Referenz vom Typ `IA` wird die überschreibende Methode angesprochen.

Wird eine konkrete Schnittstellenmethode von einem implementierenden Typ unverändert genutzt, dann muss sie über eine Schnittstellenreferenz aufgerufen werden. Im folgenden Programm sind die beschriebenen Varianten zu sehen:

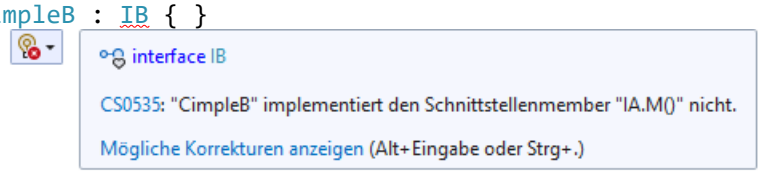
```
interface IA {
    void M1() { Console.WriteLine("IA.M1"); }
    void M2() { Console.WriteLine("IA.M2"); }
}

interface IB : IA {
    new void M1() { Console.WriteLine("IB.M1"); } // Verdecken
    void IA.M2() { Console.WriteLine("IB.M2"); }  // Überschreiben
    void M3();
}

class Cimple : IB {
    public void M3() { Console.WriteLine("Cimple.M3"); }

    static void Main() {
        Cimple cimple = new();

        ((IA)cimple).M1(); // Ausgabe: IA.M1
        ((IB)cimple).M1(); // Ausgabe: IB.M1

        Console.WriteLine();
        ((IA)cimple).M2(); // Ausgabe: IB.M2
        ((IB)cimple).M2(); // Ausgabe: IB.M2

        Console.WriteLine();
        cimple.M3();       // Ausgabe: Cimple.M3
        ((IB)cimple).M3(); // Ausgabe: Cimple.M3
    }
}
```

Ein Visual Studio - Projektordner mit dem Beispielprogramm zur Schnittstellenerweiterung ist hier zu finden:

...\BspUeb\Interfaces\Schnittstellenerweiterung

In einer abgeleiteten (erweiternden) Schnittstelle kann eine geerbte konkrete Methode durch eine abstrakte Definition ersetzt (reabstrahiert) werden, z. B.:

```
using System;

interface IA {
    void M() { Console.WriteLine("IA.M"); }
}

interface IB : IA {
    abstract void IA.M();
}

class CimpleA : IA { }
class CimpleB : IB { }
```



Die Klasse `CimpleB` muss die Instanzmethode `M()` implementieren.

9.1.3.3 Mehrfachvererbung

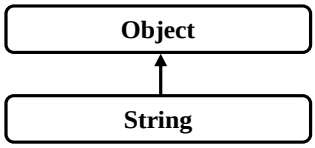
Bekanntlich wurde für C# - Klassen die risikobehaftete Mehrfachvererbung (siehe Kreft & Langer 2014 zum sogenannten *Deadly Diamond of Death*) bewusst *nicht* aus C++ übernommen. Allerdings erlauben Schnittstellen in manchen Fällen eine Ersatzlösung, denn:

- Eine Klasse darf beliebig viele Schnittstellen implementieren, sodass 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`

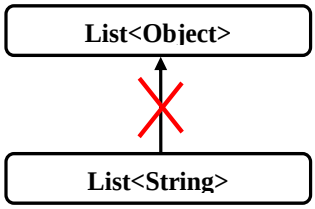
  verhalten kann. Wie wir inzwischen wissen, erbt eine Klasse beim Implementieren von Schnittstellen aber keine Methoden, sondern sie gibt Verpflichtungserklärungen ab und muss die entsprechenden Leistungen erbringen. Ein Typ erbt auch die seit C# 8 unterstützten konkreten Schnittstellenmethoden nicht. Wenn eine Klasse diese Methoden unverändert übernimmt, können sie nur über eine Referenz vom Schnittstellentyp angesprochen werden.
- Bei Schnittstellen ist die Mehrfachvererbung erlaubt. Implementiert eine Klasse oder Struktur eine Schnittstelle mit mehreren Basisschnittstellen, dann muss sie alle abstrakten Handlungskompetenzen aus den beteiligten Schnittstellen realisieren.

9.1.4 Ko- und Kontravarianz von Typformalparametern in generischen Schnittstellen

Im Abschnitt 8.2.3 wurde begründet, dass generische Kollektionstypen (z. B. **List<T>**) generell *kein kovariantes Verhalten* zeigen, dass sich also eine Spezialisierungsbeziehung zwischen zwei Aktualparametertypen wie im folgenden Beispiel



nicht auf die konkretisierten Kollektionsklassen überträgt:



Für das generell sinnvolle *invariante* Verhalten der generischen Kollektionsklassen ist aber in manchen Situationen eine Ausnahme erwünscht, was nun anhand von zwei anderen in der Spezialisierungsbeziehung stehenden Klassen erläutert werden soll. Wir betrachten 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 {
    public string Name { get; } = "unbenannt";
    public double X { get; }
    public double Y { get; }

    public Figur(string n, double x, double y) {
        Name = n;
        if (x >= 0.0 && y >= 0.0) {
            X = x;
            Y = y;
        }
    }
    public Figur() { }
}
```

Von der Klasse `Figur` stammt die Klasse `Kreis` ab:

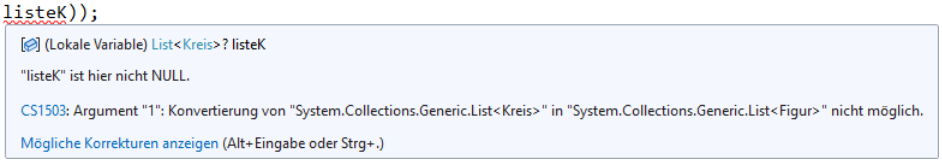
```
public class Kreis : Figur {
    public double Radius { get; }

    public Kreis(string n, double x, double y, double rad) : base(n, x, y) {
        if (rad >= 0.0)
            Radius = rad;
    }
    public Kreis() { }
}
```

In einer beliebigen Klasse wird 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)
        f (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("A", 210.0, 50.0),
        new Figur("B", 250.0, 100.0)
    };
    var listeK = new List<Kreis> {
        new Kreis("A", 250.0, 50.0, 100.0),
        new Kreis("B", 250.0, 100.0, 120.0)
    };
    Console.WriteLine(SmallestX(listeF));
    Console.WriteLine(SmallestX(listeK));
}
```



Weil sich die generischen Kollektionsklassen invariant verhalten, ist die Klasse **List**`<Kreis>` *keine* Spezialisierung der Klasse **List**`<Figur>`. Daher verhindert der Compiler z. B., dass 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 aktuell intendierten Verwendung eines **List**`<Kreis>` - Objekts als `SmallestX()` - Aktualparameter findet aber *keine Änderung* des Parameterobjekts statt. Es wäre wünschenswert, dass der

Compiler die Zuweisung erlaubt, wenn man ihm versichert, dass eine als Aktualparameter fungierende Liste nur als *Quelle* von Objekten, nicht aber als *Senke* bzw. Ablage für Objekte verwendet wird. Genau dies ermöglichen die anschließend im Abschnitt 9.1.4.1 beschriebenen *kovarianten* Typformalparameter von Schnittstellen.

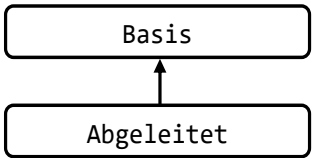
Im Abschnitt 9.1.4.2 werden *kontravariante* Typformalparameter vorgestellt, die eine weitere Flexibilisierung der Generizität in C# erlauben. Für beide Formen der Varianz (Ko- und Kontravarianz) gelten die folgenden Regeln:[1]

- Variante Typformalparameter sind nur bei generischen Schnittstellen und generischen Delegaten (siehe Abschnitt 10.1.6) erlaubt, also insbesondere *nicht* bei generischen Klassen. Um das Problem im Beispiel zu beheben, muss also eine generische Schnittstelle ins Spiel kommen.
- Eine generische Schnittstelle oder ein generischer Delegat darf sowohl kovariante als auch kontravariante Typformalparameter besitzen.
- Ko- bzw. Kontravarianz klappen nur mit Referenztypen. Das stellt aber keine Einschränkung dar, weil Werttypen nicht beerbt werden können.

9.1.4.1 Kovarianz

Das generische Interface **IEnumerable<out T>** im BCL-Namensraum **System.Collections.Generic** definiert die Voraussetzungen für eine iterierbare Kollektion (vgl. Abschnitt 9.5). Sein Typformalparameter **T** ist durch den Varianzmodifikator **out** als kovariant definiert. Damit wird festgelegt, dass **T** *nicht* als Parametertyp einer Schnittstellenmethode, sondern ausschließlich als Rückgabetyp verwendet wird.[2]

Werden zu zwei Klassen mit der Spezialisierungsbeziehung



die **IEnumerable<out T>** - Konkretisierungen **IEnumerable<Basis>** und **IEnumerable<Abgeleitet>** gebildet, dann akzeptiert der Compiler an Stelle eines Objekts vom **IEnumerable<Basis>** auch ein Objekt vom Typ **IEnumerable<Abgeleitet>**.

Für die Schnittstelle **IEnumerable<out T>** ist die Deklaration des Typformalparameters als kovariant gerechtfertigt, weil die einzige Schnittstellenmethode **GetEnumerator()** ein Objekt liefert, das die Schnittstelle **IEnumerator<T>** erfüllt (vgl. Abschnitt 9.5):

IEnumerator<out T> GetEnumerator()

Auch in der Schnittstelle **IEnumerator<out T>** ist **T** als kovariant definiert. Der Typformalparameter hat dort seinen einzigen Auftritt als **get**-Rückgabe der Eigenschaft **Current**:

T Current { get; }

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 weil außerdem **String** von **Object** abstammt:

```
IEnumerable<Object> lob = new List<String>();
```

---

[1] https://docs.microsoft.com/en-us/dotnet/standard/generics/covariance-and-contravariance

[2] Selbst Ausgabeparameter vom Typ **T** sind verboten.

Über eine Referenz vom Schnittstellentyp **IEnumerable<Object>** wird von einem Objekt der Klasse **List<String>** eine **GetEnumerator()** - Rückgabe vom Typ **IEnumerator<Object>** erwartet. Das tatsächlich von `lob` 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 den folgenden Eigenschaften:

- Das Interface **IEnumerable<out 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 Parameterdatentyp 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;
}
```

Wegen der Kovarianz des **IEnumerable<out T>** - Typformalparameters **T** kann ein Objekt vom Typ **IEnumerable**<`Kreis`> auch einer Variablen von Typ **IEnumerable**<`Figur`> zugewiesen werden, sodass der Compiler nichts gegen die Verwendung eines **List**<`Kreis`> - Objekts als Aktualparameter von `SmallestX()` einzuwenden hat:[1]

```
static void Main() {
    var listeF = new List<Figur> {
        new Figur("A", 210.0, 50.0),
        new Figur("B", 250.0, 100.0)
    };
    var listeK = new List<Kreis> {
        new Kreis("A", 250.0, 50.0, 100.0),
        new Kreis("B", 250.0, 100.0, 120.0)
    };
    Console.WriteLine(SmallestX(listeF));
    Console.WriteLine(SmallestX(listeK));
}
```

Ein Visual Studio - Projektordner mit dem Beispielprogramm zur Kovarianz ist hier zu finden:

...\BspUeb\Interfaces\Typformalparameter in generischen Interfaces\Kovarianz

9.1.4.2 Kontravarianz

Über das generische Interface **IComparable<in T>** im Namensraum **System** signalisiert ein Datentyp, dass für seine Instanzen eine Ordnung definiert ist (siehe Erläuterungen zur Klasse **String** am Anfang des aktuellen Kapitels). Das Interface verlangt von implementierenden Typen eine Methode mit dem folgenden Definitionskopf:

---

[1] Im Vorgriff auf Abschnitt 11.3.1 wird ein Kollektionsinitialisierer verwendet, was wegen der plausiblen Syntax vor dem Hintergrund der Erfahrung mit dem analogen Array-Initialisierer didaktisch akzeptabel ist.

public int CompareTo(T *other***)**

Der Typformalparameter **T** im Interface **IComparable<in T>** ist über den Varianzmodifikator **in** als *kontravariant* definiert. Damit wird dem Compiler signalisiert, dass **T** in den Schnittstellenmethoden *nicht* zur Spezifikation eines Rückgabetyps, sondern ausschließlich als Typ von Methoden*parametern* genutzt wird.

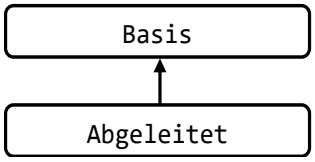
Durch die Kontravarianz des Typformalparameters von **IComparable<in T>** ist sichergestellt, dass Methoden dieser Schnittstelle ...

- lediglich **T**-Instanzen verarbeiten,
- aber keinesfalls **T**-Instanzen abliefern.

Wo ein die Schnittstelle **IComparable<in T>** implementierendes Objekt benötigt wird, kann auch ein die Schnittstelle **IComparable<in B>** implementierendes Objekt verwendet werden, wenn **B** eine Basisklasse von **T** ist, denn:

- Eine für Basisklassenobjekte **B** geeignete Methode kommt auch mit **T**-Objekten zurecht.
- Es passiert nie, dass statt eines **T**-Objekts ein (schwächer ausgestattetes) Basisklassenobjekt ausgeliefert wird.

Die Typsicherheit ist also nicht gefährdet. Infolgedessen wird bei zwei Klassen mit der Spezialisierungsbeziehung



für die beiden zugehörigen **IComparable<in T>** - Implementierungen die **Zuweisungskompatibilität umgekehrt**. Dies bedeutet, dass der Compiler an Stelle eines Objekts vom **IComparable**<Abgeleitet> auch ein Objekt des Typs **IComparable**<Basis> akzeptiert.

Zur Demonstration implementieren wir in der Klasse `Figur` im Vorgriff auf den Abschnitt 9.2 die Schnittstelle **IComparable**<Figur>:

```
using System;
public class Figur : IComparable<Figur> {
    . . .
    public int CompareTo(Figur vergl) {
        if (X < vergl.X)
            return -1;
        if (X > vergl.X)
            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<in T>** ist **IComparable**<Figur> zuweisungskompatibel zu **IComparable**<Kreis>. Wo eigentlich der Typ **IComparable**<Kreis> verlangt ist, wird also auch **IComparable**<Figur> akzeptiert.

Daher kann zum Sortieren einer Liste mit Kreisen

```
var listeK = new List<Kreis> {
    new Kreis("A", 210.0, 50.0, 100.0),
    new Kreis("B", 250.0, 100.0, 120.0),
    new Kreis("C", 130.0, 110.0, 80.0)
};
```

die Methode **Sort()** der Klasse **List<T>** verwendet werden:

```
listeK.Sort();
```

Sie erwartet, dass der Elementtyp **T** die generische Schnittstelle **IComparable<T>** oder die nicht-generische Schnittstelle **IComparable** erfüllt, wie ein Blick in die BCL-Dokumentation zeigt:

List<Kreis> implementiert zwar nicht **IComparable<Kreis>**, aber **IComparable<Figur>**, und das genügt wegen der Kontravarianz des Typformalparameters von **IComparable<in T>**.

Ein Visual Studio - Projektordner mit dem Beispielprogramm zur Kontravarianz ist hier zu finden:

...\BspUeb\Interfaces\Typformalparameter in generischen Interfaces\Kontravarianz

9.2 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.

9.2.1 Beispiel IComparable<T>

Ein Beispiel kennen Sie bereits, weil im Abschnitt 9.1.4.2 zur Erläuterung der Kontravarianz eine Variante der Klasse `Figur` vorgestellt wurde, die das Interface **IComparable<Figur>** implementiert, sodass `Figur`-Kollektionen bequem sortiert werden können:

```
public class Figur : IComparable<Figur> {
    public string Name { get; } = "unbenannt";
    public double X { get; }
    public double Y { get; }

    public Figur(string n, double x, double y) {
        Name = n;
        if (x >= 0.0 && y >= 0.0) {
            X = x;
            Y = y;
        }
    }
    public Figur() { }
```

```
    public int CompareTo(Figur vergl) {
        if (X < vergl.X)
            return -1;
        if (X > vergl.X)
            return 1;
        return 0;
    }
}
```

Eine Klasse oder Struktur muss alle abstrakt definierten Handlungskompetenzen einer in ihrem Definitionskopf reklamierten Schnittstelle implementieren. Von der generischen Schnittstelle **IComparable<in T>** wird nur eine **public**-Methode namens **CompareTo()** mit einem Parameter vom Typ **T** und einem Rückgabewert vom Typ **int** vorgeschrieben. In semantischer Hinsicht soll **CompareTo()** eine `Figur` beauftragen, sich mit dem per Aktualparameter bestimmten Artgenossen zu vergleichen. Bei der obigen Realisation werden Figuren nach der X-Koordinate ihrer linken oberen Ecke verglichen:[1]

- Liegt die angesprochene Figur links vom Vergleichspartner, dann wird die Zahl -1 zurückgemeldet.
- Haben beide Figuren in der linken oberen Ecke dieselbe X-Koordinate, dann lautet die Antwort 0.
- Ansonsten wird eine 1 gemeldet.

Weil im Beispiel der `Figur`-Vergleich auf einem **double**-Vergleich basiert, kann die `Figur`-Methode **CompareTo()** eleganter formuliert werden:

```
public int CompareTo(Figur vergl) {
   return X.CompareTo(vergl.X);
}
```

Die im Abschnitt 5.8.1 beschriebene Methodendefinition per Lambda-Symbol und Ausdruck erlaubt eine noch kompaktere Formulierung:

```
public int CompareTo(Figur vergl) => X.CompareTo(vergl.X);
```

Bei dieser für die Praxis empfohlenen Lösung ist allerdings nicht mehr erkennbar, wie sich eine **CompareTo()** - Methode verhalten muss. Weil das korrekte **CompareTo()** - Verhalten aktuell im Mittelpunkt steht, wird die ausführliche Implementation aus der `Figur` – Klassendefinition beibehalten.

Für die Implementierung einer abstrakten Schnittstellenmethode muss die Schutzstufe **public** explizit deklariert werden.[2] Anderenfalls äußert sich der Compiler z. B. 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** *nicht* angegeben werden. Das gilt auch dann, wenn ein Typ eine konkrete, öffentliche und nicht versiegelte Schnittstellenmethode überschreibt. Hier besteht ein Unterschied zum Überschreiben von abstrakten Basisklassenmethoden (siehe Abschnitt 7.9).

Eine Klasse oder Struktur kann auf das Implementieren einiger abstrakter Interface-Mitglieder verzichten und diese wie auch sich selbst als **abstract** deklarieren.

---

[1] In der `Figur`-Methode **CompareTo()** werden die im Abschnitt 4.5.4 diskutierten Genauigkeitsprobleme des binären Gleitkommatyps **double** der Einfachheit halber ignoriert.

[2] Es ist auch die im Abschnitt 9.4 beschriebene, ohne Zugriffsmodifikator auskommende explizite Schnittstellenimplementierung möglich, wobei dann aber zum Methodenaufruf eine Schnittstellenreferenz erforderlich ist.

Weil sich `Figur`-Objekte mit einem Artgenossen vergleichen können, gelingt das Sortieren einer **List<Figur>** - Kollektion mit der Methode **Sort()** mühelos, z. B.:[1]

Quellcode	Ausgabe

`var lf = new List<Figur> {`
`  new Figur("A", 250.0, 50.0),`
`  new Figur("B", 150.0, 50.0),`
`  new Figur("C", 50.0, 50.0)`
`};`
 	
`lf.Sort();`
`foreach (var v in lf)`
`  Console.Write(v.Name + " ");`
	`C B A`

Der Typformalparameter der generischen Klasse **List<T>** ist nicht restringiert:

```
public class List<T> : IList<T>, IList, IReadOnlyList<T> { ... }
```

Daher protestiert der Compiler *nicht*, wenn die Methode **Sort()** auf eine Konkretisierung von **List<T>** angewendet wird, obwohl der konkretisierende Typ weder die generische Schnittstelle **IComparable<T>** noch die Schnittstelle **IComparable** implementiert, z. B.:

```
var lnc = new List<NC> { new NC(), new NC() };
lnc.Sort();
class NC { }
```

In diesem Fall wird der Quellcode fehlerfrei übersetzt, doch beim Aufruf von **Sort()** kommt es zu einem Laufzeitfehler vom Typ **InvalidOperationException** (siehe Dokumentation der **List<T>** - Methode **Sort()**, teilweise wiedergegeben im Abschnitt 9.1.4.2).

Ob ein Typ eine Schnittstelle erfüllt, kann über den Typtest-Parameter **is** festgestellt werden, z. B.:

```
if (lf is IComparable<Figur>)
    lf.Sort();
```

9.2.2 Erben von Schnittstellenimplementationen

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 im Abschnitt 9.2.1 vorgestellten Klasse `Figur` abgeleitet, dann erfüllt auch die Klasse `Kreis` die Schnittstelle **IComparable<Figur>**, jedoch *nicht* die Schnittstelle **IComparable<Kreis>**. Dass ein die Schnittstelle **IComparable<Figur>** implementierender Typ akzeptiert wird, wenn eigentlich das Implementieren von **IComparable<Kreis>** verlangt ist, liegt an der Kontravarianz des Typformalparameters **T** in der generischen Schnittstelle **IComparable<in T>** (siehe Abschnitt 9.1.4.2). Unter diesen Voraussetzungen verarbeitet die **List<T>** - Methode **Sort()** auch eine die Schnittstelle **IComparable<Figur>** erfüllende **List<Kreis>** - Kollektion erfolgreich, z. B.:

[1] Im Vorgriff auf Abschnitt 11.3.1 wird ein Kollektionsinitialisierer verwendet, was wegen der plausiblen Syntax vor dem Hintergrund der Erfahrung mit dem analogen Array-Initialisierer didaktisch akzeptabel ist.

Quellcode	Ausgabe

`var lk = new List<Kreis> {`
`  new Kreis("A", 250.0, 50.0, 10.0),`
`  new Kreis("B", 150.0, 50.0, 20.0),`
`  new Kreis("C", 50.0, 50.0, 30.0)`
`};`
 	
`lk.Sort();`
`foreach (var v in lk)`
`  Console.Write(v.Name + " ");`
	`C B A`

9.2.3 Vorteile generischer Schnittstellen

Im Vergleich zu einem traditionellen, nicht-generischen Interface hat ein generisches Gegenstück die folgenden Vorteile:

- Man gewinnt Typsicherheit,
- erspart sich lästige Typumwandlungen
- und vermeidet bei Strukturen zeitaufwändige Boxing-Operationen.

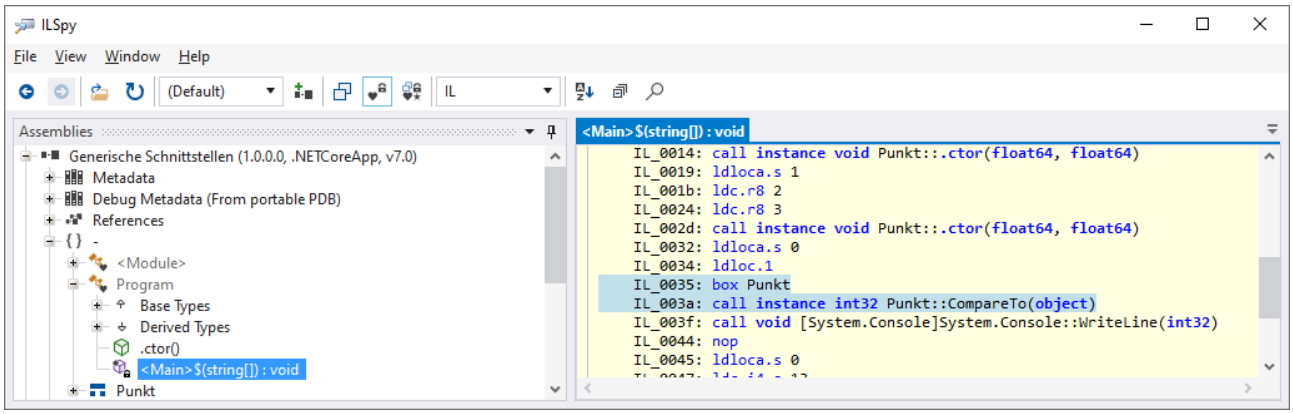
Dies soll bei einer einfachen Struktur für Punkte in der Zahlenebene und einer durch die X-Koordinate definierten Anordnung demonstriert werden. In der ersten Variante wird die traditionelle Schnittstelle **IComparable** implementiert, also eine **CompareTo()** - Methode mit **Object**-Parameter realisiert:

```
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 der Methode **CompareTo()** ist eine explizite Typumwandlung von **Object** zu `Punkt` erforderlich, die zu einer **InvalidCastException** führen kann, z. B. im folgenden Beispiel:

```
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 mit ILSpy analysierte IL-Code zeigt, erfordert außerdem jeder gelungene **CompareTo()** - Aufruf eine Boxing-Operation:



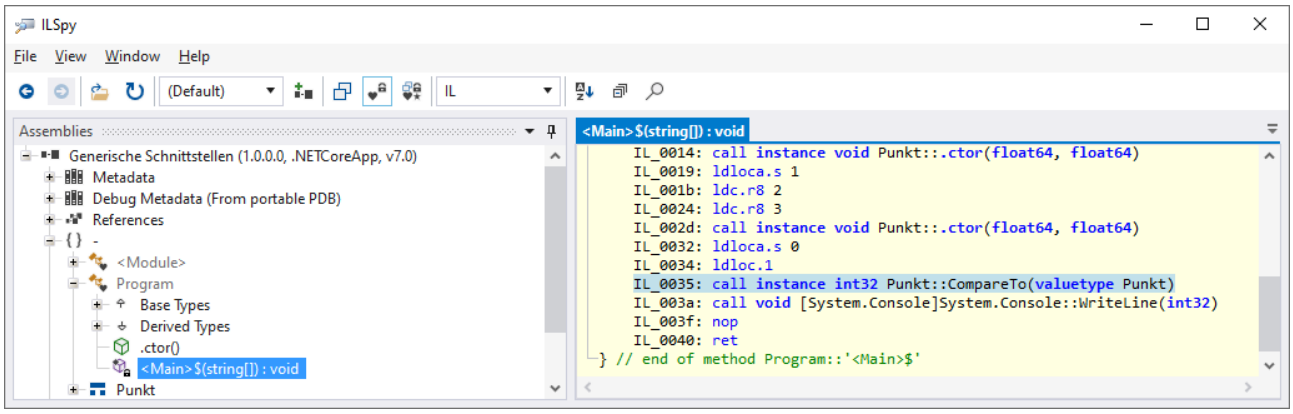
Implementiert die Struktur `Punkt` stattdessen das konkretisierte generische Interface **IComparable**<Punkt>,

```
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;
   }
}
```

dann kann die fehlerhafte Quellcodezeile nicht mehr übersetzt werden,

```
Console.WriteLine(p1.CompareTo(13));
```

und ein gelungener **CompareTo()** - Aufruf geht ohne Boxing über die Bühne:



9.3 Interfaces als Referenzdatentypen

9.3.1 Flexible Polymorphie

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

`IType[] ita = { new K1(), new K2(), new S() };`
`foreach (IType it in ita)`
`    Console.WriteLine(it.SagWas());`
 	
`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"; }`
`}`
	`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.

Ist eine öffentliche Schnittstellenmethode virtuell, also ...

- entweder abstrakt
- oder konkret und nicht versiegelt,

dann findet beim Aufruf eine späte Bildung statt (Polymorphie).

Interface-Typen sind grundsätzlich *Referenztypen*, sodass eine Variable mit einem solchen Datentyp nur eine Objektadresse aufnehmen kann. Wird einer solchen Referenzvariablen eine Strukturinstanz zugewiesen, dann ist ein Boxing fällig. In zeitkritischen Programmen muss eine hohe Zahl von Boxing-Operationen vermieden werden.

9.3.2 Tückisches Boxing bei implementierenden Strukturen

Werden Instanzen eines Strukturtyps, der ein Interface implementiert, bei der Ansprache per Interface-Referenz temporär in ein Objekt verpackt, dann kann es zu tückischen Programmierfehlern kommen. Im folgenden Beispiel wird vergeblich versucht, eine Instanzvariable einer per Interface-Referenz angesprochenen Strukturinstanz vom Typ `Employee` zu inkrementieren:[1]

```
interface IPromotion {
    void Promote();
}

struct Employee : IPromotion {
    public string Name;
    public int JobGrade;
```

[1] Herkunft des Beispiels: https://docs.microsoft.com/en-us/archive/blogs/abhinaba/c-structs-and-interface

```
        public void Promote() {
            JobGrade++;
        }

        public Employee(string name, int jobGrade) {
            Name = name;
            JobGrade = jobGrade;
        }

        public override string ToString() {
            return string.Format("{0} ({1})", Name, JobGrade);
        }
    }

    class Prog {
        static void Main() {
            Employee employee = new("Cool Guy", 65);
            IPromotion p = employee;
            Console.WriteLine(employee);
            p.Promote();
            Console.WriteLine(employee);
        }
    }
```

In der **Main()** – Methode wird durch den `Promote()` – Aufruf lediglich das per `IPromotion` - Referenz ansprechbare und per Autoboxing erstellte temporäre Objekt geändert, aber nicht die Strukturinstanz. Daher produziert das Programm die Ausgabe:

```
Cool Guy (65)
Cool Guy (65)
```

Macht man aus der Struktur `Employee` eine Klasse, dann verhält sich das Programm erwartungsgemäß:

```
Cool Guy (65)
Cool Guy (66)
```

9.4 Explizite Schnittstellenimplementierung

Virtuelle Methoden von Schnittstellen werden in der Regel durch **public**-Methoden des implementierenden Typs realisiert. Bei dieser sogenannten *impliziten* Implementation kann es zu Namenskollisionen kommen, wenn …

- ein Typ mehrere Schnittstellen implementiert,
- und zwei oder mehrere Schnittstellen eine signaturgleiche Methode besitzen, für die aber unterschiedliche Funktionsweisen vorgesehen sind.

Für diese Situation bietet C# die sogenannte *explizite* Schnittstellenimplementierung, wobei der implementierende Typ …

- die Methode mehrfach implementiert,
- bei jeder Implementation dem Methodennamen den Namen der zugehörigen Schnittstelle durch einen Punkt getrennt voranstellt,
- *keine* Zugriffsmodifikatoren angibt.

Zugriffsmodifikatoren sind verboten, weil die Methode nicht als Mitglied des implementierenden Typs verfügbar ist, sondern nur über Schnittstellen-Referenzen.

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 1;
   }

   int I2.M() {
      return 2;
   }
}
```

Objekte der Klasse `M` können *beide* Methoden ausführen, sofern sie über eine Referenz vom passenden Interface-Typ angesprochen werden, z. B.:

```
K k = new();
Console.WriteLine((k as I1).M()); // Ausgabe: 1
Console.WriteLine((k as I2).M()); // Ausgabe: 2
```

Über eine Referenzvariable vom eigenen Datentyp angesprochen, ist jedoch *keine* Methode namens `M()` verfügbar, sodass die folgende Anweisung

```
Console.WriteLine(k.M());
```

den Compiler zu einer Fehlermeldung veranlasst:

```
"K" enthält keine Definition für "M"
```

Implementiert die Klasse `K` die Methode `M()` ausschließlich implizit,

```
public int M() {
    return 3;
}
```

dann ist die Methode über Referenzen der Typen `K`, `I1` und `I2` aufrufbar.

Hat eine Klasse zwei Schnittstellen mit signaturgleichen Methoden zu implementieren, dann ist auch die Kombination aus einer impliziten und einer expliziten Implementation erlaubt, z. B.:

```
public class K1 : I1, I2 {
    public int M() => 1;
    int I2.M() => 2;
}
```

Beim Aufruf über eine `K1`-Referenz führen `K1`-Objekte die `M()` – Methode der Schnittstelle `I1` aus. Die explizit implementierte und daher nur per Schnittstellenreferenz ausführbare Methode (im Beispiel: `I2.M()`) wird auch als *versteckte* Methode bezeichnet.

Selbstverständlich kann sich eine Klasse `K2`, die ebenfalls die beiden Schnittstellen `I1` und `I2` implementiert für eine umgekehrte Verteilung der impliziten und der expliziten Implementation entscheiden, z. B.:

```
public class K2 : I1, I2 {
    int I1.M() => 1;
    public int M() => 2;
}
```

Die abweichenden Entscheidungen des `K1`- und des `K2`-Designers können trotz sorgfältiger Dokumentation für unerwartete Effekte sorgen. Ein Anwender der beiden Klassen rechnet eventuell nicht

damit, dass beim Aufruf der Methode `M()` ein `K1`-Objekt die `I1`-Variante ausführt, ein `K2`-Objekt jedoch die `I2`-Variante, z. B.:

```
public class Test {
    static void Main() {
        Console.WriteLine(new K1().M()); // Ausgabe: 1
        Console.WriteLine(new K2().M()); // Ausgabe: 2
    }
}
```

Eine explizite Schnittstellenimplementierung kommt nicht nur bei Methoden in Frage, sondern auch bei anderen ausführbaren Mitgliedern. Weitere Hinweise finden sich z. B. bei Richter (2012, Kap. 13).

9.5 Iteratoren und Enumeratoren

Durch ein Gespann aus einem Iterator und einem Enumerator entsteht eine Sequenz (z. B. bestehend aus den Elementen einer Kollektion), die nur die Vorwärtsbewegung und nur Lesezugriffe zulässt. In der Regel wird diese Sequenz im Rahmen einer **foreach**-Schleife durchlaufen.

9.5.1 IEnumerable<T> und IEnumerable

Für das im Abschnitt 9.1.4.1 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 noch zu erläutern, wie die Methode **GetEnumerator()** zu implementieren ist. Es ist 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();
}
```

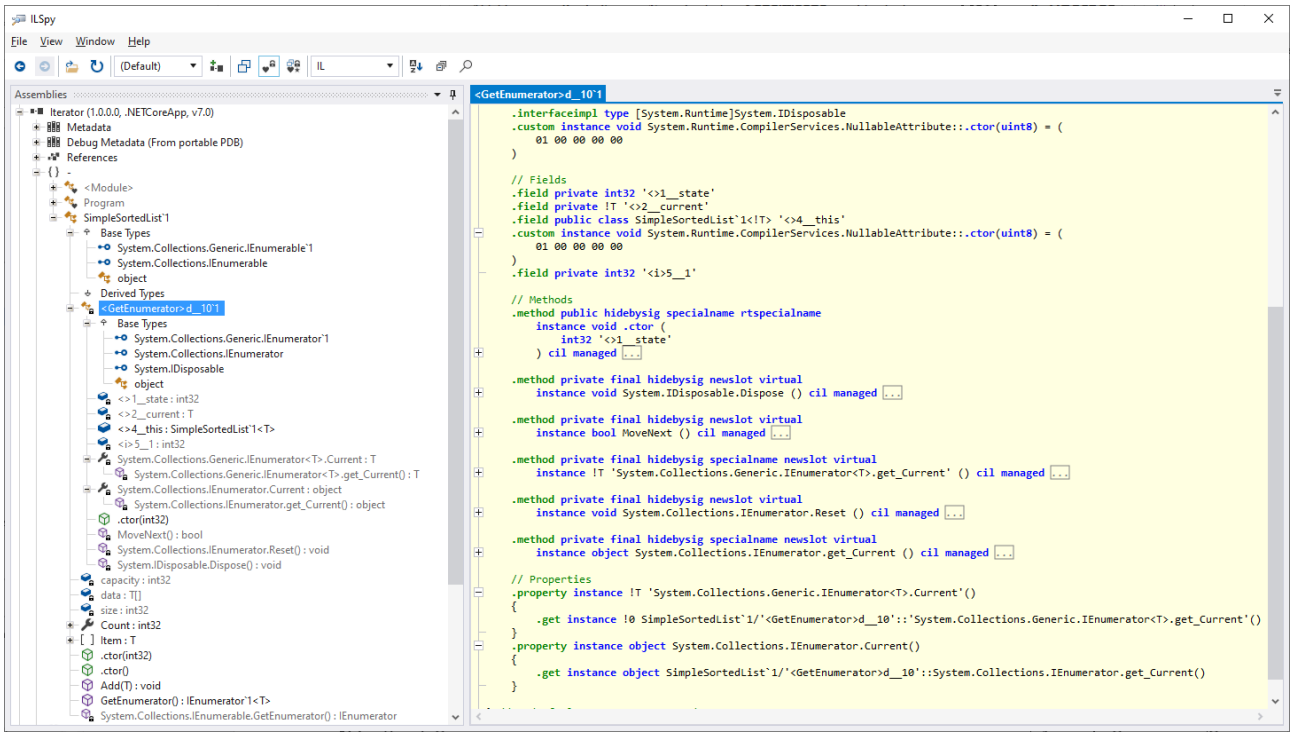
Erfreulicherweise übernimmt der Compiler fast die gesamte Arbeit.

Als Anwendungsbeispiel erweitern wir die im Abschnitt 8.2.2 erstellte Kollektionsklasse `SimpleSortedList<T>`, die ihre Elemente in einem sortieren Zustand hält, um die Möglichkeit, die Elemente per **foreach**-Schleife zu durchlaufen. Das Ziel 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];
}
```

Albahari (2022, S. 189ff) bezeichnet diese Methode als *Iterator*. Der Anweisungsblock wird nicht direkt ausgeführt, sondern vom Compiler zu einer Klasse verarbeitet, die das Interface **IEnumerator<T>** sowie dessen Basisschnittstellen **IEnumerator** und **IDisposable** erfüllt. Eine auffällige Besonderheit der **GetEnumerator()** - Methode mit Iterator-Anweisungsblock besteht daran, dass keine **return**-Anweisung mit einer Rückgabe vom Typ **IEnumerator<T>** vorhanden ist.

Wie eine Inspektion mit ILSpy zeigt, resultiert im Beispiel aus dem Iterator eine geschachtelte Klasse mit dem Namen `<GetEnumerator>d__10`:



Insbesondere sind die Methoden **MoveNext()** und **Dispose()** sowie die Eigenschaft **Current** vorhanden.

Ein Objekt dieser geschachtelten Klasse wird als *Enumerator* bezeichnet. Es basiert auf dem oben beschriebenen Iterator und versorgt die **foreach**-Schleife mit den Kollektionselementen. Während der Iterator die Erstellung des Enumerators erleichtert, vereinfacht die **foreach**-Schleife die Nutzung des Enumerators (Albahari 2022, S. 348).

Bei einem Objekt der Klasse `SimpleSortedList<T>` befinden sich die Elemente in einem privaten Array namens `data`. Der Iterator liefert in einer **for**-Schleife die Elemente Stück für Stück in einer Anweisung mit dem Namen **yield return** aus. Während der Ausführung einer **foreach**-Schleife wird nach jeder **yield return** - Ausführung die erreichte Position vom Enumerator gespeichert, und beim nächsten Aufruf des Iterators wird die Ausführung an der gespeicherten Position fortgesetzt.[1]

Weil die Klasse `SimpleSortedList<T>` von sich behaupten möchte, die Schnittstelle **IEnumerable<T>** zu implementieren,

```
public class SimpleSortedList<T> : IEnumerable<T>  where T : IComparable<T> { ... }
```

[1] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/iterators

muss sie auch die zur nicht-generischen Basisschnittstelle **IEnumerable** gehörende Methode **GetEnumerator()** implementieren. Im Beispiel genügt es, die Variante der generischen Schnittstelle aufzurufen, weil **IEnumerable<T>** von **IEnumerable** abstammt:

```
IEnumerator IEnumerable.GetEnumerator() {
   return GetEnumerator();
}
```

Die **GetEnumerator()** – Methode in der nicht-generischen Schnittstelle **IEnumerable** besitzt dieselbe Signatur wie das Gegenstück in der generischen Schnittstelle **IEnumerable<T>**, sodass bei zumindest einer Variante eine explizite Schnittstellenimplementierung (mit dem vorangestellten Schnittstellennamen, ohne Zugriffsmodifikator) erforderlich ist (siehe Abschnitt 9.4). In `SimpleSortedList<T>` wird …

- die **IEnumerable<T>** - Methode implizit implementiert, sodass sie beim Aufruf über eine `SimpleSortedList<T>` - Referenz oder über eine **IEnumerable<T>** - Referenz zum Einsatz.
- die **IEnumerable** - Methode explizit implementiert, sodass sie nur beim Aufruf über eine **IEnumerable** - Referenz zum Einsatz kommt.

So verhalten sich die meisten Kollektionen, wobei allerdings die Arrays eine Ausnahme machen und aus Kompatibilitätsgründen die **GetEnumerator()** – Methode aus der nicht-generischen Schnittstelle **IEnumerable** implizit implementieren (Albahari 2022, S. 346). Das ist aber für Anwendungsprogrammierer nur dann zu beachten, wenn ein Enumerator ausnahmsweise nicht per **foreach**-Schleife genutzt wird, sondern direkt, z. B.:

```
int[] data = { 1, 2, 3, 4, 5 };
var en = ((IEnumerable<int>)data).GetEnumerator();
while (en.MoveNext()) {
    int ic = en.Current;
    Console.Write(ic + " "); // Ausgabe: 1 2 3 4 5
}
```

In der Regel ist zum Iterieren durch eine Kollektion die **foreach**-Schleife zu bevorzugen, weil der Compiler dabei wertvolle Leistungen erbringt (siehe unten). Das ist nun auch bei einer Kollektion vom Typ `SimpleSortedList<T>` möglich, weil wir die Schnittstelle **IEnumerable<T>** korrekt implementiert und einen Iterator definiert haben, sodass der Compiler einen Enumerator erstellen konnte, z. B.:[1]

Quellcode	Ausgabe

`var si = new SimpleSortedList<int>(5) { 11, 2, 1, 4, 7 };`
`foreach (int i in si)`
`    Console.WriteLine(i);`
	1
	2
	4
	7
	11

Eine **foreach**-Anweisung wird vom Compiler ungefähr so umgesetzt:[2]

- Durch einen Aufruf der Methode **GetEnumerator()** mit dem Iterator-Anweisungsblock entsteht ein Objekt der vom Compiler erstellten geschachtelten Klasse. Dieses Objekt bezeichnen wir als *Enumerator*.

---

[1] Im Vorgriff auf Abschnitt 11.3.1 wird ein Kollektionsinitialisierer verwendet, was wegen der plausiblen Syntax vor dem Hintergrund der Erfahrung mit dem analogen Array-Initialisierer didaktisch akzeptabel ist.

[2] Siehe:
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/iterators
- ECMA 2022, S. 282ff

- Im **try**-Block einer **try-finally** - Anweisung läuft eine **while**-Schleife, die im Bedingungsteil einen Aufruf der **MoveNext()** - Methode des Enumerators enthält. Im Anweisungsteil der **while**-Schleife wird die **Current**-Eigenschaft des Enumerators verwendet, um das aktuelle Element abzurufen, das von einer **yield return** – Anweisung stammt.
- 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. Das ist z. B. relevant, wenn die Kollektionselemente aus einer Datenbank abgefragt werden durch eine Klasse, die das von **IEnumerable<T>** abgeleitete Interface **IQueryable<TSource>** implementiert (siehe Kapitel 19 in Baltes-Götz (2021). In diesem Fall muss nach dem Durchlaufen der Schleife die Datenbankverbindung geschlossen werden.

Nach Albahari (2022, S. 346) wird eine **foreach**-Schleife unter Verwendung der im Abschnitt 13.2.1.2 sowie im Abschnitt 16.2.3 von Baltes-Götz (2021) behandelten **using**-Anweisung zur automatisierten Freigabe von Ressourcen ungefähr so umgesetzt:[1]

```
using (var enumerator = si.GetEnumerator())
    while (enumerator.MoveNext()) {
        var element = enumerator.Current;
        // ...
    }
```

Aus der **using**-Anweisung erstellt der Compiler eine **try-finally** – Anweisung.

Statt in der Methode **GetEnumerator()** einen Iterator-Anweisungsblock zu formulieren, aus dem der Compiler eine geschachtelte Enumerator-Klasse erstellt, kann ein die Schnittstelle **IEnumerable<T>** implementierenden Typ den Enumerator auch selbst definieren (als Klasse oder Struktur), was z. B. die generische BCL-Kollektionsklasse **List<T>** tut.

9.5.2 Benannte Iteratoren

Als Alternative oder Ergänzung zu der eben beschriebenen Implementation der Schnittstelle **IEnumerable<out T>** lassen sich in C# benannte Iteratoren mit Methoden- oder Eigenschaftssyntax realisieren.

In der Klasse `SimpleSortedList<T>` soll als Ergänzung zum Standarditerator auch ein Iterator mit Parametern zur Bereichsspezifikation angeboten werden. Dazu eignet sich ein benannter Iterator mit Methodensyntax. Statt **GetEnumerator()** ist ein alternativer Methodenname zu verwenden, und als Rückgabetyp ist **IEnumerable<T>** vorgeschrieben, z. B.:

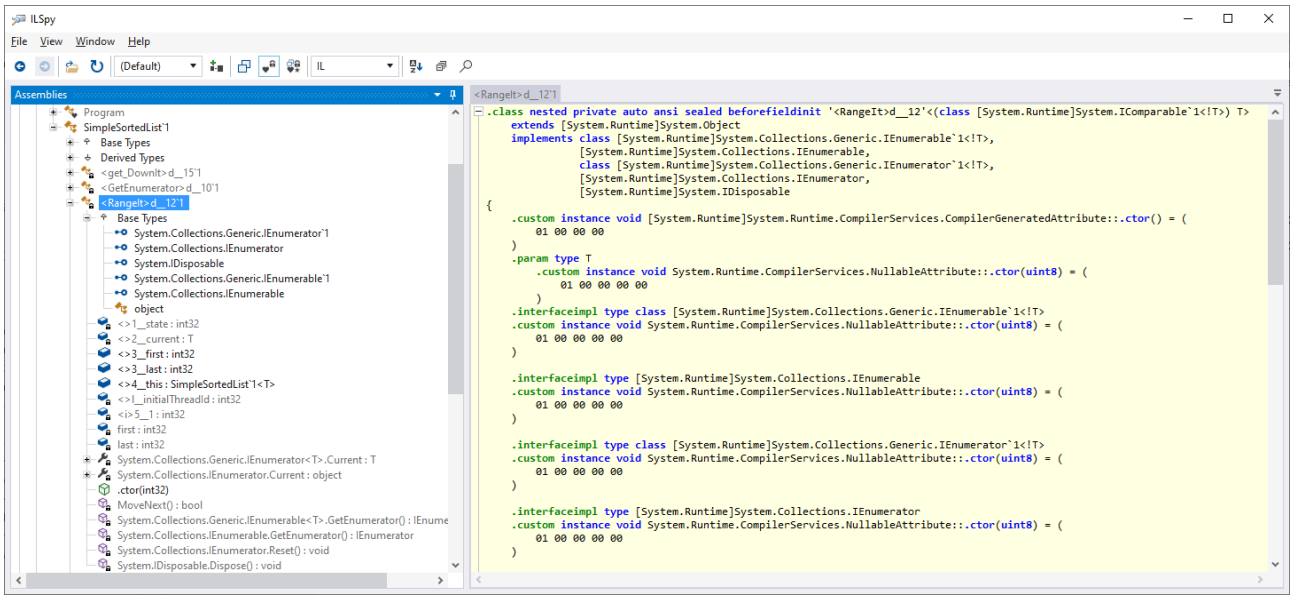
```
public IEnumerable<T> RangeIt(int first, int last) {
   if (first < 0 || last >= size)
      yield break;
   for (int i = first; i <= last; i++)
      yield return data[i];
}
```

In der **for**-Schleife werden nur noch die Kollektionselemente innerhalb des gewünschten Bereichs per **yield return** ausgeliefert. Wird die Methode mit ungeeigneten Bereichsgrenzen aufgerufen, dann liefert sie mit Hilfe der Anweisung **yield break** ein leeres **IEnumerable<T>** - Objekt.

Auch zu dem benannten Iterator erstellt der Compiler eine eingeschachtelte Klasse:

---

[1] Die **using**-Anweisung zur automatisierten Freigabe von Ressourcen ist strikt zu unterscheiden von der **using**-Direktive zum Importieren von Namensräumen in eine Quellcodedatei (siehe Abschnitt 2.6.3).



Das folgende Programm demonstriert die Bereichsiteration ohne und mit Spezifikationsfehler. Im Kopf der **foreach**-Schleife wird die zu durchlaufende Sequenz durch einen Aufruf der Bereichsiterator-Methode erzeugt:[1]

Quellcode	Ausgabe

`var si = new SimpleSortedList<int>(5) { 11, 2, 1, 4, 7 };`
 	
`Console.WriteLine("\nBereichs-Enumeration:");`
`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);`
	`Bereichs-Enumeration:`
	`2`
	`4`
	`7`
	`11`
	
	`Bereichüberschreitung:`

Natürlich lassen sich mit benannten Iteratoren auch andere Aufgaben als die Bereichsauswahl realisieren.

Ein Typ kann benannte Iteratoren *unabhängig* von der Implementation der Schnittstelle **IEnumerable<out T>** anbieten.

Das gilt auch für die benannten Iteratoren mit Eigenschaftssyntax. Zur Demonstration erhält die Klasse `SimpleSortedList<T>` einen Abwärtsiterator namens `DownIt`:

```
public IEnumerable<T> DownIt {
  get {
    for (int i = size - 1; i >= 0; i--)
      yield return data[i];
  }
}
```

Im Kopf der **foreach**-Schleife ist an das Kollektionsobjekt der Eigenschaftsname anzuhängen, um die zu durchlaufende Sequenz zu erzeugen:

---

[1] Im Vorgriff auf Abschnitt 11.3.1 wird ein Kollektionsinitialisierer verwendet, was wegen der plausiblen Syntax vor dem Hintergrund der Erfahrung mit dem analogen Array-Initialisierer didaktisch akzeptabel ist.

Quellcode	Ausgabe

`var si = new SimpleSortedList<int>(5) { 11, 2, 1, 4, 7 };`
 	
`Console.WriteLine("\nAbwärts-Enumeration:");`
`foreach (int i in si.DownIt)`
`  Console.WriteLine(i);`
	`Abwärts-Enumeration:`
	`11`
	`7`
	`4`
	`2`
	`1`

Ein Visual Studio - Projektordner zum Beispiel mit den Iteratoren und Enumeratoren ist hier zu finden:

...\BspUeb\Interfaces\Iteratoren und Enumeratoren

Weitere Details zu Iteratoren bieten z. B. Griffith (2013, S. 175ff) und Mössenböck (2019, S. 169ff).

9.6 Übungsaufgaben zum Kapitel 9

1) Erstellen Sie eine Variante unserer Klasse `Bruch` (vgl. z. B. Abschnitt 5.1.3), welche die Schnittstelle **ICloneable** und die generische Schnittstellen **IComparable<T>** implementiert. Die vorhandene `Bruch`-Methode `Klone()`

```
public Bruch Klone() {
  return new Bruch(zaehler, nenner, etikett);
}
```

wird dabei *nicht* überflüssig, weil sie im Gegensatz zur der von **ICloneable** vorgeschriebenen Methode

public Object Clone()

eine `Bruch`-Referenz abliefert und damit Typumwandlungen erspart.

2) Wie unterscheiden sich Interfaces von abstrakten Klassen?

3) Erweitern Sie in der Klasse `SimpleSortedList` den im Abschnitt 9.5.2 definierten benannten Iterator `RangeIt()` so, dass über einen optionalen Richtungsparameter auch ein Bereich von Elementen in umgekehrter Reihenfolge angefordert werden kann.

10 Delegaten und Ereignisse

In diesem Kapitel ergänzen wir unser Wissen über das Common Type System (CTS) der .NET-Plattform. Wir lernen einen neuen Datentyp kennen und erfahren endlich, um welche speziellen Mitglieder es sich bei den schon oft erwähnten Ereignissen handelt:

- **Delegatenklassen**
  Ein Objekt einer Delegatenklasse verwaltet eine Liste mit Methodenaufrufen, die einen gemeinsamen Rückgabetyp und eine gemeinsame Parameterliste besitzen. Man kann ein Delegatenobjekt wie eine Methode aufrufen, woraufhin alle dort registrierten Methodenaufrufe sukzessive ausgeführt werden.
- **Ereignisse**
  Möchte eine Klasse anderen Klassen die Möglichkeit geben, zu besonderen Gelegenheiten eine Botschaft, d.h. einen Methodenaufruf, zu erhalten, dann bietet sie ein Ereignis an. Interessenten können eine passende Methode bei einem Ereignis registrieren lassen.

Delegatenobjekte spielen eine wesentliche Rolle bei Ereignissen, haben aber auch andere Einsatzfelder (z. B. die Injektion von Funktionalität in Methoden und Typen).

10.1 Delegaten

Ein Objekt aus einer Delegatenklasse kann auf eine Methode mit einem bestimmten Rückgabetyp und mit einer bestimmten Parameterliste zeigen, die von einem bestimmten Akteur (z. B. Klasse oder Objekt) auszuführen ist. Über das Delegatenobjekt lässt sich der registrierte Methodenaufruf veranlassen, sodass z. B. die Übergabe des Delegatenobjekts als Aktualparameter an eine Rahmenmethode dort die Möglichkeit zum Aufruf der registrierten Methode bei passender Gelegenheit schafft. Man kann hier von einer *Funktionsinjektion* oder von einer *Plugin-Methode* sprechen.

Ein Delegatenobjekt darf auch eine *mehrelementige* Liste mit kompatiblen Methodenaufrufen enthalten. Sein Aufruf veranlasst dann mehrere Methodenaufrufe, und die zuletzt ausgeführte Methode dominiert beim Rückgabewert und bei eventuell vorhandenen **out**- oder **ref**-Parametern.

Die Funktionsinjektion ist auch beim Klassen- bzw. Strukturdesign von Nutzen und hier über Felder bzw. Mitgliedsobjekte mit Delegatentyp zu realisieren. Wichtige Beispiele sind die (Steuerelement)klassen von WPF-Anwendungen (z. B. **Application**, **Button**), die Ereignisse anbieten (z. B. **Exit**, **Click**), bei denen Instanzvariablen mit Delegatentyp eine wesentliche Rolle spielen (siehe Abschnitt 10.2). Ereignisinteressenten befördern eine passende Methode in das Delegatenobjekt zu einem Ereignis. Wenn der Anbieter das Ereignis auslöst, dann ruft er das zugehörige Delegatenobjekt auf, sodass die im Delegatenobjekt registrierten Methodenaufrufe nacheinander ausgeführt werden.

Vielleicht helfen die folgenden Begriffserläuterungen dabei, Missverständnisse zu vermeiden:

Bei der Definition eines **Delegatentyps** wird nach dem Schlüsselwort **delegate** angegeben:

- der Rückgabetyp von kompatiblen Methoden, die in die Aufrufliste eines Objekts mit diesem Delegatentyp aufgenommen werden können
- der Name des Delegatentyps
- die Parameterliste von kompatiblen Methoden

Beispiel:

```
delegate void DemoGate(int w);
```

Eine **die Delegatensignatur erfüllende Methode** muss einen passenden Rückgabetyp und eine passende Parameterliste besitzen, z. B.:

```
static void SagA(int w) {
    for (int i = 1; i <= w; i++)
          Console.Write('A');
}
```

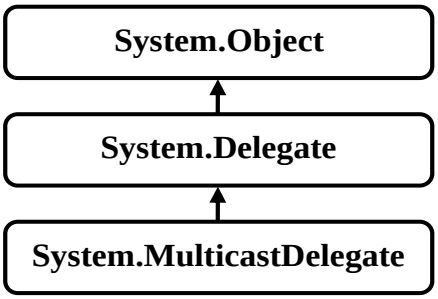
Ein **Delegatenobjekt** ...

- kann per **new**-Operator erstellt werden, indem an den Konstruktor eine kompatible Methode übergeben wird, z. B.:
  `new DemoGate(SagA);`
- enthält eine Aufrufliste mit kompatiblen Methoden

Eine **Delegatenvariable** besitzt ...

- einen Delegatentyp, z.B.
  `var deleVar = new DemoGate(SagA);`
- als Wert ...
  die Adresse eines Delegatenobjekts vom passenden Typ (z. B. `DemoGate`) oder **null**

Alle Delegatentypen sind versiegelte 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 erweitert bzw. konfiguriert 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 Verhaltenskonfiguration ist oft flexibler als die Definition von abgeleiteten Klassen. Allerdings wird die Vererbung, die zu den drei Kernmerkmalen der objektorientierten Programmierung gehört (vgl. Abschnitt 5.1.1), von den Delegaten nicht überflüssig gemacht. Die Definition einer abgeleiteten Klasse ist z. B. empfehlenswert, um eine Basisklasse um *zusätzliche* Felder und Methoden zu erweitern.[1]

[1] https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/inheritance

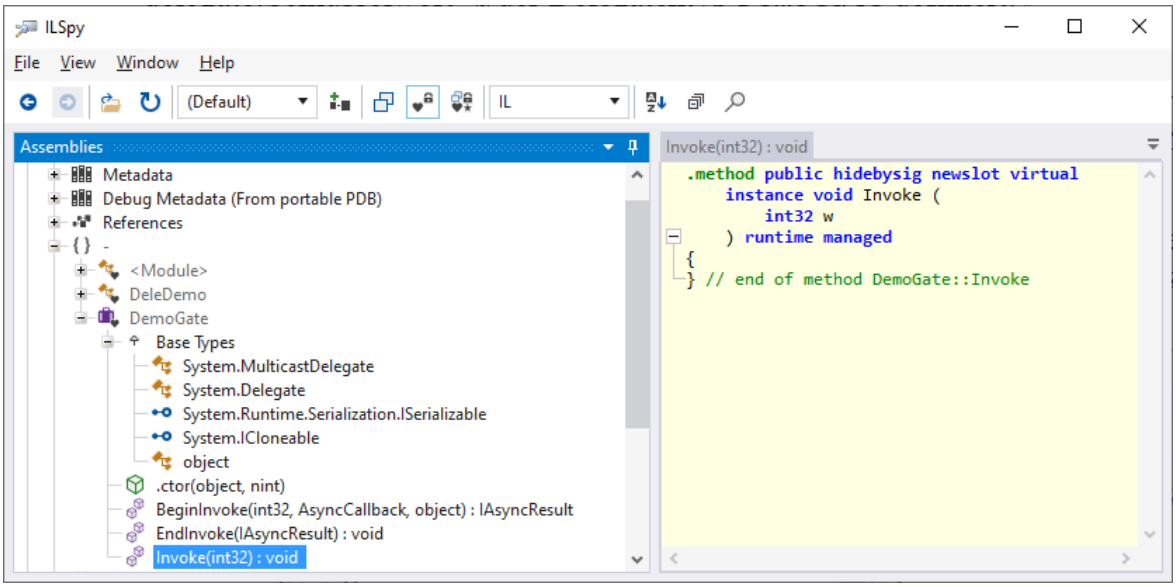
10.1.1 Delegatentypen definieren

Mit der folgenden Anweisung wird unter Verwendung des Schlüsselworts **delegate** der Delegatentyp `DemoGate` definiert:

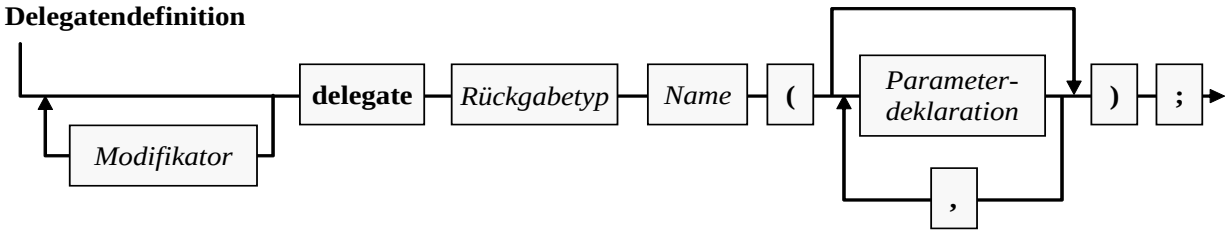
```
delegate void DemoGate(int w);
```

Über Objekte dieses Typs können Instanz- oder Klassenmethoden mit dem Rückgabetyp **void** und einem einzigen Parameter vom Typ **int** aufgerufen werden. Die Namen der Methoden sowie die Namen der Methodenparameter spielen keine Rolle.

Eine Inspektion mit dem Programm ILSpy zeigt, dass aus der obigen Definition die Klasse `DemoGate` entsteht ist, die u. a. einen Konstruktor (**.ctor()**) und die Instanzmethode **Invoke()** besitzt:



Es folgt ein leicht vereinfachtes Syntaxdiagramm zur Definition eines Delegatentyps:



Bei einem Top-Level - Delegaten sind wie bei anderen Top-Level - Typen die Zugriffsmodifikatoren **public**, **internal** und **file** erlaubt. Wird kein Zugriffsmodifikator angegeben, dann ist der Delegat nur innerhalb seines Assemblies verwendbar (Schutzstufe **internal**). Für einen eingeschachtelten Delegaten sind dieselben Zugriffsmodifikatoren verfügbar wie für andere Member (siehe Abschnitt 5.12).

In der Literatur wird oft lax behauptet, ein Delegatenobjekt könne auf Methoden mit einer bestimmten *Signatur* zeigen. Wir wissen aus den Abschnitten 5.3.5 und 8.5, dass zwei Methoden genau dann dieselbe Signatur besitzen, wenn die Namen identisch sind, ggf. (bei generischen Methoden) die Typformalparameterlisten gleich lang sind, und außerdem die Parameterlisten (hinsichtlich Datentyp und Transfermodus aller Formalparameter) übereinstimmen, während die Rückgabetypen *keine* Rolle spielen. Diese Begriffsdefinition ist in der C# - Sprachspezifikation festgelegt (ECMA 2022, Abschnitt 7.6, S. 54f). Für einen Delegatentyp ist aber der Rückgabetyp essenziell, während der Name einer implementierenden Methode keine Rolle spielt. Außerdem muss die Parameterdeklaration einer Methode in einem strengen Sinn mit der Parameterdeklaration des Delegatentyps übereinstimmen:

- Die Anzahl der Parameter muss übereinstimmen.
- Die Datentypen müssen übereinstimmen.
- Die Transfermodi müssen übereinstimmen (Wert- vs. Verweisparameter).
- Bei den Verweisparametern (siehe Abschnitt 5.3.1.3.2) müssen die Transferrichtungen (**in**, **out**, **ref**) übereinstimmen.

Wir werden im weiteren Verlauf die Anforderungen einer Delegatendefinition an kompatible Methoden als *Delegatensignatur* bezeichnen.

Die im Abschnitt 10.1.6 beschriebene Ko- und Kontravarianz für sogenannte *Methodengruppen* erlaubt eine Zuweisungsliberalität. Die zum Erstellen eines Delegatenobjekts verwendete Methode darf im Vergleich zur Delegatensignatur ...

- einen spezielleren Rückgabetyp
- und allgemeinere Parametertypen haben.

C# erlaubt auch eine *generische* Delegatendefinition. Ein wichtiges Beispiel ist der Delegatentyp **Predicate<in T>** aus dem BCL-Namensraum **System** mit einem kontravarianten Typformalparameter **T** (siehe Abschnitt 10.1.6):

public delegate bool Predicate<in T>(T *x*)

Ein Objekt der Konkretisierung **Predicate<int>** kann z. B. auf der folgenden Methode basieren (definiert per Lambda-Symbol und Ausdruck, siehe Abschnitt 5.8.1):

```
static bool IsEven(int i) => i % 2 == 0;
```

10.1.2 Delegatenobjekte erzeugen und aufrufen

In diesem Abschnitt wird an einem einfachen Beispiel demonstriert, …

- wie der Aufruf einer Methode,
- die eine bestimmte Delegatensignatur erfüllt,
- über ein Objekt mit dem passenden Delegatentyp zu realisieren ist.

Die folgende Klasse `DeleDemo`

```
class DeleDemo {
   static void SagA(int w) {
      for (int i = 1; i <= w; i++)
         Console.Write('A');
      Console.WriteLine();
   }
   static void Main() {
      var deleVar = new DemoGate(SagA);
      deleVar(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 im Abschnitt 10.1).
- In der Methode **Main()** wird die lokale Referenzvariable `deleVar` vom Delegatentyp `DemoGate` deklariert und initialisiert. Über den implizit definierten `DemoGate`-Konstruktor wird ein Objekt des Delegatentyps erzeugt.

In der Aufrufliste des über die Referenzvariable `deleVar` ansprechbaren `DemoGate`-Objekts befindet sich ausschließlich die von der Klasse `DeleDemo` auszuführende statische Methode `SagA()`.

Ein Delegatenobjekt lässt sich wie eine Methode aufrufen. Im Beispiel führt der Aufruf

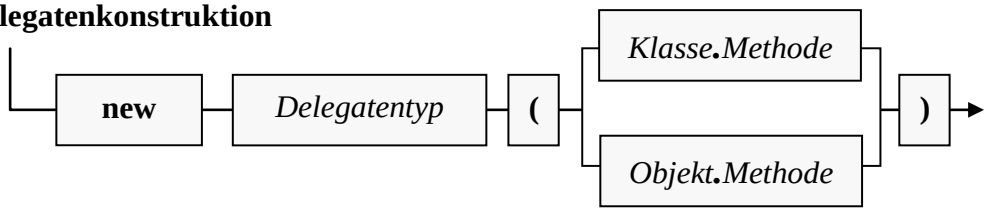
```
deleVar(3);
```

zur Ausgabe:

```
AAA
```

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):

Explizite Delegatenkonstruktion

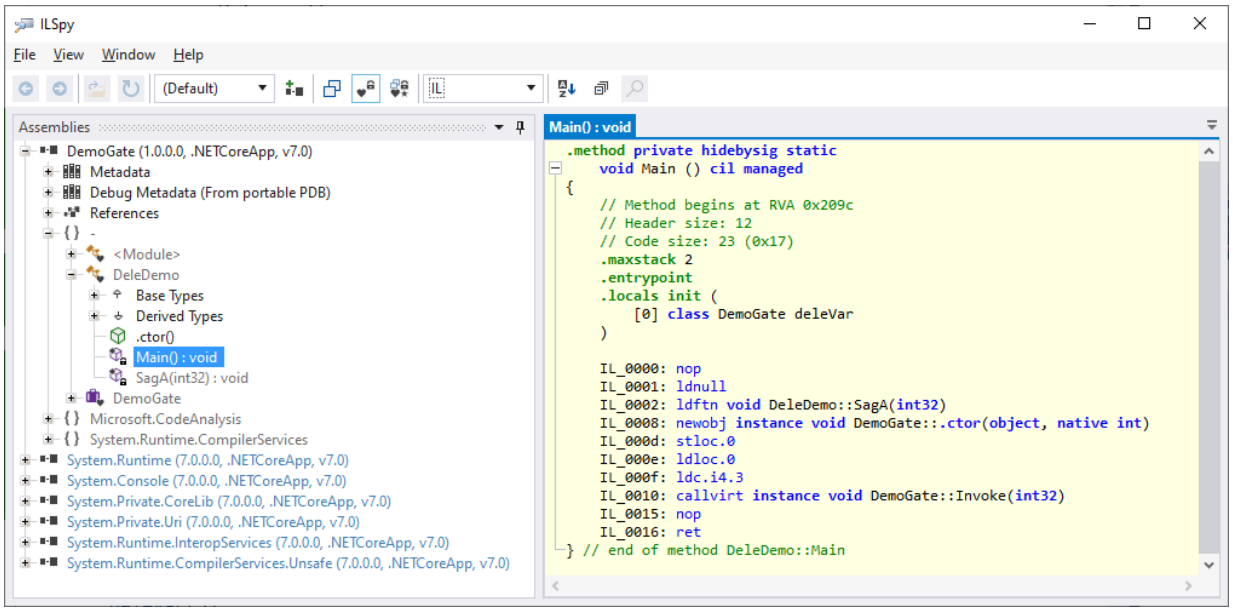


Dabei wird an den Methodennamen *keine* Parameterliste angehängt.

Eine Assembly-Inspektion mit ILSpy zeigt, dass die Anweisungen

```
deleVar = new DemoGate(SagA);
deleVar(3);
```

in der `DeleDemo`-Methode **Main()**



folgendes bewirken

- Es wird ein `DemoGate`-Objekt erstellt (IL-OpCode **newobj**).
- Dessen Adresse landet in der Variablen `deleVar` (IL-OpCode **stloc.0**).
- Das Objekt wird beauftragt, die `DemoGate`-Methode **Invoke()** auszuführen (IL-OpCode **callvirt**).

Anstelle der expliziten Delegatenobjektkreation

```
DemoGate deleVar = new DemoGate(SagA);
```

erlaubt der Compiler auch die folgende implizite Variante:[1]

```
DemoGate deleVar = SagA;
```

[1] Diese Vereinfachung wird als *Methodengruppen-Konvertierung* bezeichnet (engl.: *method group conversion*), siehe https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/conversions

Eine Methode in die Aufrufliste eines Delegatenobjekts aufzunehmen und schlussendlich auf indirekte Weise aufzurufen, ist natürlich nicht der Zweck der Delegatentechnik. Viel praxisrelevanter ist die Möglichkeit, über einen Parameter oder ein Feld mit Delegatentyp eine Funktionalität in eine Methode oder in einen Typ 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 generischen Delegatentyp **Comparison<T>** erfüllt:

public delegate int Comparison<in T>(T *x***, T** *y***)**

Beim **Sort()** - Aufruf übergibt man eine Methode, die für zwei **T**-Instanzen die Anordnung definiert, und steuert so die Sortierung. Die Rahmenmethode **Sort()** ruft die injizierte Methode auf, um für zwei **T**-Instanzen die Anordnung zu ermitteln.

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:[1]

```
static int EvenLower(int first, int second) {
   if (first % 2 == second % 2)
      return first.CompareTo(second);
   if (first % 2 == 0)
      return -1;
   return 1;
}
```

Wird die folgende Liste mit **int**-Werten[2]

```
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.Write(i + " ");
```

zum Ergebnis:

```
2 4 6 1 3 5
```

Gerade wurde per Delegatenobjekt eine Verhaltenskonfiguration bzw. Funktionserweiterung in eine Rahmenmethode übertragen. Oft steht bei der Delegatenübergabe ein kommunikativer Zweck im Vordergrund: Durch den Aufruf eines Delegatenobjekts kann die Rahmenmethode z. B. über die Beendigung einer Aufgabe informieren.[3]

---

1 Sind die beiden Zahlen Elemente der gleichen Nebenklasse mod(2) (Nebenklasse im Sinne der Gruppentheorie), dann greift die übliche Vergleichsmethode. Andernfalls gelten die geraden Zahlen als "kleiner".

2 Im Vorgriff auf Abschnitt 11.3.1 wird ein Kollektionsinitialisierer verwendet, was wegen der plausiblen Syntax vor dem Hintergrund der Erfahrung mit dem analogen Array-Initialisierer didaktisch akzeptabel ist.

3 Siehe z. B.: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates

10.1.3 Delegatenobjekte kombinieren

Nach dem Motto „Wer *A* sagt, muss auch *B* sagen“ erweitern wir die Klasse `DeleDemo` aus dem Abschnitt 10.1.2 um die statische 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:

```
deleVar += 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 `deleVar` referenzierten Objekt kombiniert. Dabei entsteht ein weiteres `DemoGate`-Delegatenobjekt mit einer *zweielementigen* Aufrufliste, und dessen Adresse landet in der Referenzvariablen `deleVar`. Die beiden Delegatenobjekte mit einelementiger Aufrufliste werden zu obsoletem Müll.

Delegatenobjekte sind ebenso unveränderlich wie z. B. die Objekte der Klasse **String** (vgl. Abschnitt 6.3.1.1).

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.:

```
deleVar -= SagB;
```

Beim kompletten Entleeren der Aufrufliste entsteht kein leeres Delegatenobjekt, sondern die Delegatenvariable erhält den Wert **null**. Über eine Delegatenvariable lassen sich also genau dann registrierte Methoden starten, wenn ihr Wert von **null** verschieden ist. Zeigt eine Delegatenvariable auf **null**, dann kommt es beim Startversuch zum einem Ausnahmefehler vom Typ **NullReferenceException**.

Neben den Aktualisierungsoperatoren += und -= kann man auch den Additions- und den Subtraktionsoperator verwenden, z. B.:

```
deleVar = deleVar + new DemoGate(SagB);
```

Besitzt ein Delegatenobjekt eine *mehrelementige* Aufrufliste, dann spricht man von einem *Multicast-Delegaten*. 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**- oder **ref**-Parametern dominiert.

Ein Visual Studio - Projekt zum `DemoGate`-Beispiel ist hier zu finden:

...\BspUeb\Delegaten und Ereignisse\Delegaten\DemoGate

10.1.4 Delegaten versus Schnittstellen

Die Injektion von Funktionalität in eine Klasse oder eine Methode kann oft alternativ über …

- ein Objekt mit einem Delegatentyp
- oder ein Objekt mit einem Schnittstellentyp

erfolgen. 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 eine Methode beigesteuert, die für zwei Instanzen vom Typ **T** per **int**-Rückgabe die Anordnung festlegt. Die beiden Überladungen sehen ähnlich aus:

- **public void Sort(Comparison<T>** *comparison***)**
  Diese aus einem Beispiel im Abschnitt 10.1.2 bekannte 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 10.1.6). 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, dann eignet sich z. B. die folgende Methode:
  ```
  public 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:
  public interface IComparer<in T>
  Auch der Typformalparameter **T** der generischen Schnittstelle **IComparer<T>** ist über das Schlüsselwort **in** als kontravariant definiert (siehe Abschnitt 9.1.4.2). Wird **T** durch **String** konkretisiert, dann muss eine Methode mit dem folgenden Definitionskopf vorhanden sein:
  public int Compare(String *x***, String** *y***)**

Für ein Design mit Delegatentyp sprechen die folgenden Argumente:

- Weil der Methodenname für die Delegatensignatur irrelevant ist, kann eine Klasse *mehrere* Methoden enthalten, die *denselben* Delegatentyp erfüllen und dabei unterschiedlich agieren. 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.
- Über eine anonyme Methode oder einen Lambda-Ausdruck (siehe Abschnitt 10.1.5) lässt sich ein Delegatenobjekt syntaktisch elegant „in-line“ realisieren, während zum Implementieren einer Schnittstelle eine explizite Typdefinition erforderlich ist.

Bei einem Design mit Schnittstelle kann man allerdings den implementierenden Klassen beliebig viele Methoden diktieren, sodass deren Objekte über mehrere Kompetenzen verfügen. Über einen Delegatentyp lässt sich demgegenüber nur *eine* Methode vorschreiben.

10.1.5 Delegatenobjekte durch anonyme Funktionen erstellen

Statt ein Delegatenobjekt mit Hilfe einer anderenorts definierten Methode zu erzeugen, kann ein Ausdruck mit einer sogenannten *anonymen Funktion* verwendet werden. Dabei sind aus historischen Gründen zwei syntaktische Varianten verfügbar:

- die anonyme Methoden
- die Lambda-Notation

Die C# - Sprachspezifikation (ECMA 2022, 11.16, S. 234) spricht sich klar für die Lambda-Notation aus:

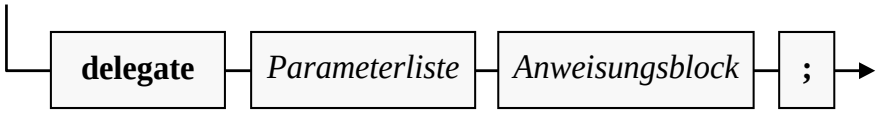
> For historical reasons, there are two syntactic flavors of anonymous functions, namely *lambda_expressions* and *anonymous_method_expressions*. For almost all purposes, *lambda_expressions* are more concise and expressive than *anonymous_method_expressions*, which remain in the language for backwards compatibility.

Wir stellen anschließend beide Varianten vor.

10.1.5.1 Anonyme Methoden

Statt beim Erzeugen eines Delegatenobjekts den Namen einer vorhandenen, andernorts definierten Methode zu übergeben, kann man einen Anweisungsblock formulieren, dem das Schlüsselwort **delegate** samt Delegatentyp-konformer Parameterliste vorangestellt wird:[1]

Delegatenobjekt aus einer anonymen Methode



Dies wird in der folgenden Variante des Beispiels aus dem Abschnitt 10.1.2 demonstriert:

Quellcode	Ausgabe

```
using System;

delegate void DemoGate(int w);

class AnoMeth {
   static void Main() {
      DemoGate deleVar =
         delegate (int w) {
            for (int i = 1; i <= w; i++)
               Console.Write('A');
            Console.WriteLine();
         };
      deleVar(3);
   }
}
```

```
AAA
```

Die benötigte Funktionalität lässt sich vor Ort realisieren, und man muss keine Kreativität in einen Methodennamen investieren.

Bei entsprechender Definition des zu erfüllenden Delegatentyps können (und müssen) anonyme Methoden selbstverständlich auch einen Rückgabewert liefern, z. B.:

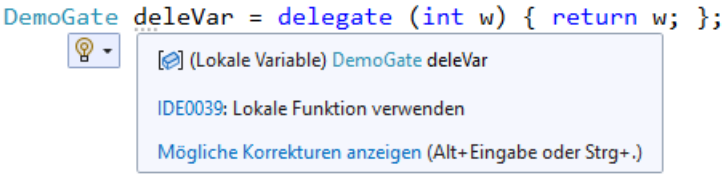
---

[1] Laut C# - Sprachspezifikation kommt hier der **delegate**-Operator zum Einsatz, siehe: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/delegate-operator

Quellcode	Ausgabe

`using System;`
 	
`delegate int DemoGate(int w);`
 	
`class AnoMeth {`
`  static void Main() {`
`  DemoGate deleVar = delegate (int w) {return w;};`
`  Console.WriteLine(deleVar(3));`
`  }`
`}`
	`3`

Ein Nachteil anonymer Methoden besteht darin, dass sie nicht an anderen Stellen genutzt werden können. Das Visual Studio empfiehlt daher, statt einer anonymen Methode eine *lokale* Methode zu verwenden (siehe Abschnitt 5.3.1.5), z. B.:



Oft besteht aber kein Bedarf für den Einsatz einer anonymen Methode an mehreren Orten.

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 deleVar =`
`  delegate (int w) {`
`  for (int i = 1; i <= w; i++)`
`  Console.Write(c);`
`  Console.WriteLine();`
`  c++;`
`  };`
`  Console.WriteLine("c vor dem Delegatenaufr.:  " + c);`
`  deleVar(3);`
`  Console.WriteLine("c nach dem Delegatenaufr.: " + c);`
`  }`
`}`
	`c vor dem Delegatenaufr.:  A`
	`AAA`
	`c nach dem Delegatenaufr.: B`

Wie das Beispiel zeigt, darf eine anonyme Methode die Umgebungsvariablen auch *verändern*. Mit dem Zugriff von anonymen Funktionen auf Kontextvariablen werden wir uns im Abschnitt 10.1.5.3 noch ausführlich beschäftigen.

Über anonyme Methoden erstellte Delegatenobjekte lassen sich nicht nur zur Initialisierung von Delegatenvariablen verwenden, sondern z. B. auch als Aktualparameter:

Quellcode	Ausgabe

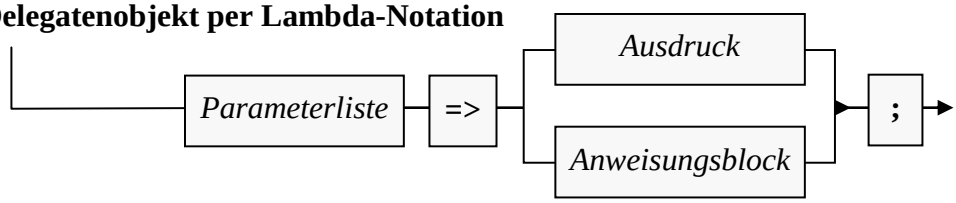
`var le7 = new List<int>() { 1, 2, 3, 4, 5, 6, 7 };`
`var evenLe7 = le7.FindAll(delegate (int i) { return i % 2 == 0; });`
`foreach (int n in evenLe7)`
`    Console.Write(n + " ");`
	`2 4 6`

10.1.5.2 Lambda-Notation

Die mit C# 3.0 zur Unterstützung der LINQ-Technik eingeführte *Lambda-Notation* ermöglicht im Vergleich zu den im Abschnitt 10.1.5.1 vorgestellten anonymen Methoden eine flexiblere „in-line“ - Realisation von Delegatenobjekten.[1]

Man kann einen einzelnen Ausdruck (Ausdrucks-Lambda) oder einen (möglichst knappen) Anweisungsblock (Anweisungs-Lambda) formulieren:

Delegatenobjekt per Lambda-Notation



Für den generischen Delegatentyp **Predicate<in T>** aus dem BCL-Namensraum **System** mit einem kontravarianten Typformalparameter **T** (siehe Abschnitt 10.1.6) und einer booleschen Rückgabe

public delegate bool Predicate<in T>(T *x***)**

sind anschließend äquivalente Realisationen über eine anonyme Methode

```
Predicate<int> even = delegate (int i) { return i % 2 == 0; };
```

und über einen Lambda-Ausdruck zu sehen:

```
Predicate<int> even = i => i % 2 == 0;
```

Bei der Parameterliste eines Lambda-Ausdrucks besteht einige Flexibilität:

- Bei einem Delegatentyp ohne Parameter ist eine leere Parameterliste anzugeben:
  ```
  Action hallo = () => Console.WriteLine("Hallo");
  ```
- Man kann auf die Angabe der Parameter*typen* verzichten, weil sich diese aus dem zu erfüllenden Delegatentyp zwingend ergeben, z. B.:
  ```
  Predicate<int> even = (i) => i % 2 == 0;
  ```
  Das vorhandene Wissen des Compilers wird per Lambda-Syntax besser ausgenutzt als bei anonymen Methoden.
- Bei einem einzelnen, implizit typisierten Parameter kann man die runden Klammern weglassen, z. B.:
  ```
  Predicate<int> even = i => i % 2 == 0;
  ```
- Zwei oder mehr Parameter werden durch Kommata getrennt, z. B.:
  ```
  Func<string, string, bool> testForEquality = (s1, s2) => s1 == s2;
  ```
- Seit C# 9 dürfen überflüssige Parameter durch das Ausschluss-Symbol ersetzt werden, z. B.:
  ```
  Func<string, string, bool> alwaysTrue = (_, _) => true;
  ```

---

[1] Mittlerweile lassen sich per Lambda-Notation auch benannte Methoden sowie Eigenschaften und Indexer erstellen (siehe Abschnitt 5.8.1, über die sogenannten *expression bodied members*).

Bei der Delegatenkonstruktion per Lambda-Notation resultiert ein Delegatenobjekt, und den Typ dieses Objekts kann man dem rechten Argument des Lambda-Operators zuordnen.[1] Seit C# 10 hat das rechte Argument des Lambda-Operators einen **natürlichen Typ**, wenn …

- die Parametertypen explizit angegeben werden,
- und der Rückgabetyp explizit angegeben wird oder erschlossen werden kann.

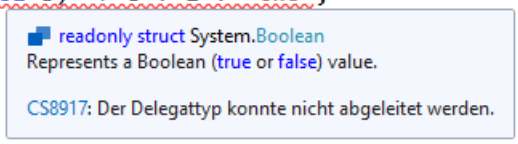
Das ermöglicht bei lokalen Variablen eine implizite Typisierung, z. B.:

```
var intComp = (int i, int j) => i.CompareTo(j);
```

Der Compiler wählt nach Möglichkeit einen **Func**- oder **Action**-Delegatentyp, sodass im Beispiel für die Variable `intComp` der Typ **Func<int, int, int>** resultiert.

Ein Ausdrucks-Lambda ohne definierten Rückgabetyp entsteht z. B. durch die Verwendung des Konditionaloperators, z. B.:[2]

```
var oneOrTwo = (bool b) => b ? 1 : "two";
```



Seit C# 10 kann man in solchen Fällen den Rückgabetyp explizit vor der Parameterliste angeben, z. B.:

```
var oneOrTwo = object (bool b) => b ? 1 : "two";
```

Im Beispiel resultiert für das Ausdrucks-Lambda der Delegatentyp **Func<bool, object>**.

Wird ein Delegatentyp mit einem von **void** verschiedenen Rückgabetyp durch ein Anweisungs-Lambda realisiert, dann muss der Rückgabewert per **return**-Anweisung geliefert werden, z. B.:

```
Predicate<string> palindrom = s => {
    int len = s.Length;
    var sb = new StringBuilder(len);
    for (int i = 0; i < len; i++)
        sb.Append(s[len - i - 1]);
    return sb.ToString().ToUpper() == s.ToUpper();
};
```

Für das `DemoGate`-Beispiel bringt die Lambda-Syntax im Vergleich zu einer anonymen Methode keinen großen Vorteil:

---

[1] Durch die etwas umständliche Formulierung wird vermieden, dem Vorbild von Microsoft folgend von einem *Lambdaausdruck* zu sprechen, der per Ausdruck oder Anweisungsblock realisiert wird.

[2] Das Beispiel wurde übernommen von:
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions

Quellcode	Ausgabe

`using System;`
`delegate void DemoGate(int w);`
`class LambdaDelegate {`
`  static char c = 'A';`
`  static void Main() {`
`  DemoGate deleVar = w => {`
`  for (int i = 1; i <= w; i++)`
`  Console.Write(c);`
`  Console.WriteLine();`
`  };`
`  deleVar(3);`
`  }`
`}`
	`AAA`

Man spart sich das Schlüsselwort **delegate** und kann beim Parameter die Fähigkeiten des Compilers zur Typinferenz nutzen.

Das Beispiel demonstriert, dass ein Ausdrucks- oder Anweisungs-Lambda (wie eine anonyme Methode) auf Variablen in der Umgebung (hier: auf ein statisches Feld) zugreifen kann. Mit dem Zugriff von anonymen Funktionen auf Kontextvariablen werden wir uns im Abschnitt 10.1.5.3 noch ausführlich beschäftigen.

Deutlicher als bei der Wertzuweisung für eine Delegatenvariable zeigt sich die syntaktische Eleganz der Lambda-Notation bei Methodenparametern mit einem Delegatentyp. Mit Hilfe der Methode **FindAll()** der generischen Kollektionsklasse **List<T>**, die einen Parameter vom Delegatentyp **Predicate<T>** besitzt, ermittelt man aus einer Liste alle Elemente mit einer bestimmten Eigenschaft:

public List<T> FindAll(Predicate<T> *match***)**

Im folgenden Beispiel werden aus einer Liste mit **int**-Werten die geraden Zahlen extrahiert:[1]

Quellcode	Ausgabe

`var le7 = new List<int>() { 1, 2, 3, 4, 5, 6, 7 };`
`var evenLe7 = le7.FindAll(n => n % 2 == 0);`
`foreach (int n in evenLe7)`
`  Console.Write(n + " ");`
	`2 4 6`

Aus der Lambda-Notation kann der Compiler nicht nur ein Delegatenobjekt erstellen, sondern auch einen sogenannten *Ausdrucksbaum*. Damit beschäftigt sich das Kapitel 20 in Baltes-Götz (2021) im Zusammenhang mit der Datenbankabfrage über eine Technik namens *LINQ-to-Entities*.

[1] Wir leisten uns wieder mal einen Vorgriff auf das Kapitel **11** über die Kollektionen und verwenden insbesondere einen Kollektionsinitialisierer, was wegen der plausiblen Syntax vor dem Hintergrund der Erfahrung mit dem analogen Array-Initialisierer didaktisch akzeptabel ist.

10.1.5.3 Zugriff auf Kontextvariablen

Wir haben in den Abschnitten 10.1.5.1 und 10.1.5.2 gesehen, dass anonyme Funktionen auf …

- lokale Variablen der umgebenden Methode
- sowie auf (statische) Felder der Umgebung

zugreifen dürfen.[1] In diesem Zusammenhang sind die folgenden Begriffe in der Literatur verbreitet:

- Die für eine anonyme Funktion *verfügbaren* Kontextvariablen werden als *äußere Variablen* (engl.: *outer variables*) bezeichnet.
- Die von der anonymen Funktion *verwendeten* Kontextvariablen nennt man *eingefangene Variablen* (engl.: *captured variables*).
- Die Zusammenfassung einer anonymen Funktion mit den von ihr eingefangenen Variablen bezeichnet man als *Abschluss* (engl.: *closure*).

Eingefangene Variablen werden nicht beim Erstellen eines Delegatenobjekts ausgewertet, sondern beim Aufruf, z. B.:

```
int tolLen = 3;
Predicate<string> isShorterThan = s => s.Length <= tolLen++;
tolLen = 5;
Console.WriteLine(isShorterThan("123456")); // Ausgabe: False
Console.WriteLine(isShorterThan("123456")); // Ausgabe: True
```

Das Beispiel demonstriert, dass eine anonyme Funktion die eingefangenen Variablen auch verändern kann.

Eine Methode kann ein von ihr mit Hilfe einer anonymen Funktion erzeugtes Delegatenobjekt per Rückgabe abliefern, und oft ist die erzeugende Methode schon beendet, wenn das Delegatenobjekt aufgerufen wird. Im folgenden Beispiel

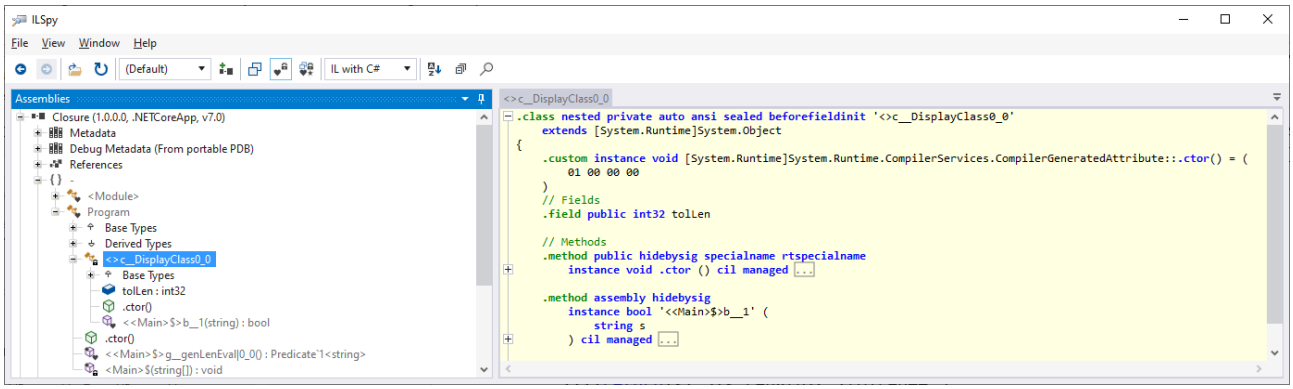
```
var isShorterThan = genLenEval();
Console.WriteLine(isShorterThan("123456")); // Ausgabe: False
Console.WriteLine(isShorterThan("123456")); // Ausgabe: True

static Predicate<string> genLenEval() {
    int tolLen = 5;
    return s => s.Length <= tolLen++;
}
```

wird ein Delegatenobjekt namens `isShorterThan` von einer statischen Methode unter Verwendung einer eingefangenen lokalen Variablen erstellt. Beim Aufruf des Delegatenobjekts ist die Methoden beendet, und ihre lokale Variable ist vom Stack verschwunden.

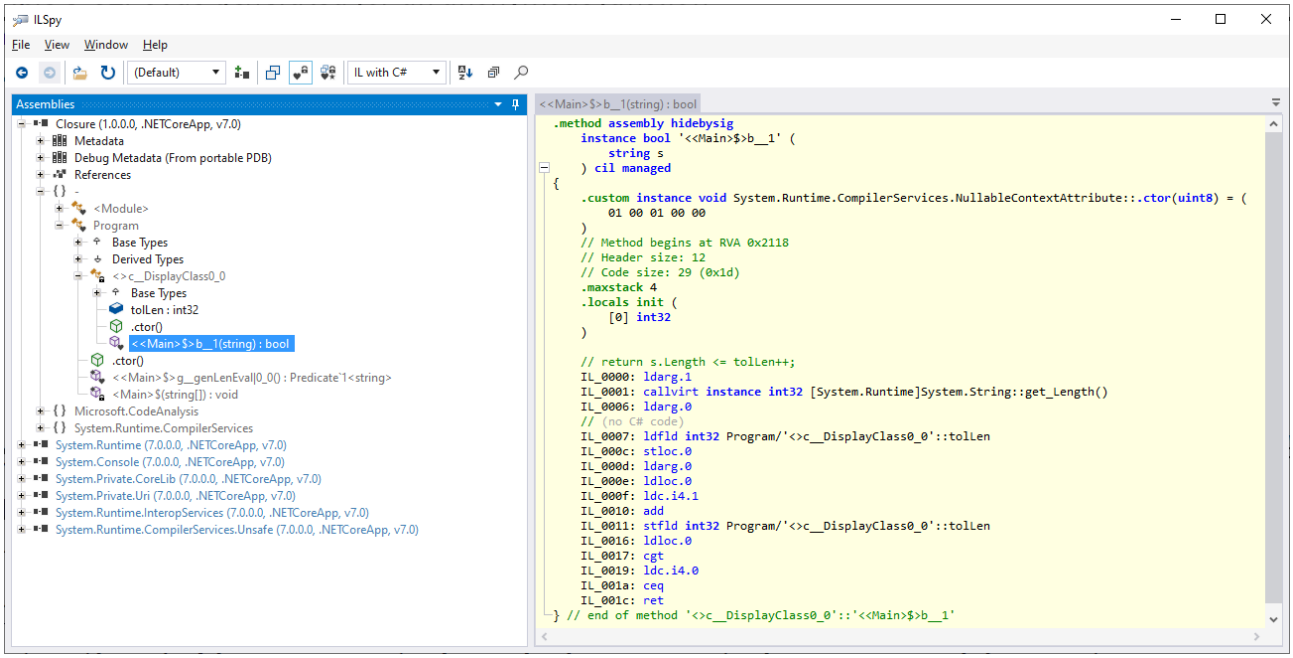
Damit ein Delegatenaufruf trotzdem klappt, erstellt der Compiler zur Aufbewahrung der eingefangenen Variablen ein Abschlussobjekt auf dem Heap. Es gehört zu einer geschachtelten Klasse mit einem in C# unzulässigen Namen, die als privat und versiegelt deklariert ist:

---

[1] Bei den lokalen Variablen sind die Verweisparameter (mit den Parametermodifikatoren **ref**, **in** und **out**) vom Zugriff ausgenommen.



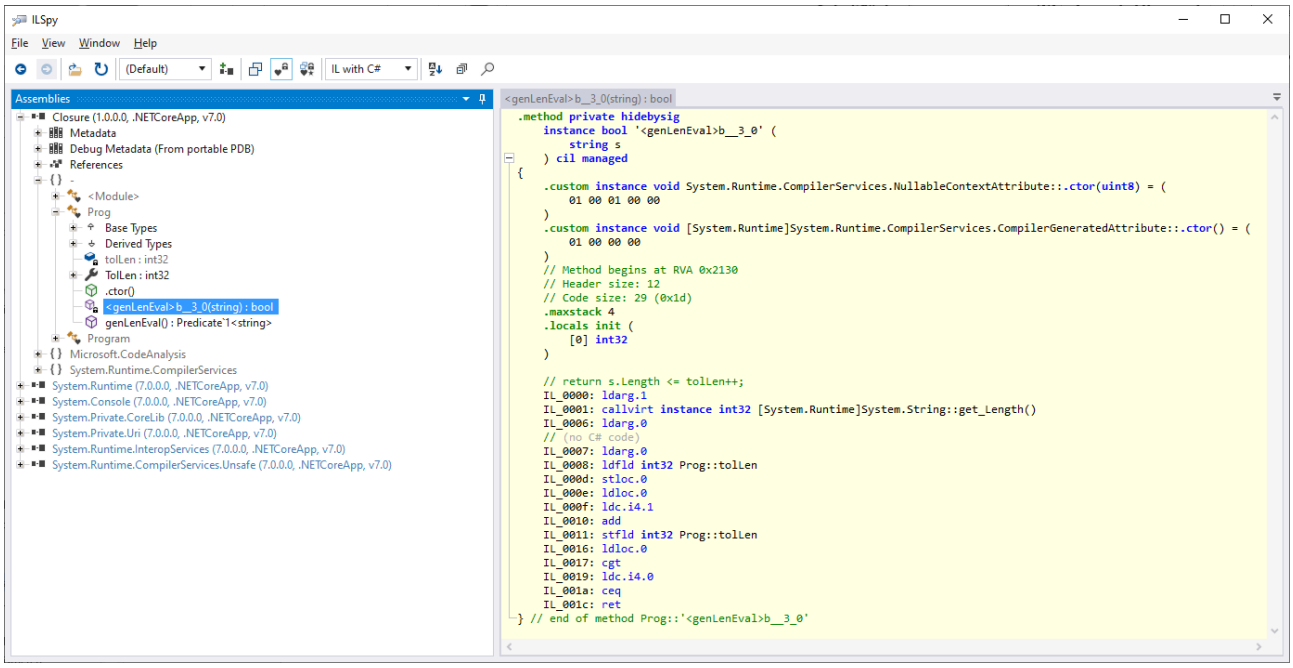
In dieser Closure-Klasse befinden sich die ehemalige lokale Variable als Feld und die Delegatenfunktionalität:



Im Fall einer eingefangenen Instanzvariablen

```
Prog p = new();
var isShorterThan = p.genLenEval();
Console.WriteLine(isShorterThan("123456")); // Ausgabe: False
Console.WriteLine(isShorterThan("123456")); // Ausgabe: True
Console.WriteLine(p.TolLen);                // Ausgabe: 7

class Prog {
    int tolLen = 5;
    public int TolLen { get => tolLen; }
    public Predicate<string> genLenEval() {
        return s => s.Length <= tolLen++;
    }
}
```

erstellt der Compiler in der betroffenen Klasse keine private geschachtelte Klasse, sondern eine private Methode mit der Delegatenfunktionalität, z. B.:



Wie das Beispiel zeigt, haben die Delegatenaufrufe einen Einfluss auf das weitere Leben des generierenden Objekts.

Seit C# 9 kann mit dem Modifikator **static** verhindert werden, dass eine anonyme Funktion lokale Variablen oder Instanzvariablen einfängt, während der Zugriff auf statische Variablen erlaubt bleibt, z. B.:

```
public  Predicate<string> genLenEval() {
    int tolLen = 1;
    return static s => s.Length <= tolLen++;
}
```

(Lokale Variable) int tolLen

CS8820: Eine statische anonyme Funktion kann keinen Verweis auf "tolLen" enthalten.

Ein Visual Studio - Projekt mit den im aktuellen Abschnitt vorgeführten Beispielen ist hier zu finden:

...\BspUeb\Delegaten und Ereignisse\Delegaten\Closure

10.1.6 Generische Delegaten, Ko- und Kontravarianz

Neben generischen Klassen, Strukturen, Schnittstellen und Methoden (siehe Kapitel 8) unterstützt C# auch generische Delegaten. Der Type **Comparison<T>** aus dem BCL-Namensraum **System**

```
public delegate int Comparison<in T>(T x, T y);
```

wurde bereits im Abschnitt 10.1.4 vorgestellt. Er kommt in einer **Sort()** - Überladung der generischen Kollektionsklasse **List<T>** (vgl. Abschnitt 8.1) als Parameterdatentyp zum Einsatz. Im folgenden Beispiel wird über ein **Comparison<String>** - Objekt, das auf eine geeignet konstruierte **String**-Vergleichsmethode zeigt, dafür gesorgt, dass in einer sortierten Namensliste „Karl“ stets der Größte ist:

Quellcode	Ausgabe

`var li = new List<string> { "Fritz", "Karl", "Anita", "Theo" };`
`li.Sort(ComKarlison);`
`foreach (var s in li)`
`    Console.WriteLine(s);`
 	
`static int ComKarlison(string a, string b) {`
`    switch (a.CompareTo(b)) {`
`        case -1: return a.Equals("Karl") ?  1 : -1;`
`        case  1: return b.Equals("Karl") ? -1 :  1;`
`        default: return 0;`
`    }`
`}`
	`Anita`
	`Fritz`
	`Theo`
	`Karl`

Weil im generischen Delegatentyp **Comparison<in T>** aus der BCL der Typformalparameter über den Varianzmodifikator **in** als kontravariant deklariert ist, kann unter den folgenden, schon mehrfach in Beispielen verwendeten Voraussetzungen

```
public class Figur { ... }
public class Kreis : Figur { ...}
```

einer Delegatenvariablen vom Typ **Comparison**<`Kreis`> ein Delegatenobjekt vom Typ **Comparison**<`Figur`> zugewiesen werden.

Im folgenden Beispielprogramm wird einiger Aufwand betrieben, um die Notwendigkeit des Varianzmodifikators **in** für den Typformalparameter des Delegatentyps beobachten zu können:

- Es wird eine Methode zum Sortieren von Kreisen mit dem Auswahlverfahren definiert (siehe Abschnitt 6.9).
- Zur Definition der Ordnung dient der Delegatentyp `Vergleich`<**in T**>.

Obwohl die Methode `Sortiere()` als dritten Parameter ein Delegatenobjekt vom Typ `Vergleich<Kreis>` verlangt, wird beim Aufruf der Methode auch ein Delegatenobjekt vom Typ `Vergleich<Figur>` akzeptiert, weil der Typformalparameter des generischen Delegatentyps als kontravariant deklariert ist:

Quellcode	Ausgabe

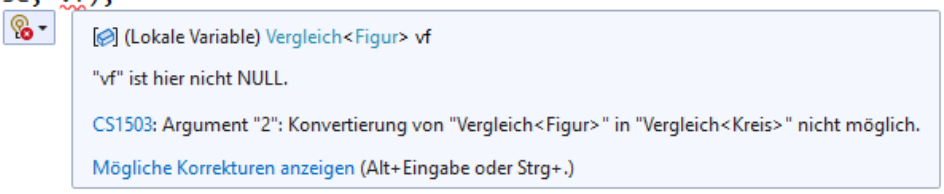
`var kreise = new Kreis[] { new Kreis(2, 2, 1),`
`                           new Kreis(8, 4, 1),`
`                           new Kreis(1, 4, 1) };`
`Vergleich<Figur> vf = (f1, f2) => f1.X.CompareTo(f2.X);`
`Sortiere(kreise, vf);`
`foreach (Kreis k in kreise)`
`    Console.WriteLine(k.X);`
 	
`static void Sortiere(Kreis[] kreise, Vergleich<Kreis> vk) {`
`    Kreis tmp;`
`    int len = kreise.Length;`
`    for (int i = 0; i < len - 1 ; i++)`
`        for (int j = i + 1; j < len; j++)`
`            if (vk(kreise[j], kreise[i]) < 0) {`
`                tmp = kreise[i];`
`                kreise[i] = kreise[j];`
`                kreise[j] = tmp;`
`            }`
`}`
 	
`public delegate int Vergleich<in T>(T x, T y);`
	`1`
	`2`
	`8`

Ist der Typformalparameter von `Vergleich<T>` *nicht* als kontravariant deklariert,

```
public delegate int Vergleich<T>(T x, T y);
```

dann verbietet der Compiler hingegen die Verwendung eines `Vergleich<Figur>` - Delegaten an Stelle eines `Vergleich<Kreis>` - Delegaten:

```
var kreise = new Kreis[] { new Kreis(2, 2, 1),
                           new Kreis(8, 4, 1),
                           new Kreis(1, 4, 1) };
Vergleich<Figur> vf = (f1, f2) => f1.X.CompareTo(f2.X);
Sortiere(kreise, vf);
```



Weil im Delegatentyp **Func<out T>** aus dem BCL-Namensraum **System** der Typformalparameter über den Varianzmodifikator **out** als kovariant definiert ist,

```
public delegate TResult Func<out TResult>();
```

kann in der `Figur-Kreis` - Konstellation auch ein Objekt vom Delegatentyp **Func<Kreis>** verwendet werden (z. B. als Aktualparameter), wenn laut Deklaration ein Objekt vom Delegatentyp **Func<Figur>** erwartet wird. Im folgenden Beispiel wird der BCL-Delegatentyp **Func<T>** durch die Eigenkreation `Funk<T>` ersetzt, damit die Notwendigkeit der **out**-Deklaration des Typformalparameters demonstriert werden kann. Die zur Ortsdokumentation dienende Methode `Where()` erwartet als Parameter einen `Figur`-Lieferanten, gibt sich aber auch mit einem `Kreis`-Lieferanten zufrieden:

Quellcode	Ausgabe

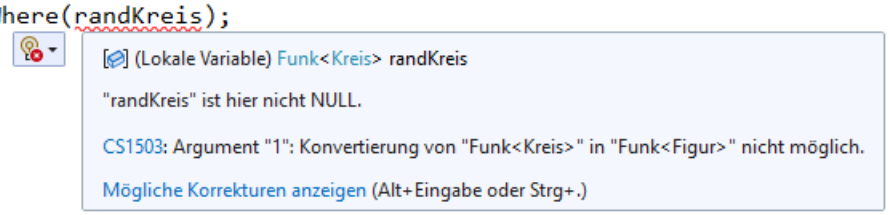
`Funk<Kreis> randKreis = delegate () {`
`  Random zzg = new();`
`  return new Kreis(zzg.Next(5), zzg.Next(5), zzg.Next(3));`
`};`
 	
`Where(randKreis);`
 	
`static void Where(Funk<Figur> figur) {`
`  Console.WriteLine($"({figur().X}, {figur().Y})");`
`}`
 	
`public delegate T Funk<out T>();`
	`(3, 0)`

Ist der Typformalparameter von `Funk<T>` *nicht* als kovariant deklariert,

```
public delegate T Funk<T>();
```

dann verbietet der Compiler hingegen die Verwendung eines `Funk<Kreis>` - Delegaten an Stelle eines `Funk<Figur>` - Delegaten:

```
Funk<Kreis> randKreis = delegate () {
   Random zzg = new();
   return new Kreis(zzg.Next(5), zzg.Next(5), zzg.Next(3));
};

Where(randKreis);
```



Eine als Ko- bzw. Kontravarianz zu beschreibende Zuweisungsliberalität besteht auch für *nichtgenerische* Delegatentypen und sogenannte *Methodengruppen*.[1] Einer Variablen vom Typ

```
delegate object NonGenericDelegate(string arg);
```

darf z. B. die folgende Methode mit einem allgemeineren Argumenttyp und einem spezielleren Rückgabetyp

```
static string TellOrHell(object arg) => arg is string s ? s : "Hello";
```

zugewiesen werden:

```
NonGenericDelegate nGenDel = TellOrHell;
```

Das folgende Programm betreibt viel Aufwand und Abstraktion, um *Hello* auf die Konsole zu schreiben:

Quellcode	Ausgabe

`NonGenericDelegate nGenDel = TellOrHell;`
`Console.WriteLine(TellOrHell(new object()));`
 	
`static string TellOrHell(object arg) => arg is string s ? s : "Hello";`
 	
`delegate object NonGenericDelegate(string arg);`
	`Hello`

10.2 Ereignisse

Möchte eine Klasse anderen Klassen die Möglichkeit geben, zu besonderen Gelegenheiten eine Botschaft, d. h. einen Methodenaufruf, zu erhalten, dann bietet sie ein *Ereignis* (engl.: *event*) an.[2] Bei einem Ereignis ist eine private Delegatenvariable (vgl. Abschnitt 10.1) und ein Paar von öffentlichen Zugriffmethoden zum (De-)Registrieren von Methoden im Spiel. Interessenten können für das Ereignis eine Behandlungsmethode registrieren lassen, die zu bestimmten Gelegenheiten aufgerufen werden soll, z. B.:

- Ein Befehlsschalter aus der Klasse **Button** in der Bedienoberfläche eines WPF-Programms informiert über sein **Click**-Ereignis registrierte Interessenten darüber, dass er vom Benutzer betätigt worden ist.
- Ein Objekt vom Typ **ObservableCollection<T>** informiert über sein **Changed**-Ereignis über eine Änderung seiner Daten. Eine registrierte Behandlungsmethode kann z. B. mit einer Aktualisierung der Bedienoberfläche des Programms reagieren.

Wir lernen mit dem Ereignis ein neues Klassenmitglied kennen, das wie die anderen Mitglieder entweder den Objekten oder der Klasse zugordnet ist. Ereignisse ermöglichen es einer Klasse oder einem Objekt, andere Akteure darüber zu informieren, dass etwas Bestimmtes passiert ist.

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/

[2] 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

10.2.1 Technische Realisation von Ereignissen

Obwohl ein Ereignis in der Regel den Zugriffsschutz **public** erhält (siehe Abschnitt 10.2.3), resultiert in der veröffentlichenden Klasse stets eine *private* Delegatenvariable. Hinzu kommen explizit oder implizit definierte *öffentliche* Zugriffsmethoden zum Erweitern oder 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 gehörenden Delegatenobjekts auf
- `-=` entfernt eine Behandlungsmethode aus der Aufrufliste des zum Ereignis gehörenden Delegatenobjekts

Im folgenden Beispiel wird eine Behandlungsmethode bei dem vom Objekt `surpriseButton` angebotenen Ereignis `Seven` registriert:

```
surpriseButton.Seven += surpriseButton_Seven;
```

Das öffentlich zugängliche Ereignis und die private Delegatenvariable stehen zueinander in derselben Beziehung wie eine Eigenschaft und die zugrunde liegende Instanzvariable (vgl. Abschnitt 5.5). Dementsprechend ähnelt die Syntax zur Definition eines Ereignisses 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** im Namensraum **System.Windows** definierte Ereignis **Exit**, das anderen Klassen die Möglichkeit bietet, sich über das bevorstehende (und unabwendbare) Programmende informieren zu lassen:[1]

```
/// <summary>
/// The Exit event is fired when an application is shutting down.
/// This event is raised by the OnExit method.
/// </summary>
public event ExitEventHandler Exit {
   add {
      VerifyAccess();
      Events.AddHandler(EVENT_EXIT, value);
   }
   remove {
      VerifyAccess();
      Events.RemoveHandler(EVENT_EXIT, value);
   }
}
```

Im **add**- und im **remove**- Block der **Exit**-Ereignisdefinition wird die private Eigenschaft **Events** der Klasse **Application** verwendet, 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.

Analog zu den automatisch implementierten Eigenschaften (siehe Abschnitt 5.5.2) kann man auch bei der Ereignisdefinition etliche Routinearbeiten dem Compiler überlassen, wie das Ereignis **Activated** der Klasse **Application** zeigt:

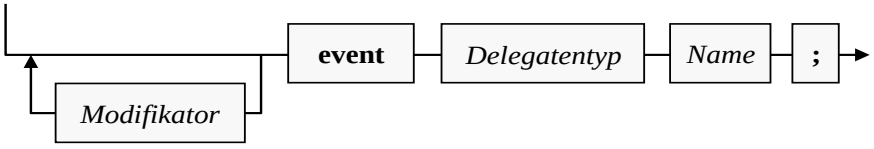
```
public event EventHandler Activated;
```

Diese Ereignisdefinition sieht aus wie die Deklaration einer Delegatenvariablen, wobei zusätzlich das Schlüsselwort **event** anzugeben ist.

---

[1] Der Quellcode stammt aus der BCL zu .NET 7; über den Zugriff auf den Quellcode informiert der Abschnitt 2.6.4.

Automatisch implementiertes Ereignis



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);
   }
}
```

Die für das Feuern des **Exit**-Ereignisses zuständige **Application**-Methode **OnExit()** holt sich das zum Ereignis gehörende Delegatenobjekt aus der Kollektion **Events** und ruft dann im Sinn von Abschnitt 10.1.2 das Delegatenobjekt auf:

```
protected virtual void OnExit(ExitEventArgs e) {
   VerifyAccess();
   ExitEventHandler handler = (ExitEventHandler)Events[EVENT_EXIT];
   if (handler != null) {
      handler(this, e);
   }
}
```

Zwischen den Methoden **OnActivated()** und den **OnExit()** bestehen zwei diskussionsbedürftige Unterschiede:

- Während **OnActivated()** die registrierten Methoden über das Ereignis startet, benutzt **OnExit()** dazu das Delegatenobjekt. Offenbar sind die beiden Verfahren äquivalent, und die Methode **OnExit()** demonstriert, dass sich ein Ereignis in den zugehörigen Delegatentyp wandeln lässt. Man kann auf die explizite Wandlung verzichten und auf die implizite Wandlung durch den Compiler vertrauen. Das Visual Studio bezeichnet die explizite Wandlung daher als redundant.
- Kniffliger ist die **OnExit()** enthaltene Vorsichtsmaßnahme, von dem Delegatenobjekt eine Kopie anzufertigen und zum Starten zu verwenden. Damit soll die Unveränderlichkeit von Delegatenobjekten ausnutzend verhindert werden, dass ein anderer Thread (Ausführungsfaden) das zum Aufruf zu verwendende Delegatenobjekt nach dem **null**-Test verwirft (durch Deregistrierung von Methoden), sodass es beim Aufruf zu einer **NullReferenceException** kommt. Leider erzwingt die Diskussion der Problematik einen Vorgriff auf das Kapitel 17 in Baltes-Götz (2021).

  Zur Thread-Sicherheit von Ereignissen existieren viele kontrovers diskutierte Lösungsvorschläge. In einem überzeugenden Blog-Beitrag begründet Stephen Cleary, dass die aktuelle Technik keine Lösung erlaubt, die nicht …
  - entweder eine Race Condition (Verwerfen des Delegatenobjekts nach dem **null**-Test durch einen anderen Thread mit einer **NullReferenceException** als Ergebnis)
  - oder einen Deadlock (gegenseitige Blockade von Threads)

  befürchten lässt.[1] Daher empfiehlt Stephen Cleary, das (De)registrieren von Ereignisbehandlungsmethoden demjenigen Thread vorzubehalten, der das Ereignis auslöst:

[1] https://blog.stephencleary.com/2009/06/threadsafe-events.html

> If an event exists on an object, then only one thread should be able to subscribe to or unsubscribe from that event, and it's the same thread that will raise the event.

Bei Beachtung dieser Regel ist die Verwendung einer temporären Delegatenvariablen überflüssig und die Lösung in **OnActivated()** angemessen.

Weil die zu einem Ereignis gehörende 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 **OnActivated()** mit dem Zugriffsschutz **protected** können abgeleitete Klassen jedoch die volle Kontrolle über die Ereignisentstehung übernehmen (siehe Abschnitt 10.2.3).

Weil bei einem Ereignis (analog zu einer Eigenschaft) ein Paar von Methoden im Spiel ist (siehe obiges Beispiel **Exit**), sind in einer Ereignisdefinition die folgenden Modifikatoren erlaubt:

- Zugriffsmodifikatoren
  Es sind dieselben Zugriffsmodifikatoren erlaubt wie bei anderen Klassenmitgliedern (siehe Abschnitt 5.12).
- **static**
- **abstract**
  Das Ereignis wird ohne Zugriffsmethoden erstellt, sodass es von abgeleiteten (nicht-abstrakten) Klassen implementiert werden muss.
- **virtual**
  Das Ereignis darf in einer abgeleiteten Klasse überschrieben werden.
- **new**, **override**
  Eine abgeleitete Klasse kann ein Ereignis der Basisklasse ersetzen oder überschreiben. Für ein überschreibendes Ereignis darf der Zugriffsschutz im Vergleich zum überschriebenen Ereignis nicht geändert werden.
- **sealed**
  Für ein überschreibendes Ereignis kann per **sealed**-Modifikator verhindert werden, dass es in abgeleiteten Klassen überschrieben wird. Warum das Schlüsselwort **sealed** nur in Kombination mit dem Schlüsselwort **override** erforderlich und erlaubt ist, wurde im Abschnitt 7.10 im Zusammenhang mit den Methoden erläutert.

10.2.2 Behandlungsmethoden registrieren

Um auf ein Ereignis reagieren zu können, sind die folgenden Schritte erforderlich:

- Man definiert eine Methode, die mit dem Delegatentyp des Ereignisses kompatibel ist (siehe Abschnitt 10.1.6 zur Zuweisungskompatibilität bei Delegaten).
- Man erzeugt ein Delegatenobjekt, das auf diese Methode zeigt.
- Man weist dem Ereignis dieses Delegatenobjekt per += - Operator zu. Dabei wird die Aufrufliste des zum Ereignis gehörenden Delegatenobjekts erweitert (siehe Abschnitt 10.1.3).

Wie Sie seit dem Abschnitt 10.1.2 wissen, kann zum Erstellen eines Delegatenobjekts unter Verwendung einer vorhandenen Methode die explizite Schreibweise mit **new**-Operator wie im folgenden Beispiel

```
surpriseButton.Seven += new SevenEventHandler(this.surpriseButton_Seven);
```

oder die Kurzform

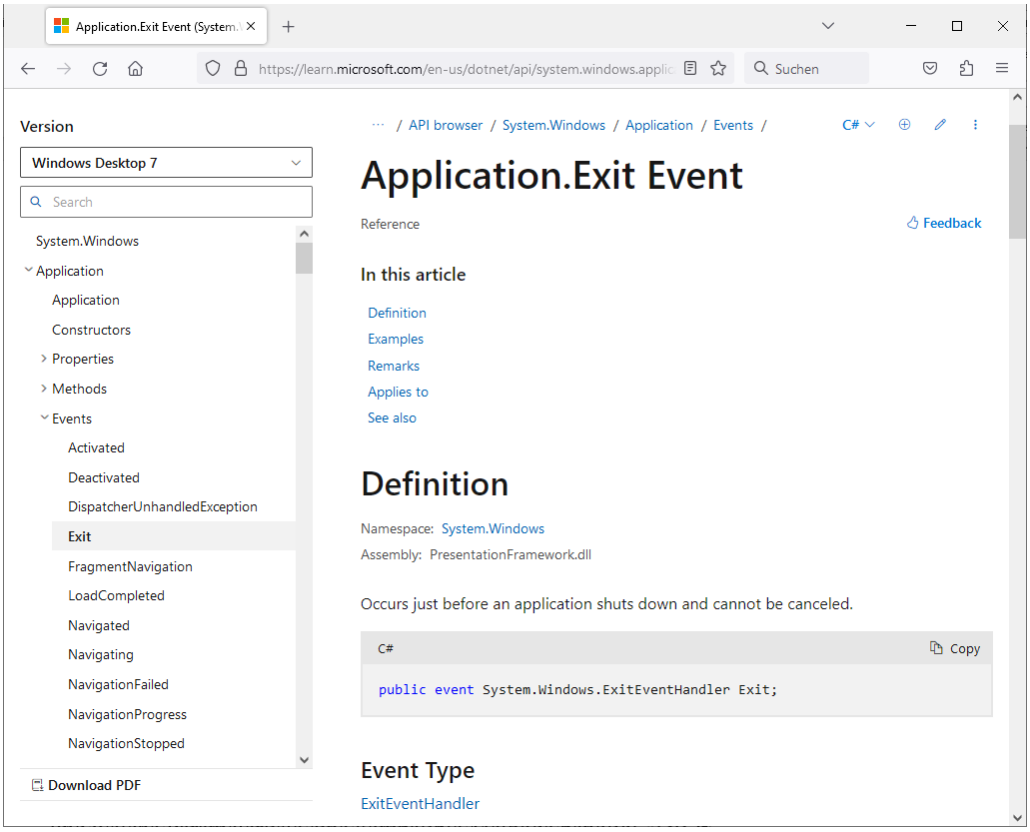
```
surpriseButton.Seven += surpriseButton_Seven;
```

verwendet werden.

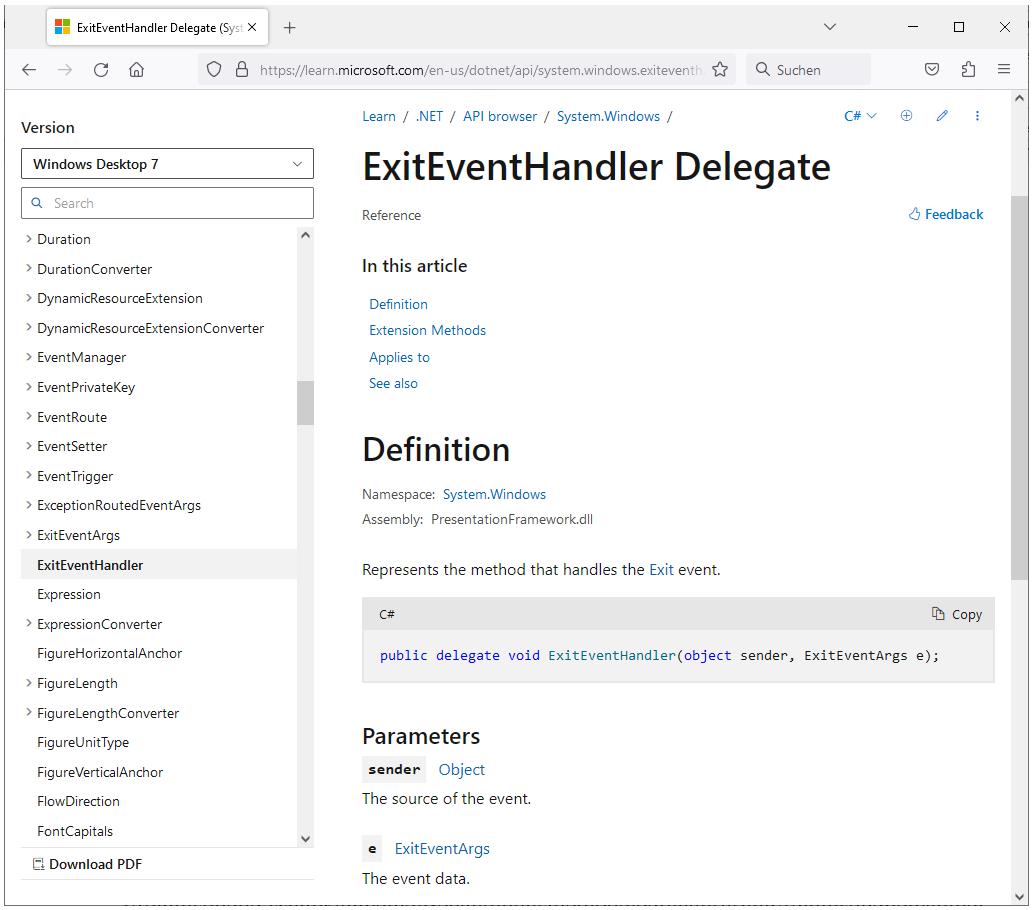
Wir betrachten wie schon im Abschnitt 10.2.1 als Beispiel das Ereignis **Exit** der WPF-Klasse **Application** im Namensraum **System.Windows**, das anlässlich der unmittelbar bevorstehenden (und

nicht mehr zu stoppenden) Beendigung einer WPF-Anwendung auftritt. Es wird von dem die Anwendung repräsentierenden Objekt ausgelöst (siehe Abschnitt 12.2.3).

Zu welchem Delegatentyp eine Ereignisbehandlungsmethode kompatibel sein muss, erfährt man in der BCL-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**-Delegatensignatur 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 übergeben.

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 bei automatisch erstellten Behandlungsmethoden zwischen die beiden Namensbestandteile einen Unterstrich.[1]

Wie Sie bereits aus eigener Erfahrung wissen (siehe z. B. Abschnitt 5.13.8), ist bei der praktischen Arbeit mit dem Visual Studio das Erstellen und Registrieren einer Ereignisbehandlungsmethode eine einfache Angelegenheit. Um ein rudimentäres Beispielprogramm samt **Exit**-Ereignisbehandlung semiautomatisch zu erstellen, kann man so vorgehen:

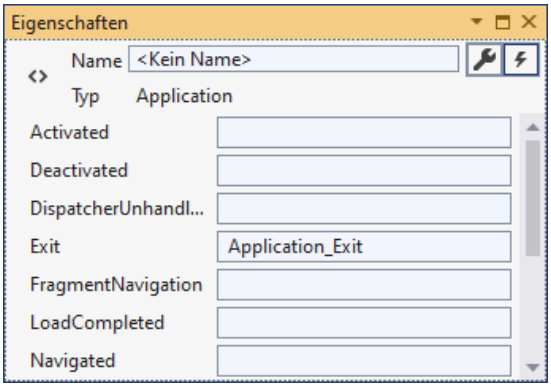
---

[1] Bei einem per Instanzvariable ansprechbaren Objekt als Ereignisquelle starten die Namen der von Assistenten erstellten Ereignisbehandlungsmethoden mit einem Kleinbuchstaben, z. B. `reduceBtn_Click()`. Den Verstoß gegen die Benennungskonventionen für Methodennamen nehmen wir in Kauf, um nicht die Namen von automatisch erstellten Methoden ändern zu müssen (vgl. Abschnitte 4.1.6 und 5.3.1).

- 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 setzt die Einfügemarke in das Element **Application**,

```
<Application x:Class="ApplicationExit.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:ApplicationExit"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

  wechselt im **Eigenschaften**-Fenster per Mausklick auf das Symbol ⚡ zur Liste mit den Ereignissen der Klasse `App` und setzt einen Doppelklick auf das Texteingabefeld 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");
}
```

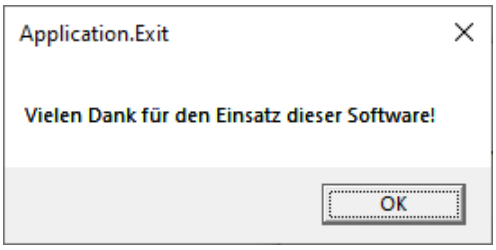
Außerdem wird in der Datei **App.g.cs** mit der vom *Visual Studio* gepflegten partiellen `App`-Klassendefinition (vgl. Abschnitt 12.3.4 zur Erläuterung und Lokalisation der Datei) die Ereignisregistrierung vorgenommen. Diese Datei entsteht aus der Vorabversion **App.g.i.cs** beim Erstellen des Programms (z. B. mit der Tastenkombination **Strg+Umschalt+B**). In der Methode **Main()** wird ein Objekt der Klasse `App` (im Namensraum `ApplicationExit`) mit dem Namen `app` erzeugt und mit der Ausführung der Methode **InitializeComponent()** beauftragt:

```
[System.STAThreadAttribute()]
public static void Main() {
    ApplicationExit.App app = new ApplicationExit.App();
    app.InitializeComponent();
    app.Run();
}
```

In **InitializeComponent()** wird die Instanzmethode `Application_Exit` in die Aufrufliste zum Ereignis `Exit` aufgenommen:

```
this.Exit += new System.Windows.ExitEventHandler(this.Application_Exit);
```

Im Beispiel kommt die Methode `Application_Exit` z. B. dann zum Einsatz, wenn der Benutzer auf das Schließkreuz in der Titelzeile des Anwendungsfensters klickt:



Bei GUI-Anwendungen mit mehreren Fenstern kann es im Zusammenhang mit der Registrierung von Ereignisempfängern zu einer Speicherverschwendung kommen (Griffiths 2013, S. 330ff):

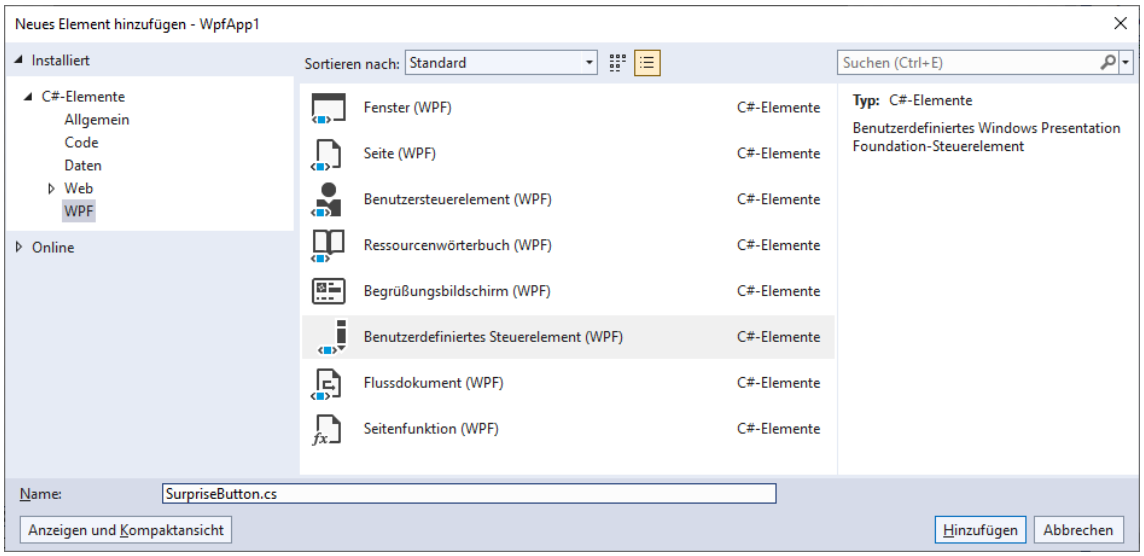
- Ein Fenster enthält ein Member-Objekt mit einer Behandlungsmethode, die bei einem Ereignis registriert ist.
- Vor dem Schließen dieses Fensters wird es versäumt, die Ereignisregistrierung aufzuheben.
- Solange die Ereignisquelle existiert, bleiben die zum geschlossenen Fenster gehörenden Objekte erreichbar, können also nicht vom Garbage Collector abgeräumt werden.

10.2.3 Ereignisse anbieten

Zwar kommt das Registrieren eigener Behandlungsmethoden bei Ereignissen von BCL-Klassen erheblich 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. Mit Vorgriffen auf die WPF-Technik haben Sie ja mittlerweile schon viel Erfahrung.

Wir erstellen eine neue **WPF-Anwendung** mit dem Namen `EventPublisher`. Nach dem **Erstellen** des Projekts ergänzen wir eine aus der BCL-Klasse **Button** abgeleitete benutzerdefinierte Steuerelementklasse, deren Objekte bei jedem Mausklick auf den Befehlsschalter eine Zufallszahl von 0 bis 9999 ziehen und das noch zu definierende Ereignis `Seven` auslösen, wenn die Zufallszahl restfrei durch 7 teilbar ist. Man könnte diesen Befehlsschalter z. B. für Werbezwecke weiterentwickeln und mit einstellbarer Ereignisfrequenz eine Werbung einblenden, die nicht in die reguläre Behandlungsmethode des Schalters integriert werden müsste.

Die nicht triviale Aufgabe, ein benutzerdefiniertes Steuerelement zu erstellen, wird vom Visual Studio gut unterstützt. 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 **C# - Elemente > WPF** die Option **Benutzerdefiniertes Steuerelement (WPF)**, und tragen Sie **SurpriseButton.cs** als **Namen** ein:



Nach dem **Hinzufügen** wird die Datei **SurpriseButton.cs** im Quellcode-Editor der Entwicklungsumgebung geöffnet.

Tragen Sie **Button** (statt **Control**) als Basisklasse für `SurpriseButton` ein.

Wenn der vom Assistenten erstellte statische Konstruktor mit stilistischem Effekt

```
static SurpriseButton() {
    DefaultStyleKeyProperty.OverrideMetadata(typeof(SurpriseButton),
        new FrameworkPropertyMetadata(typeof(SurpriseButton)));
}
```

im Quellcode verbleibt, dann ist im Anwendungsfenster von unserer **Button**-Ableitung nichts zu sehen. 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. Wir übernehmen für unsere **Button**-Ableitung das Erscheinungsbild der Basisklasse und streichen den statischen Konstruktor, sodass der folgende Zwischenstand für die `SurpriseButton`-Klassendefinition resultiert:

```
public class SurpriseButton : Button {
}
```

Ergänzen Sie (z. B. in der Quellcodedatei **SurpriseButton.cs**) eine von **System.EventArgs** abstammende Klasse, die zur Übertragung von Informationen an Interessenten für das Ereignis `Seven` 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);
```

Dieser Delegatentyp kann ebenfalls in der Quellcodedatei **SurpriseButton.cs** definiert werden.

Üblicherweise endet der Name eines zur Ereignisdefinition verwendeten 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 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 10.2.1) nicht ändern wollen, sieht die Ereignisdefinition in der Klasse `SurpriseButton` fast so aus wie eine Felddeklaration, wobei aber das Schlüsselwort **event** anzugeben ist:

```
public class SurpriseButton : Button {
    public event SevenEventHandler Seven;
}
```

Wir überschreiben in der Klasse `SurpriseButton` die von **Button** geerbte Methode **OnClick()**, die bei jedem Mausklick auf den Befehlsschalter aufgerufen wird:

```
protected override void OnClick() {
   base.OnClick();
   if (Seven != null) {
      int losnummer = zzg.Next(10_000);
      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 in der `SurpriseButton`-Definition noch eine Instanzvariable deklariert und initialisiert:

```
Random zzg = new();
```

Unser benutzerdefinierter Schalter reagiert auf einen Mausklick zunächst mit dem Basisklassenverhalten.[1] Ist ein `Seven`-Ereignisempfänger registriert, dann wird außerdem ...

- eine Zufallszahl vom Typ **int** gezogen,
- und bei Teilbarkeit durch 7 das Ereignis `Seven` ausgelöst.

Jede registrierte Ereignisbehandlungsmethode wird über die Ereignisquelle 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.[2] Das Delegatenobjekt zu einem Ereignis entsteht beim Registrieren der ersten Behandlungsmethode und verschwindet beim Entleeren seiner Aufrufliste.

Anschließend ist der komplette Quellcode in der Datei **SurpriseButton.cs** zu sehen:

```
public class SurpriseButton : Button {
    public event SevenEventHandler Seven;
    Random zzg = new();
```

---

[1] Warum die Methode **OnClick()** bei einem Mausklick auf den Befehlsschalter aufgerufen wird, erfahren Sie später.

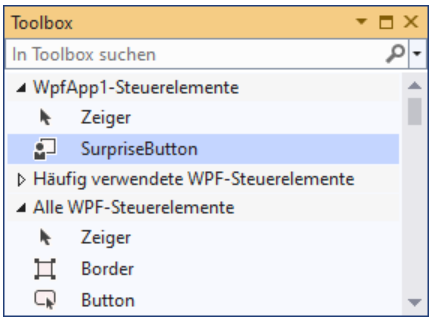
[2] Bei einer Multithreading - Anwendung sollte man sogar durch eine geeignete Synchronisation verhindern, dass ein Delegatenobjekt zwischen der Existenzprüfung und dem Aufruf durch einen anderen Thread verworfen wird (siehe Abschnitt 17.1.4 in Baltes-Götz (2021)).

```
        protected override void OnClick() {
            base.OnClick();
            if (Seven != null) {
                int losnummer = zzg.Next(10_000);
                if (losnummer % 7 == 0) {
                    SevenEventArgs sea = new() { Nr = losnummer };
                    Seven(this, sea);
                }
            }
        }

    }

    public class SevenEventArgs : EventArgs {
        public int Nr;
    }

    public delegate void SevenEventHandler(object sender, SevenEventArgs e);
```

Lassen Sie das Programm mit der Tastenkombination **Strg+Umschalt+B** oder mit dem Menübefehl:

Erstellen > Projektmappe erstellen

erstellen, damit unser benutzerdefiniertes Steuerelement `SurpriseButton` in die **Toolbox** des WPF-Designers aufgenommen wird:



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:

```
<Window x:Class="EventPublisher.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:EventPublisher"
        mc:Ignorable="d"
        Title="Event Publisher" Height="250" Width="400
```

```
        ">
    <Grid>
        <local:SurpriseButton x:Name="surpriseButton" Content="7 gewinnt"
         VerticalAlignment="Center" HorizontalAlignment="Center"
         Height="150" Width="300" FontSize="48"/>
    </Grid>
</Window>
```

Für den Schalter aus der Klasse `SurpriseButton` sind per WPF-Designer bzw. XAML-Code festzulegen:

- Werte für die optischen Eigenschaften (**Content**, **VerticalAlignment**, **HorizontalAlignment**, **Height**, **Width**, **FontSize**)
- ein Instanzvariablennamen, damit Ereignisregistrierungen vorgenommen werden können:
  `x:Name="surpriseButton"`

Weil die Klasse `MainWindow` über ein eingetretenes `Seven`-Ereignis informiert werden möchte, ...

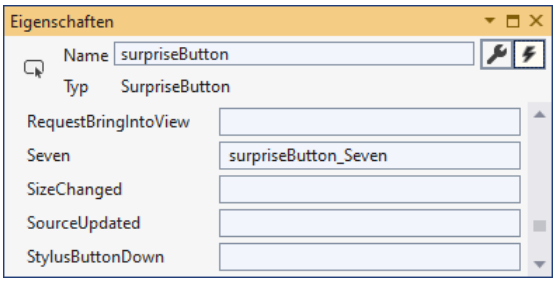
- implementiert sie eine Methode mit der Delegatensignatur `SevenEventHandler`
- und registriert diese Methode beim Ereignis `Seven` des `SupriseButton`-Steuerelements, was z. B. im `MainWindow`-Konstruktor geschehen kann.

Der folgenden `MainWindow`-Quellcode enthält die `Seven`-Ereignisbehandlungsmethode samt Registrierung:

```
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
        surpriseButton.Seven += surpriseButton_Seven;
    }

    void surpriseButton_Seven(object sender, SevenEventArgs e) {
        MessageBox.Show($"Sie haben gewonnen. Losnummer: {e.Nr}", "Event Consumer");
    }
}
```

Beim Namen der Ereignisbehandlungsmethoden passen wir uns dem Stil der Entwicklungsumgebung an und lassen auf den Namen der Ereignisquelle einen Unterstrich sowie den Ereignisnamen folgen.

Man kann die Ereignisregistrierung auch über das **Eigenschaften**-Fenster

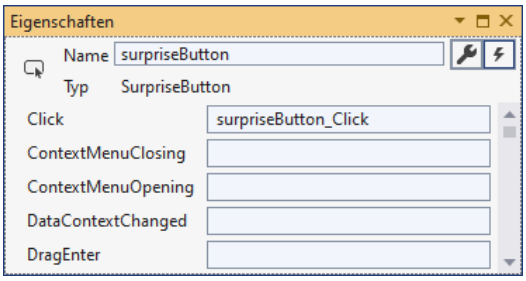


oder direkt im XAML-Code vornehmen:

```
<Grid>
    <local:SurpriseButton x:Name="surpriseButton" Content="7 gewinnt"
    VerticalAlignment="Center" HorizontalAlignment="Center"
    Height="150" Width="300" FontSize="48" Seven="surpriseButton_Seven"/>
</Grid>
```

In diesem Fall kommt es zu einer automatischen Registrierung der Methode beim Ereignis, und es sollte keine zusätzliche Registrierung (z. B. per `MainWindow`-Konstruktor) erfolgen, weil die Methode sonst *zweimal* aufgerufen wird.

Neben der Verwendung zur Werbung kann der `SurpriseButton` natürlich auch regulär genutzt werden, z. B. über eine Behandlungsmethode zum Ereignis **Click**:

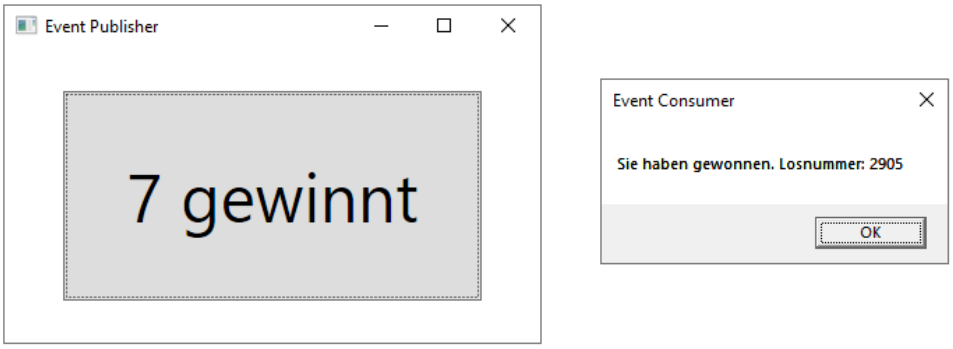


Ein Assistent legt die Behandlungsmethode an

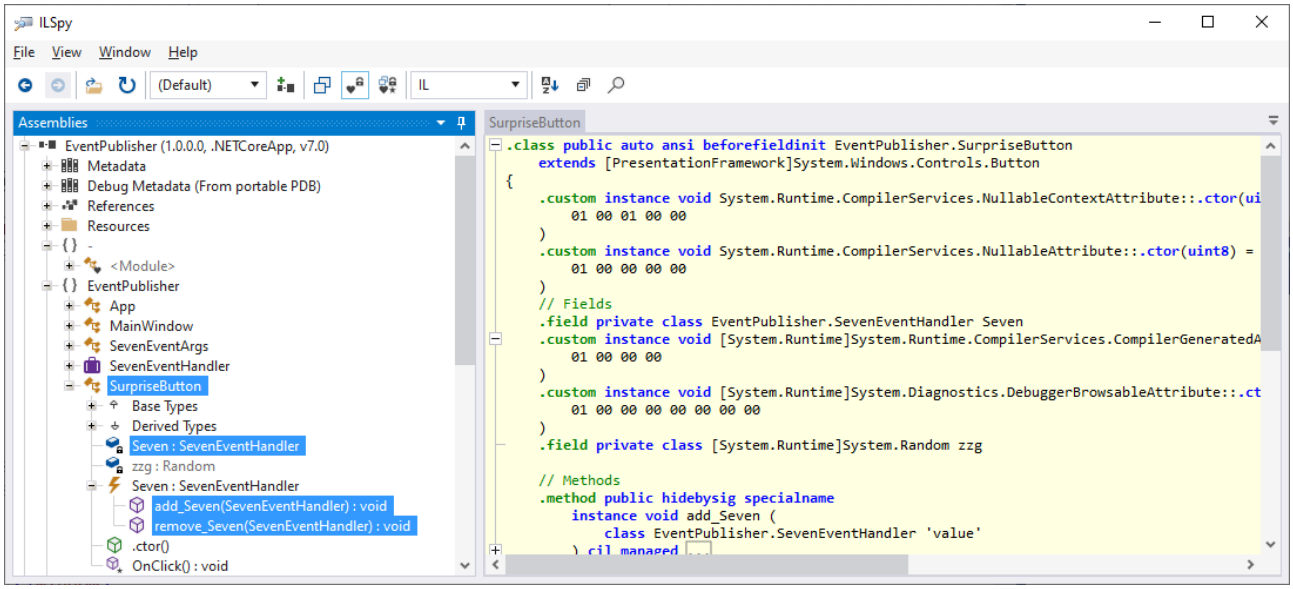
```
void surpriseButton_Click(object sender, RoutedEventArgs e) {
    MessageBox.Show("Sie haben den Schalter \"7 gewinnt\" geklickt.",
                    "Event Consumer");
}
```

und sorgt für die Registrierung beim Ereignis.

In ihrer Reaktion auf die Benachrichtigung nutzt die Methode `surpriseButton_Seven()` eine Information aus dem erhaltenen `SevenEventArgs`-Parameterobjekt mit der Ereignisbeschreibung:



Eine Assembly-Inspektion mit ILSpy bestätigt die zu Beginn von Abschnitt 10.2.1 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()`):



Weil auch bei einem Ereignis mit der Schutzstufe **public** die zum Auslösen zu verwendende Delegatenvariable die Schutzstufe **private** besitzt, können abgeleitete Klassen das Ereignis *nicht* über die Delegatenvariable auslösen. Die folgenden Maßnahmen ermöglichen einer abgeleiteten Klasse die Kontrolle über das Ereignis:

- Das Ereignis wird durch eine Methode mit der Schutzstufe **protected** ausgelöst, die von abgeleiteten Klassen aufgerufen werden kann.
- Die auslösende Methode wird als virtuell (überschreibbar) definiert. 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();

   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 SBDerivation : SurpriseButton {
   protected override void OnSeven(SevenEventArgs sea) {
      if (sea.Nr >= 5000)
         base.OnSeven(sea);
   }
}
```

Ein Visual Studio - Projekt mit dem `SurpriseButton` ist im folgenden Ordner zu finden

…\BspUeb\Ereignisse\Event\EventPublisher

Als Ereignisanbieter kommen keinesfalls nur die mit dem Benutzer interagierenden Steuerelemente in Frage. Auch die mit der Datenverwaltung (mit der Geschäftslogik) beschäftigten Klassen nutzen Ereignisse, um z. B. über Änderungen im Datenbestand zu informieren.

10.3 Übungsaufgaben zum Kapitel 10

1) Welche von den folgenden Aussagen sind richtig?

1. Ein Datenobjekt ist unveränderlich, kann also nicht verändert, sondern nur ersetzt werden.
2. Der Rückgabetyp spielt bei der Delegatensignatur keine Rolle.
3. Bei der Delegatendefinition über eine anonyme Methode oder per Lambda-Notation können lokale Variablen sowie (statische) Felder der Umgebung verwendet werden.
4. C# unterstützt bei den Typformalparametern von generischen Delegaten die Ko- und Kontravarianz.

2) Der BCL-Delegatentyp **Predicate<in 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 Parameter vom Typ **Predicate<in T>** dient in der Methode **FindAll()** der generischen Klasse **List<T>** dazu, aus der angesprochenen Liste eine Teilmenge von Elementen mit bestimmten Eigenschaften für eine neu zu erstellende 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 und testen Sie eine solche Metamethode.

4) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Ein Ereignis ist ein Datentyp.
2. 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.
3. Die Registrierung eines Ereignisempfängers wieder aufzuheben, lohnt sich nicht.
4. Ein Ereignis kann versiegelt werden.

11 Kollektionen

Ein besonders erfolgreiches Anwendungsfeld für die im Kapitel 8 beschriebene Typgenerizität sind die Klassen zur Verwaltung von Listen, Mengen, (Schlüssel-Wert) - Tabellen, Stapeln, Warteschlangen und anderen Kollektionen im Namensraum **System.Collections.Generic**, die in sehr vielen C# - Programmen zum Einsatz kommen. Auch das im Abschnitt 6.8 vorgestellte Programm zur Präsentation von RSS-Items verwendet zur Verwaltung seiner Daten ein generisches Kollektionsobjekt:

```
var items = new List<RssItem>();
```

Wer nach der Lektüre von Kapitel 8 noch Zweifel an der Relevanz der generischen Typen hatte, lernt nun zahlreiche generische Klassen mit hohem praktischem Nutzwert kennen.

Für die Objekte der im aktuellen Kapitel vorzustellenden Klassen wird im Manuskript alternativ zur offiziellen Bezeichnung *Kollektionen* aus sprachlichen Gründen gelegentlich auch die Bezeichnung *Container* verwendet.

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 Kapitel vorzustellenden Kollektionstypen entscheidend bei.

In diesem Kapitel beschäftigen wir uns kaum damit, eigene generische Kollektionstypen zu definieren (siehe einfache Beispiele im Abschnitt 8.2), sondern wir konzentrieren und darauf, die in der BCL zahlreich vorhandenen Typen zu nutzen. Diese Typen besitzen ausgefeilte Handlungskompetenzen (Methoden, Eigenschaften, Indexer) für typische Aufgabenstellungen (z. B. Vereinigung von zwei Mengen ohne Entstehung von Dubletten), damit Programmierer möglichst selten „das Rad neu erfinden müssen“. Wenn doch einmal eine Eigenbau-Kollektion sinnvoll ist, dann hält die BCL geeignete Basisklassen bereit, um den Entwicklungsaufwand zu minieren (siehe Abschnitt 11.6).

Im Manuskript geht es allmählich seltener darum, neue Bestandteile der Programmiersprache C# kennenzulernen, während die .NET - Basisbibliothek mit ihren zahlreichen Typen und Lösungen für Standardaufgaben der Software-Entwicklung immer öfter im Fokus steht.

Verwendet eine Anwendung mehrere nebenläufige *Ausführungsfäden* (Multithreading, siehe Kapitel 17 in Baltes-Götz (2021)), dann wird eventuell eine Thread-sichere Kollektion aus dem Namensraum **System.Collections.Concurrent** benötigt. Bekannte Klassen aus diesem Namensraum sind **BlockingCollection<T>**, **ConcurrentDictionary<K, V>**, **ConcurrentQueue<T>** und **ConcurrentStack<T>**.[1]

Die nicht-generischen Kollektionsklassen im älteren Namensraum **System.Collections** haben die im Abschnitt 8.1 am Beispiel der Klasse **ArrayList** beschriebenen Probleme und sollten für neue Projekte *nicht* mehr verwendet werden. Neben **ArrayList** waren aus dem Namensraum **System.Collections** auch die Klassen **Hashtable**, **Queue** und **Stack** populär, bevor die überlegenen generischen Lösungen entstanden sind.

[1] https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/

11.1 Arrays versus Kollektionen

Die im Abschnitt 6.2 vorgestellten Arrays taugen als Container für Elemente mit einem identischen, frei wählbaren Typ. Sie bieten Speichereffizienz und einen schnellen Indexzugriff auf die Elemente. Ein Array implementiert die generischen Kollektionsschnittstellen **IList<T>**, **ICollection<T>** und **IEnumerable<T>** sowie die nicht-generischen Kollektionsschnittstellen **IList**, **ICollection** und **IEnumerable** (siehe Abschnitte 11.2 und 11.3). Trotzdem bieten die Arrays für viele Standardaufgaben der Datenverwaltung keine zufriedenstellenden Lösungen. Das hat zur Entwicklung der Kollektionsklassen geführt, die aber teilweise intern mit Arrays arbeiten.

Bei den sehr häufig auftretenden listenartigen Datenstrukturen zeigen Arrays die folgenden Schwächen:

- Die Größe eines Arrays muss beim Erstellen festgelegt werden und ist später nicht mehr zu ändern. Über die statische Methode **Resize()** der Klasse **System.Array** ist scheinbar eine nachträgliche Längenkorrektur möglich (siehe Abschnitt 6.2.2). Allerdings erzeugt die Methode ein *neues* Array-Objekt und kopiert mit erheblichem Aufwand die Elemente des alten Arrays dorthin.
- Das Einfügen und Entfernen von *inneren* Elementen (insbesondere am Anfang eines Arrays) ist mit hohen Kosten verbunden, weil die neuen bzw. ehemaligen rechten Nachbarn verschoben werden müssen.

Kollektionen zur Verwaltung von Listen bieten hingegen eine Größendynamik sowie (je nach Architektur) ein performantes Einfügen und Löschen von inneren Elementen. Zwar verwenden viele Listenverwaltungs-Kollektionen im Hintergrund ein Array zum Speichern ihrer Elemente und müssen diesen Array bei einer Kapazitätsüberschreitung ersetzen, doch geschieht dies immerhin automatisch.

Außerdem hat bei Listen eine Größenveränderung nicht den im Abschnitt 6.2.2 für Arrays beschriebenen Effekt, dass zusätzliche Referenzen „abgehängt“ werden, z. B.:

Quellcode	Ausgabe

`int[] a1 = { 1, 2, 3, 4, 5 };`
`int[] a2 = a1;`
`Array.Resize(ref a1, 9);`
`for (int i = 6; i <= 9; i++)`
`  a1[i - 1] = i;`
`foreach (int i in a2)`
`  Console.Write(i + " ");`
`Console.WriteLine();`
 	
`var l1 = new List<int>() { 1, 2, 3, 4, 5 };`
`List<int> l2 = l1;`
`l1.AddRange(new List<int>() { 6, 7, 8, 9 });`
`foreach (int i in l2)`
`  Console.Write(i + " ");`
	`1 2 3 4 5`
	
	`1 2 3 4 5 6 7 8 9`

Während die „Fremdreferenz“ `a2` auf das Array `a1` nach einer Größenänderung weiterhin auf das veralteten Array zeigt und dessen Entsorgung verhindert, zeigt die Fremdreferenz `l2` auf die Liste `l1` auch nach der (automatischen) Größenänderung auf den aktuellen Zustand. Ist bei einer Kollektion mit einer Größenänderung zu rechnen, dann ist eine Liste gegenüber einem Array zu bevorzugen.

Sind für Elementsammlungen häufige Existenzprüfungen erforderlich, dann bietet ein Array wenig Unterstützung. Sind seine Elemente nicht sortiert, dann muss für jedes Element geprüft werden, ob es mit dem Kandidaten übereinstimmt. Kollektionen zur Verwaltung von Mengen (siehe Abschnitt 11.4) bieten hingegen schnelle Detektionsmöglichkeiten und verhindern automatisch 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 (oder zwei assoziierten Arrays) zu verwalten, treten die eben schon beschriebenen Probleme auf (feste Anzahl von Elementen, umständliches Einfügen und Löschen, aufwändige Suche nach Schlüsseln).

Als weiterer Nachteil von Arrays ist ihr kovariantes Verhalten hinsichtlich des Elementtyps zu nennen (siehe Abschnitt 8.2.3).

Im Zusammenhang mit der Verwaltung von veränderlichen Strukturinstanzen wird aber auch von einem Vorteil der Arrays gegenüber generischen Kollektionen zu berichten sein: Ein Indexausdruck referenziert das betroffene Element mit Werttyp, statt eine Kopie zu liefern (siehe Abschnitt 11.3.1).

Im zu modellierenden Aufgabenbereich treten oft Datenstrukturen vom Typ Liste, Menge, (Schlüssel-Wert) - Tabelle, Stapel oder Warteschlange auf, und auch die zur Lösung der Aufgaben konstruierten IT-Algorithmen verwenden oft Kollektionen mit unterschiedlichen Architekturen. Die BCL bietet zahlreiche direkt einsetzbare Kollektionstypen für Standardaufgaben und auch Basisklassen als Starthilfe für die Entwicklung von spezialisierten Kollektionen (siehe Abschnitt 11.6). Im Vergleich zur Verwendung von Arrays resultieren meist eine bessere Modellierung und ein besser lesbarer Quellcode.

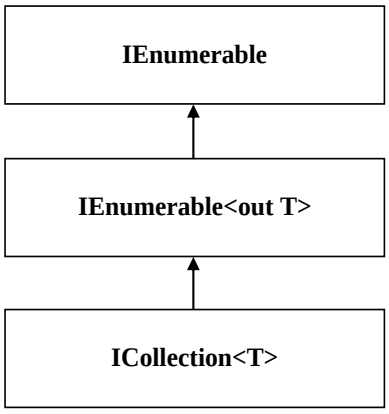
11.2 Interface ICollection<T>

Schnittstellen haben auch beim Einsatz von Kollektionstypen viele Vorteile, z. B.:

- Als Datentypen für Variablen und Parameter sollten Schnittstellen bevorzugt werden, um flexibel zwischen verschiedenen Implementationen wechseln zu können.
- Das Restringieren der Typformalparameter von generischen Typen und Methoden durch Schnittstellen stellt die erforderliche Ausstattung von konkretisierenden Typen sicher und vermeidet dabei überflüssige Einschränkungen.

Schnittstellen und Typgenerizität fördern beide die Flexibilität und Typsicherheit beim Programmieren, und in der Regel sind generische Schnittstellen zu bevorzugen.

Viele von den anschließend vorgestellten Kollektionsklassen erfüllen die Schnittstelle **ICollection<T>** sowie die Schnittstelle **IEnumerable<T>**, von der **ICollection<T>** abstammt (alle Typen aus dem Namensraum **System.Collections.Generics**). Die generische Schnittstelle **IEnumerable<T>** erweitert die nicht-generische Schnittstelle **System.Collections.IEnumerable**:



Die Schnittstelle **IEnumerable<T>** verpflichtet implementierende Typen, als Rückgabe der Methode **GetEnumerator()** ein Objekt vom Typ **IEnumerator<T>** abzuliefern, das ein Iterieren durch die Kollektionselemente erlaubt und in der Regel im Rahmen einer **foreach**-Schleife verwendet wird (siehe Abschnitt 9.5).

Eine das Interface **ICollection<T>** implementierende Klasse beherrscht die folgenden Methoden:

- **public void Add(T** *element***)**
  Das Parameterelement wird in die Kollektion aufgenommen. Bei einer Liste wird es am Ende angehängt.
- **public bool Contains(T** *element***)**
  Diese Methode informiert darüber, ob das Parameterelement in der Kollektion vorhanden ist.
- **public bool Remove(T** *element***)**
  Das erste Vorkommen des Elements wird aus der Kollektion entfernt, falls es dort vorhanden ist. Über den Rückgabewert erfährt man, ob die Kollektion durch den Aufruf verändert wurde.
- **public void CopyTo(T[]** *array***, int** *arrayIndex***)**
  Alle Kollektionselemente werden in einen kompatiblen Array ab der angegebenen Indexposition kopiert. Vorhandene Array-Elemente werden dabei überschrieben.
- **public void Clear()**
  Alle Elemente werden entfernt.

Neben Methoden stehen auch Eigenschaften im **ICollection<T>** - Pflichtenheft:

- **public int Count { get; }**
  Es wird die Anzahl der Elemente geliefert.
- **public bool IsReadOnly { get; }**
  Man erfährt, ob die Kollektion schreibgeschützt ist.

Wenn ein Typ ...

- die Schnittstelle **IEnumerable** implementiert,
- und eine Instanz- oder Erweiterungsmethode namens **Add()** zur Aufnahme eines Elements besitzt,[1]

dann kann die Kollektion bei der Erstellung auf syntaktisch einfache Weise per **Kollektionsinitialisierer** bevölkert werden, z. B.:[2]

```
var nl = new List<String> {"Otto", "Marita", "Theo"};
```

Abweichend von der Situation bei **IEnumerable<T>** und **IEnumerable** wurden die generischen BCL-Schnittstellen **ICollection<T>**, **IList<T>** und **IDictionary<K, V>** *nicht* als Erweiterungen der nicht-generischen Varianten **ICollection**, **IList** bzw. **IDictionary** definiert, weil die später entwickelten generischen Schnittstellen eine besser durchdachte Zusammenstellung von Mitgliedern besitzen (Albahari 2022, S. 351).

---

[1] Zu Erweiterungsmethoden siehe Abschnitt 7.15.

[2] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/object-and-collection-initializers

11.3 Verwaltung einer Liste

Eine Liste enthält Elemente in einer Reihenfolge mit definierter Bedeutung. Alle Elemente sind vom selben Typ (Klasse, Struktur oder Schnittstelle). Erfüllt eine Listenverwaltungsklasse die Schnittstelle **IList<T>**, dann kann man auf die Elemente wahlfrei über einen nullbasierten Index zugreifen.[1] Es ist also wie bei einem Array möglich, das Element an einer bestimmten Position abzurufen oder festzulegen.

Im Unterschied zu einem Array wird eine Liste bei Bedarf automatisch vergrößert. Bei einer generischen Liste haben wir also eine größendynamische Kollektion zur Verfügung, die Elemente von einem wählbaren Typ sortenrein (mit Compiler-Typsicherheit) verwaltet.

Für Listen finden sich sehr viele Einsatzmöglichkeiten bei der Software-Entwicklung. Man verwendet sie z. B. für ...

- die aktuell von einem Steuerelement der Bedienoberfläche angebotenen Optionen,
- die Bestellungen eines Kunden, der aus einer Datenbank geladen wurde,
- die aus einem Text sukzessiv extrahierten Wörter.

Die Elemente einer Liste müssen (im Unterschied zu den Elementen einer Menge, vgl. Abschnitt 11.4) nicht verschieden sein.

Neben den anschließend vorgestellten Typen zur Listenverwaltung enthält die BCL noch Lösungen für wichtige Spezialfälle, die im Manuskript nicht behandelt werden, z. B.:

- Warteschlange
  Die Klasse **Queue<T>** verwaltet eine Liste nach dem FIFO-Prinzip (*First In First Out*).
- Stapel
  Die Klasse **Stack<T>** verwaltet eine Liste nach dem LIFO-Prinzip (*Last In First Out*). Im Abschnitt 8.2.1 haben wir eine simple Variante selbst erstellt.

11.3.1 Klasse List<T> mit Array-Unterbau

Die generische Klasse **List<T>** zur bequemen und typsicheren Verwaltung einer Sequenz von Elementen eines festen Typs haben wir schon im Abschnitt 8.1 kennengelernt. Ein **List<T>** - Objekt verwendet zum Speichern seiner Elemente intern das Array **T[]** mit den folgenden Konsequenzen:

- Lesezugriffe sind performant.
- Veränderungen sind hingegen zeitaufwändig, wenn ...
  - neue Instanzen im Inneren der Liste (insbesondere am Anfang) eingefügt oder entfernt werden, weil dann andere Elemente zu verschieben sind.
  - aufgrund einer Kapazitätsüberschreitung ein neues Array angelegt werden muss, weil dann alle Elemente zu kopieren sind, und Arbeit für den Garbage Collector anfällt.

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);
```

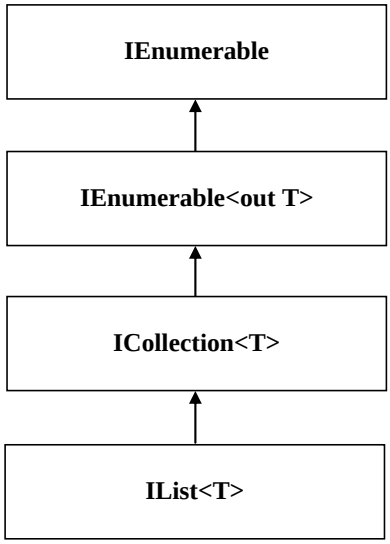
Über die Eigenschaft **Capacity** erfährt man die aktuelle Kapazität einer Liste, welche meist die per **Count**-Eigenschaft abfragbare aktuelle Länge übertrifft, z. B.:

---

[1] Weil nicht alle BCL-Typen mit Listenarchitektur diese Schnittstelle implementieren, wird sie erst im Abschnitt 11.3.1 vorgestellt.

Quellcode	Ausgabe
`var nl = new List<String>(5) {"Otto", "Marita", "Theo"};` `Console.WriteLine($"Länge:  {nl.Count}\nKapazität: {nl.Capacity}");`	`Länge:  3` `Kapazität: 5`

Die Klasse **List<T>** implementiert die Schnittstelle **IList<T>** und damit indirekt auch die Schnittstelle **ICollection<T>**, von der **IList<T>** abstammt, sowie die Schnittstellen im Stammbaum von **ICollection<T>**:



In der Schnittstelle **IList<T>** sind die folgenden abstrakten Methoden vorhanden, die folglich von der Klasse **List<T>** zusätzlich zu den Methoden der Schnittstelle **ICollection<T>** (siehe Abschnitt 11.2) zu implementieren sind:

- **public int IndexOf(T** *element***)**
  Falls das Element in der Liste vorhanden ist, liefert die Methode den Index des ersten Auftretens, anderenfalls den Wert -1.
- **public void Insert(int** *index***, T** *element***)**
  Das Parameterelement wird an der gewünschten Indexposition eingefügt.
- **public void RemoveAt(int** *index***)**
  Das Element an der angegebenen Indexposition wird entfernt.

Außerdem schreibt die Schnittstelle **IList<T>** einen Indexer vor, der es ermöglicht, das Element an der Position *index* abzurufen oder festzulegen:

public T this[int *index***] { get; set; }**

Die Klasse **List<T>** enthält über die **IList<T>** - Verpflichtungen hinaus die Instanzmethode **AsReadOnly()**, um einen schreibgeschützten Zugriff zu ermöglichen:

public ReadOnlyCollection<T> AsReadOnly()

Der Aufrufer erhält ein Objekt der Klasse **ReadOnlyCollection<T>**, das als *Sicht* auf die angesprochene Liste ...[1]

[1] https://learn.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.readonlycollection-1

- keine Veränderung zulässt,
- aber jede Änderung der zugrunde liegenden Liste sofort berücksichtigt.

Außerdem beherrscht die Klasse **List<T>** zum Suchen und Sortieren dieselben Methoden wie ein Array (vgl. Abschnitt 6.2.6), wobei die statischen Methoden der Klasse **Array** durch Instanzmethoden der Klasse **List<T>** ersetzt sind, z. B.:

Quellcode	Ausgabe

`var iar = new List<int> { 2, 13, 12, 15, 7 };`
`Console.WriteLine("Index: " +`
`  iar.FindIndex(i => i % 3 == 0));`
	`Index: 2`

Die Methode **FindIndex()** sucht ein Element mit einer bestimmten Eigenschaft. Im Beispiel wird der Index des ersten durch 3 teilbaren Elements ermittelt. Als Parameter von **FindIndex()** wird zur Beurteilung der Elemente ein per Ausdrucks-Lambda (siehe Abschnitt 10.1.5.2) realisiertes Delegatenobjekt vom Typ **Predicate<T>** verwendet (Rückgabetyp **bool**). Wird die Methode **FindIndex()** nicht fündig, dann liefert sie die Rückgabe -1.

Weil die Klasse **List<T>** den von der Schnittstelle **IList<T>** vorgeschriebenen Indexer besitzt, ist wie bei einem Array ein Elementzugriff über den **[]** - Operator möglich (zu Indexern siehe Abschnitt 5.11).

Im Zusammenhang mit dem Indexer für **List<T>** - Objekte soll noch einmal demonstriert werden, warum von veränderlichen *Strukturen* abzuraten ist (vgl. Abschnitt 6.1.1).[1] Im folgenden Programm wird ein per Indexer angesprochenes Objekt der Klasse `Punkt` gebeten, sich zu `Bewegen()`. Anschließend wird per **WriteLine()** - Aufruf nachgewiesen, dass sich das Objekt an der neuen Position befindet:

Quellcode	Ausgabe

`public class Punkt {`
`  double x, y;`
`  public Punkt(double xpar, double ypar) {`
`  x = xpar;`
`  y = ypar;`
`  }`
`  public void Bewegen(double hor, double vert) {`
`  x += hor;`
`  y += vert;`
`  }`
`  public override 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] Mit Listenelementen vom Typ einer Klasse treten beim Indexzugriff keine Überraschungen auf.

```
(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 *nicht* auf, wenn statt eines **List**<Punkt> - Objekts das 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).

Quellcode	Ausgabe

`public struct Punkt {`
`  . . .`
`}`
`class Prog {`
`  static void Main() {`
`  var pa = new Punkt[2] { new Punkt(1, 2), new Punkt(3, 4) };`
`  pa[0].Bewegen(5, 5);`
`  Console.WriteLine(pa[0]);`
`  }`
`}`
	`(6; 7)`

11.3.2 Klasse LinkedList<T> mit doppelt verketteten Knoten

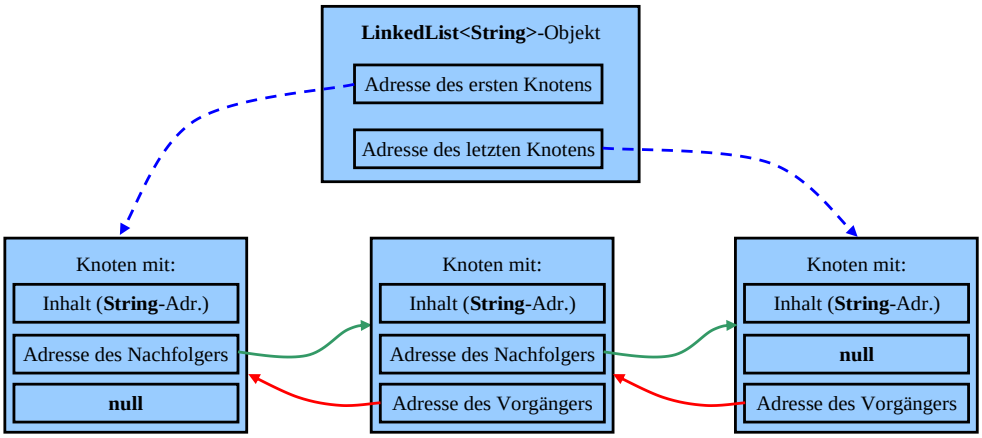
Die Klasse **List<T>** verwendet intern ein Array zum Speichern der Elemente und bietet daher einen schnellen wahlfreien Zugriff. Auch das Anhängen neuer Elemente am Ende der Liste verläuft flott, wenn nicht gerade die Kapazität des Arrays erschöpft ist. Dann wird es erforderlich, ein größeres Array anzulegen 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 doppelt verketteten Knoten der Klasse **LinkedListNode<T>**, die jeweils ein Element aufbewahren bzw. referenzieren.

Ein Objekt der Knotenklasse **LinkedListNode<T>** ...

- macht den eigentlichen Inhalt über eine get/set - Eigenschaft namens **Value** vom Typ **T** zugänglich
- und kennt über die get-only - Eigenschaften **Previous** bzw. **Next** vom Typ **LinkedListNode<T>** den vorherigen bzw. nächsten Knoten.

Anschließend ist ein **LinkedList<String>** - Objekt zu sehen, das eine Liste mit drei Knoten bzw. Elementen verwaltet:



Im Abschnitt 5.11 haben wir eine (allerdings nur einfach) verkettete Liste selbst gebaut, um die Definition eines Indexers zu demonstrieren.

Dies sind die wesentlichen Vorteile der Klasse **LinkedList<T>** im Vergleich zur Array-basierten Klasse **List<T>**:

- Die Länge der Liste ist zu keinem Zeitpunkt festgelegt, sodass keine aufwändigen Maßnahmen zur Kapazitätsanpassung erforderlich sind.
- Beim Einfügen oder Löschen eines inneren Elements müssen keine anderen Elemente verschoben werden. Stattdessen wird …
    - ein neuer Knoten für das zu ergänzende Element erzeugt
    - oder der Knoten mit dem zu löschenden Element aufgegeben (dem Garbage Collector zur Entsorgung überlassen).

  Anschließend werden die Adressketten neu verknüpft.

Eine verkettete Liste hat aber auch Nachteile im Vergleich zu einer Array-basierten Liste:

- Die Klasse **LinkedList<T>** besitzt keinen Indexer für den wahlfreien Zugriff auf Listenelemente. Um das Listenelement an einer bestimmten Position aufzusuchen, müsste die Liste ausgehend vom ersten oder letzten Element durchlaufen werden, was zu einer schlechten Performanz führen würde.
- Eine verkettete Liste benötigt mehr Speicher, weil sich jedes Element in einem selbständigen Knotenobjekt auf dem Heap befindet.

Insgesamt sind verkettete Listen geeignet für Algorithmen, die …

- häufig innere Elemente einfügen oder entfernen
- und die Elemente nur sequentiell aufsuchen.

Die Klasse **LinkedList<T>** implementiert zwar die Schnittstelle **ICollection<T>**, aber *nicht* die Schnittstelle **IList<T>** (siehe Stammbaum der Kollektions-Schnittstellen im Abschnitt 11.3.1), sodass insbesondere kein Indexer zur Verfügung steht. Vorhanden sind neben den vom Interface **ICollection<T>** vorgeschriebenen Methoden und Eigenschaften (siehe Abschnitt 11.2) u. a. die folgenden Mitglieder:

- **public LinkedListNode<T> First { get; }**
  public LinkedListNode<T> Last { get; }
  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 ein neues Objekt der Klasse **LinkedListNode<T>** zur Aufbewahrung des Parameterelements am Anfang ein und liefert eine Referenz auf das neu erstellte Objekt zurück.
- **public LinkedListNode<T> AddLast(T** *element***)**
  Es wird ein neues Objekt der Klasse **LinkedListNode<T>** zur Aufbewahrung des Parameterelements am Ende angehängt und eine Referenz auf dieses Objekt zurückgeliefert.
- **public LinkedListNode<T> AddBefore(LinkedListNode<T>** *node***, T** *element***)**
  Vor dem im ersten Parameter angegebenen Knoten wird ein neues Objekt der Klasse **LinkedListNode<T>** zur Aufbewahrung des zweiten Parameters angelegt und eine Referenz auf den eingefügten Knoten zurückgeliefert.
- **public LinkedListNode<T> AddAfter(LinkedListNode<T>** *node***, T** *element***)**
  Hinter dem im ersten Parameter angegebenen Knoten wird ein neues Objekt der Klasse **LinkedListNode<T>** zur Aufbewahrung des zweiten Parameters angelegt und eine Referenz auf den eingefügten Knoten 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.

Die Methode **Add(T** *element***)** und die Eigenschaft **IsReadOnly** aus der implementierten Schnittstelle **ICollection<T>** werden von **LinkedList<T>** durch die explizite Schnittstellenimplementierung (siehe Abschnitt 9.4) realisiert (bzw. versteckt):

```
void ICollection<T>.Add(T value) {
   AddLast(value);
}
bool ICollection<T>.IsReadOnly {
    get { return false; }
}
```

Folglich können diese Mitglieder nur über eine **ICollection<T>** - Referenz angesprochen werden, z. B.:

Quellcode	Ausgabe

`var lili = new LinkedList<string>();`
`lili.AddLast("a");`
`ICollection<string> iclili = lili;`
`iclili.Add("b");`
`foreach (var s in lili)`
`    Console.WriteLine(s);`
	`a`
	`b`

11.4 Verwaltung einer Menge

Zur Verwaltung einer *Menge* von Elementen, die im Unterschied zu einer Liste *keine Dubletten* aufweisen darf, enthält die BCL u. a. die generischen Klassen **HashSet<T>** und **SortedSet<T>**. Diese Klassen sind nützlich, wenn Mengen im Sinne der Mathematik zu modellieren sind und entsprechende Operationen benötigt werden (z. B. Durchschnitt, Vereinigung oder Differenz von zwei Mengen). Im Vergleich zu anderen Kollektionsklassen können sie Mengenzugehörigkeitsprüfungen sehr schnell ausführen, was sie unverzichtbar macht, wenn derartige Prüfungen in großer Zahl auftreten.

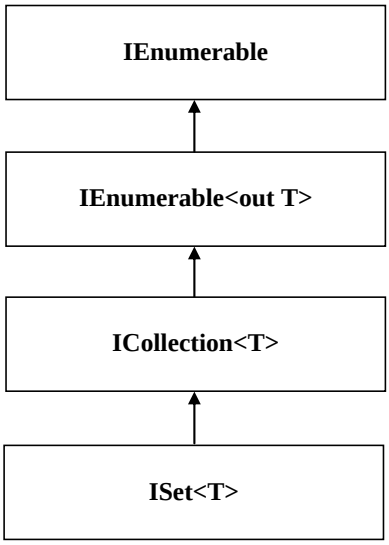
Während sich die Klasse **HashSet<T>** anbietet, wenn für die Instanzen des Elementtyps keine Anordnung besteht bzw. interessiert, ist die Klasse **SortedSet<T>** zu bevorzugen, wenn die Instanzen des Elementtyps eine vollständige Ordnung besitzen (z. B. Zeichenfolgen).

Anwendungsbeispiele:

- In einem Programm zur Urlaubsplanung in einer Firma werden die Urlaubstage der Mitarbeiter jeweils in einer Menge verwaltet, um z. B. über eine Durchschnittsbildung schwach besetzte Arbeitstage festzustellen.
- In einem Bewerbungsverfahren mit einer Liste von unverzichtbaren Kompetenzen (z. B. C#, HTML, CSS, SQL, etc.) werden die Kompetenzen jedes Bewerbers durch eine Menge verwaltet, um z. B. über eine Vereinigungsbildung festzustellen, ob bei einer bestimmten Kombination von Neueinstellungen alle Kompetenzen abgedeckt sind.

11.4.1 Interface ISet<T>

Die BCL-Mengenverwaltungsklassen implementieren das Interface **ISet<T>**, und dessen Stammbaum verrät, welche Handlungskompetenzen insgesamt vorhanden sind:



Die wichtigsten **ISet<T>** - Instanzmethoden sind:

- **public bool 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. Die **ICollection<T>** - Methode

  public void Add(T *element***)**

  wird in der erweiternden Schnittstelle **ISet<T>** durch die signaturgleiche Methode verdeckt:

  public new bool Add(T *element***)**
- **public void IntersectWith(IEnumerable<T>** *other***)**
  Das angesprochene Mengenverwaltungsobjekt streicht alle Elemente, die sich *nicht* in der Parameterkollektion (ein Objekt einer Klasse, die die Schnittstelle **IEnumerable<T>** erfüllt) befinden. Man erhält also die Schnittmenge:

  M (angesprochene Menge) ∩ **P** (Parametermenge)
- **public void UnionWith(IEnumerable<T>** *other***)**
  Das angesprochene Mengenverwaltungsobjekt nimmt aus der Parameterkollektion (ein Objekt einer Klasse, die 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 Mengenverwaltungsobjekt streicht alle Elemente, die sich in der Parameterkollektion (ein Objekt einer Klasse, die die Schnittstelle **IEnumerable<T>** erfüllt) befinden. Man erhält also die Differenzmenge:

  M (angesprochene Menge) - **P** (Parametermenge)
- **public bool IsSubsetOf(IEnumerable<T>** *other***)**
  Wenn das angesprochene Mengenverwaltungsobjekt eine Teilmenge der Parameterkollektion (ein Objekt einer Klasse, die die Schnittstelle **IEnumerable<T>** erfüllt) ist, dann wird **true** zurückgemeldet, anderenfalls **false**.
- **public bool IsProperSubsetOf(IEnumerable<T>** *other***)**
  Wenn das angesprochene Mengenverwaltungsobjekt eine *echte* Teilmenge der Parameterkollektion (ein Objekt einer Klasse, die die Schnittstelle **IEnumerable<T>** erfüllt) ist, dann

wird **true** zurückgemeldet, anderenfalls **false**. Die angesprochene Menge **M** ist eine *echte* Teilmenge der Parametermenge **P**, wenn **M** eine Teilmenge von **P** ist, und **P** mindestens ein Element enthält, das nicht zu **M** gehört.

- **public bool IsSupersetOf(IEnumerable<T>** *other***)**
  Wenn das angesprochene Mengenverwaltungsobjekt eine Obermenge der Parameterkollektion (ein Objekt einer Klasse, die die Schnittstelle **IEnumerable<T>** erfüllt) ist, dann wird **true** zurückgemeldet, anderenfalls **false**.
- **public bool IsProperSupersetOf(IEnumerable<T>** *other***)**
  Wenn das angesprochene Mengenverwaltungsobjekt eine *echte* Obermenge der Parameterkollektion (ein Objekt einer Klasse, die die Schnittstelle **IEnumerable<T>** erfüllt) ist (also eine Obermenge ist und noch mindestens ein zusätzliches Element enthält), dann wird **true** zurückgemeldet, anderenfalls **false**.

11.4.2 Hashtabellen und die Klasse HashSet<T>

Die beiden im Manuskript behandelten generischen Mengenverwaltungsklassen (**HashSet<T>** und **SortedSet<T>**) haben die folgenden Eigenschaften gemeinsam:

- Die Eindeutigkeit der Elemente ist garantiert (Verhinderung von Dubletten).
- Existenzprüfungen werden schnell ausgeführt.

Als Besonderheiten der Klasse **HashSet<T>** sind zu nennen:

- Es existiert keine bedeutsame Anordnung der Elemente.
  Der vom Interface **IEnumerable<T>** vorgeschriebenen Iterator ist vorhanden, liefert die Elemente quasi zufällig aus.
- Existenzprüfungen werden *sehr* schnell ausgeführt.

11.4.2.1 Anforderungen an den Elementtyp

Die in vielen Methoden (z. B. **Contains()** und **Remove()**) erforderlichen Übereinstimmungsprüfungen orientieren sich bei der Klasse **HashSet<T>** an der **Equals()** - Methode des Elementtyps, wenn nicht per **HashSet<T>** - Konstruktorparameter ein **IEqualityComparer<T>** - Objekt die Zuständigkeit für solche Prüfungen erhält. Ist kein externer Prüfer im Spiel, dann wird von einer Konkretisierung des **HashSet<T>** - Typformalparameters eine sinnvolle Überschreibung für die von **Object** geerbte Methode **Equals()** erwartet.

Bei Klassen ist meist eine Beurteilung auf Referenzgleichheit voreingestellt: Zwei Referenzen werden von der **Equals()** – Methode als identisch beurteilt (Rückgabe **true**), wenn sie auf dieselbe Speicherposition zeigen. Bei der Umsetzung dieses einfachen und oft sinnvollen Kriteriums stört die Möglichkeit von **null**-Werten. Wenn *beide* Referenzen den Wert **null** besitzen, dann führt ein Aufruf der Instanzmethode **Equals()** natürlich zu einem Ausnahmefehler. In diesem Fall lässt sich der Ausnahmefehler mit der statischen **Equals()** – Überladung der Klasse **Object** vermeiden, und man erhält die Rückgabe **true**, z. B.:

```
object obj1 = null, obj2 = null;
Console.WriteLine(object.Equals(obj1, obj2)); //Ausgabe: True
```

Die statische **Objekt**-Methode **Equals(Object** *objA***, Object** *objB***)** liefert …

- die Rückgabe **true**, wenn die beiden Referenzen übereinstimmen (z. B. beide gleich **null** sind)
- die Rückgabe **false**, wenn genau eine Referenz gleich **null** ist
- ansonsten die Rückgabe des Aufrufs `objA.Equals(objB)`
  Beide Referenzen sind von **null** verschieden, und das Ergebnis stammt von der virtuellen, also in abgeleiteten Klassen überschreibbaren Methode **Equals(Object** *obj***)**.

Soll bei Objekten auf jeden Fall die Referenzgleichheit beurteilt werden, dann verwendet man die statische **Object**-Methode **ReferenceEquals()**. Sie liefert genau dann die Rückgabe **true**, wenn die beiden Referenzen übereinstimmen (z. B. beide gleich **null** sind).

Zwar prüft bei BCL-Klassen die Methode **Equals()** meist auf Referenzgleichheit, doch gibt es auch einige signifikante Ausnahmen. Eine Wertgleichheitsprüfung findet statt bei …

- anonymen Klassen (siehe Abschnitt 6.5),
- **Record**-Klassen (siehe Abschnitt 6.7),
- der Klasse **String** (siehe Abschnitt 6.3.1).

Bei Werttypen nimmt die Methode **Equals()** auf jeden Fall eine Wertgleichheitsprüfung vor. Die vom Basistyp **System.ValueType** geerbte **Equals()** – Überschreibung berücksichtigt alle Felder, hat aber Performanzprobleme (vgl. Abschnitt 6.1.3):

- Die **ValueType**-Implementation muss in der Regel die zu vergleichenden Felder per Reflexion ermitteln.
- Wegen des Parametertyps **object** kommt das zeitaufwändige Boxing zum Einsatz.

Microsoft empfiehlt daher, bei selbst definierten Werttypen die Methode **Equals()** generell zu überschreiben:[1]

> For a value type, you should always override Equals, because tests for equality that rely on reflection offer poor performance.

Zur Vermeidung von Boxing-Operationen sollte bei der Definition einer Struktur nicht nur die **Object**-Methode **Equals(Object *obj*)** überschrieben, sondern auch die Schnittstelle **IEquatable<T>** implementiert, also eine **Equals(T** *other***)** – Überladung mit einem Parameter vom eigenen Typ definiert werden (siehe Abschnitt 6.1.3).

Eine aufgrund von Semantik- oder Performanzgründen definierte **Equals()** – Methode muss eine Äquivalenzrelation (im mathematischen Sinn) realisieren (vgl. Bloch 2018, S. 38ff), also den folgenden Kriterien genügen:

- Reflexivität
  Für jede nicht auf **null** zeigende Referenz `x` muss `x.Equals(x)` den Wert **true** liefern.
- Symmetrie
  Für zwei nicht auf **null** zeigende Referenzen `x` und `y` muss gelten: `x.Equals(y)` liefert den Wert **true** genau dann, wenn `y.Equals(x)` den Wert **true** liefert.
- Transitivität
  Für drei nicht auf **null** zeigende Referenzen `x`, `y` und `z` muss gelten: Wenn `x.Equals(y)` und `y.Equals(z)` den Wert **true** liefern, dann muss auch `x.Equals(z)` dieses Ergebnis besitzen.

Außerdem muss eine sinnvolle **Equals()** - Überschreibung die folgenden Bedingungen erfüllen:

- Konsistenz
  Für zwei nicht auf **null** zeigende Referenzen `x` und `y` muss `x.Equals(y)` bei *jedem* Aufruf den konstanten Wert **true** oder **false** liefern, solange bei den Objekten keine für den Vergleich relevante Änderung stattgefunden hat.
- Für jede nicht auf **null** zeigende Referenz `x` muss `x.Equals(null)` den Wert **false** liefern.

---

[1] https://learn.microsoft.com/en-us/dotnet/api/system.object.equals

- Haben zwei binäre Gleitkommavariablen `x` und `y` den Wert **NaN** (vgl. Abschnitt 4.6.2), dann muss `x.Equals(y)` das Ergebnis **true** liefern, z. B.:[1]

Quellcode	Ausgabe

`double x = 0.0 / 0.0;`
`double y = Double.PositiveInfinity - Double.PositiveInfinity;`
`Console.WriteLine($"x = {x}\ny = {y}" +`
`  $"\nx.Equals(y) = {x.Equals(y)}");`
	`x = NaN`
	`y = NaN`
	`x.Equals(y) = True`

Wenn ein Typ die **Object**-Methode **Equals()** überschreibt, dann muss er auch die **Object**-Methode **GetHashCode()** überschreiben (Bloch 2018, Item 11, S. 50ff). Ansonsten taugt der Typ nicht zur Konkretisierung des ...

- Typformalparameters **T** der Mengenverwaltungsklasse **HashSet<T>** (siehe Abschnitt 11.4.2)
- Typformalparameters **K** der Abbildungsverwaltungsklasse **Dictionary<K, V>** (siehe Abschnitt 11.5)

Weitere Anforderungen an die **GetHashCode()** - Überschreibung werden im Abschnitt 11.4.2.3 beschrieben.

Eine Überladung der Operatoren **==** und **!=** wird in der Regel konsistent mit der Methode **Equals()** vorgenommen, doch es gibt auch Ausnahmen. Im Fall der BCL-Klasse **StringBuilder** (vgl. Abschnitt 6.3.2) urteilt die Methode **Equals()** wertorientiert, während die Identitätsoperatoren sich an den Speicheradressen orientieren (Beispiel aus Albahari 2022, S. 330):

```
using System.Text;
var sb1 = new StringBuilder("abc");
var sb2 = new StringBuilder("abc");
Console.WriteLine(sb1 == sb2);       // Ausgabe: False
Console.WriteLine(sb1.Equals(sb2));  // Ausgabe: True
```

11.4.2.2 Handlungskompetenzen der Klasse HashSet<T>

Im folgenden Programm werden einige Methoden demonstriert, die **HashSet<T>** - Objekte aufgrund der implementierten Schnittstellen (u. a. **IEnumerable<T>**, **ICollection<T>** und **ISet<T>**) beherrschen:

Quellcode	Ausgabe

`using System;`
`using System.Collections.Generic;`
`class HashSetDemo {`
`  static void Main() {`
`  var m1 = new HashSet<char> { 'a', 'b', 'c' };`
`  Console.WriteLine("Menge 1:");`
`  foreach (char c in m1) Console.Write(c + " ");`
 	
`  m1.Add('b');`
`  Console.WriteLine("\n\nMenge 1 nach 2. b-Aufn.:");`
`  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'));`
	`Menge 1:`
	`a b c`
	
	`Menge 1 nach 2. b-Aufn.:`
	`a b c`
	
	`Menge 2:`
	`c d e`
	
	`b in der Menge 2? False`

[1] https://learn.microsoft.com/en-us/dotnet/api/system.object.equals

```
    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 + " ");
  }
}
```

```
Schnittmenge:
c

Vereinigungsmenge:
a b c d e

Differenzmenge:
a b

Menge 1 ohne a:
b c

Menge 1 nach Clear:
```

Weil **HashSet<T>** die Schnittstelle **IEnumerable<T>** erfüllt, kann man in einer **foreach**-Schleife über die Elemente einer **HashSet<T>** - Kollektion iterieren (siehe Abschnitt 9.5), wobei aber die Reihenfolge der Elemente quasi zufällig ist.

Damit die Instanzen eines Typs durch die Kollektionsklasse **HashSet<T>** sinnvoll verwaltet werden können, müssen in der Regel die von der Urahnklasse **Object** geerbten Methoden **Equals()** und **GetHashCode()** überschrieben werden:

- **Equals()** definiert die Gleichheit von zwei Instanzen und ist damit relevant bei der Identifikation von Dubletten und beim Entfernen von Elementen per **Remove()** – Aufruf (siehe Abschnitt 11.4.2).
- Mit Hilfe von **GetHashCode()** werden die schnellen Existenzprüfungen realisiert (siehe Abschnitt 11.4.2.3).

Über die folgende **HashSet<T>** - Konstruktorüberladung sorgt man dafür, dass für Identitätsvergleiche keine **Equals()** – Überschreibung bzw. – Überladung des Elementtyps zuständig ist, sondern ein Objekt aus einer Klasse, welche die Schnittstelle **IEqualityComparer<T>** im Namensraum **System.Collections.Generic**

public HashSet(IEqualityComparer<T> *comparer***)**

implementiert, die die beiden folgenden Methoden vorschreibt:

- **public bool Equals (T** *x***, T** *y***)**
- **public int GetHashCode (T** ***obj*****)**

Die abstrakte Klasse **System.StringComparer** deklariert in ihrem Definitionskopf das Interface **IEqualityComparer<String>**, implementiert die beiden vorgeschriebenen Methoden aber nicht. Allerdings besitzt die Klasse mehrere Eigenschaften, die Objekte aus konkreten, von **StringComparer** abgeleiteten Klassen referenzieren. Dort sind die **IEqualityComparer<String>** - Methoden auf unterschiedliche Weise implementiert, um jeweils eine gewünschte Verhaltensvariante zu realisieren. Z. B. liefert die Eigenschaft **StringComparer.OrdinalIgnoreCase** ein Objekt aus der von **StringComparer** abgeleiteten Klasse **OrdinalIgnoreCaseComparer**. In dieser Klasse ignoriert die Methode

public override bool Equals(string *x***, string** *y***)**

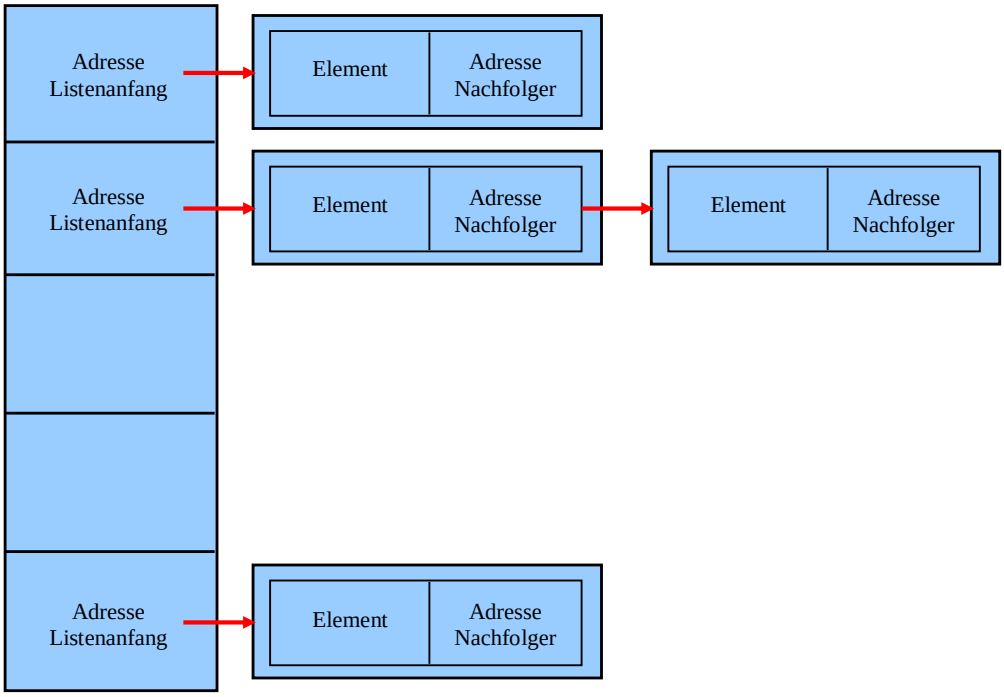
bei der Gleichheitsbeurteilung die Groß-/Kleinschreibung, sodass im folgenden Beispiel die Zeichenfolge „eins“ als Dublette von „Eins“ aus der Kollektion ferngehalten wird:

Quellcode	Ausgabe

`var hs = new HashSet<string>(StringComparer.OrdinalIgnoreCase) {`
`  "Eins", "eins", "zwei" };`
`foreach (string s in hs) Console.WriteLine(s);`
	`Eins`
	`zwei`

11.4.2.3 Hashtabellen

Zur Realisation von schnellen Existenzprüfungen verwendet die Klasse **HashSet<T>** eine sogenannte *Hashtabelle*. Zur Speicherung der Elemente kommt ein Array zum Einsatz, der sogenannte *buckets* (dt.: *Eimer*) als Elemente enthält, wobei es sich um einfach verkettete Listen handelt:



Bei der Aufnahme eines neuen Elements und auch bei der Existenzprüfung wird der Bucket-Index über die typspezifische Implementierung der bereits in der Urahnklasse **Object** vorhandenen und für einen Elementtyp geeignet zu überschreibenden Instanzmethode **GetHashCode()** ermittelt:

public int GetHashCode()

Beim Einfügen eines neuen Elements ist die Bucket-Liste zum berechneten Index idealerweise noch leer. Anderenfalls spricht man von einer *Hash-Kollision*, und es entsteht ein Zusatzaufwand zur Aufnahme in die Liste.

Wenn im **T**-Konkretisierungstyp die auf Referenzgleichheit basierende **Object**-Methode **Equals()** durch eine auf Inhaltsgleichheit basierende Methode überschrieben wird, dann muss auch das **Object**-Erbstück **GetHashCode()** überschrieben werden. Eine zum Befüllen einer Hashtabelle

einzusetzende **GetHashCode()** - Methode muss mit der **Equals()** – Methode kompatibel sein und weitere Anforderungen erfüllen:[1]

- Sind zwei Instanzen identisch im Sinne der **Equals()** - Methode, dann müssen sie bei einem **GetHashCode()** - Aufruf denselben Wert liefern.
- Für zwei verschiedene Instanzen sollte die Methode **GetHashCode()** mit sehr hoher Wahrscheinlichkeit unterschiedliche Werte liefern.
- Während eines Programmlaufs müssen alle Methodenaufrufe für ein Objekt denselben Wert liefern, solange bei diesem Objekt keine Veränderungen mit Relevanz für die **Equals()** - Methode auftreten.
- Wenn sich der Hashcode einer Instanz nach der Aufnahme in eine Hashtabelle ändert, dann kann die Instanz nicht mehr abgerufen werden. Folglich sollte die **GetHashCode()** – Rückgabe nur auf unveränderlichen Feldern basieren.
- Die **GetHashCode()** - Werte sollten möglichst zufällig und damit gleichmäßig verteilt sein. Ansonsten resultieren zu wenige Buckets mit zu langen verketteten Listen, und die Existenzprüfung wird ineffizient.
- Die **GetHashCode()** - Methode muss schnell sein und darf keine Ausnahme werfen.

Aus dem Hashcode einer Instanz und der Hashtabellen-Kapazität wird der Array-Index per Modulo-Operation ermittelt:

Array-Index = Hashcode % Kapazität

Um für eine Instanz mit der Methode **Contains()** festzustellen, ob sie bereits in der Hashtabelle bzw. in der Menge enthalten ist, muss die Instanz nicht über **Equals()** - Aufrufe mit allen Insassen verglichen werden. Stattdessen wird ihr Hashcode berechnet und der Array-Index ermittelt. Befindet sich hier noch keine Listenadresse, ist die Existenzfrage geklärt (**Contains()** - Rückmeldung **false**). Anderenfalls ist nur für die Instanzen der im Array-Element adressierten Liste eine **Equals()** - Untersuchung erforderlich.

Damit es möglichst selten zu Hash-Kollisionen kommt, sollte die Array-Größe ungefähr das 1,5 - fache der Anzahl aufzunehmender Elemente betragen (Horstmann & Cornell, 2002, S. 137). Eine Vergrößerung des Bucket-Arrays ist u. a. wegen der erforderlichen Neuberechnung der Array-Positionen aus den Hashcodes aufwändig. Daher sollte eine Vergrößerung durch die Verwendung der folgenden Konstruktorüberladung mit einem Parameter für die Kapazität vermieden werden:

public HashSet(int *capacity***)**

Es wird vielfach empfohlen, für die Kapazität einer Hashtabelle eine Primzahl zu wählen, sodass z. B. bei erwarteten 100 Elementen die Kapazität nicht auf 150, sondern auf 151 festzulegen ist. Anwender der Klasse **HashSet<T>** müssen die zu einer geplanten Kapazität passende Primzahl nicht selbst ermitteln, weil beim Einsatz des eben angegebenen Konstruktors die folgende Methode **Initialize()** ins Spiel kommt und mit Hilfe der statischen Methode **GetPrime()** der Klasse **HashHelpers** die angeforderte Kapazität durch die nächstgelegene, mindestens gleich große Primzahl ersetzt:[2]

---

[1] Der C# - Insider Eric Lippert erläutert die Anforderungen an eine **GetHashCode()** - Implementierung in seinem Blog:

https://ericlippert.com/2011/02/28/guidelines-and-rules-for-gethashcode/

Dort wird auch erläutert, warum man sich beim CTS-Design (*Common Type System*) dafür entschieden hat, dass bereits die Urahnklasse **Object** die Methode **GetHashCode()** beherrschen soll:

> I think if we were redesigning the type system from scratch today, hashing might be done differently, perhaps with an IHashable interface. But when the CLR type system was designed there were no generic types and therefore a general-purpose hash table needed to be able to store any object.

[2] Wie man den BCL-Quellcode einsehen kann, erläutert der Abschnitt 2.6.4.

```
/// <summary>
/// Initializes buckets and slots arrays. Uses suggested capacity by finding next prime
/// greater than or equal to capacity.
/// </summary>
private int Initialize(int capacity) {
    int size = HashHelpers.GetPrime(capacity);
    var buckets = new int[size];
    var entries = new Entry[size];

    _freeList = -1;
    _buckets = buckets;
    _entries = entries;
#if TARGET_64BIT
            _fastModMultiplier = HashHelpers.GetFastModMultiplier((uint)size);
#endif

    return size;
}
```

Im Vergleich zur Aufbewahrung in einer **List<T>** - Kollektion nimmt man bei einer **HashSet<T>** - Kollektion einen erhöhten Speicherbedarf in Kauf. Es resultiert aber eine enorme Zeitersparnis, wenn in einem Algorithmus viele Existenzprüfungen erforderlich sind. In einem Testprogramm wurden die Klassen **List<String>**, **LinkedList<String>** und **HashSet<String>** sowie die im Abschnitt 11.4.3 behandelte Klasse **SortedSet<String>** dazu verwendet, die folgenden Aufgaben zu erledigen:

- eine Kollektion mit 20.000 **String**-Objekten füllen
- für 20.000 neue **String**-Objekte prüfen, ob sie bereits in der Kollektion vorhanden sind

Dabei zeigen sich die folgenden Ergebnisse:[1]

```
Kollektionsklasse:                      List`1
 Zeit zum Füllen:                            6 ms
 Zeit für die Existenzprüfungen:          3011 ms

Kollektionsklasse:                      LinkedList`1
 Zeit zum Füllen:                            9 ms
 Zeit für die Existenzprüfungen:          4002 ms

Kollektionsklasse:                      HashSet`1
 Zeit zum Füllen:                           12 ms
 Zeit für die Existenzprüfungen:             3 ms

Kollektionsklasse:                      SortedSet`1
 Zeit zum Füllen:                           45 ms
 Zeit für die Existenzprüfungen:            39 ms
```

Wie erwartet, ist die Klasse **HashSet<String>** bei den Existenzprüfungen den Klassen **List<String>** und **LinkedList<String>** drastisch überlegen. Bei der Klasse **String** ist von einer effizienten Überschreibung der Methode **GetHashCode()** auszugehen.

Die gleich vorzustellende Klasse **SortedSet<T>** ist wie die Klasse **HashSet<T>** zur Verwaltung einer Menge konzipiert und implementiert ebenfalls das Interface **ISet<T>**. Sie hält aber ihre Elemente in einem sortierten Zustand und benötigt für diese Extraleistung Zeit. Bei den

---

[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:

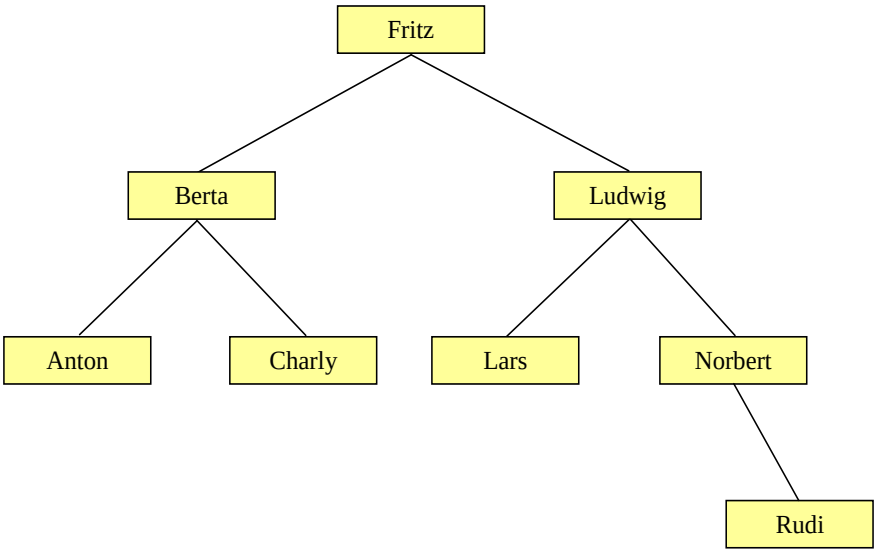
...\BspUeb\Kollektionen\ListSetContains

Existenzprüfungen benötigt die Klasse **SortedSet<T>** mehr Zeit als die Klasse **HashSet<T>**, ist aber den Klassen **List<String>** und **LinkedList<String>** deutlich überlegen.

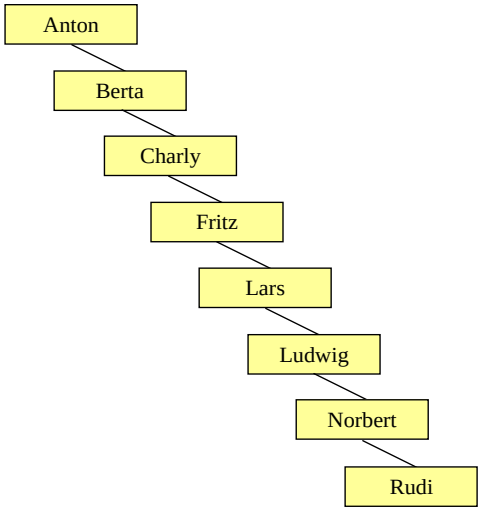
11.4.3 Balancierte Binärbäume und die Klasse SortedSet<T>

Existiert für die Elemente einer Menge eine vollständige Ordnung (z. B. bei Zeichenfolgen aufgrund der Sortierordnung), dann kann man über einen sogenannten *Binärbaum* die Elemente im sortierten Zustand halten, ohne den Aufwand bei den Mengenverwaltungsmethoden (z. B. **Add()**, **Contains()** und **Remove()**) zu groß werden zu lassen.

In einem Binärbaum hat jeder Knoten maximal zwei direkte Nachfolger, wobei der linke Nachfolger einen kleineren und der rechte Nachfolger einen höheren Rang hat, was die folgende Abbildung für Zeichenketten illustriert:



Bei einem *balancierten* Binärbaum kommen Forderungen zur maximalen Entfernung zwischen der Wurzel und einem Element hinzu, d. h. Forderungen zur Anzahl der Ebenen, um den Aufwand beim Suchen und Einfügen von Elementen zu begrenzen. Der bisher betrachtete Namensbaum ist balanciert (4 Ebenen), während in der folgenden Abbildung eine extrem unbalancierte Anordnung derselben Elemente zu sehen ist (8 Ebenen):



Zur Beurteilung des Aufwands bei der Suche nach einem Element (oder bei der Neuaufnahme eines Elements) gehen wir von einem balancierten und vollständig gefüllten Binärbaum aus. Hier haben alle Knoten, die keine Endknoten sind, genau zwei Nachfolger. In der ersten Variante des

Namensbaums bestand diese Situation vor der Aufnahme von Rudi. Der maximale Aufwand bei einer Existenzprüfung oder Neuaufnahme ist identisch mit der Zahl *m* der Ebenen, weil pro Ebene ein Vergleich vorzunehmen ist. Wir schätzen nun ab, wie viele Ebenen ein balancierter Binärbaum zur Verwaltung von *k* Elementen benötigt.

Aus der Anzahl *m* der Ebenen kann man nach der folgenden Formel die Anzahl *k* der Elemente in einem balancierten und vollständig gefüllten Binärbaum berechnen:

$$k = 2^m - 1$$

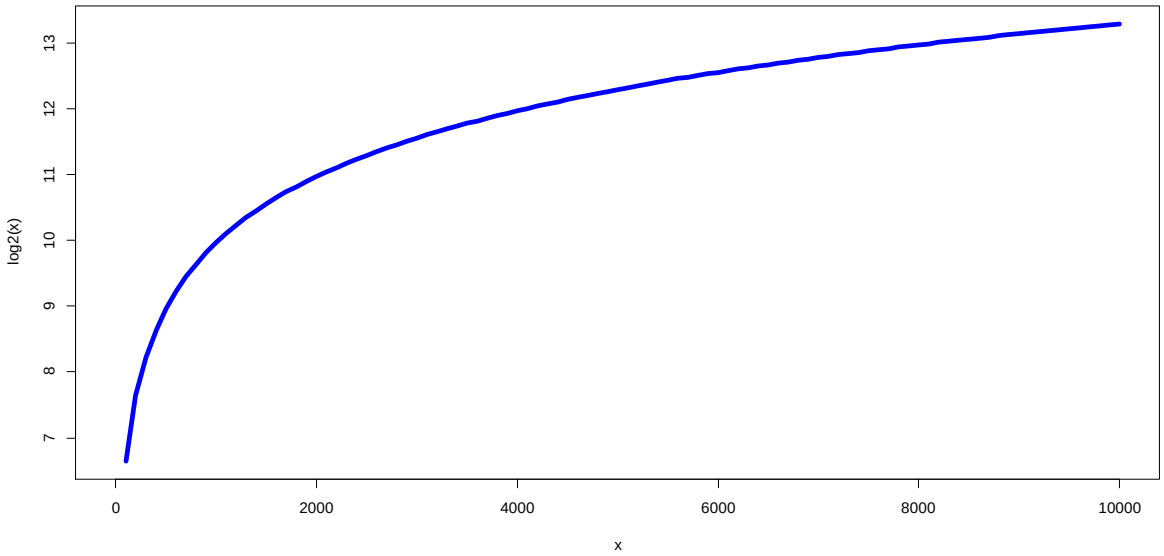
Bei $m = 3$ Ebenen resultieren z. B. 7 Elemente (siehe Beispiel). Man erhält *k* als Partialsumme der geometrischen Reihe:[1]

$$k \sum_{i=0}^{m-1} 2^i = \frac{1 - 2^m}{1 - 2} = 2^m - 1$$

Für ein hinreichend großes *k* kann man die Beziehung zwischen *k* und *m* vereinfachen und dann durch Anwendung der Logarithmusfunktion nach *m* auflösen, um die zur Verwaltung von *k* Elementen erforderliche Anzahl von Ebenen zu ermitteln:

$$\begin{aligned} k &= 2^m \\ \leftrightarrow \log_2(k) &= m \end{aligned}$$

Für hinreichend großes *k* sind also $\log_2(k)$ Ebenen erforderlich, und im ungünstigsten Fall werden ebenso viele Algorithmusschritte benötigt, um ein Element zu suchen oder die Position für ein neues Element zu bestimmen. Man sagt unter Verwendung einer Notation mit dem griechischen Großbuchstaben *O* (Omikron), der Algorithmus sei von der Ordnung $O(\log_2 k)$. Weil das monotone Wachstum der Logarithmus-Funktion relativ flach verläuft, steigt der Aufwand nur langsam mit der Anzahl *k* der Elemente an:[2]



Bei einer Hashtabelle wächst der Aufwand einer Existenzprüfung *nicht* mit der Anzahl der Elemente, und es resultiert die optimale Ordnung $O(1)$. Bei einer Liste hingegen ist der Aufwand einer

---

[1] Hier wird die Formel mit elementaren Mitteln bewiesen:
https://de.wikibooks.org/wiki/Mathe_f%C3%BCr_Nicht-Freaks:_Geometrische_Reihe

[2] Der Funktionsgraph wurde mit der Programmiersprache R für statistische Berechnungen und Grafiken folgendermaßen erstellt:

```
curve(log2(x), 0, 10000, col="blue", lwd=5)
```

Existenzprüfung direkt proportional zur Anzahl der Elemente, und es resultiert die ungünstige Ordnung $O(k)$.

In der BCL nutzt die Klasse **SortedSet<T>** das Prinzip des balancierten Binärbaums, wobei die sogenannte **Rot-Schwarz** - Architektur Verwendung findet. Sie sorgt für einen annähernd ausbalancierten Baum und damit für Effizienz beim Suchen, Einfügen und Löschen. Folglich verursacht die Anordnung der Elemente keine allzu großen Kosten für Existenzprüfungen (siehe die im Abschnitt 11.4.2.3 berichteten Ergebnisse zum Leistungsvergleich von **HashSet<String>** und **SortedSet<String>**).

Die Klassen **HashSet<T>** und **SortedSet<T>** implementieren im Wesentlichen dieselben Schnittstellen (insbesondere **ISet<T>**, siehe Abschnitt 11.4.2.2) und bieten damit ähnliche Funktionsumfänge.[1] Als Gegenleistung für den höheren Aufwand bei Existenzprüfungen ($O(\log_2 k)$ statt $O(1)$) gewinnt man bei der Klasse **SortedSet<T>** die folgenden Zusatzfunktionen:

- Die Elemente einer **SortedSet<T>** - Kollektion erscheinen beim Iterieren in der Sortierordnung, und der benannte Iterator **Reverse()** erlaubt das Iterieren in umgekehrter Richtung (siehe Abschnitt 9.5.2 zu benannten Iteratoren).
- **SortedSet<T>** besitzt die Eigenschaften **Min** und **Max**, die das kleinste bzw. größte Element in der Menge liefern:
  - **public T Min { get; }**
  - **public T Max { get; }**

Einen Indexer, der bei **HashSet<T>** aus naheliegenden Gründen fehlt, bietet auch **SortedSet<T>** nicht an, sodass kein Elementzugriff per **[]** - Operator möglich ist.

Im Unterschied zu **HashSet<T>** hängen bei **SortedSet<T>** die in Methoden wie **Contains()** und **Remove()** erforderlichen Übereinstimmungsprüfungen *nicht* vom Hashcode und von der **Equals()** – Rückgabe ab, sondern vom Vergleichsergebnis, das die **CompareTo()** – Methode des Elementtyps oder ein per Konstruktor-Parameter übergebenes **IComparer<T>** - Objekt liefert.

11.5 Verwaltung von (Schlüssel-Wert) - Paaren

Zur Verwaltung von (Schlüssel-Wert) - Paaren enthält die BCL die generische Klasse **Dictionary<K, V>**. Als Konkretisierung für die beiden Typformalparameter sind beliebige Typen erlaubt, sodass die Klasse keinesfalls auf Wörterbuchanwendungen eingeschränkt ist. Weil jeder Schlüssel (jede **K**-Instanz) auf einen Wert (eine **V**-Instanz) abgebildet wird, bezeichnet man **Dictionary<K, V>** auch als *Abbildungsverwaltungsklasse*. Die Schlüssel (mit einer Konkretisierung des Typformalparameters **K** als Datentyp) werden wie eine Menge verwaltet, sodass Eindeutigkeit garantiert ist (ohne Dubletten), aber keine relevante Anordnung besteht. Über einen Schlüssel ist der zugehörige Wert ansprechbar (mit einer Konkretisierung des Typformalparameters **V** als Datentyp), wobei der Zugriff dank Hashtabellen-Technik sehr schnell erfolgt. Wird bei der Konkretisierung des Typformalparameters **K** die von **Object** geerbte Methode **Equals()** überschrieben, dann muss das auch mit der **Object**-Methode **GetHashCode()** geschehen (vgl. Abschnitte 11.4.2.1 und 11.4.2.3).

Neben **List<T>** ist **Dictionary<K, V>** die am häufigsten verwendete BCL-Kollektionsklasse (Albahari 2022, S. 374). Man könnte durch ein **Dictionary<K, V>** – Objekt z. B. den lokalen Zwischenspeicher für eine Personaldatenbanktabelle realisieren mit ...

---

[1] Bei den implementierten Schnittstellen besteht (in .NET 7) der einzige Unterschied darin, dass **SortedSet<T>** neben der generischen Schnittstelle **ICollection<T>** auch die nicht-generische Variante **ICollection** implementiert, während sich **HashSet<T>** auf die generische Variante beschränkt.

- einer eindeutigen Personalnummer (Typ **int**) 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 bool ContainsKey(K** *key***)**
  Diese Methode informiert darüber, ob ein Element mit dem fraglichen Schlüssel vorhanden ist, und wird sehr schnell ausgeführt.
- **public bool ContainsValue(V** *value***)**
  Diese Methode informiert darüber, ob ein Element mit dem fraglichen Wert vorhanden ist, und nimmt potenziell viel Zeit in Anspruch.
- **public bool TryGetValue(K** *key***, out V** *value***)**
  Wenn ein Element mit dem angegebenen Schlüssel vorhanden ist, dann wird sein Wert in den **out**-Parameter *value* geschrieben und die Rückgabe **true** geliefert. Ist der Schlüssel *nicht* vorhanden, dann wird die Rückgabe **false** geliefert und der typspezifische Nullwert in den **out**-Parameter geschrieben. Die Rückgabe **false** zu ignorieren und z. B. beim **V**-Typ **int** eine erhaltene 0 als Information über ein Kollektionselement zu interpretieren, ist ein gravierender Programmierfehler.
- **public bool Remove(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.
- **public void Clear()**
  Es werden alle Elemente entfernt.

Die Methoden **Add(K** *key*, **V** *value***)**, **ContainsKey(K** *key***)**, **TryGetValue(K** *key*, **out V** *value***)**, und **Remove(K** *key***)** sind durch die von **Dictionary<K, V>** implementierte Schnittstelle **IDictionary<K, V>** vorgeschrieben. Diese Schnittstelle verlangt außerdem die folgenden get-only - Eigenschaften:

- **public ICollection<K> Keys { get; }**
  Diese Eigenschaft liefert eine Sicht auf die Schlüssel als Objekt der geschachtelten Klasse **Dictionary<K, V>.KeyCollection**.

```
public sealed class KeyCollection : ICollection<TKey>,
                                    ICollection, IReadOnlyCollection<TKey> {
 . . .
}
```

- **public ICollection<V> Values { get; }**
  Diese Eigenschaft liefert eine Sicht auf die Werte als Objekt der geschachtelten Klasse **Dictionary<K, V>.ValueCollection**.

```
public sealed class ValueCollection : ICollection<TValue>,
                                     ICollection, IReadOnlyCollection<TValue> {
 . . .
}
```

Schließlich schreibt die Schnittstelle **IDictionary<K, V>** auch einen Indexer

public V this[K *key***] { get; set; }**

vor, sodass sich über einen in eckige Klammern eingeschlossenen Schlüssel der zugehörigen Wert ermitteln und festlegen lässt. Es ist sogar möglich, per Indexer ein (Schlüssel-Wert) - Paar einzufügen, z. B.:

```
var fred = new Dictionary<char, int>();
fred['c'] = 1;
Console.WriteLine(fred['c']); // Ausgabe: 1
```

Für die Klasse **Dictionary<K, V>** sind zwei Kollektionsinitialisierer-Varianten verfügbar:

- Bei dieser Variante findet ein impliziter Aufruf des Indexers durch den Compiler statt:
```
var fred = new Dictionary<char, int> {
    ['c'] = 1, ['b'] = 3
};
```
- Bei dieser Variante findet ein impliziter Aufruf der Methode **Add(K, V)** durch den Compiler statt:
```
var fred = new Dictionary<char, int> {
    {'c', 1}, {'b', 3}
};
```

Durchläuft man eine **Dictionary<K, V>** - Kollektion per **foreach** - Schleife, dann ist als Elementtyp die passend konkretisierte generische Struktur **KeyValuePair<K, V>** im Einsatz, z. B.:

```
foreach (KeyValuePair<char, int> kvp in fred)
   Console.WriteLine($"{kvp.Key} : {kvp.Value}");
```

Dank der Fähigkeit des Compilers zur Typinferenz kann man sich die Typbezeichnung sparen, z. B.:

```
foreach (var kvp in fred)
   Console.WriteLine($"{kvp.Key} : {kvp.Value}");
```

Das folgende Programm verwendet ein **Dictionary<char, int>** - Objekt dazu, um für einen **String** die Häufigkeiten der enthaltenen Zeichen zu ermitteln:

Quellcode	Ausgabe

`using System;`
`using System.Collections.Generic;`
 	
`class DictionaryDemo {`
`  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`

Ein Visual Studio - Projekt mit dem Programm ist im folgenden Ordner zu finden

…\BspUeb\Kollektionen\DictionaryDemo

Um eine nach den Schlüsseln *sortierte* Tabelle mit (Schlüssel-Wert) - Paaren zu erhalten, ersetzt man die Klasse **Dictionary<K, V>** durch die Alternative **SortedDictionary<K, V>**. Analog zur Klasse **SortedSet<T>** verwendet die Klasse **SortedDictionary<K, V>** zur Verwaltung der

Schlüssel einen balancierten Binärbaum mit Rot-Schwarz - Architektur, um ein performantes Suchen, Einfügen und Löschen zu ermöglichen (vgl. Abschnitt 11.4.3).

11.6 Klasse Collection<T>

Gelegentlich wird eine Kollektionsklasse mit individuellen Verhalten benötigt, sodass sich in der BCL keine direkt verwendbare Lösung findet, z. B.:

- Das Entfernen von obligatorischen Elementen per **Remove()** soll verhindert werden.
- Jede Veränderung der Zusammenstellung soll ein Ereignis auslösen.

Im Namensraum **System.Collections.ObjectModel** befinden sich Klassen mit eine Grundausstattung an Verhaltenskompetenzen, die abgeleiteten Klassen die Kontrolle über Veränderungsoperationen erlauben und daher als Basisklassen geeignet sind. Wir beschränken uns auf die Klasse **Collection<T>**, die u. a. die Schnittstelle **IList<T>** implementiert:

```
public class Collection<T> : IList<T>, IList, IReadOnlyList<T> {
 . . .
}
```

Weil die Klasse nicht abstrakt ist, taugt sie zum Instanziieren (Erzeugen von Objekten), wobei im Vergleich zur Klasse **List<T>** eine deutlich eingeschränkte Ausstattung mit Methoden vorhanden ist.[1] In der Rolle als Basisklasse für eine Eigenkreation hat **Collection<T>** aber einen entscheidenden Vorteil gegenüber **List<T>**. Wie das Beispiel der **Collection<T>** - Methode **Insert()**

```
public void Insert(int index, T item) {
  if (items.IsReadOnly) {
    ThrowHelper.ThrowNotSupportedException(ExceptionResource.NotSupported_ReadOnlyCollection);
  }
  if ((uint)index > (uint)items.Count) {
    ThrowHelper.ThrowArgumentOutOfRange_IndexMustBeLessOrEqualException();
  }
  InsertItem(index, item);
}
```

zeigt, werden Kollektionsveränderungen durch virtuelle Methoden mit der Schutzstufe **public** vorgenommen:

```
protected virtual void InsertItem(int index, T item) {
    items.Insert(index, item);
}
```

Diese Methoden kann eine abgeleitete Klasse überschreiben, was z. B. die folgende Klasse `MyObservableCollection` tut. Ihre Leistungserweiterung gegenüber der Basisklasse besteht im Ereignis `Inserted`, das von der überschriebenen Methode `InsertItem()` ausgelöst wird:[2]

---

[1] Während die Klasse **Collection<T>** mit ca. 400 Quellcodezeilen auskommt, besitzt der **List<T>** - Quellcode einen ca. dreimal so großen Umfang. Die geringere Komplexität der Klasse **Collection<T>** muss aber kein Nachteil sein.

[2] Eine als Vorbild verwendete Klasse, die das komplexere Ereignis `Changed` anbietet, wird hier beschrieben: https://learn.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.collection-1

```
public class MyObservableCollection : Collection<string> {
    public event EventHandler<ItemInsertedEventArgs> Inserted;

    protected override void InsertItem(int index, string item) {
        base.InsertItem(index, item);
    if (Inserted != null)
        Inserted(this, new ItemInsertedEventArgs(item));
    }
}

public class ItemInsertedEventArgs : EventArgs {
    public readonly string InsertedItem;
    public ItemInsertedEventArgs(string item) {
        InsertedItem = item;
    }
}
```

Die Klasse `ItemInsertedEventArgs` dient zur Ereignisbeschreibung.

Ein Ereignis darf nicht aufgerufen werden, wenn die Aufrufliste leer und folglich die Delegatenvariable gleich **null** ist, weil es ansonsten zu einer **NullReferenceException** kommt. In der Methode `InsertItem()` wird dieser Unfall per **if**-Anweisung verhindert:

```
if (Inserted != null)
    Inserted(this, new ItemInsertedEventArgs(item));
```

In einer Multithreading-Anwendung (mit mehreren parallelen Ausführungsfäden) kann es passieren, dass eine beim Testen noch gefüllte Aufrufliste vor dem Aufruf durch einen anderen Thread entleert wird. Daher sieht man oft eine Absicherung nach dem folgenden Muster:

```
EventHandler<ItemInsertedEventArgs> temp = Inserted;
if (temp != null)
    temp(this, new ItemInsertedEventArgs(item));
```

Es wird eine für andere Threads unzugängliche Kopie der Delegatenvariablen erstellt, was der Compiler durch die implizite Wandlung vom Ereignistyp in den zugehörigen Delegatentyp unterstützt. Weil diese Absicherung das Multithreading-Problem beim Ereignisaufruf trotz der Unveränderlichkeit von Delegatenobjekten nicht zuverlässig löst, wird in der Methode `InsertItem()` darauf verzichtet. Um das Problem wirksam zu verhindern, muss das (De)registrieren von Ereignisbehandlungsmethoden demjenigen Thread vorbehalten bleiben, der das Ereignis auslöst (siehe Abschnitt 10.2.1).

Durch den folgenden Einsatz der Klasse `MyObservableCollection`

```
public class MyObservableCollectionDemo {
    private static void InsertedHandler(object source, ItemInsertedEventArgs e) {
        Console.WriteLine("Inserted Item: " + e.InsertedItem);
    }

    public static void Main() {
        MyObservableCollection myObsColl = new() { "eins", "zwei", "drei" };
        myObsColl.Inserted += InsertedHandler;
        myObsColl.Insert(0, "Meins");

        Collection<string> col = myObsColl;
        col.Insert(0, "Meins (Collection)");

        Console.WriteLine("\nPress key to exit"); Console.ReadKey();
    }
}
```

wird belegt, dass ein `MyObservableCollection` – Objekt auch bei einer Ansprache per **Collection<string>** - Referenz seine überschreibende Methode ausführt (Polymorphie):

```
Inserted Item: Meins
Inserted Item: Meins (Collection)
```

Verwendet man die Basisklasse **List<string>** und verdeckt in der abgeleiteten Klasse `ListWithInsertEvent` die geerbte Methode **Insert()**, dann wird die verdeckende Methode nur beim Aufruf per `ListWithInsertEvent` – Referenz angesprochen. Bei einem Aufruf per Basisklassenreferenz wird das Ereignis `Inserted` also *nicht* ausgelöst:

```
class ListWithInsertEvent : List<string> {
    public event EventHandler<ItemInsertedEventArgs> Inserted;

    public new void Insert(int index, string item) {
        base.Insert(index, item);
        if (Inserted != null)
            Inserted(this, new ItemInsertedEventArgs(item));
    }
}
```

Es ist nicht unbedingt sinnvoll, Aufwand in die Beobachtbarkeit bzw. Ereigniskommunikation der Klasse `MyObservedCollection` zu investieren, weil die BCL mit der von **Collection<T>** abgeleiteten Klasse **ObservableCollection<T>** eine komplette und bewährte Lösung enthält.

Visual Studio - Projekte mit den Klassen `MyObservedCollection` bzw. `ListWithInsertEvent` sind hier zu finden:

…\BspUeb\Kollektionen\MyObservableCollection
…\BspUeb\Kollektionen\ListWithInsertEvent

11.7 Unveränderliche Kollektionen

Die Vorteile unveränderlicher Instanzen wurden im Manuskript schon mehrfach erwähnt (siehe z. B. Abschnitt 5.2.5). Besonders gravierend und unmittelbar spürbar ist der durch unveränderliche Instanzen eingesparte Synchronisierungsaufwand bei der zur effektiven Nutzung aktueller CPU-Architekturen relevanten Multithreading-Programmierung (siehe Kapitel 17 in Baltes-Götz (2021)):

- Die Software-Entwickler sparen Zeit und vermeiden Fehler.
- Die Software-Anwender profitieren von einer verbesserten Performanz, denn auch eine fehlerfreie Synchronisierung kostet Leistung, weil sich die beteiligten Threads potenziell durch das Sperren von Code-Segmenten bzw. Speicherbereichen gegenseitig behindern.

Wenn unveränderliche Typen (z. B. **String**) eine Methode zur „Änderung“ von Instanzen anbieten (z. B. die **String**-Methode **Trim()** zum Entfernen von Leerzeichen am Anfang und am Ende einer Zeichenfolge), dann liefern solche Methoden eine *neue* Instanz. Auch bei einer „unveränderlichen“ Kollektion soll z. B. die Aufnahme oder das Löschen von Elementen möglich bleiben. Dabei muss aber eine *neue* Kollektion erstellt werden, damit eine laufende Verarbeitung des bisherigen Zustands ungestört zu Ende gebracht werden kann.

Im BCL-Namensraum **System.Collections.Immutable** finden sich unveränderliche Kollektionsklassen für unterschiedliche Aufgaben, die unter Beachtung der folgenden Designvorgaben entstanden sind:

- Es wird im Wesentlichen das von veränderlichen Kollektionsklassen bekannte API angeboten. Weil viele Methoden eine neue Kollektion zurückliefern müssen, waren aber Anpassungen erforderlich. Während z. B. die **List<T>** - Methode **Add()** den Rückgabetyp **void** besitzt, liefert die **ImmutableList<T>** - Methode **Add()** eine Rückgabe vom Typ **ImmutableList<T>**.
- Der Aufwand zum Erstellen einer neuen Kollektion anlässlich einer Änderung wird durch die Verwendung vorhandener Objekte reduziert.

Microsoft demonstriert den Nutzen der unveränderlichen Kollektionen in zwei wichtigen Programmen (Albahari 2022, S. 388):

- im C# - Compiler, dessen technische Basis unter dem Namen *Roslyn* bekannt ist
- im Visual Studio

Trotz des Aufwands beim Renovieren von unveränderlichen Kollektionen kommt es zu einem Leistungsgewinn, weil die einfache und effiziente Multithreading-Programmierung eine bessere Nutzung der heute üblichen Mehrkern-CPUs ermöglicht.

In der folgenden Tabelle mit den wichtigsten unveränderlichen BCL-Kollektionsklassen wird auch die intern verwendete Architektur angegeben:[1]

Klasse	**Architektur**
ImmutableList<T>	Binärbaum
ImmutableArray<T>	Array
ImmutableHashSet<T>	Binärbaum
ImmutableSortedSet<T>	Binärbaum
ImmutableDictionary<K,V>	Binärbaum
ImmutableSortedDictionary<K,V>	Binärbaum
ImmutableStack<T>	Verkettete Liste
ImmutableQueue<T>	Verkettete Liste

Unveränderlich ist bei diesen Kollektionsklassen die Zusammensetzung (z. B. die Liste bzw. Menge der Elemente), während über die Veränderlichkeit der Elemente keine Aussage gemacht wird. Eine (z. B. zum Erreichen der Thread-Sicherheit erforderliche) vollständige Unveränderlichkeit wird nur erreicht, wenn auch der Elementtyp unveränderlich ist (z. B. **String** oder **DateTime**)

Von einer unveränderlichen Kollektion ist eine auf Lesezugriffe beschränkte *Sicht* auf eine veränderliche Kollektion zu unterscheiden, die z. B. über die **List<T>** - Methode **AsReadOnly()** zu erhalten ist:

public ReadOnlyCollection<T> AsReadOnly()

Die zugrundeliegende Kollektion kann zwar nicht per **ReadOnlyCollection<T>** – Objekt geändert werden, auf andere Weise aber schon, z. B.:

Quellcode	Ausgabe

`using System.Collections.Immutable;`
 	
`var liste = new List<int>() { 1, 2, 3 };`
`var roListe = liste.AsReadOnly();`
`liste[0] = 11;`
`foreach (var element in roListe)`
`    Console.WriteLine(element);`
	`11`
	`2`
	`3`

[1] https://learn.microsoft.com/en-us/dotnet/standard/collections/
https://learn.microsoft.com/en-us/archive/msdn-magazine/2017/march/net-framework-immutable-collections

11.7.1 Erstellen und erneuern

Die oben erwähnten unveränderlichen Kollektionsklassen **ImmutableList<T>** , …, **ImmutableQueue<T>** besitzen keine Konstruktoren. Stattdessen existiert jeweils eine zugehörige statische Klasse mit generischen Fabrikmethoden:

- **ImmutableList**
- **ImmutableArray**
- **ImmutableHashSet**
- **ImmutableSortedSet**
- **ImmutableDictionary**
- **ImmutableSortedDictionary**
- **ImmutableStack**
- **ImmutableQueue**

Diese Klassen besitzen eine (überladene) **Create()** – sowie eine **CreateRange()** – Methode, z. B. die Klasse **ImmutableList**:

- **public static ImmutableList<T> Create<T>()**
  Es wird eine leere Kollektion erzeugt, z. B.:
  ```
  var immList = ImmutableList.Create<string>();
  ```
- **public static ImmutableList<T> Create<T> (params T[]** *items***)**
  Es wird eine Kollektion mit einer initialen Liste von Elementen erzeugt, z. B.:
  ```
  var immList = ImmutableList.Create<string>("eins", "zwei", "drei");
  ```
- **public static ImmutableList<T> CreateRange<T> (IEnumerable<T>** *items***)**
  Es entsteht eine unveränderliche Kollektion, die initial die Elemente aus der per Parameter benannten **IEnumerable<T>** - Instanz enthält, z. B.:
  ```
  var ls = new List<string>() { "eins", "zwei", "drei" };
  var immList = ImmutableList.CreateRange<string>(ls);
  ```

Beim Aufruf der generischen Fabrikmethoden ist der Elementtyp anzugeben.

Über die Methode **CreateBuilder<T>()** erhält man von der statischen Generatorklasse zu einer unveränderlichen Kollektionsarchitektur ein Objekt, das sich für komplexere Initialisierungen eignet:

- Man kann es zunächst wie eine veränderliche Kollektion gestalten
- und dann über die Methode **ToImmutable()** auffordern, eine unveränderliche Kollektion zu erstellen.

Im folgenden Beispiel entsteht auf diese Weise eine unveränderliche Kollektion vom Typ **ImmutableList<String>**:

Quellcode	Ausgabe

`using System.Collections.Immutable;`
 	
`var ls = new List<string>() { "eins", "zwei", "drei", "vier"};`
`var builder = ImmutableList.CreateBuilder<string>();`
`builder.AddRange(ls);`
`builder.Remove("drei");`
`var immList = builder.ToImmutable();`
`foreach (var imm in immList)`
`  Console.WriteLine(imm);`
	`eins`
	`zwei`
	`vier`

Die statischen Generatorklassen zu den unveränderlichen Kollektionen bieten über Erweiterungsmethoden für die Schnittstelle **IEnumerable<T>** noch eine zusätzliche Option zum Erstellen von unveränderlichen Kollektionen. In der Klasse **ImmutableList** ist die generische Erweiterungsmethode **ToImmutableList<T>()** definiert:

public static ImmutableList<T> ToImmutableList<T>(this IEnumerable<T> *source***)**

Wie das folgende Beispiel zeigt, kann der Compiler beim Aufruf der generischen Erweiterungsmethoden die Konkretisierung des Typformalparameters aus dem Kontext ableiten:

```
var ls = new List<string>() { "eins", "zwei", "drei", "vier" };
var ils = ls.ToImmutableList();
```

Wird zu einer unveränderlichen Kollektion eine Variante mit anderer Zusammenstellung benötigt, dann ist eine Neukreation erforderlich. Die von veränderlichen Kollektionsklassen für solche Aufgaben gewohnten Methoden (z. B. **Add()**, **Remove()**) sind auch bei unveränderlichen Kollektionen verfügbar, zeigen aber ein angepasstes Verhalten, …

- indem sie eine *neue* Kollektion erstellen
- und per Rückgabe abliefern.

Im folgenden Beispiel entsteht ein neues **ImmutableList<string>** - Objekt mit einem Element weniger:

Quellcode	Ausgabe

`using System.Collections.Immutable;`
 	
`var immList = ImmutableList.Create<string>("eins", "zwei", "drei");`
`foreach (var ele in immList)`
`    Console.WriteLine(ele);`
`Console.WriteLine();`
`immList = immList.Remove("eins");`
`foreach (var ele in immList)`
`    Console.WriteLine(ele);`
	`eins`
	`zwei`
	`drei`
	
	`zwei`
	`drei`

Dass tatsächlich ein neues Objekt entsteht, belegen die folgenden Anweisungen:

```
var immList = ImmutableList.Create<string>("eins", "zwei", "drei");
var immList2 = immList.Remove("eins");
Console.WriteLine(immList == immList2); // Ausgabe: False
```

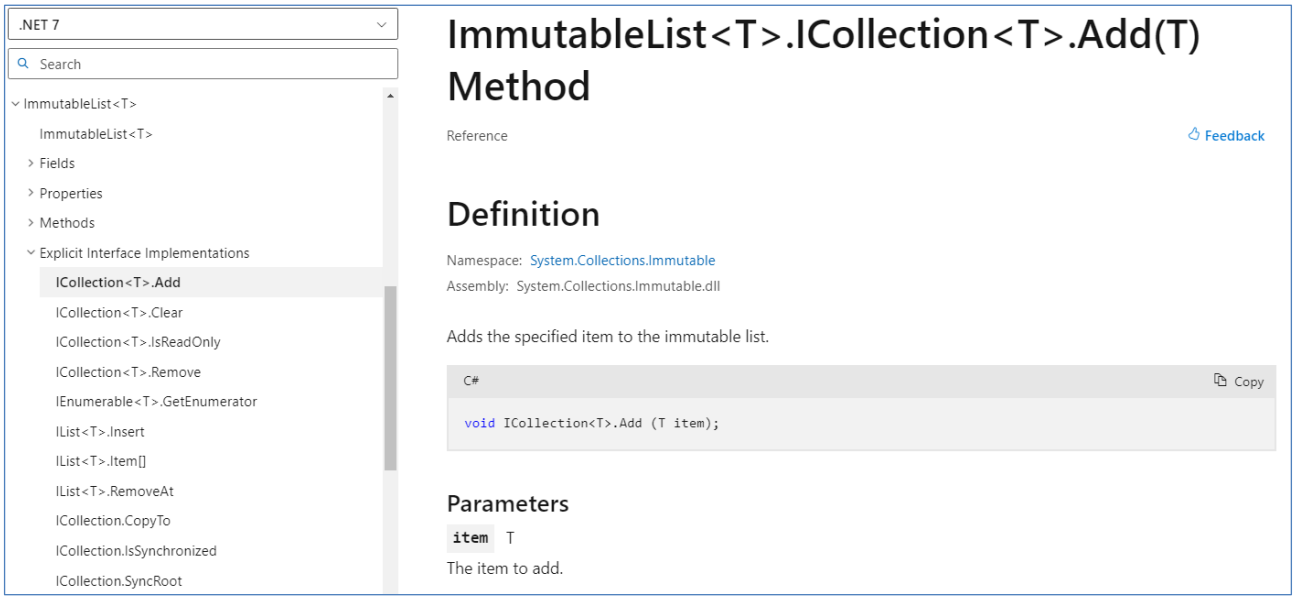
Weil die Klasse **ImmutableList<T>** die Schnittstelle **ICollection<T>** implementiert, muss sie eine Methode mit dem folgenden Definitionskopf besitzen:

public void Add (T *item***)**

Weil die eigene **Add()** – Methode mit **ImmutableList<T>** - Rückgabe

public ImmutableList<T> Add (T *value***);**

den Vorrang benötigt, und außerdem die beiden Methoden wegen identischer Signaturen nicht in einer Typdefinition koexistieren können, wird die **ICollection<T>** - Variante durch eine explizite Schnittstellenimplementierung versteckt (vgl. Abschnitt 9.4). Wie die Dokumentation der Klasse **ImmutableList<T>** zeigt, passiert das mit vielen Schnittstellenmethoden:



Eine unveränderliche Kollektion muss bei jeder „Veränderung“ erneuert werden, was auch bei den effizient programmierten BCL-Klassen einen relevanten Aufwand verursacht (vgl. Abschnitt 11.7.2). Um den Aufwand in Grenzen zu halten, kann man mit Hilfe eines Builder-Objekts mehrere Änderungen im Rahmen einer einzigen Renovierung erledigen:

- Man gewinnt aus einer unveränderlichen Kollektion durch einen Aufruf der Methode **ToBuilder()** ein Builder-Objekt.
- Dieses Builder-Objekt verhält sich wie eine veränderliche Kollektion, kann also entsprechende Methoden ausführen.
- Schließlich veranlasst man das Builder-Objekt durch einen Aufruf der Methode **ToImmutable()**, die Kollektion wieder in einen unveränderlichen Zustand zu bringen.

Diese Prozedur wird anschließend am Beispiel einer **ImmutableList<String>** - Kollektion vorgeführt:

Quellcode	Ausgabe

`using System.Collections.Immutable;`
 	
`var immList = ImmutableList.Create<string>("eins", "zwei", "drei");`
`var builder = immList.ToBuilder();`
`builder.Remove("eins");`
`builder.Add("vier");`
`immList = builder.ToImmutable();`
`foreach (var imm in immList)`
`    Console.WriteLine(imm);`
	`zwei`
	`drei`
	`vier`

11.7.2 Performanz

Wird von einer unveränderlichen Liste eine erweiterte Variante benötigt, dann muss ein neues Kollektionsobjekt erstellt werden. Bei Verwendung eines Arrays zum Speichern der Elemente entsteht ein erheblicher Aufwand, weil alle Elemente in ein neues Array zu kopieren sind. Wird hingegen ein Binärbaum oder eine verkettete Liste zur Speicherorganisation verwendet, dann können beim Erneuern einer unveränderlichen Kollektion wesentliche Teile der vorherigen Generation weiterverwendet werden, sodass der Renovierungsaufwand schrumpft.

Allerdings verursachen ein Binärbaum bzw. eine verkettete Liste im Vergleich zu einem Array deutlich mehr Aufwand beim lesenden Zugriff. Für Anwendungsfälle mit einer sehr geringen Wahrscheinlichkeit für Veränderungen und vielen Lesezugriffen befindet sich im BCL-Namensraum

System.Collections.Immutable die Struktur **ImmutableArray<T>**, die ein internes Array zum Speichern der Elemente verwendet. Das einzige Feld dieses Typs enthält eine Referenz auf das Array, und eine **ImmutableArray<T>** - Instanz benötigt folglich nicht mehr Platz als eine Speicheradresse. Daher hat man sich entschieden, den Typ als Struktur zu realisieren:[1]

```
public readonly partial struct ImmutableArray<T> :
    IReadOnlyList<T>,
    IList<T>,
    IEquatable<ImmutableArray<T>>,
    IList,
    IImmutableArray,
    IStructuralComparable,
    IStructuralEquatable,
    IImmutableList<T> {
 . . .
}
```

Um den Lese- bzw. Veränderungsaufwand bei einer **ImmutableList<String>** - sowie einer **ImmutableArray<String>** - Instanz im Vergleich zu einer (veränderlichen) **List<String>** - Instanz zu beobachten, wurden die drei Instanzen in einem Programm mit den folgenden Aufgaben konfrontiert:

- 100.000 Elemente aufnehmen
  Das erledigt bei den unveränderlichen Kollektionen ein Builder (siehe Abschnitt 11.7.1).
- 50.000 Elemente per Index abrufen
- 5.000 Erweiterungen der Kollektion vornehmen

Die folgenden Ergebnisse klare Anwendungsempfehlungen zu:[2]

```
List/Builder: System.Collections.Generic.List`1[System.String]
 Zeit zum Füllen:                          43 ms
 Zeit zum Abrufen:                          3 ms
 Zeit zum Erweitern:                        0 ms

List/Builder: System.Collections.Immutable.ImmutableList`1+Builder[System.String]
 Zeit zum Füllen:                          68 ms
 Zeit zum Abrufen:                         23 ms
 Zeit zum Erweitern:                       20 ms

List/Builder: System.Collections.Immutable.ImmutableArray`1+Builder[System.String]
 Zeit zum Füllen:                          19 ms
 Zeit zum Abrufen:                          1 ms
 Zeit zum Erweitern:                     2150 ms
```

Die unveränderliche **ImmutableArray<String>** - Instanz agiert beim Befüllen und beim Indexzugriff flotter als die veränderliche Kollektion, mit der sie den Array-Unterbau gemeinsam hat. Das Ändern einer **ImmutableArray<String>** – Instanz dauert aber so lange, dass eine Anwendung mit einer häufig zu ändernden Instanz dieses Typs viel Kritik auf sich ziehen würde. Sofern Kollektionsrenovierungen nur sehr selten stattfinden, ist die Struktur **ImmutableArray<T>** eine attraktive Option.

Die unveränderliche **ImmutableList<String>** - Instanz zeigt beim Füllen, Abrufen und Erweitern im Vergleich zu der veränderlichen Kollektion einen erhöhten Aufwand, doch fallen z. B. die für

---

[1] https://devblogs.microsoft.com/dotnet/please-welcome-immutablearrayt/
Der Quellcode stammt aus der BCL von .NET 7.

[2] Beobachtet auf einem Rechner unter Windows 10 (64 Bit) mit Intel-CPU Core i3 550. Ein Visual Studio - Projekt mit dem Testprogramm ist hier zu finden:
...\BspUeb\Kollektionen\ImmutableList-Performance

5.000 Erweiterungen der Kollektion benötigten 20 Millisekunden nicht störend auf. Bedenkt man zudem, dass eine unveränderliche Kollektion die Multithreading-Programmierung und damit die leistungsfördernde Verwendung der heutzutage universellen Mehrkern-CPUs erleichtert, dann ist bei vielen Anwendungen insgesamt eine Leistungssteigerung zu erwarten (Albahari 2022, S. 387).

11.8 Übungsaufgaben zum Kapitel 11

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Ein Objekt aus der Kollektionsklasse **List<T>** oder **LinkedList<T>** hält seine Elemente in sortiertem Zustand.
2. Die Klasse **HashSet<T>** bietet die beste Leistung bei Existenzprüfungen.
3. Die Klasse **List<T>** beherrscht zum Suchen und Sortieren dieselben Methoden wie ein Array.
4. Die zur Verwaltung von (Schlüssel-Wert) - Elementen dienende Klasse **Dictionary<K, V>** glänzt darin, dass zu einem Schlüssel sehr schnell der zugehörige Wert ermittelt werden kann.
5. Die Kollektionsklasse **SortedDictionary<K, V>** hält ihre Elemente in sortiertem Zustand.

2) Erstellen Sie eine Variante der im Abschnitt 5.11 vorgestellten Personenverwaltung, indem Sie die Klasse `PersonDB` (Eigenbau einer verketteten Liste mit Indexer) durch die generische BCL-Kollektionsklasse **List<T>** ersetzten.

12 GUI-Programmierung mit WPF-Technik

Mit den Eigenschaften und Vorteilen einer grafischen Bedienoberfläche (engl.: ***G**raphical **U**ser **I**nterface*) sind Sie sicher sehr gut vertraut. Eine GUI-Anwendung präsentiert dem Anwender standardisierte Bedienelemente zur Datenpräsentation und Benutzerinteraktion, z. B.:

- Texteingabefelder
- Befehlsschalter
- Kontrollkästchen und Optionsfelder
- Schieberegler und Auswahllisten
- Baumansichten für Dokumentenstrukturen etc.
- Tabellen zur Datenpräsentation
- Menüs
- Fortschrittsbalken
- Komponenten zur Präsentation von bildlichen und audio-visuellen Medien

Die von einer GUI-Bibliothek zur Verfügung gestellten Bedienelemente bezeichnet man als *Steuerelemente*, *controls* oder *widgets* (Wortkombination aus *window* und *gadgets*, die in dt. ungefähr als *Fenstergerät* zu übersetzen ist).

Von standardisierten Bedienelementen profitieren Entwickler und Anwender:

- Entwickler können dank direkt verwendbarer und dabei flexibel konfigurierbarer Komponenten die Bedienoberfläche einer Anwendung zügig aufbauen.
- Weil die Steuerelemente intuitiv (z. B. per Maus und/oder Tastatur) und in verschiedenen Programmen weitgehend konsistent zu bedienen sind, erleichtern sie dem Anwender den Umgang mit Software.

Die in der WPF-Bibliothek (***W**indows **P**resentation **F**oundation*) enthaltenen Steuerelemente erlauben durch ihre Vielfalt, ihre Konfigurierbarkeit und durch die Möglichkeiten zur flexiblen (z. B. hierarchisch strukturierten) Anordnung die Erstellung von individuellen, gut bedienbaren und ansehnlichen Bedienoberflächen für sehr viele Anwendungen.

Wenn sich eine Anwendung mit den fertigen WPF-Steuerelementen nicht realisieren lässt, dann stehen dem C# - Entwickler die folgenden Optionen zur Verfügung:

- Man kann die WPF-Toolbox um selbst entworfene Steuerelemente erweitern (siehe Beispiel im Abschnitt 10.2.3).
- Für die spezielle Präsentation und/oder Bearbeitung von zwei- oder dreidimensionalen Daten (z. B. bei einem Editor für Landkarten) ist eine individuelle Grafikprogrammierung erforderlich, die im Manuskript leider nicht behandelt werden kann.

Wir konzentrieren uns in diesem Manuskript auf einen Einstieg in die Erstellung von Bedienoberflächen mit Hilfe von fertigen Steuerelementen aus der WPF-Bibliothek. Weil .NET 7 (und vermutlich auch alle Nachfolger) die WPF-Technik nur unter Windows unterstützt, geht es im aktuellen Kapitel nur um Software für das Betriebssystem Windows.

Grundsätzlich ist das Erstellen einer GUI-Anwendung (für Windows) mit erheblichem Aufwand verbunden. Allerdings bietet die WPF-Technik leistungsfähige Klassen und Unterstützungsleistungen, z. B. die Option zur GUI-Deklaration in einem XML-Dialekt (*e**X**tensible **M**arkup **L**anguage*) namens *XAML* (*e**X**tensible **A**pplication **M**arkup **L**anguage*). Für eine weitere RAD-Beschleunigung (***R**apid **A**pplication **D**evelopment*) sorgen Assistenten im Visual Studio (z. B. der WPF-Fensterdesigner, den wir schon mehrfach benutzt haben).

Als WPF - Vorteile sind u. a. zu nennen:

- Es sind sehr viele Steuerelemente für diverse und komplexe Aufgaben vorhanden.
- Durch die Nutzung der XAML-Option lassen sich die Oberflächengestaltung und die Codierung der Programmlogik trennen. Das vereinfacht die GUI-Deklaration und erleichtert in großen Projekten die Kooperation von Software-Entwicklern und Grafikdesignern.
- Gute Verbindungen zwischen 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

Eine vollständige Behandlung der WPF-Bibliothek füllt eine umfangreiche Monografie (siehe z. B. Huber 2017, MacDonald 2012, Nathan 2010), sodass wir im Manuskript einige Themen auslassen müssen, z. B.:

- 2D- und 3D-Grafiken
- Stile[1]
- Animationen
- Text und Drucken[2]

12.1 Einordnung

In diesem Abschnitt werden wir …

- aktuell für C# - Anwendungen verfügbare GUI-Technologien diskutieren
- und GUI-Anwendungen mit Konsolenanwendungen vergleichen.

12.1.1 GUI-Technologien

Eine Windows-Anwendung sollte intuitiv bedienbar sein und eine attraktive Optik besitzen, um den Anwendern ein produktives Arbeiten sowie eine angenehme Freizeitgestaltung zu ermöglichen. Microsoft feilt ständig an der GUI-Technik, wobei eine Evolution mit gelegentlichen Fehlschlägen stattfindet.

12.1.1.1 WPF und WinForms

Die *Windows Presentation Foundation* (*WPF*) ist eine 2006 eingeführte GUI-Bibliothek für .NET - Anwendungen, die sich gegenüber ihrem Vorgänger *WinForms* durchgesetzt hat und aktuell (im Herbst 2023) bei der Entwicklung von Desktop-Anwendungen für Windows immer noch eine gute Wahl ist.[3] Ein Beispiel für attraktive WPF-Bedienoberfläche, die komplexen Aufgaben gerecht wird, ist im Visual Studio zu besichtigen.[4] Zwar wirken WPF-Anwendungen durch den Verzicht auf das Metro-Design (siehe Abschnitt 12.1.1.3) eher traditionell, doch tut das der Funktionalität keinen Abbruch, und die vertraute Bedienungslogik ohne Überraschungen wird vermutlich von vielen Anwendern geschätzt.

Während ein WinForms-GUI per Code erstellt werden muss, kann das GUI-Design einer WPF-Anwendung optional auch durch eine XML-Spezialisierung (*e**X**tensible **M**arkup **L**anguage*) namens

---

1 https://learn.microsoft.com/en-us/dotnet/desktop/wpf/controls/styles-templates-overview

2 https://learn.microsoft.com/en-us/dotnet/desktop/wpf/documents/printing-overview

3 In einer früheren Version dieses Manuskripts (über C#3.0, Baltes-Götz 2009) ist eine Behandlung der WinForms - GUI-Technik zu finden, siehe https://www.uni-trier.de/index.php?id=30454.

4 https://en.wikipedia.org/wiki/Visual_Studio

XAML (*eXtensible Application Markup Language*) deklariert werden. Die XAML-Option wird auch durch die anderen, anschließend beschriebenen GUI-Technologien für lokale Windows-Anwendungen unterstützt. Bei einem Umstieg bleibt also ein wesentlicher Teil des erworbenen GUI - Know-Hows erhalten.

12.1.1.2 UWP und WinUI 2

Mit Windows 8 hat Microsoft unter Bezeichnungen wie *Metro App*, *Modern App* oder *Store App* ein neues Design und Bedienkonzept für Windows-Anwendungen eingeführt. Es wurde ein neues Anwendungsmodell definiert und dabei (statt Win32-API) ein neues Laufzeitsystem namens *Windows-Runtime* (*WinRT*) verwendet, das Anwendungen in einem Sandkasten mit beschränktem Zugriff auf die Systemumgebung betreibt.[1]

Zusammen mit Windows 10 entstand daraus das UWP-Anwendungsmodell (***U**niversal **W**indows **P**latform*), das weiterhin auf dem Laufzeitsystem WinRT basiert.[2] UWP-Anwendungen lassen sich u. a. mit C# und anderen .NET - Sprachen entwickeln. Bei ihrer Einführung hatten sie wichtige Alleinstellungsmerkmale:[3]

- Vertrieb über den Windows-Store
- Unterstützung verschiedener Gerätegattungen mit Windows 10 als Betriebssystem (PC, Smartphone, Spielekonsole *Xbox*, Augmented-Reality - Brille *HoloLens*)
- Innovative Steuerelemente aus der Grafikbibliothek WinUI 2

Mittlerweile spielen die UWP-Anwendungen in Microsofts Zukunftsplänen keine große Rolle mehr, denn:

- Die in der Liste unterstützter UWP-Geräte lange Zeit in den Vordergrund gestellten Smartphones unter Windows sind vom Markt verschwunden.
- Die Entwickler von Windows-Desktopanwendungen zeigten wenig Interesse an der UWP und erstellten mehrheitlich weiterhin Anwendungen für das Win32-API (entweder nativ in C++ oder in einer .NET – Programmiersprache mit der WPF- oder WinForms-Grafikbibliothek).

Unter dem Codenamen *Reunion* hat Microsoft das *Windows App SDK* mit der Grafikbibliothek *WinUI 3* entwickelt, um die Windows-Bedienung im Metro-Design mit den Win32-API – Anwendungsmodellen zusammenzubringen (siehe Abschnitt 12.1.1.3).

Zwar sind das UWP-Anwendungsmodell und seine Grafikbibliothek WinUI 2 noch nicht als *veraltet* (engl.: *deprecated*) herabgestuft worden, doch empfiehlt Microsoft für neue Windows-Desktopanwendungen eindeutig das Windows App SDK mit der Grafikbibliothek WinUI 3, und die Grafikbibliothek von UWP-Anwendungen wird nicht über den Stand WinUI 2 hinauskommen.[4]

Neue UWP-Anwendungen sind nur dann sinnvoll, wenn es um Software für die Spielekonsole *Xbox*, die Augmented-Reality - Brille *HoloLens* oder das interaktive Whiteboard *Surface Hub* geht. Wesentliche Teile des im aktuellen Kapitel vermittelte GUI - Know-Hows lassen sich in diesem Fall weiterverwenden, weil auch in einer UWP-Anwendung die Bedienoberfläche bevorzugt mit dem XML-Dialekt XAML deklariert wird (Schacherl 2014, Abschnitt 1.3.7).

---

1 https://en.wikipedia.org/wiki/Features_new_to_Windows_8

2 https://en.wikipedia.org/wiki/Universal_Windows_Platform_apps

3 https://learn.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide

4 https://learn.microsoft.com/en-us/windows/apps/get-started/windows-developer-faq
Wie diese Webseite erläutert, wird die zusammen mit der Metro- bzw. WinUI-Bedienung entwickelte Windows Runtime (WinRT) jetzt vom Windows App SDK genutzt.

12.1.1.3 Windows App SDK und WinUI 3

Zur Überwindung der Trennung von Win32- und UWP-Anwendungen hat Microsoft im Jahr 2021 das *Windows App SDK* (früherer Name: *Project Reunion*) mit der Grafikbibliothek WinUI 3 als Kernbestandteil präsentiert.[1] Man kann nun eine Modern App ohne UWP erstellen.

Das Windows App SDK hat eine von seinem Hauptbestandteil WinUI abweichende Versionierung und ist im Oktober 2023 auf dem Stand 1.4.2 angekommen. Es unterstützt derzeit Windows 11 sowie Windows 10 ab Version 1809, sodass ca. 95% aller aktiven Windows-Rechner versorgt werden können.[2]

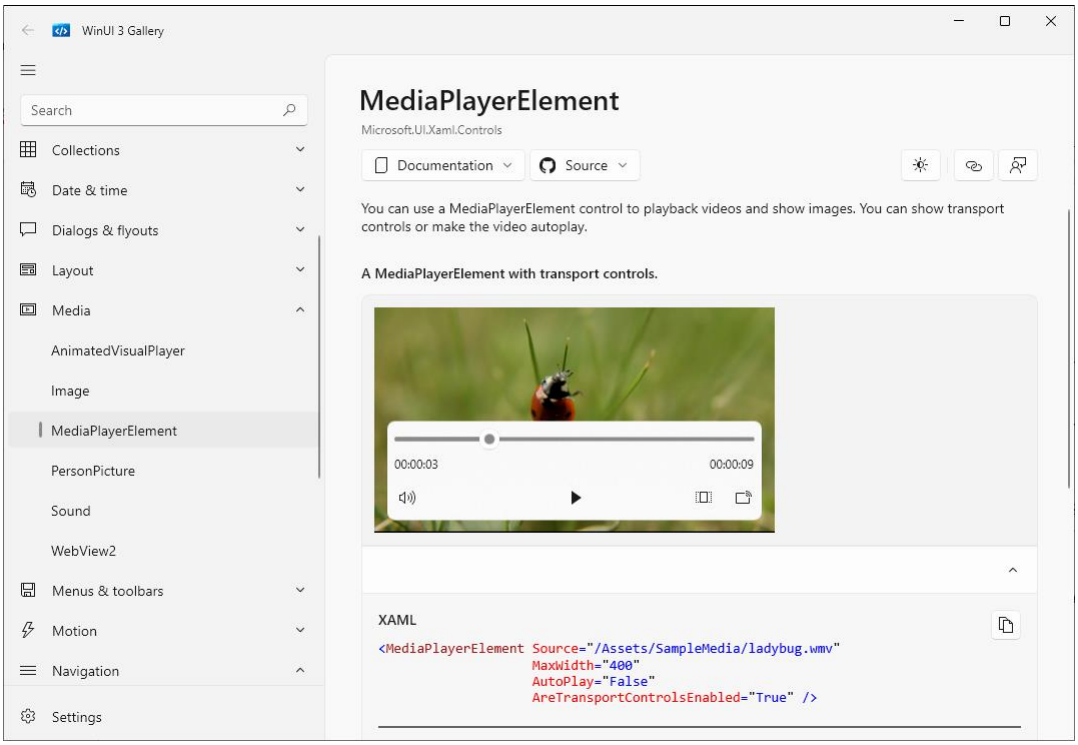
Hier findet sich eine Liste mit den WinUI-Steuerelementen:

https://learn.microsoft.com/en-us/windows/apps/design/controls/#alphabetical-index

Im Microsoft Store wird das kostenlose, per Windows App SDK 1.4 erstellte Programm **WinUI 3 Gallery** angeboten,

https://apps.microsoft.com/store/detail/winui-gallery/9P3JFPWWDZRC

das die WinUI 3 – Steuerelemente im Einsatz mitsamt dem zugehörigen XAML/C# - Quellcode zeigt.[3] Hier wird das Steuerelement zum Abspielen von Medien vorgeführt:



Naheliegenderweise verwendet die Gallery-Anwendung das WinUI 3 für seine Bedienoberfläche, wobei auf der obersten Ebene das **NavigationView**-Steuerelement zum Einsatz kommt.

Neben dem WinUI 3 enthält das Windows App SDK noch weitere Funktionen, die aber nur für spezielle Anwendungen relevant sind, z. B.

---

1 https://en.wikipedia.org/wiki/Windows_App_SDK

2 https://gs.statcounter.com/windows-version-market-share/desktop/worldwide/

3 Offenbar sind manche Metro-Steuerelemente aktuell (im Oktober 2023) nur für WinUI 2 -Anwendungen (also für UWP-Anwendungen) verfügbar (z. B. das Map Control zur Darstellung von Landkarten). Welche Steuerelemente tatsächlich schon im Windows App SDK vorhanden sind, verrät die WinUI 3 Gallery.

- Benachrichtigungen
  Einer Anwendung kann z. B. eine Push-Nachricht über die Aktualisierung von Server-Daten zugestellt werden, sodass keine regelmäßige Nachfrage der Anwendung (Polling) erforderlich ist.
- Ereignisse mit Informationen über den Akku und die Stromversorgung

Weitere Informationen über das Windows App SDK und die WinUI 3 – Grafikbibliothek bietet z. B. Ashcraft (2023).

Microsoft favorisiert aktuell für Windows-Desktopanwendungen das WinUI 3, versichert aber, dass die WPF-Technik auch in Zukunft unterstützt und weiterentwickelt wird. Wenn wir uns vorläufig auf die WPF-Technik beschränken, dann befinden wir uns nicht in einer Sackgasse oder auf einem Irrweg. Um das WinUI 3 in einer eigenen Anwendung zu nutzen, muss man lediglich die erforderlichen NuGet-Pakete einbeziehen. Eine wichtige Gemeinsamkeit von WPF und WinUI 3 besteht in der Option zur GUI-Deklaration per XAML.

12.1.1.4 MAUI

Seit .NET 6 sind C# - Anwendungen mit grafischer Bedienoberfläche möglich, die mit identischem Quellcode unter Android, iOS, macOS und Windows laufen. Diese simultane Unterstützung von Betriebssystemen für mobile Geräte und Arbeitsplatzrechner wird mit einer Technik namens *.NET MAUI* realisiert, die aus dem Produkt *Forms* der Firma Xamarin hervorgegangen ist.[1] Unter Windows resultiert eine Anwendung mit der WinUI 3 – Grafikbibliothek, wobei aber die Steuerelemente aus dem Windows App SDK (siehe Abschnitt 12.1.1.3) noch nicht zur Verfügung stehen.

Ist eine .NET - Multi-Plattform - Anwendung mit grafischer Bedienoberfläche geplant, dann bietet .NET MAUI die Chance zur Vermeidung von Parallelentwicklungen. Soll eine Anwendung hingegen nur unter Windows laufen, dann ist wegen der besseren Funktionalität und Kompatibilität eine auf Windows spezialisierte Grafikbibliothek zu bevorzugen (z. B. WPF oder WinForms). Microsoft empfiehlt in dieser Lage das Windows App SDK.[2]

12.1.1.5 Verteilte Anwendungsarchitekturen

Wir beschränken uns im Manuskript auf *lokale* Anwendungen, ignorieren also die verteilten Anwendungsarchitekturen mit …

- der Geschäftslogik (vorzugsweise erstellt mit ASP.NET Core) und der Datenverwaltung auf einem Server bzw. auf mehreren Servern
- und einer klientenseitigen Bedienoberfläche in einem Web-Browser.

Das Erstellen („rendern“) der im Browser-Rahmen ausgeführten Bedienoberfläche unter Verwendung von offenen Standards (HTML, CSS) kann auf dem Server, auf dem Klienten oder kooperativ erfolgen.

Lokal installierte Anwendungen für Desktop-Rechner unter Windows stehen durch das Aufkommen von Internet-basierten Anwendungen unter einem wachsenden Konkurrenzdruck, bieten aber immer noch einige Vorteile, z. B.:

---

[1] https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui

[2] https://learn.microsoft.com/en-us/windows/apps/windows-dotnet-maui/

- Unabhängigkeit vom Internet
- Überlegenheit beim wichtigsten Ergonomiekriterium: der Reaktionszeit
- Uneingeschränkte Nutzung der PC-Hardware
- Integration mit dem Windows-Betriebssystem und mit anderen installierten Anwendungen

Komplett lokal ausgeführte Single Page Web-Apps (z. B. in C# realisiert mit ASP.NET Core Blazor) reagieren nach dem Herunterladen ebenfalls flüssig, sind aber eher für einfache Aufgabenstellungen geeignet. Wenn sich ein Blazor WebAssembly nicht z. B. wegen seiner Vorteile bei der Software-Verteilung aufdrängt, dann lässt sich derselbe Zweck durch eine lokale Anwendung mit WPF-GUI einfacher erreichen.

12.1.2 Vergleich zwischen GUI- und Konsolenanwendungen

Im Vergleich zu Konsolenprogrammen geht es bei GUI-Anwendungen nicht nur intuitiver, sondern vor allem auch ereignisreicher[1] und mit mehr Mitspracherechten für die 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. Ausgelöst werden die Ereignisse in der Regel vom Benutzer, der mit der Hilfe von Eingabegeräten wie Maus, Tastatur, Touchscreen etc. praktisch permanent in der Lage ist, unterschiedliche Wünsche zu artikulieren. Ein GUI-Programm präsentiert mehr oder weniger viele Bedienelemente, die dem Anwender das Auslösen von Ereignissen ermöglichen. Das Programm wartet die meiste Zeit darauf, auf ein vom **Benutzer** ausgelöstes **Ereignis** mit einer vorbereiteten Behandlungsmethode zu reagieren.

Im Vergleich zu einem Konsolenprogramm ist bei einem GUI-Programm die dominante Richtung im Kontrollfluss zwischen Anwendung und Laufzeitsystem invertiert. Die Ereignisbehandlungsmethoden einer GUI-Anwendung sind Beispiele für sogenannte *Call Back - Routinen*. Man spricht auch vom *Hollywood-Prinzip*, weil in dieser Gegend oft nach der Devise kommuniziert wird: „*Don't call us. We call you*".

Während sich ein Konsolenprogramm gegenüber dem Anwender autoritär und gegenüber dem Laufzeitsystem fordernd verhält, präsentiert ein GUI-Programm dem Anwender Service-Angebote und befolgt kooperativ die Anweisungen des Laufzeitsystems:

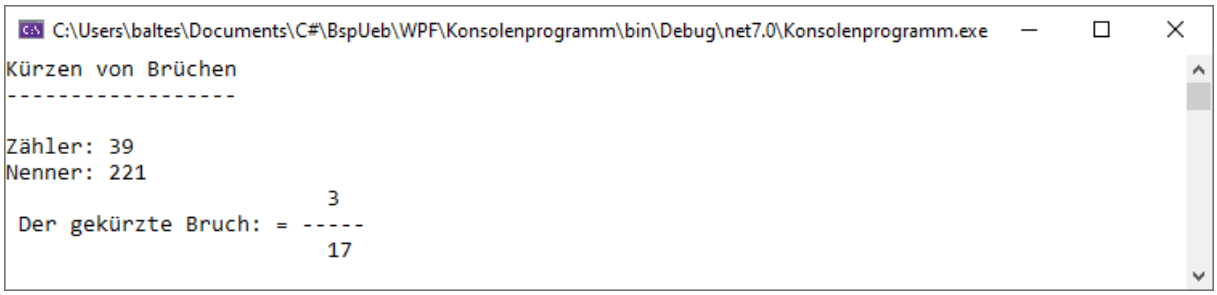
- Eine Konsolenanwendung diktiert den Ablauf und erlaubt dem Benutzer gelegentlich eine Eingabe. Um seinen Job erledigen zu können, verlangt das Programm Dienstleistungen vom Laufzeitsystem, z. B.: „Bitte den nächsten Tastendruck übermitteln." Das Laufzeitsystem erledigt solche Anforderungen und gibt die Kontrolle dann wieder an die Konsolenanwendung zurück. Eine Konsolenanwendung benimmt sich so, als wäre sie das einzige Anwendungsprogramm und hätte das Laufzeitsystem exklusiv als Dienstleister zur Verfügung.

---

[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. Im Abschnitt 16.6.3 von Baltes-Götz (2021) wird z. B. ein Konsolenprogramm erstellt, das auf Ereignisse im Dateisystem (z. B. auf das Erstellen, Umbenennen oder Löschen von Dateien) reagiert.

- Eine GUI-Anwendung besteht hingegen aus einer Sammlung von Ereignisbehandlungsmethoden, wobei die zugehörigen Ereignisse meist vom Benutzer ausgelöst werden. Die Ereignisse werden zunächst vom Laufzeitsystem registriert, das daraufhin Methoden des GUI-Programms aufruft.

Betrachten wir zur Illustration eine Konsolen- und eine GUI-Anwendung zum Kürzen von Brüchen. Das Konsolenprogramm

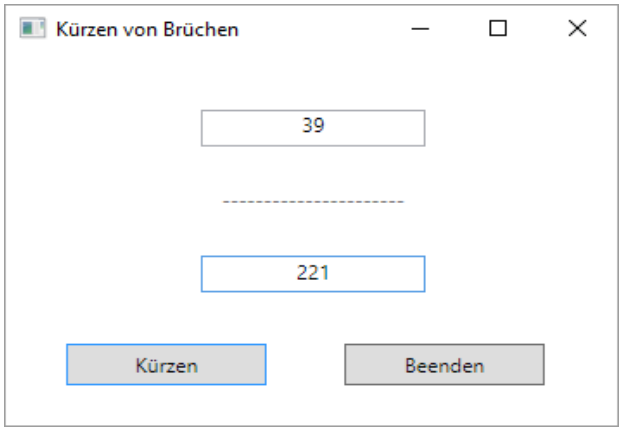


diktiert den gesamten Ablauf:

- Es fragt nach dem Zähler.
- Es fragt nach dem Nenner.
- Es schreibt das Ergebnis auf die Konsole.

Wenn der Benutzer z. B. nach der Eingabe des Nenners den Zähler noch einmal ändern möchte, dann muss er das Programm beenden und neu starten.

Im Unterschied zu diesem **programmgesteuerten Ablauf** wird bei der (im Abschnitt 5.13 erstellten) GUI-Variante



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**).

12.2 Essenzielle Klassen einer WPF-Anwendung

Dieser Abschnitt informiert anhand eines einfachen Beispiels über die bei einer WPF-Anwendung beteiligten Klassen. Außerdem wird das bei modernen GUI-Bibliotheken übliche Single-Thread – Prinzip erläutert.

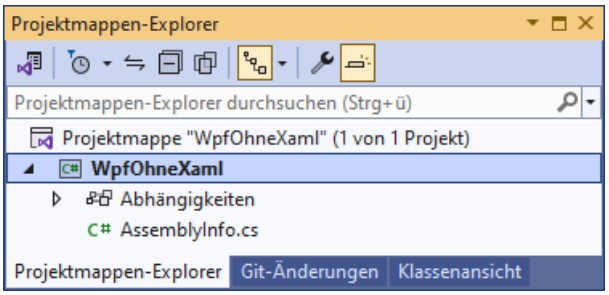
12.2.1 Eine minimalistische WPF-Anwendung ohne XAML

Um die essenziellen Klassen und den Start einer WPF-Anwendung zu studieren, vermeiden wir zunächst die XAML-Beteiligung und die damit verbundenen Hintergrundaktivitäten. Stattdessen betrachten wir ein sehr einfaches und komplett durch C# - Anweisungen realisiertes Programm. Wir legen im Visual Studio ein neues Projekt mit dem Namen `WpfOhneXaml` unter Verwendung der

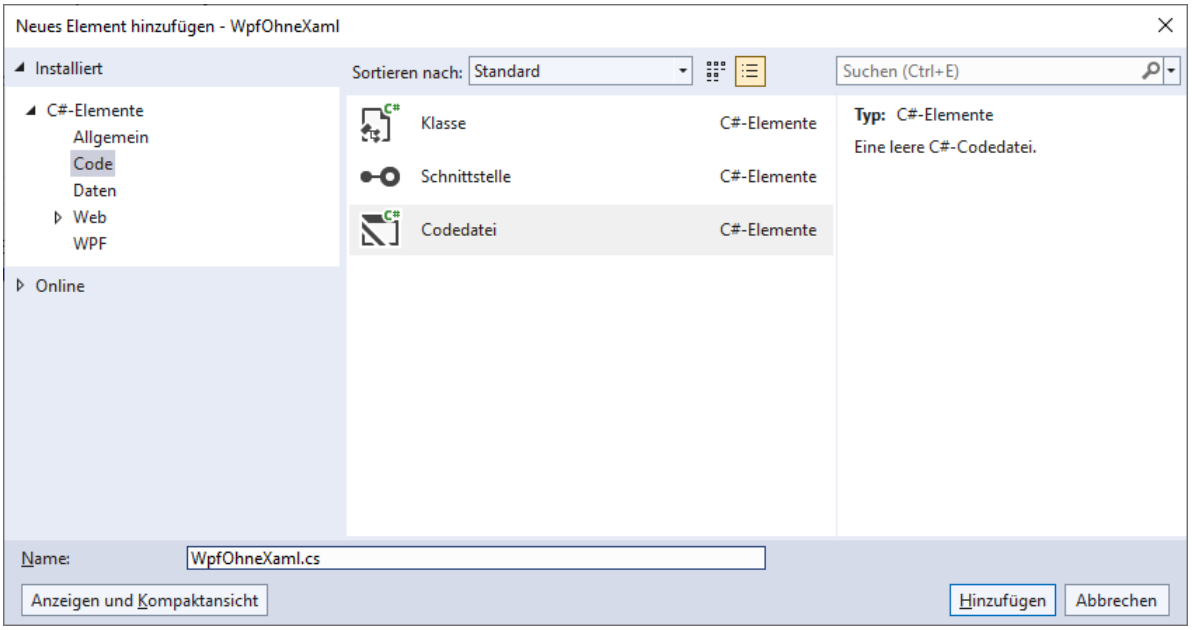
Vorlage **WPF-Anwendung** an und entfernen mit dem **Projektmappen-Explorer** die folgenden automatisch generierten Bestandteile:

- **App.xaml** (samt der zugehörigen Quellcodedatei **App.xaml.cs**) und
- **MainWindow.xaml** (samt der zugehörigen Quellcodedatei **MainWindow.xaml.cs**)

Anschließend sollte der **Projektmappen-Explorer** das folgende Bild zeigen:



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**:

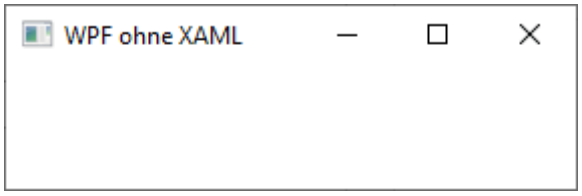
```
using System.Windows;
namespace WpfOhneXaml;

class WpfOhneXaml : Window {
   WpfOhneXaml() {
      Title = "WPF ohne XAML";
      Width = 300;
      Height = 100;
   }

   [System.STAThread]
   static void Main() {
      var app = new Application();
      var hf = new WpfOhneXaml();
      app.Run(hf);
   }
}
```

Im Beispiel wird die seit C# 10 erlaubte **namespace**-Direktive *ohne* geschweiftes Klammernpaar genutzt, wobei eine dateiglobale Gültigkeit resultiert (siehe Abschnitt 2.6.1).

Auch ein GUI-Programm besteht aus Klassen, wobei eine Startklasse mit einer statischen **Main()** - Methode vorhanden sein muss (vgl. Abschnitt 1.5). 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. Gleich ist zu sehen, dass dabei Basisklassen aus der WPF-Bibliothek zentrale Rollen spielen.

Bei einer WPF-Anwendung muss die Methode **Main()** mit dem Attribut **System.STAThread** dekoriert werden. Attribute sind Objekte, die mit einer speziellen Syntax (siehe Beispiel) an Klassen, Methoden etc. geheftet werden, um Informationen für den Compiler und/oder die CLR bereitzustellen.[1] Nähere Informationen folgen im Kapitel 14. Im konkreten Fall resultiert gegenüber dem Windows-Betriebssystem eine Deklaration zum Modus der Interprozesskommunikation per COM (***C**omponent **O**bject **M**odel*).[2] Eine WPF-Anwendung benötigt die COM-Interoperabilität z. B. beim Zugriff auf die Windows-Zwischenablage.

Ein allzu großer Funktionsumfang ist von dem minimalistischen Programm nicht zu erwarten:



Immerhin kann man das Anwendungsfenster (dank Windows und .NET) verschieben, seine Größe ändern, die Titelzeilen-Standardschaltflächen zum Minimieren, Maximieren und Beenden benutzen usw.

Im aktuellen Kapitel 12 werden Sie die folgenden Klassen als wichtige Bestandteile einer WPF-Anwendung kennenlernen:

- **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 aus dem Namensraum **System.Windows.Controls** für Steuerelemente (z. B. **Label**, **Button**, **TextBox**) und Layoutcontainer zur Verwaltung von Steuerelementen (z. B. **Grid**, **StackPanel**)
- **RoutedEvent** (Namensraum **System.Windows**)
  Für WPF-Anwendungen wurde mit den sogenannten *Routingereignissen* eine Ergänzung bzw. Erweiterung der im Kapitel 10 beschriebenen CLR-Ereignistechnik eingeführt. Ein Routingereignis basiert auf einem Objekt der Klasse **RoutedEvent**.

Im weiteren Verlauf des Abschnitts 12.2 werden wir uns mit den Klassen **Window** und **Application** beschäftigen.

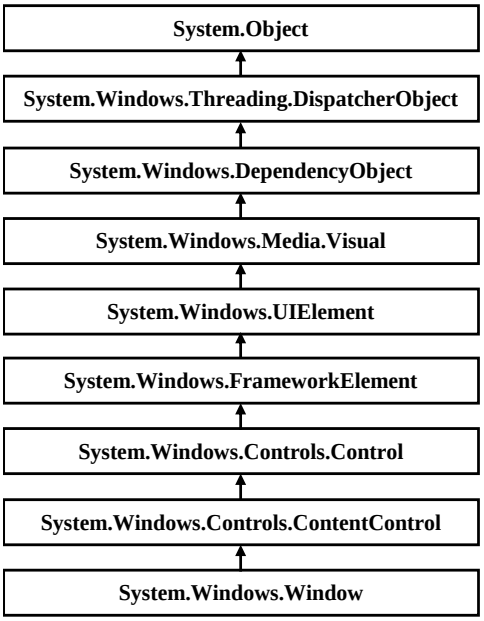
---

[1] Die aktuelle beteiligte Attributklasse heißt eigentlich **STAThreadAttribute**, doch man darf den Namensbestandteil *Attribute* weglassen, was in der Regel auch geschieht.

[2] https://docs.microsoft.com/en-us/windows/desktop/com/single-threaded-apartments

12.2.2 (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 wichtige WPF-Klassen auf, und es ist nicht nur im Hinblick auf die Klasse **Window** sinnvoll, deren Zuständigkeiten zu erläutern:[1]

- **System.Windows.Threading.DispatcherObject**
  Um eine konsistente und verzögerungsfrei reagierende Bedienoberfläche zu garantieren, verwendet das WPF-Framework die Single-Thread - Architektur: Auf ein UI-Element darf nur derjenige Thread zugreifen, der das Element erzeugt hat (siehe Abschnitt 12.2.4). Um dies sicherzustellen, stammen die meisten WPF-Klassen von der Klasse **DispatcherObject** ab und sind daher dem Dispatcher (dt.: *Verteiler*) des erzeugenden Threads (einem Objekt der Klasse **Dispatcher** im Namensraum **System.Windows.Threading** fest zugeordnet. Das **Dispatcher**-Objekt ist in einem Thread für die Verwaltung der Warteschlange mit zu erledigenden Aufgaben (Methodenaufrufen) zuständig.

  In einer von **DispatcherObject** abstammenden Klasse (also in den meisten WPF-Klassen, z. B. auch in der Klasse zum Hauptfenster einer Anwendung) kann eine Methode vor einem Zugriff auf ein **DispatcherObject** leicht überprüfen, ob sie in demjenigen Thread ausgeführt wird, der das **DispatcherObject** erstellt hat. Dazu steht in der Klasse **DispatcherObject** die Methode **VerifyAccess()** zur Verfügung, die bei einer verletzten Thread-Bedingung eine **InvalidOperationException** wirft. Im Abschnitt 10.2.1 sind **VerifyAccess()** - Aufrufe in Methoden der BCL-Klasse **Application** zu sehen, die ebenfalls von **DispatcherObject** abstammt.

[1] Weitere Details bietet die Webseite:
https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/wpf-architecture

- **System.Windows.DependencyObject**
  Objekte dieser Klasse können *Abhängigkeitseigenschaften* (engl.: *dependency properties*) anbieten (siehe Abschnitt 12.5). Diese spielen in der WPF-Technik eine zentrale Rolle und haben viele Vorteile im Vergleich zu gewöhnlichen CLR-Eigenschaften (siehe Abschnitt 12.5). Die Ausprägung einer Abhängigkeitseigenschaft kann z. B. durch verschiedene Quellen beeinflusst werden (z. B. durch ein elterliches Steuerelement, Datenbindung, WPF-Stile oder Animationen). Die meisten Eigenschaften von WPF-Steuerelementen sind als Abhängigkeitseigenschaften realisiert, können aber auch wie gewöhnliche CLR-Eigenschaften verwendet werden.
- **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 für die Layout-, Ereignis- und Fokusbehandlung bei.
- **System.Windows.FrameworkElement**
  Diese Klasse ist bei vielen WPF-Techniken beteiligt, z. B. Layout, Datenbindung, Stile und Animationen.
- **System.Windows.Controls.Control**
  Dies ist die Basisklasse für Bedienoberflächenelemente, die ihre Optik und ihr Verhalten über ein **ControlTemplate**-Objekt steuern.
- **System.Windows.Controls.ContentControl**
  Von **ContentControl** abgeleitete Steuerelementklassen enthalten ein einziges Objekt, das über die Eigenschaft **Content** vom Typ **Object** festgelegt wird. Neben der Klasse **Window**, die einen Layoutcontainer (z. B. aus der Klasse **Grid**) enthält, stammt u. a. auch die Klasse **Label**, die ein **String**-Objekt enthält, von der Klasse **ContentControl** ab.

Das im Abschnitt 12.2.1 vorgestellte Programm besteht aus der von **Window** abgeleiteten Klasse `WpfOhneXaml`

```
class WpfOhneXaml : Window { ... }
```

und erzeugt in seiner **Main()** - Methode ein Objekt aus dieser Klasse:

```
var hf = new WpfOhneXaml();
```

Als Aktualparameter im Methodenaufruf

```
app.Run(hf);
```

(gerichtet an das **Application**-Objekt `app`, das wir als **Anwendungsobjekt** bezeichnen)

```
var app = new Application();
```

wird das `WpfOhneXaml`-Objekt `hf` zum Vertreter des Anwendungs- oder Hauptfensters im Programm. Man kann `hf` als ein **Fensterobjekt** bezeichnen.

Unter den Fenstern eines Programms zeichnet sich das Hauptfenster durch die folgenden Besonderheiten aus:

- Über den an das Anwendungsobjekt gerichteten (und nicht wiederholbaren) **Run()** - Aufruf wird für die Anzeige des Hauptfensters gesorgt. Sein Auftritt muss also *nicht* über die **Window**-Methode **Show()** veranlasst werden. Mit der folgenden Alternative

```
[System.STAThread]
static void Main() {
    var hf = new WpfOhneXaml();
    hf.Show();
}
```

für die **Main()** – Methode im Abschnitt 12.2.1 resultiert ein sehr kurzer Programmeinsatz mit einem kurz aufblitzenden Fenster:

  - Die Methode **Show()** zeigt das Fenster an und kehrt sofort zurück.
  - Damit endet auch die Methode **Main()** und somit das Programm.

- In der Regel werden erhebliche Teile der Programm-Funktionalität im Hauptfenster angeboten.
- Beim Schließen des Hauptfensters wird das Programm beendet, wenn …
  - die **Application**-Eigenschaft **ShutdownMode** den Wert **OnMainWindowClose** besitzt (vgl. Abschnitt 12.2.3),
  - die **Application**-Eigenschaft **ShutdownMode** den voreingestellten Wert **OnLastWindowClose** besitzt, und nur das Hauptfenster vorhanden ist.

Unsere Beispielprogramme kommen meist mit einem einzigen Fenster aus. Selbstverständlich kann eine WPF-Anwendung auch *mehrere* Fenster verwenden.

Unsere Beispielprogramme haben also ein Anwendungsobjekt und ein Fensterobjekt. In der Regel stammt …

- das Anwendungsobjekt aus einer von **Application** abgeleiteten Klasse
- und das Fensterobjekt aus einer von **Window** abgeleiteten Klasse.

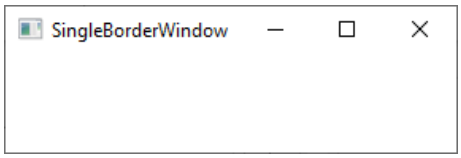
Für die Funktionalität und die Gestaltung ist das Fensterobjekt wichtiger, sodass auch für das minimalistische Beispiel im Abschnitt 12.2.1 eine eigene Fensterklasse aus **Window** abgeleitet werden musste, während keine eigene Anwendungsklasse erforderlich war.

So wie im Konstruktor der Klasse `WpfOhneXaml` mit der Anweisung

```
Title = "WPF ohne XAML";
```

die Titelzeilenbeschriftung des Hauptfensters über die **Window**-Eigenschaft **Title** festgelegt 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** mit der gleichnamigen Enumeration (im Namensraum **System.Windows**) als Datentyp beeinflusst man die Ausstattung der Titelzeile mit Standardschaltflächen und die Rahmengestaltung, z. B.

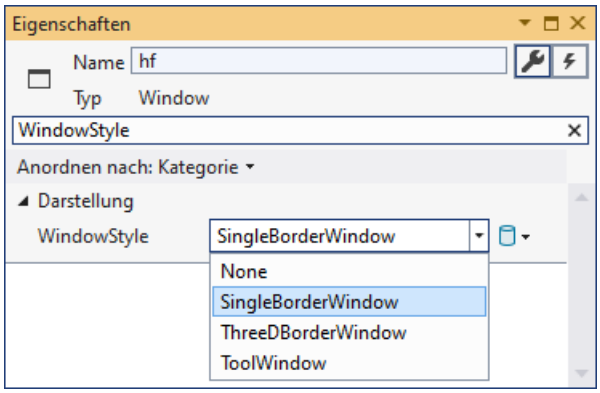


SingleBorderWindow

ToolWindow

None

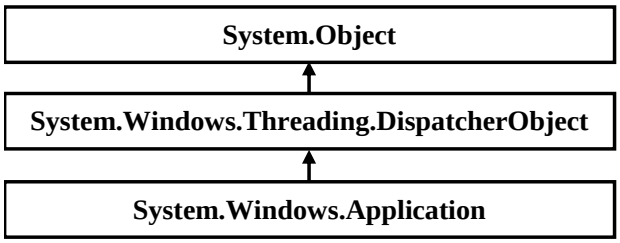
Beim Einsatz des WPF-Designers, mit dem wir schon etliche Erfahrungen gesammelt haben, ist die Eigenschaftsmodifikation zur Entwurfszeit mit dem **Eigenschaften**-Fenster bequem zu erledigen, z. B.:



Wir verzichten momentan auf diesen Komfort, um die Grundstruktur einer WPF-Anwendung an einem rudimentären Beispiel studieren zu können.

12.2.3 Nachrichtenverabeitung und die Klasse Application

Eine WPF-Anwendung benötigt ein Objekt aus der Klasse **Application** oder aus einer Spezialisierung von **Application**, das im Manuskript als *Anwendungsobjekt* bezeichnet wird. Im Vergleich zur Klasse **Window**, die für ein Fenster einer WPF-Anwendung (z. B. für das Hauptfenster) zuständig ist, hat die Klasse **Application** einen kleinen Stammbaum:



Das **Application**-Objekt einer Anwendung erbringt u. a. die folgenden Leistungen:

- Es löst wichtige Ereignisse im Lebenslauf einer Programminstanz aus, auf die Sie eventuell mit einer Behandlungsmethode reagieren möchten:
  - **Startup**
    Das Ereignis wird in der Startphase ausgelöst. Eine Behandlungsmethode eignet sich u. a. dazu, um Befehlszeilenargumente über die **Environment**-Methode **GetCommandLineArgs()** abzurufen und dann auszuwerten, z. B.:

```
using System;
using System.Windows;

namespace ApplicationStartup;

public partial class App : Application {
  private void Application_Startup(object sender, StartupEventArgs e) {
    string[] args = Environment.GetCommandLineArgs();
    for (int i = 1; i< args.Length; i++)
      MessageBox.Show("Argument " + i +": "+ args[i], "Application Startup");
  }
}
```

    Die Ereignisbehandlungsmethode muss registriert werden, was z. B. in der XAML-Datei zum Anwendungsobjekt geschehen kann:

```
<Application x:Class="ApplicationStartup.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:ApplicationStartup"
             StartupUri="MainWindow.xaml" Startup="Application_Startup">
    <Application.Resources>
    </Application.Resources>
</Application>
```

  - **Activated**, **Deactivated**
    Das Programm (d. h. eines seiner Fenster) ist in den Vordergrund geholt bzw. von dort verdrängt worden.
  - **DispatcherUnhandledException**
    Wenn im UI-Thread, der sich um die Bedienoberfläche und die Interaktion mit dem Benutzer kümmert (siehe Abschnitt 12.2.4), eine unbehandelte Ausnahme auftritt, dann 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 (siehe Abschnitt 13.2.3).[1]
  - **SessionEnding**
    Der Benutzer ist im Begriff, seine Windows-Sitzung zu beenden (Abmelden des Benutzers oder Herunterfahren des Rechners).
  - **Exit**
    Das Programm sieht seinem unmittelbar bevorstehenden (und nicht mehr zu stoppenden) Ende entgegen. Wie man auf dieses Ereignis reagieren kann, wurde schon im Abschnitt 10.2.2 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.
- Die **Run()** - Methode des **Application**-Objekts
  public int Run(Window *window***)**
  setzt die Verarbeitung der für eine Anwendung relevanten Nachrichten bzw. Ereignisse in Gang. Damit werden wir uns im weiteren Verlauf des aktuellen Abschnitts näher beschäftigen.

Durch die von Windows registrierten Benutzeraktivitäten (z. B. Mausklicks, Tastenschläge) und sonstige Ursachen entstehen **Ereignisse** (im Sinn 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 zu zeichnen. Um diese laufend eintreffenden und in eine Warteschlange eingereihten Nachrichten kümmert sich bei einer WPF-Anwendung eine über die **Application**-Instanzmethode **Run()** gestartete Routine des Laufzeitsystems in einer **while**-Schleife. Hat der Programmierer zu einer Nachricht eine Behandlungsmethode erstellt und registriert, dann wird diese aufgerufen. Man kann ein GUI-Programm als eine 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.[2]

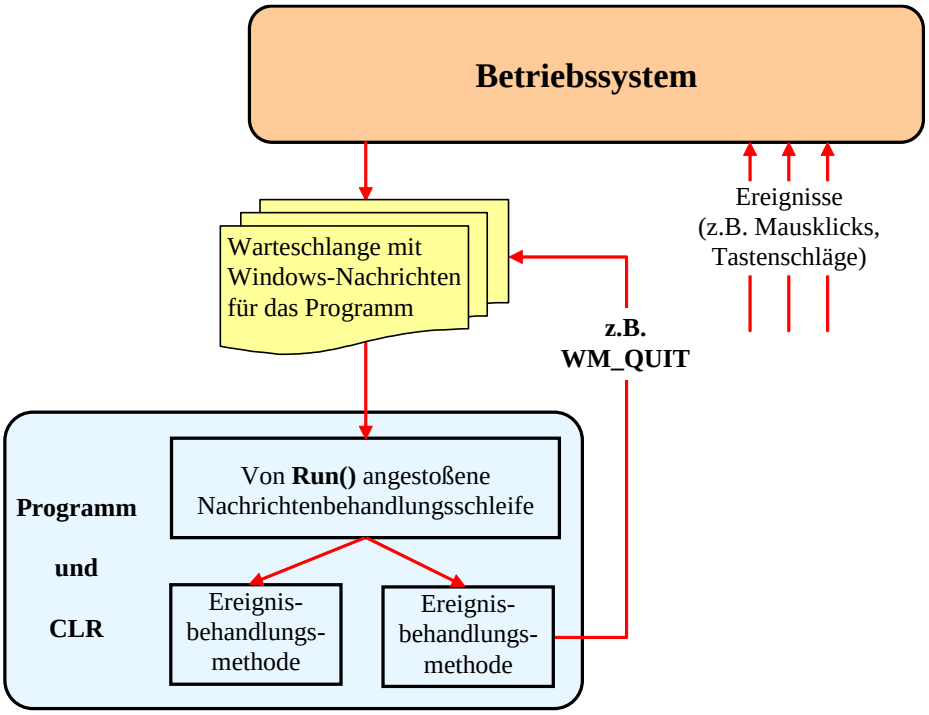
---

[1] In einer WPF-Anwendung muss man sich ggf. auch um unbehandelte Ausnahmen in einem Hintergrund-Thread kümmern, der für aufwändige Berechnungen zuständig ist, die aus dem UI-Thread herausgehalten werden müssen (siehe Abschnitt 17.4.6 in Baltes-Götz (2021)).

[2] WPF-Anwendungen, die *mehrere* Top-Level - Fenster besitzen und dementsprechend mehrere UI-Threads (jeweils mit einem eigenen Dispatcher) zur Bedienung der Fenster einsetzen, werden im Manuskript nicht behandelt (siehe https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/threading-model).

Zu manchen Nachrichten werden von Windows oder von der CLR (Common Language Runtime) ohne Zutun des Programmierers Behandlungsroutinen bereitgestellt. So kann unser Beispielprogramm z. B. auf die Standardschaltflächen in der Titelzeile (zum Maximieren, Minimieren 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:



Für den ereignisorientierten Ablauf eines WPF-Programms ist die von **Run()** angestoßene Nachrichtenbehandlungsschleife verantwortlich.

Unter den (rot gezeichneten) Nachrichtenströmen ist der nach ober gerichtete vermutlich etwas unerwartet. Dass eine Anwendung Nachrichten in die eigene Warteschlage platziert, ist aber keinesfalls ungewöhnlich.

Wird die Windows-Nachricht **WM_QUIT** aus der Warteschlange gefischt, dann endet die Nachrichtenbehandlungsschleife, und der **Run()** - Aufruf kehrt zurück.[1]

Verantwortlich für die Nachricht **WM_QUIT** ist die **Application**-Methode **Shutdown()**, die vom Laufzeitsystem oder vom Programm aufgerufen werden kann und das **Exit**-Ereignis des Anwendungsobjekts auslöst (siehe oben).[2] Durch eine Überladung mit **int**-Parameter kann man einen *Return Code* (alias: *Exit Code*) mit einer Information über den (Miss)erfolg der Programmtätigkeit an das Betriebssystem übergeben (z. B. 0: alles gut gegangen, 1: Beendigung mit Fehler).

Das Laufzeitsystem ruft die Methode **Shutdown()** z. B. in den folgenden Fällen auf:[3]

- 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**.

---

1 https://www.c-sharpcorner.com/UploadFile/SamTomato/windows-api-window-using-C-Sharp/

2 https://learn.microsoft.com/en-us/dotnet/desktop/wpf/app-development/application-management-overview

3 https://learn.microsoft.com/en-us/dotnet/api/system.windows.application.shutdownmode

- Im UI-Thread tritt eine unbehandelte Ausnahme auf, und es ist …
  - entweder keine **DispatcherUnhandledException**-Behandlungsmethode vorhanden,
  - oder die vorhandene Methode erklärt die zugrundeliegende Ausnahme *nicht* als behandelt (siehe Abschnitt 13.2.3).
- 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 für die Nachricht **WM_QUIT** zu sorgen, muss die Methode **Shutdown()** per Programm aufgerufen werden.

Eine WPF-Anwendung über die (bei einer Konsolenanwendung ohne Multithreading akzeptable) statische Methode **Environment.Exit()** zu beenden, ist in der Regel *nicht* sinnvoll, weil dabei der komplette Prozess mit allen darin vorhandenen Threads abrupt beendet wird.

Wir fassen leicht vereinfachend zusammen, was die **Application**-Methode **Run()** im Beispielprogramm von Abschnitt 12.2.1 mit der folgenden **Main()** - Methode

```
[System.STAThread]
static void Main() {
   var app = new Application();
   var hf = new WpfOhneXaml();
   app.Run(hf);
}
```

bewirkt:

- 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]

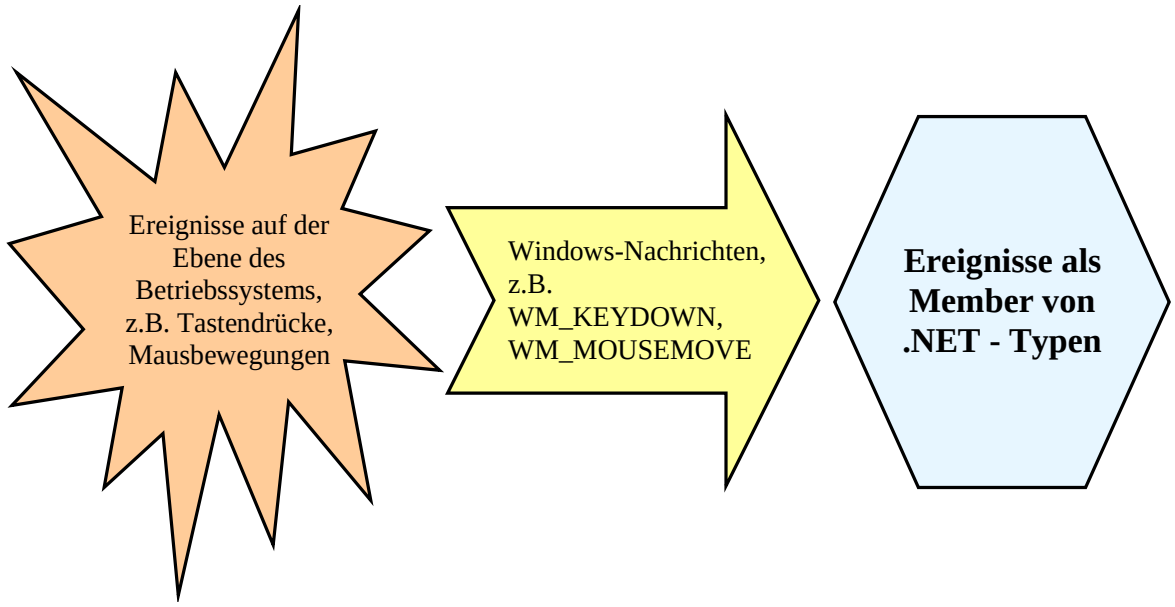
Nachdem die Nachricht **WM_QUIT** eingetroffen ist, enden die Nachrichtenschleife und die **Run()** - Methode. In der Regel stellt eine WPF-Anwendung nach der Rückkehr des **Run()** - Aufrufs ihre Tätigkeit ein (siehe **Main()** - Methode des Beispielprogramms). Ein erneuter Aufruf der **Run()** - Methode ist jedenfalls *nicht* möglich.

Weil das Win32-API durch .NET 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 10.2) durch eigene Methoden behandeln.[2] Insgesamt kann man unterscheiden (vgl. Louis & Strasser 2002, S. 614):

---

[1] Genau genommen ist für jeden Thread (vgl. Kapitel 17 in Baltes-Götz (2021)), der ein Top-Level - Fenster auf dem Bildschirm präsentiert, eine Nachrichtenwarteschlange und dementsprechend eine Nachrichtenschleife erforderlich. Ein WPF-Programm kann also *mehrere* UI-Threads enthalten, was aber in den Beispielprogrammen des Manuskripts nicht vorkommt.

[2] Grundsätzlich können auch Strukturen Ereignisse anbieten, was aber selten geschieht.



12.2.4 Single-Thread - Architektur

Wie die meisten modernen GUI-Bibliotheken ist auch die WPF-Bibliothek aus Konsistenz- und Performanzgründen nach dem Single-Thread - Prinzip konzipiert (siehe die Erläuterungen zur Klasse **DispatcherObject** im Abschnitt 12.2.2). Daher muss der Zugriff auf die GUI-Komponenten dem UI-Thread vorbehalten bleiben, der sich um die Interaktion mit dem Benutzer kümmert. 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 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 17.2.3 von Baltes-Götz (2021) vorgestellt wird. Im aktuellen Kapitel gehen wir dem Thema *Multithreading* noch aus dem Weg, weil genügend andere Herausforderungen zu bewältigen sind.

12.3 Die eXtensible Application Markup Language (XAML)

Die XML-basierte *e**X**tensible **A**pplication **M**arkup **L**anguage* (XAML) wurde von Microsoft zur Deklaration der Bedienoberfläche einer WPF-Anwendung entwickelt, wird aber auch für andere GUI-Bibliotheken aus Microsofts Entwicklungslaboren genutzt (vgl. Abschnitt 12.1.1):

- UWP (***U**niversal **W**indows **P**latform*) mit WinUI 2
- Windows App SDK mit WinUI 3
- MAUI (***M**ulti-platform **A**pp **UI***)

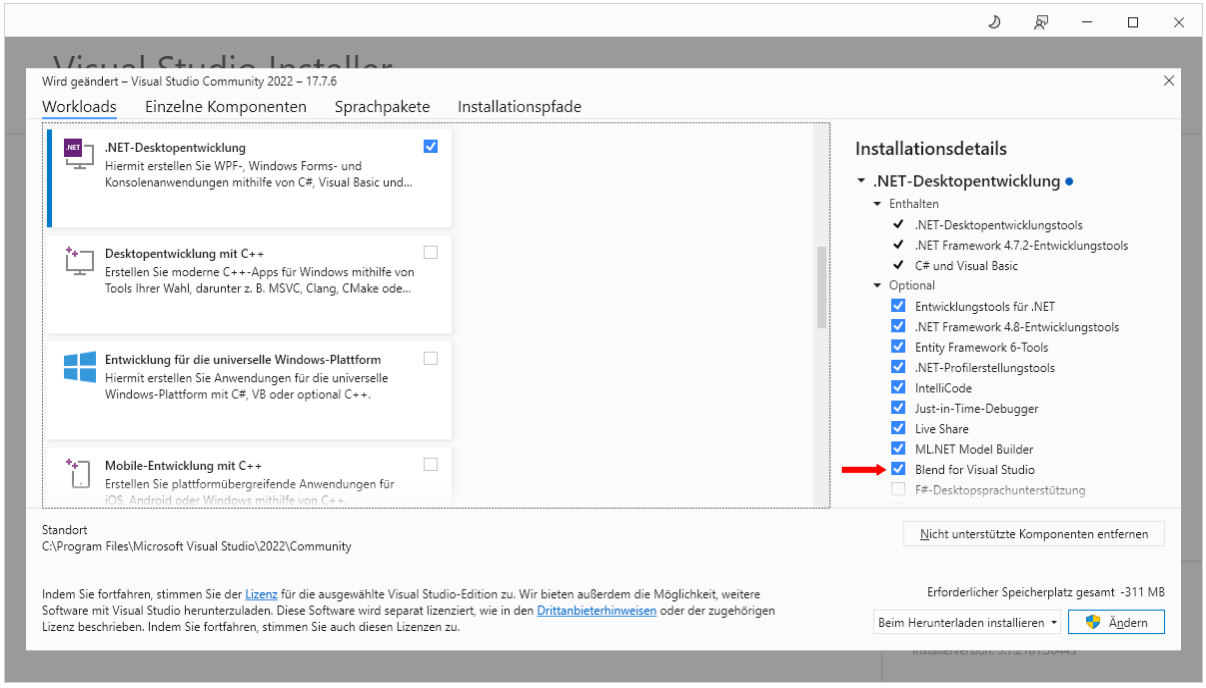
Vor Einführung der WPF-Technik waren die Anwendungslogik *und* die Bedienoberfläche einer GUI-Anwendung (mit dem WinForms-Framework) in derselben Programmiersprache zu realisieren. Eine GUI-Kreation mit C# - Anweisungen ist zwar auch im WPF-Framework möglich, doch existiert hier mit der deklarativen Sprache XAML eine Alternative mit den folgenden Vorzügen:

- Vereinfachung der GUI-Gestaltung
- Trennung von Anwendungslogik und GUI-Design
  Das erleichtert die Zusammenarbeit von Software-Entwicklern und Grafikdesignern.

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/oder multimedialen Ansprüchen steht außerdem noch das auf die GUI-Gestaltung spezialisierte Microsoft-Produkt **Blend for Visual Studio** zur Verfügung,

das optional zusammen mit dem Visual Studio als Bestandteil der Workload **.NET-Desktopentwicklung** installiert werden kann (siehe Abschnitt 3.3.6 zur Anpassung einer Visual Studio - Installation):



Im Manuskript verwenden wir oft den WPF-Designer als wertvolle Unterstützung beim Erstellen des XAML-Codes zu einem Fenster, während Blend *nicht* zum Einsatz kommt.

Dank WPF-Designer kann man in den meisten Fällen darauf verzichten, den XAML-Code zu einem Fenster direkt zu editieren und kommt somit auch ohne die im weiteren Verlauf von Abschnitt 12.3 noch folgenden, nicht immer leicht verdaulichen Erläuterungen zur XAML aus. In manchen Fällen ist aber das direkte Verfassen von XAML-Code effektiver oder sogar erforderlich.

12.3.1 Elementare Regeln zum Aufbau einer XML-Datei

Die *eXtensible Markup Language* (XML) ist zur Deklaration von hierarchisch strukturieren Daten geeignet und wird auch in .NET - Projekten intensiv verwendet, wobei besonders zu erwähnen sind:

- WPF-Fenster- und -Anwendungsdeklaration im XAML-Format
- Konfigurationsdateien im XML-Format

Wie insbesondere in den Kapiteln 2 und 3 zu sehen war, bevorzugt Microsoft mittlerweile das **JSON**-Format (*JavaScript Object Notation*) bei .NET - Projektkonfigurationsdateien (z. B. **DmToEuro.runtimeconfig.json**. Am bewährten XAML-Format zur Fensterdeklaration in diversen GUI-Bibliotheken (WPF, WinUI etc.) wird sich aber vorläufig nichts ändern. Daher lohnt es sich, im aktuellen Abschnitt elementare Regeln zum Aufbau einer XML-Datei zu erläutern.

Eine XML-Datei enthält Text und ist auch für die Lektüre durch Menschen relativ gut geeignet. In der ersten Zeile steht in der Regel die (nicht obligatorische) **XML-Deklaration** mit einer Versions- und Codierungsangabe:

```
<?xml version="1.0" encoding="utf-8" ?>
```

Bei den im aktuellen Kapitel vornehmlich relevanten XAML-Dateien wird die XML-Deklaration meist weggelassen.

Jede XML-Datei besteht aus hierarchisch verschachtelten Elementen und enthält nur *ein Wurzelelement*. Bei einer XAML-Datei zur Deklaration eines WPF-Fensters hat das Wurzelelement den Namen **Window**:

```
<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*)

Start- und Endkennung werden durch Paare spitzer Klammern begrenzt und enthalten den Namen des Elements. In der Endkennung folgt auf die einleitende spitze Klammer ein Schrägstrich.

Als Bestandteile seiner Startkennung kann ein Element beliebig viele **Attribute** enthalten, die aus Name-Wert - Paaren bestehen und durch mindestens ein Leerzeichen getrennt sind. Das folgende **Button** - Element zur Deklaration einer Schaltfläche besitzt z. B. die Attribute **Name** (benennt die Instanzvariable zum Steuerelement), **HorizontalAlignment** und **VerticalAlignment** (horizontale bzw. vertikale Ausrichtung des Steuerelements im Layoutcontainer):

```
<Button Name="button" HorizontalAlignment="Center" VerticalAlignment="Center">
  . . .
</Button>
```

Alle Attributwerte 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 ein Kommentar in einer XML-Datei untergebracht wird:

```
<!-- Kommentar -->
```

12.3.2 XAML-Beschreibung

Das Erstellen und Initialisieren der Objekte der Bedienoberfläche kann auch bei einer WPF-Anwendung per C# - Quellcode erfolgen. Es ist jedoch zu empfehlen, für diesen Zweck die eXtensible Application Markup Language (XAML) zu verwenden. Man bringt den XAML-Code zu einem Fenster in einer Textdatei mit der UTF-8 - Codierung und der Namenserweiterung **xaml** unter. Neben der Entwicklung der Fensterklassen kann in einer WPF-Anwendung auch die Entwicklung der Anwendungsklasse durch die Deklaration per XAML unterstützt werden.

12.3.2.1 Wurzelelement und XML-Namensräume

Eine XAML-Datei einhält *ein* Wurzelelement, wobei für uns in Frage kommen:

- **Application**
  Dieses Wurzelelement wird in der XAML-Datei mit der Anwendungsklassendeklaration verwendet, wobei die folgenden Attribute regelmäßig vorhanden sind:
  - Das Attribut **x:Class** nennt die Anwendungsklasse (Standardname: `App`) samt .NET-Namensraum.
  - Die Attribute **xmlns**, **xmlns:x** und **xmlns:local** vereinbaren XML-Namensräume (siehe unten).
  - Das Attribut **StartupUri** nennt die XAML-Datei zum Hauptfenster der Anwendung, das beim Programmstart automatisch angezeigt werden soll.

  Außerdem können …
  - Behandlungsmethoden zu Ereignissen der Anwendungsklasse durch weitere Attribute
  - sowie Ressourcen im geschachtelten Element **Application.Resources**

  deklariert werden.

  Welchen XAML-Code das Visual Studio für eine neue WPF-Anwendung erzeugt, betrachten wir am Beispiel unseres RSS-Feed-Reader - Projekts (mit dem Namensraum `RssFeedReader`, vgl. Abschnitt 6.8):[1]

```
<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>
```

  Der Name der XAML-Datei wird von der Anwendungsklasse übernommen und durch die Erweiterung **xaml** ergänzt, sodass per Voreinstellung **App.xaml** resultiert.

[1] Der Übersichtlichkeit halber wurden überflüssige XAML-Namensraumdeklarationen entfernt.

- **Window**
  Dieses Wurzelelement wird in der XAML-Datei zur Deklaration eines Fensters (bzw. einer Fensterklasse) verwendet. Die Klasse zum Hauptfenster einer Anwendung erhält vom Visual Studio per Voreinstellung den Namen `MainWindow`, und der zugehörige XAML-Code landet folglich in der Datei **MainWindow.xaml**. Welchen XAML-Code das Visual Studio für ein neues WPF-Fenster erzeugt, betrachten wir am Beispiel unseres RSS-Feed-Reader - Projekts:

```
<Window x:Class="RssFeedReader.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:RssFeedReader"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

  Mit Ausnahme der Eigenschaftselemente (siehe Abschnitt 12.3.2.3.2) repräsentiert jedes Element einer XAML-Datei eine Instanz eines .NET - Typs, und der Elementname ist identisch mit dem Namen dieses Typs (z. B. **Grid**) oder mit dem Namen seiner Basisklasse (z. B. **Window**).

Die Startkennung zu einem XAML - Wurzelelement enthält mehrere **xmlns**-Attribute zur Deklaration von **XML-Namensräumen**, wobei die folgende Syntax verwendet wird:

xmlns[**:***prefix*]="*URI*"

In einem Namensraum werden erlaubte Elemente und Attribute festgelegt, und zur Vermeidung von Namenskollisionen kann zu einem Namensraum ein Präfix vereinbart werden.

Relevante .NET – Namensräume (insbesondere die zur WPF-Bibliothek oder zur Anwendung gehörenden) werden auf XML-Namensräume einer XAML-Datei abgebildet. Es wäre allerdings sehr umständlich, für die auf verschiedene .NET - Namensräume verteilten WPF-Typen jeweils das korrekte Präfix angeben zu müssen. Daher wurden *alle* .NET - Namensräume mit WPF-Typen in einem einzigen XAML-Namensraum zusammengefasst. Dies ist möglich, weil keine Namenskollisionen auftreten. Um die Verwendung des sehr wichtigen XAML-Namensraums mit den WPF-Typen zu erleichtern, hat er *kein* Präfix erhalten, fungiert folglich als **XAML-Standardnamensraum**:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

Die Bezeichnung für einen XML-Namensraum verwendet das URI-Format (*Uniform Resource Identifier*), um Eindeutigkeit sicherzustellen. Wie man leicht prüfen kann, existiert im Internet kein Ort mit diesem Namen.

Das Visual Studio vereinbart außerdem einen XML-Namensraum mit dem Präfix **local** für den projektspezifischen .NET – Namensraum, z. B.:

```
xmlns:local="clr-namespace:RssFeedReader"
```

Neben den Namensräumen für .NET - Typen gibt es einen weiteren wichtigen XML-Namensraum. Er enthält die XAML-Sprachdefinition und verwendet das Präfix **x**:[1]

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

[1] Weitere Informationen über die XML-Namensräume mit den WPF-Klassen bzw. mit der XAML-Sprachdefinition finden sich hier:
https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/xaml-namespaces-and-namespace-mapping-for-wpf-xaml

Die Vereinbarung von XML-Namensräumen erfolgt meist im Start-Tag des Wurzelelements, ist aber grundsätzlich in *jedem* Start-Tag erlaubt.

Es folgt eine Liste mit oft benötigten **x**-Attributen:

- **x:Class**
  Im Wurzelelement wird per **x:Class** - Attribut die zugehörige Code-Behind - Klasse (siehe Abschnitt 12.3.3) samt .NET - Namensraum angegeben, z. B.:
  ```
  x:Class="RssFeedReader.MainWindow"
  ```
- **x:Key**
  Zu einer WPF-Anwendung existiert ein Kollektionsobjekt aus der Klasse **ResourceDictionary** mit anwendungsglobal verwendbaren Ressourcen (z. B. Farbdefinitionen). In der XAML-Datei zur Anwendungsklasse wird die Ressourcen-Kollektion befüllt und der **Application**-Eigenschaft **Resources** zugewiesen. Für ein Objekt der Ressourcen-Kollektion ist der zur Ansprache erforderliche Schlüssel festzulegen, z. B. für ein Objekt der zur Oberflächengestaltung geeigneten Klasse **SolidColorBrush**:
  ```
  <Application x:Class= ... >
      <Application.Resources>
          <SolidColorBrush x:Key="BgColor" Color="WhiteSmoke"/>
      </Application.Resources>
  </Application>
  ```
  Wie man ein Ressourcen-Objekt später verwendet (z. B. einer Eigenschaft zuweist), ist im Abschnitt 12.3.2.3.6 über die Markup-Erweiterungen zu erfahren.
- **x:Name**
  Über dieses Attribut legt man für die Instanz, die aus der XAML-Laufzeitverarbeitung eines XAML-Elements entsteht, einen Namen fest, z. B.:
  ```
  <TextBox x:Name="textBox" ... />
  ```
  Es resultiert eine Instanzvariable mit dem gewählten Namen, die in der Code-Behind - Datei (siehe Abschnitt 12.3.3) einen Instanzzugriff erlaubt. Man muss nicht unbedingt für jede aus einem XAML-Element resultierende Instanz einen Namen vereinbaren. Es ist z. B. kein Name erforderlich, um bei den Ereignissen eines Steuerelements Behandlungsmethoden zu registrieren.

Zum Attribut **x:Name** existiert unter den folgenden Voraussetzungen die funktionsverwandte Alternative **Name** (ohne Namensraum-Präfix):

- Der Typ besitzt die Instanzeigenschaft **Name**.
- Der Typ ist mit dem Attribut **RuntimeNameProperty** dekoriert (zu Attributen siehe Kapitel 14).

Die Klasse **FrameworkElement**, von der alle WPF-Steuerelemente abstammen, erfüllt diese Voraussetzungen, sodass man statt des XAML-Attributs **x:Name** auch das Attribut **Name** verwenden kann. Das vorherige Beispiel lässt sich also vereinfachen:

```
<TextBox Name="textBox" ... />
```

Dann übernimmt der XAML-Parser die zum Attribut **Name** angegebene Zeichenfolge für die Referenzvariable *und* für die Instanzeigenschaft (MacDonald 2012, 28f)

12.3.2.2 Instanzelemente

Mit Ausnahme der Eigenschaftselemente (siehe Abschnitt 12.3.2.3.2) repräsentiert jedes XAML-Element 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 oder sonstige Inhalte vorhanden, dann ist die Startkennung mit einem Schrägstrich und einer schließenden spitzen Klammer zu komplettieren, z. B.

```
<TextBlock Text="Stop" Margin="5" VerticalAlignment="Center" />
```

Sind untergeordnete Elemente (z. B. Steuerelemente in einem Layoutcontainer) oder ein sonstiger Inhalt (z. B. ein Text) vorhanden, dann beendet man die Startkennung mit einer schließenden spitzen Klammer, lässt die untergeordneten Elemente bzw. den sonstigen Inhalt 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 Source="vs.png" Width="20" VerticalAlignment="Center"/>
    <TextBlock Text="Click" Margin="5" VerticalAlignment="Center" />
</StackPanel>
```

Im **StackPanel**-Element wird von den beiden im Abschnitt 12.3.2.1 beschriebenen **Name**-Attributen die Variante aus dem XAML-Standardnamensraum (mit den WPF-Typen) verwendet (ohne Namensraumpräfix **x**).

Ein Instanzelement stellt den Auftrag an die XAML-Verarbeitung dar, eine Instanz vom angegebenen Typ zu erstellen. Die Instanzkreation findet zur Laufzeit über einen Aufruf des parameterlosen Konstruktors zum jeweiligen Typ statt (siehe Abschnitt 12.3.4).

Beim Erstellen und Initialisieren von Instanzen per XAML-Code muss man sich nicht auf die WPF-Typen beschränken. Es werden beliebige .NET - Typen unterstützt, sofern diese über einen parameterlosen und öffentlichen Konstruktor verfügen. Damit die Ansprache im XAML-Code klappt, muss der zugehörige .NET - Namensraum per Attribut einbezogen werden (siehe Beispiel im Abschnitt 12.3.2.1 mit dem Namensraum des Projekts). Das Instanziieren einer zum projektspezifischen Namensraum gehörigen Klasse ist uns im Abschnitt 10.2.3 im Zusammenhang mit einem benutzerdefinierten WPF-Steuerelement begegnet:

```
<Grid>
   <local:SurpriseButton x:Name="surpriseButton" Content="7 gewinnt"
    Margin="30" FontSize="36"/>
</Grid>
```

12.3.2.3 Eigenschaftsausprägungen zuweisen

Um eine Instanzeigenschaft mit einem Wert zu versorgen, genügt oft die Zuweisung einer Zeichenfolge (per Attributsyntax, siehe Abschnitt 12.3.2.3.1), während in anderen Fällen eine Member-Instanz zu erstellen ist (per Eigenschaftselementsyntax, siehe Abschnitt 12.3.2.3.2).

12.3.2.3.1 Attributsyntax

Viele Instanzeigenschaften können in der Instanzelement-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="Stop" 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 letzten 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 **TypeConverter**-Attribut (im Sinn von Kapitel 14) zum Typ der Eigenschaft festgelegt, im Beispiel:

---

[1] Das Visual Studio verwendet doppelte Anführungszeichen, doch sind auch einfache erlaubt.

```
[TypeConverter(typeof(ThicknessConverter))]
[Localizability(LocalizationCategory.None, Readability = Readability.Unreadable)]
public struct Thickness : IEquatable<Thickness> { ... }
```

12.3.2.3.2 Eigenschaftselementsyntax

Die Attributsyntax eignet sich *nicht*, wenn einer Eigenschaft eine komplexe Member-Instanz 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 ist ein Instanzelement mit dem Typ der Eigenschaft zu erstellen. Mit Ausnahme des Wurzelelements treten alle Instanzelemente einer XAML-Datei bei der Zuweisung einer Eigenschaftsausprägung auf.

Im folgenden Beispiel wird der **Content**-Eigenschaft eines **Button**-Objekts (Datentyp: **Object**) ein **StackPanel**-Layoutobjekt (siehe Abschnitt 12.6.3) zur Verwaltung der **Button**-Oberfläche zugewiesen:

```
<Button Name="button" Background="WhiteSmoke">
  <Button.Content>
    <StackPanel Orientation="Horizontal">
      . . .
    </StackPanel>
  </Button.Content>
</Button>
```

Beschränkt man sich bei der **Content**-Eigenschaft eines **Button**-Instanzelements auf eine Beschriftung, dann kommt man mit der Attributsyntax aus, z. B.:

```
<Button Name="button" Content="Stop" Background="WhiteSmoke"/>
```

Die Attributsyntax ist oft eine bequeme, per Typkonverter (vgl. Abschnitt 12.3.2.3.1) ermöglichte Kurzform der Eigenschaftselementsyntax. Im letzten Beispiel wird der **Button**-Instanzeigenschaft **Background** vom Typ der Klasse **Brush** (Namensraum **System.Windows.Media**) per Attributsyntax ein Objekt vom Typ **SolidColorBrush** zugewiesen, und man spart sich die folgende, aufwändigere Eigenschaftselementsyntax:

```
<Button Name="button" Content="Stop">
  <Button.Background>
      <SolidColorBrush Color="WhiteSmoke"/>
  </Button.Background>
</Button>
```

12.3.2.3.3 Inhaltseigenschaft

Jede WPF-Klasse kann von ihren Instanzeigenschaften genau eine als sogenannte *Inhaltseigenschaft* (engl.: *content property*) deklarieren. Dazu wird der Klasse das Attribut **ContentProperty** (im Namensraum **System.Windows.Markup**) angeheftet, wobei der Begriff *Attribut* im Sinn von Kapitel 14 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, dann kann auf das Kennungspaar gemäß XAML-Eigenschaftselementsyntax (siehe Abschnitt 12.3.2.3.2) verzichtet werden. Weil die

Klasse **Button** indirekt von der Klasse **ContentControl** abstammt, kann das im Abschnitt 12.3.2.3.2 präsentierte Beispiel mit einer Deklaration für die **Button**-Eigenschaft **Content**

```
<Button Name="button" Background="WhiteSmoke">
  <Button.Content>
    <StackPanel Orientation="Horizontal">
      . . .
    </StackPanel>
  </Button.Content>
</Button>
```

unter Verzicht auf die Einrahmung

```
<Button.Content> ... </Button.Content>
```

einfacher geschrieben werden:

```
<Button Name="button" Background="WhiteSmoke">
  <StackPanel Orientation="Horizontal">
      . . .
  </StackPanel>
</Button>
```

12.3.2.3.4 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:

- 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.
- Für den Typ ist eine Klasse mit entsprechenden Kompetenzen als Typkonverter definiert (vgl. Abschnitt 12.3.2.3.1), z. B.:

  ```
  <Button.Background>
      LightBlue
  </Button.Background>
  ```

- Der Typ ist im XAML-Sprachumfang bekannt, was bei vielen elementaren Datentypen (z. B. **Int32**, **Double**) der Fall ist, z. B.:

  ```
  <Button.Height>
      40
  </Button.Height>
  ```

Die Textinhaltsyntax wird meist bei der *Inhaltseigenschaft* (siehe Abschnitt 12.3.2.3.3) verwendet, sodass sich das erste Beispiel des aktuellen Abschnitts vereinfachen lässt:

```
<Button>Knopf</Button>
```

Diese Lösung ist sogar kürzer als die äquivalente Attributsyntax:

```
<Button Content="Knopf"/>
```

12.3.2.3.5 Kollektionssyntax

Wenn der Typ einer Eigenschaft …

- das (non-generische) Interface **IList** implementiert,
- oder (non-generische) das Interface **IDictionary** implementiert,
- oder ein Array-Typ ist,

dann wird der Typ bei der XAML-Verarbeitung als Kollektion erkannt, und die *Kollektionssyntax* ist erlaubt, d. h.:

- Man verzichtet auf ein Instanzelement zur Kollektion.
- Man listet Instanzelemente auf, die schlussendlich über die Methode **Add()** in die Kollektion aufgenommen werden sollen.

Implementiert der Typ einer Eigenschaft zwar die *generische* Schnittstelle **IList<T>** bzw. **IDictionary<K, V>**, aber nicht die non-generische Variante, dann scheitert die Kollektionserkennung bei der XAML-Verarbeitung. Für selbst definierte Klassen schlägt Microsoft als Ausweg vor, …

- die Klasse **List<T>**, die neben dem generischen Interface **IList<T>** auch das non-generische Interface **IList** implementiert,
- bzw. die Klasse **Dictionary<K, V>**, die neben dem generischen Interface **IDictionary<K, V>** auch das non-generische Interface **IDictionary** implementiert,

(als Basisklassen) zu verwenden.[1]

Im folgenden Beispiel wird zur Hintergrundgestaltung ein Farbverlauf durch ein Objekt der Klasse **LinearGradientBrush** mit der Eigenschaft **GradientStops** per Eigenschaftselementsyntax definiert. Statt ein Instanzelement vom Typ **GradientStopCollection** samt Kindelementen anzugeben, werden ausschließlich die Kindelemente (vom Typ **GradientStop**) aufgelistet:

```
<Button Content="Stop" Margin="10" FontSize="48">
    <Button.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStop Offset="0.0" Color="Red" />
                <GradientStop Offset="1.0" Color="Blue" />
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

Hier ist die umständlichere Konstruktion mit dem expliziten **GradientStopCollection**-Instanzelement zu sehen:[2]

---

[1] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/xaml-syntax-in-detail

[2] Auf das explizite Instanzelement zur Kollektion sollte auf jeden Fall verzichtet werden, weil dazu ein parameterloser Konstruktor verfügbar sein muss. In manchen Fällen scheitert die umständliche Lösung, während die bequeme Kollektionssyntax klappt.

```
<Button Content="Stop" Margin="10" FontSize="48">
    <Button.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0.0" Color="Red" />
                    <GradientStop Offset="1.0" Color="Blue" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

12.3.2.3.6 Markup-Erweiterungen

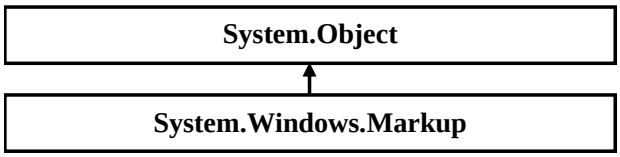
Mit den bisher vorgestellten Techniken zur Festlegung von Eigenschaftsausprägungen ist es *nicht* möglich, ...

- eine Referenz zu einem bereits existenten Objekt zuzuweisen,
- eine Verbindung zu einer Quelle von variablen Werten herzustellen (Datenbindung).

Dies gelingt mit einer sogenannten *Markup-Erweiterung*, die meist per Attributsyntax unter Verwendung einer geschweift eingeklammerten Zeichenfolge realisiert wird.[1] Im folgenden Beispiel wird einer Eigenschaft eine Referenz auf ein anderenorts definiertes Objekt zugewiesen:

```
Background="{StaticResource BgColor}"
```

Eine Markup-Erweiterung ist ein Objekt mit einem Typ, der bei Verwendung der Attributsyntax unmittelbar nach der öffnenden Klammer anzugeben ist. Die Erweiterungsklassen stammen von der Klasse **MarkupExtension** im .NET - Namensraum **System.Windows.Markup** ab:



Viele für WPF-Anwendungen wichtige Markup-Erweiterungsklassen wurden in den XAML-Standardnamensraum (mit den WPF-Typen) aufgenommen, sodass bei der Typbezeichnung kein Namensraumpräfix erforderlich ist. Besonders wichtig sind:

[1] Es ist auch möglich, allerdings weniger gebräuchlich, Markup-Erweiterungen in Eigenschaftselementen zu verwenden (siehe MacDonald 2012, S. 34).

- **StaticResource**
  In WPF-Anwendungen werden häufig sogenannte *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 zur **Application**-Eigenschaft **Resources** (Datentyp **ResourceDictionary**) per XAML-Eigenschafts- und -Kollektionssyntax ein **SolidColorBrush**-Objekt aufgenommen, das anschließend anwendungsglobal über den vereinbarten Schlüssel `BgColor` verfügbar ist:

```
<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 der Markup-Erweiterungsklasse 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 6.8.6 haben wir im Zusammenhang mit dem RSS-Feed - Reader schon Bekanntschaft mit der WPF-Datenbindungstechnologie gemacht. Für das zur formatierten Anzeige eines **ListBox**-Steuerelements verwendete **TextBlock**-Objekt

```
<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. Eine Begrenzung des Eigenschaftsnamens durch Anführungszeichen ist *nicht* erforderlich bzw. erlaubt, weil es ansonsten zu einer schwer interpretierbaren Verschachtelung von Anführungszeichen käme.

Manche Markup-Erweiterungsklassen sind *nicht* im XAML-Standardnamensraum (mit den WPF-Typen) enthalten, sondern im generellen XAML-Namensraum, sodass dem Typnamen das Namensraumpräfix **x** voranzustellen ist. 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, um der Steuerelementeigenschaft **Background** das durch die statische Eigenschaft **ControlDarkBrush** der Klasse **SystemColors** referenzierte Objekt zuzuweisen. Dabei darf der Namensbestandteil *Extension* wegfallen:

```
Background="{x:Static SystemColors.ControlDarkBrush}"
```

Durch geschweifte Klammern begrenzt wird eine Zeichenfolge mit dem folgenden Aufbau erwartet:

Prefix:*MarkupClass Typ*.*FieldOrProperty*

Das Präfix ist nur bei Markup-Erweiterungsklassen anzugeben, die *nicht* zum XAML-Standardnamensraum (mit den WPF-Typen) gehören.

12.3.2.4 XAML-Registrierung von Ereignisbehandlungsmethoden

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 (ohne Parameterklammern) 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 12.3.3).

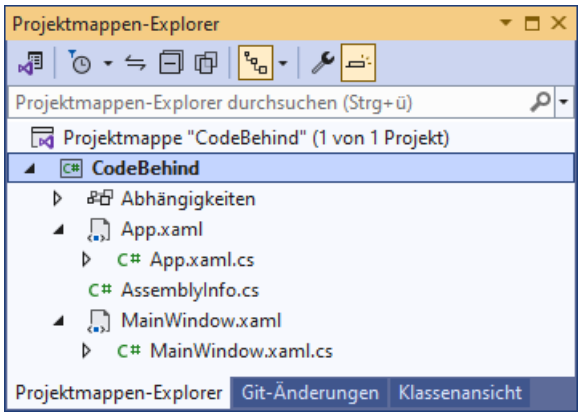
Man kann darüber diskutieren, ob hier eventuell das Prinzip der Trennung von Anwendungslogik und GUI-Gestaltung verletzt ist.

12.3.3 Code-Behind - Dateien

Ein vom Visual Studio über die Vorlagen **WPF-Anwendung** erstelltes Projekt enthält die beiden XAML-Dateien

- **App.xaml** zur Deklaration der Anwendungsklasse
- **MainWindow.xaml** zur Deklaration der Hauptfensterklasse

Wie der **Projektmappen-Explorer** nach dem Aufklappen der Zweige zu den beiden XAML-Dateien zeigt, gehört jeweils eine C# - Quellcodedatei dazu, die ebenfalls nach der definierten Klasse benannt ist:



Die beiden C# - Quellcodedateien 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 Anwendungsklasse `App`, die von der BCL-Klasse **Application** abstammt (vgl. Abschnitt 12.2.3):

```
using System;
  . . .
using System.Windows;

namespace CodeBehind {
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application {
    }
}
```

  Das Schlüsselwort **partial** signalisiert dem Compiler, dass zu der Klassendefinition noch ein Ergänzungsstück in einer anderen Quellcodedatei gehört, wobei die beiden partiellen Definitionen gleichberechtigt und voneinander abhängig ein Ganzes ergeben.

  Wenn man im WPF-Designer der Entwicklungsumgebung eine Ereignisbehandlung zum Anwendungsobjekt anfordert, dann 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:

  - Die Datei **App.xaml** per Doppelklick auf den Eintrag im **Projektmappen-Explorer** öffnen
  - Das Wurzelelement **Application** der XAML-Datei per Mausklick markieren
  - Im **Eigenschaften**-Fenster per Mausklick auf das Symbol ⚡ die Liste mit den Ereignissen anfordern
  - Doppelklick auf das Texteingabefeld zum Ereignis **Exit**

  Dabei erhält das Wurzelelement **Application** einen Wert für das Attribut bzw. Ereignis **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">
  . . .
</Application>
```

- **MainWindow.xaml.cs**
  In der Datei **MainWindow.xaml.cs** findet sich eine partielle Definition der Fensterklasse `MainWindow`, die von der BCL-Klasse **Window** abstammt (vgl. Abschnitt 12.2.2):

```
using System;
  . . .
using System.Windows.Shapes;

namespace CodeBehind {
 /// <summary>
 /// Interaction logic for MainWindow.xaml
 /// </summary>
 public partial class MainWindow : Window {
    public MainWindow() {
       InitializeComponent();
    }
 }
}
```

Wenn man im WPF-Designer der Entwicklungsumgebung eine Ereignisbehandlung zum Fenster oder zu einem Steuerelement auf der Fensteroberfläche anfordert, dann 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:

- Die Datei **MainWindow.xaml** per Doppelklick auf den Eintrag im Projektmappen-Explorer öffnen
- Mit dem WPF-Designer nötigenfalls per Drag & Drop ein Steuerelement der Klasse **Button** auf das Anwendungsfenster befördern
- Doppelklick auf das **Button**-Objekt setzen

Dabei erhält in der Datei **MainWindow.xaml** das zum **Button**-Objekt gehörige Instanzelement einen Wert für das Attribut bzw. für die Eigenschaft **Click**, z. B.:

```
<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 **Eigenschaften**-Fenster per Mausklick auf das Symbol ⚡ die Liste mit den Ereignissen anfordern
- Doppelklick auf das Texteingabefeld zum Ereignis

Im nächsten Abschnitt ist zu erfahren, wo die Ergänzungsstücke zu den partiellen Klassendefinitionen in den Dateien **App.xaml.cs** und **MainWindow.xaml.cs** stecken.

12.3.4 XAML-Verarbeitung beim Erstellen und Starten einer WPF-Anwendung

In diesem Abschnitt beschäftigen wir uns mit der XAML-Verarbeitung beim Erstellen und Starten einer WPF-Anwendung und stoßen dabei auf interessante Details. Wir werden z. B. die Startmethode **Main()** lokalisieren, die auch in 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, sodass Sie den Abschnitt im Vertrauen auf die WPF-Magie ignorieren dürfen.

Enthält eine WPF-Anwendung XAML-Dateien (= Normalfall), dann werden beim Erstellen der Anwendung (z. B. angefordert mit dem Menübefehl **Erstellen > Projektmappe neu erstellen**) die folgenden Dateien generiert:[1]

- eine Datei mit *binärem* XAML-Code zu jeder Fensterklasse
- eine C# - Quellcode zu jeder Fensterklasse sowie zur Anwendungsklasse

Diese Dateien landen bei einem Projekt mit der Zielplattform **Any CPU** und der Build-Konfiguration **Debug** im Projektunterordner

...\obj\Debug\net7.0-windows

12.3.4.1 BAML-Dateien zu den Fensterklassen

Zu jeder Fensterklasse wird die zugehörige XAML-Datei (z. B. **MainWindow.xaml**) in eine binäre (validierte und effizient zu verarbeitende) Variante mit der Namenserweiterung **baml** übersetzt (z. B. **MainWindow.baml**). Die BAML-Dateien zu den Fensterklassen werden als Ressourcen in das vom Compiler erzeugte Assembly eingebunden und beim Programmstart vom XAML-Parser ausgewertet (siehe Abschnitt 12.3.4.2). Dabei entstehen die im XAML-Code konfigurierten GUI-Objekte.

12.3.4.2 Generierter C# - Quellcode

Aus jeder XAML-Datei (zu einem Fenster oder zur Anwendung gehörend) entsteht eine C# - Quellcodedatei mit einer partiellen Klassendefinition. In einem Projekt mit der Anwendungsklasse `App` und einem einzigen Fenster (Hauptfensterklasse `MainWindow`) resultieren also die beiden folgenden Dateien:

- **MainWindow.g.cs**[2]
  In dieser Datei mit einer partiellen Definition der Hauptfensterklasse[3]
  ```
  public partial class MainWindow :
    System.Windows.Window, System.Windows.Markup.IComponentConnector {
     . . .
  }
  ```
  finden sich die Referenzvariablen zu den benannten Steuerelementen des Fensters, z. B.:
  ```
  internal System.Windows.Controls.Button button;
  ```
  Die Variablennamen stammen aus den **Name** - oder **x:Name** - Attributen der zugehörigen XAML-Instanzelemente (vgl. Abschnitt 12.3.2.1).
  Außerdem befindet sich in der Datei **MainWindow.g.cs** die im Fensterklassenkonstruktor (siehe Abschnitt 12.3.3 zur Code-Behind - Datei **MainWindow.xaml.cs**) aufgerufene Methode **InitializeComponent()**. Diese Methode sorgt durch einen Aufruf der statischen **Application**-Methode **LoadComponent()** für das Laden der im Abschnitt 12.3.4.1 beschriebenen BAML-Ressource zum Hauptfenster und damit für das Instanziieren und Initialisieren der zugehörigen Objekte, z. B.:

---

[1] Beim Erstellen einer Anwendung wird die **Microsoft-Build-Engine** tätig.

[2] Neben der Datei **MainWindow.g.cs** gibt es im selben Ordner noch eine Datei mit dem Namen **MainWindow.g.i.cs**. Während die Datei **MainWindow.g.cs** beim Erstellen des Programms entsteht, wird die Datei **MainWindow.g.i.cs** schon *vor* dem Erstellen des Programms angelegt und sukzessiv an gespeicherte Änderungen im XAML-Code angepasst.

[3] Die Farbe mancher Klassennamen in der Datei **MainWindow.g.cs** ist nicht beim Zwischenablagentransfer verloren gegangen, sondern fehlt auch im Quellcodeeditor der Entwicklungsumgebung. Daher wurde keine manuelle Korrektur vorgenommen.

```
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 von der implementierten Schnittstelle **IComponentConnector** (im Namensraum **System.Windows.Markup**) gefordert wird. Hier werden die Ereignisbehandlungsmethoden zu den Steuerelementen registriert, die in der Code-Behind - Datei **MainWindow.xaml.cs** implementiert werden (siehe Abschnitt 12.3.3), z. B.:

```
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 im Methodenkopf der Schnittstellenname angegeben werden muss und kein Zugriffsmodifikator gesetzt werden darf (siehe Abschnitt 9.4).

- **App.g.cs**[1]
  Hier findet sich die partielle Definition der von **System.Windows.Application** abgeleiteten Anwendungsklasse mit dem Namen `App`, z. B.:[2]

[1] Neben der Datei **App.g.cs** gibt es im selben Ordner noch eine Datei mit dem Namen **App.g.i.cs**. Während die Datei **App.g.cs** beim Erstellen des Programms entsteht, wird die Datei **App.g.i.cs** schon *vor* dem Erstellen des Programms angelegt und sukzessiv an gespeicherte Änderungen im XAML-Code angepasst.

[2] Die Farbe mancher Klassennamen in der Datei **App.g.cs** ist nicht beim Zwischenablagentransfer verloren gegangen, sondern fehlt auch im Quellcodeeditor der Entwicklungsumgebung. Daher wurde keine manuelle Korrektur vorgenommen.

```
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-Datei zur 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 im Abschnitt 12.2.1 wird das Hauptfenster nicht per **Run()** - Parameter festgelegt. Es ist dem Anwendungsobjekt bereits über seine **StartupUri**-Eigenschaft bekannt.

Die Behandlungsmethoden für `App`-Ereignisse werden in der eben beschriebenen Methode **InitializeComponent()** registriert und in der Code-Behind - Datei **App.xaml.cs** zur Anwendungsklasse implementiert (siehe Abschnitt 12.3.3).

Die mit **#line** startenden Zeilen in **MainWindow.g.cs** bzw. **App.g.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:

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives

- im **Projektmappen-Explorer** über den Symbolschalter **alle Dateien anzeigen** lassen
- den Pfad **obj\Debug\net7.0-windows** mit den generierten Dateien öffnen
- die gewünschte Datei per Doppelklick im Quellcode-Editor öffnen

12.4 Routingereignisse

Wie zu Beginn von Kapitel 12 erwähnt, geht es bei der GUI-Programmierung sehr ereignisreich zu. Folgerichtigerweise wurde für die Windows Presentation Foundation mit den sogenannten *Routingereignissen* eine Ergänzung bzw. Erweiterung der CLR-Ereignistechnik eingeführt. Microsoft nennt zwei Besonderheiten der Routingereignisse im Vergleich zu den im Abschnitt 10.2 behandelten Ereignissen:[1]

- Funktionsweise
  Eine Behandlungsmethode (z. B. zum **Click**-Ereignis eines WPF-Steuerelements) kann nicht nur bei der Ereignisquelle registriert werden, sondern auch bei den übergeordneten UI-Elementen bis hinauf zum Fensterobjekt. Tritt ein Ereignis auf, dann werden nicht nur die bei der Ereignisquelle registrierten Behandlungsmethoden aufgerufen, sondern nacheinander auch die bei übergeordneten Elementen registrierten Behandlungsmethoden. Dabei kann die Weiterleitung von der Quelle bis zum Fenster erfolgen oder auch umgekehrt. Weil UI-Elementverschachtelungen noch nicht explizit betrachtet wurden, soll ein typisches Beispiel skizziert werden:
  - Ein **TextBlock**-Objekt, das zur Quelle eines **MouseDown**-Ereignisses werden kann,
  - befindet sich in einem **StackPanel**-Layoutcontainer,
  - der zu einem **Button**-Objekt gehört,
  - das von einem **Grid**-Layoutcontainer verwaltet wird,
  - der in einem **Window**-Objekt als Wurzel- Layoutcontainer arbeitet.
- Implementierung
  Ein Routingereignis basiert auf einem Objekt der Klasse **RoutedEvent** und wird vom WPF-Ereignissystem verarbeitet.

Hinsichtlich der automatischen Weiterleitung entlang einer Hierarchie von potenziellen Interessenten zeigt das WPF-Ereignissystem eine Verwandtschaft zur Kommunikation von Ausnahmefehlern durch das Laufzeitsystem (siehe Kapitel 13).

Um jede Verwechslung der im Abschnitt 10.2 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 für ein Routingereignis nur bei der *Ereignisquelle* eine Behandlungsmethode registriert, was wir bisher getan haben, dann bestehen keine relevanten Unterschiede gegenüber einem CLR-Ereignis. Wir haben z. B. für den im Abschnitt 6.8 entwickelten RSS-Feed - Reader mit Assistentenhilfe (in der Code-Behind - Datei zur Hauptfensterklasse `MainWindow`) eine Behandlungsmethode zum **Click**-Ereignis des **Button**-Steuerelements erstellt:

```
private void button_Click(object sender, RoutedEventArgs e) {
   . . .
}
```

Diese wird in der vom Visual Studio generierten Quellcodedatei **MainWindow.g.cs** (vgl. Abschnitt 12.3.4) wie bei einem CLR-Ereignis (vgl. Abschnitt 10.2.2) „an der Quelle“ registriert:

---

[1] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/events/routed-events-overview

```
this.button.Click += new System.Windows.RoutedEventHandler(this.button_Click);
```

Um diese Verwendung von Routingereignissen mit der vertrauten Syntax für Ereignisse zu ermöglichen, bieten die meisten Routingereignisse ein CLR-Ereignis als Wrapper (dt.: *Verpackung*). Wie man ein Routingereignis definiert und ein CLR-Ereignis ergänzt, um Ereignisbehandlungsmethoden auf konventionelle Weise registrieren zu können, zeigt der Abschnitt 12.4.5.

12.4.1 Routingstrategien

Nach der verwendeten Weiterleitungsstrategie sind die folgenden Kategorien von Routingereignissen zu unterscheiden:

- **Tunnelereignisse**
  Hier kommt als Routingstrategie das sogenannte *Tunneling* zum Einsatz. Das von einem GUI-Element (z. B. einem **Image**-Objekt als Inhaltsbestandteil einer **Button**-Oberfläche) ausgelöste Ereignis wird zuerst dem obersten Element in der GUI-Elementhierarchie (also dem **Window**-Objekt) zur Behandlung angeboten, d. h. die dort registrierten Behandlungsmethoden werden ggf. aufgerufen. 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 **Preview**. Dementsprechend werden Tunnelereignisse gelegentlich auch als *Vorschauereignisse* bezeichnet.
  Beispiel: **PreviewMouseDown**-Ereignis der Klasse **UIElement**
- **Blasenereignisse**
  Hier kommt als Routingstrategie das sogenannte *Bubbling* zum Einsatz. Zuerst werden die bei der Ereignisquelle registrierten Behandlungsmethoden aufgerufen. Dann werden die bei übergeordneten Elementen in der GUI-Elementhierarchie (bis hinauf zum **Window**-Objekt) registrierten Behandlungsmethoden nacheinander aufgerufen.
  Beispiel: **Click**-Ereignis der Klasse **ButtonBase**
- **Direktereignisse**
  Nur die bei der Ereignisquelle 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 12.4.4).
  Beispiele: **MouseEnter**-Ereignis der Klasse **UIElement**

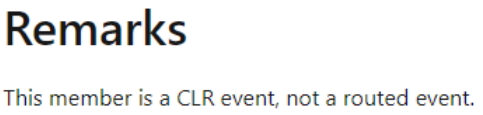
Ein Tunnel- oder Blasenereignis ist *nicht* erledigt, nachdem *eine* Behandlungsmethode aufgerufen worden ist, sondern es wird entlang der Hierarchie weiter angeboten, bis es seine Endstation erreicht hat oder als behandelt deklariert wird. Dazu ist in einer Behandlungsmethode die Eigenschaft **Handled** des übergebenen Beschreibungsobjekts auf den Wert **true** zu setzen.

Für Routingereignisse wird in der Dokumentation u. a. die Routingstrategie angegeben, z. B. beim **Click**-Ereignis der Klasse **ButtonBase**:

Routed Event Information

Identifier field	ClickEvent
Routing strategy	Bubbling
Delegate	RoutedEventHandler

Dass ein Ereignis einer WPF-Klasse ausnahmsweise vom CLR-Typ ist, erkennt man meist am Fehlen der Detailinformationen über Routingereignisse. Gelegentlich wird explizit darauf hingewiesen, dass es sich „nur“ um ein CLR-Ereignis handelt:



Bei einem Routingereignis muss eine Behandlungsmethode den folgenden Delegatentyp erfüllen:

public delegate void RoutedEventHandler(Object *sender***, RoutedEventArgs** *e***)**

Infolgedessen erhält die Behandlungsmethode beim Aufruf die folgenden Parameter:

- **Object** *sender*
  Dieses Objekt hat mit Hilfe der **UIElement**-Methode **RaiseEvent()** (siehe Abschnitt 12.4.5) die Behandlungsmethode aufgerufen. Bei einem Tunnel- oder Blasenereignis wird durch den ersten Parameter dasjenige Objekt referenziert, bei dem das Ereignis auf seiner Route gerade angekommen ist. In diesem Fall zeigt der erste Parameter also i. A. *nicht* auf die Ereignisquelle.
- **RoutedEventArgs** *e*
  Das Ereignisbeschreibungsobjekt besitzt u. a. die beiden folgenden Eigenschaften:
  - **Source**
    Man erfährt, welchem Objekt das Routingereignis ursprünglich widerfahren ist.
  - **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 der Elementhierarchie 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 auch 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*).

Als Ausnahme von der Paarbildungsregel existiert zum Blasenereignis **Click** der Klasse **ButtonBase** *kein* Tunnelereignis.

12.4.2 Einsatzempfehlungen

Blasenereignisse bewähren sich z. B. in den folgenden Einsatzszenerien:

- **Gemeinsame Ereignisbehandlung für mehrere Steuerelemente**
  Befinden sich z. B. in einem Layoutcontainer mehrere **Button**-Objekte, und sollen deren **Click**-Ereignisse durch eine gemeinsame Methode behandelt werden, dann genügt eine einmalige Registrierung dieser Methode beim Layoutcontainer. Wäre **Click** *kein* Blasenereignis, dann müsste die Registrierung für jeden Schalter wiederholt werden.
- **Höherwertige 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 Orientation="Horizontal">
    <Image Source="vs.png"/>
    <TextBlock Text="Click"/>
  </StackPanel>
</Button>
```

  Die ein **Click**-Ereignis konstituierenden elementaren Ereignisse (engl.: *low-level events*) wie **MouseDown** und **MouseUp** können jeweils bei einem anderen Inhaltsbestandteil auftreten, z. B. ...
    - beim **Image**-Objekt das Routingereignis **MouseDown**
    - und wegen einer Mausbewegung beim **TextBlock**-Objekt das Routingereignis **MouseUp**.

  Weil alle Ereignisse an das **Button**-Objekt weitergeleitet werden, kann dort ein Klick auf den Schalter erkannt und als neues, höherwertiges Routingereignis (engl.: *high-level event*) ausgelöst werden.

Tunnelereignisse werden von Anwendungsprogrammen 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 irreguläre Zeichen vorhanden sind.
- **eine Ereignisbehandlung vorzubereiten**

Der Abschnitt 12.4 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).

12.4.3 Beobachtungsstudie

Um das WPF-Ereignissystem bei der Arbeit beobachten zu können, erstellen wir eine Anwendung mit der folgenden Hierarchie von UI-Elementen:

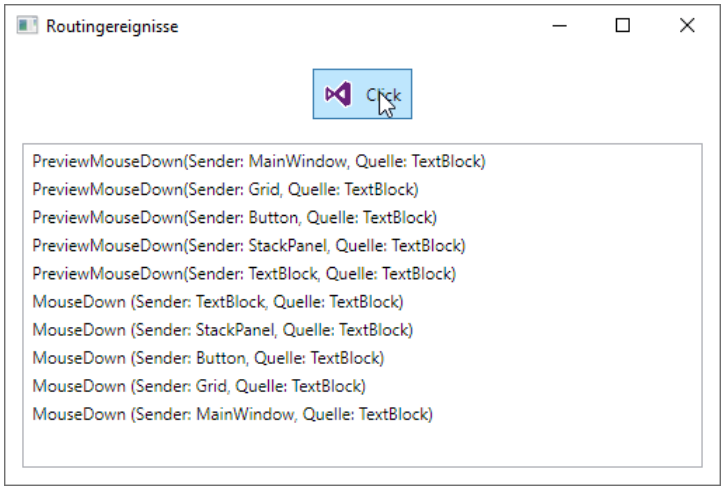
Ein Wurzelelement aus der von **Window** abstammenden Klasse `MainWindow`,
darin ein Layoutcontainer aus der Klasse **Grid** (siehe Abschnitt 12.6.1),
in der **Grid**-Zelle (0, 0) u. a. ein Befehlsschalter aus der Klasse **Button**,
darin ein Layoutcontainer aus der Klasse **StackPanel** (siehe Abschnitt 12.6.2),
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:

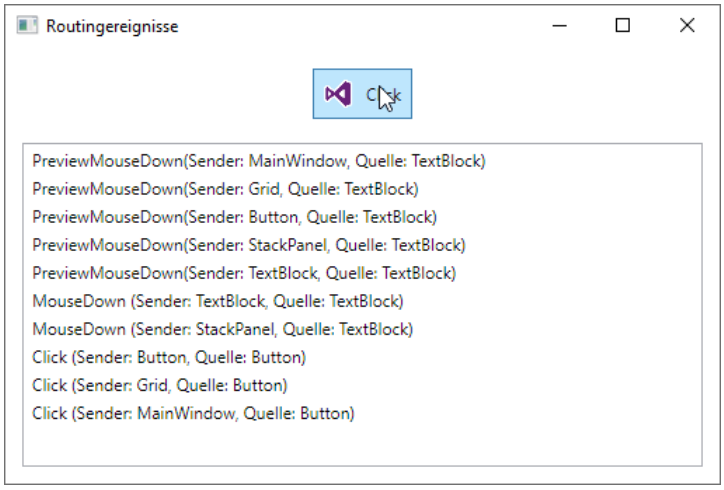
- Tunnelereignis **PreviewMouseDown** aus der Klasse **UIElement**
- Blasenereignis **MouseDown** aus der Klasse **UIElement**
- Blasenereignis **Click** aus der Klasse **ButtonBase** (Basisklasse von **Button**)

Beim folgenden Einsatz



ist das **TextBlock**-Objekt wegen Drückens der *rechten* Maustaste zur Quelle für das Tunnelereignis **PreviewMouseDown** geworden. Das Ereignis ist gemäß Abschnitt 12.4.1 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 Drücken der rechten Maustaste: Das Tunnelereignis **PreviewMouseDown** mit dem **TextBlock**-Objekt als Quelle wird an der Wurzel beginnend auf allen Hierarchieebenen behandelt. Danach startet erwartungsgemäß das aufperlende Gegenstück **MouseDown** an der Quelle, endet allerdings beim **StackPanel**-Objekt. Das **Button**-Objekt verlängert nicht die Serie von **MouseDown**-Ereignisbehandlungen, sondern feuert stattdessen das Blasenereignis **Click**, das seinen Weg nach oben bis zur Wurzel der GUI-Elementhierarchie nimmt.

Das Programm verwendet für jedes von den drei beobachteten Routingereignisse eine Behandlungsmethode, die bei allen UI-Elementen in der oben beschriebenen Verschachtelung registriert wird. Neben dem Ereignistyp protokollieren die drei Behandlungsmethoden auch ...

- den Aufrufer der Methode
  Sein Laufzeitdatentyp wird über das erste Parameterobjekt (*sender*) festgestellt.
- die Ereignisquelle
  Ihr Laufzeitdatentyp wird über die **Source**-Eigenschaft des zweiten Parameterobjekts (Typ **RoutedEventArgs**) festgestellt.

Die Behandlungsmethoden werden in der Code-Behind - Datei mit der partiellen Definition der Fensterklasse untergebracht (vgl. Abschnitt 12.3.3):

```
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 seinem Member-Kollektionsobjekt aus der Klasse **ItemCollection** übergeben, das über die **ListBox**-Eigenschaft **Items** ansprechbar ist und u. a. die Methode **Add()** beherrscht (vgl. Abschnitt 12.7.4.5.1).

Das Registrieren der Ereignisbehandlungsmethoden ist per XAML-Code einfacher zu bewerkstelligen als mit C# - Code:

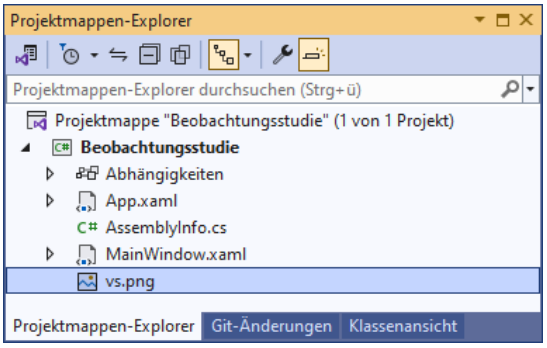
```
<Window x:Class="Beobachtungsstudie.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Routingereignisse" Height="350" Width="525"
        PreviewMouseDown="HandlePreviewMouseDown" MouseDown="HandleMouseDown"
        ButtonBase.Click="HandleClick">
```

```
    <Grid PreviewMouseDown="HandlePreviewMouseDown" MouseDown="HandleMouseDown"
          Button.Click="HandleClick">
        <Button Name="button" Margin="0,15" Height="36" Width="71"
                VerticalAlignment="Top" HorizontalAlignment="Center"
                PreviewMouseDown="HandlePreviewMouseDown" MouseDown="HandleMouseDown"
                Click="HandleClick">
            <StackPanel Name="stackPanel" Orientation="Horizontal"
                        PreviewMouseDown="HandlePreviewMouseDown"
                        MouseDown="HandleMouseDown">
                <Image Name="image" Width="20" Margin="5" Source="vs.png"
                       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="12,68,12,12" />
    </Grid>
</Window>
```

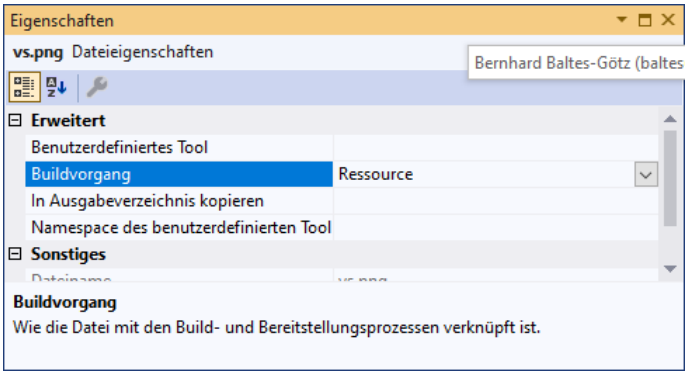
Auf die **Button**-Oberfläche werden nebeneinander ein **Image**- und ein **TextBlock**-Objekt platziert, die von einem **StackPanel**-Layoutcontainer verwaltet werden (siehe Abschnitt 12.6.3).

Die **Source**-Eigenschaft des **Image**-Objekts wird folgendermaßen über die Datei **vs.png** mit dem Bild versorgt:

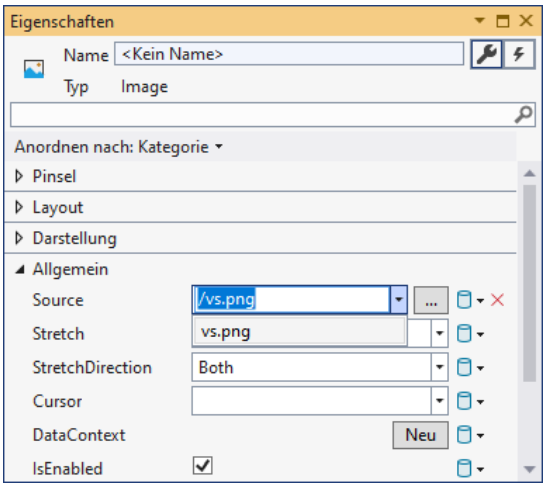
- Bilddatei per Windows-Explorer in den Projektordner kopieren, z. B.:



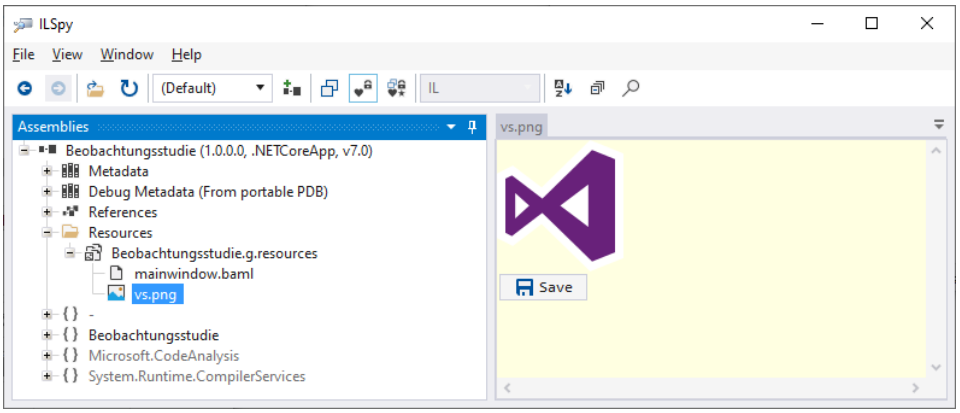
- Im Eigenschaftendialog zum **Projekt-Explorer** – Eintrag der Bilddatei **Resource** als **Buildvorgang** wählen, z. B.:



- Im Eigenschaftendialog zum **Image**-Objekt die Bilddatei als Wert für die Eigenschaft **Source** eintragen, z. B.:



Eine Inspektion mit dem Programm ILSpy zeigt, dass die Datei **vs.png** als Ressource in das Assembly aufgenommen wurde:



Wenn nach dem Befolgen der obigen Ratschläge ein Bild zwar im WPF-Designer zu sehen ist, aber nicht im laufenden Programm, dann sollten die folgenden Maßnahmen helfen:

- Schließen Sie das Projekt.
- Löschen Sie die Projekt-Unterordner **obj** und **bin**.
- Öffnen sie das Projekt wieder.

Wie im Abschnitt 12.3.2.4 beschrieben, eignet sich die XAML-Attributsyntax auch zur Registrierung von Ereignisbehandlungsmethoden, wobei der Methodenname ohne Parameterklammern anzugeben ist.

Besondere Aufmerksamkeit verdient die Registrierung der **Click**-Ereignisbehandlungsmethode beim **Window**- bzw. **Grid**-Objekt, z. B.:

```
<Window x:Class="Beobachtungsstudie.MainWindow"
   . . .
        ButtonBase.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 sogenanntes **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 Registrierung

```
ButtonBase.Click="HandleClick"
```

wurde direkt in den XAML-Code eingetragen. In der Ereignisliste des **Eigenschaften**-Fensters zum **Window**-Objekt fehlen angefügte Ereignisse.

Die angefügten Ereignisse ermöglichen die im Abschnitt 12.4.2 beschriebene rationelle Behandlung von Blasenereignissen: Befinden sich mehrere Steuerelemente desselben Typs in einem Layoutcontainer, dann 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 festgestellt werden.

Ein Visual Studio - Projekt mit der Beobachtungsstudie zu den Routingereignissen ist im folgenden Ordner zu finden:

…\BspUeb\WPF\Routingereignisse\Beobachtungsstudie

12.4.4 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, dann wird die statische Behandlungsmethode *vor* der objektbezogenen ausgeführt.

Zur Demonstration leiten wir von **Button** eine Klasse namens `MaiButton` ab, die eine zur **ClickEvent**-Behandlung geeignete statische Methode definiert und im statischen Konstruktor über die statische Methode **RegisterClassHandler()** der Klasse **EventManager** beim WPF-Ereignissystem zur **ClickEvent**-Behandlung für Objekte vom `MaiButton` registriert:

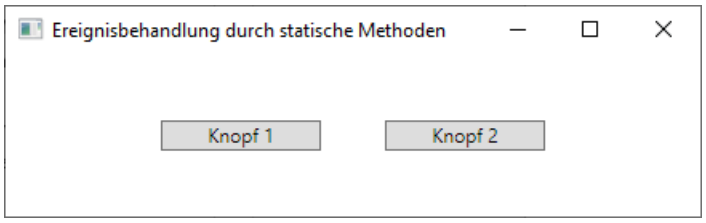
```
using System.Windows;
using System.Windows.Controls;

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 registriert, das dank Wrapper (vgl. Abschnitt 12.4.5) als CLR-Ereignis namens **Click** angesprochen werden kann.

Die folgende WPF-Anwendung (handgestrickt ohne XAML, vgl. Abschnitt 12.2.1) demonstriert die Routingereignisbehandlung durch eine statische Methode:

```
using System;
using System.Windows;
using System.Windows.Controls;

class StaticEventHandling : Window {
    StaticEventHandling() {
        Height = 140; Width = 450;
        Title = "Ereignisbehandlung durch statische Methoden";
        var layoutManager = new StackPanel {
            Orientation = Orientation.Horizontal,
            HorizontalAlignment = HorizontalAlignment.Center,
            VerticalAlignment = VerticalAlignment.Center
        };
        Content = layoutManager;
        var k1 = new MaiButton { Content = "Knopf 1", Width = 100,
                                 Margin = new Thickness(20) };
        layoutManager.Children.Add(k1);
        var k2 = new MaiButton { Content = "Knopf 2", Width = 100,
                                 Margin = new Thickness(20) };
        layoutManager.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());
    }
}
```

Das Programm präsentiert zwei Befehlsschalter aus der Klasse `MaiButton`,



wobei der linke Schalter über eine eigene **Click**-Behandlungsmethode verfügt. Beim Klick auf einen beliebigen Schalter findet zunächst die klassenbezogene Ereignisbehandlung statt:

Wurde der linke Schalter getroffen, dann folgt noch die objektbezogene Ereignisbehandlung:

Ein Visual Studio - Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

…\BspUeb\WPF\Routingereignisse\Ereignisbehandlung durch statische Methoden

12.4.5 Routingereignisse definieren

Der aktuelle Abschnitt ist primär relevant für Entwickler, die Routingereignisse definieren wollen. Er kann also vorläufig ausgelassen werden, wenn …

- keine selbst definierten Routingereignisse benötigt werden,
- und kein Interesse an Hintergrundinformationen zur Beziehung zwischen Routingereignissen und CLR-Ereignissen besteht.

In der folgenden Anweisung aus der abstrakten BCL-Klasse **ButtonBase** (Namensraum **System.Windows.Controls.Primitives**), von der z. B. die Klasse **Button** abstammt, wird das **Click**-Routingereignis (ein Objekt der Klasse **RoutedEvent**) durch die statische Methode **RegisterRoutedEvent()** der Klasse **EventManager** (Namensraum **System.Windows**) erzeugt und registriert:[1]

```
public static readonly RoutedEvent ClickEvent = EventManager.RegisterRoutedEvent(
   "Click", RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(ButtonBase));
```

Die Objektadresse landet in einem schreibgeschützten, öffentlichen und statischen Feld vom Typ **RoutedEvent**. Sein Name **ClickEvent** endet eine Konvention beachtend mit *Event*.

Beim Registrieren sind zu einem Routingereignis anzugeben:

- der Name
- die Routingstrategie (siehe Abschnitt 12.4.1)
- die von Behandlungsmethoden zu erfüllende Delegatenklasse
- der Datentyp des Eigentümers
  Hier gibt man die Klasse an, die das Routingereignis registriert.

In der Regel wird ein statisches Feld vom Typ **RoutedEvent** durch ein (als *Wrapper* bezeichnetes) instanzbezogenes CLR-Ereignis ergänzt, sodass die gewohnte Logik und Syntax zum Registrieren von Ereignisbehandlungsmethoden bei einem Objekt anwendbar ist. Im Beispiel wird von der Klasse **ButtonBase** das CLR-Ereignis **Click** mit der im Abschnitt 10.2.1 beschriebenen Syntax definiert, wobei die explizit realisierten **add**- und **remove**-Methoden die Verbindung zum **RoutedEvent**-Objekt herstellen:

```
public event RoutedEventHandler Click {
   add {
      AddHandler(ClickEvent, value);
   }
   remove {
      RemoveHandler(ClickEvent, value);
   }
}
```

Aufgerufen wird das **Click**-Routingereignis in der **ButtonBase**-Methode **OnClick()**, wobei aber *nicht* wie bei einem CLR-Ereignis das Delegatenobjekt aufgerufen wird (vgl. Abschnitt 10.2.3), sondern die **UIElement**-Methode **RaiseEvent()**:

---

[1] Statt zum Bezug des BCL-Quellcodes auf den Abschnitt 2.6.4 zu verweisen, geben wir diesmal den Link zur Klasse **ButtonBase** direkt an:
https://source.dot.net/#PresentationFramework/System/Windows/Controls/Primitives/ButtonBase.cs

```
protected virtual void OnClick() {
   RoutedEventArgs newEvent = new RoutedEventArgs(ButtonBase.ClickEvent, this);
   RaiseEvent(newEvent);
   MS.Internal.Commands.CommandHelpers.ExecuteCommandSource(this);
}
```

12.5 Abhängigkeitseigenschaften

Die WPF-Designer haben nicht nur die CLR-Ereignisse durch die leistungsfähigeren Routingereignisse ergänzt, sondern auch zu den CLR-Eigenschaften, die meist aus einer Instanzvariablen und einem Methodenpaar für den lesenden bzw. schreibenden Zugriff bestehen (vgl. Abschnitt 5.5), mit den sogenannten *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 ein elterliches Steuerelement, Datenbindung, WPF-Stile oder Animationen).

Um die neue Funktionalität zu realisieren, wurde keine .NET - Programmiersprache angepasst (von XAML mal abgesehen), sondern die WPF-Bibliothek entsprechend ausgestattet. Alle Klassen, die Abhängigkeitseigenschaften verwenden sollen, müssen von der Klasse **DependencyObject** im Namensraum **System.Windows** abstammen.

Die meisten Eigenschaften von WPF-Steuerelementen sind als Abhängigkeitseigenschaften realisiert, können aber auch wie gewöhnliche CLR-Eigenschaften verwendet werden. Obwohl wir in Beispielen schon etliche WPF-Abhängigkeitseigenschaften verwendet haben, ist bisher nur in einem Fall eine Besonderheit gegenüber den gewöhnlichen CLR-Eigenschaften aufgefallen.[1]

Ob es sich bei einer Eigenschaft einer WPF-Steuerelementklasse um eine Abhängigkeitseigenschaft handelt, verrät die Anwesenheit der **Dependency Property Information** in der BCL-Dokumentation, z. B.:

Dependency Property Information

Identifier field	BackgroundProperty
Metadata properties set to `true`	AffectsRender, SubPropertiesDoNotAffectRender

Außerdem ist zu einer Abhängigkeitseigenschaft stets ein statisches Feld mit dem Zugriffsschutz **public** und einem Namen mit dem Wort *Property* als terminalem Bestandteil vorhanden (z. B. **BackgroundProperty**, vgl. Abschnitt 12.5.4). Inspiziert man z. B. die über 130 Eigenschaften der Klasse **Button**, dann finden sich nur wenige CLR-Eigenschaften.

Die allgemeine Technik und Funktionsvielfalt der Abhängigkeitseigenschaften darzustellen (siehe z. B. Huber 2017, MacDonald 2012), würde den Rahmen des Manuskripts sprengen. Wir konzentrieren uns auf drei schon beim WPF-Einstieg unvermeidliche bzw. nützliche Funktionen des WPF-Eigenschaftssystems:

[1] Im Abschnitt 6.8.6 kamen bei der Entwicklung des RSS-Feed – Readers zwei Optionen von Abhängigkeitseigenschaften im Vorgriff auf das WPF-Kapitel zum Einsatz (eine angefügte Eigenschaft und die Datenbindung).

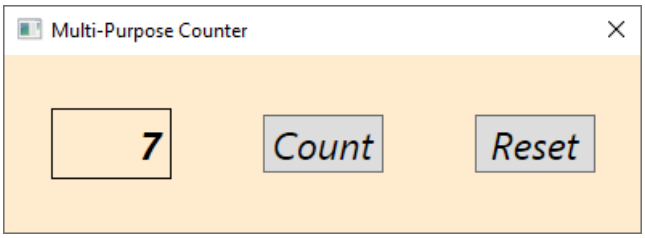
- Bei vielen WPF-Abhängigkeitseigenschaften wird die einem GUI-Element zugewiesene Ausprägung (z. B. die Schriftgröße) auf eingeschachtelte GUI-Elemente übertragen („vererbt“), falls keine andere Eigenschaftsquelle dominiert (siehe Abschnitt 12.5.1).
- Mit den sogenannten *angefügten Eigenschaften* werden spezielle Abhängigkeitseigenschaften vorgestellt, die zur Verwaltung der in Layoutcontainern enthaltenen Steuerelemente unverzichtbar und auch für andere Zwecke nützlich sind (siehe Abschnitt 12.5.2).
- Die Datenbindungstechnik der WPF-Bibliothek setzt auf der Empfängerseite eine Abhängigkeitseigenschaft voraus, und auch quellseitig lässt sich eine Abhängigkeitseigenschaft unproblematisch verwenden (siehe Abschnitt 12.5.3).

Zu den mit Hilfe von Abhängigkeitseigenschaften realisierten WPF-Optionen, die im Manuskript aus Zeitgründen fehlen, gehören die Stile und Animationen.

Trotz der Konzentration auf die Anwendung der in WPF-Typen zahlreich vorhandenen Abhängigkeitseigenschaften bieten wir im Abschnitt 12.5.4 auch einige Informationen für Entwickler, die eigene Abhängigkeitseigenschaften definieren wollen.

12.5.1 Eigenschaftsübertragung auf eingeschachtelte Elemente

Bei vielen WPF-Abhängigkeitseigenschaften findet eine sogenannte *Eigenschaftswertvererbung* statt, d. h. die an ein Element in der GUI-Hierarchie eines Fensters vergebene Ausprägung wird auf eingeschachtelte Elemente übertragen. Allerdings hängt die Eigenschaftsausprägung bei einem konkreten eingeschachtelten Element eventuell noch von anderen Quellen ab. Im folgenden Beispiel mit einem **Label**-Objekt und zwei **Button**-Objekten in einem **Grid**-Layoutcontainer mit drei Spalten (vgl. Abschnitt 12.7.4 über Befehlsschalter)



werden für das XAML-Wurzelelement **Window** eine Hintergrundfarbe, eine Schriftgröße und ein Schriftstil durch die Abhängigkeitseigenschaften **Background**, **FontSize** und **FontStyle** (alle definiert in der Klasse **Control** aus dem Namensraum **System.Windows.Controls**) 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"
        Loaded="Window_Loaded"
        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>
```

Die beiden Schriftattribute werden an die Steuerelemente „vererbt“, die Hintergrundfarbe hingegen nicht. Weil das **Label**-Objekt als Hintergrundfarbe die transparente Voreinstellung besitzt, sieht es so aus, als würde es die Hintergrundfarbe des Fensters übernehmen.

Ob die Werte einer Eigenschaft auf eingeschachtelte Elemente übertragen werden, hängt davon ab, ob in ihrem Metadaten-Objekt die (Meta-)Eigenschaft **Inherits** auf **true** gesetzt ist. Bei der Eigenschaft **FontSize** ist das der Fall, wie eine Inspektion der BCL-Dokumentation zeigt:

Dependency Property Information

Identifier field	FontSizeProperty
Metadata properties set to `true`	AffectsMeasure, AffectsRender, Inherits

Dasselbe gilt für die Eigenschaft **FontStyle**. Von der Eigenschaft **Background** wird die Übertragung von Eigenschaftswerten hingegen *nicht* unterstützt:

Dependency Property Information

Identifier field	BackgroundProperty
Metadata properties set to `true`	AffectsRender, SubPropertiesDoNotAffectRender

Es ist für die Übertragung der Eigenschaftswerte vom **Window**-Objekt auf die **Button**-Steuerelemente *kein* Problem, dass der zwischengeschaltete **Grid**-Container die Eigenschaften **FontSize** und **FontStyle** nicht kennt.

Man spricht bei der Übertragung von Eigenschaftsausprägungen auf eingeschachtelte Elemente auch von *Eigenschaftswertvererbung* (engl. *property value inheritance*), doch hat diese Spezialität des WPF-Eigenschaftssystems *nichts* mit der im Kapitel 7 beschriebenen Ableitung von Klassen zu tun.[1]

[1] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/properties/property-value-inheritance

Ein Visual Studio - Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

…\BspUeb\WPF\Abhängigkeitseigenschaften\Eigenschaftsübertragung

12.5.2 Angefügte Eigenschaften

Bei der Visual Studio - Projektvorlagen **WPF-Anwendung** kommt per Voreinstellung ein **Grid**-Layoutcontainer zum Einsatz, der im allgemeinen mehrere Zeilen und/oder Spalten zur Platzierung der enthaltenen Steuerelemente verwaltet. Im Abschnitt 12.6.1.1 ist zu erfahren, wie man über **ColumnDefinition**- und **RowDefinition**-Elemente die Struktur eines **Grid**-Containers festlegt. Fügt man ein Steuerelement ohne Ortsangabe in einen **Grid**-Container ein, dann landet es in der 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 null-basierten Nummer der Zeile bzw. Spalte zu versorgen. Hier wird ein **Button**-Objekt in die Zelle (1, 1) einer **Grid**-Matrix mit drei Spalten und zwei Zeilen gesteckt:

```
<Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition /> <ColumnDefinition /> <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition /> <RowDefinition />
    </Grid.RowDefinitions>
    <Button Name="butt" Content="Knopf"
            HorizontalAlignment="Center" VerticalAlignment="Center"
            Grid.Row="1" Grid.Column="1" />
    . . .
</Grid>
```

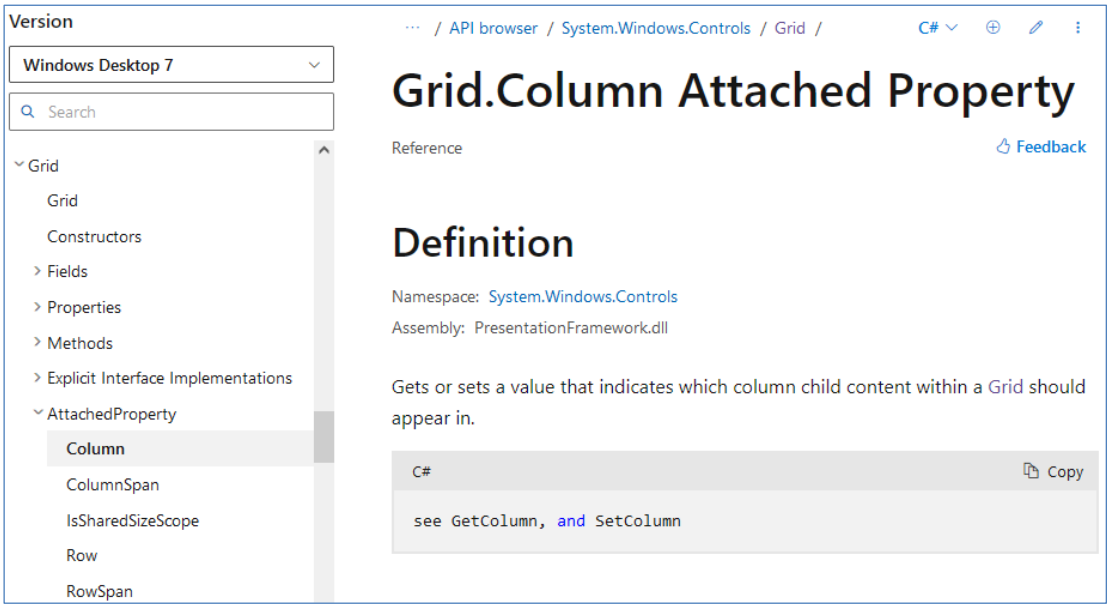
Während sich in der Steuerelementklasse **System.Windows.Controls.Button** zu XAML-Attributen wie **Content**, **HorizontalAlignment** etc. jeweils eine korrespondierende Eigenschaft findet, sucht man in der Klassendefinition die Eigenschaften **Grid.Row** und **Grid.Column** oder auch **Row** und **Column** vergeblich.

Grid.Row und **Grid.Column** sind sogenannte *angefügte Eigenschaften* (engl.: *attached properties*), die von der Klasse **Grid** zur Verfügung gestellt werden, damit die im **Grid**-Layoutcontainer verwalteten GUI-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**).
- Eine angefügte Eigenschaft kann im C# - Quellcode *nicht* wie eine normale CLR-Eigenschaft verwendet werden. Stattdessen sind in der anbietenden Klasse statische **Set**- und **Get**-Methoden vorhanden, was gleich für das Beispiel demonstriert wird.

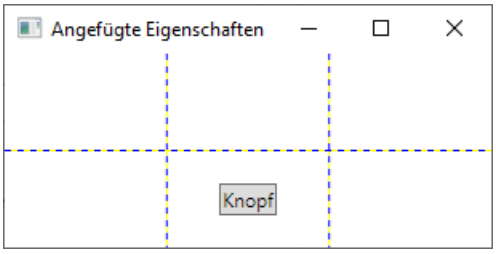
In der englischen BCL-Dokumentation erscheinen die von einer Klasse angebotenen angefügten Eigenschaften in der eigenen Kategorie **AttachedProperty**, z. B.:



Im Beispiel korrespondieren zur 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 Konstruktor der folgenden Fensterklasse wird *ohne* XAML-Hilfe ein (3 × 2) - **Grid** erstellt und ein **Button**-Objekt in die Zelle (1, 1) gesteckt, um die Verwendung 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";
      var grid = new Grid();
      grid.ColumnDefinitions.Add(new ColumnDefinition());
      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;

      var butt = new Button {
         Content = "Knopf",
         HorizontalAlignment = HorizontalAlignment.Center,
         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 im Programm 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 ein die angefügte Eigenschaft beschreibendes Objekt (siehe Abschnitt 12.5.4) als erster Parameter übergeben wird. Hier ist die statische **Grid**-Methode **SetRow()** zu sehen:[1]

```
public static void SetRow(UIElement element, int value) {
   if (element == null) {
      throw new ArgumentNullException("element");
   }
   element.SetValue(RowProperty, value);
}
```

Weil die **DependencyObject**-Methoden **SetValue()** und **GetValue()** öffentlich sind, kann man sie auch direkt aufrufen und somit die statischen Methoden der angefügten Eigenschaft übergehen. Im Beispiel lassen sich die Anweisungen

```
Grid.SetRow(butt, 1);
Grid.SetColumn(butt, 1);
```

ersetzen durch:

```
butt.SetValue(Grid.RowProperty, 1);
butt.SetValue(Grid.ColumnProperty, 1);
```

Von den angefügten Eigenschaften profitiert die Erweiterbarkeit der WPF-Technik (MacDonald 2012, S. 35):

- Die Steuerelementklassen werden nicht mit Eigenschaften belastet, die lediglich in bestimmten Kontexten relevant sind.
- Man kann jederzeit weitere Layoutcontainer entwickeln mit angefügten Eigenschaften zur Verwaltung ihrer Elemente.

Ein Visual Studio - Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

…\BspUeb\WPF\Abhängigkeitseigenschaften\Angefügte Eigenschaften

[1] Wie man den BCL-Quellcode einsehen kann, erläutert der Abschnitt 2.6.4.

12.5.3 Datenbindung zwischen zwei Steuerelementen

In sehr vielen WPF-Anwendungen spielt die zuverlässige und unaufwändige Kopplung zwischen

- Datenbeständen in CLR-Objekten einerseits
- und GUI-Elementen zur Anzeige und Modifikation der Daten andererseits

eine wichtige Rolle. Im RSS-Feed Reeder, den wir im Abschnitt 6.8 entwickelt haben, stellt ein **ListBox**-Steuerelement die in einer **List**<`RssItem`> - Kollektion namens `items` aufbewahrten Elemente der von uns (mit Assistentenunterstützung) definierten Klasse `RssItem` dar:

```
internal class RssItem {
   public string Title { get; internal set; }
   public string Description { get; internal set; }
   public string Url { get; internal set; }
}
```

Der **ListBox**-Eigenschaft **ItemsSource** wird die externe Kollektion zugewiesen:

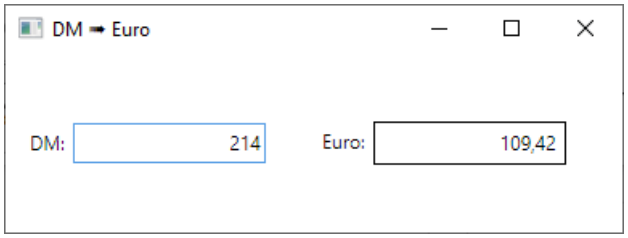
```
listBox.ItemsSource = items;
```

Aus einem Element der Quelle (mit den CLR-Eigenschaften `Title` und `Description`) beziehen zwei vom **ListBox**-Objekt aufgrund einer **DataTemplate**-Designspezifikation angezeigte **TextBlock**-Objekte ihre Daten. Dazu wird jeweils die **TextBlock**-Abhängigkeitseigenschaft **Text** über eine XAML – Markup-Erweiterung vom Typ **Binding** (siehe Abschnitt 12.3.2.3.6) mit einer CLR-Eigenschaft der Quelle verbunden:

```
<ListBox x:Name="listBox" Margin="10,49,10,10">
  <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>
```

Offenbar macht die WPF-Datenbindungstechnik bei einer Realisation per XAML erfreulich wenig Aufwand. Dabei besteht die entscheidende Voraussetzung, dass es sich beim Empfänger um eine Abhängigkeitseigenschaft handeln muss.

Wie das Beispiel zeigt, benötigt man bei einer Datenbindung quellseitig *keine* Abhängigkeitseigenschaft. Vielfach bietet sich aber eine Datenbindung zwischen zwei Abhängigkeitseigenschaften an, die sich in verschiedenen Steuerelementen eines Programms befinden, was im nächsten Beispiel vorgeführt wird. Das folgende Programm konvertiert den in einem **TextBox**-Steuerelement erscheinenden DM-Betrag spontan in einen Euro-Betrag, den ein **Label**-Steuerelement anzeigt:



Dass im XAML-Code

```
<Window x:Class="DmToEuroBinding.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DM &#x27a0; Euro" Height="150" Width="400" Loaded="Window_Loaded" >
    <Grid Margin="10">
        <Grid.ColumnDefinitions>
            <ColumnDefinition/> <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <StackPanel Orientation="Horizontal" VerticalAlignment="Center">
            <Label Content="DM:" />
            <TextBox Name="tbEingabe" VerticalContentAlignment="Center" MinWidth="120"
             TextAlignment="Right" />
        </StackPanel>
        <StackPanel Orientation="Horizontal" Grid.Column="1" VerticalAlignment="Center">
            <Label Content="Euro:"/>
            <Label BorderBrush="Black" BorderThickness="1" MinWidth="120"
                   HorizontalContentAlignment="Right"
                   Content="{Binding ElementName=tbEingabe, Path=Text,
                            Converter={StaticResource DmToEuroConverter}}" />
        </StackPanel>
    </Grid>
</Window>
```

ein **StackPanel**-Layoutcontainer in einen **Grid**-Layoutcontainer geschachtelt wird, ignorieren wir vorläufig (siehe Abschnitt 12.6.7). Relevant ist vor allem das **Label**-Steuerelement auf der rechten Seite des horizontalen **StackPanel**-Containers in der zweiten Spalte des **Grid**-Containers. Seine **Content**-Eigenschaft wird über ein **Binding**-Objekt mit einer Zeichenfolge versorgt:

- Die **Binding**-Eigenschaft **ElementName** benennt als Quelle das **TextBox**-Element mit dem Namen `tbEingabe`.
- Die **Binding**-Eigenschaft **Path** benennt die zu verwendende Eigenschaft des Quellobjekts.

Die **Text**-Eigenschaft des Quellelements wird aber nicht direkt verwendet, sondern konvertiert durch die in der Code-Behind - Datei **MainWindow.xaml.cs** definierte Klasse `DmToEuroConverter`, die das Interface **IValueConverter** implementiert, das u. a. die Methode **Convert()** vorschreibt:

```
using System;
using System.Windows;
using System.Windows.Data;

namespace DmToEuroBinding;
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
    }

    private void Window_Loaded(object sender, RoutedEventArgs e) {
        tbEingabe.Focus();
    }
}
```

```
public class DmToEuroConverter : IValueConverter {
    const double factor = 1.95583;
    public object Convert(object value, Type targetType, object parameter,
                          System.Globalization.CultureInfo culture) {
        string s = "";
        try {
            s = string.IsNullOrEmpty((string)value) ? "" :
                (System.Convert.ToDouble(value) / factor).ToString("0.00");
        } catch { }
        return s;
    }
    public object ConvertBack(object value, Type targetType, object parameter,
                              System.Globalization.CultureInfo culture) {
        throw new NotImplementedException();
    }
}
```

Der Konditionaloperator in der **return**-Anweisung der `DmToEuroConverter`-Methode **Convert()** liefert ...

- eine leere Zeichenfolge, wenn die **Text**-Eigenschaft der Quelle entweder leer oder nicht als **double**-Wert interpretierbar ist,
- eine Zeichenfolge mit dem Euro-Betrag, wenn die Eingabe konvertierbar ist.

In der Datei **App.xaml** wird ein Objekt der Klasse `DmToEuroConverter` als Ressource vereinbart:

```
<Application x:Class="DmToEuroBinding.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:DmToEuroBinding"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <local:DmToEuroConverter x:Key="DmToEuroConverter"/>
    </Application.Resources>
</Application>
```

Dieses Objekt wird im XAML-Code zum Anwendungsfenster der **Converter**-Eigenschaft des **Binding**-Objekts zugewiesen:

```
<Label BorderBrush="Black" BorderThickness="1" MinWidth="120"
       HorizontalContentAlignment="Right"
       Content="{Binding ElementName=eingabe, Path=Text,
                 Converter={StaticResource DmToEuroConverter}}" />
```

Im folgenden Ordner befindet sich ein Visual Studio – Projekt mit dem Konvertierungsprogramm per Datenbindungstechnik:

…\BspUeb\WPF\Abhängigkeitseigenschaften\DmToEuroBinding

12.5.4 Abhängigkeitseigenschaften definieren

Der aktuelle Abschnitt ist primär relevant für Entwickler, die Abhängigkeitseigenschaften in eigenen Typen definieren wollen. Er kann also vorläufig ausgelassen werden, wenn …

- keine selbst definierten Abhängigkeitseigenschaften benötigt werden,
- und kein Interesse an Hintergrundinformationen zur Beziehung zwischen den Abhängigkeitseigenschaften und den CLR-Eigenschaften besteht.

In der folgenden Anweisung aus der BCL-Klasse **Image** (Namensraum **System.Windows.Controls**) wird die Abhängigkeitseigenschaft **Source** durch die statische

Methode **Register ()** der Klasse **DependencyProperty** (Namensraum **System.Windows**) erzeugt und registriert:[1]

```
public static readonly DependencyProperty SourceProperty =
    DependencyProperty.Register(
            "Source",
            typeof(ImageSource),
            typeof(Image),
            new FrameworkPropertyMetadata(
                    null,
                    FrameworkPropertyMetadataOptions.AffectsMeasure |
                      FrameworkPropertyMetadataOptions.AffectsRender,
                    new PropertyChangedCallback(OnSourceChanged),
                    null),
            null);
```

Es entsteht ein beschreibendes Objekt aus der Klasse **DependencyProperty**, dessen Adresse in einem schreibgeschützten, öffentlichen und statischen Feld landet. Der Feldname **SourceProperty** endet eine Konvention beachtend mit *Property*.

Beim Aufruf der verwendeten **Register()** – Überladung sind zu einer Abhängigkeitseigenschaft anzugeben:

- der Name
- der Datentyp
- der Datentyp des Eigentümers
  Hier gibt man die Klasse an, die die Abhängigkeitseigenschaft registriert.
- ein Objekt vom Typ **PropertyMetadata** mit Metadaten
  In den Metadaten kann z. B. festgelegt werden, dass eine Eigenschaftsvererbung stattfinden soll (siehe Abschnitt 12.5.1), was bei der Eigenschaft **Source** sinnlos ist, bei der Eigenschaft **FontSize** hingegen nützlich.
- die von einer Rückrufmethode zur Validierung zugewiesener Werte zu erfüllende Delegatenklasse

Das über ein statisches Feld vom Typ **DependencyProperty** ansprechbare Beschreibungsobjekt zu einer Abhängigkeitseigenschaft wird durch eine (manchmal nicht ganz treffend als *Wrapper* bezeichnete) instanzbezogene CLR-Eigenschaft ergänzt, sodass die gewohnte Logik und Syntax zum Eigenschaftszugriff anwendbar ist. Im Beispiel wird von der Klasse **Image** die CLR-Eigenschaft **Source** definiert:

```
public ImageSource Source {
    get { return (ImageSource)GetValue(SourceProperty); }
    set { SetValue(SourceProperty, value); }
}
```

Für die Verbindung zur Abhängigkeitseigenschaft sorgen die **DependencyObject**-Methoden **GetValue()** und **SetValue()**. Aus diesem Grund muss jede Klasse, die Abhängigkeitseigenschaften definiert, von der Klasse **DependencyObject** (im Namensraum **System.Windows**) abstammen (Petzold 2008). In der Definition der CLR-Eigenschaft zu einer Abhängigkeitseigenschaft sollte über die beiden Methodenaufrufe hinaus kein weiterer Code enthalten sein.

---

[1] Statt zum Bezug des BCL-Quellcodes auf den Abschnitt 2.6.4 zu verweisen, geben wir diesmal den Link zur Klasse **Image** direkt an:
https://source.dot.net/#PresentationFramework/System/Windows/Controls/Image.cs

12.6 Layoutcontainer

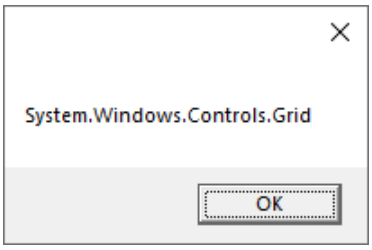
Zur Verwaltung der mehr oder weniger zahlreichen Steuerelemente in einem Fenster einer WPF-Anwendung dienen Layoutcontainer, die z. B. auf eine geänderte Größe und/oder auf ein geändertes Seitenverhältnis des Fensters mit einer Neuberechnung der Positionen und Größen der Steuerelemente reagieren und so ein *responsives Layout* ermöglichen. Bei einer neuen WPF-Anwendung wird vom Visual Studio für das Hauptfenster ein *Wurzel-Layoutcontainer* (engl.: *root layout container*) aus der Klasse **Grid** (Namensraum **System.Windows.Controls**) vorgeschlagen. Dies zeigt ein Blick auf den XAML-Code des Hauptfensters:[1]

```
<Window x:Class="WindowContent.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>
```

Übernimmt man den Vorschlag, dann entsteht beim Programmstart ein **Grid**-Objekt, das der **Content**-Eigenschaft des Hauptfensters zugewiesen wird. Um dieses Detail zu demonstrieren, wird in der folgenden Fensterklassendefinition eine Behandlungsmethode zum **Window**-Ereignis **Loaded** erstellt und registriert:

```
using System.Windows;
namespace WindowContent;
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
        Loaded += Window_Loaded;
    }
    private void Window_Loaded(object sender, RoutedEventArgs e) {
        MessageBox.Show(Content.ToString());
    }
}
```

Beim Programmstart erfährt man:



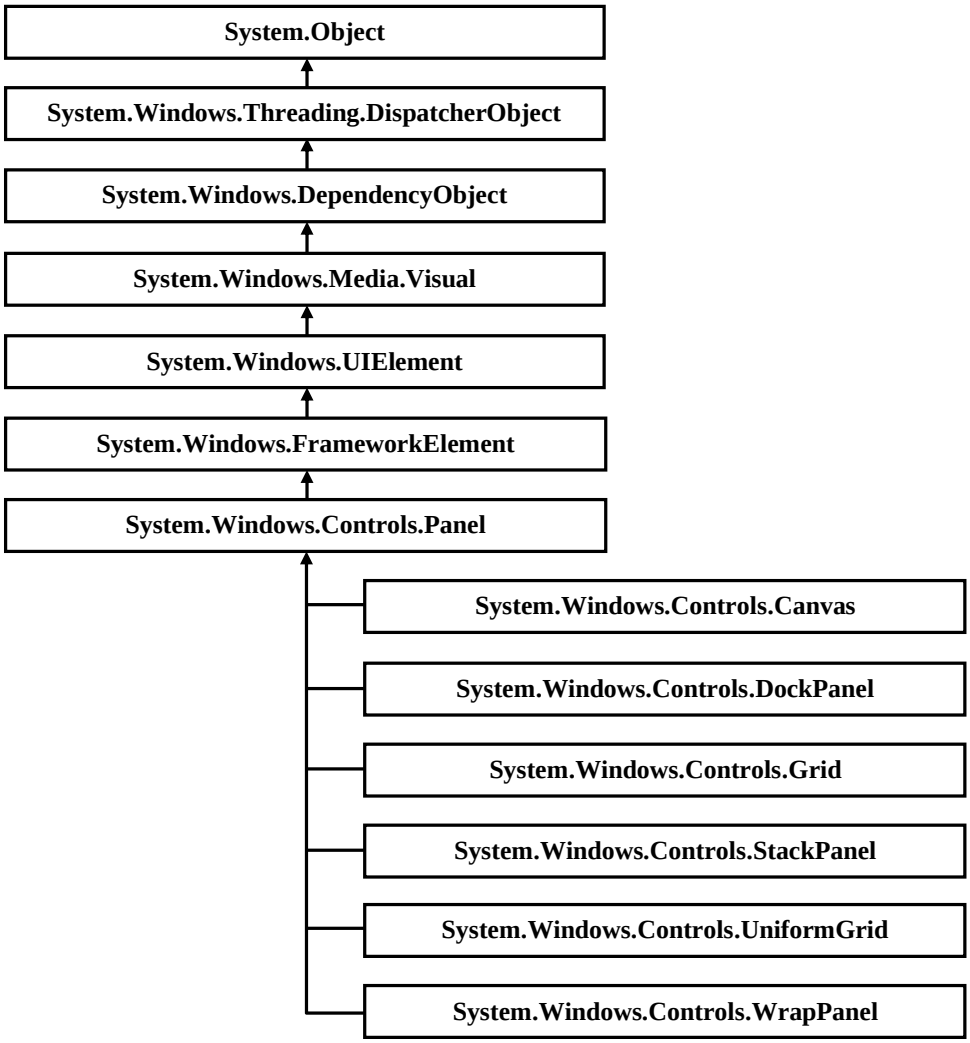
Wenn wir im WPF-Designer Steuerelemente per Drag & Drop aus der **Toolbox** auf das Hauptfenster befördern, dann landen diese im **Grid**-Element des XAML-Codes, z. B.:

[1] Der Übersichtlichkeit halber wurden überflüssige XAML-Namensraumdeklarationen entfernt.

```
<Window x:Class="WindowContent.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>
```

Von der Gitterstruktur, die der Klassenname **Grid** erwarten lässt, ist noch nichts zu sehen. Initial besitzt der Layoutmanager nur *eine* Zeile und *eine* Spalte, und eingefügte Steuerelemente landen in der **Grid**-Zelle (0, 0). Wie man die **Grid**-Flexibilität nutzt, erfahren Sie im Abschnitt 12.6.1.

Alle WPF-Layoutcontainer stammen von der Klasse **System.Windows.Controls.Panel** ab. Hier sind die meistbenutzten Klassen mit ihrem Stammbaum zu sehen:



Von den Mitgliedern, die ein Layoutcontainer von seiner Basisklasse **Panel** erbt, ist besonders die Eigenschaft **Children** zu erwähnen, die auf ein Objekt der Klasse **UIElementCollection** zeigt, das die im Container enthaltenen **UIElement**-Objekte verwaltet. Es ist die Inhaltseigenschaft der Klasse **Panel** (vgl. Abschnitt 12.3.2.3.3):

```
[ContentProperty("Children")]
public abstract class Panel : FrameworkElement, IAddChild {
 . . .
   public UIElementCollection Children {
      get {
         return InternalChildren;
      }
   }
. . .
}
```

Für die Top-Level - Fenster einer WPF-Anwendung verwendet man in der Regel ein **Grid**- oder ein **DockPanel**-Objekt als Wurzel-Layoutcontainer. Ein flexibles Fensterdesign erfordert oft geschachtelte Layoutcontainer, und dabei kommen auch andere **Panel**-Ableitungen zum Einsatz.

12.6.1 Grid

In diesem Abschnitt werden einige Details zum voreingestellten Wurzel-Layoutcontainer einer WPF-Anwendung beschrieben.

12.6.1.1 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 12.3.2.3 zur XAML-Syntax für Eigenschaftselemente).

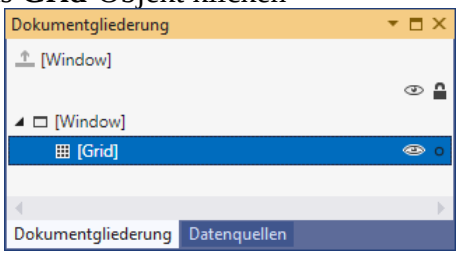
Im Element **Grid.RowDefinitions** werden einzelne Zeilen durch **RowDefinition**-Elemente vereinbart, z. B.:

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
   </Grid.RowDefinitions>
</Grid>
```

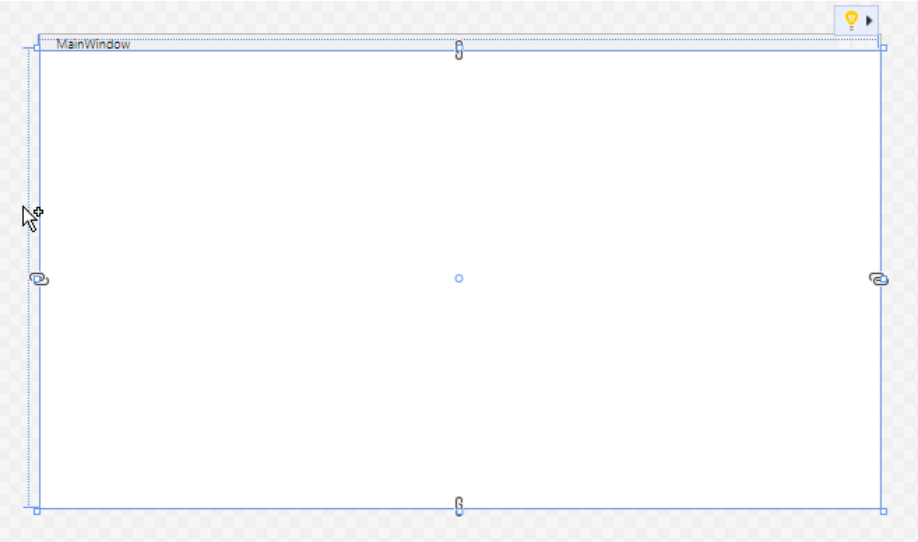
Die **Grid**-Eigenschaft **RowDefinitions** ist vom Typ **System.Windows.Controls.RowDefinitionCollection**, und die XAML-Kollektionssyntax erlaubt es in diesem Fall, auf ein Instanzelement zum Kollektionsobjekt zu verzichten (siehe Abschnitt 12.3.2.3.5).

Die XAML-Elemente zur Definition der Zeilenstruktur kann man auch über den WPF-Designer im Visual Studio erstellen. Markieren Sie zunächst das **Grid**-Element mit einer von den folgenden Techniken:

- Mausklick auf das **Grid**-Element im XAML-Fenster
- **Dokumentengliederung** öffnen (z. B. mit **Ansicht > Weitere Fenster > Dokumentengliederung**) und auf das **Grid**-Objekt klicken



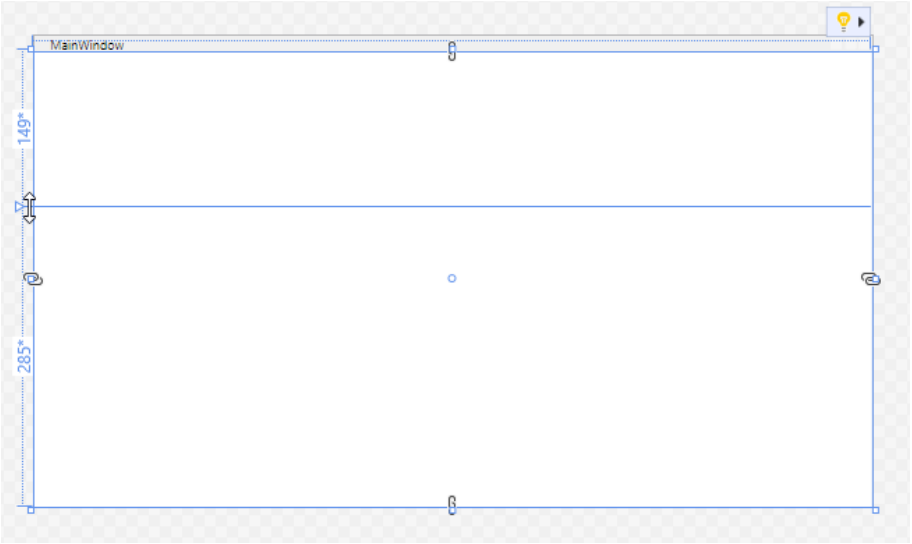
Setzen Sie dann mit dem Mausinstrument einen Klick auf die am linken Rand des **Grid**-Elements befindliche Zeilendefinitionszone, z. B.:



Daraufhin wird vom WPF-Designer eine Trennlinie eingefügt, und im XAML-Code erscheinen:

- ein Eigenschaftselement von Typ **Grid.RowDefinitions**
- zwei **RowDefinition**-Elemente mit **Height**-Werten

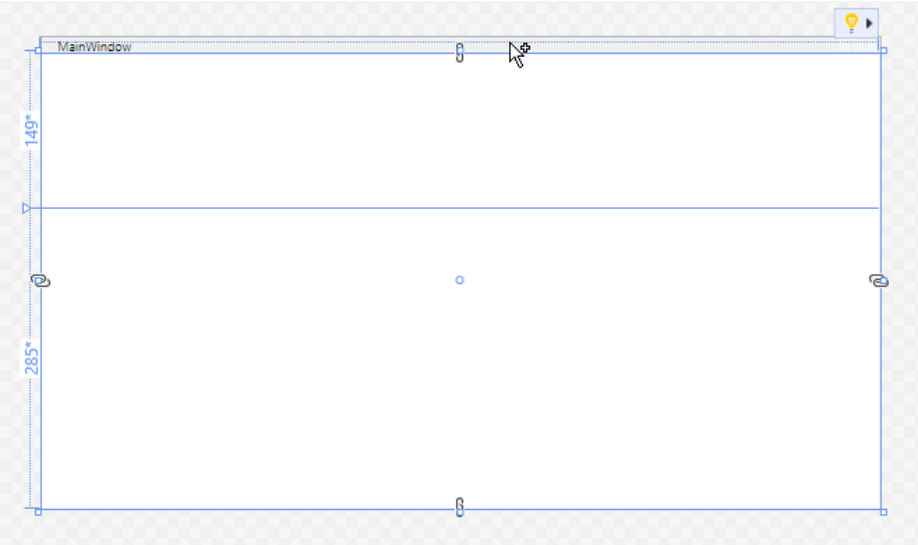
Die Trennlinie lässt sich anschließend mit Effekt auf die **Height**-Werte der beiden Zeilen per Maus verschieben, z. B.:



Aus den beschriebenen Mausaktivitäten resultiert das folgende XAML-Ergebnis:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="149*"/>
    <RowDefinition Height="285*"/>
  </Grid.RowDefinitions>
</Grid>
```

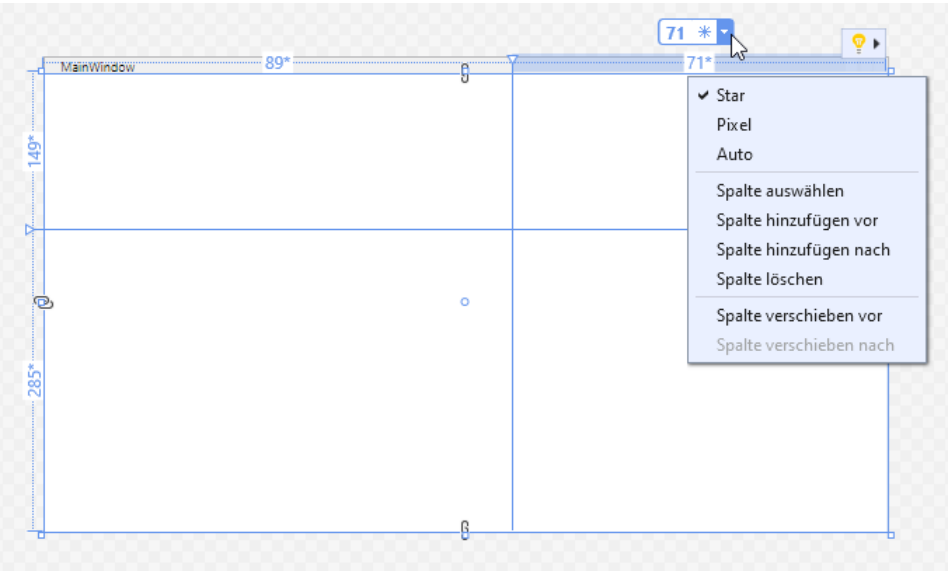
Analog lässt sich durch direktes Editieren des XAML-Codes oder durch Mausklicks auf die am oberen Rand des **Grid**-Elements befindliche Spaltendefinitionszone



die Spaltenstruktur vereinbaren, z. B. mit dem folgenden XAML-Ergebnis:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="89*"/>
        <ColumnDefinition Width="71*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="149*"/>
        <RowDefinition Height="285*"/>
    </Grid.RowDefinitions>
</Grid>
```

Wenn man den Mauszeiger links neben eine Zeile oder über eine Spalte positioniert, dann erscheint ein Werkzeug zur Modifikation von Größe und Anordnung, z. B.:



Das Symbol neben der numerischen Größenangabe steht für das (z. B. per Drop-Down – Menü einstellbare) Verfahren zur Größenbestimmung:

- Angabe einer festen Anzahl von geräteunabhängigen Pixeln (1/96 Zoll pro Einheit), z. B.: 355
- Angabe eines Anteils an der (vertikalen bzw. horizontalen) Ausdehnung des **Grid**-Containers, z. B.: 71 ✳
- Automatische Größenberechnung aufgrund des Inhalts, z. B.: auto

Eine numerische, als Pixelzahl oder Anteil zu interpretierende Größenangabe lässt sich nach einem Mausklick ändern.

Die Anteilswerte aller Spalten bzw. Zeilen addieren sich nicht auf 100. Im Beispiel beansprucht die rechte Spalte den Anteil

$$\frac{71}{89+71}$$

von der verfügbaren Fensterbreite.

Per Drop-Down - Menü kann man z. B. das Verfahren zur Größenbestimmung festlegen oder eine Spalte bzw. Zeile löschen.

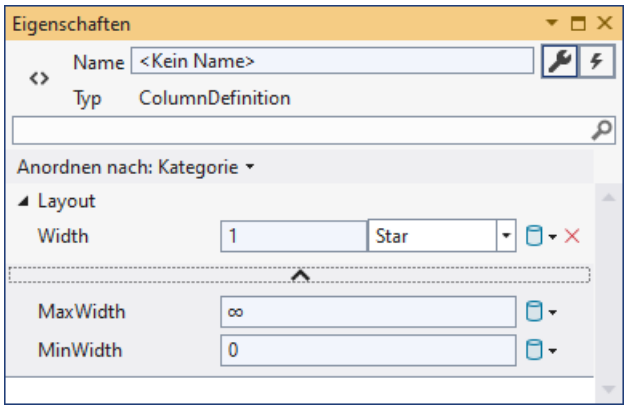
Anschließend geht es darum, wie der horizontal bzw. vertikal verfügbare Platz per XAML-Code auf mehrere Spalten bzw. Zeilen eines **Grid**-Layoutcontainers verteilt wird.

12.6.1.2 Platzaufteilung

Die gewünschte Breite einer Spalte bzw. die gewünschte Höhe einer Zeile beim Programmstart lässt sich per **Width**- bzw. **Height**-Attribut festlegen, wobei die folgenden Alternativen zur Verfügung stehen:

- Bei einer fehlenden Größenangabe
  ```
  <ColumnDefinition/>
  ```
  oder bei der folgenden Anforderung
  ```
  <ColumnDefinition Width="*" />
  ```
  beansprucht die Spalte bzw. Zeile den gesamten noch nicht vergebenen Platz. Verhalten sich alle Konkurrenten so, dann 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.
  Eine Spalte mit der Breite 0 ist unsichtbar:
  ```
  <ColumnDefinition Name="col1" Width="0"/>
  ```
  Für eine Spalte mit einem Feldnamen lässt sich die Breite im Programm mit Hilfe der Struktur **GridLength** und der Enumeration **GridUnitType** neu festlegen, z. B.:
  ```
  col1.Width = new GridLength(1, GridUnitType.Star);
  ```
- Besitzen Spalten bzw. Zeilen eine feste (nicht als Anteil zu verstehende) Ausdehnung in geräteunabhängigen Pixeln (1/96 Zoll pro Einheit), dann werden die angeforderten Pixel ausgegeben, solange der Vorrat reicht. Eine nachgeordnete Spalte bzw. Zeile wird eventuell eingeschränkt oder (bei erschöpftem Vorrat) überhaupt nicht angezeigt. Per Stern-Syntax um den freien Platz konkurrierende Spalten bzw. Zeilen gehen bei erschöpftem Pixelvorrat leer aus (unabhängig von ihrer Position).
- Vergibt man den Wert **Auto**, dann orientiert sich die Breite einer Spalte bzw. die Höhe einer Zeile am maximalen Platzbedarf der enthaltenen Elemente.

Statt den **Width**- bzw. **Height**-Wert direkt im XAML-Code einzutragen, kann man bei passender Markierung auch das **Eigenschaften**-Fenster benutzen, z. B.



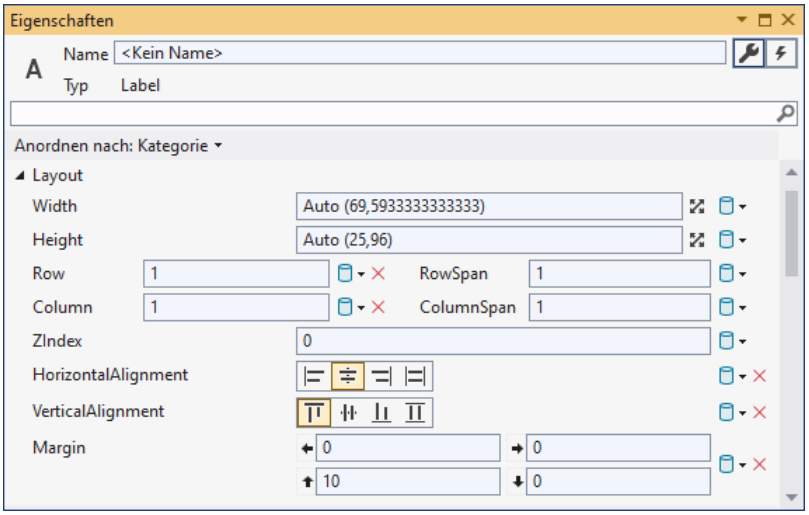
Zur Verwaltung des Platzbedarfs einer Spalte bzw. Zeile können zusätzlich die Eigenschaften **MinWidth**, **MaxWidth, MinHeight** und **MaxHeight** verwendet werden (vgl. Abschnitt 12.7.3.2).

12.6.1.3 Ortsangaben

Besitzt ein per **Grid**-Container verwaltetes Steuerelement keine Ortsangabe, dann landet es in der **Grid**-Zelle (0, 0), also links oben. Um für ein Steuerelement eine alternative Zelle festzulegen, sind im zugehörigen XAML-Element die Attribute bzw. die angefügten Eigenschaften **Grid.Row** bzw. **Grid.Column** mit der null-basierten Nummer der Zeile bzw. Spalte zu versorgen (siehe Abschnitt 12.5.2 zu den angefügten Eigenschaften). Hier wird ein **Label**-Objekt in die Zelle (1, 1) gesetzt:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Label Content="Bewertung:" HorizontalAlignment="Center" VerticalAlignment="Top"
           Margin="0,10,0,0" Grid.Column="1" Grid.Row="1" />
    . . .
</Grid>
```

Natürlich kann man auch das Eigenschaftenfenster zur Platzanweisung benutzen:

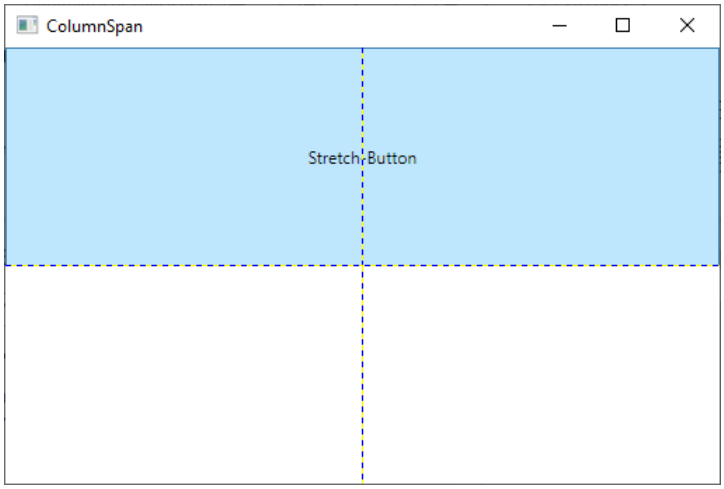


12.6.1.4 Mehrzellige Elemente

Es ist nicht vorgeschrieben, dass sich ein Steuerelement vollständig innerhalb *einer* **Grid**-Zelle befinden muss. Mit Hilfe der angefügten Eigenschaften **Grid.RowSpan** sowie **Grid.ColumnSpan** kann ein Steuerelement *mehrere* Zeilen und/oder Spalten belegen. 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="ColumnSpan.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 das **Grid**-Attribut **ShowGridLines** wird dafür gesorgt, dass die (2 × 2) - Struktur des Containers optisch präsent ist:[1]



Ein Visual Studio - Projekt mit dem Stretch-Button - Beispiel ist im folgenden Ordner zu finden:

…\BspUeb\WPF\Layoutcontainer\Grid mit ColumnSpan

Dass sich mehrere Steuerelemente eine **Grid**-Zelle teilen können, haben Sie schon wiederholt beobachtet (z. B. im Abschnitt 12.4.3). Bei einer vom Visual Studio neu angelegten WPF-Anwendung verwendet das Hauptfenster zunächst einen einzelligen **Grid**-Container, und alle aus der Toolbox per Drag & Drop übernommenen Steuerelemente landen in der einzigen Zelle mit den Koordinaten (0, 0). Sofern die Größen und Verankerungen der Elemente geschickt gewählt werden, resultiert ein sinnvoll nutzbares Fenster. Für ein komplexes Layout in einem Fenster mit variabler Größe und variablem Seitenverhältnis ist ein einzelliger **Grid**-Container aber keine gute Lösung.

---

[1] Bei den Werten **true** und **false** für XAML-Attribute zu Eigenschaften vom Typ **bool** ist die Groß-/Kleinschreibung ausnahmsweise irrelevant.

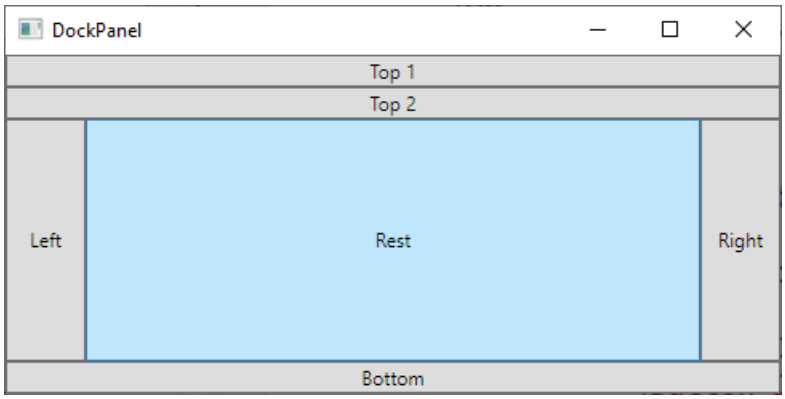
12.6.2 DockPanel

Ein Layoutcontainer der 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 und diese komplett zu belegen. Wollen mehrere Elemente eine Seite besetzen, dann werden sie dort in der Beitrittsreihenfolge gestapelt. In der zur Andock-Seitenlinie orthogonalen Richtung erhalten die Steuerelemente nach Möglichkeit ihre gewünschte Ausdehnung. Wird zuletzt ein Element *ohne* **Dock**-Angabe eingefügt, dann belegt es den gesamten restlichen Platz.[1]

In 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 1</Button>
        <Button DockPanel.Dock="Top">Top 2</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 das zuletzt, *ohne* **Dock**-Angabe eingefügte **Button**-Objekt belegt den frei gebliebenen Raum im Zentrum:



Der **DockPanel**-Layoutcontainer 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.

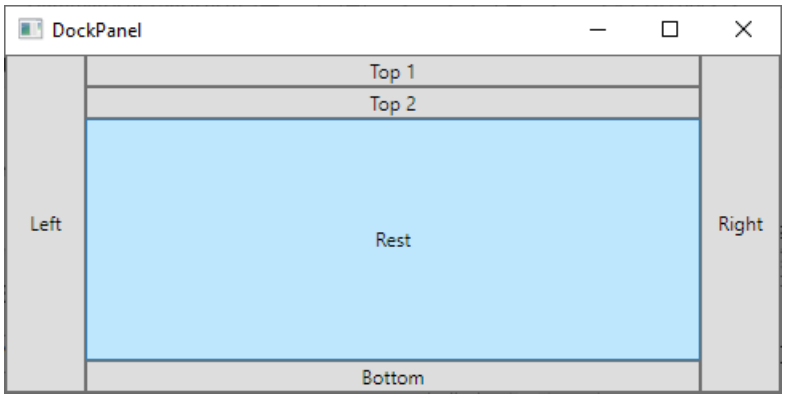
Ob die Ecken von den an einer horizontalen Seite (oben bzw. unten) oder von den an einer vertikalen Seite (links bzw. rechts) angedockten Steuerelementen eingenommen werden, hängt von der Aufnahmereihenfolge ab. Aus der Layoutdefinition

---

[1] Wer sich an das **BorderLayout** im traditionsreichen GUI-Framework **Swing** der Programmiersprache Java erinnert fühlt, liegt genau richtig.

```
<DockPanel>
   <Button DockPanel.Dock="Left" Width="50">Left</Button>
   <Button DockPanel.Dock="Right" Width="50">Right</Button>
   <Button DockPanel.Dock="Top">Top 1</Button>
   <Button DockPanel.Dock="Top">Top 2</Button>
   <Button DockPanel.Dock="Bottom">Bottom</Button>
   <Button>Rest</Button>
</DockPanel>
```

folgt diese Anordnung:



Ein Visual Studio - Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

…\BspUeb\WPF\Layoutcontainer\DockPanel

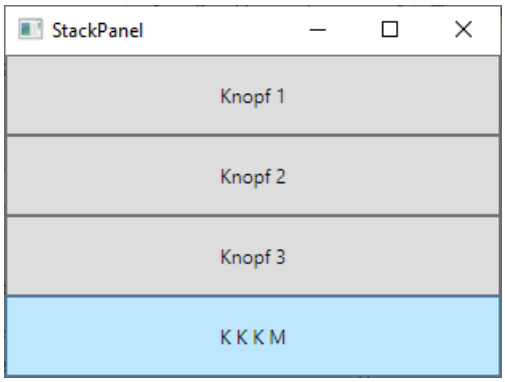
12.6.3 StackPanel

Mit einem Layoutcontainer der Klasse **StackPanel** realisiert man einen vertikalen oder horizontalen Stapel von Steuerelementen. Die Elemente erhalten die gewünschten Ausdehnungen, solange der Platz ausreicht. Ist bei einem vertikalen Stapel die Gesamthöhe bzw. bei einem horizontalen Stapel die Gesamtbreite unzureichend, dann 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 übereinandergestapelt:



Weil die **Width**-Eigenschaften der Schalter keinen Wert erhalten, gilt die Voreinstellung **Stretch** (Ausdehnung über die komplette Breite, vgl. Abschnitt 12.7.3.2).

Im **Window**-Element sorgt die Eigenschafts- bzw. Attributzuweisung

```
SizeToContent="Height"
```

dafür, dass die Fensterhöhe an den vom **StackPanel** benötigten Platz angepasst wird.

Um einen *horizontalen* Stapel zu erhalten, setzt man die **StackPanel**-Eigenschaft **Orientation** auf den Wert **Horizontal**:

```
<StackPanel Orientation="Horizontal">
  . . .
</StackPanel>
```

Ein Visual Studio - Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

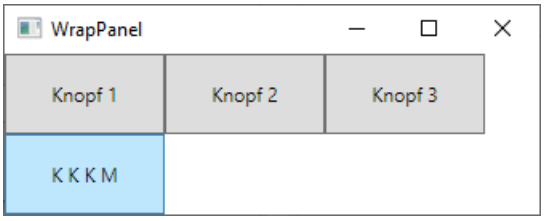
…\BspUeb\WPF\Layoutcontainer\StackPanel

12.6.4 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:



Ein Visual Studio - Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

…\BspUeb\WPF\Layoutcontainer\WrapPanel

12.6.5 UniformGrid

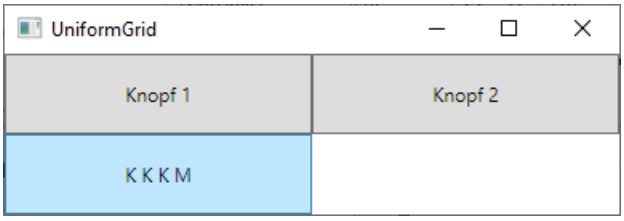
Von der sehr flexiblen und sehr oft verwendeten Klasse **Grid** unterscheidet sich die selten und nur für spezielle Zwecke (z. B. für eine virtuelle numerische Tastatur) genutzte Klasse **UniformGrid** trotz der Namensähnlichkeit erheblich:

- Die Zahl der Zeilen bzw. Spalten kann über die Attribute bzw. Eigenschaften **Rows** bzw. **Columns** festgelegt werden, z. B.:
```
<UniformGrid Rows="1" Columns="3">
    . . .
</UniformGrid>
```
- Verzichtet man auf die Definition der Matrixstruktur, dann entsteht eine *quadratische* Matrix mit einer automatisch ermittelten Anzahl von Zeilen bzw. Spalten. Ist die Anzahl der eingefügten Elemente z. B. größer als 4 und kleiner als 10, dann resultiert eine (3 × 3) - Matrix.
- Beim Einfügen von Elementen findet keine Platzanweisung statt. Stattdessen werden die Elemente sukzessiv auf die Matrixzellen verteilt, wobei jede Zelle genau *ein* Element aufnimmt.
- Alle Zellen (ggf. auch die unbesetzten) sind gleich groß.

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, also zunächst die erste Zeile gefüllt wird:



Ein Visual Studio - Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

…\BspUeb\WPF\Layoutcontainer\UniformGrid

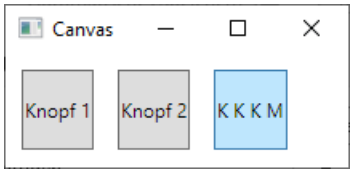
12.6.6 Canvas

Die Klasse **Canvas** stellt eine Zeichenfläche ohne jede Layout-Logik bei Größen- oder Seitenverhältnisänderungen zur Verfügung. Durch die angefügten Eigenschaften **Canvas.Left** und **Canvas.Top** legt man für die verwalteten Elemente den Abstand zum linken bzw. zum oberen Rand fest. Wer unbedingt will, kann mit diesem Layoutcontainer Steuerelemente mit fester Größe und Position über eine sorgfältige Berechnung von Koordinaten 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:



Ein Visual Studio - Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

…\BspUeb\WPF\Layoutcontainer\Canvas

12.6.7 Geschachtelte Layoutcontainer

Das folgende, aus dem Abschnitt 12.5.3 bekannte Layout besitzt ein **TextBox**-Element und drei **Label**-Elemente mit einem vertikal zentrierten und gemeinsam ausgerichteten Auftritt:

Würde man alle Steuerelemente in einen einzelligen **Grid**-Layoutcontainer stecken, dann wäre die gemeinsame vertikale Ausrichtung mit Positionierungsaufwand verbunden. Die folgende Konstruktion verwendet zwar drei Layoutcontainer ist aber einfacher herzustellen und sorgt für eine gute Bedienbarkeit bei unterschiedlichen Fenstergrößen:

- Als Wurzel-Layoutcontainer wird **Grid**-Objekt mit zwei Spalten und einer Zeile verwendet.
- In jede **Grid**-Zelle wird ein horizontal orientierter **StackPanel**-Layoutcontainer aufgenommen, der in seiner Zelle vertikal zentriert ist.
- Jeder **StackPanel**-Container nimmt zwei Elemente auf.

Mit dem Wert **True** für das **Grid**-Attribut **ShowGridLines** wird dafür gesorgt, dass die Struktur des Containers optisch präsent ist:[1]

[1] Bei den Werten **true** und **false** für XAML-Attribute zu Eigenschaften vom Typ **bool** ist die Groß-/Kleinschreibung ausnahmsweise irrelevant.

```
<Grid ShowGridLines="True">
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <StackPanel Orientation="Horizontal" VerticalAlignment="Center">
        <Label Content="DM:" />
        <TextBox Name="tbEingabe" VerticalContentAlignment="Center" MinWidth="120"
                 TextAlignment="Right" />
    </StackPanel>
    <StackPanel Orientation="Horizontal" Grid.Column="1" VerticalAlignment="Center">
        <Label Content="Euro:"/>
        <Label BorderBrush="Black" BorderThickness="1" MinWidth="120"
               HorizontalContentAlignment="Right" />
    </StackPanel>
</Grid>
```

Ein Visual Studio - Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

…\BspUeb\WPF\Layoutcontainer\Geschachtelte Layoutcontainer

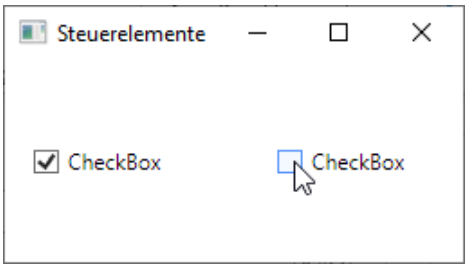
12.7 Basiswissen über Steuerelemente

Durch zahlreiche Klassen zur Realisation von Steuerelementen (Schaltflächen, Textfeldern, Kontrollkästchen, Listen etc.) ermöglicht die WPF-Bibliothek für praktisch jedes Programm eine gut bedienbare und attraktive Bedienoberfläche.

12.7.1 Steuerelemente im Vergleich zu anderen Objekten

Als Besonderheiten der Steuerelementklassen (z. B. im Vergleich zu Klassen wie **String** oder **List<T>**) sind zu nennen:

- Ihre Objekte können **auf dem Bildschirm auftreten** und dabei selbständig **mit dem Benutzer interagieren**. Wenn wir z. B. ein Kontrollkästchen (ein Objekt der Klasse **CheckBox**) in ein Fenster einbauen, dann erscheint bzw. verschwindet bei einem Mausklick die Markierung,

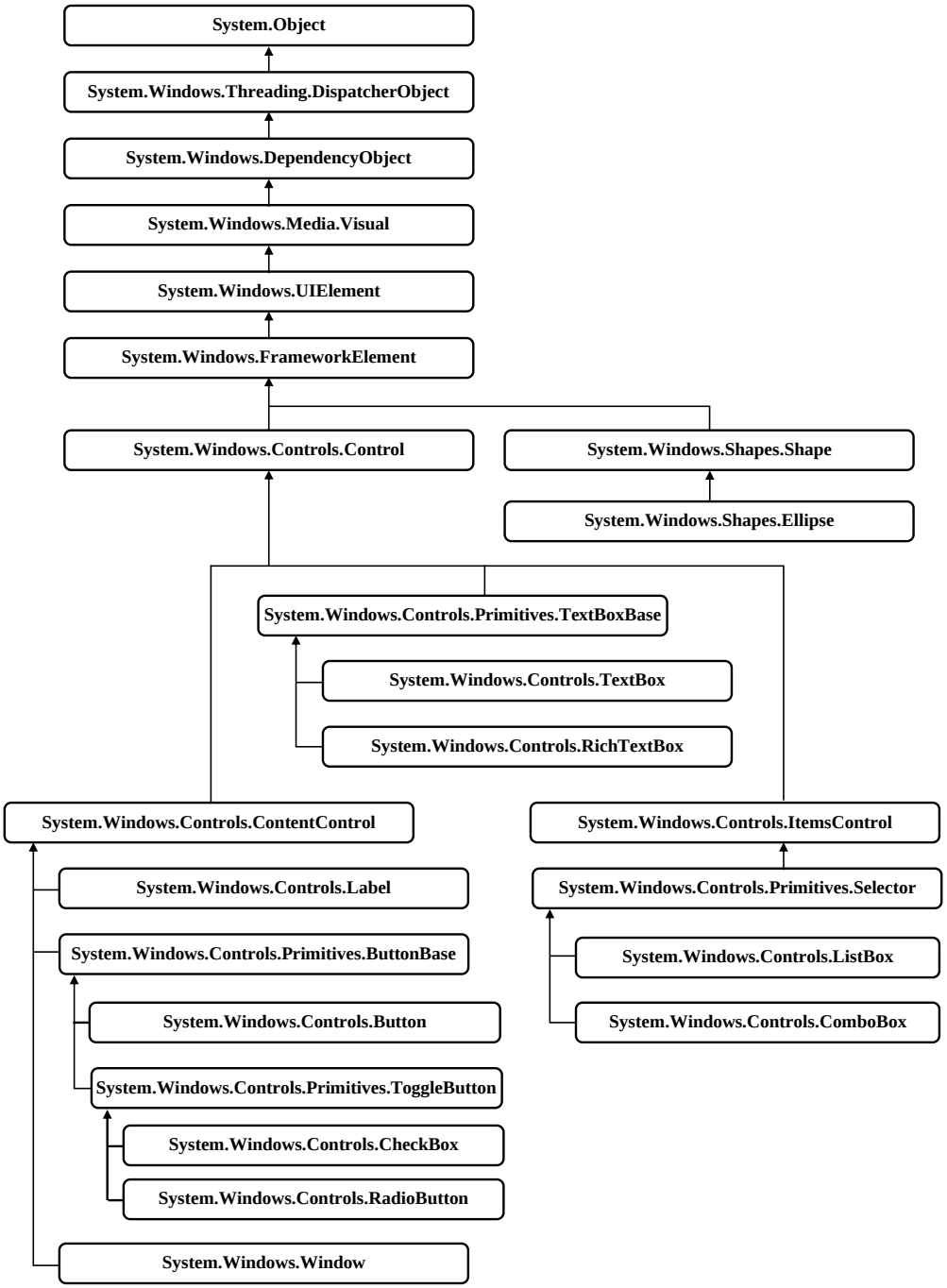


  ohne dass wir uns um diese Anpassung der Optik kümmern müssen. Hinter dieser Verhaltenskompetenz steckt eine aufwändige, von den WPF-Entwicklern geleistete Grafikprogrammierung.
- Die Steuerelemente kommunizieren über **Ereignisse** mit anderen Klassen. Will man z. B. über die gesetzte Markierung eines Kontrollkästchen informiert werden, dann registriert man eine Behandlungsmethode, die den Delegatentyp **RoutedEventHandler** (Namensraum **System.Windows**) erfüllt, bei seinem Ereignis **Checked**. Bei den Ereignissen von Steuerelementen handelt es sich meist um Routingereignisse (siehe Abschnitt 12.4). Es treten aber auch CLR-Ereignisse auf.

- Die Eigenschaften von Steuerelementen (z. B. **Text**, **Height**, **HorizontalAlignment**, **Focusable**) können zur Entwurfszeit über **Werkzeuge der Entwicklungsumgebung** konfiguriert werden (siehe z. B. Abschnitt 5.13.5). Wie im Abschnitt 12.5 zu erfahren war, handelt es sich bei den Eigenschaften von Steuerelementen in den meisten Fällen um Abhängigkeitseigenschaften.

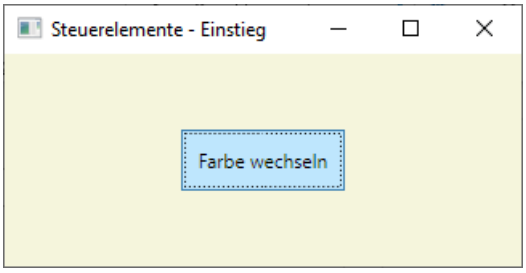
12.7.2 Abstammungsverhältnisse

In der WPF-Bibliothek 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** abgeleitet sind. Der folgende Stammbaum beschränkt sich auf die im Manuskript diskutierten Klassen:



12.7.3 Verwendung

Wir beschäftigen uns zunächst anhand eines sehr einfachen Beispielprogramms



damit, ...

- wie ein per Drag & Drap übernommenes **Toolbox**-Element zum Member-Objekt der Anwendungsfensterklasse wird,
- wie man seine Eigenschaften zur Entwurfszeit bestimmt,
- wie man Behandlungsmethoden bei seinen Ereignissen registriert.

12.7.3.1 Instanziieren

Wird mit dem WPF-Designer im Visual Studio ein Steuerelement aus der **Toolbox** auf ein Fenster gezogen, dann landet es in einem Layoutcontainer. Bei einer neuen WPF-Anwendung ist für das Hauptfenster per Voreinstellung ein **Grid**-Layoutcontainer zuständig, der sich vorläufig auf die Zelle (0, 0) beschränkt, also nur *eine* Zeile bzw. Spalte besitzt. Im XAML-Code zur Hauptfensterklasse erscheint im **Grid**-Element ein Eintrag für das neue Steuerelement, z. B.:

```
<Grid>
    <Button Content="Button" HorizontalAlignment="Left"
     Margin="345,169,0,0" VerticalAlignment="Top"/>
</Grid>
```

Dem XAML-Code ist zu entnehmen, dass ...

- auf dem Hauptfenster ein Steuerelement der Klasse **Button** erscheint,
- der Schalter die Beschriftung `Button` besitzt,
- der Schalter am linken Rand der **Grid**-Zelle (0, 0), also am linken Rand des Fensters, verankert ist mit dem Abstand 345,
- der Schalter am oberen Rand des Fensters verankert ist mit dem Abstand 169.

Wir haben im Abschnitt 12.3.4 erfahren, dass ...

- bei der Programmerstellung aus der XAML-Datei zur Hauptfensterklasse eine Binär-Variante mit der Namenserweiterung **baml** entsteht,
- beim Programmstart die **baml**-Datei geladen wird, wobei die dort beschriebenen Objekte entstehen. Im Beispiel resultiert für das Hauptfenster ein Member-Objekt vom Typ **Button**.

12.7.3.2 Elementare Eigenschaften

Durch die folgenden Eigenschaften eines Steuerelements werden seine Ausdehnung sowie seine Position im übergeordneten Element (Layoutcontainer(zelle) oder **ItemsControl**-Steuerelement) geregelt. Sie sind in der Klasse **System.Windows.FrameworkElement** definiert, also auch bei Layoutcontainern und Grafikelementen anwendbar. In der Regel handelt es sich um Abhängigkeitseigenschaften (siehe Abschnitt 12.5):

- **Width**, **Height**
  Mit den Eigenschaften **Width** und **Height** vom Typ **double** wählt man für die Breite bzw. Höhe eine Anzahl von geräteunabhängigen Pixeln (1/96 Zoll pro Einheit). Durch den voreingestellten Wert **NaN** werden die Breite bzw. Höhe passend zum Inhalt und zum angeforderten Innenrand (siehe unten) festgelegt.
- **MinWidth**, **MaxWidth, MinHeight, MaxHeight**
  Mit den Eigenschaften **MinWidth**, **MaxWidth, MinHeight** und **MaxHeight** vom Typ **double** wählt man für die Breite bzw. Höhe einen minimalen bzw. maximalen Wert. Im folgenden Beispiel resultiert ein Fenster mit der fixierten Höhe 200 und der initialen Breite 400, die vom Benutzer zwischen 300 und 500 verändert werden kann:[1]

  ```
  <Window . . .
   Height="200" MinHeight="200" MaxHeight="200"
   Width="400" MinWidth="300" MaxWidth="500" >
  . . .
  </Window>
  ```

  Bei Konflikten zwischen **MinWidth**, **MaxWidth** und **Width** wird so verfahren:
  - **MinWidth** dominiert **MaxWidth**
  - **MaxWidth** dominiert **Width**

  Eine analoge Aussage gilt für **MinHeight**, **MaxHeight** und **Height**.
- **Margin**
  Mit dieser Eigenschaft vom Typ **System.Windows.Thickness** legt man fest, wie viel Platz um das Element herum bis zu den Ankerseiten des übergeordneten Elements (Layoutcontainer(zelle) oder **ItemsControl**-Steuerelement) oder bis zu den benachbarten Elementen freibleiben soll.

  Das übergeordnete Element entscheidet darüber, ob sich die **Margin**-Eigenschaft …
  - auf die Abstände zu den Ankerseiten auswirkt (z. B. beim **Grid**-Container)
  - oder auf die Abstände zu den benachbarten Elementen (z .B. beim **StackPanel**-Container).

  Die Ankerseiten hängen von den anschließend beschriebenen Eigenschaften **HorizontalAlignment** und **VerticalAlignment** ab.

  Man gibt für die Randbreite per **double**-Wert eine Anzahl von geräteunabhängigen Pixeln an, wobei unterschiedlich detaillierte Angaben möglich sind:
  - Mit *einer* Zahl legt man für alle Seiten denselben Rand fest, z. B.:
    ```
    Margin="50"
    ```
  - Von *zwei* Zahlen legt die erste Zahl die beiden horizontalen und die zweite Zahl die beiden vertikalen Ränder fest, z. B.:
    ```
    Margin="50,100"
    ```
  - *Vier* Zahlen beziehen sich (in dieser Reihenfolge) auf den linken, oberen, rechten und den unteren Rand, z. B.:
    ```
    Margin="20,30,40,50"
    ```
- **HorizontalAlignment**
  Diese Eigenschaft entscheidet darüber, ob das Steuerelement am linken und/oder rechten Rand des übergeordneten Elements unter Beachtung des gewünschte Randabstands (Eigenschaft **Margin**) verankert wird. Es sind vier Werte (aus der Enumeration **HorizontalAlignment** im Namensraum **System.Windows**) möglich:

---

[1] Die Klasse **Window** stammt von der Klasse **Control** ab (siehe Abschnitt 12.7.2) und besitzt daher die aktuell beschriebenen Eigenschaften von Steuerelementen.

  - **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, sodass sich seine horizontale Ausdehnung zusammen mit dem übergeordneten Element ändert.
  - **Center**
    Das Element wird horizontal zentriert. Ein per **Margin**-Eigenschaft eingestellter Randabstand wirkt sich auf die Zentrierung aus, sodass z. B. ein Unterschied zwischen dem linken und dem rechten Rand eine Verschiebung aus der Zentrallage bewirkt.

  Per Voreinstellung besitzt die Eigenschaft **HorizontalAlignment** den Wert **Stretch**, wobei aber ein expliziter **Width**-Wert dominiert.

- **VerticalAlignment**
  Diese Eigenschaft entscheidet darüber, ob das Steuerelement am oberen und/oder unteren Rand des übergeordneten Elements unter Beachtung des gewünschte Randabstands (Eigenschaft **Margin**) verankert wird. 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, sodass sich seine vertikale Ausdehnung zusammen mit dem übergeordneten Element ändert.
  - **Center**
    Das Element wird vertikal zentriert. Ein per **Margin**-Eigenschaft eingestellter Randabstand wirkt sich auf die Zentrierung aus, sodass z. B. ein Unterschied zwischen dem oberen und dem unteren Rand eine Verschiebung aus der Zentrallage bewirkt.

  Per Voreinstellung besitzt die Eigenschaft **VerticalAlignment** den Wert **Stretch**, wobei aber ein expliziter **Height**-Wert dominiert.

Welche Werte für die Eigenschaften **HorizontalAlignment** und **VerticalAlignment** realisierbar sind, hängt vom Typ des Layoutcontainers ab, z. B. ...

- haben die Elemente in einem **Grid**-Container per Voreinstellung die horizontale Ausrichtung **Stretch** und können alternative horizontale Ausrichtungen erhalten,
- sind die Elemente in einem horizontal orientierten **StackPanel**-Container grundsätzlich von links nach rechts angeordnet, d. h. das **HorizontalAlignment**-Attribut der Elemente ist wirkungslos.

Bei der vertikalen Ausrichtung haben die Elemente der genannten Layoutcontainer die freie Wahl.

Es folgen einige Eigenschaften, die den *Inhalt* von Steuerelementen betreffen:

- **Content**
  Über die in **System.Windows.Controls.ContentControl** definierte Eigenschaft **Content** vom Typ **Object** wird 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 vom Typ **System.Windows.Thickness** legt man eine freizuhaltende Randzone innerhalb der Steuerelementfläche (einen Innenrand) fest. Die Angaben sind analog zur Eigenschaft **Margin** zu machen (siehe oben).
- **HorizontalContentAlignment**, **VerticalContentAlignment**
  Mit diesen Eigenschaften bestimmt man die Ausrichtung des Inhalts innerhalb der rechteckigen Steuerelementfläche. Es sind dieselben Werte möglich wie bei **HorizontalAlignment** bzw. **VerticalAlignment** (siehe oben).

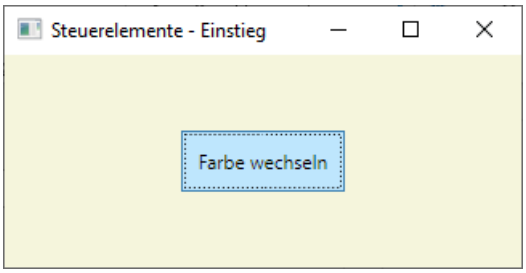
12.7.3.3 Ereignisbehandlung

Zur Behandlung der von einem WPF-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 (siehe Abschnitt 12.7.3.3.1).
- Um ein Ereignis auszulösen, rufen WPF-Klassen bis auf wenige Ausnahmen jeweils eine Methode mit dem Namenspräfix **On** auf, die aufgrund der Modifikatoren **virtual** und **protected** in abgeleiteten Klassen überschrieben werden darf. Eine abgeleitete Klasse kann in der überschreibenden Methode das Ereignis behandeln und/oder die Behandlungslogik ändern (siehe Abschnitt 12.7.3.3.2).

12.7.3.3.1 Behandlungsmethoden registrieren

Bei den Ereignissen eines Fensters oder eines Steuerelements lassen sich Behandlungsmethoden registrieren, die den geforderten Delegatentyp erfüllen. In der Regel (aber nicht notwendigerweise) werden die Behandlungsmethoden in der Code-Behind - Datei der betroffenen Fensterklasse definiert, sodass die erforderlichen Referenzen und Zugriffsrechte vorhanden sind. Im aktuellen Beispiel



soll das **Click**-Ereignis des **Button**-Objekts behandelt werden. Dazu fordern wir im WPF-Designer bei markiertem **Button**-Objekt im **Eigenschaften**-Fenster per Mausklick auf das Symbol  die Liste mit den Ereignissen an und setzen einen Doppelklick auf das Texteingabefeld **Click**. Weil das **Click**-Ereignis bei einem Befehlsschalter das Standardereignis ist, führt ein Doppelklick auf das **Button**-Steuerelement 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 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 bei der eben beschriebenen Vorgehensweise der WPF-Designer erledigt, z. B.:

```
<Button Content="Farbe wechseln"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Padding="10" Click="button_Click" />
```

Wie Sie aus dem Abschnitt 12.3.4 wissen, ist die automatisch erstellte partielle Fensterklassendefinition in der Datei **MainWindow.g.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;
}
```

Wie der XAML-Code zum Beispiel zeigt, war es nicht erforderlich, für das **Button**-Objekt einen Namen festzulegen.

12.7.3.3.2 On-Methoden überschreiben

Wird eine WPF-Klasse beerbt, dann besteht eine Alternative zu der im Abschnitt 12.7.3.3.1 beschriebenen Technik der Ereignisbehandlung. Um ein Ereignis auszulösen, rufen WPF-Klassen bis auf wenige Ausnahmen jeweils eine Methode mit dem Namenspräfix **On** auf, z. B.

- **OnMouseEnter()**
- **OnMouseDown()**

Weil die **On**-Methoden als **protected** sowie **virtual** definiert sind, kann man sie in abgeleiteten Klassen überschreiben, um …

- die gewünschte Ereignisbehandlung direkt in der **On**-Methode vorzunehmen
- und/oder die Behandlungslogik zu ändern.

Das Überschreiben der zugehörigen **On**-Methode ist bei den Ereignissen von Steuerelementen (z. B. **Button**) nicht unbedingt die bevorzugte Technik, weil dazu eine abgeleitete Klasse definiert werden muss. 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 zunächst 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>
```

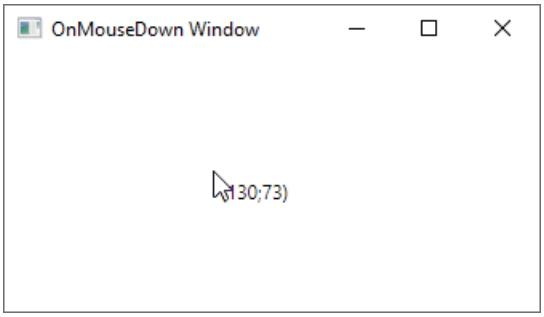
Die in der Code-Behind - Datei

```
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() + ")";
      }
   }
}
```

implementierte Methode `Window_MouseDown()` ...

- legt die Position eines **Label**-Steuerelements über dessen **Margin**-Eigenschaft (siehe Abschnitt 12.7.3.2) neu auf die Klickstelle fest, die über die Methode **GetPosition()** des Parameterobjekts vom Typ **MouseButtonEventArgs** zu ermitteln ist,
- und verarbeitet die Koordinaten der Klickstelle zum neuen Wert der **Content**-Eigenschaft des **Label**-Objekts.

Somit kann das Programm die Koordinaten einer Klickstelle am Ort des Geschehens anzeigen, z. B.:



Eine alternative Technik zur **MouseDown**-Behandlung besteht darin, in der abgeleiteten Klasse die **Window**-Methode **OnMouseDown()** zu überschrieben und in der Überschreibung die Ereignisbehandlung vorzunehmen, z. B.:

```
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() + ")";
}
```

Dabei sollte in der Regel in der ersten Anweisung die überschriebene Basisklassenmethode aufgerufen werden. Diesen Aufruf zu unterlassen, hat bei vielen Ereignissen zur Folge, dass

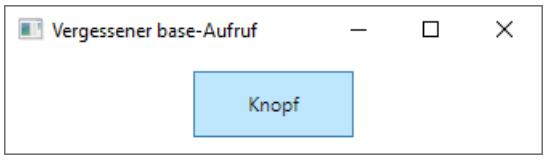
registrierte Behandlungsmethoden abgehängt werden, weil das Ereignis in der **On**-Methode der Basisklasse ausgelöst wird. Das trifft z. B. für die Methode **OnClick()** der Klasse **ButtonBase** zu:[1]

```
protected virtual void OnClick() {
   RoutedEventArgs newEvent = new RoutedEventArgs(ButtonBase.ClickEvent, this);
   RaiseEvent(newEvent);
   MS.Internal.Commands.CommandHelpers.ExecuteCommandSource(this);
}
```

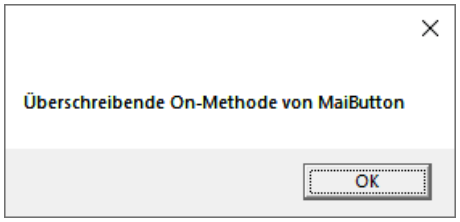
Bei einer Überschreibung ohne Basisklassenmethodenaufruf, werden beim Ereignis registrierte Methoden abgehängt, z. B.:

```
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("Überschreibende On-Methode von MaiButton");
    }
}

class ForgottenBaseCall : Window {
    ForgottenBaseCall() {
        Height = 100; Width = 300;
        Title = "Vergessener base-Aufruf";
        MaiButton knopf = new();
        knopf.Content = "Knopf";
        knopf.Width = 100;
        knopf.Margin = new Thickness(10);
        knopf.Click += new RoutedEventHandler(knopf_Click);
        AddChild(knopf);
    }

    private void knopf_Click(object sender, RoutedEventArgs e) {
        MessageBox.Show("Click-Handler von Knopf");
    }

    [STAThread]
    static void Main() {
        new Application().Run(new ForgottenBaseCall());
    }
}
```

Aufgrund des Programmierfehlers wird bei einem Mausklick auf den Schalter



nur noch die **OnClick()** - Überschreibung ausgeführt:

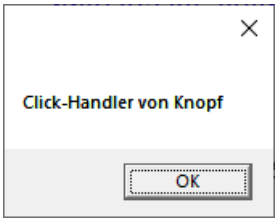


[1] Wie man den BCL-Quellcode einsehen kann, erläutert der Abschnitt 2.6.4.

Mit der korrekten Überschreibung

```
protected override void OnClick() {
   base.OnClick();
   MessageBox.Show("Überschreibende On-Methode von MaiButton");
}
```

bleiben andere Ereignisbehandlungsmethoden ggf. im Spiel, z. B.:



Bei der **Window**-Methode **OnMouseDown()** hat die Unterlassung des Basismethodenaufrufs keine erkennbaren Konsequenzen. Halten Sie sich trotzdem grundsätzlich daran, zu Beginn einer überschreibenden **On**-Methode die Basisklassenvariante aufzurufen.

Wird eine Ereignisbehandlung durch das Überschreiben der zuständigen **On**-Methode erledigt, dann ist keine Registrierung (z. B. im XAML-Code) erforderlich, weil die im Abschnitt 12.7.3.3.1 beschriebene Delegatentechnik der Ereignisbehandlung *nicht* zum Einsatz kommt.

12.7.4 Standardkomponenten

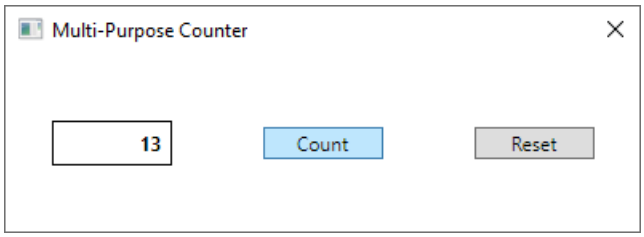
Dieser Abschnitt behandelt häufig benötigte Steuerelemente: Befehlsschalter, Kontrollkästchen, Optionsfelder, Texteingabefelder, Listen- und Kombinationsfelder.

12.7.4.1 Schaltflächen

Schaltflächen werden durch die Klasse **Button** realisiert.

12.7.4.1.1 Beispiel

Im folgenden *Multi-Purpose Counter*, der z. B. zur Verkehrszählung taugt, kommen zwei **Button**-Objekte zum Einsatz:



Das GUI-Design und die Ereignismethodenregistrierung 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 im Wurzelelement das **XAML**-Attribut bzw. die **Window**-Eigenschaft **ResizeMode** den Wert **NoResize**. Dadurch verschwindet erwartungsgemäß auch der Titelzeilenschalter zum Maximieren des Fensters. Allerdings geht auch der Titelzeilenschalter zum Minimieren des Fensters verloren, sodass man das Programm nicht mehr in die Taskleiste beordern kann. Um die Fenstergröße zu fixieren, ohne den Minimieren-Schalter zu verlieren, kann man die **Window**-Eigenschaften bzw. -Attribute **MinWidth**, **MaxWidth, MinHeight** und **MaxHeight** verwenden (siehe Abschnitt 12.7.3.2).

Die **Height**-Eigenschaften der Steuerelemente werden unter Berücksichtigung von Schriftgröße und Innenrand automatisch bestimmt.

Für das **Label**-Steuerelement wird die Schriftauszeichnung **fett** eingestellt durch den Wert **Bold** für das **XAML**-Attribut **FontWeight**:

```
FontWeight="Bold"
```

Außerdem erhält das Label einen Rahmen in schwarzer Farbe

```
BorderThickness="1" BorderBrush="Black"
```

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;

    public MainWindow() {
        InitializeComponent();
    }

    private void Button_Click(object sender, RoutedEventArgs e) {
        if (e.Source == count)
            anzahl++;
        else
            anzahl = 0;
        label.Content = anzahl.ToString();
    }
}
```

Weil es um ein Ereignis mit der Routingstrategie *Bubbling* geht, kann die Behandlungsmethode dem gemeinsamen **Grid**-Container zugeordnet werden, statt sie bei *beiden* **Button**-Objekten zu registrieren (siehe Abschnitt 12.4):

```
<Grid Button.Click="Button_Click">
  . . .
</Grid>
```

Wir verwenden hier ein sogenanntes *angefügtes Ereignis* (siehe Abschnitt 12.4.3), und der beim Aufruf an `Button_Click()` übergebene **sender**-Parameter zeigt auf den **Grid**-Layoutmanager.

Ein Visual Studio - Projekt mit dem Beispielprogramm ist hier zu finden:

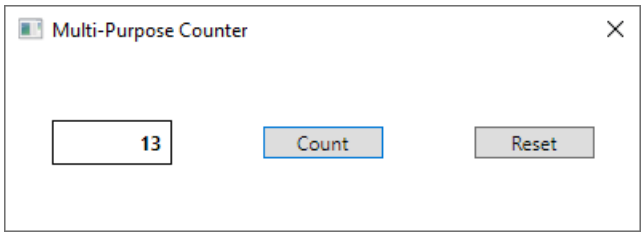
…\BspUeb\WPF\Steuerelemente\Button\Text

12.7.4.1.2 Standard- und Escape-Schalter

Damit im aktuellen Beispielprogramm die **Enter**-Taste das **Click**-Ereignis des Schalters zum Inkrementieren des Zählerstands auslöst, wird die **IsDefault**-Eigenschaft dieses Schalters auf den Wert **true** gesetzt:

```
<Button Name="count" Content="Count" Width="75" Grid.Column="1"
   HorizontalAlignment="Center" VerticalAlignment="Center" IsDefault="True" />
```

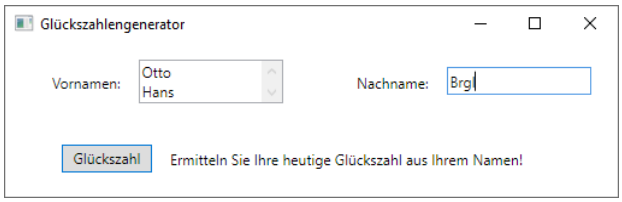
Der Schalter wird damit zum **Standardschalter** des Fensters, und sein Privileg kommt im laufenden Programm durch eine blaue Umrandung zum Ausdruck:



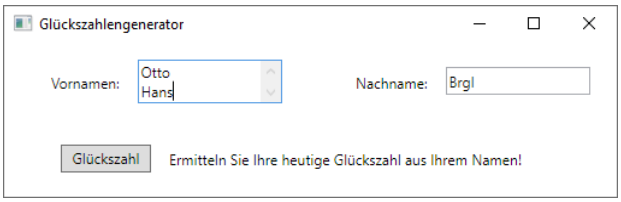
Ein Multi-Counter - Benutzer kann nach dem Start sofort die **Enter**-Taste zum Zählen verwenden, ohne zuvor den Eingabefokus (per Maus oder Tabulatortaste) auf den Count-Schalter setzen zu müssen (siehe Abschnitt 12.7.4.4).

Die Standardschaltfläche verliert das Privileg, per **Enter**-Taste auslösbar zu sein, wenn ein anderes, an der **Enter**-Taste interessiertes Steuerelement (z. B. ein anderer Schalter oder ein mehrzeiliges Texteingabefeld) den Tastatur-Eingabefokus besitzt (siehe Abschnitt 12.7.4.4). Im folgenden Beispielprogramm (vgl. Abschnitt 12.7.4.3) ist der Schalter genau dann per **Enter**-Taste auslösbar, wenn das mehrzeilige (linke) Textfeld zur Eingabe der Vornamen den Eingabefokus *nicht* besitzt:

Schalter ist per **Enter**-Taste ansprechbar:



Schalter ist *nicht* per **Enter**-Taste ansprechbar:



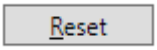
Analog zur Standardschaltflächen-Ernennung kann ein **Button**-Objekt über die Eigenschaft **IsCancel** als Escape-Schalter des Fensters festgelegt werden. Sein **Click**-Ereignis lässt sich dann auch per **Esc**-Taste auslösen.

12.7.4.1.3 Alt-Zugriffstaste

Um das **Click**-Ereignis eines Befehlsschalters auch über einen **Alt**-Tastenbefehl (über eine sogenannte *Zugriffstaste*) auslösen zu können, verwendet man ein **AccessText**-Element, z. B.:

```
<Button Name="reset" Width="75" Grid.Column="2" HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <AccessText>_Reset</AccessText>
</Button>
```

Man wählt die auslösende Alt-Zugriffstaste, indem man dem zugehörigen Buchstaben im **AccessText**-Inhalt einen Unterstrich voranstellt. Spätestens nach dem einmaligen Drücken der **Alt**-Taste ist der Buchstabe im laufenden Programm durch eine Unterstreichung hervorgehoben, z. B.:



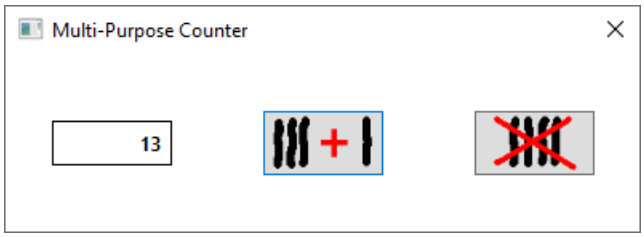
Im Beispiel wird das **AccessText**-Element als Wert der Inhaltseigenschaft des **Button**-Objekts interpretiert (vgl. Abschnitt 12.3.2.3.3). Folglich darf im **Button**-Element kein **Content**-Attribut vorhanden sein, weil sonst die Eigenschaft **Content** doppelt versorgt wäre.

Mit Hilfe eines **AccessText**-Elements lässt sich eine Zugriffstaste auch für eine Schaltfläche realisieren, die ein Bild anstatt oder zusätzlich zu einer Beschriftung auf der Oberfläche zeigt (siehe Abschnitt 12.7.4.1.4). Trägt ein Schalter (wie im aktuellen Beispiel) nur eine Beschriftung, dann lässt sich eine Zugriffstaste bequemer über das **Content**-Attribut zum **Button**-Element vereinbaren. Dazu setzt man im Wert des **Content**-Attributs einen Unterstich vor den Buchstaben, der in Kombination mit der **Alt**-Taste das **Click**-Ereignis des Schalters auslösen soll, z. B.:

```
<Button Name="reset" Content="_Reset" Width="75" Grid.Column="2"
    HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

12.7.4.1.4 Bilder auf Schaltflächen

Um dem oben vorgestellten Mehrzweck-Zählprogramm ein individuelles Design zu geben, kann man die Beschriftungen der Schaltflächen durch Bilder 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.

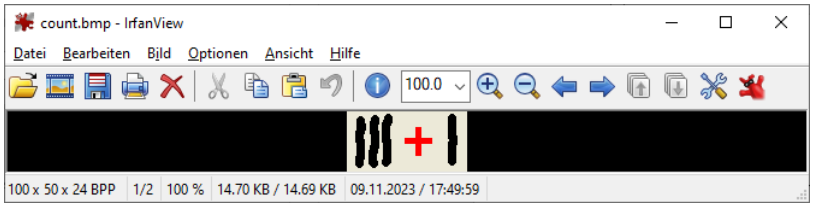
Wie man Bilder in ein Projekt aufnimmt und als Ressourcen nutzbar macht, wird im Abschnitt 12.4.3 beschrieben. Um ein Bild auf eine Schaltfläche zu befördern, verzichtet man im XAML-Code zum **Button**-Element auf das **Content**-Attribut und ergänzt stattdessen ein Element vom Typ **Image** als Wert der Inhaltseigenschaft z. B.:

```
<Button Name="count" Width="75" Height="40" Grid.Column="1" . . . IsDefault="True">
    <Image Source="count.png" />
</Button>
```

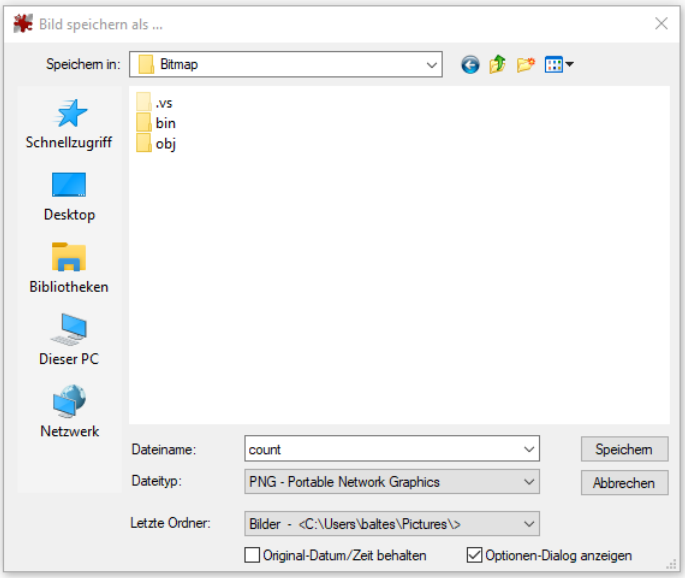
Wenn die Höhe des Schalters nicht der Bitmap-Datei überlassen werden soll, dann ist ein **Height**-Attribut erforderlich.

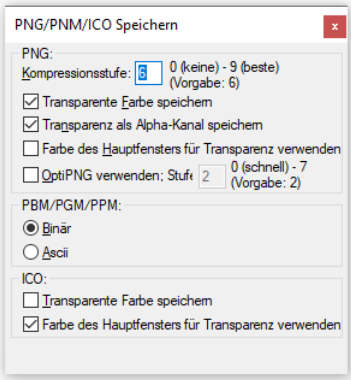
Macht man die Hintergrundfarbe der Bilder transparent, dann harmonieren die bemalten Schalter unabhängig vom eingestellten Windows-Design optisch mit den restlichen Fensterbestandteilen. Dieses Ziel ist leicht durch entsprechend präparierte Dateien zu realisieren. Eine **png**-Datei

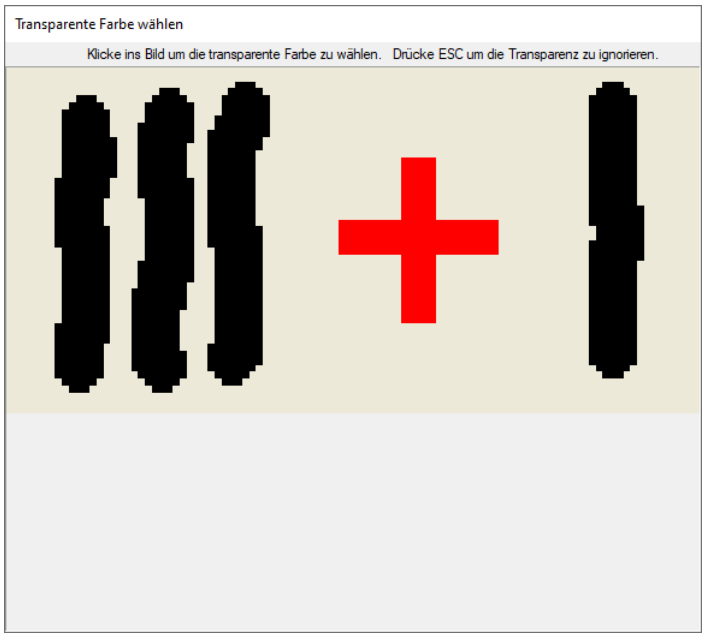
(*Portable Network Graphics*) mit transparenter Hintergrundfarbe lässt sich z. B. durch die Freeware **IrfanView**[1]



durch eine Option im **Speichern unter** - Dialog erstellen:







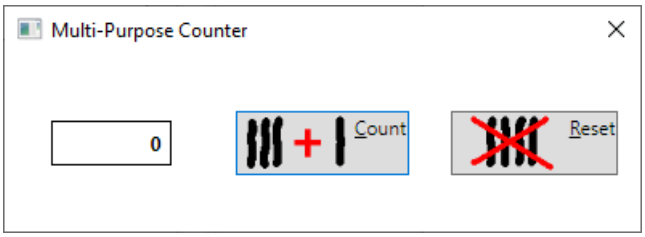
Soll ein **Button**-Steuerelement eine **Image**-Oberfläche *und* eine Zugriffstaste erhalten, dann führt das folgende Verfahren zum Ziel:

[1] Homepage: http://www.irfanview.de/

- Man verwendet einen horizontal orientierten **StackPanel**-Layoutcontainer als Wert der Inhaltseigenschaft.
- Man setzt neben das Bild noch ein **AccessText**-Element (siehe Abschnitt 12.7.4.1.3) auf die Schaltfläche, z. B.:

```
<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 eine zum Bild passende horizontale Ausdehnung des **Button**-Objekts lässt sich der unerwünschte optische Auftritt des **AccessText**-Elements



verhindern.

Ein Visual Studio - Projekt zum Programm mit den bebilderten Schaltflächen ist hier zu finden:

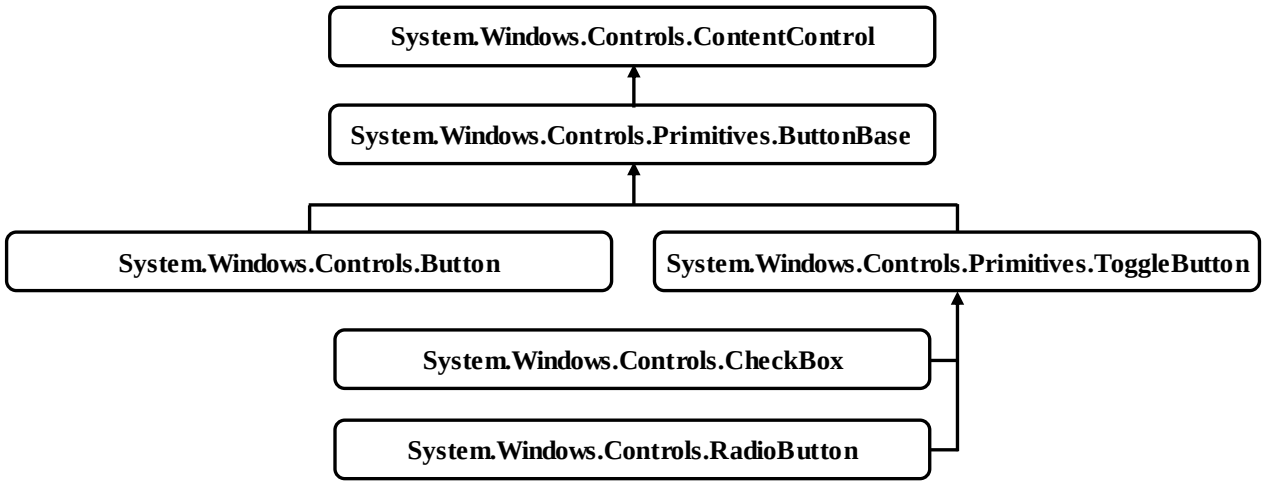
…\BspUeb\WPF\Steuerelemente\Button\Bitmap

12.7.4.2 Kontrollkästchen und Optionsfelder

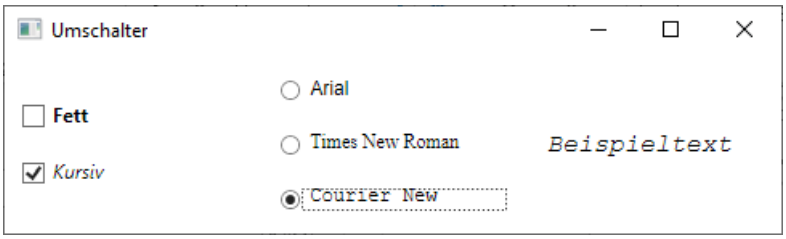
In diesem Abschnitt werden zwei Umschalter (zwischen den Werten *An* und *Aus* wechselnde Schalter) behandelt:

- Für **Kontrollkästchen** steht die Klasse **CheckBox** bereit.
- Für ein **Optionsfeld** aus Schaltern, von denen zu einem Zeitpunkt genau einer eingerastet sein kann, verwendet man Objekte der Klasse **RadioButton**.

Die Klassen **CheckBox** und **RadioButton** haben **ToggleButton** als gemeinsame Basisklasse und stammen zusammen mit der schon im Abschnitt 12.7.4.1 vorgestellten Klasse **Button** von der abstrakten Basisklasse **ButtonBase** ab:



Im folgenden Programm kann man für den Text eines **Label**-Steuerelements über zwei Kontrollkästchen die Schriftauszeichnung (normal, **fett**, *kursiv*, ***fett-kursiv***) und über ein Optionsfeld die Schriftartenfamilie wählen:



Beim Fensterdesign per XAML-Code kommt ein dreispaltiger **Grid**-Layoutcontainer zum Einsatz. In den beiden ersten Spalten arbeitet jeweils ein vertikal orientierter **StackPanel**-Layoutcontainer mit den **CheckBox**- bzw. **RadioButton**-Objekten. 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" Margin="10"
                      FontWeight="Bold" />
            <CheckBox Name="chkItalic" Content="Kursiv" Margin="10"
                      FontStyle="Italic"/>
        </StackPanel>
        <StackPanel  Grid.Column="1" VerticalAlignment="Center"
                     ButtonBase.Click="RadioButton_Click">
            <RadioButton Name="rbArial" Content="Arial" Margin="10"
                         FontFamily="Arial" IsChecked="True"/>
            <RadioButton Name="rbTimesNewRoman" Content="Times New Roman" Margin="10"
                         FontFamily="Times New Roman"/>
            <RadioButton Name="rbCourierNew" Content="Courier New" 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 beim Programmstart 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 Namen bilden eine Gruppe, z. B.:

```
<RadioButton Name="rbOne" GroupName="g1" Content="One" ... />
```

Für die beiden Kontrollkästchen (`chkBold` und `chkItalic` genannt) ist dieselbe **Click**-Behandlungsmethode `CheckBox_Click()` zuständig, die für das separate Ein- bzw. Ausschalten der Schriftattribute **fett** und *kursiv* sorgt:[1]

[1] Im Beispielprogramm wird die sogenannte *Ungarische Notation* zur Bezeichnung der Instanzvariablennamen verwendet, wobei ein Präfix den Datentyp andeutet (z. B. `lblTextbeispiel`). Die in früheren Zeiten der Windows-Programmierung sehr verbreitete Konvention gilt mittlerweile als veraltet. Microsoft empfiehlt auf der Webseite zu Namenskonventionen

```
private void CheckBox_Click(object sender, RoutedEventArgs e) {
    lblTextbeispiel.FontWeight = chkBold.IsChecked.GetValueOrDefault() ?
        FontWeights.Heavy : FontWeights.Normal;
    lblTextbeispiel.FontStyle = chkItalic.IsChecked.GetValueOrDefault() ?
        FontStyles.Italic : FontStyles.Normal;
}
```

Wer annimmt, die Eigenschaft **IsChecked** sei vom Typ **bool**, der empfindet die Formulierung des logischen Ausdrucks in den Konditionaloperatoren vermutlich als umständlich, z. B.:

```
chkBold.IsChecked.GetValueOrDefault()
```

Tatsächlich hat **IsChecked** aber den Typ **Nullable<bool>** bzw. **bool?** (vgl. Abschnitt 8.3), und die Methode **GetValueOrDefault()** liefert bei Anwendung auf eine **bool?** – Instanz die benötigte Instanz vom Typ **bool** , wobei deren Wert genau dann gleich **true** ist, wenn die angesprochene **bool?** – Instanz den definierten Wert **true** hat.

Weil es um ein Ereignis mit der Routingstrategie *Bubbling* geht, kann die Behandlungsmethode dem gemeinsamen **StackPanel**-Layoutcontainer zugeordnet werden, statt sie bei *beiden* **CheckBox**-Objekten zu registrieren (siehe Abschnitt 12.4):

```
<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 (e.Source is RadioButton rbSource)
        lblTextbeispiel.FontFamily = rbSource.FontFamily;
}
```

Die drei benötigten Objekte vom Typ **System.Windows.Media.FontFamily** können von den **RadioButton**-Objekten übernommen werden.[1]

Auch für `RadioButton_Click()` genügt eine Registrierung beim gemeinsamen **StackPanel**-Container:

```
<StackPanel Grid.Column="1" VerticalAlignment="Center"
            ButtonBase.Click="RadioButton_Click">
    <RadioButton Name="rbArial" . . ./> <RadioButton Name="rbTimesNewRoman" . . . />
    <RadioButton Name="rbCourierNew" . . . />
</StackPanel>
```

Ein Visual Studio - Projekt mit dem Beispielprogramm ist hier zu finden:

…\BspUeb\WPF\Steuerelemente\Umschalter

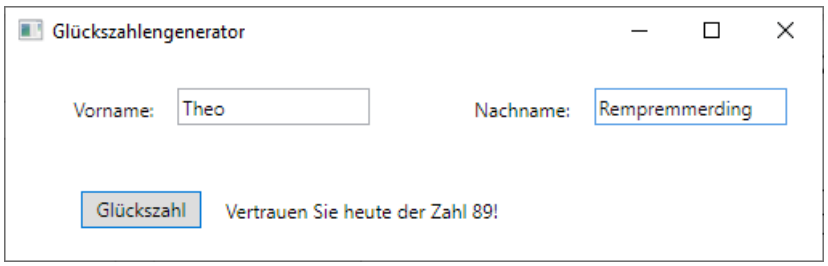
https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions

explizit: DO NOT use Hungarian notation. Wir verwenden sie im aktuellen Beispiel trotzdem, weil der zentrale Kritikpunkt nicht allzu zwingend erscheint: Bei einem Wechsel des Datentyps müsse der Name geändert werden. Der Wechsel des Datentyps ist bei einem Steuerelement unwahrscheinlich.

[1] Die Idee, die **FontFamily**-Objekte von den **RadioButton**-Objekten zu übernehmen, stammt von Jens Weber (Universität Trier).

12.7.4.3 Texteingabefelder

Kurze Texteingaben des Benutzers erfasst man mit Steuerelementen aus der Klasse **TextBox**. Auf dem folgenden 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" />
    </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" />
    </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 Glückszahlenanzeige verschwindet, sobald sich einer der zugrundeliegenden Texte ändert:

```
private void TextBox_TextChanged(object sender, TextChangedEventArgs e) {
   if (valToRemove) {
      info.Content = strInst;
      valToRemove = false;
   }
}
```

Weil es um ein Ereignis mit der Routingstrategie *Bubbling* geht, kann die Behandlungsmethode dem gemeinsamen **Grid**-Layoutcontainer zugeordnet werden, statt sie bei den beiden **TextBox**-Objekten zu registrieren:

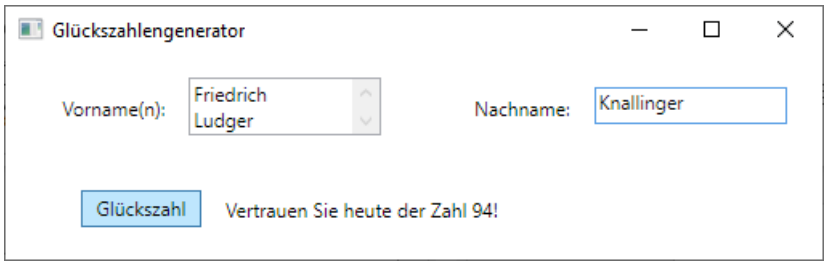
```
<Grid TextBox.TextChanged="TextBox_TextChanged">
  . . .
</Grid>
```

TextBox-Steuerelemente bieten den Benutzern einigen Bedienungskomfort, 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**
- Mehrstufige Wiederherstellung der zuletzt verworfenen Änderungen über **Strg+Y**
- Kontextmenü mit den Items **Ausschneiden**, **Kopieren** und **Einfügen**
- Rechtschreibekontrolle, zu aktivieren über die angefügte Eigenschaft **IsEnabled** der Klasse **SpellCheck**, z. B.:

```
<TextBox Name="antwort" SpellCheck.IsEnabled="True" />
```

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" />
```

Im Beispiel verliert die Ernennung des **Button**-Objekts zum Standardschalter (über den Wert **true** der Eigenschaft **IsDefault**, siehe Abschnitt 12.7.4.1.1) ihre Gültigkeit, wenn das **Enter**-berechtige mehrzeilige Texteingabefeld den Tastatur-Eingabefokus besitzt (siehe Abschnitt 12.7.4.4).

Trotz **TextWrapping** eignet sich die Klasse **TextBox** nur für kurze Texteingaben. Mit Hilfe des Steuerelements **RichTextBox** kann man einen kompletten Texteditor erstellen.

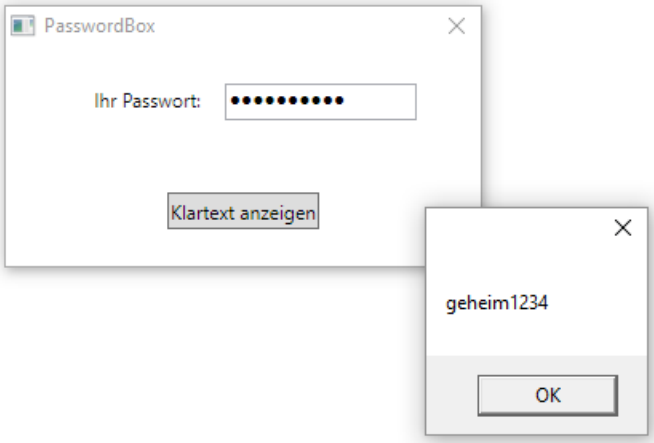
Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:

…\BspUeb\WPF\Steuerelemente\Texterfassung\TextBox

Zur Erfassung von **Passwörtern** steht die Klasse **PasswordBox** mit einer im Vergleich zur verwandten Klasse **TextBox** leicht modifizierten Funktionalität zur Verfügung:

- Statt der eingegebenen Zeichen werden gefüllte Kreise angezeigt.
- Die Eigenschaft **Text** wird durch die Eigenschaft **Password** ersetzt.
- Ein Passwort lässt sich nicht in die Zwischenablage schreiben.

Trotz des geänderten Eigenschaftsnamens ist ein erfasstes Passwort im Klartext vorhanden:



Durch den folgenden XAML-Code wird das **PasswordBox**-Objekt des Beispiels deklariert:

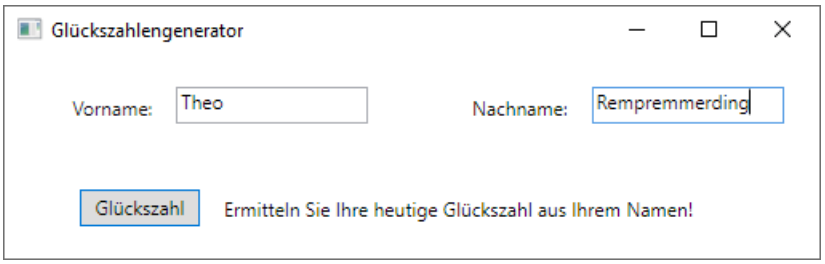
```
<PasswordBox Name="pw" Height="23" Width="120" />
```

Ein Visual Studio - Projekt mit dem Passwort-Programm ist hier zu finden:

…\BspUeb\WPF\Steuerelemente\Texterfassung\PasswordBox

12.7.4.4 Tastatur-Eingabefokus

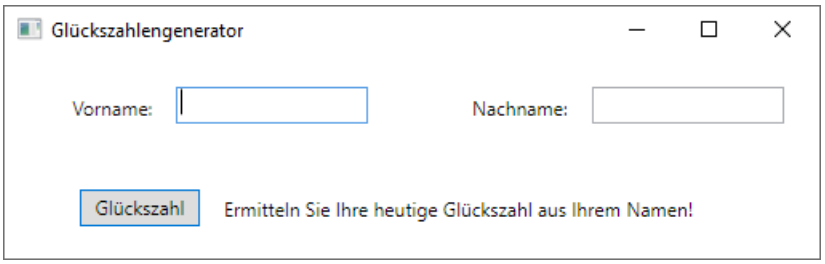
Im Beispielprogramm zum **TextBox**-Steuerelement (siehe Abschnitt 12.7.4.3) erhält der Schalter zur Anforderung einer persönlichen Glückszahl die Rolle des Standardschalters. Folglich kann er auch *dann* per **Enter**-Taste ausgelöst werden, wenn eines der Texteingabefelder den Eingabefokus besitzt (u. a. erkennbar an der Texteinfügemarke):



In diesem Beispielprogramm bietet es sich sogar an, die Schaltfläche über den Wert **false** für die Eigenschaft **Focusable** aus der Liste der fokussierbaren Steuerelemente zu entfernen, damit die Tabulatortaste nur noch zwischen den beiden Texteingabefeldern wechselt:

```
<Button Content="Glückszahl" . . . IsDefault="True" Focusable="False" />
```

Im Glückszahlengenerator sollte das Texteingabefeld für den Vornamen schon beim Programmstart den Eingabefokus haben:



Eine Lösungsmöglichkeit besteht darin, das zu privilegierende Steuerelement über die Methode **Focus()** der Klasse **UIElement** aufzufordern, den Fokus zu übernehmen, z. B.:

```
vorname.Focus();
```

Damit diese Anweisung beim Laden des Fensters ausgeführt wird, steckt man sie in eine Behandlungsmethode zum **Loaded**-Ereignis der Fensterklasse, z. B. so:

- Man markiert im WPF-Designer das Fenster (das **Window**-Objekt).
- Man fordert im **Eigenschaften**-Fenster per Mausklick auf das Symbol ⚡ die Liste mit den Ereignissen an.
- Man setzt einen Doppelklick auf das Texteingabefeld zum Ereignis **Loaded**. Daraufhin wird in der Code-Behind - Datei **MainWindow.xaml.cs** die Instanzmethode `Window_Loaded()` angelegt:

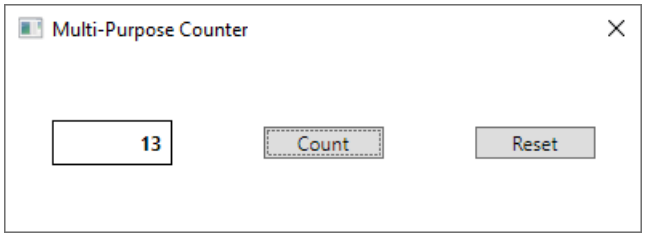
  ```
  private void Window_Loaded(object sender, RoutedEventArgs e) {
  }
  ```

  Diese Methode wird außerdem per XAML zur **Loaded**-Ereignisbehandlung registriert:

  ```
  <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" Loaded="Window_Loaded">
  ```

- Man ergänzt im Rumpf der Methode `Window_Loaded()` den oben beschriebenen **Focus()** - Aufruf.

Nachdem ein Schalter per Mausklick oder Tabulatortaste den **Tastatur-Eingabefokus** erhalten hat, kann er auch per Leertaste angesprochen werden. Diese Bedienungsoption ist optisch an einem gestrichelten Innenrand zu erkennen:[1]

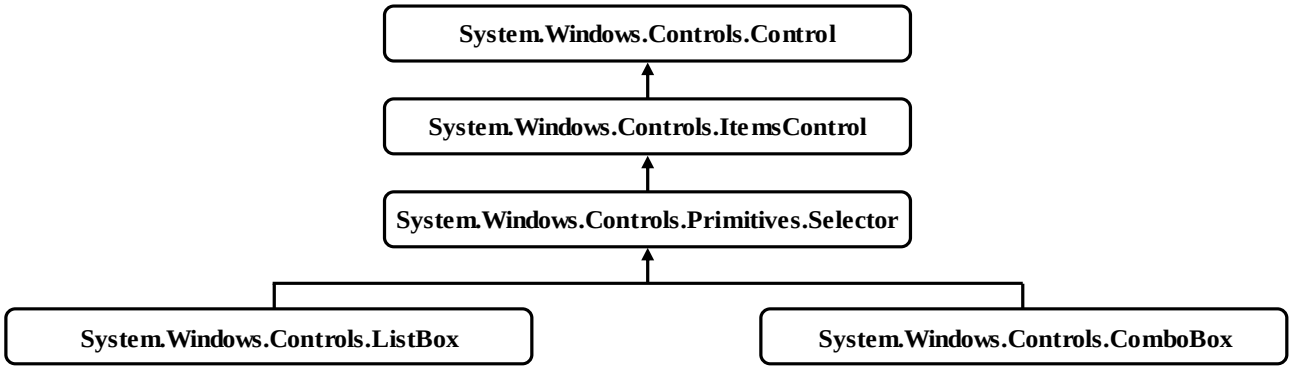


Wenn der *Standardschalter* einen gestrichelten Innenrand besitzt, dann entfällt die blaue Standardschalter-Umrandung.

[1] Nach einer Fokusübertragung per Tabulatortaste erscheint der gestrichelte Innenrand sofort. Bei einer Fokusübertragung per Maus wird gleichzeitig das **Click**-Ereignis ausgelöst, und der gestrichelte Innenrand erscheint erst, nachdem das Programm in den Hintergrund verdrängt und dann wieder in den Vordergrund geholt wurde.

12.7.4.5 Listen- und Kombinationsfelder

In diesem Abschnitt werden die Steuerelementklassen **ListBox** und **ComboBox** vorgestellt, die gemeinsame Vorfahren besitzen:

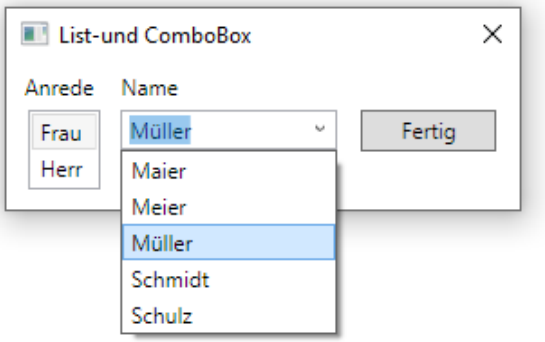


Ein **ListBox**-Steuerelement präsentiert eine Liste von Elementen, von denen der Benutzer (z. B. durch Mausklicks) eines oder mehrere wählen kann.

Das **ComboBox**-Steuerelement erfragt einen Text durch eine Kombination aus einem einzeiligen Texteingabefeld und einer Drop-Down - Liste, und der Benutzer hat *zwei* Möglichkeiten zur Beantwortung, sofern die Eigenschaft **IsEditable** den Wert **true** besitzt (siehe unten):

- Text eingeben
- Drop-Down - Liste öffnen und ein Item wählen

Das folgende Programm erlaubt per **ListBox** die Wahl einer Anrede und erleichtert per **ComboBox** die Eingabe des Namens, indem es eine Liste mit häufig auftretenden Namen bereithält:



Ein Visual Studio - Projekt mit dem Beispiel befindet sich im Ordner:

…\BspUeb\WPF\Steuerelemente\Selector\List-und ComboBox

12.7.4.5.1 ListBox mit Items befüllen

Ein **ListBox**-Steuerelement kann per XAML-Code über die Kollektionssyntax (siehe Abschnitt 12.3.2.3.3) mit Elementen versorgt werden, z. B.:

```
<ListBox Name="listBox" Margin="10,0,0,0" Width="40" SelectedIndex="0">
    <TextBlock>Frau</TextBlock>
    <TextBlock>Herr</TextBlock>
</ListBox>
```

Diese landen in einem Objekt der Klasse **ItemCollection**, das über die von **ItemsControl** geerbte Eigenschaft **Items** ansprechbar ist und Elemente vom Typ **ListBoxItem** verwaltet. Man darf Elemente von einem beliebigen „vorzeigbaren“ Typ in eine **ListBox** einfüllen, z. B. **TextBlock**-Elemente. Diese werden automatisch in **ListBoxItem**-Objekte verpackt. Mit expliziten **ListBoxItem**-Elementen sieht das Beispiel so aus:

```
<ListBox Name="listBox" Margin="10,0,0,0" Width="40" SelectedIndex="0">
    <ListBoxItem>
        <TextBlock>Frau</TextBlock>
    </ListBoxItem>
    <ListBoxItem>
        <TextBlock>Herr</TextBlock>
    </ListBoxItem>
</ListBox>
```

Im Beispiel (mit einfachen Texten für die Anrede) ist die explizite **ListBoxItem**-Verpackung überflüssig. In anderen Fällen ist der **ListBoxItem**-Einsatz sinnvoll, weil die von **ContentControl** abstammende Klasse z. B. einen Layoutmanager aufnehmen kann, sodass zusammengesetzte Items möglich sind (z. B. aus einem **Image** und einem **TextBlock** bestehend).[1]

Über die Kollektionsmethoden **Add()**, **Remove()** usw. lässt sich die **ItemCollection** zur Laufzeit per Programm verändern, z. B.:

```
listBox.Items.Add(new TextBlock {Text = "Hrl."});
```

12.7.4.5.2 ListBox mit Bindung an eine vorhandene Datenkollektion

Oft befinden sich die in einem **ListBox**-Steuerelement anzuzeigenden Elemente bereits in einer Datenkollektion, und es soll eine Datenbindung zwischen dieser Datenkollektion als Quelle und dem **ListBox**-Objekt als Ziel vorgenommen werden. Dazu wird die Datenkollektion der **ListBox**-Eigenschaft **ItemsSource** zugewiesen. Besonders geeignet sind Datenkollektionen aus der Klasse **ObservableCollection<T>** wegen ihrer Fähigkeit, Änderungen der Listenzusammenstellung per Ereignis an ein verbundenes **ListBox**-Steuerelement zu melden, das daraufhin seine Anzeige aktualisiert.

Sobald der **ItemsSource**-Eigenschaft eine externe Datenkollektion zugewiesen wurde, ist es nicht mehr möglich, die interne, per **Items**-Eigenschaft ansprechbare **ItemCollection** des **ListBox**-Steuerelements zu verändern. Ein Versuch führt zu einer **InvalidOperationException**.

Wir haben im `RssFeedReeder`-Projekt (vgl. vor allem Abschnitt 6.8.6) ein realistisches Beispiel für die Bevölkerung eines **ListBox**-Steuerelements durch die Elemente einer externen Datenkollektion kennengelernt. Dort haben wir 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; }
}
```

in eine Kollektion vom Typ **List<`RssItem`>** eingefüllt und eine Referenz auf diese Kollektion an die **ListBox**-Eigenschaft **ItemsSource** übergeben:[2]

---

[1] Auch die folgende Variante ist möglich, wobei dann aber die bei **TextBlock**-Elementen vorhandene (und in einem späteren Beispiel genutzte) Möglichkeit zum Formatieren der Texte fehlt:

```
<ListBox Name="listBox" Margin="10,0,0,0" Width="40" SelectedIndex="0">
    <ListBoxItem>Frau</ListBoxItem>
    <ListBoxItem>Herr</ListBoxItem>
</ListBox>
```

[2] Die permanente Synchronisation einer variablen und beobachtbaren Liste mit dem **ListBox**-Steuerelement haben wir im `RssFeedReeder`-Projekt nicht benötigt. Auf die folgende Weise könnte man aus der **List<`RssItem`>** - Kollektion `items` eine *beobachtbare* Kollektion erstellen:

```
ObservableCollection<RssItem> obsItems = new ObservableCollection<RssItem>(items);
```

Der Lösungsvorschlag stammt von der Webseite:

https://stackoverflow.com/questions/18095932/how-to-cast-a-listt-to-an-observablecollectiont-in-wpf/18095988

```
listBox.ItemsSource = items;
```

Um die **ListBox**-Items informativ und attraktiv zu präsentieren, verwendet 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="10,50,10,10"
     ScrollViewer.HorizontalScrollBarVisibility="Disabled"
     MouseDoubleClick="listBox_MouseDoubleClick">
  <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 12.3.2.3.3 zu Inhaltseigenschaften). Zur Versorgung der Eigenschaft **VisualTree** ist ein Wurzel-Layoutcontainer anzugeben.[1] Im Beispiel wird ein **StackPanel**-Layoutcontainer verwendet, der zwei vertikal gestapelte **TextBlock**-Elemente enthält. Deren **Text**-Eigenschaft wird jeweils per Markup-Erweiterung (vgl. Abschnitt 12.3.2.3.6) vom Typ **Binding** mit einer Eigenschaft (`Title` bzw. `Description`) der Elemente in der externen Datenkollektion verbunden.

12.7.4.5.3 ComboBox mit Items befüllen

Zur Erläuterung des **ComboBox**-Steuerelements kehren wir zum Beispielprogramm mit Wahlmöglichkeiten für Anrede und Namen zurück (siehe Einstieg im Abschnitt 12.7.4.5). Im XAML-Code zeigen sich kaum Unterschiede zwischen dem **ComboBox**- und dem **ListBox**-Element:

```
<ComboBox Name="comboBox" Margin="10,0,0,0" Width="120" IsEditable="True">
    <TextBlock>Maier</TextBlock>
     . . .
    <TextBlock>Schulz</TextBlock>
</ComboBox>
```

Es bestehen noch mehr Gemeinsamkeiten, z. B. …

- bei der (von **ItemsControl** geerbten) Eigenschaft **Items**, die auch bei der Klasse **ComboBox** auf ein Objekt der Klasse **ItemCollection** zeigt,
- und beim Verfahren zur Datenbindung.

In seiner **ItemCollection** verwaltet ein **ComboBox**-Objekt Elemente vom Typ **ComboBoxItem**, die analog zum Typ **ListBoxItem** (siehe Abschnitt 12.7.4.5.1) entweder implizit oder explizit erstellt werden (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.

---

[1] https://learn.microsoft.com/en-us/dotnet/api/system.windows.frameworktemplate.visualtree

12.7.4.5.4 Gewähltes Item ermitteln

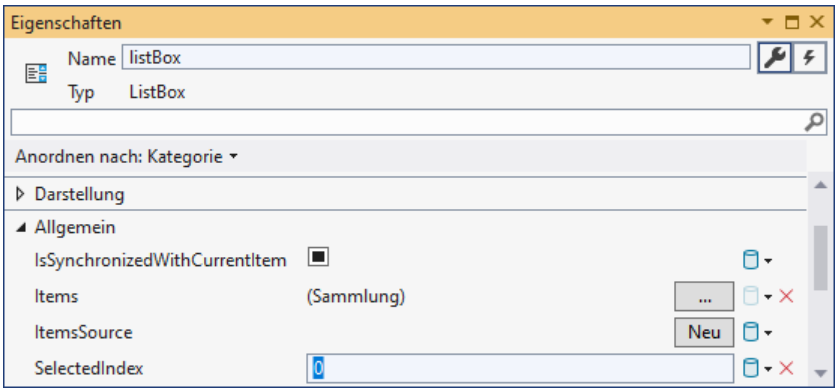
Wie sich in der folgenden **Click**-Behandlungsmethode zur Schaltfläche des Beispielprogramms mit Wahlmöglichkeiten für Anrede und Namen zeigt, ist bei einem **ListBox**-Objekt die aktuelle Wahl über die Eigenschaft **SelectedItem** ansprechbar:

```
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 die Textextraktion erst nach einer Typumwandlung möglich (zur Wahl des Zieltyps **TextBlock** siehe Abschnitt 12.7.4.5.1).

Von einem **ComboBox**-Objekt erfährt man die aktuelle Wahl bzw. Eintragung des Benutzers über die Eigenschaft **Text**. Beim Zugriff auf die hier ebenfalls vorhandene Eigenschaft **SelectedItem** ist zu beachten, dass eine Texteintragung durch den Benutzer zum Wert **null** führt.

Bei einem **ListBox**-Objekt kann man über die Eigenschaft **SelectedIndex** dafür sorgen, dass zur Bedienungserleichterung initial ein Listenelement markiert ist, z. B.:



Bei einem **ComboBox** wählt man mit derselben Eigenschaft ein Listenelement, das initial im Texteingabefeld erscheinen soll.

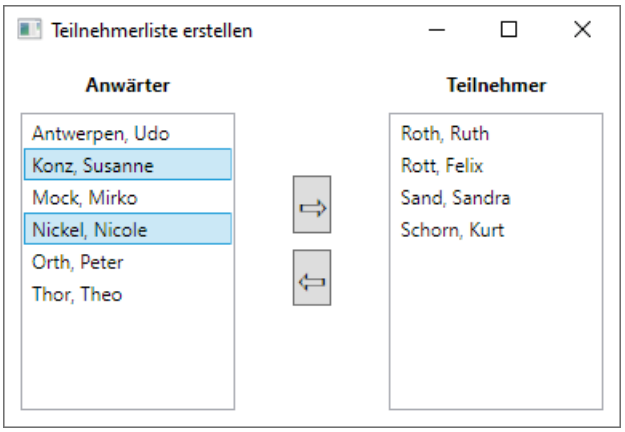
Die Klassen **ListBox** und **ComboBox** bieten zahlreiche Ereignisse an (z. B. **SelectionChanged**), die aber im aktuellen Beispielprogramm nicht von Interesse sind.

12.7.4.5.5 Dynamische Listen und Mehrfachwahl

In einem weiteren **ListBox**-Beispielprogramm sollen die folgenden Funktionen demonstriert werden:

- Dynamische Aufnahme und Entfernung von Listenelementen
- Mehrfachauswahl von Listenelementen

Wir befüllen ein erstes **ListBox**-Objekt mit den Namen von Anwärtern (für was auch immer) und halten ein zweites **ListBox**-Objekt für die ausgewählten Teilnehmer bereit. In den beiden **ListBox**-Steuerelementen wird jeweils das interne **ItemCollection**-Objekt zur Verwaltung der Namen verwendet (siehe Abschnitt 12.7.4.5.1). Auf dem Anwendungsfenster befinden sich zwischen den beiden **ListBox**-Objekten zwei Transportschalter zum Verschieben von Namen zwischen den Listen:



Im XAML-Code wird durch verschachtelte Layoutcontainer für eine gute Bedienbarkeit bei unterschiedlichen Fenstergrößen 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>Antwerpen, Udo</TextBlock>
                <TextBlock>Konz, Susanne</TextBlock>
                . . .
                <TextBlock>Sand, Sandra</TextBlock>
                <TextBlock>Schorn, Kurt</TextBlock>
                <TextBlock>Thor, Theo</TextBlock>
            </ListBox>
        </DockPanel>
        <StackPanel Grid.Column="1" VerticalAlignment="Center"
                    Button.Click="Button_Click">
            <Button Name="cmdRein" Content="&#x21e8;" HorizontalAlignment="Center"
                    Margin="5" FontSize="24" FontWeight="Bold" />
            <Button Name="cmdRaus" Content="&#x21e6;" HorizontalAlignment="Center"
                    Margin="5" FontSize="24" FontWeight="Bold"/>
        </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" />
        </DockPanel>
    </Grid>
</Window>
```

Dem dreispaltigen Top-Level - Container vom Typ **Grid** sind untergeordnet:

- Links und rechts jeweils ein **DockPanel**-Container, der ein **Label**- und ein **ListBox**-Objekt verwaltet, wobei das zuletzt eingefügte **ListBox**-Objekt den gesamten unverbrauchten Raum einnimmt.
- In der Mitte ein vertikal orientierter **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:

- Bei gedrückter **Strg**-Taste lassen sich *mehrere* Elemente nacheinander per Mausklick markieren.
- Um eine Serie von hintereinander positionierten Elementen zu markieren, klickt man zunächst auf das erste und danach mit gedrückter **Umschalt**-Taste auf das letzte Element.
- Durch gemeinsames Drücken von **Strg**- und **Umschalt**-Taste lassen sich mehrere Serien und Einzelelemente durch Mausklicks in die Markierung einbeziehen.

Erhält die Eigenschaft **SelectionMode** den Wert **Multiple**, dann haben die **Strg**- und die **Umschalt**-Taste keine Funktion, jedoch können durch einfache Mausklicks mehrere Elemente nacheinander markiert werden.

Für die Pfeile auf den Transportschaltern des Programms bietet die folgende Webseite des Unicode-Konsortiums eine reichhaltige Auswahl:

https://www.unicode.org/charts/PDF/U2190.pdf

In der **Click**-Ereignisbehandlungsmethode `Button_Click()` zu den beiden Schaltflächen (`cmdRein`, `cmdRaus`) werden die markierten Elemente über die **ListBox**-Eigenschaft **SelectedItems** angesprochen. Die zeigt auf ein Objekt, das die Schnittstelle **System.Collections.IList** erfüllt.

```
private void Button_Click(object sender, RoutedEventArgs e) {
    if (e.Source == cmdRein)
        MoveItems(listeAnwaerter, listeTeilnehmer);
    else
        MoveItems(listeTeilnehmer, listeAnwaerter);

    void MoveItems(ListBox listFrom, ListBox listTo) {
        Object[] tmpFrom = new Object[listFrom.SelectedItems.Count];
        listFrom.SelectedItems.CopyTo(tmpFrom, 0);
        foreach (object o in tmpFrom) {
            listFrom.Items.Remove(o);
            listTo.Items.Add(o);
        }
        listTo.Items.SortDescriptions.Clear();
        listTo.Items.SortDescriptions.Add(sortDescription);
    }
}
```

Die zu verschiebenden Namen werden in einem temporären **Object**-Array gesammelt, über den anschließend eine **foreach**-Schleife iteriert, wobei die Listen verändert werden. Wenn eine **foreach**-Schleife z. B. über die Kollektion `listeAnwaerter.SelectedItems` iteriert, dann kommt es zu einer **InvalidOperationException**, weil sich die Kollektion einer **foreach**-Schleife während der Iteration nicht ändern darf.

Nach einer Änderung der Listen muss dafür gesorgt werden, dass sie in sortiertem Zustand bleiben. Die **ItemCollection** einer **ListBox** besitzt eine Eigenschaft namens **SortDescriptions**, die auf ein Objekt der Klasse **SortDescriptionCollection** zeigt. Diese Kollektion enthält Instanzen der Struktur **SortDescription**, die jeweils ein Kriterium zum Sortieren der **ItemCollection**-Elemente beschreiben über …

- den Namen der Elementeigenschaft, nach der sortiert werden soll,
- und die Sortierrichtung (auf- oder absteigend).

Im Beispiel sollen **TextBlock**-Elemente nach ihrer **Text**-Eigenschaft aufsteigend sortiert werden. Daher wird in der Fensterklasse die folgende **SortDescription** - Member-Instanz zur Verwendung von beiden **ListBox**-Steuerelementen erstellt:

```
public partial class MainWindow : Window {
    SortDescription sortDescription;

    public MainWindow() {
        InitializeComponent();
        sortDescription = new SortDescription("Text", ListSortDirection.Ascending);
    }

   . . .

}
```

Offenbar muss man nach der Aktualisierung einer **ItemCollection** die zugehörige **SortDescriptions**-Kollektion „ändern“, damit sie neu angewendet wird, z. B.:[1]

```
listeAnwaerter.Items.SortDescriptions.Clear();
listeAnwaerter.Items.SortDescriptions.Add(sortDescription);
```

Weil das von `Button_Click()` behandelte **Click**-Ereignis die Routingstrategie *Bubbling* verwendet, kann die Behandlungsmethode dem gemeinsamen **StackPanel**-Container zugeordnet werden, statt sie bei beiden **Button**-Objekten zu registrieren (vgl. Abschnitt 12.4):

```
<StackPanel Grid.Column="1" VerticalAlignment="Center"
            Button.Click="Button_Click">
    <Button Name="cmdRein" . . . />
    <Button Name="cmdRaus" . . . />
</StackPanel>
```

In der Ereignismethode lässt sich daher die Ereignisquelle nicht über den Parameter *sender* feststellen, der stets auf das **StackPanel**-Objekt zeigt. Stattdessen ist die Eigenschaft **Source** des Ereignisbeschreibungsobjekts aus der Klasse **RoutedEventArgs** zu verwenden, auf das der zweite Aktualparameter zeigt.

Ein Visual Studio - Projekt mit dem Beispiel befindet sich im Ordner:

…\BspUeb\WPF\Steuerelemente\Selector\AddRemoveItems

12.7.4.6 ToolTip

Über ein **ToolTip**-Objekt realisiert man eine temporär sichtbare Information zu einem Steuerelement:

- Die Anzeige erscheint in der Nähe des Steuerelements, wenn sich die Maus über diesem Element befindet.
- Die Anzeige endet, wenn der Mauszeiger den Bereich des zu erläuternden Steuerelements verlässt, oder die Anzeigedauer abgelaufen ist (Voreinstellung: 5 Sekunden).

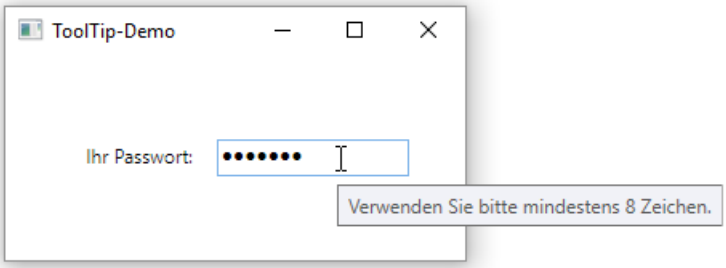
In der Regel genügt es, im XAML-Code für das zu erläuternde Element ein **ToolTip**-Attribut zu formulieren, z. B.:

---

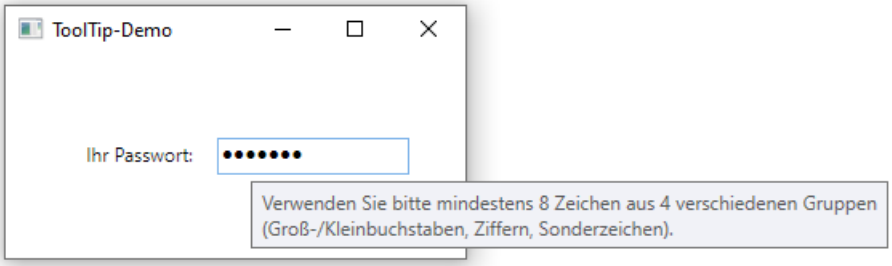
[1] Das Aktualisieren der Sortierung nach einer Änderung der Liste sollte einfacher zu erreichen sein bzw. automatisch erfolgen. Wenn es ein entsprechendes Verfahren gibt, dann ist es gut versteckt.

```
<PasswordBox ToolTip="Verwenden Sie bitte mindestens 8 Zeichen"
             Height="23" Name="pw" Width="120"/>
```

Hier erhalten die Benutzer einen Tipp zur Passwortlänge:



Wenn sich ein **ToolTip**-Text über mehrere Zeilen erstrecken soll,



dann eignet sich die folgende Lösung mit einem Eigenschaftselement und einem **TextBlock** als Wert für die Eigenschaft **ToolTip**:

```
<PasswordBox Height="23" Name="pw" Width="120">
    <PasswordBox.ToolTip>
        <TextBlock>
        Verwenden Sie bitte mindestens 8 Zeichen aus 4 verschiedenen Gruppen
        <LineBreak/>
        (Groß-/Kleinbuchstaben, Ziffern, Sonderzeichen).
        </TextBlock>
    </PasswordBox.ToolTip>
</PasswordBox>
```

Um die voreingestellte Anzeigedauer von 5000 Millisekunden zu verändern, verwendet man die angefügte Eigenschaft (vgl. Abschnitt 12.5.2) **ShowDuration** der Klasse **ToolTipService** in einem Attribut für das zu erläuternde Element, z. B.:

```
<PasswordBox Name="pw" ToolTipService.ShowDuration="3000" Height="23" Width="120">
    <PasswordBox.ToolTip>
        . . .
    </PasswordBox.ToolTip>
</PasswordBox>
```

Weitere **ToolTip**-Einstellungen (z. B. Anzeigeposition, Startverzögerung, Ausstattung mit einem Schlagschatten) lassen sich ebenfalls über angefügte Eigenschaften der Klasse **ToolTipService** ändern.

Ein Visual Studio - Projekt mit dem Beispiel befindet sich im Ordner:

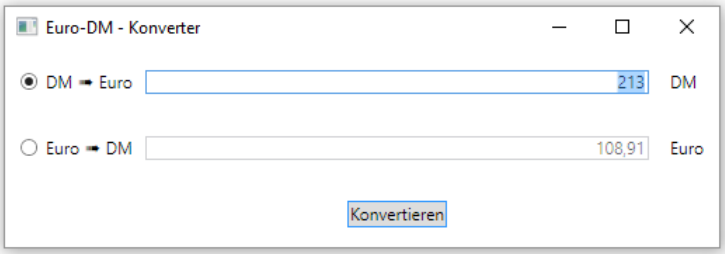
…\BspUeb\WPF\Steuerelemente\ToolTip

12.8 Übungsaufgaben zum Kapitel 12

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. WinUI 3 wird in Zukunft keine große Rolle spielen.
2. Wird eine WPF-Klasse beerbt, dann ist ein Ereignis dieser Klasse bequem durch das Überschreiben der zugehörigen On-Methode zu behandeln.
3. Für das GUI-Design einer WPF-Anwendung muss die XAML verwendet werden.
4. Befinden sich in einem Layoutcontainer mehrere **Button**-Objekte, dann können deren **Click**-Ereignisse durch eine gemeinsame Methode behandelt werden, die beim Layoutcontainer registriert wird.
5. Die WPF-Layoutcontainer ermöglichen ein responsives Layout, sodass ein Fenster gut bedienbar bleibt, wenn ein Benutzer die Größe und/oder das Seitenverhältnis ändert.

2) Erstellen Sie eine verbesserte Variante des Euro-DM - Konverters, den wir im Abschnitt 3.3.7 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"
```

13 Ausnahmebehandlung

Durch Programmierfehler (z. B. versuchter Feldzugriff mit ungültigem Indexwert, Ganzzahldivision durch 0) oder durch besondere Umstände (z. B. Speichermangel, unterbrochene Netzwerkverbindung) kann die reguläre Ausführung einer Methode scheitern. C# bietet ein modernes Verfahren zur Meldung und Behandlung von Störungen der Programmausführung: 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 Methode wird somit über das Problem informiert und mit Daten für die Behandlung versorgt.

Initiiert wird die Meldung eines Ausnahmefehlers …

- entweder 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.
- oder vom **Anwendungsprogramm**, wozu auch die verwendeten Bibliothekstypen gehören.
  Zum Werfen einer Ausnahme dient die **throw**-Anweisung (siehe Abschnitt 13.5).

Statt von einem *Ausnahmefehler* werden wird ab jetzt der Kürze halber meist von einer *Ausnahme* sprechen, wenn kein Missverständnis droht.

Die unmittelbar von einer Ausnahme betroffene Methode steht meist am Ende einer Sequenz verschachtelter Aufrufe, und entlang der Aufrufersequenz haben die beteiligten Methoden jeweils die folgenden Optionen:

- das 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 sogenannten *Exception-Handler*) bereithält. Gegebenenfalls wird dieser Exception-Handler ausgeführt und erhält quasi als Aktualparameter das Ausnahmeobjekt mit Informationen über das Problem. Nach der Ausnahmebehandlung kann die Methode ...
  - entweder ihre Tätigkeit mit einem angepassten Handlungsplan fortsetzen
  - oder ihrerseits ein Ausnahmeobjekt werfen (entweder das ursprüngliche oder ein informativeres) und somit die Kontrolle an ihren Aufrufer zurückgeben.
- das 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.

Man kann von keinem Programm erwarten, dass es unter allen widrigen Umständen normal funktioniert, doch müssen Schäden (z. B. Datenverluste) nach Möglichkeit verhindert werden. Außerdem sollte der Anwender eine nützliche Information zum aufgetretenen Problem erhalten, und nötigenfalls sollten technische Informationen zur Klärung der Problemursache hinterlassen werden (z. B. durch einen Eintrag in einer Protokolldatei). Bei vielen Methodenaufrufen ist es ratsam, auf Störungen des normalen Ablaufs vorbereitet zu sein. Dies folgt schon aus **Murphy’s Law** (zitiert nach Wikipedia):[1]

> Anything that 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.

---

[1] https://de.wikipedia.org/wiki/Murphys_Gesetz

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 eine eigene Ausnahmeklasse definiert.

13.1 Unbehandelte Ausnahmen

Findet die CLR zu einer geworfenen Ausnahme entlang der Aufrufersequenz bis hinauf zur Methode **Main()** keinen passenden Exception-Handler, dann wird das Programm mit einer Fehlermeldung beendet.[1] Das folgende Konsolenprogramm soll die Fakultät zu einer Zahl berechnen, die beim Start als Befehlszeilenargument übergeben wurde. 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 BCL-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("Beim Programmstart 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 *= i;
         Console.WriteLine($"Fakultät von {args[0]}: {fakul}");
      }
      else
         Console.WriteLine("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
      Console.Read();
   }
}
```

Ein Programm sollte sich generell bemühen, Ausnahmefehler nach Möglichkeit durch Kontrollmaßnahmen (z. B. Parametervalidierung) zu verhindern. Im Beispiel überprüft die Methode **Main()**, ob tatsächlich ein Befehlszeilenargument in `args[0]` vorhanden ist, bevor sie diese **String**-Referenz beim Aufruf der Methode `Kon2Int()` als Aktualparameter übergibt. So wird eine Ausnahme vom Typ **IndexOutOfRangeException** vermieden, wenn der Benutzer das Programm *ohne* Befehlszeilenargument startet.

---

[1] Diese Aussage stimmt *nicht* bei der Multithreading-Programmierung mit der sogenannten *Task Parallel Library*. Im Abschnitt 17.4.6 von Baltes-Götz (2021) finden sich Empfehlungen für den Umgang mit unbehandelten Ausnahmen, die innerhalb einer sogenannten *Aufgabe* auftreten.

In diesem Fall beendet **Main()** das Programm durch einen Aufruf der statischen Methode **Environment.Exit()**, der als Aktualparameter ein Exit Code übergeben wird. Nach einem beendeten Programmeinsatz unter Windows in einem per **cmd.exe** gestarteten Konsolenfenster befindet sich der **return**-Wert in der Pseudo-Umgebungsvariablen **errorlevel**, z. B.:

```
>fakul
Kein Argument angegeben

>echo %errorlevel%
1
```

Nach einem störungsfrei verlaufenen Programmeinsatz enthält **errorlevel** in der Regel den Exit Code 0.

Eine WPF-Anwendung sollte *nicht* durch den „aggressiven" Methodenaufruf **Environment.Exit()** beendet werden. Um eine WPF-Anwendung per Programm zu beenden, eignet sich die **Application**-Methode **Shutdown()**.

Die Reaktion des Beispielprogramms auf einen Start ohne Befehlszeilenargument kann als akzeptabel gelten. Anstelle einer für viele Benutzer wenig hilfreichen Fehlermeldung durch das Laufzeitsystem

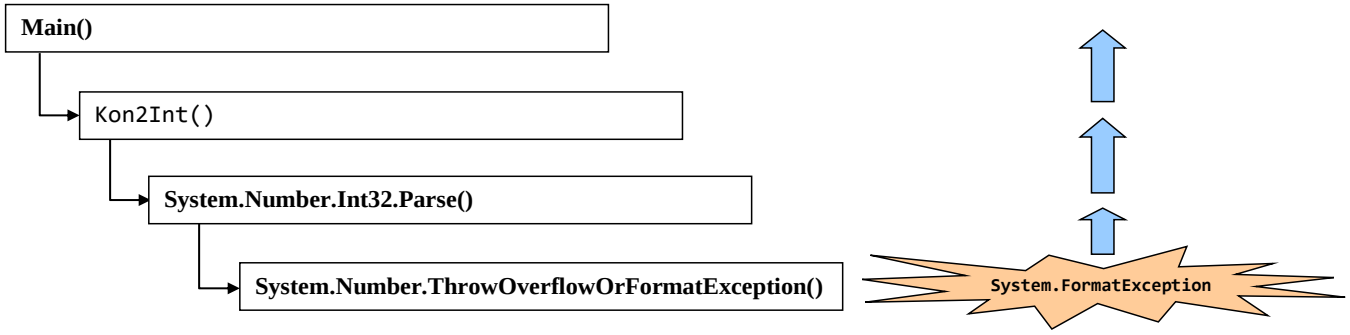
```
Unbehandelte Ausnahme: System.IndexOutOfRangeException: Der Index war
außerhalb des Arraybereichs.
   bei Fakul.Main(String[] args) in U:\ ... \Fakul.cs:Zeile 18.
```

erscheint eine verständliche Erläuterung des Problems:

```
Beim Programmstart kein Argument angegeben
```

Die Methode `Kon2Int()` lässt den **String**-Parameter durch die statische **Int32**-Methode **Parse()** interpretieren und überprüft, ob die ermittelte **int**-Zahl außerhalb des zulässigen Wertebereichs für eine Fakultätsberechnung (mit **double**-Ergebniswert) liegt, und meldet ggf. den Wert -1 als Fehlerindikator zurück. **Main()** kennt die spezielle Bedeutung dieser Rückgabe, sodass die unsinnige Fakultätsberechnung für ein negatives Argument und der wenig hilfreiche Ergebniswert Unendlich für ein Argument > 170 vermieden werden. Diese traditionelle Fehlerbehandlung per **Rückgabewert** (engl.: *return code*) ist *nicht* grundsätzlich als überholt und ineffizient zu bezeichnen, aber in vielen Situationen der gleich vorzustellenden Kommunikation über Ausnahmeobjekte unterlegen (siehe Abschnitt 13.3 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 **Int32**-Methode **Parse()** lässt die Formatprüfung im Hintergrund durch die **Number**-Methode **ParseBinaryInteger()** ausführen, die bei einem negativen Prüfergebnis die **Number**-Methode **ThrowOverflowOrFormatException()** beauftragt, eine **FormatException** zu werfen. Diese Ausnahme wird von der CLR entlang der Aufrufsequenz an alle beteiligten Methoden bis hinauf zu **Main()** gemeldet:



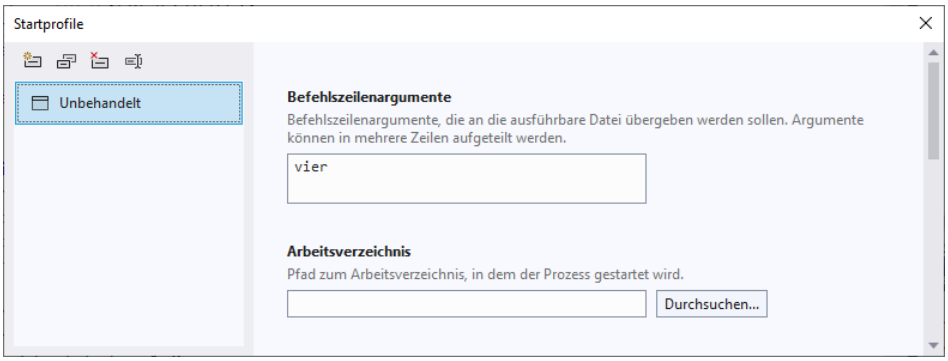
Weil kein Aufrufer eine geeignete Behandlungsroutine bereithält, beendet die CLR das Programm mit der folgenden Fehlermeldung:

```
Unhandled exception. System.FormatException:
      The input string 'vier' was not in a correct format.
   at System.Number.ThrowOverflowOrFormatException(ParsingStatus status,
                                                   ReadOnlySpan`1 value, TypeCode type)
   at System.Int32.Parse(String s)
   at Fakul.Kon2Int(String instr) in C:\ ... \Fakul.cs:line 5
   at Fakul.Main(String[] args) in C:\ ... \Fakul.cs:line 18
```

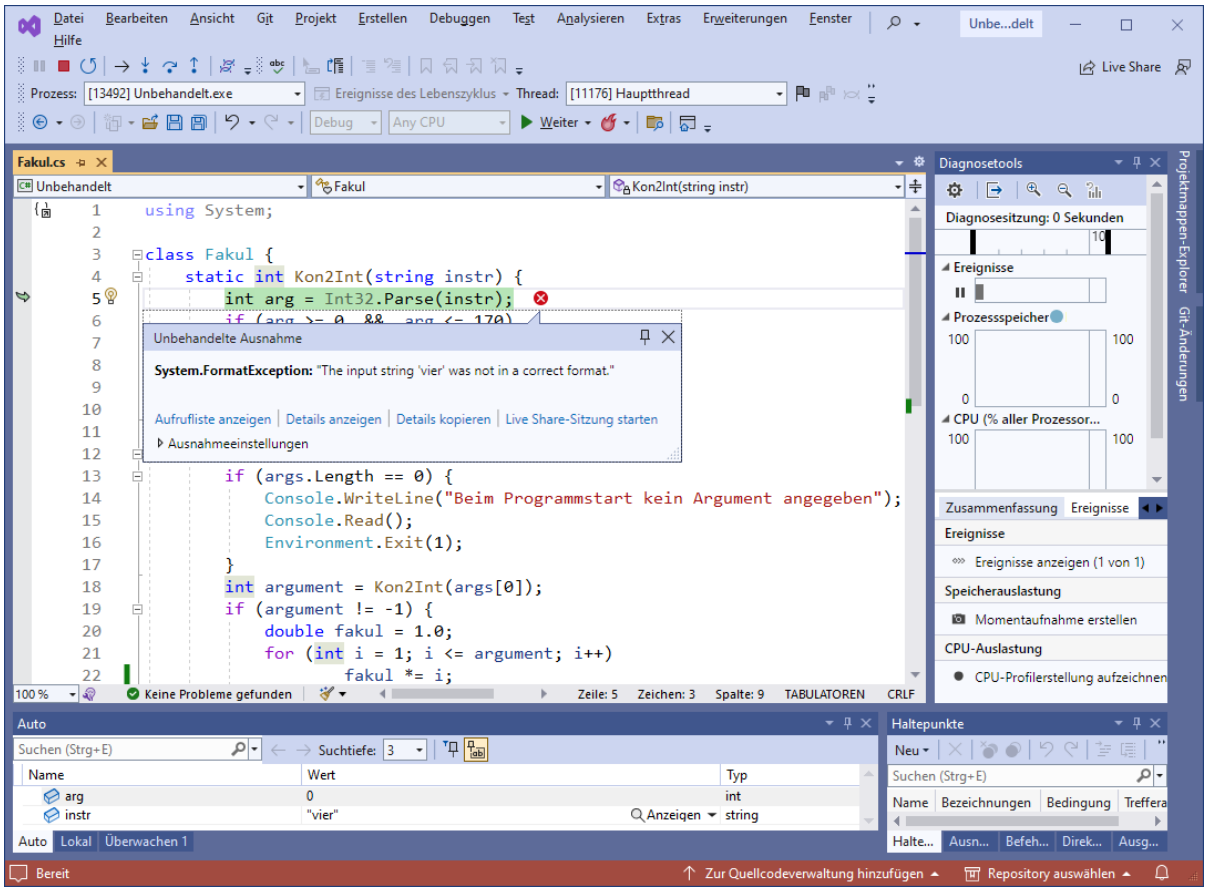
Um die Reaktion des Programms auf ein (fehlerhaftes) Befehlszeilenargument zu beobachten, muss man es nicht außerhalb der Entwicklungsumgebung in einem Konsolenfenster starten. Das Visual Studio 2022 nimmt nach dem Menübefehl

Debuggen > Debugeigenschaften

die zu simulierenden **Befehlszeilenargumente** entgegen, z. B.:



Endet eine aus dem Visual Studio (z. B. mit **F5**) im **Debug-Modus** gestartete Anwendung mit einer unbehandelten Ausnahme, dann informiert die Entwicklungsumgebung über den Typ der Ausnahme und die Unfallstelle:



Um am Projekt weiterarbeiten zu können, muss der Debug-Modus beendet werden (z. B. mit dem Symbolschalter ■ oder mit der Tastenkombination **Umschalt+F5**).

Ein Visual Studio - Projekt mit dem aktuellen Entwicklungsstand des Fakultätsberechnungsprogramms ist hier zu finden:

…\BspUeb\Ausnahmebehandlung\Unbehandelt

13.2 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.

13.2.1 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 aus der ersten Ausnahmeklasse
    oder aus einer daraus abgeleiteten Klasse
}
// Optional können weitere Ausnahmen abgefangen werden:
catch (Ausnahmeklasse2 Parametername) {
    Anweisungen für die Behandlung von Ausnahmen der zweiten Ausnahmeklasse
    oder aus einer daraus abgeleiteten Klasse
}

  . . .

// Optionaler finally-Block mit Abschluss- bzw. Bereinigungsarbeiten.
// Bei vorhandenem finally-Block, ist kein catch-Block erforderlich.
finally {
    Anweisungen, die unabhängig vom Auftreten einer Ausnahme ausgeführt werden sollen
}
```

Die Anweisungen für den ungestörten Ablauf setzt man in den **try**-Block. Nachdem eine Anweisung des **try**-Blocks eine Ausnahme verursacht oder aktiv geworfen hat, werden die weiteren Anweisungen des **try**-Blocks *nicht* mehr ausgeführt. In einem **try**-Block sollten Anweisungen stehen, die allesamt erfolgreich ausgeführt werden müssen, damit ein sinnvolles Ergebnis entsteht.

Treten bei der Ausführung dieses überwachten Blocks *keine* Fehler auf, dann wird das Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**-Block ausgeführt wird.

C# erlaubt die folgenden Varianten der **try**-Anweisung:

- **try-catch** - Anweisung
- **try-finally** - Anweisung
- **try-catch-finally** - Anweisung

Ein **try**-, **catch**- oder **finally**-Block benötigt auch dann ein einrahmendes Paar geschweifter Klammern, wenn nur *eine* Anweisung enthalten ist.

Weil es der obigen Syntaxbeschreibung im Quellcodedesign trotz Unterstützung durch Kommentare an Präzision fehlt, sollen Sie in einer Übungsaufgabe ein Syntaxdiagramm erstellen (siehe Abschnitt 13.7).

13.2.1.1 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 die Parameterliste eines **catch**-Blocks 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 den zugehörigen Anweisungsblock 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 im Abschnitt 13.2.2.

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:

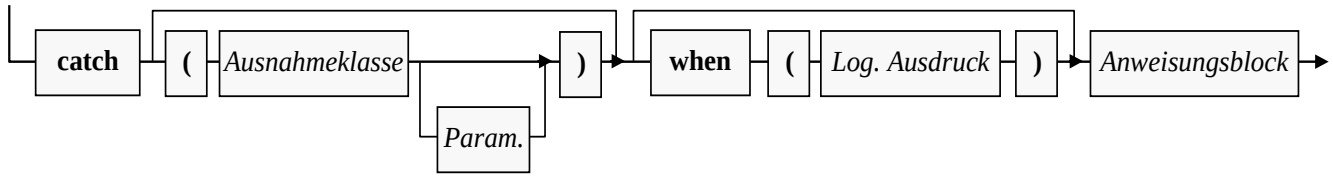
  catch {
  Anweisungen für die Behandlung der Ausnahmeklasse ***Exception***
  }

  Die Behandlung der Ausnahmeklasse **Exception** kommt z. B. dann in Frage, wenn im **catch**-Block lediglich ein Eintrag in eine Protokolldatei geschrieben und dann die abgefangene Ausnahme erneut geworfen werden soll, was durch die folgende **throw**-Anweisung geschehen kann (siehe Abschnitt 13.5):

  ```
  throw;
  ```

Seit C# 6.0 kann der Zuständigkeitsbereich eines **catch**-Blocks durch eine **when**-Klausel eingeschränkt werden:

catch-Block



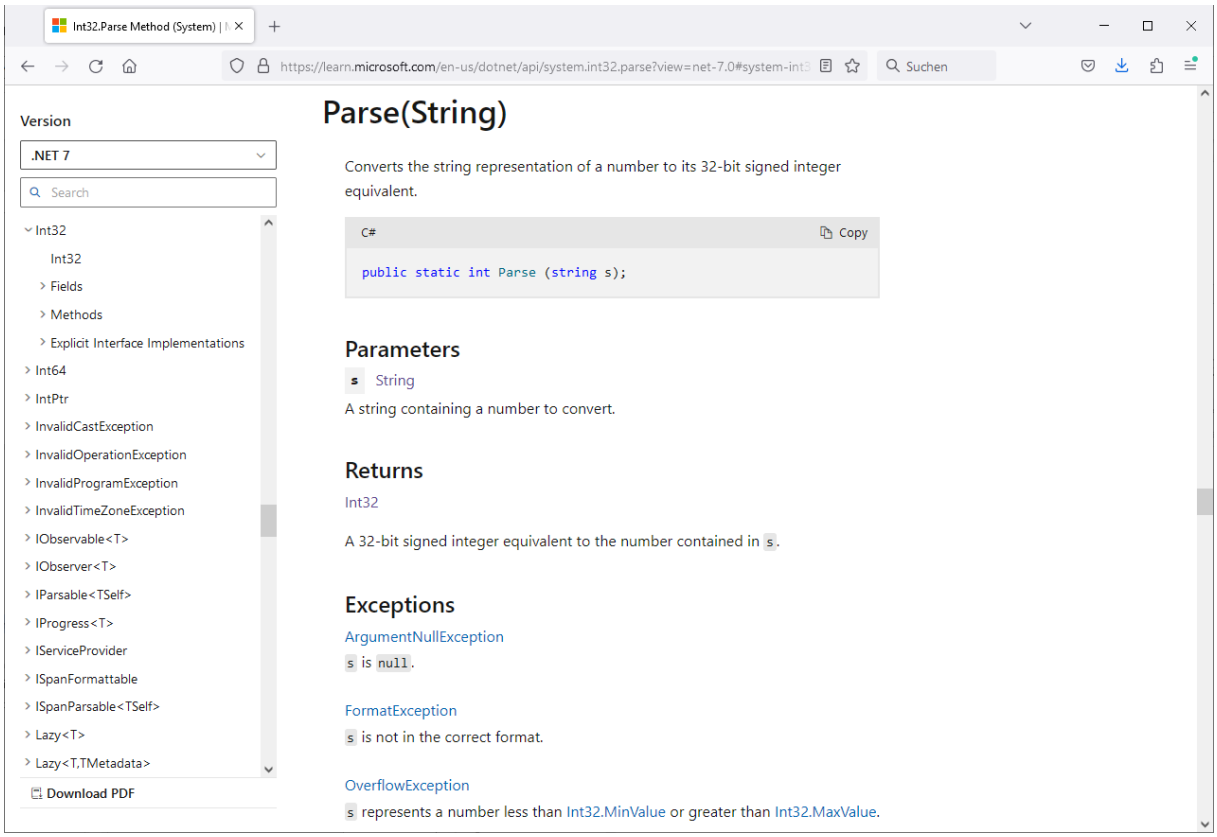
Der **catch**-Block wird nur dann ausgeführt, wenn ...

- eine Ausnahme zu seiner Klasse oder zu einer abgeleiteten Klasse gehört,
- *und* der logische Ausdruck den Wert **true** besitzt.

Hinsichtlich der Sequenz von **catch**-Blöcken gilt bei Verwendung von **when**-Klauseln (kurz: *Filterbedingungen*):

- Es sind mehrere **catch**-Blöcke mit derselben Ausnahmeklasse erlaubt, sofern sich die Filterbedingungen unterscheiden.
- Von mehreren **catch**-Blöcken mit derselben Ausnahmeklasse wird nur der erste zutreffende ausgeführt.
- Befindet sich unter mehreren **catch**-Blöcken mit derselben Ausnahmeklasse ein Block *ohne* Filterbedingung, dann muss dieser **catch**-Block zuletzt aufgeführt werden, weil er den größten Zuständigkeitsbereich besitzt.

Welche Ausnahmen von den Methoden eines BCL-Typs zu erwarten sind, erfährt man in der Dokumentation, z. B. bei der Methode **Int32.Parse(String)**:



Neben der bereits besprochenen **System.FormatException** (Zeichenfolge nicht konvertierbar) sind bei **Int32.Parse(String)** auch Ausnahmen aus den folgenden Klassen möglich:

- **System.ArgumentNullException**
  Es ist keine Zeichenfolge vorhanden.
- **System.OverflowException**
  Das Konvertierungsergebnis kann nicht in einer **int**-Variablen abgelegt werden.

In der folgenden Variante der Methode `Kon2Int()` werden die von **Int32.Parse()** zu erwartenden Ausnahmen abgefangen. Eine **ArgumentNullException** kann im Beispielprogramm ausgeschlossen werden, weil die Methode **Main()** eine entsprechende Kontrolle vornimmt. Nach einer **OverflowException** liefert `Kon2Int()` den Wert -1 zurück (wie bei einem **int**-Wert außerhalb des Intervalls [0, 170]):

```
static int Kon2Int(ref string instr) {
   int arg;
    try {
        arg = Int32.Parse(instr);
    } catch (OverflowException) {
        return -1;
    } catch (FormatException) when (Double.TryParse(instr, out double d)) {
        arg = (int)d;
        if (arg == d)
            instr = arg.ToString();
        else
            return -2;
    } catch (FormatException) {
        return -3;
    }
    if (arg >= 0 && arg <= 170)
        return arg;
    else
        return -1;
}
```

Im ersten **FormatException**-Handler wird die Filterung durch eine **when**-Klausel demonstriert. Wenn die eingegebene Zeichenfolge zwar nicht als ganze Zahl, aber als Gleitkommazahl interpretierbar ist, dann überprüft `Kon2Int()`, ob sich diese Zahl verlustfrei in einen **int**-Wert wandeln lässt. Gelingt dies, dann wird ...

- der **int**-Wert bei der weiteren Verarbeitung verwendet
- und seine **String**-Repräsentation an den **ref**-Parameter mit der Kandidatenzeichenfolge übergeben, sodass der Aufrufer die korrigierte Zeichenfolge erhält.

Anderenfalls meldet `Kon2Int()` den Wert -2 zurück (z. B. beim Aufruf mit dem Aktualparameter „5,1“).

Zur Beurteilung der Interpretierbarkeit als **double**-Wert dient die statische **Double**-Methode **TryParse()**, die per **bool**-Rückgabewert über die Konvertierbarkeit berichtet.[1] Weil im Beispielprogramm nach einer misslungenen Ganzzahlinterpretation mit erheblicher Wahrscheinlichkeit auch die Interpretation als Gleitkommazahl scheitert, wird die per Ausnahmeobjekt kommunizierende Methode **Double.Parse()** vermieden. Der nach einer erfolgreichen Konvertierung von **TryParse()** als **out**-Parameter gelieferte Wert landet in einer inline deklarierten Variablen (siehe Abschnitt 5.3.1.3.2.2).

Der zweite **FormatException**-Handler kommt zum Einsatz, wenn keine numerische Interpretation der Eingabe möglich ist. In diesem Fall erhält der Aufrufer als Rückgabe den Wert -3.

Die **catch**-Blöcke verwenden das von **Int32.Parse()** geworfene Ausnahmeobjekt *nicht* und verzichten daher auf einen Parameternamen.

Man kann sich fragen, ob `Kon2Int()` nicht auf den potenziellen Ausnahmewerfer **Int32.Parse()** und damit auch auf die Ausnahmebehandlung per **try-catch** - Anweisung verzichten und stattdessen die mit Rückgabewert arbeitende Methode **Int32.TryParse()** verwenden sollte. Nach den Performanzüberlegungen im Abschnitt 13.3 wäre dies eine akzeptable Lösung. Trotzdem erfüllt das Beispiel in seiner jetzigen Form wohl die Aufgabe, die in vielen Situationen außerordentlich wichtige **try-catch** - Ausnahmebehandlung zu demonstrieren.

---

[1] **Double.TryParse()** interpretiert z. B. die Zeichenfolge „5...0“ als 50, weil die Punkte als Gruppentrennzeichen aufgefasst werden. Durch eine Überladung mit **NumberStyles**-Parameter lässt sich die Interpretation beeinflussen.

Je nach Algorithmus kommen als Aufgaben für einen **catch**-Block in Frage (selbstverständlich auch in Kombination):

- Reparatur
  Manchmal ist es möglich, den aufgetretenen Fehler zu beheben oder zu kompensieren (z. B. zu umgehen). In der Methode `Kon2Int()` unternimmt der erste **FormatException** - Handler einen Reparaturversuch.
- Rückabwicklung
  Bereits realisierte und aufgrund der Ausnahme nunmehr unerwünschte Effekte des unterbrochenen **try**-Blocks sollten nach Möglichkeit wieder rückgängig gemacht werden (z. B. durch ein Rollback nach einer gescheiterten Datenbank-Transaktion).
- Ersetzung der Ausnahme durch eine informativere Alternative
  Viele **catch**-Blöcke betätigen sich als Informationsvermittler und werfen selbst eine Ausnahme, um dem Aufrufer einen leichter verständlichen Unfallbericht zu liefern (siehe Abschnitt 13.5).
- Fehlermeldung und Fehlerprotokollierung
  Wenn eine gescheiterte Operation abgebrochen werden muss, sollte der Benutzer eine gut verständliche Fehlermeldung erhalten. Ein Eintrag in eine Logdatei kann den Software-Entwickler oder einen Administrator dabei unterstützen, die Ursache des Fehlers zu finden (siehe Kapitel 16 in Baltes-Götz (2021) zur Dateiausgabe). Nach einer Fehlermeldung oder -protokollierung ist es in der Regel sinnvoll, die abgefangene Ausnahme erneut zu werfen, was durch die folgende **throw**-Anweisung geschehen sollte (siehe Abschnitt 13.5):

  ```
  throw;
  ```

In der **Main()** - Methode des Beispielprogramms ist die 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]);
   switch (argument) {
      case int arg when arg >= 0:
         double fakul = 1.0;
         for (int i = 1; i <= argument; i++)
            fakul *= i;
         Console.WriteLine($"Fakultät von {args[0]}: {fakul}");
         break;
      case -1:
         Console.WriteLine("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
         break;
      case -2:
         Console.WriteLine("Die Eingabe ist keine ganze Zahl: " + args[0]);
         break;
      case -3:
         Console.WriteLine("Die Eingabe ist nicht numerisch interpretierbar: " + args[0]);
         break;
   }
}
```

In der ersten **case**-Klausel wird die im Abschnitt 4.7.2.3.3 beschriebene Deklaration mit Typmuster verwendet.

Ein Programmstart mit dem Befehlszeilenargument „vier“ führt z. B. zur Fehlermeldung:

```
Die Eingabe ist nicht numerisch interpretierbar: vier
```

Die Methode `Kon2Int()` beherrscht nun das Abfangen von Ausnahmen zur Analyse und Behandlung von Problemen, verwendet aber zur Fehlersignalisierung an ihren Aufrufer die traditionelle Technik per Rückgabewert. Diese Mischung ist nicht grundsätzlich fehlerhaft oder ineffektiv, wird sich aber im weiteren Verlauf des Kapitels 13 noch ändern, nachdem wir uns mit dem Werfen von Ausnahmen beschäftigt haben. Wie man an der Koexistenz der Methoden **Parse()** und **TryParse()** in der BCL-Struktur **Int32** sieht, haben beide Verfahren zur Fehlersignalisierung ihre Berechtigung.

Ein Visual Studio - Projekt mit dem aktuellen Entwicklungsstand des Fakultätsberechnungsprogramms ist hier zu finden:

…\BspUeb\Ausnahmebehandlung\catch

13.2.1.2 finally

Der **finally**-Block einer **try**-Anweisung wird unter fast allen Umständen ausgeführt:

- Nach der ungestörten Ausführung des **try**-Blocks
- Nach dem Auftreten einer unbehandelten Ausnahme im **try**-Block
- Nach einer Ausnahmebehandlung in einem **catch**-Block (auch beim Verlassen des **catch**-Blocks durch eine neue Ausnahme)
- Beim Verlassen der **try**-Anweisung durch eine **goto**-Anweisung (siehe Abschnitt 4.7.3.5) im **try**-Block oder in einem **catch**-Block
- Beim Beenden der Methode durch eine **return**-Anweisung im **try**-Block oder in einem **catch**-Block

Nur zwei Ursachen können die Ausführung eines **finally**-Blocks verhindern (Albahari 2022, S. 184):

- Der **try**-Block oder ein **catch**-Block hängt in einer Endlosschleife.
- Im **try**-Block oder in einem **catch**-Block wird die statische Methode **Exit()** der Klasse **Environment** aufgerufen und somit der komplette Prozess terminiert.

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-, Datenbank- und Netzwerkverbindungen. Wir verwenden (dem Kapitel 16 in Baltes-Götz (2021) 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:[1]

[1] Ein Visual Studio - Projekt mit Programm ist hier zu finden:
…\BspUeb\Ausnahmebehandlung\finally\Explizit

```
using System;
using System.IO;

class FinallyDemo {
  static void Mean(String dateiname) {
    StreamReader sr = null;
    try {
      string s;
      int n = 0;
      double summe = 0.0;
      sr = new(dateiname);
      while ((s = sr.ReadLine()) != null) {
        summe += Convert.ToDouble(s);
        n++;
      }
      Console.WriteLine($"Deskriptive Statistiken zur Datei {dateiname}\n");
      Console.WriteLine($"Anzahl:\t{n}");
      Console.WriteLine($"Summe:\t{summe}");
      Console.WriteLine($"Mittel:\t{summe / n}");
    } catch (Exception ex) {
      Console.WriteLine($"Fehler in Mean() beim Lesen der Datei {dateiname}\n{ex}\n");
      throw;
    } finally {
      if (sr != null)
        sr.Dispose();
    }
  }

  static void Main() {
    try {
      Mean(AppDomain.CurrentDomain.BaseDirectory + "daten.txt");
    } catch {
      Console.WriteLine("Fehler bei der Mittelwertsberechnung");
    }
  }
}
```

Das Programm greift über ein **StreamReader**-Objekt lesend auf die Textdatei zu, sodass andere Programme simultan lesen dürfen, aber nicht schreiben.

Das Ergebnis eines erfolgreichen Aufrufs:

```
Deskriptive Statistiken zur Datei C:\Users\ ... \daten.txt

Anzahl: 21
Summe:  106
Mittel: 5,0476190476190474
```

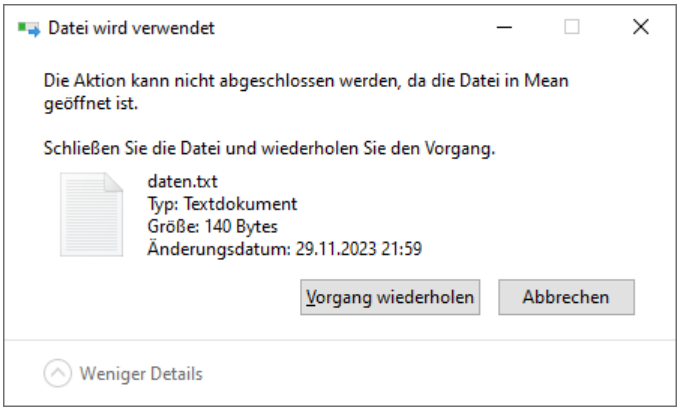
Vom **FileStream**-Konstruktor, der eine vorhandene Datei öffnen soll, sind diverse Ausnahmen zu erwarten, u. a.:

- **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.

Im Beispiel kann *nach* dem erfolgreichen Öffnen der Datei ein Ausnahmefehler auftreten, wenn die Methode **Convert.ToDouble()** auf eine nicht konvertierbare Zeichenfolge trifft.

Eine geöffnete Datei sollte möglichst früh per **Dispose()** - Aufruf geschlossen werden, um andere Programme sowie die Benutzer möglichst wenig zu behindern. Das muss auch für den Ausnahmefall sichergestellt werden, indem das Schließen in einem **finally**-Block erfolgt. Stünde der **Dispose()**

- Aufruf z. B. am Ende des **try**-Blocks, bliebe die Datei nach einem Ausnahmefehler in **Convert.ToDouble()** geöffnet bis zum Programmende, sodass anderen Anwendungen kein schreibender Zugriff möglich wäre. Infolgedessen könnte ein Benutzer die Datei z. B. nicht mit Hilfe des Windows-Explorers löschen:



Wird ein Programm aufgrund einer unbehandelten Ausnahme beendet, dann werden die geöffneten Dateien automatisch geschlossen, und es tritt keine Behinderung auf. Im Beispiel muss die Methode `Mean()` aber damit rechnen, dass eine vorgeordnete Methode (z. B. **Main()**) die in `Mean()` aufgetretenen Ausnahmen behandelt und das Programm fortgesetzt wird, sodass es zu einer Behinderung aufgrund der weiterhin geöffneten Datei kommt.

Der **catch**-Block des Beispielprogramms schreibt eine Fehlermeldung auf die Konsole, z. B.:

```
Fehler in Mean() beim Lesen der Datei C:\Users\ ... \daten.txt
System.FormatException: The input string 'sieben' was not in a correct format.
   at System.Number.ThrowOverflowOrFormatException(ParsingStatus status, ReadOnlySpan`1 value,
                                                   TypeCode type)
   at System.Convert.ToDouble(String value)
   at FinallyDemo.Mean(String dateiname) in C:\Users\ ... \FinallyDemo.cs:line 13
```

Außerdem wirft er per **throw**-Anweisung (siehe Abschnitt 13.5) dieselbe Ausnahme erneut, sodass die Methode `Mean()` nach dem **finally**-Block beendet und der Aufrufer über das Scheitern informiert wird.

Wie im Kapitel 16 von Baltes-Götz (2021) über Dateibearbeitung im Detail erläutert wird, lässt sich durch eine **using**-Anweisung, die strikt von der **using**-Direktive zu unterscheiden ist, das garantierte Schließen einer vom Programm geöffneten Datei einfacher erreichen, wobei der Compiler einen **finally**-Block mit **Dispose()** - Aufruf automatisch erstellt:[1]

```
static void Mean(String dateiname) {
    using StreamReader sr = new(dateiname);
    string s;
    int n = 0;
    double summe = 0.0;
    while ((s = sr.ReadLine()) != null) {
        summe += Convert.ToDouble(s);
        n++;
    }
    Console.WriteLine($"Deskriptive Statistiken zur Datei {dateiname}\n");
    Console.WriteLine($"Anzahl:\t{n}");
    Console.WriteLine($"Summe:\t{summe}");
    Console.WriteLine($"Mittel:\t{summe / n}");
}
```

[1] Im Beispiel wird die (seit C# 8 erlaubte) vereinfachte **using**-Syntax verwendet (ohne Blockanweisung mit eigener Einrückebene). Ein Visual Studio - Projekt mit Programm ist hier zu finden:
…\BspUeb\Ausnahmebehandlung\finally\Implizit

Ausnahmen werden von der **using**-Anweisung *nicht* behandelt, sondern an den Aufrufer weitergeleitet. Außerdem ist es nicht immer erwünscht, dass ein Ressourcen nutzendes Objekt am Ende des Kontexts, in dem es erstellt wurde, verworfen wird.[1] Daher ist die explizite **try-catch-finally** - Anweisung gelegentlich gegenüber der **using**-Anweisung zu bevorzugen.

Eine im **finally**-Block aufgetretene Ausnahme wird an den Vorgänger in der Aufrufverschachtelung übermittelt, wobei es sich auch um eine umgebende **try**-Anweisung handeln kann. Dabei wird eine im **try**-Block oder in einem **catch**-Block geworfene Ausnahme ersetzt und somit die Fehleranalyse erschwert, z. B.:

Quellcode	Ausgabe

`try {`
`  M(-1);`
`} catch (Exception e) {`
`  Console.WriteLine(e.GetType());`
`}`
 	
`static void M(int i) {`
`  try {`
`  if (i < 0)`
`  throw new ArgumentException();`
`  else`
`  Console.WriteLine(i);`
`  } finally {`
`  throw new Exception();`
`  }`
`}`
	`System.Exception`

Daher sollte in einem **finally**-Block keine Ausnahme geworfen werden.

13.2.2 Programmablauf bei der Ausnahmebehandlung

Findet das Laufzeitsystem für eine Ausnahme in der verantwortlichen 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.

Initiator der Ausnahme kann sein:

- die betroffene Methode selbst,
- eine aufgerufene Methode, die die Ausnahme geworfen, aber nicht behandelt hat,
- die CLR (z. B. bei einer Ganzzahldivision durch 0).

13.2.2.1 Beispiel

Das folgende Beispiel demonstriert einige Programmabläufe aufgrund von Ausnahmen, 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 als ganze Zahl und ermittelt den Rest aus der Division der Zahl 10 durch das Argument:[2]

---

[1] https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/using-objects

[2] Ein Visual Studio - Projekt mit Programm ist hier zu finden:
…\BspUeb\Ausnahmebehandlung\Sequenzen

```
using System;
class Sequenzen {
   static int Calc(String instr) {
      int erg = 0;
      try {
         Console.WriteLine("try-Block von Calc()");
         erg = 10 % Convert.ToInt32(instr);
      } catch (FormatException) {
         Console.WriteLine("FormatException-Handler in Calc()");
      } finally {
         Console.WriteLine("finally-Block von Calc()");
      }
      Console.WriteLine("Nach der 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 der try-Anweisung in Main()");
   }
}
```

Die Methode **Main()** lässt die eigentliche Arbeit von der Methode `Calc()` erledigen und bettet den `Calc()` - Aufruf in eine **try-catch-finally** - Anweisung mit **catch**-Block für die **ArithmeticException** ein, die z. B. bei einer Ganzzahldivision durch 0 von der CLR geworfen wird. `Calc()` benutzt die Klassenmethode **Convert.ToInt32()** sowie den Modulo-Operator im **try**-Block einer **try-catch-finally** - Anweisung, wobei nur die potenziell von **Convert.ToInt32()** zu erwartende **FormatException** in einem **catch**-Block 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 den folgenden Ausgaben:

```
try-Block von Main()
try-Block von Calc()
finally-Block von Calc()
Nach der try-Anweisung in Calc()
10 % 8 = 2
finally-Block von Main()
Nach der try-Anweisung in Main()
```

b) Exception in `Calc()`, die dort auch behandelt wird

Wird `Calc()` bei der Ausführung der Anweisung

```
erg = 10 % Convert.ToInt32(instr);
```

mit einer **FormatException** konfrontiert, die z. B. wegen des Befehlszeilenarguments „acht“ von **Convert.ToInt32()** initiiert, aber nicht behandelt wurde, dann kommt der zugehörige **catch**-Block in `Calc()` 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. Wenn eine Methode auf traditionelle Weise per Rückgabewert einen Fehler signalisiert, dann kann es hingegen passieren, dass die warnende Rückgabe ignoriert und der laufende Algorithmus fortgesetzt wird (siehe Abschnitt 13.3).

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 der try-Anweisung in Calc()
10 % acht = 0
finally-Block von Main()
Nach der try-Anweisung in Main()
```

Zu der unsinnigen Ausgabe

```
10 % acht = 0
```

kommt es, weil die **FormatException** in `Calc()` nicht *sinnvoll* behandelt wird. Wenn ein **catch**-Block lediglich eine Fehlermeldung ausgibt und/oder einen Logdateieintrag schreibt, sollte er in der Regel die gefangene Ausnahme unbedingt erneut werfen oder stattdessen eine informativere Ausnahme werfen. Das aktuelle Beispiel soll nur dazu dienen, Programmabläufe bei der Ausnahmebehandlung zu demonstrieren.

c) Exception in `Calc()`, die in Main() behandelt wird

Das Befehlszeilenargument „0“ hat zur Folge, dass im **try**-Block der **try-catch-finally** - Anweisung der Methode `Calc()` von der CLR eine **ArithmeticException** ausgelöst wird. Weil sich in `Calc()` kein passender Exception-Handler befindet, wird die Methode nach der Ausführung des **finally**-Blocks verlassen, um entlang der Aufrufersequenz nach einem geeigneten Handler zu suchen.

Im `Calc()` - 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 der try-Anweisung in Main()
```

d) Exception in Main(), die nirgends behandelt wird

Übergibt der Benutzer kein Befehlszeilenargument, dann tritt in **Main()** beim Zugriff auf `args[0]` eine von der von der CLR initiierte **IndexOutOfRangeException** auf. Weil sich kein zuständiger Handler findet, wird das Programm von der CLR beendet. Zuvor wird noch der **finally**-Block von **Main()** ausgeführt, die Anweisungen hinter der **try**-Anweisung aber nicht:

```
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()
```

13.2.2.2 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.

13.2.3 Unbehandelte Ausnahmen in einer WPF-Anwendung abfangen

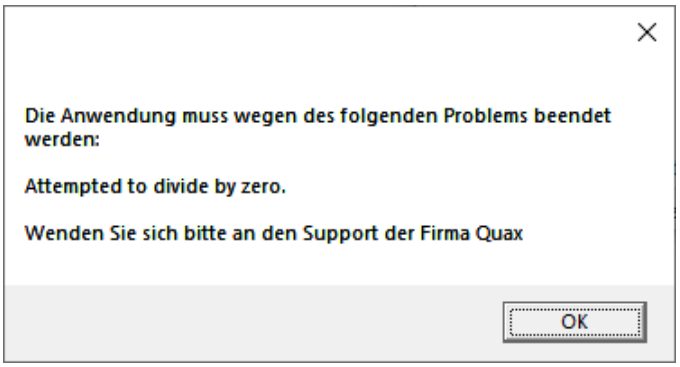
Durch eine Behandlungsmethode zum Ereignis **DispatcherUnhandledException** der Klasse **Application** kann eine WPF-Anwendung auf eine ansonsten unbehandelte Ausnahme reagieren. Wird die folgende Methode

```
private void App_DispatcherUnhandledException(object sender,
               System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e) {
  MessageBox.Show("Die Anwendung muss wegen des folgenden Problems beendet werden:\n\n" +
                  e.Exception.Message +
                  "\n\nWenden Sie sich bitte an den Support der Firma Quax"); ;
  e.Handled = true;
  Application.Current.Shutdown();
}
```

in der Code-Behind - Datei **App.xaml.cs** zur Anwendungsklasse implementiert und in der Datei **App.xaml** beim **Application**-Ereignis **DispatcherUnhandledException** registriert,

```
<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 Info-Dialog:

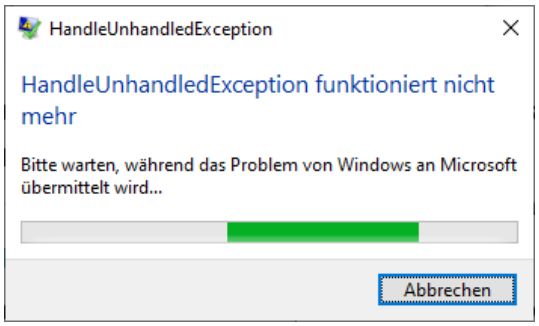


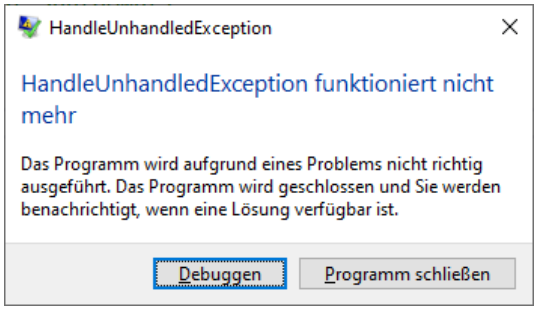
Über die Eigenschaft **Exception** der Klasse **DispatcherUnhandledExceptionEventArgs** ist die ursächliche Ausnahme ansprechbar, sodass die Benutzer informiert werden können. Im Beispiel wird nur die **Message**-Eigenschaft der Ausnahme angezeigt,

```
e.Exception.Message
```

weil der **StackTrace** sehr lang und eher für einen Protokolldateieintrag geeignet ist.

Um die Windows-Standarddialoge für havarierte Anwendungen





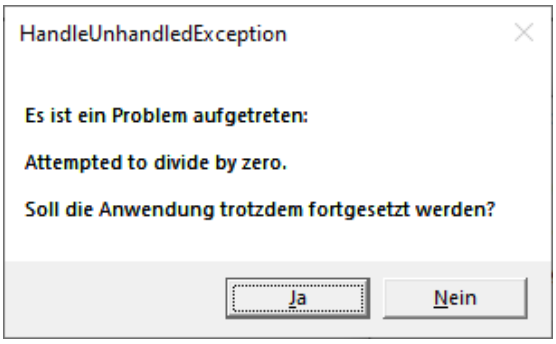
zu verhindern, wird die zugrundeliegende Ausnahme als behandelt erklärt:

```
e.Handled = true;
```

Dann wird die Anwendung mit der **Application**-Methode **Shutdown()** beendet:

```
Application.Current.Shutdown();
```

Wenn es zu verantworten ist, und vom Anwender gewünscht wird,



kann man die Ausnahme als erledigt deklarieren und die Anwendung fortsetzen:

```
private void App_DispatcherUnhandledException(object sender,
                System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e) {
    e.Handled = true;
    if (MessageBox.Show("Es ist ein Problem aufgetreten:\n\n" +
          e.Exception.Message +
          "\n\nSoll die Anwendung trotzdem fortgesetzt werden?",
          "HandleUnhandledException", MessageBoxButton.YesNo) == MessageBoxResult.No)
        Application.Current.Shutdown();
}
```

Ein Visual Studio - Projekt mit dem Beispielprogramm ist hier zu finden:

…\BspUeb\Ausnahmebehandlung\HandleUnhandledException

13.3 Ausnahmeobjekte im Vergleich zur traditionellen Fehlerbehandlung

Die konventionelle Fehlerbehandlung verwendet meist den **Rückgabewert** einer Methode zur Berichterstattung über Probleme bei der Ausführung. Ein Rückgabewert kann …

- ausschließlich zur Fehlermeldung dienen
  Meist wird dann ein ganzzahliger **Returncode** mit dem Datentyp **int** verwendet, wobei die 0 einen erfolgreichen Ablauf signalisiert, während andere Zahlen jeweils für einen bestimmten Fehlertyp stehen. Soll nur zwischen Erfolg und Misserfolg unterschieden werden, dann bietet sich der Rückgabewert **bool** an.
- oder neben den Ergebnissen einer ungestörten Ausführung über spezielle Werte auch Problemfälle signalisieren (siehe Methode `Kon2Int()` des Beispielprogramms im Abschnitt 13.1 bzw. 13.2).

Wenn der Rückgabewert mit der Fehlersignalisierung komplett ausgelastet ist, dann kann zum Ergebnistransport ein **out**- oder **ref**-Parameter verwendet werden (siehe Abschnitt 5.3.1.3.2 zu den Verweisparametern). So verhält sich z. B. die statische Methode **TryParse()** der BCL-Struktur **Int32** (siehe unten). Wenn die Fehlerkommunikation per Rückgabewert ausscheidet, und eine Erweiterung der Parameterliste ebenso unerwünscht ist wie die Kommunikation über Ausnahmeobjekte, dann kommt ein Fehlerstatus in Frage, der z. B. über eine statische Methode abzufragen ist. Allerdings ist das Verfahren umständlich, und die Beachtung ist bei einem Fehlerstatus ebenso wenig garantiert wie bei einem Returncode.

Während bei Methoden die Fehlerkommunikation per Rückgabewert oft realisierbar ist, steht diese Technik bei anderen ausführbaren Klassenmitgliedern (Eigenschaften, Indexern und Operatorüberladungen) nur eingeschränkt zur Verfügung. Beim **get**-Block einer Eigenschaft sollte keine Fehlerkommunikation erforderlich sein. Der fehlende Rückgabewert ist hier irrelevant, und von der technisch möglichen Kommunikation per Ausnahmeobjekt rät Microsoft ab.[1] Beim **set**-Block einer Eigenschaft ist eine Option zur Fehlerkommunikation erforderlich und per Ausnahmeobjekt realisierbar, während kein Rückgabewert vorhanden ist. Bei einem Indexer (vgl. Abschnitt 5.11) kann der **get**-Block auf einen ungeeigneten Index mit der Rückgabe **null** reagieren oder eine Ausnahme werfen (z. B. vom Typ **ArgumentOutOfRangeException**). Dem **set**-Block fehlt die Möglichkeit, Probleme mit dem Index oder mit dem gewünschten neuen Wert per Rückgabe zu artikulieren. Eine Fehlerkommunikation per Ausnahmeobjekt ist hingegen möglich und oft gegenüber der stillschweigenden Leistungsverweigerung zu bevorzugen.

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:

[1] https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/property

```
static void Main() {
    int returncode = M1();
    // Behandlung von potenziellen M1() - Fehlern
    if (returncode == 1) {
        // ...
        Environment.Exit(11);
    }
    ...
    returncode = M2();
    // Behandlung von potenziellen M2() - Fehlern
    if (returncode == 1) {
        // ...
        Environment.Exit(21);
    }
    ...
    returncode = M3();
    // Behandlung von potenziellen M3() - Fehlern
    if (returncode == 1) {
        // ...
        Environment.Exit(31);
    }
    ...
}
```

Wir nehmen nun an, dass die drei Methoden `M1()`, `M2()` und `M3()` durch Ausnahmeobjekte über Fehler informieren. Dabei bleibt im Kernalgorithmus die Übersichtlichkeit erhalten:

```
static void Main() {
   try {
      M1();
      M2();
      M3();
   } catch (ExceptionA a) {
      // Behandlung von Ausnahmen aus der Klasse ExceptionA
   } catch (ExceptionB b) {
      // Behandlung von Ausnahmen aus der Klasse ExceptionB
   } catch (ExceptionC c) {
      // Behandlung von Ausnahmen aus der Klasse ExceptionC
   }
}
```

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.

Das traditionelle Verfahren der Fehlerrückmeldung hat neben dem unübersichtlichen Quellcode noch weitere Nachteile:

- Ungesicherte Beachtung von Rückgabewerten
  Gut gesetzte Rückgabewerte bleiben wirkungslos, wenn sich die Aufrufer nicht darum kümmern.
- Umständliche Weiterleitung von Fehlern
  Wenn ein Fehler nicht an Ort und Stelle behandelt werden soll, dann muss die Fehlerinformation aufwändig entlang der Aufrufersequenz weitergemeldet werden.
- Beschränkung auf Methoden
  Die Rückgabe eines Fehlerindikators ist keine gute Option bei Eigenschaften, Indexern oder überladenen Operatoren.

Gegenüber der konventionellen Fehlerbehandlung hat die Kommunikation über Ausnahmeobjekte u. a. folgende Vorteile:

- Bessere Lesbarkeit des Quellcodes
  Mit Hilfe einer **try**-Anweisung erreicht man eine bessere Trennung zwischen den Anweisungen für den normalen Programmablauf und den diversen Ausnahmebehandlungen, sodass der Quellcode übersichtlich bleibt.
- Garantierte Beachtung von Ausnahmen
  Im Unterschied zu Rückgabewerten können Ausnahmen nicht ignoriert werden. Reagiert ein Programm nicht darauf, wird es vom Laufzeitsystem beendet.
- Automatische Weitermeldung bis zur bestgerüsteten Methode
  Manchmal ist die unmittelbar betroffenen Methode nicht gut gerüstet zur Behandlung einer Ausnahme, z. B. nach dem vergeblichen Öffnen einer Datei. Dann sollte eine „höhere" Methode über das weitere Vorgehen entscheiden und z. B. beim Benutzer eine alternative Datei erfragen.
- Bessere Fehlerinformationen für den Aufrufer
  Über ein **Exception**-Objekt kann der Aufrufer beliebig genau über einen aufgetretenen Fehler informiert werden, was bei einem traditionellen Rückgabewert nicht der Fall ist.
- Uneingeschränkte Verfügbarkeit in allen ausführbaren Klassenmitgliedern

Wie die Realisation des ersten **catch**-Blocks zur **FormatException** im Abschnitt 13.2.1 gezeigt hat (Aufruf von **TryParse()** statt **Parse()**), ist die Fehlermeldung per Ausnahmeobjekt dem traditionellen Rückgabewert nicht grundsätzlich überlegen. Bei der Entscheidung für eine Technik zur Fehlerkommunikation ist u. a. die Wahrscheinlichkeit für das Auftreten des Fehlers relevant:

- Wenn ein **Fehler mit erheblicher Wahrscheinlichkeit** auftritt, dann sollte eine routinemäßige, aktive Kontrolle stattfinden. Daher sollte eine Methode, die einen solchen Fehler 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 aufwändige Kommunikationstechnik. Die Ausnahmebehandlung sollte wegen des hohen Rechenzeitaufwands nicht zum Bestandteil der Programmablaufsteuerung werden.
- Bei **Fehlern mit geringer Wahrscheinlichkeit** haben jedoch häufige, meist überflüssige Kontrollen von Rückgabewerten eine Leistungseinbuße zur Folge. Hier sollte man es besser auf eine Ausnahme ankommen lassen. Die hohen Kosten einer Ausnahme entstehen nur dann, wenn sie tatsächlich geworfen wird.

Eine gute Wahl des Verfahrens zur Fehlerkommunikation ist besonders dann relevant, wenn eine Bibliotheksmethode für einen größeren Nutzerkreis entstehen soll. Microsoft spricht in seinen *Design Guidelines* für diesen Fall eine unmissverständliche Empfehlung aus:[1]

> **DO NOT** return error codes.
> Exceptions are the primary means of reporting errors in frameworks.

Wer eine Methode (z. B. aus einer BCL-Klasse) nutzt, muss das dort realisierte Verfahren zur Fehlerkommunikation kennen und sich darauf einstellen. Manchmal 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:

[1] https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/exception-throwing

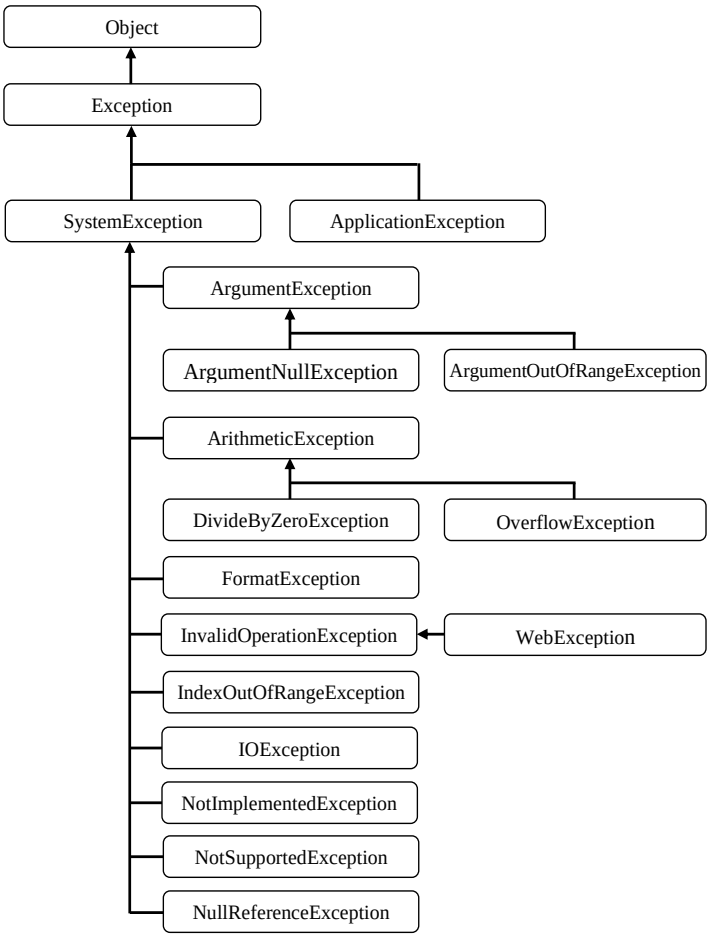
- **public static int Parse(String *s*)**
  Diese Methode transportiert das Ergebnis einer erfolgreichen Wandlung per Rückgabewert und reagiert auf ein ungeeignetes Argument mit einem Ausnahmeobjekt:

Bedingung	**Ausnahmeklasse**
s ist gleich **null**	**ArgumentNullException**
s ist nicht konvertierbar	**FormatException**
Das Konvertierungsergebnis liegt nicht im **int**-Wertebereich	**OverflowException**

- **public static bool TryParse(String *s*, out int *result*)**
  Diese Methode benutzt den Rückgabewert als Fehlerindikator und transportiert das Ergebnis per **out**-Parameter. Scheitert die Konvertierung, erhält der **out**-Parameter den Wert 0. Die Rückgabe **false** zu ignorieren und den Wert 0 wie das Ergebnis einer erfolgreichen Wandlung zu verarbeiten, ist ein gravierender Programmierfehler.

13.4 Ausnahmeklassen in der BCL

Die BCL 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* Ausnahmeklassen durch Wahl einer entsprechend breiten Basisklasse behandelt werden.

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 folgende Aufruf der statischen Methode **ToInt32()** in der BCL-Klasse **Convert**

  ```
  Convert.ToInt32("vier")
  ```

  sorgt z. B. für eine **FormatException** mit der **Message**-Eigenschaft:

  ```
  The input string 'vier' was not in a correct format.
  ```

- **StackTrace**
  Diese **String**-Eigenschaft beschreibt den Aufrufstapel mit den beim Auftreten der Ausnahme aktiven Methoden. Im Abschnitt 13.1 war schon die **StackTrace**-Eigenschaft der **FormatException** zu sehen, die im Beispielprogramm zur Fakultätsberechnung aus einem irregulären Befehlszeilenargument resultiert:

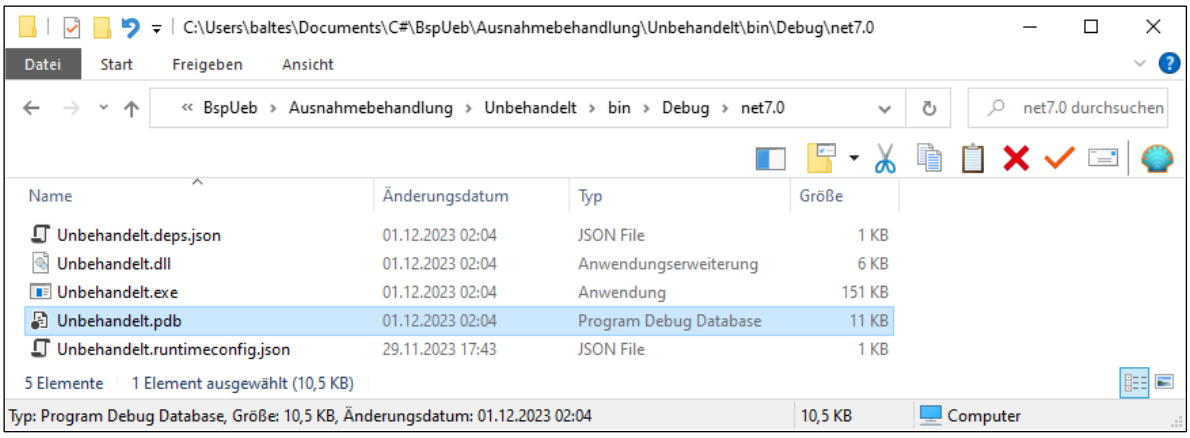
  ```
  Unhandled exception. System.FormatException:
        The input string 'vier' was not in a correct format.
     at System.Number.ThrowOverflowOrFormatException(ParsingStatus status,
                                                     ReadOnlySpan`1 value, TypeCode type)
     at System.Int32.Parse(String s)
     at Fakul.Kon2Int(String instr) in C:\ ... \Fakul.cs:line 5
     at Fakul.Main(String[] args) in C:\ ... \Fakul.cs:line 18
  ```

  Dateinamen und Zeilennummern enthält die Aufrufreihenfolge nur dann, wenn die **Debugsymbole** nicht abgeschaltet wurden. Im Visual Studio 2022 nimmt man die Einstellung nach

  Projekt > Einstellungen > Build > Allgemein

  vor, wobei für ein Programm mit dem **Zielframework** .NET 7.0 voreingestellt ist:

  Die Vorbereitung für die Ausgabe einer genauen Fehlerlokalisation ist an der Anwesenheit einer **PDB-Datei** (*program debug database*) mit der Namenserweiterung **pdb** im Ordner mit dem Assembly zu erkennen, z. B.:



  Durch Optimierungsmaßnahmen des Compilers (z. B. Inlining) kann die Aufrufersequenz anders 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 13.6). Um dem Aufrufer auch die ursprüngliche Ausnahme zur Verfügung zu stellen, kann man ihre Adresse in der Eigenschaft **InnerException** mitliefern (siehe Abschnitt 13.6).
- **Data**
  Über die **Data**-Eigenschaft mit dem Interface-Typ **IDictionary** (siehe Abschnitt 11.5) lassen sich Zusatzinformationen in einer Kollektion mit beliebig vielen (Schlüssel-Wert) - Paaren unterbringen. Man fügt die Zusatzinformationen mit der **IDictionary**-Methode **Add()** ein:

  public void Add (object *key***, object** *value***)**

  Im folgenden Beispiel werden drei Einträge in die **Data**-Liste eines Ausnahmeobjekts aufgenommen:

```
if (arg < 0 || arg > 170) {
   var 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
- die **Message**-Zeichenfolge (die beim Erzeugen der Ausnahme formulierte Fehlermeldung)
- die **StackTrace**-Zeichenfolge (die Aufrufreihenfolge)

Beispiel:

```
System.FormatException: The input string 'vier' was not in a correct format.
   at System.Number.ThrowOverflowOrFormatException(ParsingStatus status,
                                                   ReadOnlySpan`1 value, TypeCode type)
   at System.Int32.Parse(String s)
   at Fakul.Kon2Int(String& instr) in C:\Users\ ... \Fakul.cs:line 7
```

Objekte aus den folgenden BCL-Ausnahmeklassen (alle im Namensraum **System**) werden in C# - Programmen oft behandelt oder geworfen:

- **ArgumentException**
  Ein Aktualparameter entspricht nicht den Anforderungen einer Methode. In der Klasse **Dictionary<K, V>**, die (Schlüssel-Wert) – Paare verwaltet und dabei Dubletten verhindert, wirft die Methode **Add(K** *key*, **V** *value***)** eine **ArgumentException**, wenn der Schlüssel eines neu aufzunehmenden Paares bereits in der Kollektion vorhanden ist. Die beiden folgenden **ArgumentException**-Ableitungen werden oft benötigt:
  - **ArgumentNullException**
    Ein typisches Parameterproblem besteht darin, dass bei einem Referenzdatentyp der ungeeignete Wert **null** auftritt. Für diesem Fall enthält die BCL die Klasse **ArgumentNullException**. Weil oft für einen Parameter der **null**-Test durchzuführen und nötigenfalls eine **ArgumentNullException** zu werfen ist, erledigt die statische Methode **ThrowIfNull()** der Klasse **ArgumentNullException** diese Arbeiten, z. B.:
    `ArgumentNullException.ThrowIfNull(instr);`
  - **ArgumentOutOfRangeException**
    Wenn z. B. für einen **int**-Parameter einer Methode der minimale Wert 18 vorgeschrieben ist, dann kann die Methode bei einem Aufruf mit einem zu kleinen Wert mit einer Ausnahme vom Typ **ArgumentOutOfRangeException** reagieren.

- **FormatException**
  Ein Aktualparameter (z. B. eine zu konvertierende Zeichenfolge oder eine Formatierungszeichenfolge) hat einen ungültigen Aufbau.
- **InvalidOperationException**
  Eine Methode kann unabhängig von den Aktualparameterwerten nicht ausgeführt werden (z. B. Änderung einer Kollektion während einer Iteration, Zugriff auf ein UI-Element durch einen Thread, der das Element nicht erstellt hat.)
- **IndexOutOfRangeException**
  Diese Ausnahme wirft die CLR bei einem versuchten Zugriff auf ein nicht vorhandenes Array-Element.
- **NotImplementedException**
  In einer Methode ist die angeforderte Operation noch nicht implementiert.
- **NotSupportedException**
  Von einer Methode oder Eigenschaft wird eine angeforderte Operation generell nicht unterstützt. Die **Collection<T>** - Methode **Insert()** wirft z. B. eine **NotSupportedException**, wenn die angesprochene Kollektion nur Lesezugriffe erlaubt.
- **NullReferenceException**
  Diese Ausnahme wird von der CLR geworfen beim Versuch, ein nicht existentes Objekt anzusprechen.

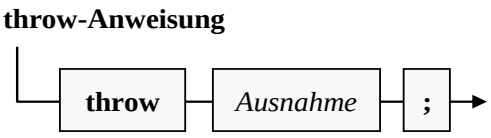
13.5 Ausnahmen werfen (throw)

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 der folgenden Variante der Methode `Kon2Int()` aus dem Standardbeispiel von Kapitel 13 wird ein Ausnahmeobjekt aus der BCL-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(nameof(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 (aus der Klasse **System.Exception** oder aus einer abgeleiteten Klasse):



Wie im obigen 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 der Name und der Wert des irregulären Arguments sowie eine Fehlermeldung anzugeben sind. Den Namen des **Parse()** - Parameters mit der zu wandelnden Zeichenfolge liefert der **nameof**-Operator, den wir hier im Vorgriff auf den Abschnitt 14.4 verwenden. Aus dem Aufruf

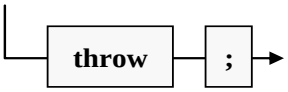
```
try {
   argument = Kon2Int(args[0]);
} catch (Exception e) {
   Console.WriteLine(e.Message);
    . . .
}
```

mit dem Argument 188 resultiert die Meldung:

```
Argument ausserhalb [0, 170] (Parameter 'instr')
Actual value was 188.
```

In einem **catch**-Block darf das Schlüsselwort **throw** auch *ohne* Ausnahmeobjekt-Referenz stehen:

throw-Anweisung ohne Ausnahme-Objekt



In diesem Fall wird die gerade behandelte Ausnahme erneut geworfen. Dieses Verhalten kommt z. B. in Frage, wenn ...

- lediglich eine Meldung ausgegeben oder ein Protokolleintrag geschrieben werden soll.
- sich nach einem Lösungsversuch herausstellt, dass der **catch**-Block das Problem nicht aus der Welt schaffen kann, sodass die Ausnahme an den Vorgänger in der Aufrufverschachtelung übergeben werden muss, wobei es sich auch um eine umgebende **try**-Anweisung handeln kann.

Die im **catch**-Block erlaubte reduzierte Form der **throw**-Anweisung hat den Vorteil, dass die **StackTrace**-Information im Ausnahme-Objekt erhalten bleibt. Z. B. wird die folgende Aufruferse-quenz

```
at System.Number.ThrowOverflowOrFormatException(ParsingStatus status,
                                                 ReadOnlySpan`1 value, TypeCode type)
at System.Int32.Parse(String s)
at Fakul.Kon2Int(String& instr) in C:\Users\ ... \Fakul.cs:line 9
at Fakul.Main(String[] args) in C:\Users\ ... \Fakul.cs:line 34
```

durch die explizite Verwendung der übergebenen **Exception**-Referenz in der **throw**-Anweisung

```
catch (FormatException fo) {
     . . .
    throw fo;
}
```

reduziert zu:

```
at Fakul.Kon2Int(String& instr) in C:\Users\ ... \Fakul.cs:line 20
at Fakul.Main(String[] args) in C:\Users\ ... \Fakul.cs:line 34
```

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:

- Meist wird eine *spezifischere* Ausnahmeklasse verwendet, die eine genauere Fehlerbeschreibung ermöglicht. Damit die ursprüngliche Ausnahme als Anlage beigefügt werden kann, bieten die BCL-Ausnahmeklassen einen Konstruktor mit einem Parameter vom Typ **Exception**. Ein Handler kann ggf. über die Eigenschaft **InnerException** auf die Anlage zugreifen.
- Umgekehrt können Sicherheitsüberlegungen zur Wahl einer *allgemeineren* Ausnahmeklasse führen, um wenig technische Details gegenüber potenziellen Angreifern offenzulegen (Albahari 2022, S. 187).

In der aktuellen `Kon2Int()` - Variante wird auf die Behandlung der von **Int32.Parse()** potenziell zu erwartenden Ausnahmen (**ArgumentNullException**, **FormatException**, **OverflowException**) verzichtet. Folglich hat ein `Kon2Int()` - Nutzer insgesamt mit den folgenden Ausnahmeklassen zu rechnen:

- **ArgumentNullException**
- **FormatException**
- **OverflowException**
- **ArgumentOutOfRangeException**

In der `Kon2Int()` - Dokumentation muss darüber informiert werden.

Während die aktuelle `Kon2Int()` - Version durch das Werfen einer Ausnahme aus einer von vier verschiedenen Klassen über Fehler informiert, verzichtet die im Abschnitt 13.2.1.1 vorgestellte Version auf das Werfen von Ausnahmen und kommuniziert Fehler traditionell über einen Returncode mit unterschiedlichen Werten für die Fehlerkategorien. Selbstverständlich ist auch eine Mischung aus der modernen und der traditionellen Fehlersignalisierung möglich. Im Abschnitt 13.3 finden sich Hinweise zur Wahl des Verfahrens.

Seit der Version 7.0 unterstützt C# neben der bisher beschriebenen **throw**-Anweisung auch den **throw**-Ausdruck. Das ermöglicht das Werfen von Ausnahmen an Stellen, die syntaktisch einen Ausdruck verlangen, z. B. im zweiten oder dritten Operanden eines Konditionaloperators:

Quellcode	Ausgabe

`Console.WriteLine(Log2(8));`
`Console.WriteLine(Log2(0));`
 	
`static double Log2(double arg) {`
`  return arg > 0 ?`
`  Math.Log(arg) / Math.Log(2) :`
`  throw new ArgumentException("arg must be > 0");`
`}`
	`3`
	`Unhandled exception.`
	`System.ArgumentException:`
	`arg must be > 0`

Weitere Einsatzorte für einen **throw**-Ausdruck sind:

- Ergebnisausdruck eines **switch**-Verzweigungsarms (siehe Abschnitt 15.2), z. B.:
  ```
  null => throw new ArgumentException(message: "No person found")
  ```
- Null-Koaleszenzoperator (siehe Abschnitt 8.3), z. B.:
  ```
  static void Main(string[] args) {
    Console.WriteLine("Argument: " +
      args[0] ?? throw new ArgumentNullException("Argument missing"));
  }
  ```
- Methodendefinition mit Lambda-Operator (siehe Abschnitt 5.8.1), z. B.:
  ```
  static string GetPassword(string name) => throw new NotImplementedException();
  ```

Ein Visual Studio - Projekt mit dem aktuellen Entwicklungsstand des Fakultätsberechnungsprogramms ist hier zu finden:

…\BspUeb\Ausnahmebehandlung\throw

13.6 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 nicht auf die BCL-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, z. B.:
  `public class BadFaculArgException : Exception { ... }`
- Der Klassenname sollte mit dem Wort *Exception* enden.

[1] https://learn.microsoft.com/en-us/dotnet/standard/exceptions/how-to-create-user-defined-exceptions
Es ist nicht sicher, ob die Empfehlung der Basisklasse **Exception** im *strikten* Sinn gemeint ist, oder ob auch eine **Exception**-Ableitung (im Beispiel: **ArgumentException**) als Basisklasse erlaubt ist. Im Beispiel hat die Verwendung der Basisklasse **Exception** keinen erkennbaren Nachteil, sodass die Empfehlung wörtlich eingehalten wird.

- Die folgenden Konstruktoren sollten mit **public** - Verfügbarkeit implementiert werden:
  - Ein parameterfreier Konstruktor, z. B.:
    ```
    public BadFaculArgException() {
    }
    ```
  - Ein Konstruktor mit einem **string**-Parameter für die Fehlermeldung, z. B.:
    ```
    public BadFaculArgException(string message) : base(message) {
    }
    ```
  - 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, z. B.:
    ```
    public BadFaculArgException(string message, Exception innerException)
      : base(message, innerException) {
    }
    ```

Beim parameterlosen `BadFaculArgException`-Konstruktor beschränken wir uns auf den (impliziten) Aufruf des parameterlosen Basisklassenkonstruktors. Bei den restlichen Konstruktoren rufen wir explizit eine passende Überladung des Basisklassenkonstruktors auf, um die Eigenschaften **Message** und **InnerException** zu initialisieren.

Durch ein Objekt der selbstdefinierten Ausnahmeklasse `BadFaculArgException` kann ausführlich über Probleme mit dem Argument 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 jüngste `Kon2Int()` - Version kümmert sich um die von **ToInt32.Parse()** zu erwartenden Ausnahmen und wirft bei allen Fehlerursachen eine spezielle `BadFaculArgException`:[1]

[1] Eine **ArgumentNullException** ist nicht möglich, weil `Kon2Int()`einen **ref**-Formalparameter verwendet, sodass der Compiler einen Aufruf mit dem Aktualparameterwert **null** verhindert.

```
static int Kon2Int(ref String instr) {
   int arg;
   try {
      arg = Convert.ToInt32(instr);
   } catch (FormatException e) when (Double.TryParse(instr, out double d)) {
      arg = (int)d;
      if (arg == d)
         instr = arg.ToString();
      else
         throw new BadFaculArgException("Fehler beim Konvertieren", instr, 2, -1, e);
   } catch (FormatException e) {
      throw new BadFaculArgException("Fehler beim Konvertieren", instr, 2, -1, e);
   } catch (OverflowException e) {
      throw new BadFaculArgException("Ganzzahl-Überlauf", instr, 3, -1, e);
   }

   if (arg < 0 || arg > 170)
      throw new BadFaculArgException("Wert außerhalb [0, 170]", instr, 4, arg);
   else
      return arg;
}
```

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]);
      double fakul = 1.0;
      for (int i = 1; i <= argument; i++)
         fakul *= i;
      Console.WriteLine($"Fakultät von {args[0]}: {fakul}");
      Console.Read();
   } catch (BadFakulArgException e) {
      Console.WriteLine($"Message:\t{e.Message}");
      Console.WriteLine($"Fehlertyp:\t{e.Type}\nEingabe:\t{e.Input} ");
      Console.WriteLine($"Wert:\t\t{e.Value}");
      if (e.InnerException != null)
         Console.WriteLine($"Orig. Message:\t{e.InnerException.Message}");
      Console.Read();
      Environment.Exit(1);
   }
}
```

Bei einem Programmstart mit dem Befehlszeilenargument „vier“ resultiert z. B. die Ausgabe:

```
Message:        Fehler beim Konvertieren
Fehlertyp:      2
Eingabe:        vier
Wert:           -1
Orig. Message:  The input string 'vier' was not in a correct format.
```

Eine eigene Ausnahmeklasse passt gut zur Strategie, Probleme aus diversen Teilschritten einer Aufgabenbearbeitung an einer Stelle zusammenzuführen, wo über das weitere Vorgehen nach einem Scheitern entschieden werden soll. An dieser Entscheidungsstelle muss dann nur *ein* Ausnahmeobjekt behandelt werden, das genügend Informationen über die Art des Problems enthält.

Ein Visual Studio - Projekt mit dem aktuellen Entwicklungsstand des Fakultätsberechnungsprogramms ist hier zu finden:

…\BspUeb\Ausnahmebehandlung\BadFaculArgException\Eigene Felder und Eigenschaften

Zum Transport von Zusatzinformationen benötigt eine (selbst erstellte) Ausnahmeklasse nicht unbedingt zusätzliche Felder bzw. Eigenschaften. Jedes Ausnahmeobjekt enthält eine über die **Exception**-Eigenschaft **Data** ansprechbare Kollektion vom Interface-Typ **IDictionary** (siehe Abschnitt 11.5), die in Elementen vom Typ **DictionaryEntry** beliebig viele (Schlüssel-Wert) - Paare mit Informationen über das Ausnahmeobjekt aufnehmen kann (vgl. Abschnitt 13.4). Bei der Klasse `BadFaculArgException` könnten wir uns auf die folgende Standarddefinition beschränken:[1]

```
using System;

public class BadFaculArgException : Exception {
   public BadFaculArgException() {
   }
   public BadFaculArgException(String message) : base(message) {
   }
   public BadFaculArgException(String message, Exception innerException)
      : base(message, innerException) {
   }
}
```

In die **Data**-Kollektion eines Ausnahmeobjekts lassen sich (Schlüssel-Wert) - Paar mit den benötigten Zusatzinformationen per **Add()** aufnehmen, z. B.:

```
static int Kon2Int(ref string instr) {
  int arg;
  try {
    arg = Convert.ToInt32(instr);
  } catch (FormatException e) when (Double.TryParse(instr, out double d)) {
    arg = (int)d;
    if (arg == d)
      instr = arg.ToString();
    else {
      var bfa = new BadFaculArgException("Fehler beim Konvertieren", e);
      bfa.Data.Add("Input", instr);
      bfa.Data.Add("Type", 2);
      bfa.Data.Add("Value", -1);
      throw bfa;
    }
  }
  . . .
}
```

Das macht aber mehr Aufwand als die Verwendung der oben beschriebenen Ausnahmeklasse mit zusätzlichen Eigenschaften und passenden Konstruktoren, z. B.:

---

[1] Diese eigene Klasse hätte im Vergleich zur BCL-Klasse **ArgumentException** nur noch den Vorteil, dass informierte Entwickler vom Klassennamen auf die Anwesenheit einer gefüllten **Data**-Kollektion schließen können.

```
static int Kon2Int(ref String instr) {
  int arg;
  try {
    arg = Convert.ToInt32(instr);
  } catch (FormatException e) when (Double.TryParse(instr, out double d)) {
    arg = (int)d;
    if (arg == d)
      instr = arg.ToString();
    else
      throw new BadFaculArgException("Fehler beim Konvertieren", instr, 2, -1, e);
  }
  . . .
}
```

Trotzdem ist die **Data**-Liste eine nützliche Option:

- Sie kann ohne Nachteile für vorhandene, eine Ausnahme nutzende Klassen erweitert werden.
- Die Möglichkeit, eine Ausnahmeklasse aus der BCL mit individuellen Zusatzinformationen auszustatten, macht in vielen Fällen die Definition einer eigenen Ausnahmeklasse überflüssig.

Ein **catch**-Block kann auf die **DictionaryEntry**-Elemente in der **Data**-Kollektion eines Ausnahmeobjekts per Indexer

```
Console.WriteLine($"Wert = {e.Data["Value"]}");
```

oder mit Hilfe der **DictionaryEntry**-Eigenschaften **Key** und **Value**

```
foreach (DictionaryEntry de in e.Data)
   Console.WriteLine($"{de.Key}\t:\t{de.Value}");
```

zugreifen.

13.7 Übungsaufgaben zum Kapitel 13

1) Erstellen Sie ein Syntaxdiagramm zur **try** - Anweisung (vgl. Abschnitt 13.2.1).

2) Im Beispielprogramm zur Demonstration von möglichen Sequenzen bei der Ausnahmebehandlung (siehe Abschnitt 13.2.2) verzichtet die Methode `Calc()` darauf, die potenziell von der Methode **Convert.ToInt32()** zu erwartende **OverflowException** abzufangen. 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 weitergerechnet wird, und erst am Ende eines längeren Rechenwegs das Ergebnis **NaN** auftaucht. Im folgenden Beispiel wird eine Methode namens `Log2()` zur Berechnung des

dualen Logarithmus[1] verwendet, welche auf die BCL-Methode **Math.Log()** zurückgreift und daher bei ungeeigneten Argumenten (≤ 0) als Rückgabewert **Double.NaN** liefert:[2]

Quellcode	Ausgabe

`using System;`
 	
`class Prog {`
`  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);`
`  }`
`}`
	`NaN`

Erstellen Sie eine Version, die bei ungeeigneten Argumenten eine **ArgumentOutOfRange-Exception** wirft.

4) Erstellen Sie eine Variante der im Abschnitt 8.2.1 vorgestellten generischen Stapelverwaltungsklasse mit einem parametrisierten Konstruktor, der auf eine beantragte nicht-positive Größe mit einer **InvalidOperationException** reagiert, statt die voreingestellte Größe zu verwenden.

---

[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).

[2] Ein Visual Studio - Projekt mit dem Beispiel befindet sich im Ordner:
…\BspUeb\Ausnahmebehandlung\DuaLog\NaN

14 Attribute und Reflexion

An Typen (Klassen, Strukturen, Schnittstellen, usw.), Member (Methoden, Eigenschaften, usw.), Formalparameter, Typformalparameter und Rückgabewerte sowie an Assemblies kann man *Attribute* anheften, um Metainformationen bereitzustellen, 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 im Quellcode vergebenen Attributobjekte als Metadaten im erzeugten Assembly. Bei einfachen Attributen besteht die Information über den Träger in der schlichten An- bzw. Abwesenheit des Attributs. Jedoch lassen sich in einem Attributobjekt auch Detailinformationen unterbringen, die über öffentliche Felder oder Eigenschaften verfügbar sind.

Nur beachtete Metadaten entfalten eine Wirkung, und im Fall von Attributen sind die wesentlichen Rezipienten:

- Compiler
  Etliche Attribute veranlassen den Compiler dazu, Bedingungen zu prüfen und ggf. Warnungen auszusprechen. Mit dem **CLSCompliantAttribute** fordert man den Compiler z. B. dazu auf, die CLS-Kompatibilität einer Klasse oder eines Assemblies zu überwachen (vgl. Abschnitt 2.3.3). Im Abschnitt 15.1.4 werden Attribute vorgestellt, mit denen die Entwickler den Compiler bei der Nullzustandsanalyse unterstützen (z. B. **MemberNotNull**, **NotNullWhen**).
- Entwicklungsumgebung
  Ein Beispiel ist das **DebuggerStepThroughAttribute** für Methoden, das den Debugger im Visual Studio (vgl. Abschnitt 5.3.4) davon abhält, in der Methode befindliche Haltepunkte zu beachten.
- Programm
  Ein Beispiel ist das **SerializableAttribute**, mit dem für die Instanzen eines Typs das sogenannte *binäre Serialisieren*, d. h. das Speichern in einen Datenstrom (z. B. in eine Datei) erlaubt wird (siehe Abschnitt 16.5.1 in Baltes-Götz (2021)). Fehlt dieses Attribut für eine Instanz, die in einen Serialisierungsstrom gerät (z. B. als direktes oder indirektes Mitglied eines anderen Typs), dann reagiert die Methode **Serialize()** der Klasse **BinaryFormatter** mit einem Ausnahmefehler.[1] In komplexen objektorientierten Softwaresystemen (Frameworks) spielt die als **Reflexion** (engl.: *reflection*) bezeichnete Ermittlung von Informationen über Typen, Methoden usw. zur Laufzeit eine große Rolle. Dabei leisten Attribute einen wichtigen Beitrag.

Wir lernen mit den **System.Attribute** – Ableitungen eine weitere Option zur *deklarativen Programmierung* kennen. Sie ergänzen die im C# - Sprachumfang verankerten Modifikatoren (z. B. **public**, **abstract** , **sealed**) für Typen, Methoden etc. und bieten dabei eine enorme Flexibilität. Auch die Modifikatoren gehören zu den Attributen in einem erweiterten Sinn und führen ebenfalls zu (besonders effizient abgelegten) Metadaten im erzeugten Assembly. Die **System.Attribute** - Ableitungen werden in Abgrenzung von den deklarativen Bestandteilen der Programmiersprache C# auch als *benutzerdefinierte Attribute* bezeichnet (engl.: *custom attributes*).[2] Wichtige Methoden zur Auswertung von benutzerdefinierten Attributen (siehe Abschnitt 14.2) führen das Wort *Custom* im Namen (z. B. **GetCustomAttribute()**).Während die zum Sprachumfang gehörenden Attribute (z. B. die Modifikatoren) über viele Jahre kaum ergänzt wurden, können **System.Attribute** - Ableitungen

---

[1] Das binäre Serialisieren ist performant und angenehm einfach handhabbar. Leider bestehen gravierende Sicherheitsprobleme, die das Deserialisieren von Daten aus unsicheren Quellen betreffen (siehe Abschnitt 16.5.1.1 in Baltes-Götz (2021)).

[2] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/attributes/creating-custom-attributes

von den Software-Entwicklern jederzeit neu erstellt werden. Das Manuskript orientiert sich bei der Begriffsverwendung an der C# - Sprachspezifikation (ECMA 2022, Kap. 21), wo nur die von **System.Attribute** abstammenden Klassen als *Attribute* bezeichnet werden.

Von *deklarativer Programmierung* sprachen wir auch bei der Gestaltung einer WPF-Bedienoberfläche per XAML-Code (siehe Abschnitt 5.13.2). Die aktuell behandelten, in den C# - Code zu platzierenden Attribute für Klassen, Methoden, Assemblies etc. sind streng von den Attributen der XAML-Elemente zu unterscheiden. Man kann aber durchaus im gemeinsamen deklarativen Ansatz der beiden Optionen zur Entwicklung von .NET - Software eine Ursache für die übereinstimmende Bezeichnung sehen.

Wir behandeln zunächst die Vergabe und Auswertung von BCL-Attributen, beschäftigen uns später aber auch mit der Definition von *eigenen* Attributen.

Im Zusammenhang mit den Attributen für Assemblies ist zu erfahren, wie man Versions- und andere Produktinformationen vereinbart, die u.a. zur Information der Kunden dienen.

14.1 Attribute vergeben

Sollen z. B. die Benutzer einer Klassenbibliothek dazu gebracht werden, einer neuen Klasse den Vorzug gegenüber einer alten zu geben, dann setzt man der alten Klassendefinition zwischen eckigen Klammern das Attribut **Obsolete** voran, sodass der Compiler bei Verwendung dieser Klasse eine Warnung ausgibt, z. B.:[1]

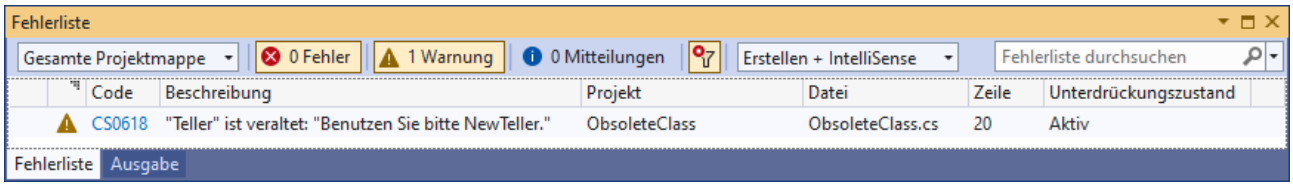
```
using System;

[Obsolete("Benutzen Sie bitte NewTeller.")]
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();
        N
        Consol
    }
}
```

```
[veraltet] class Teller

CS0618: "Teller" ist veraltet: "Benutzen Sie bitte NewTeller."

Mögliche Korrekturen anzeigen (Alt+Eingabe oder Strg+.)
```

Im Editor der Entwicklungsumgebung wird der unerwünschte Zugriff auf die obsolete Klasse unterschlängelt und bei passender Mauszeigerpositionierung zusätzlich kommentiert. In der **Fehlerliste** erscheint eine **Warnung**:

---

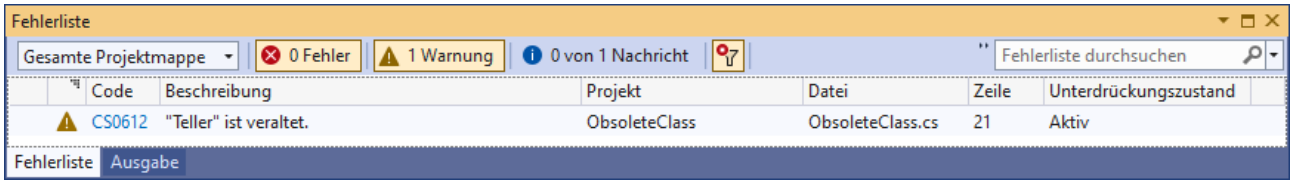
[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\Attribute\ObsoleteClass



Wenn der Klassenbibliotheksdesigner bzw. -renovierer den parameterfreien Konstruktor verwendet,

```
[Obsolete]
```

dann fällt die Compiler-Warnung etwas dürftiger aus, und die **Message**-Eigenschaft des Objekts ist leer:



Je nach verwendeter Konstruktorüberladung sind Aktualparameter anzugeben, wobei eine leere Aktualparameterliste weggelassen werden darf. Als weitere syntaktische Besonderheit darf bei der Vergabe eines Attributs der Namensteil *Attribute* im Klassennamen entfallen. Im Beispiel kommt die Klasse **ObsoleteAttribute** (aus dem Namensraum **System**) zum Einsatz. Weitere Informationen zu Attributparametern folgen im Abschnitt 14.6.

Eine Klasse (oder ein anderer Träger) kann *mehrere* Attribute erhalten, die hintereinander anzugeben sind:

- entweder in separaten Klammerpaaren, z. B.
  ```
  [Obsolete] [Serializable]
  public class Teller { ... }
  ```
- oder durch Kommata getrennt in einem gemeinsamen Klammerpaar, z. B.:
  ```
  [Obsolete, Serializable]
  public class Teller { }
  ```

Bei der Vergabe eines Attributs entsteht ein Objekt, das im entstehenden Assembly die Metadaten des Typs ergänzt.

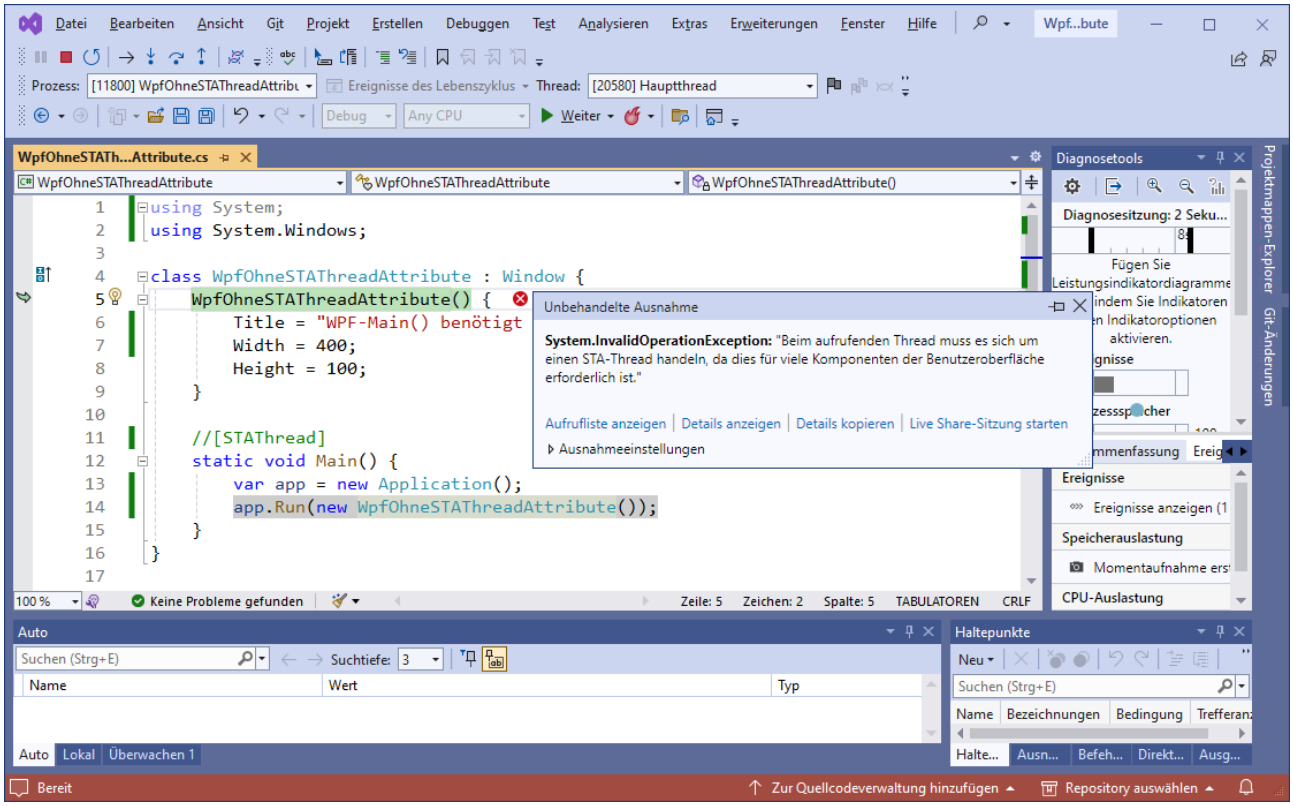
Neben Klassen können auch andere Programmbestandteile mit Attributen versehen werden, z. B. Methoden:[1]

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\Attribute\ObsoleteMethod

```
using System;

class Teller {
    [Obsolete("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();
    }
}
```

```
[veraltet] void Teller.Tell()

CS0618: "Teller.Tell()" ist veraltet: "Benutzen Sie bitte TellEx()."

Mögliche Korrekturen anzeigen (Alt+Eingabe oder Strg+.)
```

Wie Sie aus dem Abschnitt 12.2.1 wissen, muss in einer WPF-Anwendung die Methode **Main()** generell das **STAThreadAttribute** erhalten. Anderenfalls scheitert die Ausführung des Fensterklassenkonstruktors an einer **InvalidOperationException**, z. B.:[1]



Mit dem **STAThreadAttribute** wird signalisiert, dass bei der Kooperation mit dem Component Object Model (COM), der noch weit verbreiteten Windows-Komponententechnologie, das COM-Threading-Modell STA (*Singlethread-Apartment*) zum Einsatz kommen soll.

---

1 Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\Attribute\WpfOhneSTAThreadAttribute

Seit C# 7.3 besteht die Möglichkeit, auch das private *backing field* einer automatisch implementierten Eigenschaft (siehe Abschnitt 5.5.2) mit Attributen zu dekorieren.[1] Dazu werden der Attributbezeichnung das Schlüsselwort **field** und ein Doppelpunkt vorangestellt. Im folgenden Beispiel geschieht dies bei einer Eigenschaft zur Erfassung der Stimmung einer Person, um die Speicherung oder Übertragung der Eigenschaftsausprägung im Rahmen einer binären Serialisierung (siehe Abschnitt 16.5.2 in Baltes-Götz (2021)) zu verhindern:

```
public class Person {
    . . .
   [field: NonSerialized]
   public int Mood { get; set; }
    . . .
}
```

Im weiteren Verlauf von Kapitel 14 wird noch klarer, dass man bei der Vergabe von Attributen in der Regel nicht nur den Compiler informiert, sondern auch die Programmausführung beeinflusst, sofern beteiligte Methoden die Existenz und den Zustand von Attributen per Reflexion ermitteln und in ihrem Verhalten berücksichtigen.

14.2 Attribute per Reflexion auswerten

Die .NET - Plattform bietet leistungsfähige Reflexionstechniken, um die Attributausstattung von Programmbestandteilen zur Laufzeit zu analysieren. Mit der statischen Methode **IsDefined()** der Klasse **System.Attribute** lässt sich feststellen, ob ein bestimmtes Attribut vorhanden ist. Das folgende Programm prüft für eine Klasse und für eine Methode, ob das **ObsoleteAttribute** angeheftet ist:[2]

```
using System;
using System.Reflection;

[Obsolete("Benutzen Sie bitte NewTeller.")]
public class Teller {
    [Obsolete("Benutzen Sie bitte TellEx().")]
    public void Tell() {
        Console.WriteLine("Hallo!");
    }
    public void TellEx() {
        Console.WriteLine("Hallo, Wilhelm!");
    }
}
```

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-7.3/auto-prop-field-attrs

[2] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\Attribute\IsDefined

```
class Prog {
    static void Main() {
        Type typeTeller = typeof(Teller);
        Type typeObsolete = typeof(ObsoleteAttribute);

        if (Attribute.IsDefined(typeTeller, typeObsolete))
            Console.WriteLine("Die Klasse {0}\n ist obsolet", typeTeller.Name);

        MemberInfo[] mi = typeTeller.FindMembers(MemberTypes.Method,
            BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public,
            Type.FilterName, "T*");

        foreach (var m in mi)
            if (Attribute.IsDefined(m, typeObsolete))
                Console.WriteLine("\nDie Methode {0}\n ist obsolet", m.Name);
    }
}
```

Beide Prüflinge werden als obsolet erkannt:

```
Die Klasse 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 **MemberInfo** stammt u.a. die Klasse **Type** ab, die einen Datentyp (Klasse, Struktur, Schnittstelle etc.) repräsentiert. Im Beispiel wird **IsDefined()** zweimal aufgerufen:

- Im ersten Aufruf ist der potenzielle Attributträger eine Klasse, deren **Type**-Objekt als erster **IsDefined()** – Aktualparameter angegeben wird:
  ```
  Type typeTeller = typeof(Teller);
  Attribute.IsDefined(typeTeller, typeObsolete)
  ```
- Im zweiten Aufruf wird die Anwesenheit eines Methodenattributs untersucht:
  ```
  Attribute.IsDefined(m, typeObsolete)
  ```
  Der **IsDefined()** – Aufruf enthält als ersten Aktualparameter ein Element aus einem **MemberInfo[]** – Array, dessen Herkunft gleich beschrieben wird.

Als zweiter **IsDefined()** - Aktualparameter ist das **Type**-Objekt zur interessierenden Attributklasse anzugeben, was im Beispiel mit Hilfe des **typeof**-Operators geschieht:

```
Type typeObsolete = typeof(ObsoleteAttribute);
```

Anders als bei der Attributvergabe (siehe Abschnitt 14.1) ist dabei der Name der Attributklasse vollständig (inkl. Namenbestandteil *Attribute*) zu schreiben.

Das im zweiten **IsDefined()** - Aufruf benötigte **MemberInfo**-Objekt zur `Teller`-Methode `Tell()` besorgt die Instanzmethode **FindMembers()** der Klasse **Type**. Sie liefert Informationen 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 Klassen- *und* Instanz-bezogene Member zugelassen, sofern diese als **public** deklariert sind:[1]

  ```
  BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public
  ```

- Als dritter Aktualparameter sorgt im Beispiel das durch die Eigenschaft **Type.FilterName** referenzierte Delegatenobjekt vom Typ **MemberFilter** für eine namensorientierte Filterung (unter Beachtung der Groß-/Kleinschreibung). Mit der Rückgabe **true** signalisiert das Delegatenobjekt, dass der Name eines Mitglieds dem im letzten **FindMembers()** - Parameter genannten Kriterium entspricht.
- Als letzten Parameter übergibt man den Namen(sanfang) der interessierenden Member, wobei am Ende das eine beliebige Zeichenfolge vertretende Jokerzeichen ***** erlaubt ist.

Soll nicht nur die Existenz eines Attributs festgestellt, sondern auch sein Innenleben exploriert werden, dann eignet sich die statische Methode **GetCustomAttribute()** der Klasse **System.Attribute**. Sie rekonstruiert das angeheftete Objekt aus den Metadaten im Assembly, sodass 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:[2]

```
static void ObsoleteMethodCheck(Type tt) {
 MemberInfo[] members = tt.FindMembers(MemberTypes.Method,
       BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public,
       Type.FilterName, "*");
 Console.WriteLine($"Obsolete-Prüfung für die Methoden des Typs {tt.FullName}:");
 Attribute attrib;
 foreach (MemberInfo mi in members) {
  attrib = Attribute.GetCustomAttribute(mi, typeof(ObsoleteAttribute));
  if (attrib != null) {
       Console.WriteLine($"\nDie Methode {mi.Name}() ist obsolet.");
       Console.WriteLine("Message: " + (attrib as ObsoleteAttribute).Message);
  } else
       Console.WriteLine($"\nDie Methode {mi.Name}() ist noch aktuell.");
 }
}
```

Zu jeder Methode erhalten wir per **GetCustomAttribute()** ggf. das angeheftete **ObsoleteAttribute**-Objekt. Weil die Rückgabe den Typ **Attribute** besitzt, kann die **Message**-Eigenschaft erst nach einer Typumwandlung ermittelt werden.

Über die folgende Klasse `Teller`

---

[1] Wie die folgende Webseite

https://docs.microsoft.com/en-us/dotnet/api/system.reflection.bindingflags

erläutert, muss zusammen mit **BindingFlags.Instance** und/oder **BindingFlags.Static** unbedingt auch **BindingFlags.Public** und/oder **BindingFlags.NonPublic** angegeben werden:

> You must specify Instance or Static along with Public or NonPublic or no members will be returned.

Wenn z. B. **BindingFlags.Instance** angegeben wird, und eine **public**-deklarierte Methode vorhanden ist, welche die Filterbedingung erfüllt, dann muss auch **BindingFlags.Public** angegeben werden, damit die Methode gefunden wird. Neben **Instance**, **Static**, **Public** und **NonPublic** hat die Enumeration **BindingFlags** noch weitere, auf der eben angegebenen Webseite erläuterte Werte.

[2] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:

…\BspUeb\Attribute\GetCustomAttribute

```
class Teller {
   [Obsolete("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: 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.
```

Anstelle der statischen **Attribute**-Methode **GetCustomAttribute()** kann die gleichnamige Erweiterungsmethode der Klasse **CustomAttributeExtensions** (Namensraum **System.Reflection**) wie eine Instanzmethode der Klasse **MemberInfo** verwendet werden:

public Attribute GetCustomAttribute(Type *attributeType***)**

Im Beispiel ist die Zeile

```
attrib = Attribute.GetCustomAttribute(mi, typeof(ObsoleteAttribute));
```

zu ersetzen durch:

```
attrib = mi.GetCustomAttribute(typeof(ObsoleteAttribute));
```

In der **Main()** - Methode des nächsten Beispielprogramms werden mit der statischen **Attribute**-Methode **GetCustomAttributes()**

public static Attribute[] GetCustomAttributes(MemberInfo *target***, bool** *inherit***)**

für eine per **Type**-Objekt beschriebene Klasse *alle* angehefteten Attribute ermittelt, wobei *geerbte* Attribute nicht interessieren (Wert **false** für den Parameter *inherit*):[1]

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:

…\BspUeb\Attribute\GetCustomAttributes

Aufgrund der Nutzung von Anweisungen auf oberster Ebene wird die Startklasse samt **Main()** – Methode vom Compiler definiert (siehe Abschnitt 4.1.2).

Quellcode	Ausgabe

`Type type = typeof(Teller);`
`Attribute[] atar =`
`  Attribute.GetCustomAttributes(type, false);`
`foreach (var at in atar)`
`  Console.WriteLine(at);`
 	
`[Obsolete] [Serializable]`
`class Teller {`
`  public static void Tell() {`
`  Console.WriteLine("Hallo!");`
`  }`
`}`
	`System.ObsoleteAttribute`
	`System.SerializableAttribute`

Alternative Überladungen erwarten zur Benennung des Attributträgers als ersten Parameter ein Objekt aus den Klassen **Assembly** oder **ParameterInfo**.

Anstelle der statischen **Attribute**-Methode **GetCustomAttributes()** kann die folgende, abstrakte Instanzmethode der Klasse **MemberInfo** verwendet werden, die von abgeleiteten Klassen (z. B. **Type**) überschrieben bzw. implementiert wird:

public abstract object[] GetCustomAttributes(bool *inherit***)**

Im Beispiel ist die Anweisung

```
Attribute[] atar = Attribute.GetCustomAttributes(type, false);
```

zu ersetzen durch:

```
object[] atar = type.GetCustomAttributes(false);
```

14.3 Attribute für Assemblies

Auch Assemblies können Attribute erhalten, wobei aber für diese Übersetzungseinheiten der Bezug nicht durch die Platzierung der Attribute im Quellcode herzustellen ist. Stattdessen setzt man den Attributen eine explizite Widmung voran, z. B.:

```
[assembly: CLSCompliant(true)]
```

Im .NET Framework, das auch unter Windows für neue Projekte nicht mehr verwendet werden sollte, wurden Assembly-Attribute in der Datei **AssemblyInfo.cs** (im Projektunterordner **Properties**) deklariert. Auch zu einer neuen WPF-Anwendung für .NET ab Version 5 legt das Visual Studio die Datei **AssemblyInfo.cs** an, diesmal allerdings im Projekthauptordner und auf ein einziges Attribut beschränkt (Kommentare weggelassen):[1]

```
using System.Windows;

[assembly: ThemeInfo(
    ResourceDictionaryLocation.None,
    ResourceDictionaryLocation.SourceAssembly
)]
```

Bei Verwendung von anderen Projektvorlagen (z. B. **Konsolen-App**) erstellt das Visual Studio die Datei **AssemblyInfo.cs** *nicht*, und sie wird auch nicht benötigt, um Assembly-Attribute festzulegen.

---

[1] Ein Visual Studio - Projekt mit dem im aktuellen Abschnitt verwendeten Beispiel ist hier zu finden:
…\BspUeb\Attribute\WpfApp

Für die meisten Assembly-Attribute existieren nun Projekteigenschaften, sodass die Attribute in der Projektdatei vergeben werden können, bei einem Projekt mit dem Namen `WpfApp` also in der Datei **WpfApp.csproj** im Hauptordner des Projekts:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net7.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <UseWPF>true</UseWPF>
    <Company>Marco Saft</Company>
    <Product>Your Tools</Product>
    <Copyright>Marco Saft</Copyright>
    <AssemblyName>YourTools</AssemblyName>
    <AssemblyVersion>1.4.2.3</AssemblyVersion>
    <InformationalVersion>1.4.2.3 Build 4</InformationalVersion>
  </PropertyGroup>

</Project>
```

Die durch das Erstellungsverfahren unterstützten Assembly-Attribute sind der folgenden Tabelle sind zu entnehmen:[1]

Projekt-eigenschaft	Assembly-Attribut	Projekteigenschaft zum Deaktivieren der Attribut-Erstellung
Company	AssemblyCompanyAttribute	GenerateAssemblyCompanyAttribute
Configuration	AssemblyConfigurationAttribute	GenerateAssemblyConfigurationAttribute
Copyright	AssemblyCopyrightAttribute	GenerateAssemblyCopyrightAttribute
Description	AssemblyDescriptionAttribute	GenerateAssemblyDescriptionAttribute
FileVersion	AssemblyFileVersionAttribute	GenerateAssemblyFileVersionAttribute
InformationalVersion	AssemblyInformationalVersionAttribute	GenerateAssemblyInformationalVersionAttribute
Product	AssemblyProductAttribute	GenerateAssemblyProductAttribute
AssemblyTitle	AssemblyTitleAttribute	GenerateAssemblyTitleAttribute
AssemblyVersion	AssemblyVersionAttribute	GenerateAssemblyVersionAttribute
NeutralLanguage	NeutralResourcesLanguageAttribute	GenerateNeutralResourcesLanguageAttribute

Lässt man im Projekt `WpfApp` das Programm mit der Debug-Konfiguration erstellen, dann werden aufgrund der Projekteigenschaften in der folgende Quellcodedatei

...\obj\Debug\net7.0-windows\WpfApp.AssemblyInfo.cs

die Assembly-Attribute deklariert:

[1] https://learn.microsoft.com/en-us/dotnet/core/project-sdk/msbuild-props#assembly-attribute-properties

```
using System;
using System.Reflection;

[assembly: System.Reflection.AssemblyCompanyAttribute("Marco Saft")]
[assembly: System.Reflection.AssemblyConfigurationAttribute("Debug")]
[assembly: System.Reflection.AssemblyCopyrightAttribute("Marco Saft")]
[assembly: System.Reflection.AssemblyFileVersionAttribute("1.4.2.3")]
[assembly: System.Reflection.AssemblyInformationalVersionAttribute("1.4.2.3 Build 4")]
[assembly: System.Reflection.AssemblyProductAttribute("Your Tools")]
[assembly: System.Reflection.AssemblyTitleAttribute("YourTools")]
[assembly: System.Reflection.AssemblyVersionAttribute("1.4.2.3")]
[assembly: System.Runtime.Versioning.TargetPlatformAttribute("Windows7.0")]
[assembly: System.Runtime.Versioning.SupportedOSPlatformAttribute("Windows7.0")]
```

Es wäre nutzlos, in dieser Datei Änderungen vorzunehmen.

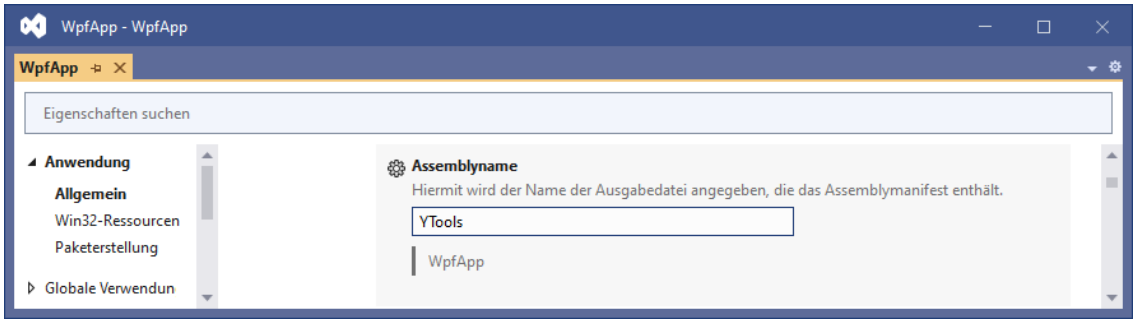
Einige Attribute werden vom MSBuild-Erstellungssystem automatisch mit Werten versorgt, z. B.:

- Das Attribut **AssemblyConfigurationAttribute** enthält die aktive Erstellungskonfiguration (**Release** oder **Debug**).
- Für das Attribut **AssemblyFileVersionAttribute** wird der zum Attribut **AssemblyVersionAttribute** vereinbarte Wert übernommen.
- Das Attribut **AssemblyTitleAttribute**, das den Assembly-Dateinamen festlegt, erhält den Projektnamen.

Ein im Visual Studio nach

Projekt > Eigenschaften

eingetragener **Assemblyname**



wird als **AssemblyName**-Element in die Projektdatei übernommen

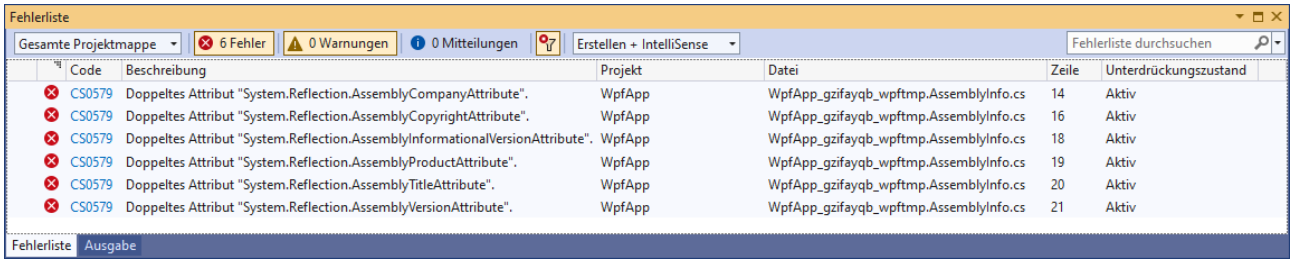
```
<AssemblyName>YTools</AssemblyName>
```

und bei der Programmerstellung in der Datei **WpfApp.AssemblyInfo.cs** als Wert für das **AssemblyTitleAttribut** verwendet:

```
[assembly: System.Reflection.AssemblyTitleAttribute("YTools")]
```

Über die Bedeutung der Assembly-Attribute mit Versionsangaben informiert der Abschnitt 2.4.2.2.

Einträge in der Datei **AssemblyInfo.cs** im Hauptordner des Projekts werden vom Compiler durchaus beachtet. Werden dort Assembly-Attribute vereinbart, dann beschwert sich der Compiler über **doppelte Attribute**, z. B.:



Um die Doppelvereinbarung von Assembly-Attributen zu verhindern, kann man die entsprechende Aktivität des MSBuild-Erstellungssystems blockieren durch das Element **GenerateAssemblyInfo** mit dem Wert **false** in der Projektdatei:

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net7.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
  </PropertyGroup>

</Project>
```

Die obige Tabelle mit den vom Erstellungssystem unterstützten Assembly-Attributen enthält Projekteigenschaften, um für einzelne Assembly-Attribute die Erstellung per MSBuild zu deaktivieren.

In der Tabelle fehlt das Attribut **CLSCompliant**, das zwar für alle Sprachbestandteile zugelassen, für Assemblies aber besonders geeignet ist. Es fordert die Beachtung der CLS (Common Language Specification, siehe Abschnitt 2.3.3) ein und sollte in der Regel für jedes Assembly vergeben werden.[1] Um es einem Assembly anzuheften, trägt man die folgende Deklaration mit der Zielangabe vor der Attributbezeichnung in eine Quellcodedatei ein:

```
[assembly: CLSCompliant(true)]
```

Ist eine Datei mit dem Namen **AssemblyInfo.cs** vorhanden (z. B. aufgrund der WPF-Projektvorlage im Visual Studio), dann spricht nichts gegen die Erweiterung der dort vorhandenen, kurzen Attributliste, z. B.:

```
using System.Windows;

[assembly: ThemeInfo(
    ResourceDictionaryLocation.None,
    ResourceDictionaryLocation.SourceAssembly
)]
[assembly: CLSCompliant(true)]
```

Man kann auch eine andere Quellcodedatei verwenden, wobei zu beachten ist, dass sich eine Assembly-Attribut – Deklaration *nicht* in einem Namensraum-Gültigkeitsbereich befinden darf, also *vor* jeder **namespace**-Direktive stehen muss.

Auf die mit Assembly-Attributen assoziierten Projekteigenschaften zu verzichten und alle Assembly-Attribute in einer Quellcodedatei (z. B. in **AssemblyInfo.cs**) zu deklarieren, hat den Vorteil, dass alle Assembly-Attribute an einem Ort versammelt sind, z. B.:

```
using System.Reflection;
using System.Windows;

[assembly: ThemeInfo(
    ResourceDictionaryLocation.None,
    ResourceDictionaryLocation.SourceAssembly
)]
[assembly: AssemblyCompany("Marco Saft")]
[assembly: AssemblyProduct("Your Tools")]
[assembly: AssemblyCopyright("Marco Saft")]
[assembly: AssemblyTitle("YTools")]
[assembly: AssemblyVersion("1.4.2.3")]
[assembly: AssemblyInformationalVersion("1.4.2.3 Build 4")]
[assembly: CLSCompliant(true)]
```

---

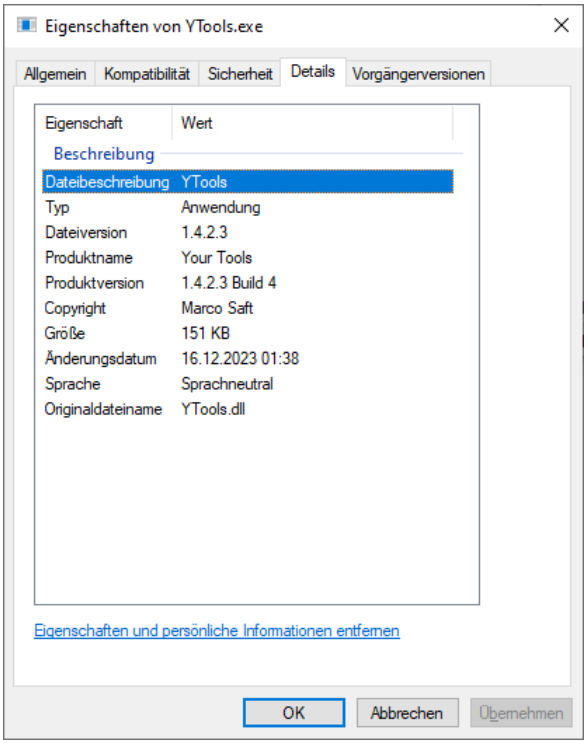
[1] https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca1014

Allerdings muss dann die parallele Deklaration durch das Erstellungssystem verhindert werden (siehe oben). Wenn statt der generellen Unterdrückung

```
<GenerateAssemblyInfo>false</GenerateAssemblyInfo>
```

vorsichtshalber *einzelne* Attribute von der Erstellung durch MSBuild ausgenommen werden, dann entsteht einiger Aufwand. Daher ist die Ergänzung der Assembly-Attribut - Projekteigenschaften durch die Attribut-Vergabe per Quellcode (z. B. für **CLSCompliant**) die bessere Lösung.

Der Windows-Explorer präsentiert etliche Assembly-Attribute im Eigenschaftsdialog der Assembly-Datei, z. B.:



14.4 Einschub: nameof-Operator

Im weiteren Verlauf des Abschnitts macht sich der **nameof**-Operator mehrfach nützlich, und statt weiterer Vorgriffe auf eine spätere Behandlung, für die sich kaum noch eine ideale Stelle finden lässt, erläutern wir den **nameof**-Operator jetzt und unterbrechen dazu die Behandlung der Attribute.

Man verwendet den **nameof**-Operator, wenn im Quellcode der Name einer Variablen, eines Typs oder eines Mitglieds als **String**-Objekt benötigt wird. Das folgenden Programm demonstriert einige **nameof**-Argumente:

```
var person = new Person("otto", "Rempremerding");
Console.WriteLine($"Typ:                    {nameof(Person)}");
Console.WriteLine($"Variable:               {nameof(person)}");
Console.WriteLine($"Mitglied (Eigenschaft): {nameof(Person.FirstName)}");
Console.WriteLine($"Mitglied (Methode):     {nameof(Person.UpperName)}");
Console.WriteLine($"Mitglied, indirekt:     {nameof(Person.FirstName.Length)}");
```

```
public class Person {
    public string FirstName;
    public string LastName;
    public Person(string first, string last) {
        FirstName = first; LastName = last;
    }
    public void UpperName() {
        Console.WriteLine($"{LastName}, {FirstName}".ToUpper());
    }
}
```

Es produziert die Ausgabe:

```
Typ:                    Person
Variable:               person
Mitglied (Eigenschaft): FirstName
Mitglied (Methode):     UpperName
Mitglied, indirekt:     Length
```

Zur praxisnahen Demonstration des **nameof**-Operators betrachten wir den Konstruktor der BCL-Klasse **PropertyChangedEventArgs**. Dort ist als **String**-Parameter der Name einer Eigenschaft anzugeben:

public PropertyChangedEventArgs(String *propertyName***)**

Dieser Konstruktor ist beteiligt, wenn ein Ereignis mit dem Delegatentyp **PropertyChanged-EventHandler** aufgerufen wird, um registrierte Wertveränderungs-Interessenten zu informieren. Diese erfahren über ein **PropertyChangedEventArgs**-Objekt, welche Eigenschaft geändert wurde, z. B.:

```
PropertyChanged.Invoke(this, new PropertyChangedEventArgs("Einkommen"));
```

Den Namen der Eigenschaft als Zeichenfolgenliteral anzugeben, ist nicht zu empfehlen:

- Der Compiler kann Tippfehler nicht verhindern.
- Wenn sich der Name ändert, werden Zeichenfolgenliterale bei der Refaktorierung durch die Entwicklungsumgebung nicht berücksichtigt.

Mit Hilfe des **nameof**-Operators lässt sich das Zeichenkettenliteral vermeiden:

```
new PropertyChangedEventArgs(nameof(Einkommen))
```

Wird im Beispiel die Eigenschaft `Einkommen` per Refaktorierung umbenannt in `Income`, dann wird der Konstruktoraufruf automatisch aktualisiert:

```
new PropertyChangedEventArgs(nameof(Income))
```

Auch bei einer Fehlerbeschreibung (z. B. im **Message**-Parameter einer Ausnahme) kann man per **nameof**-Operator die beschriebenen Vorteile nutzen, z. B.:

```
new ArgumentException(string.Format(
    $"Falscher Wert {value} bei Eigenschaft {nameof(Einkommen)}"))
```

14.5 Weitere BCL-Attribute

Von den zahlreichen bislang unerwähnt gebliebenen BCL-Attributen werden in diesem Abschnitt einige aus unterschiedlichen Gründen vorgestellt:

- Einige Attribute haben im typischen Programmieralltag eine relevante Einsatzwahrscheinlichkeit (z. B. **CallerMemberNameAttribute**, **DebuggerStepThroughAttribute**, **ConditionalAttribute**).
- Einige Attribute demonstrieren Reflexionstechniken (z. B. **Flags**).

- Mit dem **StructLayoutAttribute** und dem **FieldOffsetAttribute** wird eine Erklärung zur Funktionsweise des im Abschnitt 4.3.5.1 vorgestellten Programms `FloatBits` nachgeliefert.

Der Abschnitt hat überwiegend Nachschlagecharakter und kann gefahrlos übersprungen werden.

14.5.1 Aufruferinformations-Attribute

In diesem Abschnitt werden Aufruferinformations-Attribute aus dem Namensraum **System.Runtime.CompilerServices** vorgestellt, die für *optionale* Methodenparameter (mit Voreinstellungswerten, siehe Abschnitt 5.3.3.2) zugelassen sind und den Compiler veranlassen, die Aktualparameterwerte unter Verwendung von Informationen aus dem Quellcode des Aufrufers zu setzen:[1]

- **CallerMemberName**
  Der Compiler fügt den Namen des Aufrufers (z. B. Methode oder Eigenschaft) ein.
- **CallerFilePath**
  Der Compiler fügt den Pfad der Quellcodedatei ein.
- **CallerLineNumber**
  Der Compiler fügt die Quellcodezeile mit dem Aufruf ein.
- **CallerArgumentExpression**
  Dieses seit C# 10 verfügbare Attribut veranlasst den Compiler, zum dekorierten Formalparameter die vom Aufrufer verwendete Aktualparameter-Zeichenfolge einzufügen.

Mögliche Einsatzzwecke für diese Attribute sind:

- Implementation der Schnittstelle **INotifyPropertyChanged** (siehe unten)
- Protokolldateieinträge zur Unterstützung der Fehlersuche
- Ablaufverfolgung, Debuggen und Erstellung von Diagnosewerkzeugen

Im folgenden Programm[2]

```
using System;
using System.Runtime.CompilerServices;

int a = 3, b = 4;
M(a + b); // Zeile 5

static void M(int argument,
              [CallerArgumentExpression(nameof(argument))] string argString = null,
              [CallerMemberName] string memberName = null,
              [CallerFilePath] string filePath = null,
              [CallerLineNumber] int? lineNumber = null) {
    Console.WriteLine($"Aktualparameter-Wert          {argument}");
    Console.WriteLine($"Aktualparameter-Zeichenfolge: {argString}");
    Console.WriteLine($"Aufrufer:                     {memberName}");
    Console.WriteLine($"Quellcodedatei:               {filePath}");
    Console.WriteLine($"Zeile mit dem Aufruf:         {lineNumber}");
}
```

weiß die Methode `M()` einiges über ihren Aufrufer:

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/attributes/caller-information

[2] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\Attribute\Aufruferinformationsattribute

```
Aktualparameter-Wert          7
Aktualparameter-Zeichenfolge: a + b
Aufrufer:                     <Main>$
Quellcodedatei:               C:\Users\ ... \Program.cs
Zeile mit dem Aufruf:         5
```

Wenn ein Typ die Schnittstelle **INotifyPropertyChanged** im Namensraum **System.ComponentModel**[1]

public interface INotifyPropertyChanged {
event PropertyChangedEventHandler? PropertyChanged;
}

erfüllt und folglich das Ereignis **PropertyChanged** mit dem Delegatentyp

public delegate void PropertyChangedEventHandler(object *sender***,**
PropertyChangedEventArgs *e***)**

anbietet, der ein Parameterobjekt aus der Klasse **PropertyChangedEventArgs** mit dem folgenden Konstruktor

public PropertyChangedEventArgs(String? *propertyName***)**

benötigt, dann können sich Interessenten über Wertveränderungen bei beliebigen Instanzeigenschaften informieren lassen. Dabei gelingt es mit Hilfe des Attributs **CallerMemberName**, Literale mit Eigenschaftsnamen im Quellcode zu vermeiden. Weil ein praxisnahes Programm zur Demonstration der Datenbindung mit Hilfe der Schnittstelle **INotifyPropertyChanged** viel Lesezeit beanspruchen würde, beschränken wir uns auf eine kurze Konsolenanwendung, die zumindest die nützliche Mitwirkung des Attributs **CallerMemberName** zeigt:[2]

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

var cust = new Customer();
cust.PropertyChanged += delegate (object sender, PropertyChangedEventArgs e) {
        Console.WriteLine(e.PropertyName + " geändert auf: " + cust.Name);
    };
cust.Name = "Rempremerding";
```

[1] Die Bedeutung des Fragezeichens am Ende von Typnamen (z. B. **PropertyChangedEventHandler?**) wird im Abschnitt 15.1 erklärt.

[2] Ein Visual Studio - Projekt mit dem Beispiel enthält der folgende Ordner:
…\BspUeb\Attribute\NotifyPropertyChangedDemo
Ähnliche Programme finden sich bei Albahari 2022 (S. 230) und auf der folgenden Webseite:
https://learn.microsoft.com/en-us/dotnet/api/system.componentmodel.inotifypropertychanged

```
public class Customer : INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;

    void NotifyPropertyChanged([CallerMemberName] string propertyName = null) {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }

    string name;
    public string Name {
        get {
            return name;
        }
        set {
            if (value != name) {
                name = value;
                NotifyPropertyChanged();
            }
        }
    }
}
```

Dass die (vom Compiler aufgrund von Anweisungen auf oberster Ebene) definierte Startklasse mit Hilfe einer Ereignisbehandlungsmethode über die per Hauptmethode selbst veranlasste Eigenschaftsänderung informiert wird, ist sicher kein praxistaugliches Muster:[1]

```
Name geändert auf: Rempremerding
```

In der Klasse `Customer` wird das **PropertyChanged** – Ereignis ausgelöst, ohne die betroffene Eigenschaft (über den Feldnamen oder eine Zeichenfolge) anzugeben. Dabei spielt das Attribut **CallerMemberName** in der Methode `NotifyPropertyChanged` eine wesentliche Rolle:

```
void NotifyPropertyChanged([CallerMemberName] string propertyName = null) {
    PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}
```

Während die Benennung einer geänderten Eigenschaft über ein Zeichenfolgenliteral nach Möglichkeit zu vermeiden ist, damit der Quellcode bei einer Umbenennung per Refaktorierung (siehe Abschnitt 3.3.5) gültig bleibt, ist die Ansprache über einen Variablennamen unproblematisch, sofern sie im Quellcode der betroffenen Eigenschaft erfolgt. Mit Hilfe des **nameof**-Operators (siehe Abschnitt 14.4) könnte man im Beispiel auf die Methode `NotifyPropertyChanged` (und damit auch auf das **CallerMemberNameAttribute**) verzichten und das Änderungsereignis für die Eigenschaft `Name` folgendermaßen aufrufen:

---

[1] An dem aus didaktischen Gründen einfach gehaltenen Programm ist zudem zu kritisieren, dass der **PropertyChanged**-Handler in der Lage sein sollte, auf Änderungen bei *beliebigen* Eigenschaften zu reagieren. In dieser Anweisung

```
Console.WriteLine(e.PropertyName + " geändert auf: " + cust.Name);
```

wird aber beim Zugriff auf den neuen Eigenschaftswert das Wissen um die selbst initiierte Änderung der `Name`-Eigenschaft genutzt. Über ein Objekt der Klasse **PropertyInfo** aus dem Namensraum **System.Reflection**, geliefert von der **Type**-Methode **GetProperty()**, kann der **PropertyChanged**-Handler den neuen Wert einer beliebigen Eigenschaft ermitteln:

```
object s = cust.GetType().GetProperty(e.PropertyName).GetValue(cust);
Console.WriteLine(e.PropertyName + " geändert auf: " + s);
```

```
string name;
public string Name {
    get {
        return name;
    }
    set {
        if (value != name) {
            name = value;
            PropertyChanged.Invoke(this, new PropertyChangedEventArgs(nameof(Name)));
        }
    }
}
```

14.5.2 Bedingte Methodenausführung per ConditionalAttribute

Wenn Kontrollausgaben oder andere Operationen nur in bestimmten Projektphasen (z. B. bei der Fehlersuche) erfolgen sollen, dann kann man folgendermaßen verfahren:

- Man definiert eine Methode mit den bedingt auszuführenden Operationen.
- Diese Methode erhält das Attribut **Conditional** aus dem Namensraum **System.Diagnostics**, wobei dem Konstruktor eine Bezeichnung der Bedingung als Aktualparameter zu übergeben ist, z. B.:

```
public class Logger {
   [Conditional("Condition")]
   public static void Log(string msg) {
      Console.WriteLine(msg);
   }
}
```

- Diese Methode wird wie gewohnt in Programmen aufgerufen.
- Damit die Aufrufe tatsächlich ausgeführt werden, muss im Quellcode die Zeichenfolge aus dem **Conditional**-Konstruktor per Präprozessor-Kommando definiert werden. z. B.:

```
#define Condition
using System;
using System.Diagnostics;
 . . .
```

In der **Main()** – Methode der folgenden Klasse[1]

```
class ConditionalDemo {
    static void Main() {
        Random ran = new();

        int anzahl = 1_000;
        double zuf = 0.0;
        for (int i = 0; i < anzahl; i++) {
            zuf += ran.NextDouble();
            if (i >= 100 && i % 100 == 0)
                Logger.Log($"{i,10}   {zuf/i}");
        }
        Console.WriteLine($"Mittelwert = {zuf/anzahl}");
    }
}
```

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\Attribute\ConditionalAttribute

werden mit Hilfe der bedingt auszuführenden statischen Methode `Logger.Log()` Kontrollausgaben während der Ausführung einer Schleife produziert:[1]

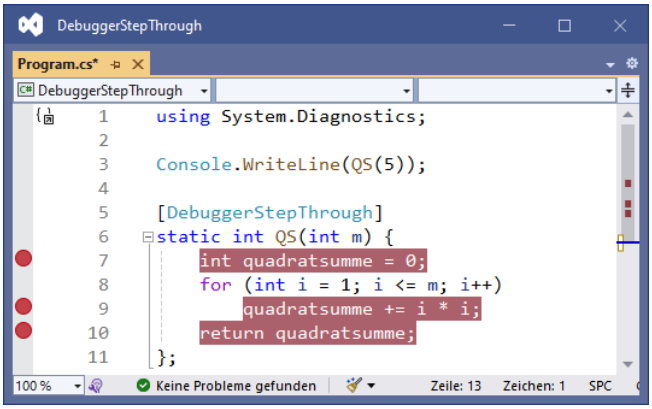
```
        100   0,4492180282255962
        200   0,4497154585241864
        300   0,4797622750051245
        400   0,4900652632449492
        500   0,49642957455652326
        600   0,49998410493366224
        700   0,5018408939733087
        800   0,5103107900235444
        900   0,5072669379730661
Mittelwert = 0,5068202839899024
```

Die `Log()` - Methodenaufrufe unterbleiben bei einem Programmeinsatz ohne das Präprozessor-Kommando.

Neben **#define** kennt C# noch etliche weitere Präprozessor-Kommandos, von denen einige im Abschnitt 12.3.4.2 vorgestellt wurden. In der C# - Online-Dokumentation findet sich eine komplette Liste.[2]

14.5.3 Haltepunkte im Debug-Modus ignorieren

Ist eine Methode mit dem **DebuggerStepThroughAttribute** dekoriert, dann werden bei der Ausführung im Debug-Modus in der Methode befindliche Haltepunkte ignoriert, z. B.:



Bei der Fehlersuche im Debug-Modus lassen sich die in einer Methode befindlichen Haltepunkte temporär deaktivieren, ohne sie löschen zu müssen.

---

[1] Der Mittelwert von **double**-Zufallszahlen aus dem Intervall [0, 1) strebt mit wachsender Stichprobengröße nach dem Gesetz der großen Zahl gegen den Erwartungswert 0,5.

[2] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives/

14.5.4 Bitfelder per FlagsAttribute

Bei einem Enumerationstyp (vgl. Abschnitt 6.4) signalisiert der Entwickler 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 und 1). 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 **ModifierKeys**-Wert

```
ModifierKeys.Control | ModifierKeys.Shift
```

  wird z. B. in der folgenden Behandlungsmethode für das **KeyDown**-Ereignis geprüft, ob die **Strg**- und die Umschalttaste simultan gedrückt sind:

```
void this_KeyDown(object sender, KeyEventArgs e) {
  if (e.KeyboardDevice.Modifiers == (ModifierKeys.Control | ModifierKeys.Shift))
    MessageBox.Show(e.KeyboardDevice.Modifiers.ToString());
}
```

Bei einem gewöhnlichen Enumerationstyp (ohne **FlagsAttribute**) …

- stehen die Werte für die sich *gegenseitig ausschließenden* Ausprägungen *eines* Merkmals. Z. B. codieren bei der im Abschnitt 12.7.3.2 erwähnten Enumeration **HorizontalAlignment** (im Namensraum **System.Windows**) die ersten vier nicht-negativen **int**-Werte jeweils eine horizontale Orientierung eines WPF-Steuerelements im übergeordneten Element (**Left**, **Center**, **Right**, **Stretch**):

```
public enum HorizontalAlignment {
   Left = 0,
   Center = 1,
   Right = 2,
   Stretch = 3
}
```

- Eine bitweise ODER-Verknüpfung der Werte ist zwar syntaktisch erlaubt, aber sinnlos.

Das folgende WPF-Programm[2]

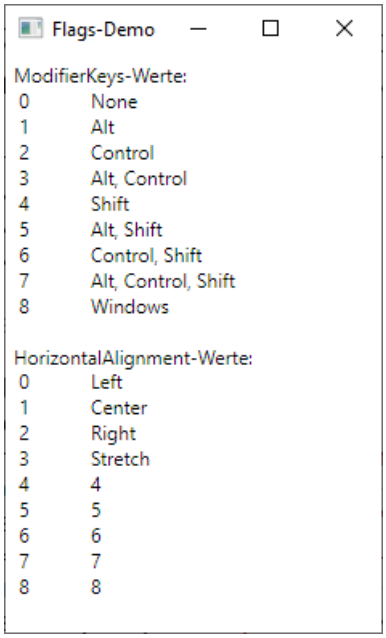
---

[1] Wie man den BCL-Quellcode einsehen kann, erläutert der Abschnitt 2.6.4.

[2] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
…\BspUeb\Attribute\Flags

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;

class FlagsDemo : Window {
  FlagsDemo() {
    Label lbl = new();
    AddChild(lbl);
    string flags = "ModifierKeys-Werte:\n";
    for (ModifierKeys i = 0; (int)i <= 8; i++)
      flags += " " + (int)i + "\t" + i + "\n";
    flags += "\nHorizontalAlignment-Werte:\n";
    for (HorizontalAlignment i = 0; (int)i <= 8; i++)
      flags += " " + (int)i + "\t" + i + "\n";
    lbl.Content = flags;
    Title = "Flags-Demo"; Width = 250; Height = 400;
    this.KeyDown += this_KeyDown;
  }

  private void this_KeyDown(object sender, KeyEventArgs e) {
    if (e.KeyboardDevice.Modifiers == (ModifierKeys.Control | ModifierKeys.Shift))
      MessageBox.Show(e.KeyboardDevice.Modifiers.ToString());
  }

  [System.STAThread]
  static void Main() {
    new Application().Run(new FlagsDemo());
  }
}
```

demonstriert, dass die Summe von zwei **ModifierKeys**-Werten wieder ein sinnvoller Wert dieses Typs ist, die Summe von zwei **HorizontalAlignment**-Werten hingegen nicht:

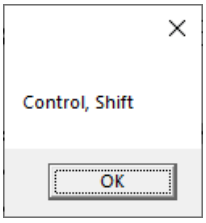


Die im Konstruktor zum Erstellen der **Content**-Zeichenfolge `flags` für das **Label**-Steuerelement implizit aufgerufene Methode **ToString()** liefert eine informative Rückgabe, weil die von der Enumerationsbasisklasse **Enum** geerbte Methode die Existenz des **Flags**-Attributs überprüft

```
string result = enumInfo.HasFlagsAttribute ?
                            FormatFlagNames(enumInfo, value) :
                            GetNameInlined(enumInfo, value);
```

und ggf. für jeden Wert der Enumeration eine Komma-separierte Liste der Merkmale mit einem angeschalteten Bit ausgibt. Hier orientiert also eine Methode ihr Verhalten an der Anwesenheit eines Attributs.

Die beim Anwendungsfenster registrierte Behandlungsmethode für das **KeyDown**-Ereignis sorgt für das Erscheinen der folgenden Dialogbox, wenn die **Strg**- und die Umschalttaste simultan gedrückt werden:



14.5.5 Unions per StructLayoutAttribute und FieldOffsetAttribute

Von der im Abschnitt 4.3.5.1 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, um eine **Union** im Sinn der Programmiersprache C nachzubilden.[1] In unseren Begriffen handelt es sich dabei um eine Struktur, deren Instanzvariablen im Speicher (zumindest teilweise) überlappend abgelegt sind. Im Beispiel soll damit nicht etwa Speicherplatz gespart, sondern eine unterschiedliche Interpretation desselben Speicherinhalts ermöglicht werden.

Die mit dem (frei gewählten) Namen `Union` definierte Struktur besitzt Felder vom Typ **float** und **uint**:

```
[StructLayout(LayoutKind.Explicit)]
public struct Union {
   [FieldOffset(0)]
   public float f;
   [FieldOffset(0)]
   public uint u;
}
```

Per **StuctLayoutAttribut** mit dem 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**, **uint**) haben denselben Platzbedarf von vier Bytes (siehe Abschnitt 4.3.4).

Das Programm `FloatBits` schreibt den vom Benutzer angegebenen **float**-Wert in das `f`-Feld einer `Union`-Instanz und liest anschließend über das `u`-Feld der Instanz aus demselben Speicherbereich einen **uint**-Wert:

---

[1] Ein Visual Studio - Projekt mit dem Programm `FloatBits` ist hier zu finden:
…\BspUeb\Elementare Sprachelemente\Bits\FloatBits

```
class FloatBits {
    static void Main() {
        Union uni = new();
        Console.Write("float: ");
        if (!float.TryParse(Console.ReadLine(), out uni.f)) {
            Console.WriteLine("Falsches Format");
            return;
        }
        Console.WriteLine("\nBits:\n1 12345678 12345678901234567890123");
        Console.WriteLine($"{uni.u:b32}"[..1] + " " + $"{uni.u:b32}"[1..9] + " " +
                          $"{uni.u:b32}"[9..]);
    }
}
```

14.6 Eigene Attribute definieren

Wir müssen uns nicht auf die Vergabe und Auswertung von BCL-Attributen beschränken, sondern können auch eigene Attribute definieren und zur deklarativen Programmierung nutzen. Eine eigene Attributklasse …

- wird aus der Klasse **System.Attribute** (direkt oder indirekt) abgeleitet
- und erhält einen Namen, der mit dem Wort *Attribute* endet.

Um den Compiler darüber zu informieren, welchen Programmbestandteilen das neue Attribut angeheftet werden darf, verwendet man ein Metaattribut aus der Klasse **AttributeUsageAttribute** (ein Attribut, das an Attribute geheftet wird). Über die erlaubten Ziele für Attribute informiert die C# -Online-Dokumentation.[1] Im folgenden Beispiel erhält der Konstruktorparameter *validOn* den Wert **AttributeTargets.Class**, sodass nur Klassen mit dem neu definierten `NonsenseAttribute` dekoriert werden dürfen. Außerdem wird mit dem Wert **false** für die **AttributeUsageAttribute**-Eigenschaft **Inherited** verhindert, dass eine dekorierte Klasse das `NonsenseAttribute` an abgeleitete Klassen vererbt:

```
[AttributeUsage(AttributeTargets.Class, Inherited=false)]
public sealed class NonsenseAttribute : Attribute {
   public int Level { get; }
   public NonsenseAttribute(int level_) {
      Level = level_;
   }
}
```

Mit dem Wert **true** für die **AttributeUsageAttribute**-Eigenschaft **AllowMultiple** könnte man es erlauben, einer Klasse mehrere `NonsenseAttribute`-Objekte anzuheften.

Die drei genannten Attribut-Argumente besitzen die folgenden Voreinstellungen:

Parameter *validOn*	**AttributeTargets.All**
Eigenschaft **Inherited**	**true**
Eigenschaft **AllowMultiple**	**false**

Wie der Tabelle zu entnehmen ist, werden per Voreinstellung …

- die Attribute einer Klasse an Ableitungen übertragen,
- die Attribute von Klassenmitgliedern (z. B. Methoden) an überschreibende Mitglieder von abgeleiteten Klassen übertragen.

Wie das Metaattribut **AttributeUsage** demonstriert, sind bei der Attributvergabe wie bei einem Objekt- bzw. Instanz-Initialisierer (siehe Abschnitt 5.4.3.2) nach den Positionsparametern eines

[1] https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/

Konstruktors auch Namensparameter erlaubt, um öffentliche Eigenschaften oder Felder (in beliebiger Reihenfolge) zu initialisieren.

Für Positions- oder Namensparameter eines Attributs sind ausschließlich die folgenden Datentypen erlaubt:[1]

- **bool**, **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, **double,**
- **Object**, **String**, **Type** (alle aus dem Namensraum **System**)
- Aufzählungstypen
- Eindimensionale Arrays mit einem Elementtyp aus der obigen Liste

Bei der Vergabe eines Attributs sind zum Initialisieren der Positions- oder Namensparameter nur *konstante Ausdrücke* erlaubt. Folglich misslingt z. B. der Versuch, ein Attribut unter Verwendung einer Klassenvariablen zu konstruieren und einer eingeschachtelten Klasse anzuheften:[2]

```
class Prog {
    static int level = 13;
    [NonsenseAttribute(level)]
    class NestedCls { }
```

(Feld) static int Prog.level

CS0182: Ein Attributargument muss ein constant-, typeof- oder Arrayerstellungsausdruck eines Attributparametertyps sein.

Microsoft empfiehlt, eigene Attributklassen als versiegelt (**sealed**) zu definieren, damit Reflexionsmethoden (z. B. **GetCustomAttribute()**, siehe Abschnitt 14.2) keine abgeleiteten Klassen berücksichtigen müssen und somit Zeit sparen.[3]

In einer Übungsaufgabe zum aktuellen Kapitel sollen Sie das eben definierte `NonsenseAttribute` auf eine Klasse anwenden und zur Laufzeit auswerten (siehe Abschnitt 14.8).

14.7 Sonstige Attributoptionen

In diesem Abschnitt werden kürzlich in C# eingeführte Attributoptionen vorgestellt, die Einsteiger beim ersten Lesen ignorieren dürfen.

14.7.1 Generische Attribute

Bei der Erweiterung von C# um das typgenerische Programmieren (mit C# 2.0, im Jahr 2005) wurde den (von Beginn an in der Sprache enthaltenen) Attributen diese Verbesserung vorenthalten. Erst mit C# 11 wurde diese Lücke geschlossen.[4]

14.7.1.1 Traditionelle Kompensation per Type-Parameter

Vor C# 11 konnte bei Attributklassen die fehlende Generizität über einen Parameter vom Datentyp **System.Type** kompensiert werden. Um die Schwäche dieser Lösung in einem nicht zu aufwändigen Beispiel zu demonstrieren, betrachten wir das `PersistenceHelperAttribute`, das Klassen einen Helfer für das Sichern (Persistieren) ihrer Objekte vermitteln soll:[5]

---

1 https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/attributes#2224-attribute-parameter-types

2 Jon Skeet diskutiert in einem Blogbeitrag weitere Details zur Einschränkung von Attribut-Argumenten auf konstante Ausdrücke:
https://codeblog.jonskeet.uk/2014/08/22/when-is-a-constant-not-a-constant-when-its-a-decimal/

3 https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca1813

4 https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history
https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-11#generic-attributes

5 Eine solche Funktionsinjektion ist sicher auch anders zu realisieren. Für die Lösung per Attribut spricht u. a., dass keine Änderungen am Quellcode einer Klasse erforderlich sind.

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class PersistenceHelperAttribute : Attribute {
    public Type PersistenceHelperType { get; }

    public PersistenceHelperAttribute(Type persHlp) {
        if (typeof(IPersistenceHelper).IsAssignableFrom(persHlp))
            PersistenceHelperType = persHlp;
        else
            throw new Exception("Kein IPersistenceHelper");
    }
}
```

Bei der Vergabe des Attributs versorgt ein **typeof**-Ausdruck den **Type**-Parameter mit dem Persistenzhelfertyp, z. B.:

```
[PersistenceHelper(typeof(PersHlprSim))]
public class TestKlasse { }
```

Ein in Frage kommender Persistenzhelfer muss die Schnittstelle `IPersistenceHelper` implementieren:

```
public interface IPersistenceHelper {
    void Persist(object obj);
}
```

Die Klasse `PersHlprSim` implementiert die Schnittstelle, beschränkt sich aber auf das Simulieren einer Persistenzunterstützung:

```
public class PersHlprSim : IPersistenceHelper {
    public void Persist(object obj) {
        Console.WriteLine("Simuliertes Persistieren");
    }
}
```

Zum Persistieren von Objekten aus unterstützten (mit dem Attribut `PersistenceHelper` dekorierten) Klassen eignet sich die folgende Methode, die den unterstützenden Typ über das Feld `PersistenceHelperType` ermittelt und über die statische Methode **CreateInstance()** der Klasse **System.Activator** ein Objekt dieser Klasse erzeugt. Wenn das erzeugte Objekt die Schnittstelle **IPersistenceHelper** implementiert, dann wird es zur Ausführung der Methode `Persist()` aufgefordert:

```
static void Persistiere(object obj) {
  var attributes = obj.GetType().GetCustomAttributes(typeof(PersistenceHelperAttribute),
                                                     false);
  if (attributes.Length == 1 && attributes[0] is PersistenceHelperAttribute persHlpAtt) {
    var persistenceHelperType = persHlpAtt.PersistenceHelperType;
    if (Activator.CreateInstance(persistenceHelperType) is IPersistenceHelper persHlp)
      persHlp.Persist(obj);
    }
}
```

Das folgende Top-Level – Programm[1]

```
var testObj = new TestKlasse();
Persistiere(testObj);
```

produziert die erwartete Ausgabe:

```
Simuliertes Persistieren
```

Das Beispiel übernimmt Ideen und technische Lösungen aus einem Blog-Beitrag von Thomas Claudius Huber:
https://www.thomasclaudiushuber.com/2023/01/17/csharp-11-generic-attributes/

[1] Ein Visual Studio - Projekt mit der im aktuellen Abschnitt vorgestellten Lösung ist hier zu finden:
…\BspUeb\Attribute\Generische Attribute\typeof

Im bisherigen Verlauf des Abschnitts waren nützliche Details zur Attributverwendung und zur Reflexion (Objektkreation per **Activator**) zu sehen, aber kein Problem mit dem **Type**-Parameter der Klasse `PersistenceHelperAttribute`. Die folgende Attributvergabe schafft aber ein Problem, das vom Compiler nicht zu verhindern ist:

```
[PersistenceHelper(typeof(int))]
public class TestKlasse { }
```

Die Struktur **Int32** (alias **int**) ist als Persistenzhelfer nicht geeignet, doch kann der Compiler die Verwendung der Struktur für den **Type**-Parameter im `PersistenceHelperAttribute`-Konstruktor nicht verhindern. Weil ein möglichst früh geworfenes Ausnahmeobjekt im Vergleich zu einem undefinierten Programmverhalten das kleinere Problem ist, reagiert der Konstruktor auf den ungeeigneten Typ mit einem Laufzeitfehler:[1]

```
public PersistenceHelperAttribute(Type persHlp) {
    if (typeof(IPersistenceHelper).IsAssignableFrom(persHlp))
        PersistenceHelperType = persHlp;
    else
        throw new Exception("Kein IPersistenceHelper");
}
```

Eine vom Compiler verweigerte Übersetzung ist aber im Vergleich zu einem Laufzeitfehler klar zu bevorzugen, und diese Verbesserung wird durch die im nächsten Abschnitt beschriebenen generischen Attribute ermöglicht.

14.7.1.2 Generische Lösung

Wenn ein Attribut generisch definiert wird und über die Konkretisierung seines Typformalparameters vom assoziierten Typ erfährt, dann können ungeeignete Typen über Restriktionen für den Typformalparameter abgewiesen werden. Wir ersetzen im Beispiel aus dem vorherigen Abschnitt die relativ aufwändige Attributdefinition mit **Type**-Parameter und Vorsichtsmaßnahme im Konstruktor durch eine deutlich einfachere Lösung,

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class PersistenceHelperAttribute<T> : Attribute
                                                    where T : IPersistenceHelper { }
```

die zudem eine Typüberwachung durch den Compiler ermöglicht:

```
[PersistenceHelper<int>]
public class TestKla
```

readonly struct System.Int32
Represents a 32-bit signed integer.

CS0315: Der Typ "int" kann nicht als Typparameter "T" im generischen Typ oder in der generischen Methode "PersistenceHelperAttribute<T>" verwendet werden. Es ist keine Boxing-Konvertierung von "int" in "IPersistenceHelper" vorhanden.

An der Methode `Persistiere()` sind nur minimale Änderungen erforderlich.[2] Den Typ des Persistenzhelfers erfährt man über die **Type**-Eigenschaft **GenericTypeArguments**, die einen Array mit den Typaktualparametern referenziert:

---

[1] Diese Idee stammt von der Webseite:
https://programmers.com/blog/first_class_support_for_attributes_in_cs11

[2] Ein Visual Studio - Projekt mit der im aktuellen Abschnitt vorgestellten Lösung ist hier zu finden:
…\BspUeb\Attribute\Generische Attribute\Typformalparameter

```
static void Persistiere(object obj) {
  var persHlpAtt =
   obj.GetType().GetCustomAttributes(typeof(PersistenceHelperAttribute<>), false)[0];
  if (persHlpAtt != null) {
    var persHlpType = persHlpAtt.GetType().GenericTypeArguments[0];
    if (Activator.CreateInstance(persHlpType) is IPersistenceHelper persHlp)
        persHlp.Persist(obj);
  }
}
```

Zu erwähnen ist die für offene (nicht-konkretisierte) generische Typen in der Rolle eines **typeof**-Parameters vorgeschriebene Syntax: Es ist ein Paar spitzer Klammern *ohne* Typformalparameter anzugeben, z. B.:[1]

```
typeof(PersistenceHelperAttribute<>)
```

Bei der Konkretisierung eines Typformalparameters zu einem Attribut sind dieselben Typen erlaubt wie beim Parameter des **typeof**-Operators. Verboten sind also die folgenden Typen:

- **dynamic**
  Der Datentyp **dynamic** sollte nur in begründeten Ausnahmefällen (z. B. für die Kooperation mit typfreien Skriptsprache wie JavaScript) zum Einsatz kommen und wird im Manuskript nicht behandelt.[2]
- Referenztypen mit expliziter **null**-Zulassung
  Im Manuskript sind **null**-fähige Referenztypen bisher eine Selbstverständlichkeit. Um die Gefahr einer **NullReferenceException** zu reduzieren, beherrscht der Compiler seit C# 8 eine optionale Nullzustandsüberwachung (siehe Abschnitt 15.1) und warnt in diesem Betriebsmodus, sobald für eine Variable mit einem Referenztyp der Wert **null** möglich ist. Seit C# 10 wird die Nullzustandsanalyse vom Visual Studio für neue Projekte durch das folgende Element in der Projektdatei aktiviert:
  ```
  <Nullable>enable</Nullable>
  ```
  Bei aktiver Nullzustandsanalyse kann für einen Referenztyp durch eine Typbezeichnung mit angehängtem Fragezeichen (z. B. **string?**) der Wert **null** explizit erlaubt werden. Die Konkretisierung eines Attribut-Typformalparameters durch einen Referenztypen mit expliziter **null**-Zulassung ist verboten.
- die im Abschnitt 6.6 beschriebenen Tupeltypen, z. B. `(int X, int Y)`

In der Online-Dokumentation werden die Einschränkungen erläutert und einfache Lösungsmöglichkeiten beschrieben.[3]

14.7.2 Attribute für Delegaten in Lambda-Notation

Seit C# 10 dürfen Attribute auch an einen durch Lambda-Notation definierten Delegaten (siehe Abschnitt 10.1.5.2) geheftet werden, während das bei anonymen, über den **delegate**-Operator erstellten Methoden (siehe Abschnitt 10.1.5.1) verboten bleibt. Beim Delegatenstart per **Invoke()** bleiben allerdings Attribute unbeachtet, sodass man z. B. nicht durch das **ConditionalAttribute** (siehe

---

[1] Hier finden sich weitere Regeln (z. B. für den Fall von *mehreren* Typformalparametern):
https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/generics-and-attributes
Die Regeln gelten für beliebige generische Typen, also nicht nur für generische Attribute.

[2] Hier finden Sie die Online-Dokumentation zum Typ **dynamic**:
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types#the-dynamic-type

[3] https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-11#generic-attributes
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-11.0/generic-attributes

Abschnitt 14.5.2) für eine bedingte Ausführung sorgen kann.[1] Es spricht aber nichts gegen die Verwendung von Attributen zur Codeanalyse (z. B. durch den Compiler) oder zur Reflexion.

Das folgende, per Ausdrucks-Lambda realisierte Delegatenobjekt enthält jeweils ein Attribut für die Methode, für einen Formalparameter und für den Rückgabewert:

```
Predicate<string> TrueString = [AMethod] [return: AReturnValue]
          ([AParameter] string spar) => spar != null && spar.Length > 0;
```

Um die Interpreterbarkeit der Anweisung sicherzustellen, muss …

- das Attribut für den Rückgabewert eine Adressatenangabe erhalten,
- und die Parameterliste zum Ausdrucks-Lambda eingeklammert werden.

Für die verwendeten Attribute werden die zugelassenen Träger durch das Metaattribut **AttributeUsage** festgelegt:

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class AMethodAttribute : Attribute { }

[AttributeUsage(AttributeTargets.Parameter)]
public sealed class AParameterAttribute : Attribute { }

[AttributeUsage(AttributeTargets.ReturnValue)]
public sealed class AReturnValueAttribute : Attribute { }
```

Über die für Delegaten definierte Erweiterungsmethode **GetMethodInfo()** der Klasse **RuntimeReflectionExtensions** erhält man ein Objekt der Klasse **MethodInfo**, das die von einem Delegatenobjekt referenzierte Methode beschreibt:[2]

```
LambdaAttribute la = new();
MethodInfo mi = la.TrueString.GetMethodInfo();
```

Auf Befragen mit **GetCustomAttributes()** liefert das **MethodInfo**-Objekt die Methodenattribute (vom angegebenen Typ):

```
object[] methAttributes = mi.GetCustomAttributes(typeof(AMethodAttribute), true);
if (methAttributes.Length > 0) {
    Console.WriteLine($"Methode {mi.Name} hat die Attribute:");
    for (int k = 0; k < methAttributes.Length; k++)
        Console.WriteLine($"\t{methAttributes[k]}");
}
```

Weil die oben definierten Attribute keine Eigenschaften bzw. Felder besitzen, beschränken wir uns auf das Protokollieren der Namen.

Vom **MethodInfo**-Objekt ist per **GetParameters()** ein Array mit Elementen vom Typ **ParameterInfo** erhältlich. Das zu einem Parameter gehörende Element …

- liefert nach der Aufforderung durch **GetCustomAttributes()** die Methodenattribute (vom angegebenen Typ),
- und kennt die Position, den Namen sowie den Datentyp des Parameters.

Diese Informationen werden zusammen mit dem Namen des Attributes protokolliert:

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions

[2] Es spielt keine Rolle, ob es sich um einen Single- oder Multicast-Delegaten handelt, weil bei einem Multicast-Delegaten die Delegatensignaturen der referenzierten Methoden identisch sind.

```
ParameterInfo[] parameters = mi.GetParameters();
object[] paramAttributes =
             parameters[0].GetCustomAttributes(typeof(AParameterAttribute), false);
if (paramAttributes.Length > 0) {
    Console.WriteLine("\nParameter {0}, Name = {1}, Type = {2} hat die Attribute:",
        parameters[0].Position, parameters[0].Name, parameters[0].ParameterType);
    for (int k = 0; k < paramAttributes.Length; k++)
        Console.WriteLine("\t{0}", paramAttributes[k]);
}
```

Die **MethodInfo**-Eigenschaft **ReturnTypeCustomAttributes** referenziert ein Objekt vom Typ **ICustomAttributeProvider**, von dem per **GetCustomAttributes()** die Attribute zur Rückgabe erhältlich sind:

```
object[] returnAttributes = mi.ReturnTypeCustomAttributes.
                 GetCustomAttributes(typeof(AReturnValueAttribute), false);
if (returnAttributes.Length > 0) {
    Console.WriteLine($"\nMethode {mi.Name} hat die Rückgaben-Attribute:");
    for (int k = 0; k < returnAttributes.Length; k++)
        Console.WriteLine($"\t{returnAttributes[k]}");
}
```

Die beschriebenen Quellcodesegmente gehören zu einem Programm, das die Attribute zum oben definierten Ausdrucks-Lambda folgendermaßen beschreibt:[1]

```
Methode <.ctor>b__2_0 hat die Attribute:
        AMethodAttribute

Parameter 0, Name = spar, Type = System.String hat die Attribute:
        AParameterAttribute

Methode <.ctor>b__2_0 hat die Rückgaben-Attribute:
        AReturnValueAttribute
```

14.8 Übungsaufgaben zum Kapitel 14

1) Ergänzen Sie im Beispiel von Abschnitt 14.6 die ziemlich sinnlose Klasse `Dummy`

```
[Serializable] [NonsenseAttribute(13)]
class Dummy {
}
```

sowie eine Testklasse mit einer Methode, die alle benutzerdefinierten Attribute der Klasse `Dummy` und den `Level`-Wert des `NonsenseAttribut`-Objekts ausgibt.

---

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
…\BspUeb\Attribute\Lambda-Attribute

15 C# für Fortgeschrittene

In diesem Kapitel werden anspruchsvolle und bisher aus didaktischen Gründen gemiedene C# - Sprachbestandteile vorgestellt. Obwohl im weiteren Verlauf des Manuskripts wichtige BCL-Typen (z. B. zur Realisation von Datei-, Datenbank- und Netzwerkzugriffen oder zur Parallelverarbeitung) im Vordergrund stehen, ist die Beschreibung der Programmiersprache C# noch nicht abgeschlossen, weil die Sprache zur Unterstützung der genannten Anwendungsfelder um einige Schlüsselwörter und Syntaxregeln erweitert worden ist.

15.1 Vermeidung von null-Referenz - Ausnahmefehlern

Bei manchen jüngeren Programmiersprachen (z. B. Kotlin, Rust) wird die Vermeidung von **null**-Referenz - Laufzeitfehlern als wesentlicher Vorzug herausgestellt. C# bemüht sich seit der (im September 2019 erschienenen) Version 8 um eine verbesserte **null**-Sicherheit.

Für Werttypen wurde durch die **Nullable<T>** - Verpackung die **null**-Zulässigkeit nachgerüstet (siehe Abschnitt 8.3), sodass man z. B. einer **int?** - Variablen den gelegentlich sinnvollen Wert *Unbekannt* zuweisen kann. Bei Referenzvariablen ist hingegen der Wert **null** seit C# 1.0 möglich und dient sogar zur Standardinitialisierung. Seit C# 8 kann man für Referenztypen optional in einem sogenannten *Nullable-Kontext* per **?**-Annotation am Namensende zum Ausdruck bringen, ob der Wert **null** erlaubt sein soll (z. B. beim Typ **String?**) oder nicht (z. B. beim Typ **String**). Diese Unterscheidung hat wichtige Konsequenzen:

- Hat eine Variable einen **null**-unzulässigen Referenztyp, dann …
  - sollte sie den Wert **null** nie erhalten,
  - ist es akzeptabel, bei der Dereferenzierung (Objektansprache) in der Regel auf einen null-Test zu verzichten.
- Hat eine Variable einen **null**-zulässigen Referenztyp, dann …
  - darf sie den Wert **null** erhalten
  - muss vor jeder Dereferenzierung geprüft werden, ob sie aktuell den Wert **null** besitzt.

Der Compiler nutzt die Unterscheidbarkeit, um durch gezielte Warnungen das Risiko von Laufzeitfehlern vom Typ **NullReferenceException** zu minimieren.

Weil die **NullReferenceException** häufig auftritt und für die Software-Qualität von besonderer Bedeutung ist, hat sich sogar die Abkürzung *NRE* eingebürgert. Eine Ursache für NRE-Probleme sind die fehlenden Werte, die in zu modellierenden empirischen Systemen oft auftreten. Die resultierenden Probleme für die Software-Entwicklung sind beherrschbar, verursachen aber einigen Aufwand.[1]

Zwar wird die Qualität eines Programms durch eine **NullReferenceException** erheblich beeinträchtigt, doch stellt ein *nicht* per Ausnahmefehler gestopptes Fehlverhalten des Programms ein noch viel größeres Problem dar.

15.1.1 Explizite null - (Un)zulässigkeit von Referenztypen

Seit C# 8 bzw. .NET 5 kann z. B. durch das Projektdatei-Element **Nullable** bekundet werden, dass Referenztypen den Wert **null** per Voreinstellung *nicht* annehmen sollen, und dass **null**-zulässige Referenztypen (engl. *nullable reference types*) durch ein **?**-Suffix am Namensende zu kennzeichnen sind:

---

[1] Auch in der statisch-empirischen Forschung sind fehlende Werte in manchen Projekten für eine hohen Anteil des Gesamtaufwands verantwortlich.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Referenztypen mit dem gewohnten Namen (*ohne* **?**-Suffix) gelten nunmehr als *non-nullable*, d. h.:

- Sie sollen den Wert **null** nicht erhalten.
- Variablen von einem solchen Typ dürfen ohne **null**-Test dereferenziert werden.[1]

Den gewohnten Referenztypen wird also nun die non-nullable – Intention unterstellt.

Es kommen Referenztypen hinzu, die durch ein **?**-Suffix am Ende des Typnamens (z. B. **String?**) als *nullable* gekennzeichnet sind:

- Sie dürfen den Wert **null** annehmen.
- Variablen von einem solchen Typ dürfen *nicht* ohne **null**-Test dereferenziert werden.

Dieselbe **?**-Syntax wird auch zur Deklaration von **null**-fähigen Werttypen verwendet (siehe Abschnitt 8.3).

Neu sind in C# also die **null**-zulässigen Referenztypen, sodass man zurecht …

- für die Spracherweiterung die engl. Bezeichnung *nullable reference types* verwendet,
- das zum Aktivieren der Spracherweiterung genutzte Projektdatei-Element als ***Nullable*** bezeichnet
- und von einem *Nullable-Kontext* spricht, wenn für einen Quellcodebereich die Spracherweiterung aktiviert ist.

Der Nullable-Kontext wird seit .NET 6 bei neuen Projekten durch das zu Beginn des Abschnitts angegebene **Nullable**-Element in der Projektdatei voreingestellt.

Bei der Spracherweiterung haben die C#-Designer angenommen, dass **null**-fähige Referenztypen eher selten benötigt werden, und folglich diese Variante mit einer Kennzeichnungspflicht belastet (Torgersen 2017).

Während für den Compiler die **null**-(Un-)zulässigkeit von Referenzvariablen seit C# 8 eine große Bedeutung besitzt, weiß die CLR nichts davon. Zur Laufzeit gibt es *keinen* Unterschied zwischen Referenzdatentypen mit bzw. ohne **null**-Zulässigkeit, also z. B. zwischen den Datentypen **string?** und **string**. Auf Befragen mit **GetType()** nennen bei aktiviertem Nullable-Kontext eine Referenzvariable mit bzw. ohne **null**-Zulässigkeit denselben Typ, z. B.:

Quellcode	Ausgabe

`string s1 = "Test";`
`Console.WriteLine(s1.GetType());`
`string? s2 = "Test";`
`Console.WriteLine(s2.GetType());`
	`System.String`
	`System.String`

Demgegenüber sind z. B. der **null**-zulässige Werttyp **int?** (alias **Nullable<int>**) und der zugrundeliegende Werttyp **int** zur Laufzeit unterscheidbare Datentypen.

---

[1] Wenn bei Verwendung eines **null**-unzulässigen Referenztyps keine Compiler-Warnungen vor einer riskanten Dereferenzierung auftauchen, dann ist es akzeptabel, auf routinemäßige **null**-Tests zu verzichten. Allerdings ist auch in dieser Situation eine **NullReferenceException** leider *nicht* ausgeschlossen (siehe Abschnitt 15.1.2).

Für die Rückgabe des **default**-Operators (siehe Abschnitt 8.6) spielt die **null**-Zulässigkeit ebenfalls keine Rolle, sodass z. B. im Nullable-Kontext die Ausdrücke

```
default(string)
```

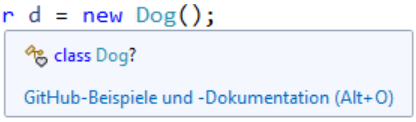
und

```
default(string?)
```

den gemeinsamen Wert **null** liefern.

Lokale Referenzvariablen mit einem vom Compiler erschlossenen Datentyp besitzen bei aktiviertem Nullable-Kontext die explizit **null**-zulässige Typvariante, z. B.:

```
var d = new Dog();
```

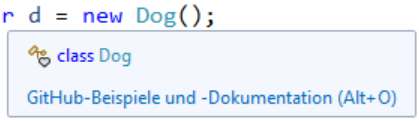


Wird das Element

```
<Nullable>enable</Nullable>
```

aus der Projektdatei entfernt (anschließendes Sichern nicht vergessen), dann besteht kein Nullable-Kontext, und der Compiler erschließt den traditionellen, implizit **null**-zulässigen Datentyp, z. B.

```
var d = new Dog();
```



Auch auf die typgenerische Programmierung wirkt sich die explizite **null** - (Un)zulässigkeit aus:[1]

- Wird ein Typformalparameter **T** durch einen **null**-unzulässigen Referenztyp (z. B. **String**) konkretisiert, dann steht **T?** für den korrespondierenden **null**-zulässigen Referenztyp (z. B. **String?**).
- Wird ein Typformalparameter **T** durch einen **null**-unzulässigen Werttyp (z. B. **int**) konkretisiert, dann steht **T?** für denselben **null**-unzulässigen Werttyp (also im Beispiel ebenfalls für **int**). Ein Werttyp (z. B. **int**) und die zugehörige **null**-fähige Variante (z. B. **int?** alias **Nullable<int>**) sind zur Laufzeit unterschiedliche Datentypen, die z. B. nicht beide in einer sortenreinen Kollektion mit Elementen vom Typ **T** auftauchen können. Bei einem Referenztyp gibt es hingegen zur Laufzeit keinen Unterschied zwischen den Varianten mit bzw. ohne **null**-Zulässigkeit.
- Wird ein Typformalparameter **T** durch einen **null**-zulässigen Referenztyp (z. B. **String?**) konkretisiert, dann steht **T?** für denselben **null**-zulässigen Referenztyp.
- Wird ein Typformalparameter **T** durch einen **null**-zulässigen Werttyp (z. B. **int?**) konkretisiert, dann steht **T?** für denselben **null**-zulässigen Werttyp.

Zur Restriktion eines Typformalparameters sind die folgenden Schlüsselwörter mit Bezug zur **null**-Zulässigkeit erlaubt:

---

[1] Weitere Regeln sind hier zu finden:
https://learn.microsoft.com/en-us/dotnet/csharp/nullable-references

- **class**
  T darf nur durch einen **null**-unzulässigen Referenztyp konkretisiert werden.
- **class?**
  Zur Konkretisierung von **T** ist ein Referenztyp vorgeschrieben, wobei die **null**-unzulässige Variante (z. B. **String**) und die **null**-zulässige Variante (z. B. **String?**) erlaubt sind.
- **notnull**
  T darf nur durch einen **null**-unzulässigen Referenz- oder Werttyp konkretisiert werden.

Auch ein Ereignis ist ein Klassenmitglied mit Referenztyp (konkret: mit Delegatentyp), und im Nullable-Kontext reklamiert der Compiler eine fehlende Initialisierung, z. B. bei der im Abschnitt 10.2.3 vorgestellten Klasse `SurpriseButton`. Der Quellcodeeditor im Visual Studio kündigt die Compiler-Warnung durch eine grüne Unterschlängelung an:

```
public class SurpriseButton : Button {
    public event SevenEventHandler Seven;
    Random zzg = new();

    protected override void OnClick() {
        . . .
    }
}
```

Weil ein Ereignis meist ein Informationsangebot an andere Klassen ist, kommt die Zuweisung eines Delegatenobjekts im Konstruktor oder Feldinitialisierer allerdings kaum in Betracht. Zur Beseitigung der Warnung bleibt daher nur die Verwendung eines **null**-zulässigen Delegatentyps, z. B.:

```
public event SevenEventHandler? Seven;
```

Im Abschnitt 10.2.3 wurde diese Komplikation aus didaktischen Gründen ignoriert, und im Manuskript tauchte weder die warnende Unterschlängelung noch der Delegatentyp mit Fragezeichen am Namensende auf. Im Abschnitt 14.5.1 ließ sich das Fragezeichen als Bestandteil eines BCL-Delegatenklassennamens nicht weiter totschweigen und musste in einer Fußnote angesprochen werden:

```
public interface INotifyPropertyChanged {
    event PropertyChangedEventHandler? PropertyChanged;
}
```

In der Online-Dokumentation tauchen die Namen von **null**-zulässigen Referenzdatentypen mit terminalem Fragezeichen oft auf, z. B.:

public delegate void PropertyChangedEventHandler(object? *sender***,**
PropertyChangedEventArgs *e***)**

Die ausstehende Behandlung der **?**-Annotation im Manuskript hat vielleicht für Verunsicherung bei den Lesern gesorgt, die nun endlich behoben ist.

15.1.2 Statische Nullzustandsanalyse

Im Nullable-Kontext erlaubt der Compiler nicht nur das **?**-Suffix für **null**-zulässige Typen, sondern er führt auch eine Nullzustandsanalyse durch. Die wird als *statisch* bezeichnet, weil sie nicht zur Laufzeit stattfindet, sondern aufgrund des Quellcodes durchgeführt wird. Der Compiler unterscheidet bei einem Ausdruck die folgenden Nullzustände

- **not-null**
- **maybe-null**

und warnt vor **null**-Referenz – Risiken, z. B.:

- Beim Laden einer Klasse bzw. bei der Erstellung eines Objekts erhält ein Feld mit einem **null**-unzulässigen Referenztyp keinen **not-null** Initialisierungswert (per Konstruktor oder Feldinitialisierer). Die folgende Klassendefinition wird als fehlerhaft kritisiert:

```
public class Person {
    public string Name;
}
// Warnung CS8618:
// Non-Nullable-Feld "Name" muss beim Beenden des Konstruktors einen Wert
// ungleich NULL erhalten. Erwägen Sie eine Deklaration von "Name" als Nullable.
```

- Einer **null**-unzulässigen Referenzvariablen wird ein Ausdruck mit dem Zustand **maybe-null** zugewiesen, z. B.

```
string s = "Test";
if (s.Length == 13)
    s = null;
string mess = s;
// Warnung CS8600:
// Das NULL-Literal oder ein möglicher NULL-Wert wird
// in einen Non-Nullable-Typ konvertiert.
```

- Eine Variable mit einem Referenztyp *mit oder ohne* **null**-Zulässigkeit wird dereferenziert, obwohl der Compiler den Wert **null** nicht ausschließen kann (**maybe-null** diagnostiziert), z. B.:

```
Console.WriteLine($"Die Länge von mess ist {mess.Length}");
// Warnung:
// CS8602: Dereferenzierung eines möglichen Nullverweises.
```

  Die Warnung vor Dereferenzierungsfehlern erfolgt auch bei Referenztypen mit expliziter **null**-Zulässigkeit (deklariert per **?**-Suffix am Ende der Typbezeichnung). Wie das folgende Beispiel zeigt, wird in diesem Fall eine **null**-Zuweisung kommentarlos akzeptiert, beim Dereferenzieren aber gewarnt, wenn der Wert **null** nicht ausgeschlossen ist:

```
string? sn = null;
Console.WriteLine(sn.Length);
```

Eine garantiert unbegründete Warnung des Compilers vor einer möglichen **null**-Zuweisung oder **null**-Dereferenzierung lässt sich mit dem **null**-Toleranzoperator (engl.: *null-forgiving operator*) unterdrücken, der durch ein Ausrufezeichen notiert wird, z. B.:

```
int len = s!.Length;
```

Per Voreinstellungen belässt es der Compiler bei Warnungen und übersetzt einen Quellcode trotz vorhandener Warnungen. Das tut er schon deswegen, weil ohne die Unterstützung durch die Entwickler (per **null**-Toleranzoperator oder die Vergabe von Attributen, siehe Abschnitt 15.1.4) viele Warnungen unbegründet sind.

Man darf sich keinesfalls darauf verlassen, dass **NullReferenceException** - Probleme durch die Nullzustandsanalyse des Compilers verhindert werden. Auch die erfolgreiche Elimination sämtlicher Warnungen garantiert keine **null**-Sicherheit.

Haben z. B. die Elemente eines neu erzeugten Arrays einen **null**-unzulässigen Referenztyp, dann verzichtet der Compiler auf eine Warnung:

```
var sar = new string[3];
Console.WriteLine(sar[1].Length); // Keine Warnung im Nullable-Kontext!
```

Auch beim Array mit den Befehlszeilenargumenten unterbleibt eine Warnung im Nullable-Kontext, z. B.:

```
static void Main(string[] args) {
   Console.WriteLine($"{args[0]} hat {args[0].Length} Buchstaben"); // Keine Warnung!
}
```

Zu diesem Verhalten haben sich die C# - Designer entschlossen, weil die meisten Arrays von Warnungen betroffen wären, und keine Maßnahme zur Vermeidung des Problems verfügbar ist (Torgersen 2017).

Bei Strukturdefinitionen verzichtet der Compiler auf Warnungen, wenn Felder mit einem **null**-unzulässigen Referenztyp keine Initialisierung erhalten, z. B.:

Warnung bei Klassen	*Keine* Warnung bei Strukturen

`public class Person {`
`  public string Name;`
`}`
	`public struct Person {`
	`  public string Name;`
	`}`

Beim Zugriff auf ein nicht-initialisiertes Strukturfeld erscheint aber eine Warnung, z. B.:

```
var pstruct = new Person();
Console.WriteLine(pstruct.Name.ToUpper());
string s = pstruct.Name.ToUpper();
```

Hier liegt also keine ernsthafte Lücke der Nullzustandsanalyse vor, sondern nur ein irritierendes Verhalten.

Aus dem Bemühen um eine möglichst kleine Zahl von **null**-Warnungen resultierte wohl auch die Entscheidung, vor dem **maybe-null** – Status eines Ausdrucks nicht wiederholt zu warnen. Im obigen Beispiel bleibt daher die dritte Zeile trotz **null**-Dereferenzierung ohne Warnung.

Andererseits produziert die Nullzustandsanalyse auch falsch-positive Urteile mit entsprechenden Warnungen, z. B.:

```
public class Tiger {
    public string Name { get; set; }
    public Tiger(string? s) {
        Name = CheckNotNull(s) ? s : "Unknown";
    }
    static bool CheckNotNull(string? s) { return s != null; }
}
```

Zur Beseitigung dieser Warnungen eignen sich:

- der **null**-Toleranzoperator, z. B.:
  ```
  public Tiger(string? s) {
      Name = (CheckNotNull(s) ? s : "Unknown")!;
  }
  ```
- Attribute mit Informationen zum Verhalten von Methoden (siehe Abschnitt 15.1.4).

In der Online-Dokumentation sind mehr als 50 Nullable-Warnungen des Compilers zusammen mit Tipps zur Beseitigung beschrieben.[1] Auch nach der Beseitigung aller Warnungen kann eine **NullReferenceExeption** nicht ausgeschlossen werden. Trotzdem sollte der Nullable-Kontext (mit expliziter Deklaration der **null** - (Un)zulässigkeit von Referenztypen und Nullable-Warnungen) für neue Projekte (wie seit .NET 6 voreingestellt) aktiviert sein:

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-messages/nullable-warnings

- Die Nullzustandsanalyse verbessert die Code-Qualität.
- Das (bei Referenz- und Werttypen verwendbare) **?**-Suffix macht den Quellcode informativer. Im folgenden Beispiel aus Torgersen (2017) kommt klar zum Ausdruck, dass für jede Person ein `FirstName` und ein `LastName` erwartet wird, während ein `MiddleName` nicht vorhanden sein muss:

```
public class Person {
    public string FirstName;   // not-null
    public string? MiddleName; // maybe-null
    public string LastName;    // not-null
    . . .
}
```

  Vor C# 8 konnten Entwickler ihre Intentionen nicht so klar für andere Teammitglieder und den Compiler formulieren.

Über eine Projektkonfiguration kann man dafür sorgen, dass bestimmte Warnungen das Gewicht von Fehlern erhalten und somit die Übersetzung des Quellcodes verhindern (siehe Abschnitt 15.1.3.1).

15.1.3 Nullable-Kontexte für Referenztypen

Seit .NET 6 ist der Nullable-Kontext bei neuen Projekten aktiviert, doch kann diese Einstellung einfach und flexibel geändert werden.

15.1.3.1 Kontextvarianten

Wie man für ein ganzes Projekt über das folgende **Nullable**-Element

```
<Nullable>enable</Nullable>
```

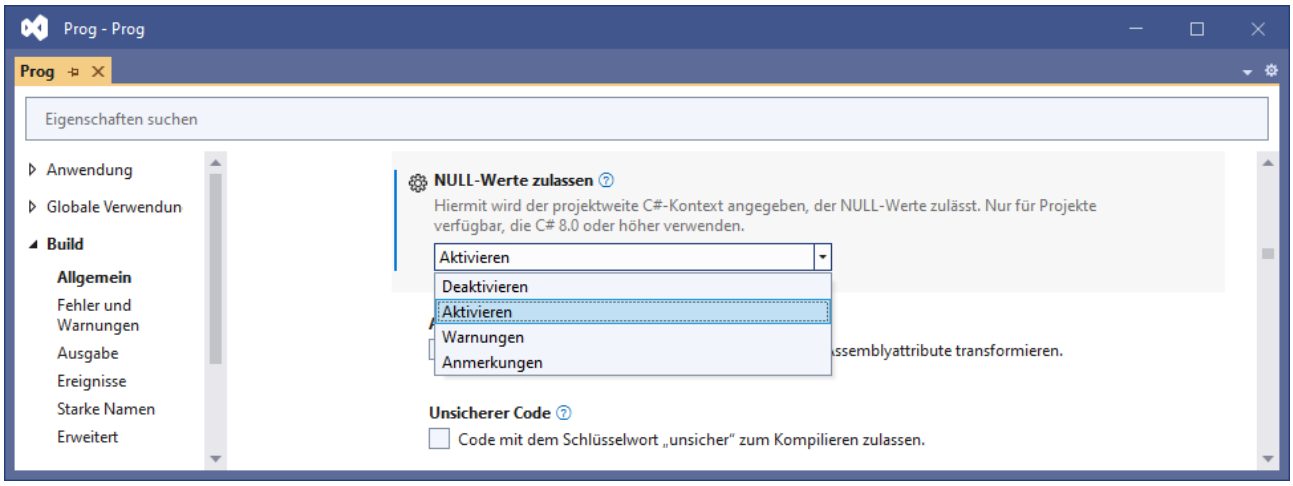
in der Projektdatei den Nullable-Kontext aktiviert, war schon im Abschnitt 15.1.1 zu sehen. Das hat im Detail die folgenden Konsequenzen:

- Für Referenztypen besteht **null**-Unzulässigkeit.
- Mit dem Typbezeichnungs-Suffix **?** erhält man die **null**-zulässige Variante zu einem Referenztyp.
- Der Compiler führt die Nullzustandsanalyse durch und formuliert **null**-Referenz – Warnungen (vor der **null**-Zuweisung und vor der **null**-Dereferenzierung).
- Durch den **null**-Toleranzoperator **!** (siehe Abschnitt 15.1.2) und durch Attribute zum Verhalten von Methoden (siehe Abschnitt 15.1.4) lassen sich die Warnungen vor potenziellen **null**-Zuweisungen oder **null**-Dereferenzierungen unterdrücken.

Statt die Projektdatei direkt zu editieren, kann man im Visual Studio nach

Projekt > Eigenschaften > Build > Allgemein

den Nullable-Kontext einstellen:



Den Ausgangszustand per **Nullable**-Element in der Projektdatei

```
<Nullable>disable</Nullable>
```

herzustellen, ist nicht erforderlich, weil er auch bei Abwesenheit eines **Nullable**-Elements besteht. Über das im Abschnitt 15.1.3.2 vorzustellende Präprozessor-Kommando **#nullable** kann der Ausgangszustand aber für einen begrenzten Quellcodebereich (abweichend vom Projektstandard) hergestellt werden.

Im Ausgangszustand gilt (wie vor C# 8):

- Für Referenztypen besteht **null**-Zulässigkeit.
- Das Typbezeichnungs-Suffix **?** ist überflüssig und produziert ab C# 8 bei Gebrauch die Warnung CS8632:

  Die Anmerkung für Nullable-Verweistypen darf nur in Code innerhalb eines #nullable-Anmerkungskontexts verwendet werden.
- Der Compiler führt *keine* Nullzustandsanalyse durch und formuliert keine **null**-Referenz – Warnungen (weder vor der **null**-Zuweisung, noch vor der **null**-Dereferenzierung).
- Der **null**-Toleranzoperator **!** ist erlaubt, aber wirkungslos.

Im Rahmen der Warnung CS8632 tauchte eben der Begriff **Anmerkungskontext** auf. Dieser Kontext wird z. B. mit dem folgenden **Nullable**-Element in der Projektdatei

```
<Nullable>annotations</Nullable>
```

mit den folgenden Konsequenzen aktiviert:

- Das Typbezeichnungs-Suffix **?** darf verwendet werden, um einen Referenztyp mit expliziter **null**-Zulässigkeit auszuweisen.
- Durch eine Referenztypbezeichnung ohne **?**-Suffix drückt ein Entwickler seine Intention aus, dass der Wert **null** nicht zulässig sein soll. Das bleibt ohne Effekt auf das Verhalten des Compilers.
- Der Compiler führt *keine* Nullzustandsanalyse durch und formuliert keine **null**-Referenz – Warnungen (weder vor der **null**-Zuweisung, noch vor der **null**-Dereferenzierung).
- Der **null**-Toleranzoperator **!** ist erlaubt, aber wirkungslos.

Der Anmerkungs- oder Annotationskontext unterscheidet sich vom Ausgangszustand (ohne **Nullable**-Element in der Projektdatei) lediglich darin, dass man das (als *Annotation* oder *Anmerkung* bezeichnete) Typnamenssuffix **?** ohne Compiler-Warnung verwenden darf.

Schließlich kann z. B. mit dem folgenden **Nullable**-Element in der Projektdatei

```
<Nullable>warnings</Nullable>
```

noch ein Nullable-Kontext aktiviert werden, der sich auf eine Nullzustandsanalyse durch den Compiler beschränkt und daher als **Warnungskontext** bezeichnet wird:

- Für Referenztypen besteht **null**-Zulässigkeit.
- Das Typbezeichnungs-Suffix **?** produziert bei Gebrauch die Warnung CS8632:

  Die Anmerkung für Nullable-Verweistypen darf nur in Code innerhalb eines #nullable-Anmerkungskontexts verwendet werden.

  Es ist aber nicht nutzlos, sondern sorgt für begründete **null**-Dereferenzierungswarnungen, die ansonsten unterbleiben, z. B.:

```
class Prog {
    public string? Name;
    public void WriteName() => Console.WriteLine(Name.ToUpper());
}
```

- Der Compiler führt eine Nullzustandsanalyse durch, warnt aber nur vor der **null**-Dereferenzierung.
- Durch den **null**-Toleranzoperator **!** lässt sich die Warnung vor einer potenziellen **null**-Dereferenzierung unterdrücken.

Enthält die Projektdatei mehrere **Nullable**-Elemente, dann ist nur das letzte wirksam.

Gerät ohne Nullable-Kontext erstellter Bestandscode in einen vollständig aktivierten Nullable-Kontext, dann erscheinen in der Regel irritierend viele Warnungen in Form von …

- grün unterschlängelten Passagen im Quellcodeeditor
- bzw. Warnmeldungen beim Übersetzen.

Man kann nun …

- den vollständig aktivierten Nullable-Kontext beibehalten und die Warnungen durch die Ergänzung von Attributen beseitigen, was z. B. in der BCL geschehen ist,
- oder den Anmerkungskontext (z. B. per Projektdatei) einstellen
  Dann unterbleiben die Nullable-Warnungen, die durch ihre Fülle die Beachtung andere Warnungen erschweren, während die Deklaration von explizit **null**-zulässigen Referenzdatentypen per **?**-Namenssuffix erlaubt ist.
- oder auf die Unterstützung zur Vermeidung von **null**-Referenz – Fehlern komplett verzichten und den Nullable-Ausgangszustand (wie vor C# 8) einstellen.

Über das Element **WarningsAsErrors** in der Projektkonfigurationsdatei ist zu erreichen, dass bestimmte Warnungen das Gewicht von Fehlern erhalten und somit die Übersetzung des Quellcodes verhindern, z. B.:

```
<PropertyGroup>
  <Nullable>enable</Nullable>
  <WarningsAsErrors>CS8600;CS8601;CS8602;CS8618</WarningsAsErrors>
  . . .
</PropertyGroup>
```

Nun können die vom Compiler erkannten **null**-Risiken nicht mehr ignoriert werden, z. B.:

```
class Prog {
    string s;
    static void Main() {
        var p = new Prog();
        string? sl = null;
        p.s = sl;
        int len = p.s.Length;
        Console.WriteLine(len);
    }
}
```

15.1.3.2 Kontextverwaltung durch Präprozessor-Kommandos

Statt einen Nullable-Kontext projektglobal über ein Element in der Projektdatei einzustellen, kann man über Präprozessor-Kommandos Einstellungen für begrenzte Quellcodebereiche vornehmen. Das Präprozessor-Kommando **#nullable** ist wirksam für die nachfolgenden Quellcodezeilen, z. B.:

```
#nullable enable warnings
```

Es dominiert ggf. die Projekteinstellung und bleibt gültig bis zu einem abweichenden **#nullable** -Kommando. Ein Präprozessor-Kommando am Dateianfang, dem im weiteren Verlauf nicht widersprochen wird, legt also z. B. den Nullable-Kontext für die gesamte Quellcodedatei fest.

Das Präprozessor-Kommando **#nullable** kennt die folgenden Varianten:

- **#nullable enable**
  Der Annotationskontext und der Warnungskontext werden aktiviert.
- **#nullable disable**
  Der Annotationskontext und der Warnungskontext werden deaktiviert.
- **#nullable restore**
  Die Projekteinstellung (per **Nullable**-Element in der Projektdatei) wird restauriert.
- **#nullable enable annotations,**
  #nullable disable annotations
  #nullable restore annotations
  Die Annotationen werden aktiviert, deaktiviert oder auf die Projekteinstellung gesetzt.
- **#nullable enable warnings,**
  #nullable disable warnings
  #nullable restore warnings
  Die Warnungen (vor der **null**-Dereferenzierung) werden aktiviert, deaktiviert oder auf die Projekteinstellung gesetzt.

Während über das **Nullable**-Element in der Projektdatei eine von den sich ausschließenden Alternativen **disable**, **enable**, **annotations** und **warnings** zu wählen ist, wirken die Präprozessor-Kommandos kumulativ, sodass z. B. zunächst mit

```
#nullable enable annotations
```

die Anmerkungen erlaubt und später zusätzlich mit

```
#nullable enable warnings
```

die Warnungen aktiviert werden können, wobei dann der Nullable-Kontext wie bei

```
#nullable enable
```

vollständig aktiviert ist.[1]

15.1.4 Attribute zur Unterstützung der Nullzustandsanalyse

Damit aus der mit C# 8 eingeführten statischen Nullzustandsanalyse trotz prinzipieller Unzulänglichkeiten eine erfolgreiche Software-Technik wird, müssen die Entwickler den Compiler in vielen Fällen über die Wirkungsweise von Methoden und über die Nullzustände von Ausdrücken informieren:

- Der Compiler führt bei der Nullzustandsanalyse für aufgerufene Methoden keine Ablaufverfolgung durch und benötigt daher Unterstützung, z. B. in diesen Fällen:

---

[1] Im vollständig aktivierten Nullable-Kontext wird vor der **null**-Zuweisung und vor der **null**-Dereferenzierung gewarnt, während das Präprozessor-Kommando

```
#nullable enable warnings
```

allein lediglich Warnungen vor der **null**-Dereferenzierung bewirkt.

  - Manche Methoden führen **null**-Tests für Parameterobjekte durch und informieren durch eine **bool**-Rückgabe über das Ergebnis.
  - Die Rückgabe mancher Methoden hat den Status **not-null** oder **maybe-null** in Abhängigkeit vom Nullzustand eines Parameterobjekts.
- Den Nullzustand mancher Ausdrücke (**not-null** oder **maybe-null**) kann der Entwickler besser beurteilen als der Compiler.

Zur Nullzustands-Schützenhilfe für den Compiler taugen neben dem **null**-Toleranzoperator (siehe Abschnitt 15.1.2) vor allem Attribute aus dem Namensraum **System.Diagnostics.CodeAnalysis**, die an Methoden, Parameter, Rückgabewerte usw. geheftet werden. Im BCL-API sind diese Attribute seit .NET 5 vorhanden.

15.1.4.1 NotNullWhen

Wenn eine Methode für ihren Parameter mit Referenztyp den **null**-Status prüft und per **bool**-Rückgabe meldet, dann kann über das Parameter-Attribut **NotNullWhen** vereinbart werden, dass der zur Attribut-Initialisierung verwendete **bool**-Wert für den Methodenparameter den Status **not-null** signalisiert. Im folgenden Beispiel[1]

```
using System.Diagnostics.CodeAnalysis;

var t = new Tiger(null);
Console.WriteLine($"Der Name {t.Name} hat {t.Name.Length} Zeichen");

public class Tiger {
    public string Name { get; set; }

    public Tiger(string? s) {
        Name = CheckNotNull(s) ? s : "Unknown";
    }

    static bool CheckNotNull([NotNullWhen(true)] string? s) {
        return s != null;
    }
}
```

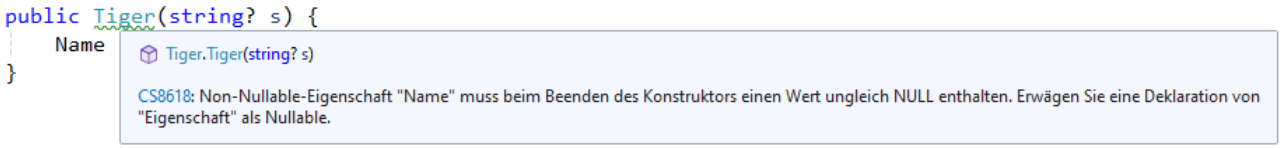
verhindert das mit dem Wert **true** initialisierte Attribut **NotNullWhen** zum Parameter der Methode `CheckNotNull()` die Compiler-Warnung CS8618

> Mögliche Nullzuweisung

für die Wertzuweisung an die Eigenschaft `Name` sowie die Compiler-Warnung CS8618

> Non-Nullable-Eigenschaft "Name" muss beim Beenden des Konstruktors einen Wert ungleich NULL erhalten.

für den Konstruktor:



Gegeben die Definition der Methode `CheckNotNull()` handelt es sich um falsch-positive Warnungen.

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\C# für Fortgeschrittene\Nullable-Kontext für Referenztypen\NotNullWhen

Der Compiler kann das Verhalten der Methode `CheckNotNull()` nicht analysieren und vertraut blind dem Nullable-Attribut, sodass die **NullReferenceException**-Prävention durch ein fehlerhaftes Nullable-Attribute ausgehebelt wird, z. B.:

```
static bool CheckNotNull([NotNullWhen(true)] string? s) {
    return true;
}
```

15.1.4.2 NotNullIfNotNull

Das an eine Methodenrückgabe mit Referenztyp geheftete Attribut **NotNullIfNotNull** garantiert dem Compiler eine **not-null** - Rückgabe, sofern ein per Attribut-Argument benannter Methodenparameter von **null** verschieden ist. Im folgenden Beispiel[1]

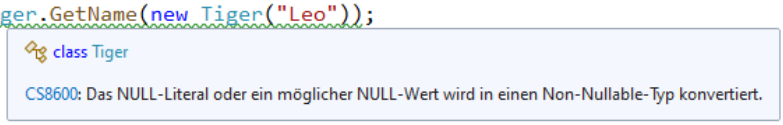
```
using System.Diagnostics.CodeAnalysis;

string tn = Tiger.GetName(new Tiger("Leo"));
Console.WriteLine(tn);

public class Tiger {
    public string Name { get; set; }

    public Tiger(string s) {
        Name = s;
    }

    [return: NotNullIfNotNull(nameof(tiger))]
    static public string? GetName(Tiger? tiger) {
        return tiger != null ? tiger.Name : null;
    }
}
```

verhindert das Attribut für den Rückgabewert der Methode `GetName()` die Compiler-Warnung CS8600

Das NULL-Literal oder ein möglicher NULL-Wert wird in einen Non-Nullable-Typ konvertiert.

für die folgende Wertzuweisung:

```
string tn = Tiger.GetName(new Tiger("Leo"));
```



Wie sich die **return**-Anweisung in `GetName()`

```
return tiger != null ? tiger.Name : null;
```

mit dem **null**-bedingten Operator vereinfachen lässt,

```
return tiger?.Name;
```

ist bald im Abschnitt 15.1.5.2 zu erfahren.

---

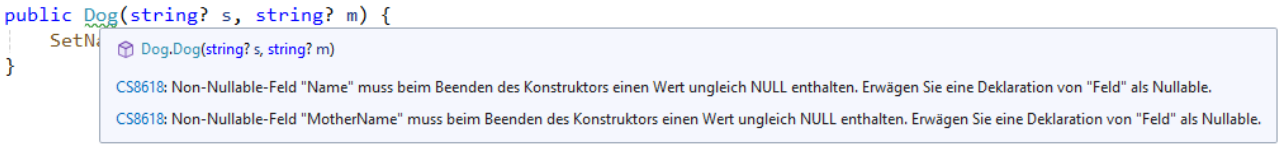
[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\C# für Fortgeschrittene\Nullable-Kontext für Referenztypen\NotNullIfNotNull

15.1.4.3 MemberNotNull und MemberNotNullWhen

Das an eine zur Initialisierung von Member-Objekten dienende Methode geheftete Attribut **MemberNotNull** versichert dem Compiler, dass die per **params**-Argument vom Typ **String[]** benannten Member-Objekte durch die Methode initialisiert werden. Im folgenden Beispiel[1]

```
using System.Diagnostics.CodeAnalysis;

var d = new Dog(null, null);
Console.WriteLine($"Der Name des Hundes hat {d.Name.Length} Zeichen.");

public class Dog {
    public string Name;
    public string MotherName;

    public Dog(string? s, string? m) {
        SetName(s, m);
    }

    [MemberNotNull(nameof(Name), nameof(MotherName))]
    private void SetName(string? name, string? mname) {
        Name = name ?? "Unknown";
        MotherName = mname ?? "Unknown";
    }
}
```

verhindert das Attribut zwei Compiler-Warnungen CS8618 für den Konstruktor:



Das Attribut **MemberNotNullWhen** eignet sich für eine Methode, die ein Member-Objekt (oder mehrere per **params**-Parameter benannte Member-Objekte) nur unter einer Bedingung initialisiert und per **bool**-Parameter über die (verweigerte) Erledigung berichtet. Im folgenden, etwas verspielten Beispiel wird eine Zeichenfolge nur dann als `Name` akzeptiert, wenn es sich um ein Palindrom handelt (siehe Abschnitt 6.6):

```
[MemberNotNullWhen(true, nameof(Name))]
private bool SetName(string name) {
    int len = name.Length;
    var sb = new StringBuilder(len);
    for (int i = 0; i < len; i++)
        sb.Append(name[len - i - 1]);
    if (sb.ToString().ToUpper() == name.ToUpper()) {
        Name = name;
        return true;
    }
    return false;
}
```

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\C# für Fortgeschrittene\Nullable-Kontext für Referenztypen\MemberNotNull

15.1.4.4 AllowNull

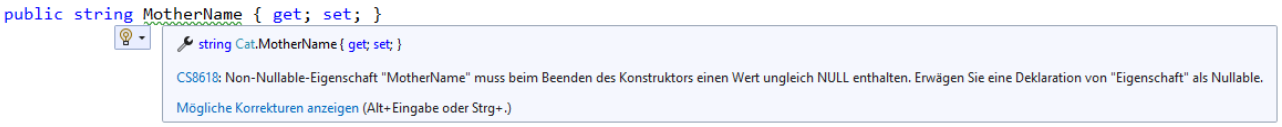
Das Attribut **AllowNull** kann an Felder und Eigenschaften geheftet werden und ist äquivalent zum **?**-Suffix für den Typnamen. Im folgenden Beispiel[1]

```
using System.Diagnostics.CodeAnalysis;

public class Cat {
    string name = "Unknown";
    public string Name {
        get { return name; }
        set { if (value != null) name = value; }
    }

    [AllowNull]
    public string MotherName { get; set; }

    [AllowNull]
    public string FatherName;
}
```

verhindert das Attribut die Compiler-Warnung CS8618

> Non-Nullable-Eigenschaft "MotherName" muss beim Beenden des Konstruktors einen Wert ungleich NULL enthalten.

für die Eigenschaft `MotherName`.



15.1.5 null-Operatoren

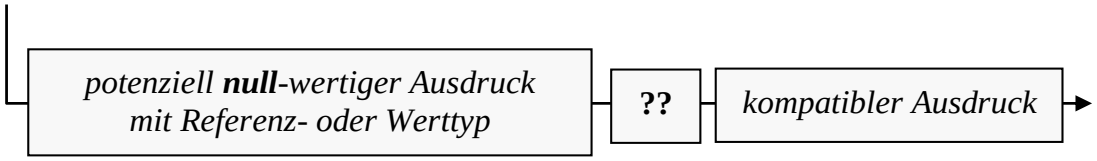
C# kennt vier Operatoren zur Behandlung von **null**-Werten, von denen bisher drei im Manuskript behandelt wurden.

15.1.5.1 Bereits behandelte null-Operatoren

Die im Manuskript bereits behandelten **null**-Operatoren sollen nochmal in Erinnerung gebracht werden:

- **null**-Koaleszenzoperator
  Mit dem null-Koaleszenzoperator, der durch zwei Fragezeichen ausgedrückt wird, lässt sich die Zuweisung eines potenziell **null**-wertigen Ausdrucks an eine Variable mit **null**-unzulässigem Typ samt Ersatzwert für die kritische **null**-Situation bequem formulieren (siehe Abschnitt 8.3):

  null-Koaleszenzoperator

  Ist der linke **??** - Operand von **null** verschieden, dann liefert er den Wert des Ausdrucks. Anderenfalls kommt der rechte Operand zum Zug, der kompatibel und von **null** verschieden sein muss, z. B.:

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\C# für Fortgeschrittene\Nullable-Kontext für Referenztypen\AllowNull

```
var p = new Person("Otto", null, "Rempremerding");
string mn = p.MiddleName ?? "<Kein weiterer Vorname>";

public class Person {
    public string FirstName;
    public string? MiddleName;
    public string LastName;
    public Person(string firstName, string? middleName, string lastName) {
        FirstName = firstName; MiddleName = middleName; LastName = lastName;
    }
}
```

Die Anweisung

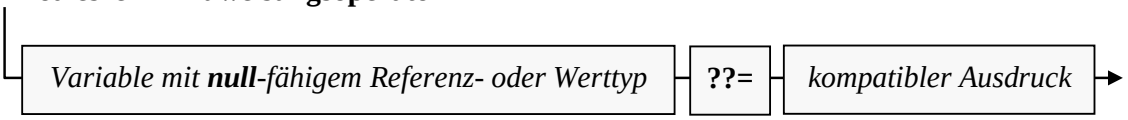
```
string mn = p.MiddleName ?? "<Kein weiterer Vorname>";
```

ist äquivalent zu:

```
string mn = p.MiddleName != null ? p.MiddleName : "<Kein weiterer Vorname>";
```

- **null**-Koaleszenz - Zuweisungsoperator
  Durch den null-Koaleszenz - Zuweisungsoperator (siehe Abschnitt 8.3)

  null-Koaleszenz - Zuweisungsoperator

  wird einer Variablen mit einem **null**-fähigen Typ genau dann der Wert des kompatiblen rechten Ausdrucks zugewiesen, wenn sie aktuell den Wert **null** besitzt, z. B.:

Quellcode	Ausgabe

`string? s = null;`
`s ??= "undefiniert";`
`Console.WriteLine(s);`
	`undefiniert`

  Die Anweisung

```
s ??= "undefiniert";
```

  ist äquivalent zu:

```
if (s == null)
    s = "undefiniert";
```

- **null**-Toleranzoperator
  Bei der Nullzustandsanalyse in einem Nullable-Kontext (siehe Abschnitt 15.1.2) lässt sich der **null**-Toleranzoperator (engl.: *null-forgiving operator*), der durch ein Ausrufezeichen notiert wird, dazu verwenden, um eine unbegründete Compiler-Warnung vor einer **null**-Zuweisung oder einer **null**-Dereferenzierung zu unterdrücken, z. B.:

```
int len = s!.Length;
```

15.1.5.2 null-bedingter Operator

Vor dem Zugriff auf ein Instanzmitglied (z. B. Methode, Feld oder Array-Element) muss zur Vermeidung einer **NullReferenceException** für die Instanz (mit einem Referenztyp oder einem **null**-fähigen Werttyp) im Zweifelsfall ein **null**-Test durchgeführt werden, was den Quellcode durch häufig auftretende Routinepassagen belastet. Im folgenden Beispiel soll aus einem **String**-Array die Länge des Elements mit dem Index 1 ermittelt werden, was eine doppelte Existenzprüfung erfordert (für den Array und für das Element an der Position 1):

```
string[]? ass = null;
 . . .
int len1 = (ass != null && ass[1] != null) ? ass[1].Length : -1;
Console.WriteLine($"Länge: {len1}");
```

Der in C# 6.0 eingeführte **null**-bedingte Operator (engl.: *null-conditional operator*) vereinfacht den Instanzzugriff mit vorherigem **null**-Test. Für den Member- bzw. Indexzugriff sind zwei leicht verschiedene Syntaxvarianten zu unterscheiden:[1]

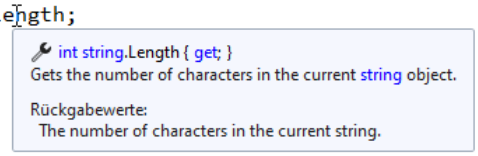
- *instance***?.***member*
  Als Datentyp hat der Ausdruck den nötigenfalls **null**-erweiterten Membertyp, d. h. (siehe ECMA 2022, Abschnitt 11.7.7):
  - Hat *member* den **null**-unzulässigen Werttyp T, dann hat *instance***?.***member* den Typ T?.
  - Hat *member* den **null**-zulässigen Typ T, dann hat *instance***?.***member* ebenfalls den Typ T.

  Hat *instance* den Wert **null**, dann liefert der Ausdruck den Wert **null** (Referenz auf Nichts bei einer Klasse oder undefinierte Ausprägung bei einem Werttyp). Im folgenden Beispiel wird die **Length**-Eigenschaft einer **String**-Variablen mit dem Wert **null** angefragt, und es resultiert der Typ **int?**:
  ```
  string? s = null;
  int? sl = s?.Length;
  ```
  Während die **String**-Eigenschaft **Length** den Typ **int** besitzt, hat `sl` den Typ **int?**.[2]
  Im nächsten Beispiel besitzt die angefragte Instanz einen **null**-fähigen Strukturtyp:
  ```
  Point? pn = null;
  int? i1 = pn?.X;

  public struct Point {
      public int X;
      public int Y;
  }
  ```
  Der **null**-bedingte Operator vereinfacht die **null**-gesicherte Koordinatenabfrage:
  ```
  int? i2 = pn.HasValue ? pn.Value.X : null;
  ```
- *instance***?[***index***]**
  Als Datentyp hat der Ausdruck den nötigenfalls **null**-erweiterten Elementtyp, d. h. (siehe ECMA 2022, Abschnitt 11.7.7):
  - Haben die Elemente den **null**-unzulässigen Werttyp T, dann hat *instance***?[***index***]** den Typ T?.
  - Haben die Elemente den **null**-zulässigen T, dann hat *instance***?[***index***]** ebenfalls den Typ T.

  Hat *instance* den Wert **null**, dann liefert der Ausdruck den Wert **null** (Referenz auf Nichts bei einer Klasse oder undefinierte Ausprägung bei einem Werttyp). Im folgenden Beispiel mit einem Indexzugriff auf die **char**-Elemente eines **null**-zulässigen **String**-Objekts hat der **null**-bedingte Ausdruck den Typ **char?**:
  ```
  string? sn = null;
  char? cn = sn?[0];
  ```

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators#null-conditional-operators--and-

[2] Die **String**-Eigenschaft **Length** behält trotz **?**-Annotation den Typ **int**:
```
string? s = null;
int? sl = s?.Length;
```


Im zweiten Beispiel mit Array-Elementen vom Typ **int** resultiert der Typ **int?**:

```
int[]? ai = null;
int? ai1 = ai?[1];
```

Der linke Operand des **null**-bedingten Operators muss einen **null**-fähigen Typ haben, also einen Referenztyp oder einen Strukturtyp in **Nullable<T>** - Verpackung (vgl. Abschnitt 8.3). Ein **null**-bedingter Zugriff unterscheidet sich vom normalen Member- bzw. Indexzugriff nur dann, wenn die angefragte Variable den Wert **null** besitzt: Dann wird keine **NullReferenceException** geworfen, sondern der Wert **null** abgeliefert.

Das Einstiegsbeispiel lässt sich mit zwei **null**-bedingten Operatoren kürzer formulieren:

```
string[]? ass = null;
. . .
int len1 = ass?[1]?.Length ?? -1;
Console.WriteLine($"Länge: {len1}");
```

Damit ein Ergebnis mit dem Werttyp **int** resultiert, kommt im Beispiel zusätzlich der **null**-Koaleszenzoperator zum Einsatz (siehe Abschnitt 15.1.5.1).

Auch beim Aufruf eines Delegatenobjekts (vgl. Kapitel 10) lässt sich mit dem **null**-bedingten Operator die Existenzprüfung vereinfachen, z. B.:

```
delegate void DemoGate(int w);
 . . .
DemoGate? demoVar = null;
demoVar?.Invoke(2);
```

Dabei ist ein expliziter **Invoke()** - Aufruf erforderlich (vgl. Abschnitt 10.1).

Der **null**-bedingte Operator ist nur auf den allerersten Blick mit der Deklaration eines **null**-zulässigen Typs zu verwechseln:

- Beim **null**-bedingten Operator steht das Fragezeichen hinter einem *Variablennamen*, dem ein Punktoperator oder ein Elementzugriffsoperator folgt.
- Bei der Deklaration eines **null**-zulässigen Typs steht das Fragezeichen am Ende eines *Typnamens*.

15.2 Musterabgleich

Seit der C# - Version 7 sind Kompetenzen zum Musterabgleich (engl.: *pattern matching*) im Sprachumfang enthalten. In der **switch**-Anweisung wurde die seit C# 1 mögliche Detektion von Fällen über konstante Werte (nun als *konstantes Muster* bezeichnet) ergänzt durch die Falldetektion über sogenannte *Typmuster* (siehe Abschnitt 15.2.3). In den C# - Versionen 8 und 9 sind weitere, gleich zu beschreibende Optionen zum Musterabgleich dazugekommen, die in verschiedenen Kontexten zur Verfügung stehen:

- in der **switch**-Anweisung
- im **switch**-Ausdruck
- im **is**-Ausdruck

Damit sich nicht zu viele abstrakte Begriffe am Beginn des Abschnitts häufen und den Einstieg erschweren, präsentieren wir vorab ein Beispiel aus dem Abschnitt 15.2.3. Hier wird eine Flächenberechnung für verschiedene geometrische Figuren nach einer typbasierten Fallunterscheidung vorgenommen:

```
static double Flaeche(object figur) {
    return figur switch {
        Rechteck r => r.Breite * r.Hoehe,
        Kreis k => Math.PI * k.Radius * k.Radius,
        _ => 0.0
    };
}
```

Microsoft möchte durch die Kompetenzen zum Musterabgleich die Eignung von C# für Aufgabenstellungen verbessern, die durch eine partielle Trennung von Daten und Funktionalität gekennzeichnet sind:

- Es sind unterschiedliche Typen zu verarbeiten, die nicht zu einer gemeinsamen Hierarchie gehören, sodass z. B. eine Lösung über eine polymorph implementierte, abstrakte Basisklassenmethode ausscheidet (vgl. Abschnitt 7.9).
- Die zu realisierende Funktionalität gehört nicht in den primären Verantwortungsbereich der Klassen.

In einem Tutorial zum Musterabgleich beschreibt Microsoft ein Beispiel, das für unterschiedliche Fahrzeugkategorien (private PKWs, Taxis, Busse, Lieferwagen) die Mautberechnung in Abhängigkeit von unterschiedlichen, nicht für alle Klassen identisch anzuwendenden Kriterien erfordert (z. B. Anteil der besetzten Plätze, Gewicht).[1] Die Fahrzeugkategorien werden durch Klassen modelliert, die außer **System.Object** keine gemeinsamem Basisklasse besitzen, und das lässt sich mangels Kontrolle über den Quellcode nicht ändern. Außerdem gehört die von externen Bedingungen (z. B. aktuell befahrener Stadtteil) abhängige Mautberechnung nicht zu den Kernaufgaben der Klassen.

Im Beispiel bietet es sich an, …

- die Mautberechnung in eine Hilfsklasse zu verlagern
- und die erforderlichen Fallunterscheidungen per Musterabgleich vorzunehmen.

Die folgende Aussage von der eben zitierten Webseite fasst Microsofts Einsatzempfehlungen für die Musterabgleichtechnik prägnant zusammen:

> Pattern matching makes some types of code more readable and offers an alternative to object-oriented techniques when you can't add code to your classes.

Hier werden als wesentliche Einsatzzwecke genannt:

- Lesbarkeit des Quellcodes verbessern (z. B. durch die Vereinfachung von Fallunterscheidungen)
- Alternative Lösungen für Einsatzfelder der Polymorphie ermöglichen

Von den beiden **switch**-Optionen in C# (Ausdruck und Anweisung) ist vor allem der **switch**-*Ausdruck* als Einsatzort für Musterabgleiche geeignet, weil …

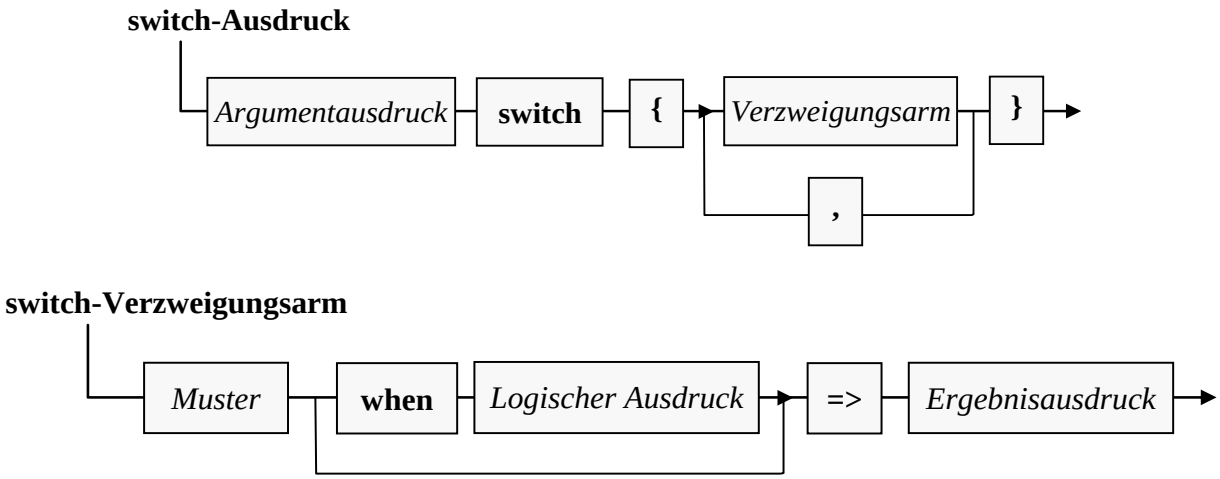
- man oft nur *einen* Ergebniswert in Abhängigkeit von der Fallzugehörigkeit benötigt,
- und die **switch**-Anweisung einen größeren syntaktischen Aufwand erfordert.

Daher tauchen in den Beispielen des aktuellen Abschnitts (neben **is**-Ausdrücken) nur **switch**-*Ausdrücke* auf.

Der schon im Abschnitt 4.7.2.4 in Grundzügen beschriebene **switch**-Ausdruck hat den folgenden Aufbau:

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/pattern-matching



Im Ergebnisausdruck kann auch eine Ausnahme geworfen werden, weil C# seit der Version 7.0 neben der **throw**-Anweisung auch den **throw**-Ausdruck kennt (vgl. Abschnitt 13.5), z. B.:

```
null => throw new ArgumentException(message: "No person found")
```

Wegen der Beteiligung von Musterabgleichen können *mehrere* Verzweigungsarme zutreffen, wobei dann der erste zutreffende das Rennen macht. Der Compiler bemüht sich um eine sinnvolle Reihenfolge der Verzweigungsarme und verweigert die Übersetzung, wenn er einen nicht erreichbaren Arm feststellt. Diese Diagnose wird allerdings durch die Anwesenheit von **when**-Klauseln verhindert, sodass die Reihenfolge der Verzweigungsarme in dieser Situation sorgfältig zu wählen ist.

Anschließend werden die in C# unterstützten Muster beschrieben.

15.2.1 Konstantenmuster

Das seit C# 1.0 im Rahmen der **switch**-Anweisung erlaubte *Konstantenmuster* (engl. *constant pattern*), das aus einem schon zur Übersetzungszeit feststehenden Wert (Literal oder konstante Variable) besteht, kann auch in einem **switch**-Ausdruck verwendet werden. Das folgende Beispiel wurde in ähnlicher Form schon im Abschnitt 4.7.2.4 (mit einer Kurzbeschreibung des in C# 8 eingeführten **switch**-Ausdrucks) vorgestellt:[1]

Quellcode	Ausgabe

`int charCode = 3;`
`string charLabel = charCode switch {`
`  1 => "melancholisch",`
`  2 => "cholerisch",`
`  3 => "phlegmatisch",`
`  4 => "sanguinisch"`
`};`
`Console.WriteLine($"Charakter: {charLabel}");`
	`Charakter: phlegmatisch`

Ein Konstantenmuster kann auf den Typ des Argumentausdrucks (im Beispiel: **int**) angewendet werden, wenn die Konstanten (wie im Beispiel) denselben Typ besitzen oder implizit in diesen Typ konvertierbar sind.[2] Bei einem Argumentausdruck vom Typ **Object** ist die implizite Typanpassung immer möglich, was zusammen mit dem **is**-Operator demonstriert werden soll:

---

[1] Der angemessene und im Abschnitt 15.2.2 behandelte residuale Unterstrich-Verzweigungsarm wird hier bewusst weggelassen, sodass ein Argument außerhalb der Menge {1, 2, 3, 4} zu einem Ausnahmefehler führt.

[2] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/patterns

Quellcode	Ausgabe

`object obj = 13;`
`if (obj is 13)`
`    Console.WriteLine("Yes!");`
	`Yes!`

Der Ausdruck

```
obj is 13
```

ist äquivalent zur Konjunktion (Albahari 2022, S. 223)

```
obj is int  &&  (int)obj == 13
```

mit einem Typmuster im **is**-Ausdruck (siehe Abschnitt 15.2.3).

Seit C# 11 ist bei den BCL-Typen **Span<char>** und **ReadOnlySpan<char>** (siehe Abschnitt 15.3.2) das Konstantenmuster mit einem **String**-Literal erlaubt, z. B.:

Quellcode	Ausgabe
`static bool IsABC(ReadOnlySpan<char> s) => s is "ABC";` `Console.WriteLine(IsABC("abc".AsSpan()));`	`False`

In dem von Microsoft übernommenen Beispiel erlaubt der **is**-Operator eine besonders kompakte Schreibweise.[1]

Die für performantes Programmieren konzipierten Strukturen **Span<T>** und **ReadOnlySpan<T>** erfahren neuerdings in C# bzw. in der BCL eine große Unterstützung. Ihre konkretisierten Varianten **Span<char>** und **ReadOnlySpan<char>** sind besonders oft im Einsatz.

15.2.2 Residuale Muster

Das durch einen Unterstrich ausgedrückte *Ausschussmuster* (in der englischen Literatur als *discard pattern* bezeichnet) spricht in einem **switch**-Ausdruck alle bisher unbehandelten Fälle des Argumentausdrucks an. Er übernimmt die Rolle des **default**-Falls aus der **switch**-Anweisung. Im folgenden Beispiel liefert ein **switch**-Ausdruck zum Herkunftsland eines Kunden den Mehrwertsteuersatz:

Quellcode	Ausgabe

`double? res = Vat("Lichtenstein");`
`if (res != null)`
`    Console.WriteLine("Value added tax: " + res);`
`else`
`    Console.WriteLine("Value added tax: Unknown");`
 	
`static double? Vat(string country) => country switch {`
`    "Deutschland" => 19.0,`
`    "Frankreich"  => 20.0,`
`    "Luxemburg"   => 17.0,`
`    _             => null`
`};`
	`Value added tax: Unknown`

Zum Ausschussmuster passt auch der Wert **null**, der bei Bedarf durch einen separaten Verzweigungsarm mit dem **null**-Muster abfangen werden kann, z. B.:

```
null => throw new ArgumentException("Missing country"),
```

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-11.0/pattern-match-span-of-char-on-string

Durch Anwendung des **not**-Kombinators (siehe Abschnitt 15.2.8) auf das **null**-Muster lässt sich der Fall ansprechen, dass der Argumentausdruck auf ein existentes Objekt zeigt, das aber von keinem vorherigen Verzweigungsarm behandelt wurde, z. B.:

```
not null => throw new ArgumentException("Unknown country: " + country)
```

Denselben Fall kann man auch mit dem **{}** – Muster ansprechen, das ein spezielles Merkmalsmuster ist (siehe Abschnitt 15.2.4), z. B.:

```
{} => throw new ArgumentException("Unknown country: " + country)
```

Allerdings ist diese Formulierung weniger klar.

Wenn ein residuales Muster fehlt, und der Argumentausdruck einen unbehandelten Wert annimmt, dann wirft die CLR eine Ausnahme vom Typ **SwitchExpressionException** (Namensraum **System.Runtime.CompilerServices**), z. B. mit der **Message**-Eigenschaft:

```
Non-exhaustive switch expression failed to match its input.
Unmatched value was Lichtenstein.
```

15.2.3 Typmuster und Deklarationsmuster

Durch das mit C# 7.0 eingeführte *Typmuster* (engl.: *type pattern*) lässt sich für eine Variable prüfen, ob sie einen vorgeschriebenen Laufzeittyp besitzt. Das *Deklarationsmuster* (engl.: *declaration pattern*) enthält zusätzlich eine lokale Variable, die nach einer erfolgreichen Typprüfung den Instanzzugriff erlaubt. Im folgenden Beispiel wird für Instanzen zum Modellieren von Figuren die Fläche typabhängig bestimmt:[1]

```
static double Flaeche(object figur) {
    return figur switch {
        Rechteck r => r.Breite * r.Hoehe,
        Kreis k => Math.PI * k.Radius * k.Radius,
        _ => 0.0
    };
}
```

Bei Strukturinstanzen ist keine Flächenberechnung über die polymorphe Nutzung einer in der gemeinsamen Basisklasse (abstrakt) definierte Methode oder Eigenschaft möglich. Eine analoge Lage besteht für gemeinsam zu verarbeitende Klassen, wenn (außer **Object**) keine gemeinsame Basisklasse existiert, und kein Zugriff auf den Quellcode besteht. In solchen Fällen lässt sich per Typmuster unter Verwendung des Argumenttyps **Object** auf einfache Weise eine Fallunterscheidung und eine typspezifischen Berechnung realisieren.

Trotz der leistungsfähigen Musterabgleich-Optionen von C# sollte bei der Entwicklung einer Lösung, die mit unterschiedlichen Typen kooperieren kann, zunächst eine Klassenhierarchie mit Polymorphie erwogen werden. Berechnet z. B. eine Software für das Malerhandwerk zu einer Liste mit unterschiedlichen geometrischen Figuren die Gesamtfläche per Polymorphie, dann kann die Maler-Software ohne erneute Übersetzung mit später entwickelten Klassen zur Modellierung geometrischer Figuren kooperieren. Bei einer Lösung per Typmuster muss die Maler-Software hingegen zur Unterstützung von neuentwickelten Figurenklassen jeweils angepasst werden. Für eine Lösung per Polymorphie sprechen die folgenden Merkmale der Aufgabenstellung:

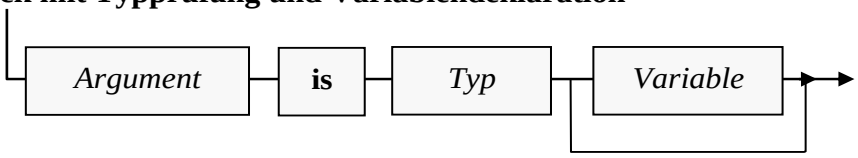
- Man hat den Quellcode der Klassen unter Kontrolle.
- Die benötigte Handlungskompetenz fällt in den primären Aufgabenbereich der Klassen (z. B. Flächenberechnung bei geometrischen Figuren).

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\C# für Fortgeschrittene\Musterabgleich\Typmuster

Den **is**-Ausdruck mit Typ- bzw. Deklarationsmuster haben wir schon im Abschnitt 7.8 kennengelernt:

is-Ausdruck mit Typprüfung und Variablendeklaration



Im Beispiel könnte die iterative Flächenberechnung um eine Infoausgabe für Kreise erweitert werden:

```
static void Main() {
    var r = new Rechteck() { Breite = 5, Hoehe = 4 };
    var k = new Kreis() { Radius = 5 };
    var lob = new List<object> { r, k };
    double gesamt = 0.0;
    foreach (var e in lob) {
        gesamt += Flaeche(e);
        if (e is Kreis kr)
            Console.WriteLine($"Kreis mit Radius: {kr.Radius}");
    }
    Console.WriteLine($"Gesamtfläche: {gesamt:f2}");
}
```

Weil im **is**-Ausdruck beliebige Typen erlaubt sind, lässt sich auch das Implementieren einer Schnittstelle prüfen. Im folgenden Beispiel aus dem Abschnitt 9.2.1 wird keine lokale Variable benötigt:

```
if (lf is IComparable<Figur>)
    lf.Sort();
```

15.2.4 Merkmalsmuster

Das mit C# 8 eingeführte *Merkmalsmuster* (engl.: *property pattern*) macht es möglich, eine Fallzugehörigkeit über eine Merkmalsausprägung oder über mehrere, durch Komma getrennte Merkmalsausprägungen zu definieren. Die Liste der geforderten Merkmalsausprägungen, die aus Feldern oder Eigenschaften stammen können, wird zwischen geschweiften Klammern formuliert, z. B.:

```
{ Passagiere: 1, Elektro: false }
```

Im folgenden Beispiel regelt eine Stadt die Fahrberechtigung für PKWs und LKWs:

- PKWs mit Elektromotor dürfen fahren.
- PKWs mit Verbrennungsmotor dürfen genau dann fahren, wenn …
  - sich mindestens 2 Personen an Bord befinden,
  - und das Kennzeichen eine gerade Nummer hat.
- LKWs dürfen genau dann fahren, wenn das Kennzeichen eine gerade Nummer hat, und das zulässige Gesamtgewicht 7,5 t nicht überschreitet.

Im Beispiel sind die beiden Typen `PKW` und `LKW` beteiligt (zu Record-Strukturen siehe Abschnitt 6.7.5):[1]

```
public record struct PKW(int Passagiere, bool Elektro, bool GeradeNummer);
public record struct LKW(double Gewicht, bool GeradeNummer);
```

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\C# für Fortgeschrittene\Musterabgleich\Merkmalsmuster

Zunächst beschränken wir uns auf die PKWs und entscheiden über die Fahrberechtigung in der folgenden Methode mit Hilfe eines **switch**-Ausdrucks:

```
static bool PkwZugelassen(PKW pkw) =>
    pkw switch {
        { Passagiere: 1, Elektro: false } => false,
        { Elektro: true } => true,
        { GeradeNummer: true } => true,
        _ => false
    };
```

Es werden drei Verzweigungsarme mit Merkmalsmuster und ein Verzweigungsarm mit Ausschussmuster verwendet.

Der folgende **switch**-Ausdruck entscheidet über *alle* Fahrzeuge und nutzt dabei die Möglichkeit, das Typ- und das Merkmalsmuster zu kombinieren:

```
static bool Zugelassen(object obj) =>
    obj switch {
        PKW { Passagiere: 1, Elektro: false } => false,
        PKW { Elektro: true } => true,
        PKW { GeradeNummer: true } => true,
        LKW { GeradeNummer: true } lkw => lkw.Gewicht <= 7.5,
        _ => false
    };
```

Wie zu einem Typmuster kann auch zu einem Merkmalsmuster eine lokale Variable definiert werden, die anschließend einen Instanzzugriff erlaubt, z. B.:

```
LKW { GeradeNummer: true } lkw => lkw.Gewicht <= 7.5,
```

Im Merkmalsmuster darf man nach dem Doppelpunkt nicht nur einen konstanten Wert angeben (das Konstantenmuster nutzen), sondern es sind auch andere Muster erlaubt, z. B. das Relationsmuster (siehe Abschnitt 15.2.7).[1] Damit lässt sich im Beispiel der `LKW`-Fall einfacher formulieren:

```
LKW { Gewicht: <= 7.5, GeradeNummer: true } => true,
```

Besitzt eine Instanz eine Mitgliedsinstanz, dann können Merkmalsmuster geschachtelt werden. Im folgenden Beispiel besitzt die Record-Struktur `Kreis` das Feld `Radius` vom elementaren Typ **double** und das Feld `Zentrum` vom Record-Strukturtyp `Punkt`:

```
public record struct Punkt(double X, double Y);
public record struct Kreis(Punkt Zentrum, double Radius);
```

Per **is**-Ausdruck wird für eine `Kreis`-Instanz getestet, ob sie den `Radius` 5 und ein Zentrum mit der X-Koordinate 7 besitzt:

```
bool erg = k is { Radius: 5, Zentrum: { X: 7 } };
```

Seit C# 10 ist für geschachtelte Merkmalsmuster eine vereinfachte Schreibweise erlaubt mit einem durch Punkte getrennten Pfad zu einem Mitgliedsmerkmal, z. B.

```
bool erg = k is { Radius: 5, Zentrum.X: 7 };
```

Das Speichern in einer lokalen Variablen gelingt nicht nur für eine komplette Instanz, sondern auch für einzelne Merkmale. Auf diese Weise können Beziehungen zwischen *verschiedenen* Merkmalen zu einem Fallkriterium werden. Im folgenden Beispiel wird für eine `Kreis`-Instanz getestet, ob der `Radius` gleich 5 ist, und das `Zentrum` auf der Hauptdiagonalen liegt, also identische X- und Y-Koordinaten hat:

---

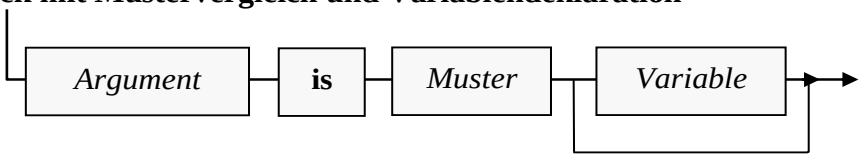
[1] Wegen der Möglichkeit zur Verwendung eingeschachtelter Muster wird das Merkmalsmuster als *rekursiv* bezeichnet, siehe z. B.:
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns

```
bool erg = k is { Radius: 5, Zentrum: { X: var x, Y: var y } } && x == y;
```

Zum Extrahieren einer Merkmalsausprägung fügt man nach dem Doppelpunkt ein **var**-Muster ein und nutzt somit die oben erwähnte Option zur Verwendung eingeschachtelter Muster.

Wie die Kreisbeispiele gezeigt haben, ist der **is**-Ausdruck keinesfalls auf das Typmuster beschränkt, sodass die im Abschnitt 7.8 angekündigte Verallgemeinerung des **is**-Ausdrucks demonstriert ist:

is-Ausdruck mit Mustervergleich und Variablendeklaration



Ein Merkmalsmuster kann nur zutreffen, wenn der Argumentausdruck einen von **null** verschiedenen Wert besitzt. Folglich wirkt eine leere Merkmalsliste **{}** wie ein **null**-Test, z. B.:

```
{ } => false,
```

Allerdings führt die Anwendung des **not**-Kombinators (siehe Abschnitt 15.2.8) auf das **null**-Muster zu einem besser lesbaren Quellcode, z. B.:

```
not null => false,
```

15.2.5 Tupelmuster

Das mit C# 8 eingeführte *Tupelmuster* (engl.: *tuple pattern*) kann als mehrdimensionale Variante des Konstantenmusters aufgefasst werden. Eine Abbildung vom Kreuzprodukt mehrerer Mengen ($A_1 \times A_2 \times ... \times A_k$) in eine Zielmenge B wird durch eine Tabelle beschrieben. In einem künstlichen Beispiel betrachten wir 3 dichotome Variablen A_1, A_2 und A_3 (mit den Werten 0 und 1) und eine Abbildung in die Menge der booleschen Werte {**true**, **false**}, die ...

- den Wert **true** liefert, wenn zwei benachbarte Variablen denselben Wert (0 oder 1) haben, die dritte Variable jedoch einen abweichenden Wert besitzt,
- in allen anderen Fällen den Wert **false** liefert.

Die Beschreibung der Abbildung durch eine Tabelle erfordert Fleißarbeit, führt aber zu einem übersichtlichen Ergebnis:[1]

```
static bool GenauZweiNachbarnGleich(int a1, int a2, int a3) =>
    (a1, a2, a3) switch {
        (0, _, 0) => false,
        (0, 0, 1) => true,
        (0, 1, 1) => true,
        (1, 0, 0) => true,
        (1, _, 1) => false,
        (1, 1, 0) => true
    };
```

Ist in einem Verzweigungsarm ein Tupelelement irrelevant, dann ersetzt man es durch einen Unterstrich. Weitere Optionen zur Definition eines Verzweigungsarms werden im folgenden Abschnitt über das *Positions-* bzw. *Dekonstruktionsmuster* beschrieben, das die Anwendung des Tupelmusters auf Typen mit einem Dekonstruktor erlaubt.

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
…\BspUeb\C# für Fortgeschrittene\Musterabgleich\Tupelmuster

Funktional äquivalent zum Beispiel, weniger Schreibaufwand verursachend, aber vielleicht auch weniger übersichtlich ist die folgende Lösung mit einem logischen Ausdruck:

```
public static bool GenauZweiNachbarnGleich(int a1, int a2, int a3) =>
    a2 == a1 != (a2 == a3);
```

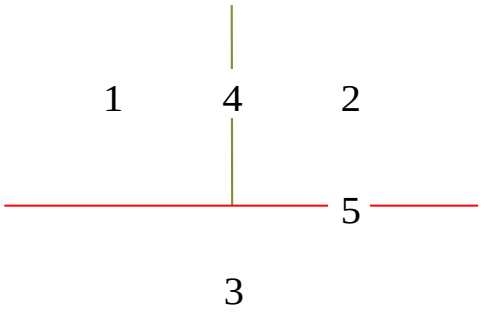
15.2.6 Positions- bzw. Dekonstruktionsmuster

Besitzt ein Typ eine **Deconstruct()** -Methode (vgl. Abschnitt 6.6.5), dann lässt sich deren Tupel-Produktion als Eingabe für einen **switch**-Ausdruck verwenden. Das mit C# 8 eingeführte *Positions-* bzw. *Dekonstruktionsmuster* (engl.: *positional pattern*) verallgemeinert das im letzten Abschnitt beschriebene Tupelmuster auf Typen mit einem Dekonstruktor.

Zur Definition eines **switch**-Verzweigungsarms kann man (beim Tupel- und beim Dekonstruktionsmuster) an den einzelnen Positionen des Tupels ...

- bestimmte Werte vorschreiben,
- per **var**-Muster lokale Variablen zur näheren Beurteilung in einer **when**-Klausel ablesen,
- oder durch die Verwendung des Unterstrichs beliebige Werte zulassen.

Als Beispiel betrachten wir eine Klassifikation von Punkten auf der Zahlenebene:



Instanzen der folgenden Record-Struktur `Punkt` sollen einer von 5 Kategorien zugeordnet werden:

```
public record struct Punkt(double X, double Y);
```

Der Compiler liefert für einen Record-Typ …

- den primären Konstruktor (mit einem Parameter für jede Eigenschaft)
- und eine **Deconstruct()** – Methode.

Im folgenden Programm[1]

```
var p1 = new Punkt(-2, 2);
var p2 = new Punkt(2, 2);
var p3 = new Punkt(-2, -2);
var p4 = new Punkt(0, 2);
var p5 = new Punkt(-2, 0);
var pl = new List<Punkt> { p1, p2, p3, p4, p5 };
for (int i = 0; i < pl.Count; i++)
    Console.WriteLine($"({pl[i].X,2},{pl[i].Y,2}): Zone {Zone(pl[i])}");
```

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
…\BspUeb\C# für Fortgeschrittene\Musterabgleich\Positionsmuster

```
static int Zone(Punkt punkt) =>
     punkt switch {
         var (x, y) when x < 0 && y > 0 => 1,
         var (x, y) when x > 0 && y > 0 => 2,
         (_, var y) when y < 0 => 3,
         (0, var y) when y > 0 => 4,
         (_, 0) => 5
     };
```

werden einige Punkte klassifiziert:

```
(-2, 2): Zone 1
( 2, 2): Zone 2
(-2,-2): Zone 3
( 0, 2): Zone 4
(-2, 0): Zone 5
```

15.2.7 Relationsmuster

Das mit C# 9 eingeführte *Relationsmuster* (engl.: *relational pattern*) erlaubt die Definition von Verzweigungsarmen durch …

- einen relationalen Operator (**<**, **<=**, **>**, **>=**)
- und einen konstanten Ausdruck (z. B. ein Literal) mit einem numerischen Typ (Gleitkomma oder Ganzzahl inklusive **char**) oder einem Enumerationstyp.

Der Argumentausdruck muss entweder direkt oder nach einer impliziten Konversion mit den Konstanten vergleichbar sein.[1]

Per Relationsmuster lässt sich in **switch**-Ausdrücken oft die seit C# 7 verfügbare **when**-Klausel ersetzen. Das folgende Programm berechnet für LKWs eine gewichtsabhängige Maut

```
Console.WriteLine("Maut: " + Maut(new LKW(1500)));
Console.WriteLine("Maut: " + Maut(new LKW(8000)));

static decimal Maut(LKW lkw) =>
        lkw.Gewicht switch {
            <= 1000 => 10.0m,
            <= 2000 => 18.5m,
            <= 7500 => 25.2m,
            _       => 30.8m
        };

public record struct LKW(double Gewicht);
```

mit dem Ergebnis:

```
Maut: 18,5
Maut: 30,8
```

15.2.8 Kombinatoren und logische Muster

Seit C# 9 ist es möglich, die Prüfergebnisse mehrerer Muster durch sogenannte *Kombinatoren* zu einem *logischen Muster* zusammenzuführen. Für die Negation, Konjunktion und Disjunktion sind Schlüsselwörter mit den folgenden, absteigend geordneten Prioritäten zu verwenden:

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/patterns3#relational-patterns

- **not**
  Das Ergebnis des Musterabgleichs wird invertiert.
- **and**
  Der Argumentausdruck erfüllt das logische Muster, wenn er die *beiden* zusammengeführten Muster erfüllt.
- **or**
  Der Argumentausdruck erfüllt das logische Muster, wenn er *mindestens eines* von den beiden zusammengeführten Mustern erfüllt.

Wie bei den logischen Operatoren lässt sich die Auswertungsreihenfolge durch runde Klammern steuern.[1]

Durch Konjunktionen von Relationsmustern lässt sich eine Intervalleinteilung realisieren, z. B.:

Quellcode	Ausgabe

`int i = 17;`
`int k = i switch {`
`  < 10  => 1,`
`  >= 10 and < 15 => 2,`
`  >= 15 and < 20 => 3,`
`  >= 20  => 4`
`};`
`Console.WriteLine(k);`
	`3`

Im **is**-Ausdruck des folgenden Beispiels werden mehrere Relationsmuster und ein Konstantenmuster kombiniert:

Quellcode	Ausgabe

`char c = 'ä';`
`Console.WriteLine(c is (>= 'a' and <= 'z' or 'ü' or 'ö' or 'ä')`
`  and not 'x');`
	`True`

Es wird getestet, ob die lokale **char**-Variable `c` einen von ‚x' verschiedenen Kleinbuchstaben enthält.

Bei den binären Musterkombinatoren (**and**, **or**) fällt im Vergleich zu den binären logischen Operatoren (z. B. **&&**, ||) auf, dass Variablennamen nicht wiederholt werden müssen. In der folgenden Variante des letzten Beispiels tritt der logische Operator **&&** zusammen mit Musterkombinatoren auf:

Quellcode	Ausgabe

`char c = 'ä';`
`Console.WriteLine(c is >= 'a' and <= 'z' or 'ü' or 'ö' or 'ä'`
`  && c is not 'x');`
	`True`

In diesem Fall haben die Musterkombinatoren eine höhere Priorität (Bindungskraft).

Beim **null**-Test (z. B. für das **String**-Objekt `s`) besteht zur Formulierung

```
if (s != null) Console.WriteLine(s.Length);
```

dank **null**-Muster und **not**-Kombinator die Alternative:

```
if (s is not null) Console.WriteLine(s.Length);
```

---

[1] Manchmal wird vom *Klammermuster* gesprochen (engl.: *parenthesized pattern*) wenn in einem logischen Muster runde Klammern zum Einsatz kommen, z. B.:
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns#parenthesized-pattern

15.2.9 Listenmuster

Das mit C# 11 eingeführte *Listenmuster* (engl.: *list pattern*) macht es möglich, einen Array oder eine Liste mit einer Sequenz von Mustern zu vergleichen.[1] Ein Array bzw. eine Liste (ab jetzt kurz: *eine Kollektion*) erfüllt ein Listenmuster (bzw. eine Mustersequenz), wenn jedes Element der Kollektion das korrespondierende (positionsgleiche) Element der Mustersequenz erfüllt.

Wir betrachten zunächst einfache, aus Konstanten bestehende Mustersequenzen:[2]

```
int[] zahlen = { 1, 4, 7, 8 };

Console.WriteLine(zahlen is [1, 4, 7, 8]);          // Ausgabe: True
Console.WriteLine(zahlen is [1, 7, 4, 8]);          // Ausgabe: False
Console.WriteLine(zahlen is [1, 4, 7, 8, 9]);       // Ausgabe: False
```

Für ein positives Prüfergebnis müssen die Anzahl und die Abfolge der Elemente übereinstimmen.

Mit dem Listenmuster **[]** wird eine *leere* Kollektion angesprochen, z. B.:

```
int[] zahlen0 = { };
int[] zahlen1 = { 1 };
Console.WriteLine(zahlen0 is not []);               // Ausgabe: False
Console.WriteLine(zahlen1 is not []);               // Ausgabe: True
```

Die Flexibilität der Listenmuster steigt erheblich durch die Möglichkeit, für jedes Element ein separates Muster zu wählen, z. B.:

```
Console.WriteLine(zahlen is [1, 4 or 5, <= 7, int]);  // Ausgabe: True
Console.WriteLine(zahlen is [1, > 4, 7, int]);        // Ausgabe: False
```

Als Elemente eines Listenmusters (bzw. einer Mustersequenz) sind erlaubt:

- Konstantenmuster
- Typmuster
- Merkmalsmuster
- Relationsmuster
- Logische Muster
- Ausschussmuster und Bereichsmuster (siehe unten)
- Listenmuster
  Diese Option wird für eine Kollektion von Kollektionen benötigt.

15.2.9.1 Ausschuss- und Bereichsmuster

Um das Element an einer bestimmten Position der Kandidatenkollektion von der Prüfung auszunehmen, verwendet man (bei bedarf beliebig oft) das durch einen Unterstrich ausgedrückte Ausschussmuster, z. B.:

```
Console.WriteLine(zahlen is [1, 4, _, 8]);          // True
Console.WriteLine(zahlen is [_, 4, _, 8]);          // True
if (zahlen is [1, 4, 7, var ende])
    Console.WriteLine($"Kandidatensequenz passt, letztes Element: {ende}");
```

Über das im letzten Beispiel verwendete **var**-Muster wird das ungeprüfte Element zwecks anschließender Analyse in eine lokale Variable befördert.

---

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns#list-patterns

[2] Ein Visual Studio - Projekt mit den im aktuellen Abschnitt vorgestellten Beispielen ist hier zu finden:
…\BspUeb\C# für Fortgeschrittene\Musterabgleich\Listenmuster

Während per Ausschussmuster genau *ein* Element der Kandidatenkollektion von der Prüfung ausgenommen wird, erlaubt das durch zwei aufeinanderfolgende Punkte ausgedrückte *Bereichsmuster* (engl.: *slice pattern*) die analoge Behandlung für eine Sequenz von 0 oder mehr Elementen, z. B.:

```
Console.WriteLine(zahlen is [.., 7, 8]);             // True
Console.WriteLine(zahlen is [1, 4, ..]);             // True
Console.WriteLine(zahlen is [1, .., 8]);             // True
Console.WriteLine(zahlen is [.. { Length: 3 }, 8]);  // True
```

Damit wird die Restriktion überwunden, dass erfüllenden Kollektionen eine bestimmte Länge haben müssen. In einem Listenmuster darf nur *ein* Bereichsmuster auftreten, wobei eine beliebige Position erlaubt ist.

Im letzten Beispiel wird das Bereichsmuster durch ein Merkmalsmuster erweitert, sodass sich für die ungeprüfte Teilkollektion eine Ausprägung der Eigenschaft **Length** vorscheiben lässt.

Um die ungeprüfte Teilkollektion analysieren zu können, erweitert man das Bereichsmuster durch ein **var**-Muster und deklariert eine lokale Variable mit einer Referenz auf die ungeprüfte Teilkollektion, z. B.:

```
if (zahlen is [1, .. var mitte, 8]) {
    Console.Write("Kandidatensequenz passt, Mittelteil:");
    foreach (var e in mitte)
        Console.Write(" " + e);
}
```

Durch eine explizite Typangabe für die deklarierte lokale Variable lässt sich die Lesbarkeit des Quellcodes verbessern:

```
if (zahlen is [1, .. int[] mitte, 8]) {
    . . .
}
```

15.2.9.2 Typanforderungen für Listenmuster-Kandidaten

Bislang wurden Arrays und Listen als zulässige Listenmuster-Kandidaten genannt. Außerdem kann u. a. für Objekte der Klasse **String** untersucht werden, ob eine Sequenz von **char**-basierten Mustern vorliegt. Die folgende Methode `ExtractPermEndings` liefert von Zeichenfolgen, die mit einem Buchstaben und einer Ziffer starten, den Rest, sofern er mindestens ein und maximal fünf Zeichen enthält:

```
Console.WriteLine(ExtractPermEndings("K7usw")); // Ausgabe: usw
string? ExtractPermEndings(string s) {
    if (s is [>= 'a' and <= 'z' or >= 'A' and <= 'Z',
              >= '0' and <= '9',
              .. { Length: >= 1 and <= 5 } rest])
        return rest;
    return null;
}
```

Einfache Zeichenfolgeauswertungen lassen sich per Listenmuster erledigen, sodass man auf die leistungsfähigere, aber auch anspruchsvollere Technik der regulären Ausdrücke verzichten kann, die in C# durch die Klasse **Regex** im Namensraum **System.Text.RegularExpressions** realisiert wird (siehe Abschnitt 6.8.6).

Damit eine Kollektion als Kandidat für ein Listenmuster in Frage kommt, muss ihr *deklarierter* Typ die folgenden Voraussetzungen erfüllen:[1]

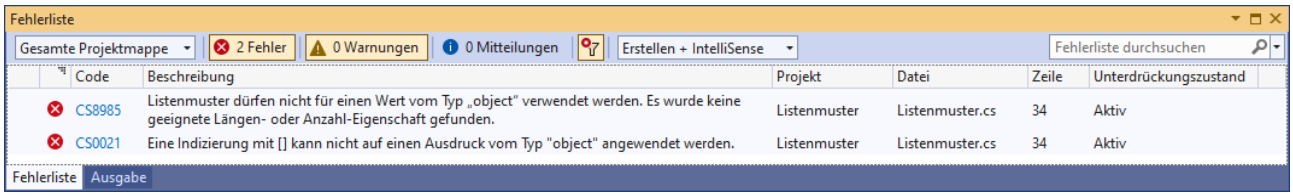
[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-11.0/list-patterns

- Zur Feststellung der Länge ist eine **int**-Eigenschaft namens **Length** oder **Count** verfügbar. Sind beide vorhanden, dann bevorzugt der Compiler die Eigenschaft **Length**:
```
public int Length { get; }
public int Count { get; }
```
- Es ist ein Indexer mit **System.Index** – oder **int** – Parameter verfügbar. Sind beide vorhanden, dann bevorzugt der Compiler den Indexer mit **System.Index** – Parameter:
```
public object this[System.Index index] => throw null;
public object this[int index] => throw null;
```

Im folgenden Beispiel

```
string text = "1478";
object obj = text;
Console.WriteLine(obj is ['1', '4', '7', '8']);
```

sind die Voraussetzungen für den Laufzeittyp erfüllt, aber nicht für den deklarierten Typ, sodass die Übersetzung scheitert:



Griffiths (2023) weist darauf hin, dass bei anderen Musterabgleichen die Typanforderungen und die Instanzdetails zur Laufzeit geprüft werden, z. B.:

```
var tuple = (1, 2);
object obj = tuple;
Console.WriteLine(obj is (1 or 2, >= 2)); // Ausgabe: True
```

Wie im Abschnitt 15.2.9.1 zu sehen war, kann ein Bereichsmuster (**..**) optional durch ein Muster für den ungeprüften Bereich ergänzt werden, z. B. …

- durch ein Merkmalsmuster, das eine Länge der Teilkollektion vorschreibt
- oder durch ein **var**-Muster, das eine Variable für den Zugriff auf die Teilkollektion deklariert.

Damit eine Kollektion als Kandidat für ein *erweitertes* Listenmuster in Frage kommt, muss ihr deklarierter Typ über die oben genannten Voraussetzungen hinaus …

- entweder einen Indexer mit **Range**-Parameter
- oder eine **Slice()** - Methode

besitzen, um eine Bereichsauswahl zu ermöglichen (Griffiths 2023). Sind beide Optionen vorhanden, dann bevorzugt der Compiler den Indexer mit **System.Range** – Parameter:[1]

```
public object this[System.Range index]
public object Slice(int start, int length)
```

[1] Die Details zur Realisation der Typanforderungen stammen von:
https://www.meziantou.net/csharp-11-list-patterns-create-compatible-types.htm

15.3 Schlüsselwort ref

Das Schlüsselwort **ref** ist uns bisher als Modifikator für Methodenparameter begegnet (siehe Abschnitt 5.3.1.3.2.1). Jüngere C# - Versionen erlauben das Schlüsselwort auch in anderen Kontexten, wobei es meist um die Leistungsoptimierung geht.[1] Für Einsteiger sind diese C# - Spracherweiterungen weniger zu empfehlen.

15.3.1 Methoden mit ref-Variablen und ref-Rückgabewerten

Seit C# 7.0 kann man in einer Methodendefinition durch die **ref**-Deklaration von lokalen Variablen und Rückgabewerten für mehr Sicherheit und Leistung sorgen.

15.3.1.1 ref-Variablen in Methoden

Im Abschnitt 15.3.1 geht es primär um Methoden mit einem Verweis auf eine Strukturinstanz als Rückgabe. Weil eine solche Rückgabe oft in einer lokalen Variablen des Aufrufers landet, beschäftigen wir uns zunächst mit **ref**-Variablen in Methoden. Das sind ...

- lokale Variablen,
- die eine Referenz auf eine im aktuellen Kontext sichtbare Variable aufnehmen können.

Im Unterschied zu den Referenzvariablen im Sinn von Abschnitt 4.3.2.2 muss es sich beim Referenzziel nicht um ein komplettes Objekt auf dem Heap handeln. Alternative Referenzziele sind z. B.

- ein einzelne Instanzvariable
- ein Array-Element
- eine Strukturinstanz auf dem Stack

Durch eine **ref**-Variable sind u. a. die folgenden Vorteile zu realisieren:

- Durch das Referenzieren von Strukturinstanzen kann man Kopiervorgänge vermeiden und somit CPU-Zeit sowie Speicher sparen (wie bei Methodenparametern mit **ref**-Modifikator, siehe Abschnitt 5.3.1.3.2.1).
- Wenn nur eine einzelne Instanzvariable eines Objekts oder ein einzelnes Array-Element angesprochen werden kann, ist das Fehlerrisiko reduziert.

In einer Trockenübung wird zunächst demonstriert, dass mit dem Schlüsselwort **ref** eine lokale Variable deklariert werden kann, die auf ein Array-Element im selben Gültigkeitsbereich zeigt:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int[] iar = { 1, 2, 3 };`
`  ref int loref = ref iar[2];`
`  loref = 4;`
`  Console.WriteLine(iar[2]);`
`  }`
`}`
	`4`

Eine **ref**-Variable muss bei der Deklaration initialisiert werden, wobei das Schlüsselwort **ref** dem Datentyp *und* dem Referenzziel vorangestellt werden muss.

Seit C# 7.3 ist es erlaubt, das Verweisziel einer **ref**-Variablen zu ändern, z. B.:

```
loref = ref iar[0];
```

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref

Seit C# 7.2 kann man eine lokale **ref**-Variable als **readonly** deklarieren, sodass nur lesende Zugriffe auf das Referenzziel möglich sind, z. B.:

```
ref readonly int loref = ref iar[2];
```

15.3.1.2 Methoden mit ref-Rückgabewert

Meist wird eine lokale **ref**-Variable dazu verwendet, die von einer Methode als **ref**-Rückgabewert gelieferte Referenz aufzubewahren. Im folgenden Beispiel werden Instanzen der Record-Struktur `Strecke`

```
public record struct Strecke(int Li, int Re);
```

durch eine Klasse namens `Streckensammlung` verwaltet. In einem statischen Array befinden sich 5 Strecken mit einem zufällig bestimmten Startpunkt aus dem Intervall von 0 bis 95 und der Länge 10:

```
public class Streckensammlung {
    const int anz = 5, max = 100, laenge = 10;
    static readonly Strecke[] strecken;
    static Strecke nf = new(-1, -1);

    static Streckensammlung() {
        strecken = new Strecke[anz];
        var zzg = new Random();
        for (int i = 0; i < anz; i++) {
            int start = zzg.Next(max - 4);
            strecken[i] = new Strecke(start, start + laenge);
        }
    }

    public static ref Strecke FindeStrecke(int x) {
        for (int i = 0; i < anz; i++)
            if (x >= strecken[i].Li && x <= strecken[i].Re)
                return ref strecken[i];
        return ref nf;
    }

    public static void ProtStrecken() {
        for(int i = 0; i < anz; i++)
            Console.WriteLine($"Strecke {i}: ({strecken[i].Li}, {strecken[i].Re})");
    }
}
```

Die Methode `FindeStrecke()` sucht für den Aktualparameter die erste enthaltende Strecke und liefert diese als **ref**-Rückgabewert. Dazu muss das Schlüsselwort **ref** ...

- im Definitionskopf vor den Rückgabetyp
- und in jeder **return**-Anweisung vor den Rückgabewert

gesetzt werden.

Weitere Regeln:

- Die Gültigkeit der **ref**-Rückgabe muss die Ausführung der Methode überdauern. Das ist z. B. der Fall …
  - bei einer Instanzvariablen eines Objekts,
  - bei einem Array-Element,
  - bei einem statischen Feld einer Klasse,
  - bei einem **ref**-Aktualparameter, den die Methode erhalten hat.
- Bei einer asynchronen Methode ist keine **ref**-Rückgabe möglich (zu asynchronen Methoden siehe Kapitel 17 von Baltes-Götz (2021)).

In der Startmethode der folgenden Klasse wird einer **ref**-Variablen die **ref**-Rückgabe der Methode `FindeStrecke()` zugewiesen, wobei das Schlüsselwort **ref** in der Variablendeklaration dem Datentyp *und* dem Referenzziel vorangestellt werden muss:

```
class Prog {
    static void Main() {
        Streckensammlung.ProtStrecken();
        ref Strecke strecke = ref Streckensammlung.FindeStrecke(12);
        if (strecke.Li != -1) {
            Console.WriteLine("Treffer in (" + strecke.Li + ", " + strecke.Re + ")");
            strecke.Li = 0; strecke.Re = 0;
        } else
            Console.WriteLine("Kein Treffer");
        Console.WriteLine();
        Streckensammlung.ProtStrecken();
    }
}
```

Der Zweck einer **ref**-Rückgabe besteht darin, dem Aufrufer den lesenden und schreibenden Zugriff auf das Original zu ermöglichen. Wie das Beispiel zeigt, klappt das auch bei einem privaten Feld einer anderen Klasse:

```
Strecke 0: (63, 73)
Strecke 1: (95, 105)
Strecke 2: (2, 12)
Strecke 3: (8, 18)
Strecke 4: (6, 16)
Treffer in (2, 12)

Strecke 0: (63, 73)
Strecke 1: (95, 105)
Strecke 2: (0, 0)
Strecke 3: (8, 18)
Strecke 4: (6, 16)
```

Ein Ordner mit dem Visual Studio – Projekt zum Beispiel ist hier zu finden:

…\BspUeb\C# für Fortgeschrittene\ref\ref-Rückgabe

Wenn der Aufrufer als Rückgabe keine *Referenz* auf das Original, sondern eine *Kopie* erhalten soll, dann lässt man in der initialisierenden Variablendeklaration die beiden **ref**-Schlüsselwörter weg, z. B.:

```
Strecke strecke = strKoll.FindeStrecke(12);
```

Seit C# 7.2 kann man eine **ref**-Rückgabe als **readonly** deklarieren, sodass nur lesende Zugriffe auf das Referenzziel möglich sind, z. B.:

```
public ref readonly Strecke FindeStrecke(int x) {
    . . .
}
```

Damit einer lokalen **ref**-Variablen die **ref readonly** - Rückgabe zugewiesen werden kann, muss diese ebenfalls als **ref readonly** deklariert werden, z. B.:

```
ref readonly Strecke strecke = ref strKoll.FindeStrecke(12);
```

15.3.1.3 ref-Rückgabe durch den Konditionaloperator

Seit C# 7.2 kann der Konditionaloperator eine **ref**-Rückgabe liefern. Im folgenden Beispiel wird in Abhängigkeit von einer Bedingung eine Referenz auf das erste oder zweite Element eines Arrays geliefert:

```
ref var r = ref (ar[1] % 2 == 0 ? ref ar[0] : ref ar[1]);
```

15.3.2 ref-Strukturen

Seit C# 7.2 kann bei der Definition einer Struktur durch den Modifikator **ref** verhindert werden, dass Instanzen der Struktur auf den Heap gelangen (z. B. per Boxing oder als Array-Elemente). Von dieser Option profitieren vor allem die generischen BCL-Strukturen **Span<T>**[1]

```
public readonly ref struct Span<T> {
    // A byref or a native ptr
    internal readonly ref T _reference;
    // The number of elements this Span contains.
    private readonly int _length;
    . . .
}
```

und **ReadOnlySpan<T>**.

Aus der Heap-Abstinenz resultieren etliche Einschränkungen.[2] Eine **ref**-Struktur …

- ist nicht als Elementtyp eines Arrays zugelassen,
- ist in einer Klasse oder in einer normalen Struktur (ohne **ref**-Modifikator) nicht als Instanzvariablentyp erlaubt,
- kann keine Schnittstellen implementieren,
- kann nicht per Boxing in ein Objekt verpackt werden,
- kann keinen Typformalparameter konkretisieren,
- kann nicht in der Definition eines Delegatenobjekts (per Lambda-Notation oder anonyme Methode) oder in der Definition einer lokalen Methode als Bestandteil der umgebenden Methode eingefangen werden,
- kann nicht in einer Methode mit **async**-Modifikator (siehe Kapitel 17 von Baltes-Götz (2021)) verwendet werden,
- kann nicht in einem Iterator verwendet werden.

Die BCL-Strukturen **Span<T>** und **ReadOnlySpan<T>** werden von ca. 5.000 BCL-Methoden verwendet und sorgen auch dann für eine bessere Leistung, wenn sie in einem Programm nicht direkt zum Einsatz kommen.[3] Sie erlauben die typsichere und speichersichere Verwendung eines zusammenhängenden Bereichs von beliebigem Arbeitsspeicher, z. B.[4]

- Array oder Segment davon
- **String**-Objekt oder Segment davon
- per **stackalloc** belegter Speicherbereich auf dem Stack

Der Leistungsgewinn resultiert vor allem daraus, dass …

- die aufwändige Kreation von Heap-Objekten nach Möglichkeit vermieden
- und damit auch der Garbage Collector entlastet wird.

---

1 Wie man den BCL-Quellcode einsehen und herunterladen kann, erläutert Abschnitt 2.6.4.

2 https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/ref-struct

3 https://blog.ndepend.com/improve-c-code-performance-with-spant/

4 https://learn.microsoft.com/de-de/dotnet/api/system.span-1

Seit C# 11 kann in einer **ref**-Struktur (und nur dort) ein **ref**-Feld deklariert werden (siehe obigen Quellcode der Struktur **Span<T>**). Es dient (wie eine lokale **ref**-Variable in einer Methode, siehe Abschnitt 15.3.1.1) dazu, Kopiervorgänge zu vermeiden. Wie ein Blick auf den **Span<T>** - Konstruktor zeigt, dient das **ref**-Feld `_reference` dazu, einen zusammenhängenden Speicherbereich ohne Kopiervorgang zu verwenden:

```
public Span(ref T reference) {
    _reference = ref reference;
    _length = 1;
}
```

Die Definition eigener **ref**-Strukturen kommt sehr viel seltener in Frage als die Nutzung der BCL-Strukturen **Span<T>** und **ReadOnlySpan<T>**. Daher sparen wir uns weitere technische Details und demonstrieren stattdessen an einem Beispiel die mit Hilfe der BCL-Klasse **ReadOnlySpan<T>** realisierbare Leistungssteigerung im Vergleich zu einer Standardlösung mit zahlreichen Objektkreationen.[1] Für 1001 positive Zahlen, die sich durch Kommata getrennt in einer Zeichenfolge befinden, soll die Summe ermittelt werden. Die Zeichenfolge stammt in der Praxis z. B. aus einer Datendatei im CSV-Format (*Comma Separated Variables*). In der zum Leistungsvergleich dienenden Klasse `ReadOnlySpanBenchmark` erzeugt der Einfachheit halber ein statischer Konstruktor die Zeichenfolge:

```
static ReadOnlySpanBenchmark() {
    StringBuilder sb = new();
    var rnd = new Random(13);
    for (int i = 0; i < 1000; i++) {
        sb.Append(rnd.Next(99999));
        sb.Append(",");
    }
    sb.Append(rnd.Next(99999));
    csvData = sb.ToString();
}
```

In der Methode `MeanWithSplit()`

```
public void MeanWithSplit() {
    string[] arrayOfString = csvData.Split(',');
    var length = arrayOfString.Length;
    int sum = 0;
    for (int i = 0; i < length; i++)
        sum += int.Parse(arrayOfString[i]);
}
```

wird die Zeichenfolge (in der Variablen `csvData`) mit Hilfe der **String**-Methode **Split()** in einen **String**-Array mit einer Zeichenfolge pro Zahl zerlegt. Die Array-Elemente werden durch die **uint**-Methode **Parse()** numerisch interpretiert und aufaddiert.

In der Methode `MeanWithSpan()` wird zunächst über die **String**-Methode **AsSpan()** aus der Zeichenfolge `csvData` die Instanz `span` vom Typ **ReadOnlySpan<char>** erstellt:

---

[1] Einige technische Details stammen von:
https://blog.ndepend.com/improve-c-code-performance-with-spant/

```
public void MeanWithSpan() {
    ReadOnlySpan<char> span = csvData.AsSpan();
    int nextCommaIndex = 0;
    bool noMoreComma = false;
    int sum = 0;
    while (!noMoreComma) {
        int indexStart = nextCommaIndex;
        nextCommaIndex = csvData.IndexOf(',', indexStart);
        noMoreComma = (nextCommaIndex <= 0);
        if (noMoreComma)
            nextCommaIndex = csvData.Length;
        ReadOnlySpan<char> slice = span.Slice(indexStart,
                                              nextCommaIndex - indexStart);
        sum += int.Parse(slice);
        nextCommaIndex++;
    }
}
```

Für jede Zahl wird anhand des Trennzeichens die zugehörige Teilzeichenfolge ermittelt, aus der die **ReadOnlySpan<char>** – Methode **Slice()** eine **ReadOnlySpan<char>** - Instanz erstellt, die von der **uint**-Methode **Parse()** als Parameter akzeptiert wird.

Um einen Leistungsvergleich mit hoher Genauigkeit zu erhalten, setzen wir das NuGet-Paket **BenchmarkDotNet** ein.[1]

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Order;
using BenchmarkDotNet.Running;
using System;
using System.Text;

[RankColumn]
[Orderer(SummaryOrderPolicy.FastestToSlowest)]
[MemoryDiagnoser]
public class ReadOnlySpanBenchmark {
    static readonly string csvData;

    static ReadOnlySpanBenchmark() {
        . . .
    }

    [Benchmark(Baseline = true)]
    public void MeanWithSplit() {
        . . .
    }

    [Benchmark]
    public void MeanWithSpan() {
        . . .
    }
}
```

Schließlich starten wir den Leistungsvergleich in der Release-Konfiguration des Projekts mit der folgenden Anweisung auf oberster Ebene, die einen Aufruf der statischen und generischen Methode **Run()** der Klasse **BenchmarkRunner** (Namensraum **BenchmarkDotNet.Running**) enthält:

```
BenchmarkRunner.Run<ReadOnlySpanBenchmark>();
```

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

Die Ergebnistabelle zeigt, dass mit Hilfe der Struktur **ReadOnlySpan<char>** fast 50% der Rechenzeit eingespart werden konnte (Messungen in Mikrosekunden):

```
|        Method |     Mean |    Error |   StdDev |    Gen0 | Allocated |
|-------------- |---------:|---------:|---------:|--------:|----------:|
|  MeanWithSpan | 35.05 us | 0.600 us | 0.667 us |       - |         - |
| MeanWithSplit | 60.43 us | 1.187 us | 2.727 us | 19.0430 |   40080 B |
```

In den beiden letzten Spalten sind die Ursachen für die Zeiteinsparung zu erkennen: Die Methode `MeanWithSpan()` legt keine Heap-Objekte an und belastet den Garbage Collector nicht mit Maßnahmen für Objekte aus der Generation 0.

Ein Ordner mit dem Visual Studio – Projekt zum Beispiel ist hier zu finden:

…\BspUeb\C# für Fortgeschrittene\ref\Span

15.4 Übungsaufgaben zum Kapitel 15

1) In der folgenden Klasse wird eine Methode namens `Reverse()` zum Invertieren von Zeichenfolgen definiert und zum Palindrom-Test verwendet:[1]

```
using System.Diagnostics.CodeAnalysis;

class Prog {
  static string? Reverse(string? original) {
    if (original == null)
      return null;
    var charArray = original.ToCharArray();
    Array.Reverse(charArray);
    return new string(charArray);
  }

  static void Main(string[] args) {
    if (Reverse(args[0]).ToUpper() == args[0].ToUpper())
      Console.WriteLine($"{args[0]} ist ein Palindrom mit {args[0].Length} Buchstaben");
  }
}
```

Im Nullable-Kontext ignoriert der Compiler die Gefahr eines nicht-existenten Befehlszeilenarguments im Array `args` (vgl. Abschnitt 15.1.2), äußert aber eine Dereferenzierungswarnung zur `Reverse()` – Rückgabe. Wie lässt sich diese Warnung durch eine Attributvergabe vermeiden?

2) In der empirisch-statistischen Forschung sind metrische Merkmale (z. B. Alter in Jahren) meist nützlicher als geordnet-kategoriale Merkmale (z. B. Alter in fünf Kategorien). Allzu oft beschränkt sich schon die Datenerfassung auf die geordnet-kategoriale Information. Dabei ist es leicht, bei Bedarf aus einem metrischen Merkmal eine informationsreduzierte geordnet-kategoriale Variante zu erstellen, z. B. mit der folgenden Recodierungsvorschrift:

Altersintervall	Kategorie
< 15	1
[15, 30)	2
[30, 45)	3
[45, 60)	4
≥ 60	5

Führen Sie diese Recodierung per Musterabgleich durch.

---

[1] Ein Palindrom besitzt in beiden Leserichtungen dieselbe Buchstabensequenz, z. B. Reittier.
Das Verfahren zum Invertieren von Zeichenfolgen stammt von der Webseite:
https://code-maze.com/csharp-reverse-string/

3) In eine Stadt dürfen nur PKWs einfahren, die entweder mit mindestens drei Personen besetzt oder elektrisch angetrieben sind. Verwenden Sie z. B. die (wenig vorbildliche) Klasse

```
public class PKW {
    public int Passagiere;
    public bool Elektromobil;
    public bool Zugelassen() { ... }
}
```

und implementieren Sie die Methode `Zugelassen()`.

16 Veröffentlichung von .NET – Software

In diesem Kapitel wird die Veröffentlichung von .NET – Software mit dem Visual Studio beschrieben. Wer VS Code bevorzugt, erzielt dieselben Ergebnisse per dotnet-CLI (Befehl **dotnet publish**).[1] Während bisher bei der Programmerstellung vor allem gute Voraussetzungen zum Erlernen der Programmiersprache C# wichtig waren, steht nun die Bereitstellung einer kundenfreundlichen Lösung im Vordergrund.

16.1 Optionen

Die Faktoren mit Einfluss auf ein zu veröffentlichendes Programm sind teilweise schon behandelt worden. Schließlich haben wir von Beginn an funktionstüchtige und prinzipiell auch für Kunden verwertbare Programme hergestellt. In diesem Abschnitt beschreiben wir die vom Visual Studio in einem Veröffentlichungsprofil präsentierten Optionen. Im Abschnitt 16.2 wird eine Veröffentlichung mit einem attraktiven Format (ReadyToRun-Einzeldatei) vorgeführt.

16.1.1 Konfiguration

Bei der Programmerstellung hängt das Verhalten des C# - Compilers von der Projektkonfiguration ab:

- **Debug-Konfiguration**
  Per Voreinstellung ist die Debug-Konfiguration aktiv, sodass der Compiler ein gut testbares, aber nicht leistungsoptimiertes Assembly im Projektunterordner **…\bin\Debug** erstellt.
- **Release-Konfiguration**
  Ist die Release-Konfiguration aktiv, dann erstellt der Compiler ein leistungsoptimiertes, aber weniger gut testbares Assembly im Projektunterordner **…\bin\Release**.

Wegen der besseren Performanz sollten Kunden in der Regel das Erstellungsergebnis der Release-Konfiguration erhalten (vgl. Abschnitt 3.3.8.3 zu weiteren Details).

16.1.2 Zielframework

Der **TagetFramework**-Eintrag in der Projektdatei definiert, welche Laufzeitumgebung (.NET – Implementation mit Version, Anwendungstyp) für die Ausführung des zu erstellenden Assemblies erforderlich ist, z. B.:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Bei der Version wird auch aktuellerer Wert akzeptiert.

Durch das Zielframework sind die für das Assembly verfügbaren .NET - APIs festgelegt.

Die zur Definition des Zielframeworks verwendete Zeichenfolge kann neben der obligatorischen .NET – Implementation samt Version (z. B. **net6.0**) optional …

---

[1] https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-publish
https://learn.microsoft.com/en-us/dotnet/core/deploying/deploy-with-cli

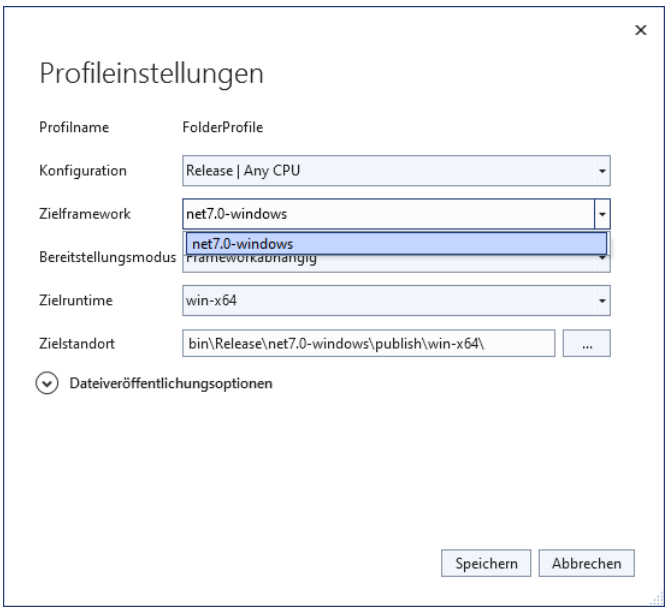
- auch ein Betriebssystem (z. B. **net6.0-windows**, **net6.0-ios**)
- und eine Betriebssystemversion (z. B. **net6.0-ios15.0**)

enthalten.[1] In diesem Fall werden neben den .NET - APIs auch Betriebssystem-APIs in die Zielframework-Definition einbezogen, sodass z. B. eine WPF-Bedienoberfläche möglich ist. Eine fehlende Betriebssystemversion wird durch eine vom SDK festgelegte Minimalversion ersetzt (z. B. die Windows-Version 7 bei .NET 6 bzw.7).

In der Regel lassen wir die Projektdatei samt **TargetFramework**-Eintrag als Bestandteil eines neuen Projekts vom Visual Studio erstellen, wobei wir eine Projektvorlage und eine .NET – Version wählen. So ist z. B. die folgende Projektdatei zu einer WPF-Anwendung für .NET 7 entstanden:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net7.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

</Project>
```

Bei der späteren Veröffentlichung der Software mit dem Visual Studio kann man in den Einstellungen des Veröffentlichungsprofils (vgl. Abschnitt 16.2) das **Zielframework** nicht ändern, z. B.:



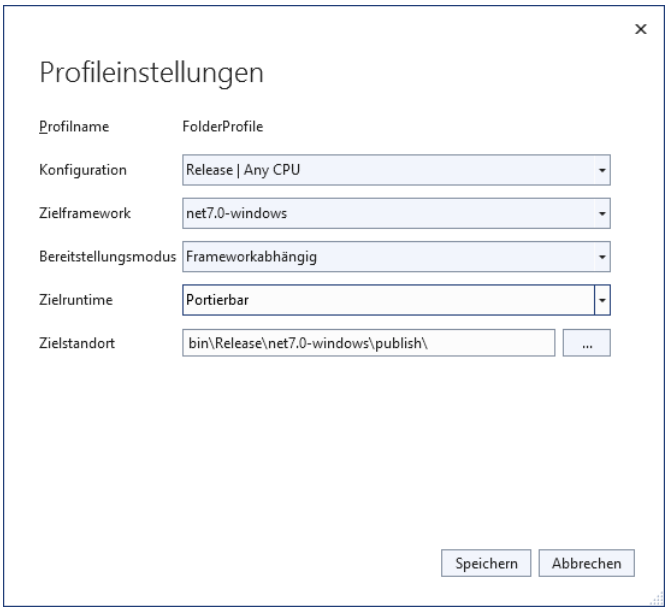
Die zur Definition des Zielframeworks verwendete Zeichenfolge wird *Target Framework Moniker* (*TFM*) genannt.[2]

Weil es sich um ein WPF-Programm handelt, ist mit **net7.0-windows** ein betriebssystem-spezifisches Zielframework erforderlich. Durch den TFM mit Windows-Bezug sind aber andere Betriebssysteme formal nicht ausgeschlossen. Es muss insbesondere keine **Zielruntime** (vgl. Abschnitt 16.1.3) mit **win** im Runtime-Bezeichner (RID) angegeben werden, z. B.:

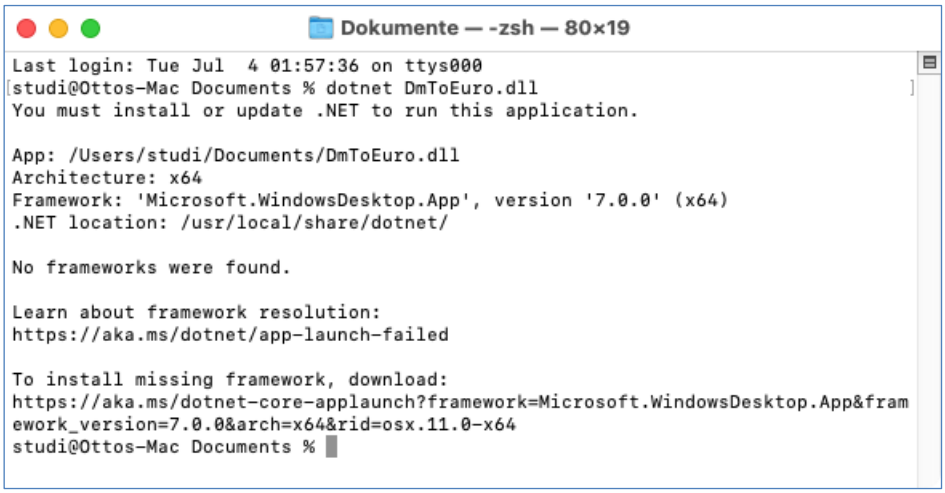
---

1 https://learn.microsoft.com/en-us/dotnet/standard/frameworks

2 https://learn.microsoft.com/de-de/dotnet/standard/frameworks
Die nicht Fachsprachen-taugliche Übersetzung von *moniker* durch *Spitzname* wird von Microsoft im Übersetzungsvorschlag *Zielframeworkmoniker* vermieden.



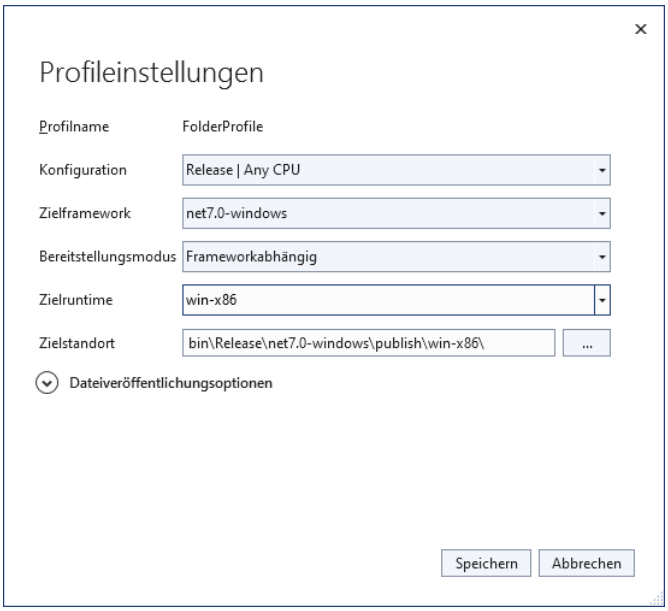
Auf dem ausführenden Rechner wird aber wegen des Zielframeworks **net7.0-windows** eine Laufzeitumgebung mit dem Anwendungstyp **Microsoft.WindowsDesktop.App** vorausgesetzt. Auf einem Mac beschwert sich das zur Ausführung des Programms aufgeforderte dotnet-CLI nicht über das falsche Betriebssystem, sondern über die fehlende Laufzeitumgebung:



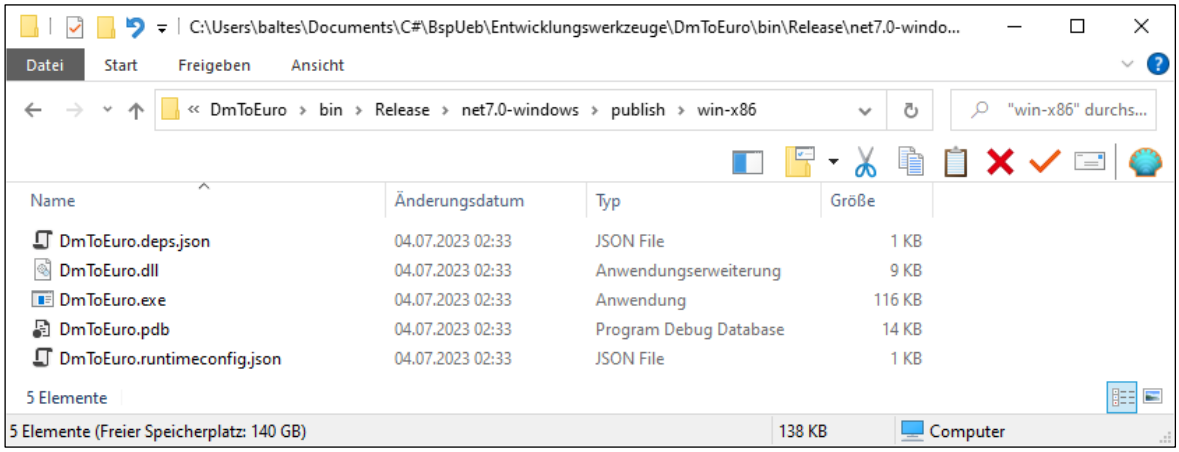
16.1.3 Zielruntime

In .NET 7 wird per Voreinstellung bei der Veröffentlichung (auch) ein Maschinencode-Programm (als Starthelfer oder als AOT-Übersetzungsergebnis) geliefert (siehe Abschnitt 16.1.4). Folglich muss als sogenannte *Zielruntime* über einen *Runtime-Bezeichner* (*RID*) eine Kombination aus einem Betriebssystem und einer CPU-Architektur bekannt sein (z. B. **osx-x64**, **win-x86**).[1] Bei der Veröffentlichung mit dem Visual Studio (vgl. Abschnitt 16.2) kann die **Zielruntime** im Dialog mit den **Profileinstellungen** angegeben werden, z. B.:

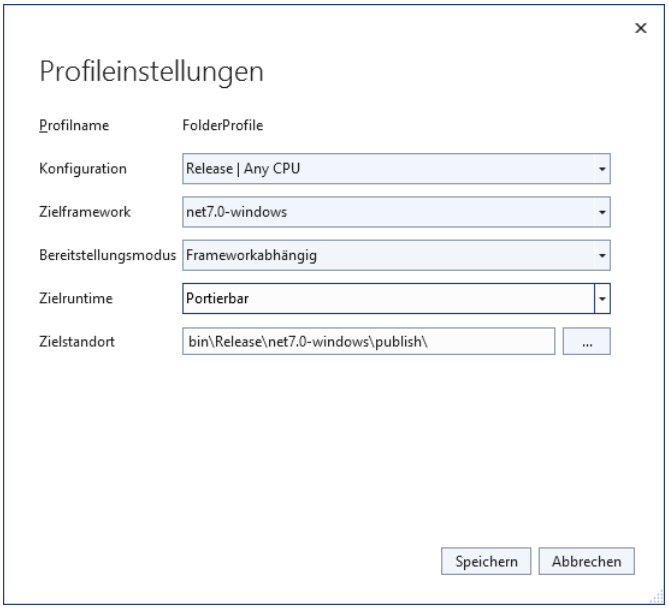
[1] https://learn.microsoft.com/en-us/dotnet/core/rid-catalog



Im Beispiel mit dem Währungskonverter (siehe Abschnitt 3.3.7) resultieren das ausführbare Assembly **DmToEuro.dll** sowie das 32-Bit – Startprogramm **DmToEuro.exe**:



Aus dem folgenden Veröffentlichungsprofil mit der deklarierten Zielruntime **Portierbar**



resultieren unter Win-64 *dasselbe* ausführbare Assembly sowie ein *64-Bit* – Starthelfer. Die Zielruntime **win-x86** anzugeben, hat also den kleinen Vorteil, dass der Starthelfer auf *jedem* Windows-Rechner (mit 32 oder 64 Bit) funktioniert.

Wegen der drohenden Verwechslungsgefahr werden die beiden Begriffe **Zielruntime** und **Zielframework** noch einmal gegenübergestellt:

- Das **Zielframework** legt fest, welche APIs ein Projekt verwendet. Werden auch betriebssystem-spezifische APIs benötigt (z. B. WPF), dann sind …
  - die .NET – Implementation,
  - sowie das Betriebssystem und (implizit) seine Version

  anzugeben (z. B. **net7.0-windows**). Auf dem ausführenden Rechner muss eine Laufzeitumgebung mit allen benötigten APIs vorhanden sein.
- Die **Zielruntime** wirkt sich …
  - bei einem traditionellen IL-Assembly auf den mitgelieferten Starthelfer aus.
  - bei einem AOT-Übersetzungsergebnis auf das R2R-Assembly (mit Maschinencode) bzw. auf das NativeAOT-Programm aus.

  Man muss das Betriebssystem (optional mit Version), die Adressbreite und die CPU-Architektur angeben (z. B. **win-x64**, **win-arm64**, **win10-x64**, **osx-arm64, osx.13-x64**).

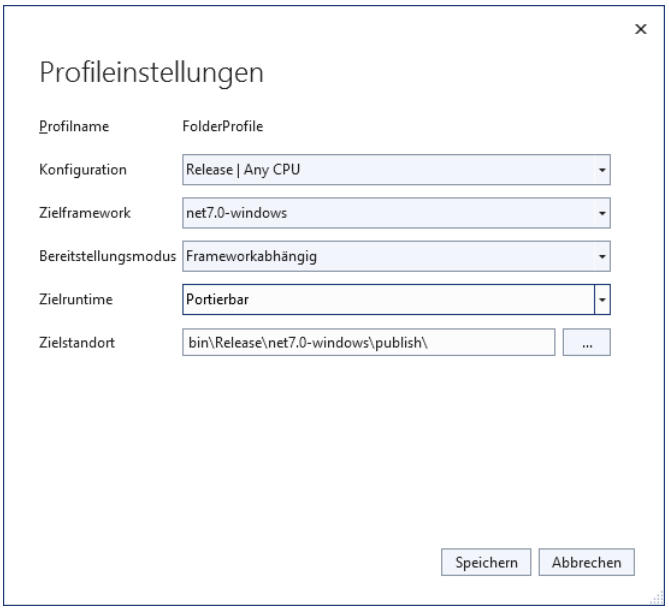
16.1.4 Bereitstellungsmodi

Bei der Veröffentlichung von .NET – Anwendungen sind drei Bereitstellungsmodi zu unterscheiden:

- **Framework-abhängig** (engl.: *Framework-Dependent Deployment*, *FDD*)
  Auf einem Installationsrechner wird eine Laufzeitumgebung mit passender Version (gemäß **TagetFramework**-Eintrag in der Projektdatei, siehe Abschnitt 16.1.2) vorausgesetzt. Es wird eine **dll**-Datei mit dem ausführbaren Assembly ausgeliefert, die per dotnet-CLI zu starten ist, z. B.:

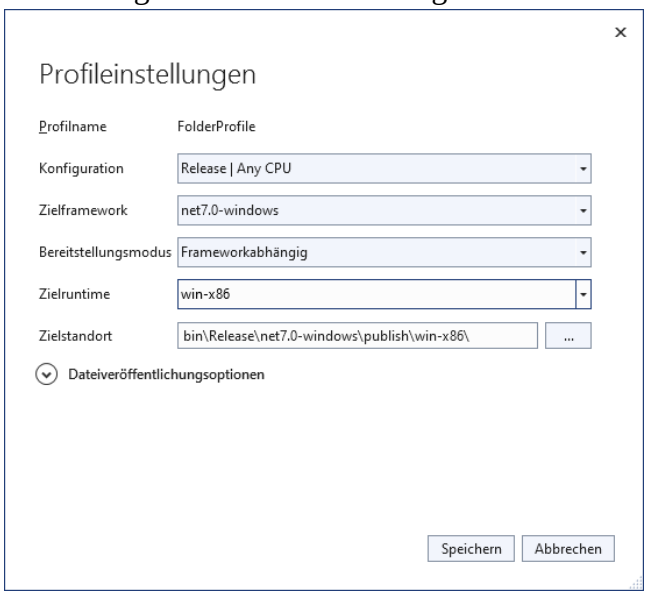
  ```
  >dotnet dmtoeuro.dll
  ```

  Bei der Veröffentlichung mit dem Visual Studio (vgl. Abschnitt 16.2) eignen sich die folgenden Profileinstellungen:



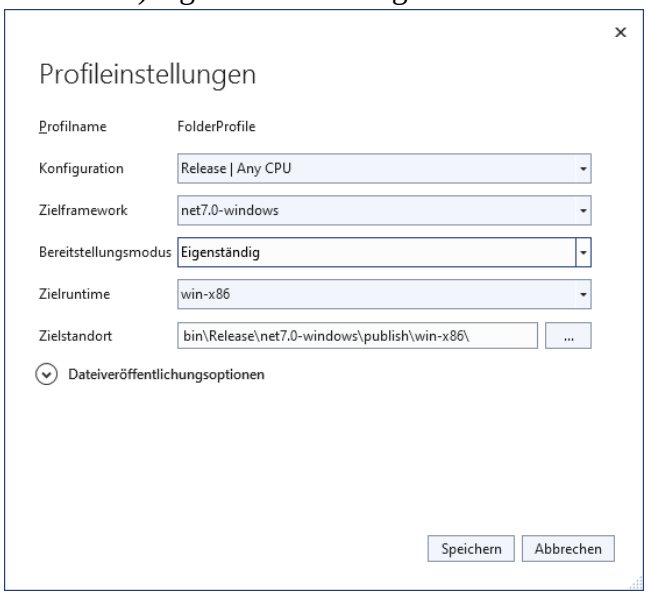
Der Lieferumfang für das `DmToEuro`-Beispielprogramm (vgl. Abschnitt 3.3.7) beträgt 23 KB.[1]

- **Framework-abhängig mit Starter** (engl.: *Framework-Dependent Executable, FDE*)
  Wie beim FDD-Modus wird auf einem Installationsrechner eine Laufzeitumgebung mit passender Version (gemäß **TagetFramework**-Eintrag in der Projektdatei, siehe Abschnitt 16.1.2) vorausgesetzt. Im Unterschied zum FDD-Modus wird zum bequemen Starten des ausführbaren Assemblies (z. B. per Doppelklick) *zusätzlich* ein Maschinencode-Programm (z. B. **DmToEuro.exe**) geliefert. Bei der Veröffentlichung mit dem Visual Studio (vgl. Abschnitt 16.2) eignen sich die folgenden Profileinstellungen:



Der Lieferumfang für das `DmToEuro`-Beispielprogramm beträgt 138 KB.

- **Eigenständig**
  Das Programm wird inklusive Laufzeitumgebung ausgeliefert, setzt also keine Laufzeitumgebung auf dem Installationsrechner voraus. Wie beim FDE-Modus wird zum bequemen Starten ein Maschinencode-Programm mitgeliefert. Bei der Veröffentlichung mit dem Visual Studio (vgl. Abschnitt 16.2) eignen sich die folgenden Profileinstellungen:



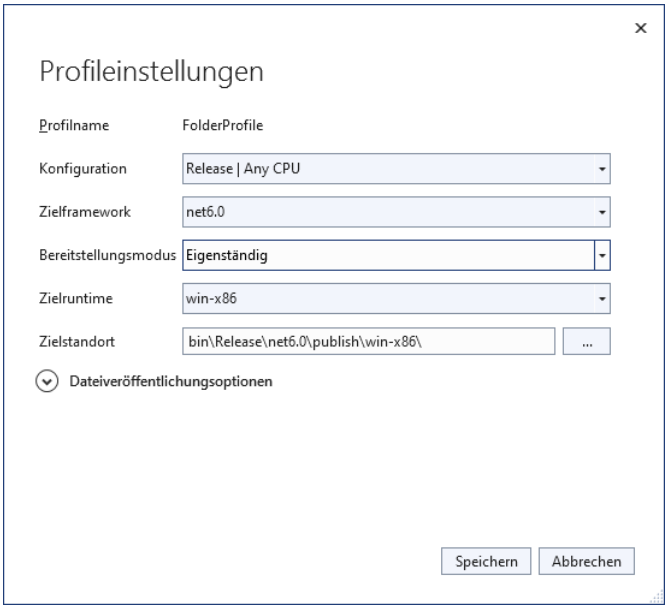
Der Lieferumfang für das `DmToEuro`-Beispielprogramm beträgt 145 MB.

---

[1] Die nicht angeforderte, bei der Veröffentlichung per Visual Studio aber trotzdem erstellte Datei **DmToEuro.exe** wurde gelöscht.

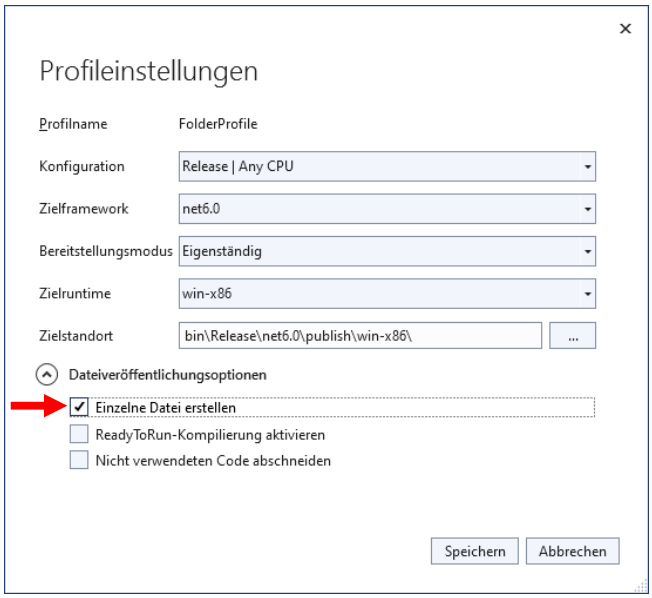
16.1.5 Veröffentlichung von Einzeldateien

Wir betrachten in diesem Abschnitt (wie auch im Abschnitt 16.2) als Beispiel das Konsolenprogramm zur Bruchaddition, dessen Quellcode Sie aus dem Kapitel 1 kennen. Wenn das Programm mit dem Visual Studio anhand der folgenden Profileinstellungen als eigenständige Anwendung veröffentlicht wird,

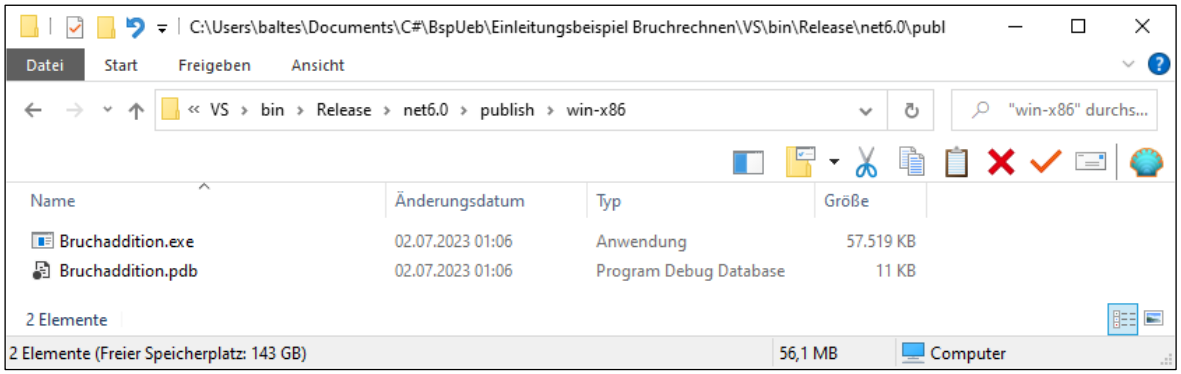


dann resultiert ein Lieferumfang von 62,3 MB, wobei vor allem die Zusammensetzung aus 228 Dateien stört. Durch den Verlust einer Datei kann es zu schlecht aufzuklärenden Störungen kommen. Vermutlich bevorzugen es die meisten Kunden, *eine einzige* Datei zu erhalten.

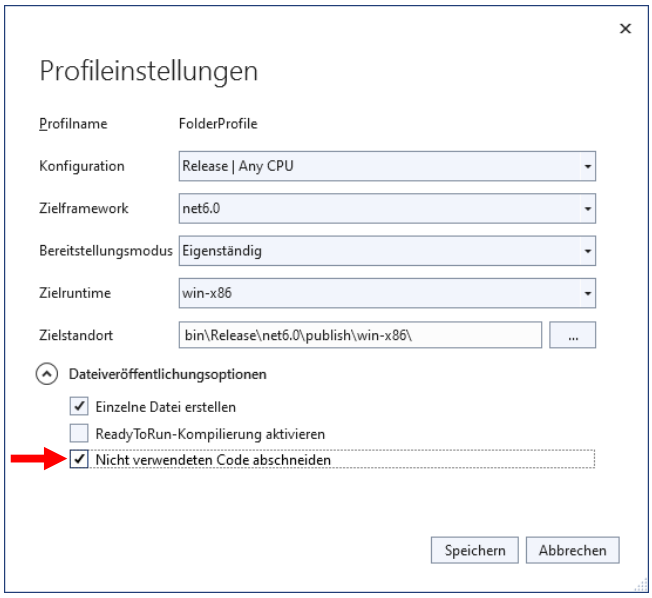
Dieser Wunsch ist leicht zu erfüllen, z. B. im Visual Studio – Dialog mit dem Veröffentlichungsprofil:



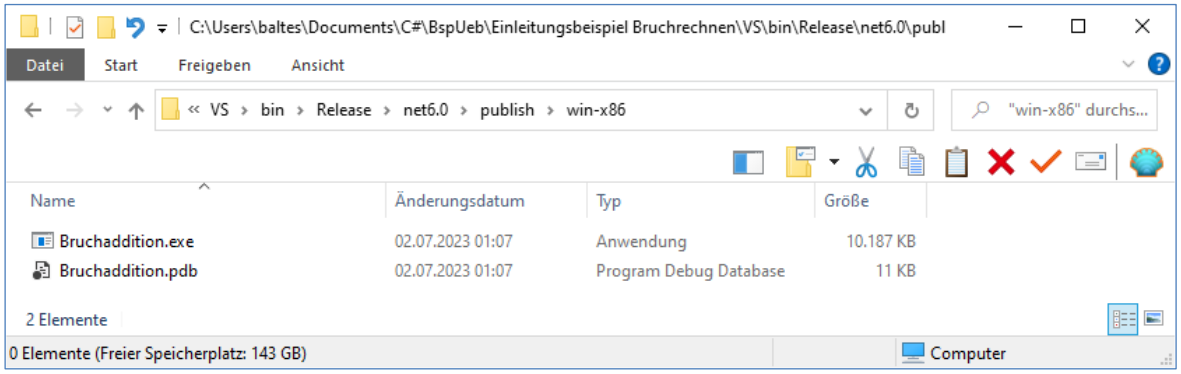
Nun resultiert eine nicht nur eigenständige, sondern vor allem auch sehr übersichtliche Veröffentlichung, die im Wesentlichen aus der Datei **Bruchaddition.exe** besteht:



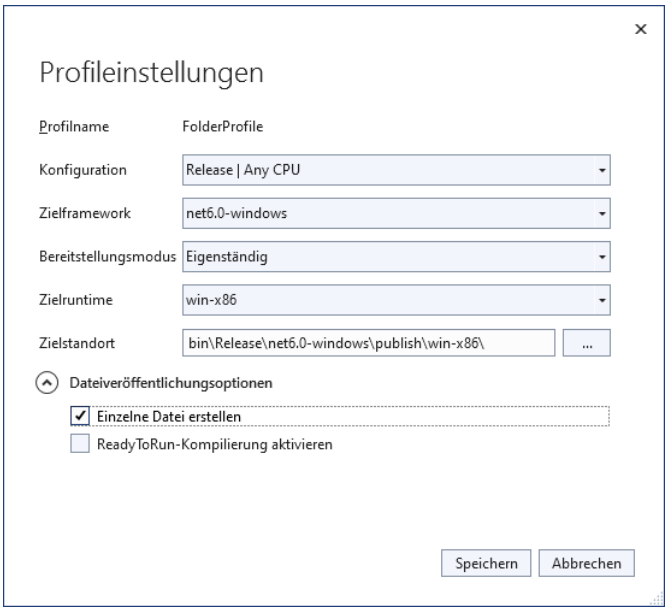
Die Programmdatei enthält mit der kompletten BCL sehr viel nicht verwendeten IL-Code, und seit .NET 6.0 kann dieser Code entfernt werden:



Durch das sogenannte *Trimmen* schrumpft die Datei erheblich:

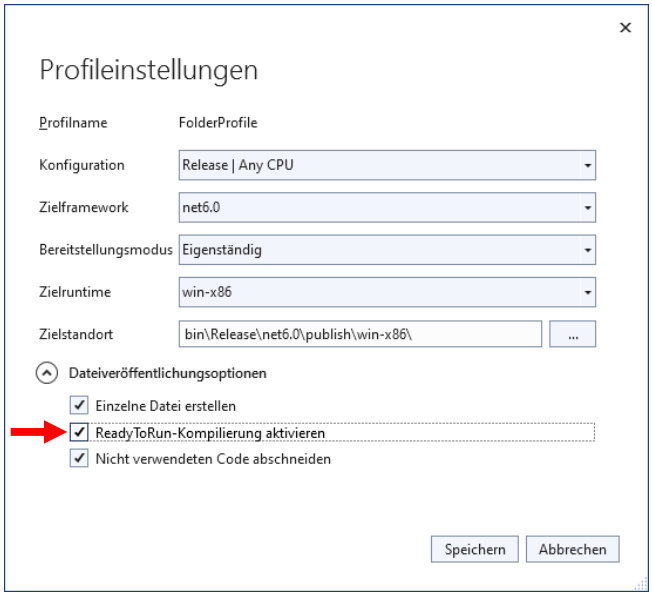


Der Versuch, eine *WPF-Anwendung* bei der Veröffentlichung per Visual Studio zu trimmen, scheitert leider am fehlenden Kontrollkästchen:



16.1.6 ReadyToRun-Übersetzung

.NET – Software wird normalerweise als eine Sammlung von Assemblies mit IL-Code ausgeliefert, den der JIT-Compiler der CLR in Maschinencode übersetzt. Alternativ erlaubt der .NET – Erstellungsprozess aber auch eine AOT – Übersetzung (Ahead Of Time) in den Maschinencode einer bestimmten Kombination aus einem Betriebssystem und einer CPU-Architektur (z. B. **win-x64**, **osx-x64**). In der aktuellen .NET – Version 7 sind mit R2R (ReadyToRun) und NativeAOT *zwei* AOT-Verfahren verfügbar, die im Abschnitt 2.5.2 vergleichend beschrieben werden. In der mit Langzeitunterstützung ausgestatteten .NET – Version 6 ist nur die R2R-Option vorhanden, die im Visual Studio in jedem Fall (d. h. bei .NET 6/7) bequemer zugänglich ist über ein Kontrollkästchen in einem Veröffentlichungsprofil:



Im Abschnitt 16.2 folgt ein komplettes Beispiel für die R2R-Veröffentlichung.

16.2 Veröffentlichung einer ReadyToRun-Einzeldatei-Anwendung

Auch in Zeiten von .NET 7 ist von den Optionen zur AOT-Übersetzung in der Regel das R2R-Format (Ready To Run) gegenüber der Alternative NativeAOT zu bevorzugen (vgl. Abschnitt 2.5.2):

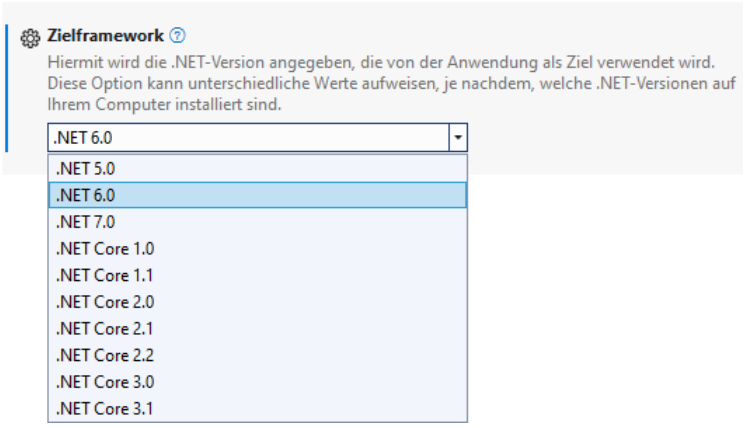
- Wenn .NET 6 wegen seiner Langzeitunterstützung bevorzugt wird, steht nur R2R zur Verfügung.
- R2R unterstützt neben Linux und Windows auch macOS.
- Unter Windows können auch WPF- und WinForms-Anwendungen im R2R-Format veröffentlicht werden.
- Das R2R-Format bietet die Kombination von Maschinencode und IL-Code, sodass bei langlaufenden Anwendungen die Vorteile des JIT-Compilers erhalten bleiben.

In diesem Abschnitt wird die Veröffentlichung einer R2R-Konsolenanwendung in einer einzelnen Datei beschrieben. Als Beispiel verwenden wir das Programm zur Bruchaddition, dessen Quellcode Sie aus dem Kapitel 1 kennen. Ein Visual Studio – Projekt mit diesem Beispiel (siehe Abschnitt 5.1.3 zur Erstellung) befindet sich in diesem Ordner:

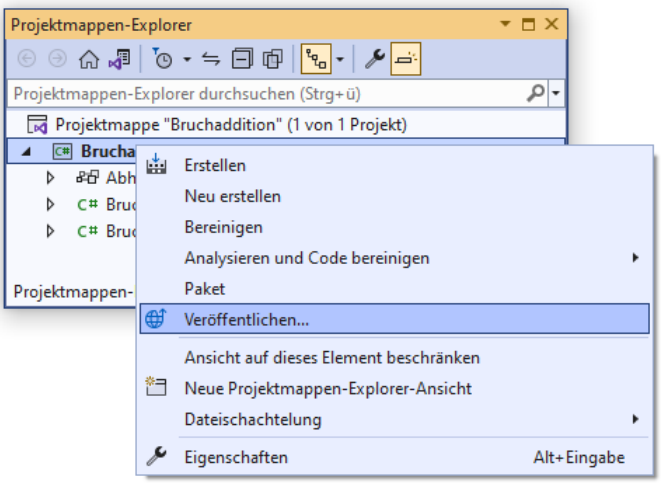
…\BspUeb\Einleitungsbeispiel Bruchrechnen\VS

Eine R2R-Anwendung ist *nicht* portabel, sondern auf die beim Veröffentlichen angegebene Zielruntime angewiesen. Allerdings ist die Veröffentlichung für *mehrere* Zielruntimes kein großer Aufwand, was am Beispiel für die Zielruntimes **win-x86** und **osx-x64** demonstriert wird.

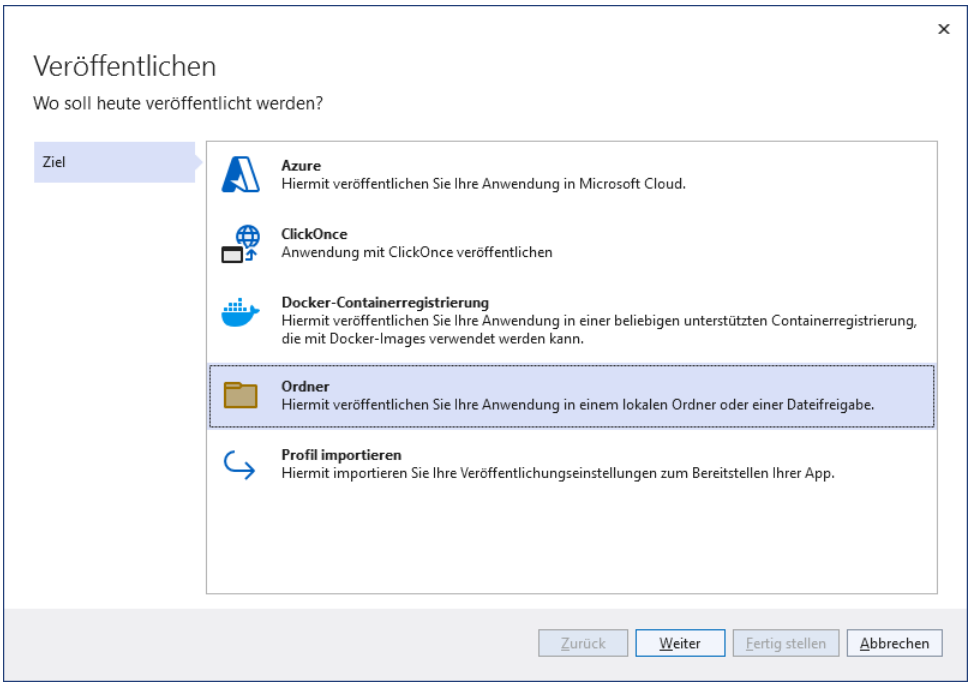
Zunächst wählen wir über den **Eigenschaften**-Dialog des Projekts die .NET – Version 6 wegen der Langzeitunterstützung:



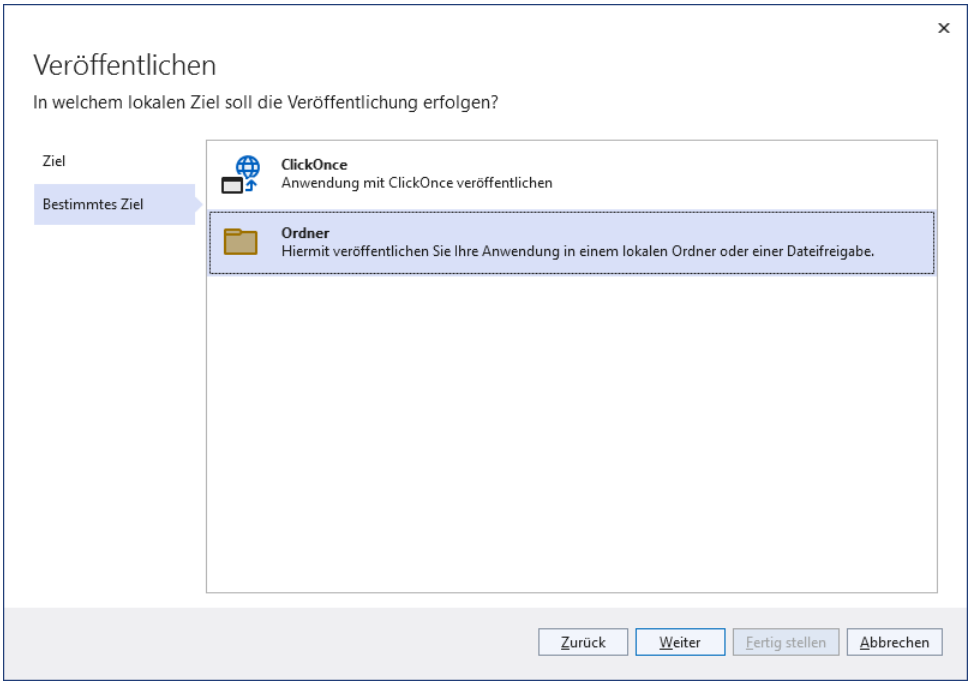
Wir starten das **Veröffentlichen** über das Kontextmenü zum Projekteintrag im **Projektmappen-Explorer**, z. B.:



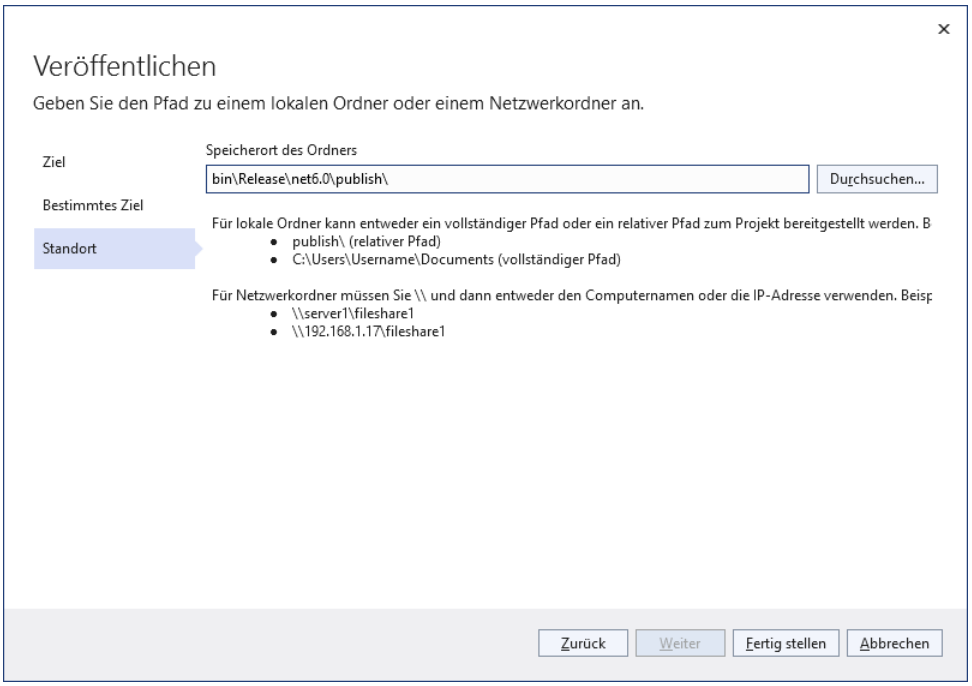
Im ersten Dialog entscheiden wir uns dafür, die veröffentlichte Anwendung in einem lokalen **Ordner** abzulegen:



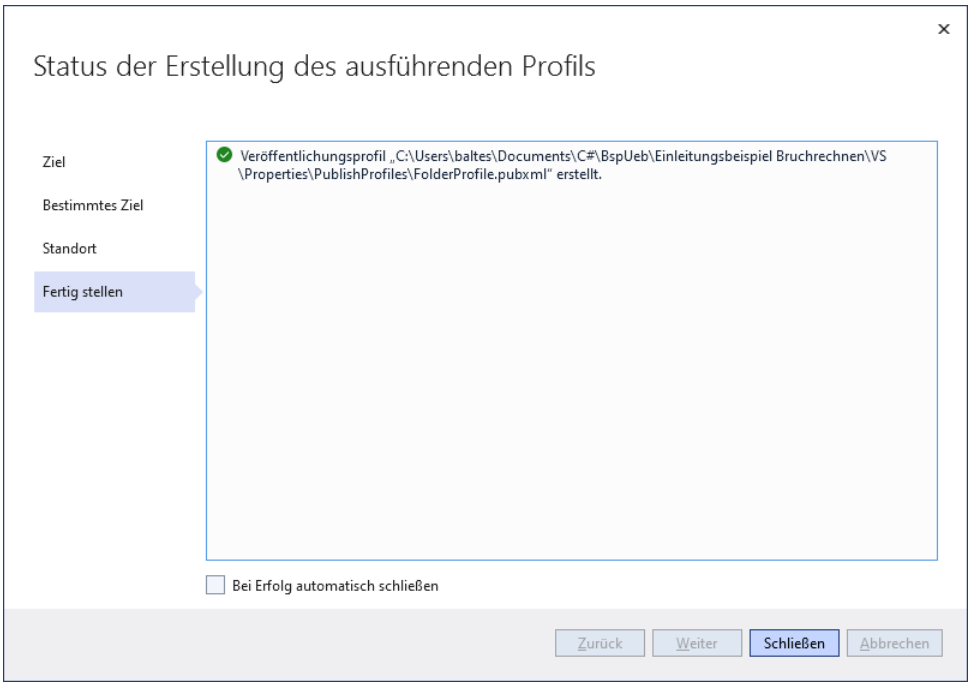
Wir machen **weiter** und bleiben bei unserer Entscheidung:



Der **Speicherort des Ordners** kann beibehalten werden:

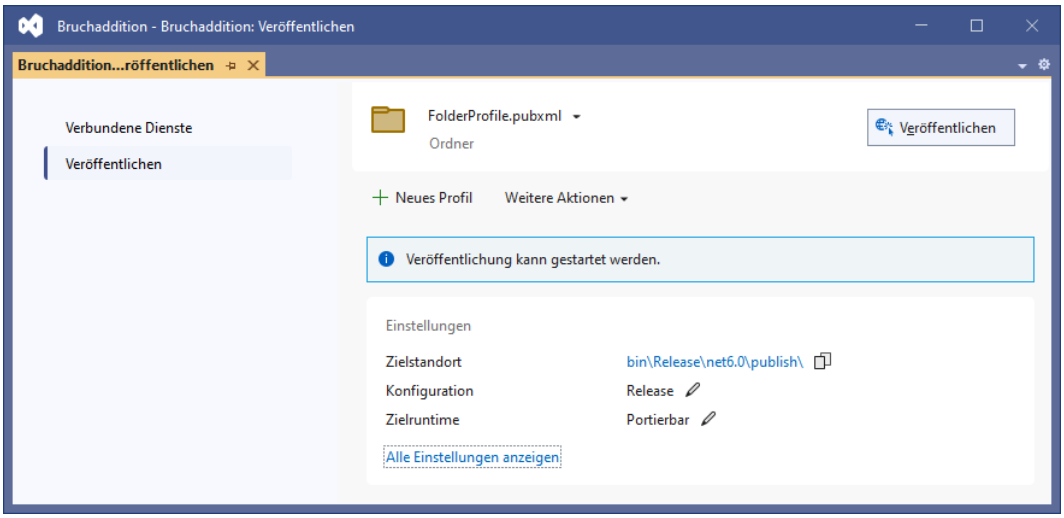


Nach einem Klick auf **Fertig stellen** ist das **Veröffentlichungsprofils** erstellt,

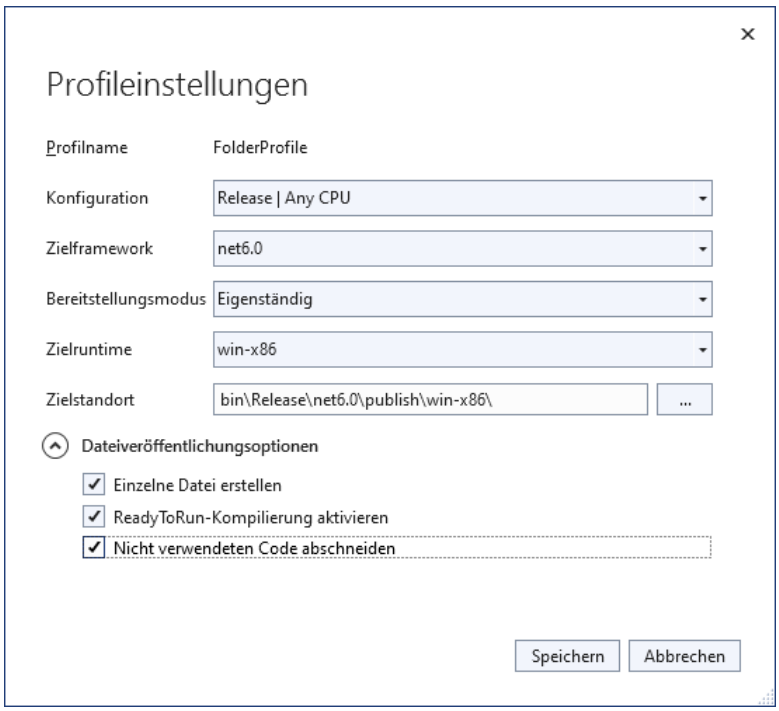


und wir **schließen** das Fenster.

Im folgenden Fenster bleiben wir auf der Registerkarte **Veröffentlichen** und klicken auf den Link **Alle Einstellungen anzeigen**:



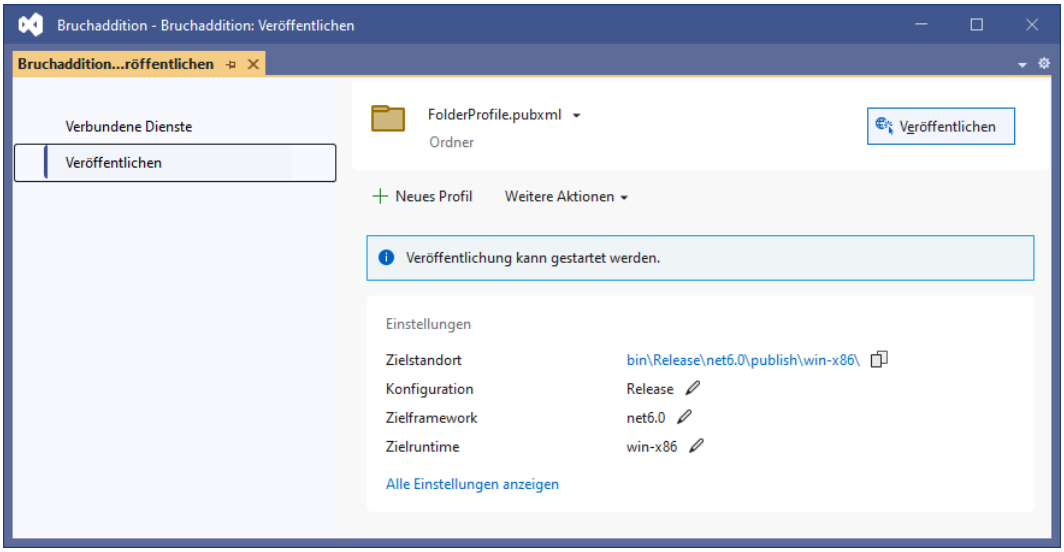
Es erscheint ein Dialog mit **Profileinstellungen**:



- Die **Konfiguration**, das **Zielframework** und der **Zielstandort** können beibehalten werden. Der Assistent hat insbesondere die bei einer Veröffentlichung sinnvolle **Release**-Konfiguration voreingestellt, obwohl im Visual Studio **Debug** als **Projektmappenplattform** aktiv ist (vgl. Abschnitt 3.3.8.4).
- Als **Bereitstellungsmodus** wählen wir **Eigenständig** (statt der Voreinstellung **Frameworkabhängig**), damit auf dem Einsatzrechner keine Laufzeitumgebung erforderlich ist.
- Als **Zielruntime** wählen wir **win-x86**, sodass die Anwendung auf jedem Windows-Rechner (mit 32 oder 64 Bit) nativ ausgeführt werden kann. Das Visual Studio bietet die folgenden Alternativen an:

- Wir markieren alle **Dateiveröffentlichungsoptionen**:
  - **Einzelne Datei erstellen**
  - **ReadyToRun-Kompilierung aktivieren**
    Mit dieser Einstellung wird die AOT-Kompilierung in das R2R-Format angefordert (vgl. Abschnitt 2.5.2.1).
  - **Nicht verwendeten Code abschneiden**
    Durch das Entfernen von überflüssigem Code wird die Größe der Ausgabedatei von 68 MB auf 13 MB reduziert.

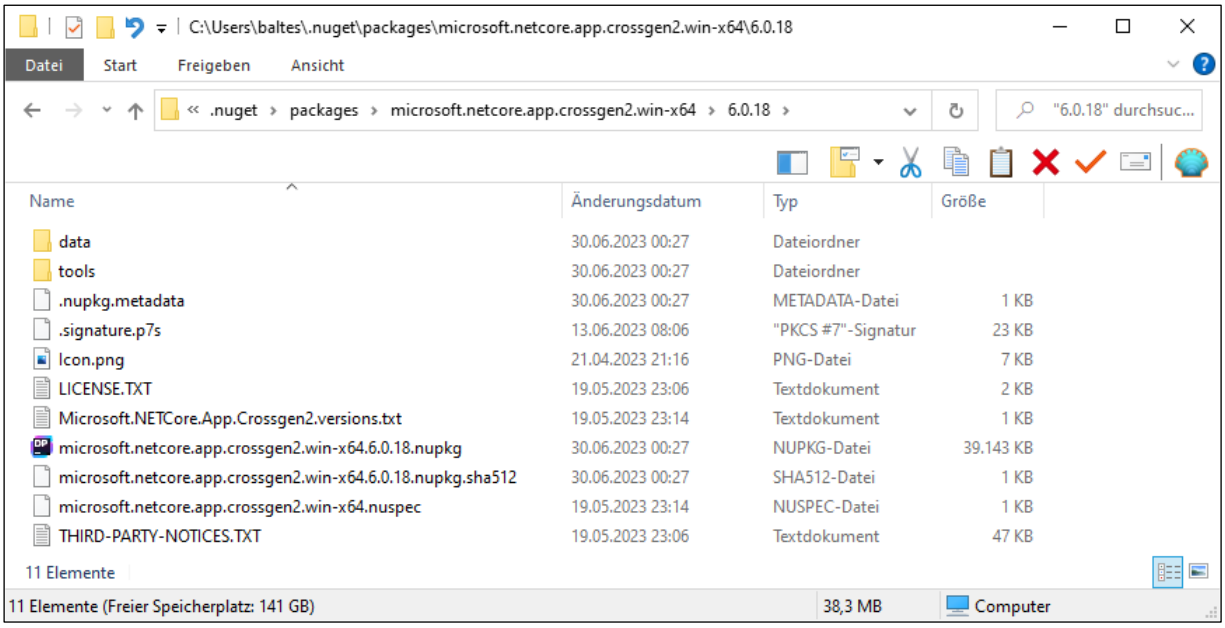
Wir **speichern** den Dialog mit den **Profileinstellungen**, und betätigen auf der Registerkarte **Veröffentlichen** den gleichnamigen Schalter:



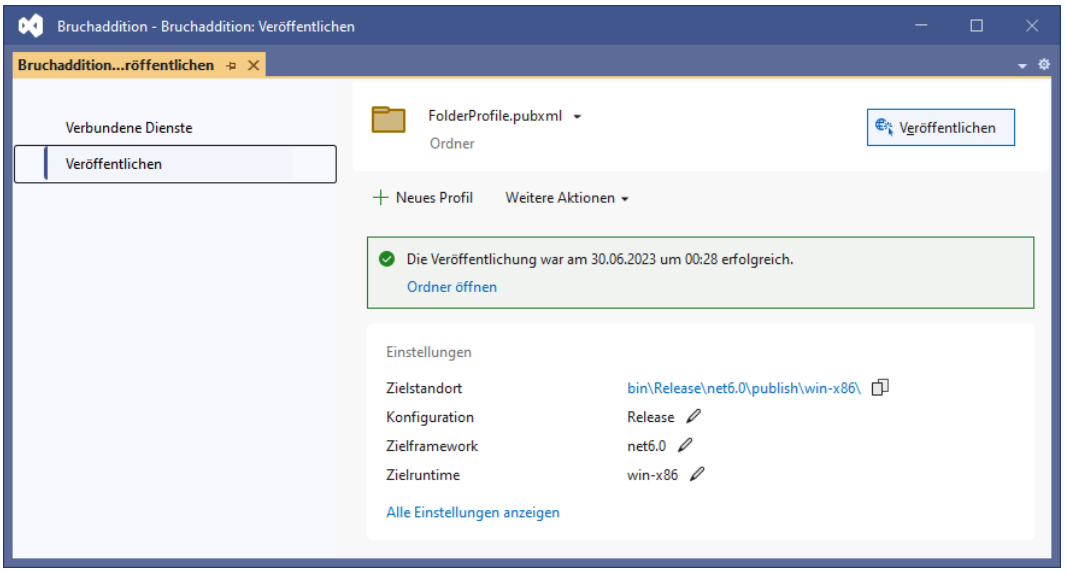
Aus dem umfangreichen Protokoll im **Ausgabe**-Fenster wird nur der Tätigkeitsnachweis des Programms **crossgen2** wiedergegeben. Es übersetzt den IL-Code in Maschinencode für die Zielruntime und kopiert diesen Maschinencode in das Assembly:

```
2>C:\Users\baltes\.nuget\packages\microsoft.netcore.app.crossgen2.win-
x64\6.0.18\tools\crossgen2.exe --targetos:windows
2>--targetarch:x86
```

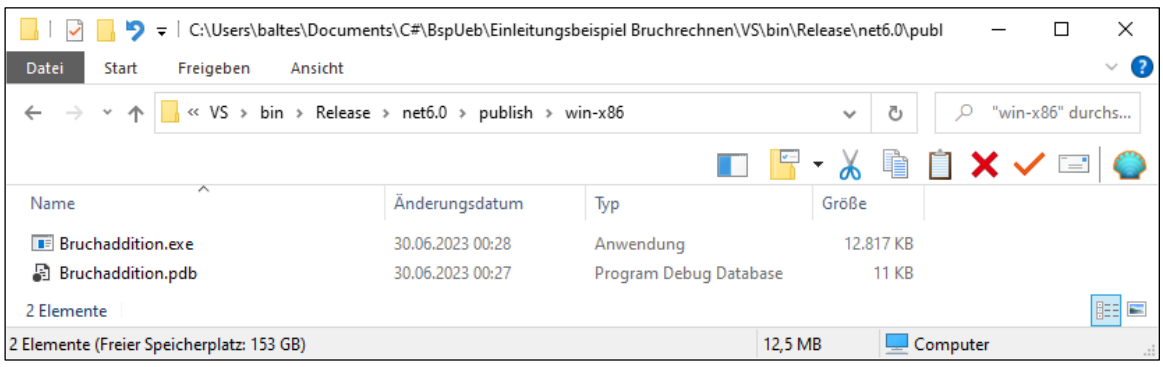
Das Programm **crossgen2** wird automatisch über ein NuGet-Paket installiert:



Das Fenster **Veröffentlichen** zeigt eine Erfolgsmeldung,



und der Ausgabeordner enthält neben der verzichtbaren **pdb**-Datei mit Debug-Hilfen (siehe Abschnitt 3.3.7.5) eine **exe**-Datei von akzeptabler Größe, die auf jedem aktuellen Windows-Rechner ohne Installation ausgeführt werden kann. Den Kunden muss also nur eine einzige Datei übergeben werden:



Dasselbe Ergebnis ist auch per dotnet-CLI zu erreichen:

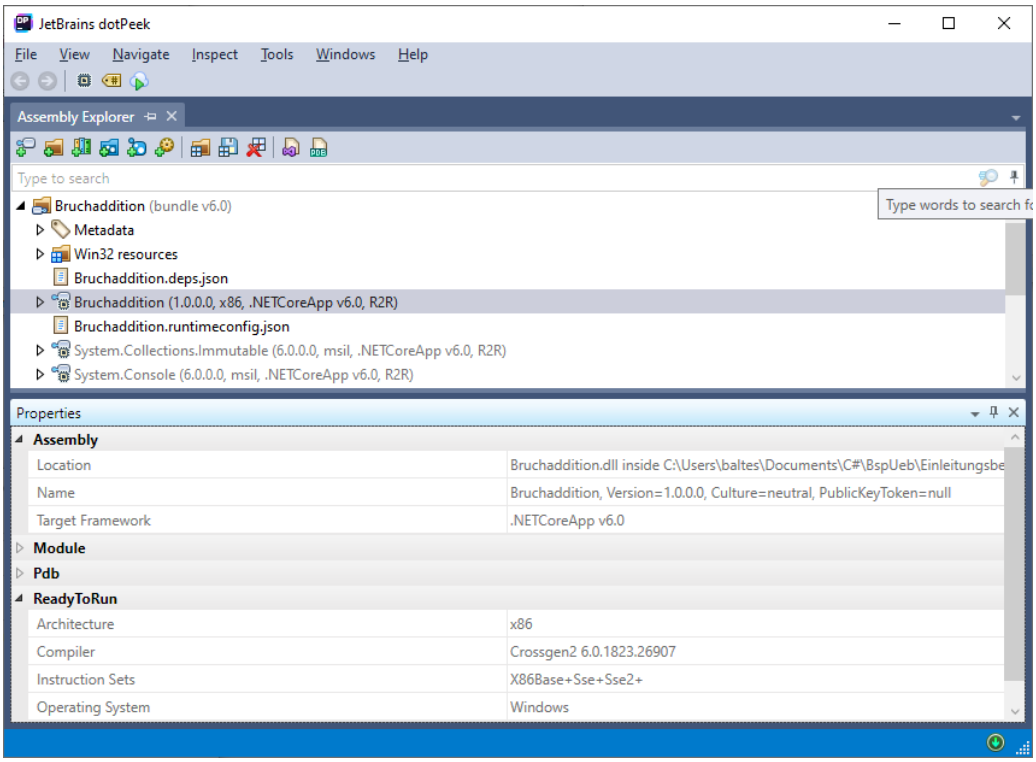
```
>dotnet publish -c release -r win-x86 -p:PublishReadyToRun=true -p:PublishSingleFile=true
--self-contained true -p:PublishTrimmed=true
```

Während sich das Visual Studio 17.6.4 weigert, ein R2R-Kompilat für macOS zu erstellen, gelingt das per dotnet-CLI durch einfaches Austauschen der Zielruntime:

```
>dotnet publish -c release -r osx-x64 -p:PublishReadyToRun=true -p:PublishSingleFile=true
--self-contained true -p:PublishTrimmed=true
```

Das resultierende Programm befindet sich komplett in der Datei **Bruchaddition** und lässt sich auf einem Mac per Doppelklick starten:

Das Diagnoseprogramm dotPeek beschreibt das für Windows erstellte ReadyToRun-Einzeldatei-Programm **Bruchaddition.exe** als *Bundle*, in dem das (von **crossgen2** 6.0.18 erzeugte) R2R-Assembly `Bruchaddition` enthalten ist:



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.[1]

Operator	**Bedeutung**
x.*y*	Member-Zugriff
Methode(*Parameter*)	Methoden- oder Delegatenaufruf
[]	Index-Zugriff
?., ?[	null-bedingter Operator
x++, *x* --	Postinkrement bzw. -dekrement
Ausdruck!	null-Toleranzoperator
new	Objekterzeugung
typeof(*Type*)	Typ eines Bezeichners ermitteln
nameof(*Var*)	Name eines Bezeichners ermitteln
checked, unchecked	Ganzzahl-Überlaufdiagnose
sizeof(*Type*)	Speicherbedarf eines Typs ermitteln
delegate	Anonyme Methode erstellen
default(*Type*)	Standardwert eines Typs Wert ermitteln
-*x*	Vorzeichenumkehr
!	Negation
~	Bitweise Negation
++*x*, --*x*	Präinkrement bzw. -dekrement
(*Type*)*x*	Typumwandlung
await	Auf die Beendigung einer Task warten

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/

Operator	Bedeutung
x..y	Bereich
switch with	**switch**-Ausdruck **with**-Ausdruck
*, / %	Multiplikation, Division Modulo (Divisionsrest)
+, - +	Addition, Subtraktion String-Verkettung
<<, >>	Links- bzw. Rechts-Shift
>, <, >=, <= is, as	Vergleichsoperatoren Typtest
==, !=	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-Koaleszenzoperator
? :	Konditionaloperator

Operator	Bedeutung
=	Wertzuweisung
+=, -=, *=, /=, %=	Verbundzuweisung
??=	Null-Koaleszenz - Zuweisungsoperator
=>	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 & Johannsen (2020, S. 66ff).

B. Lösungsvorschläge zu den Übungsaufgaben

Kapitel 1 (Einstieg in die objektorientierte Software-Entwicklung mit C#)

Aufgabe 1

1. **Richtig**
2. **Falsch**
   Eine Klasse besitzt in der Regel mehrere öffentliche Methoden und somit eine Kommunikationsschnittstelle (ein API). Allerdings werden gelegentlich auch Methoden für den ausschließlich internen Gebrauch benötigt.
3. **Richtig**
4. **Falsch**
   Nur die Startklasse eines C# - Programms besitzt eine statische Methode namens **Main()**. Mit dieser Methode, die von der Laufzeitumgebung aufgerufen wird, startet die Programmausführung.
5. **Falsch**
   Bei realisierter Datenkapselung ist den Methoden fremder Klassen der *direkte* Zugriff auf die Instanzvariablen eines Objekts verboten. Eine Klasse bietet jedoch oft öffentliche Eigenschaften und/oder Methoden an, die kontrollierte Änderungen am Zustand eines Objekts ermöglichen.

Aufgabe 2

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.

Kapitel 2 (Grundzüge der .NET – Technologie)

Aufgabe 1

1. **Falsch**
   .NET – Konsolenprogramme laufen unter Linux, macOS und Windows. In diese Kategorie fallen auch die per ASP.NET Core erstellten Server-Programme (Web-Anwendungen und Web-Dienste). Mit .NET MAUI lassen sich Klienten-Programme mit grafischer Bedienoberfläche erstellen, die unter macOS, Windows, Android und iOS laufen.
2. **Richtig**
3. **Richtig**
   Dazu müssen allerdings die Regeln der Common Language Specification eingehalten werden.
4. **Falsch**
   Eine WPF-Anwendung läuft nur unter Windows. Um eine GUI-Anwendung zu erstellen, die unter macOS, Windows, Android und iOS läuft, wird .NET MAUI benötigt.
5. **Falsch**
   Das Programm sollte mit .NET entwickelt werden, damit die aktuelle C# - Version und die aktuelle BCL zur Verfügung stehen. Als GUI-Lösungen stehen WPF und WinForms zur Verfügung.

Aufgabe 2

Namensräume und Assemblies sind zwei voneinander *unabhängige* Organisationsstrukturen:

- Die Typen eines Namensraums können in verschiedenen Assemblies implementiert werden.
- In einem Assembly können Typen aus verschiedenen Namensräumen implementiert werden.

Aufgabe 3

IL	Ein .NET - Compiler übersetzt den Quellcode nicht in die Maschinensprache einer bestimmten CPU, sondern in die Intermediate Language (IL). Diesen Zwischencode übersetzt der JIT-Compiler der CLR in den Maschinencode der lokalen CPU.
BCL	Die Base Class Library enthält in zahlreichen Assemblies ausgereifte Lösungen für viele Routineaufgaben der Programmierung. Für jeden Anwendungstyp (z. B. Windows-Desktop, ASP.NET Core, MAUI) existiert eine Erweiterung der BCL.
JIT	Der JIT-Compiler in der CLR übersetzt den IL-Code eines Assemblies in den Maschinencode der lokalen CPU. Er arbeitet keinesfalls „auf Lager“, indem er ein Assembly beim ersten Zugriff komplett übersetzt. Stattdessen wird nur der tatsächliche zur Ausführung anstehende IL-Code (just in time) übersetzt, und der erzeugte Maschinencode landet für den Fall einer erneuten Verwendung im Speicher.

Kapitel 3 (Werkzeuge zum Entwickeln von C# - Programmen)

Aufgabe 2

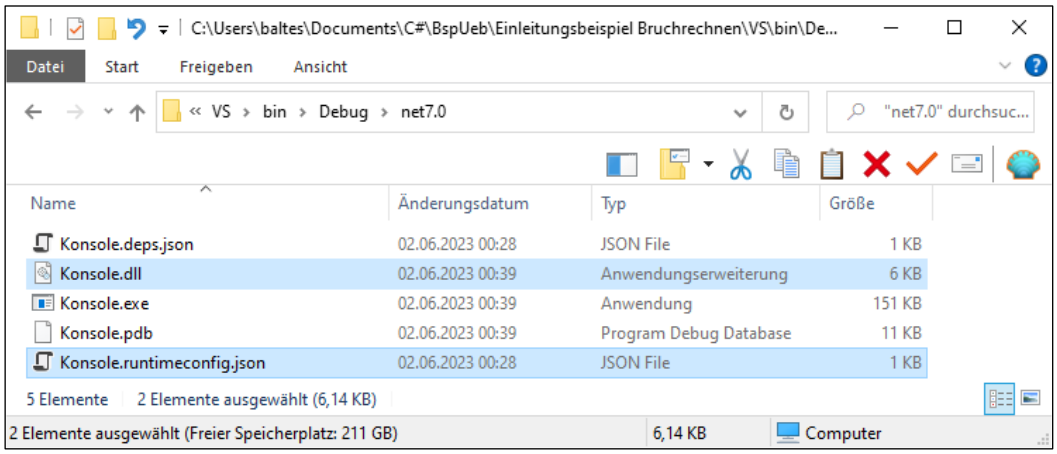
- Das Schlüsselwort **using** wird klein geschrieben.
- Die Startmethode muss den Namen **Main()** haben.
- Der Methodenname **WriteLine()** ist falsch geschrieben.
- Die Zeichenfolge im Parameter des **WriteLine()** - Aufrufs muss mit dem "-Zeichen abgeschlossen werden.
- Die schließende Klammer zum Rumpf der Klassendefinition fehlt.

Aufgabe 3

1. **Richtig**
2. **Richtig**
3. **Richtig**
4. **Falsch**
   Eine WPF-Anwendung kann mit VS Code geöffnet und modifiziert werden, wobei zur Gestaltung der Benutzeroberfläche der XAML-Code editiert werden muss.
5. **Richtig**

Aufgabe 4

Bei einem framework-abhängigen[1] Programm müssen von den Dateien im Ausgabeordner einer Erstellung



mindestens ausgeliefert werden:

- Die **dll**-Datei mit dem Assembly (im Beispiel: **Konsole.dll**)
  Diese Datei enthält u.a. den IL-Code.
- Die Datei mit der Namenserweiterung **runtimeconfig.json**
  Dieser Datei ist u.a. zu entnehmen, welche .NET - Version zur Ausführung des Programms benötigt wird.

Kapitel 4 (Elementare Sprachelemente)

Abschnitt 4.1 (Einstieg)

Aufgabe 1

Der 1. Aufruf scheitert:	Der Methodenname **Main** muss großgeschrieben 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.

[1] Mit dem .NET Framework (vgl. Abschnitt 2.2.1) hat der Begriff *framework-abhängig* nichts zu tun (vgl. Abschnitt 3.1.5).

Aufgabe 2

Unzulässig sind:

- `4you`
  Namen müssen mit einem Buchstaben oder mit einem Unterstrich beginnen.
- `else`
  Schlüsselwörter wie **else** sind als Namen verboten.

Abschnitt 4.2 (Ausgabe bei Konsolenanwendungen)

Aufgabe 1

Von den beiden im **WriteLine()** - Parameter auftretenden Plus-Operatoren

```
Console.WriteLine("3,3 + 2 = " + 3.3 + 2);
```

Bestandteil der Zeichenkette — Erster Plus-Op. — Zweiter Plus-Op.

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
`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 4.3 (Variablen und Datentypen)

Aufgabe 1

Eine lokale Variable ist verfügbar von der deklarierenden Anweisung bis zur schließenden Klammer des Blocks, in dem sich die Deklaration befindet.

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: {pi * radius * radius:f3}");
  }
}
```

Aufgabe 3

Ab C# 11 (.NET 7) kann der **WriteLine()** – Aufruf (gegeben eine explizite oder implizite using-Direktive für den Namensraum **System**) z. B. unter Verwendung eines mehrzeiligen Zeichenfolgenliterals geschrieben werden:

```
Console.WriteLine("""
    Dies ist ein Zeichenfolgenliteral:
        "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';
```

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DmToEuroDec

Für die Eingabe 1000000000000,13 liefert die neue Programmversion das korrekte Ergebnis,

während bei Verwendung des Datentyps **double** nach dem Runden auf 2 Nachkommastellen ein Cent zu viel ausgegeben wird.

Das Auftreten eines Fehlers an der 15. Mantissenstelle steht *nicht* im Widerspruch zur Norm IEEE-754 (64 Bit), die beim Typ **double** eine *Speicherung* mit mind. 15 signifikanten Dezimalstellen in der Mantisse verspricht. Der Speicherfehler wurde durch die Gleitkommadivision vergrößert.

Abschnitt 4.5 (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 Zahl 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 Anweisung auf den Wert 3 erhöhte Variable `i` und setzt sie auf den Wert 4.
- Dies ist auch der Wert des Ausdrucks `++i`, sodass 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 Bindungskraft als die logischen Operatoren und der Zuweisungsoperator, sodass z. B. in der folgenden Anweisung

```
la1 = 2 > 3 && 2 == 2 ^ 1 == 1;
```

auf runde Klammern verzichtet werden konnte. Besser lesbar ist aber die äquivalente Variante:

```
la1 = (2 > 3) && (2 == 2) ^ (1 == 1);
```

`la1` erhält den Wert **false**, denn der Operator ^ besitzt eine höhere Bindungskraft als der Operator **&&**.

`la2` erhält den Wert **true**, weil die runden Klammern dafür sorgen, dass der Operator ^ zuletzt ausgeführt wird.

`la3` erhält den Wert **false**.

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\Exp

Aufgabe 5

i) *a* / *b* + *c* * *d*
Aus der Bindungskraftregel resultiert die folgende implizite Klammerung (Zuordnung der Operanden):

$$(a / b) + (c * d)$$

Aus der Links-vor-rechts - Regel ergibt sich für die Auswertung der Operanden bzw. Ausführung der Operatoren die folgende Reihenfolge:

a, *b*, /, *c*, *d*, *, +

ii) *a* / (*b* + *c*) * *d*

Aus der gemeinsamen Links-Assoziativität der Operatoren **/** und ***** resultiert die folgende implizite Klammerung (Zuordnung der Operanden):

$$(a / (b + c)) * d$$

Aus der Links-vor-rechts - Regel ergibt sich für die Auswertung der Operanden bzw. Ausführung der Operatoren die folgende Reihenfolge:

a, *b*, *c*, +, /, *d*, *

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DmToEuroSep

Abschnitt 4.7 (Anweisungen)

Aufgabe 1

Weil die **else**-Klausel der *zweiten* (nach oben nächstgelegenen) **if**-Anweisung zugeordnet wird, ergibt sich der folgende 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

Aufgabe 4

Das Semikolon am Ende der Zeile

```
while (i < 100) ;
```

wird vom Compiler als die zur **while**-Schleife gehörige (leere) Anweisung interpretiert, sodass 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\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 4.3.5.1)

$$(-1)^0 \cdot 2^{(1023-1023)} \cdot (1+0{,}0)$$

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

$$1{,}0 + 2^{-53} \approx 1{,}0 + 1{,}1102230246251565 \cdot 10^{-16} \approx 1{,}00000000000000011102230246251 57$$

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

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 6

Lösungsvorschläge mit den beiden Algorithmus-Varianten befinden sich in den Ordnern:

...\BspUeb\Elementare Sprachelemente\GGT.Diff

...\BspUeb\Elementare Sprachelemente\GGT.Mod

Kapitel 5 (Klassen und Objekte)

Aufgabe 1

1. **Falsch**
2. **Stimmt**
   Die Methodenüberladung ist erlaubt und nützlich (siehe Abschnitt 5.3.5), steht allerdings teilweise in Konkurrenz mit moderneren Techniken (z. B. generische Methoden, optionale Parameter).
3. **Falsch**
   Es darf kein Rückgabetyp angegeben werden (siehe Abschnitt 5.4.3.1).
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 Klasse, sofern entsprechende Referenzen vorhanden sind. Der Klassendesigner ist für das sinnvolle Verhalten der Methoden verantwortlich.
5. **Stimmt**
6. **Falsch**
   Der Rückgabetyp einer Methode ist irrelevant für ihre Signatur (siehe Abschnitt 5.3.5).

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 geändert werden.

Aufgabe 3

In der Anweisung

```
return Anzahl;
```

wurde `Anzahl` versehentlich großgeschrieben, sodass sich die **get**-Methode der Eigenschaft `Anzahl` selbst aufruft. Diese ungeplante Rekursion führt zu einem Stack-Überlauf (vgl. Abschnitt 5.8.2).

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\Rekursion

Eine Berechnung der Fibonacci-Zahl zum Argument 50 zeigt, dass der rekursive Algorithmus absolut und in Relation zum iterativen Algorithmus sehr viel Rechenzeit verbraucht:

```
Fibonacci-Zahl (1) oder Fakultät (2) berechnen? 1

Fibonacci-Argument (Minimum 0): 50
Rekursiv berechnete Fibonacci-Zahl von 50:       12586269025
Berechnet in 00:06:39.1398495
Anzahl Methodenaufrufe: 40730022147
Iterativ berechnete Fibonacci-Zahl von 50:       12586269025
Berechnet in 00:00:00.0005849

Beenden mit Enter
```

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Einleitungsbeispiel Bruchrechnen\GUI (mit Bruch-Instanzvariable)

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 6 (Weitere .NETte Typen)

Aufgabe 1

Dank Autoboxing kann man einer **object**-Variablen auch einen **int**-Wert zuweisen, wobei ein neues Objekt auf dem Heap entsteht, 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 6.3.1.2.2).

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Arrays\Lotto

Aufgabe 3

Hier finden Sie den Lösungsvorschlag für eine Konsolen-Anwendung:

...\BspUeb\Weitere .NETte Typen\Arrays\Eratosthenes

Damit bei der Konsolenausgabe der Primzahlen keine Umbrüche auftreten, wird der von einer ganzen Zahl `i` inklusive einer folgenden Leerstelle benötigte Platz im Programm folgendermaßen ermittelt:

```
(int) Math.Log10(i) + 2
```

Hier finden Sie den Lösungsvorschlag für eine WPF-Anwendung:

...\BspUeb\WPF\Eratosthenes

Die Behandlungsmethode zum **Click**-Ereignis des Befehlsschalters in der WPF-Anwendung ist erfreulich einfach:

```
private void btnStart_Click(object sender, RoutedEventArgs e) {
    Cursor oldCursor = Cursor;
    Cursor = Cursors.Wait;
    try {
        listBox.ItemsSource = null;
        var items = new List<int>();

        // Kandidaten-Array erzeugen und vorbereiten
        int grenze = Convert.ToInt32(textBox.Text);
        if (grenze < 2 || grenze > Array.MaxLength)
            throw new ArgumentException("Unzulässiges Argument");
        int maxBasis = (int)(Math.Sqrt(grenze) + 0.5);
        bool[] prim = new bool[grenze + 1];
        for (int i = 1; i <= grenze; i++)
            prim[i] = true;

        // Eratosthenes-Sieb
        for (int basis = 2; basis <= maxBasis; basis++)
            if (prim[basis])
                for (int i = 2; i * basis <= grenze; i++)
                    prim[i * basis] = false;

        // Ergebnis ausgeben
        for (int i = 2; i <= grenze; i++)
            if (prim[i])
                items.Add(i);
        listBox.ItemsSource = items;
    } catch (Exception ex) {
        listBox.ItemsSource = null;
                MessageBox.Show(this, ex.Message, "Es ist ein Fehler aufgetreten.",
                                MessageBoxButton.OK, MessageBoxImage.Error);
    } finally {
        Cursor = oldCursor;
    }
}
```

Vor dem Start der potenziell zeitaufwändigen Sieboperation wird der aktuelle Cursor (Mauszeiger) in einer Variablen vom Typ der Klasse **Cursor** (Namensraum **System.Windows.Input**) gesichert. Dann wird der Cursor mithilfe der Eigenschaft **Wait** der Klasse **Cursors** (Namensraum **System.Windows.Input**) ersetzt durch den Wait- bzw. Sanduhr-Cursor:

```
Cursor oldCursor = Cursor;
Cursor = Cursors.Wait;
```

In der **finally**-Klausel der **try-catch-finally** - Anweisung zur Absicherung gegen Ausnahmefehler wird der Standard-Cursor reaktiviert. Damit findet die Reaktivierung auf jeden Fall statt, sowohl beim normalen Ablauf wie auch nach einem Ausnahmefehler.

Wenn die gewünschte Obergrenze des Suchbereichs das erlaubte Maximum für die Anzahl der Elemente eines Arrays übertrifft oder < 2 ist, dann wirft die Methode eine Ausnahme:

```
if (grenze < 2 || grenze > Array.MaxLength)
    throw new ArgumentException("Unzulässiges Argument");
```

Folglich können in der **catch**-Klausel, die sich um beliebige Objekte aus der allgemeinen Ausnahmeklasse **Exception** kümmert, alle Fehler durch denselben Aufruf der Methode **MessageBox.Show()** berichtet werden.

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 7 (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 aufzurufen versucht (vgl. Abschnitt 7.4).

Aufgabe 2

In der Klasse `Figur` haben `xpos` und `ypos` den voreingestellten Zugriffsschutz **private**. Damit haben die Methoden der `Kreis`-Klasse keinen direkten Zugriff (auch die Konstruktoren nicht). Soll dieser Zugriff für eine abgeleitete Klasse (aber *nicht* für beliebige Klassen) möglich sein, dann müssen `xpos` und `ypos` in der `Figur`-Definition die Schutzstufe **protected** 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 einer Basisklassenmethode durch eine Unterklassenmethode mit dem gleichen Namen und einer identischen 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 dann die *Basisklassenvariante* zum Einsatz (frühe Bindung).
- Bei überschreibenden Methoden wird jedoch die *Unterklassenvariante* benutzt (späte bzw. dynamische Bindung).

Bei klassenbezogenen Methoden kommt das späte Binden bzw. Überschreiben *nicht* in Frage.

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\Bruch\b9 ToString-Überschreibung

Kapitel 8 (Typgenerisches Programmieren)

Aufgabe 1

1. **Richtig**
2. **Falsch** (siehe Abschnitt 8.5)
3. **Falsch**
   Wenn die Aussage korrekt wäre, hätte C# ein gravierendes Fehlerrisiko. Leider liegt genau diese Situation (die sogenannte *Kovarianz* des Elementdatentyps) bei Arrays vor (siehe Abschnitt 8.2.3).
4. **Richtig**

Aufgabe 2

Ein typisches Ergebnis (gemessen auf einem Rechner unter Windows 10 (64 Bit) mit Intel-CPU Core i3 550):

```
Zeit in Millisek. für List<int>: 17
Zeit in Millisek. für ArrayList: 175
```

Das zugehörige Programm finden Sie in:

...\BspUeb\Typgenerisches Programmieren\Listenwerte

Aufgabe 3

Lösungsvorschlag:

Quellcode	Ausgabe

`Console.WriteLine("int-max:\t" + Max(4, 12, 13, 56));`
`Console.WriteLine("double-max:\t" + Max(2.16, 47.11, 34.2, 79.71));`
`Console.WriteLine("String-max:\t" + Max("abc", "def", "zeta"));`
`//Console.WriteLine("null-Parameter:\t" + Max<int>(null));`
 	
`static T Max<T>(params T[] ta) where T : IComparable<T> {`
`  if (ta == null || ta.Length == 0)`
`    throw new ArgumentException("Nicht-leerer Array erforderlich");`
`  T max = ta[0];`
`  foreach (T t in ta)`
`    if (t.CompareTo(max) > 0)`
`      max = t;`
`  return max;`
`}`
	`int-max:  56`
	`double-max:  79,71`
	`String-max:  zeta`

Kapitel 9 (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 traditionelles 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 Instanz-Konstruktoren und -Felder verboten. Seit C#8 sind in Schnittstellen auch konkrete (implementierte) Methoden erlaubt, die von einem implementierenden Typ verwendet oder durch eine eigene Implementation überschrieben werden können. Zur Verwendung einer konkreten Schnittstellenmethode ohne eigene Implementation ist jedoch eine Referenz vom Schnittstellentyp erforderlich. Der implementierende Typ erbt die konkreten Schnittstellenmethoden also nicht.
- **Abstammungsverhältnisse (Typkompatibilitäten)**
  Eine Klasse kann nur *eine* abstrakte Basisklasse besitzen, aber beliebig viele Interfaces implementieren.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Interfaces\Iteratoren und Enumeratoren

Kapitel 10 (Delegaten und Ereignisse)

Aufgabe 1

1. **Stimmt**
2. **Falsch**
3. **Stimmt**
4. **Stimmt**

Aufgabe 2

Lösungsvorschlag:

```
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);
foreach (string s in k4)
    Console.WriteLine(s);
```

Aufgabe 3

Im folgenden Lösungsvorschlag ist die Metamethode statisch realisiert:

```
var list = new List<string> { "Doro", "Liselotte", "Friedrich", "Theo", "Walter" };
Console.WriteLine("\nVerwendung der Metamethode ListLE():");
for (int i = 4; i <= 10; i += 2) {
    Console.WriteLine($"\nListe der Kurznamen mit max. {i} Zeichen:");
    var k = list.FindAll(ListLE(i));
    foreach (string s in k)
        Console.WriteLine(s);
}
static Predicate<string> ListLE(int k) {
   return (s => s.Length <= k);
}
```

Aufgabe 4

1. **Falsch**
   Ein Ereignis ist ein (statisches) Feld mit einem Delegatentyp, weist aber Besonderheiten im Vergleich zu einer gewöhnlichen Delegatenvariablen auf.
2. **Falsch**
   Wie vorzugehen ist, wird im Abschnitt 10.2.3 erläutert.
3. **Falsch**
   Am Ende von Abschnitt 10.2.2 wird eine zu vermeidende Speicherplatzverschwendung durch die unterlassene Aufhebung einer Registrierung beschrieben.
4. **Richtig**

Kapitel 11 (Kollektionen)

Aufgabe 1

1. **Falsch**
   Es wird die Einfügereihenfolge konserviert, aber keine Sortierung vorgenommen, z. B.:

Quellcode	Ausgabe

`var liste = new List<String> {"Otto", "Thea", "Andrea"};`
`foreach (var s in liste)`
`  Console.WriteLine(s);`
	`Otto`
	`Thea`
	`Andrea`

2. **Richtig**
3. **Richtig**
4. **Richtig**
5. **Richtig**

Aufgabe 2

Sie finden einen Lösungsvorschlag in:

...\BspUeb\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:

```
using System;
class Person : IComparable<Person> {
    public string Vorname { get; set; }
    public string Name { get; set; }

    public Person(string vorname, string nachname) {
        Vorname = vorname;
        Name = nachname;
    }

    public int CompareTo(Person p) {
        return (this.Name + this.Vorname).CompareTo(p.Name + p.Vorname);
    }
}
```

Infolgedessen können die Elemente der **List**<`Person`> - Kollektion sortiert werden.

Kapitel 12 (GUI-Programmierung mit WPF)

Aufgabe 1

1. **Falsch**

   Während WinUI 2 als Bestandteil der Universal Windows Platform (UWP) von Microsoft nicht mehr weiterentwickelt wird, ist WinUI 3 die zentrale Komponente im Windows App SDK, das aktuell (im Herbst 2023) von Microsoft aktiv unterstützt wird. Zwar ist WinUI 3 eine ernst zu nehmende Konkurrenz für die Windows Presentation Foundation (WPF), doch ist die fortlaufende WPF-Unterstützung gewährleistet. WinUI 3 wird unter Windows auch für MAUI-Anwendungen verwendet.
2. **Richtig**
3. **Falsch**
   Das GUI wird durch die XAML erleichtert, doch kann man die Bedienoberfläche auch komplett in C# erstellen.
4. **Richtig**
5. **Richtig**

Aufgabe 2

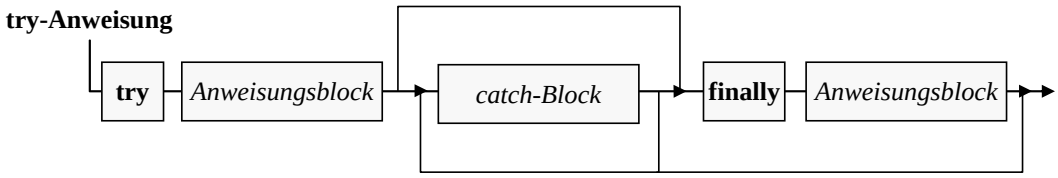
Sie finden einen Lösungsvorschlag im Verzeichnis:

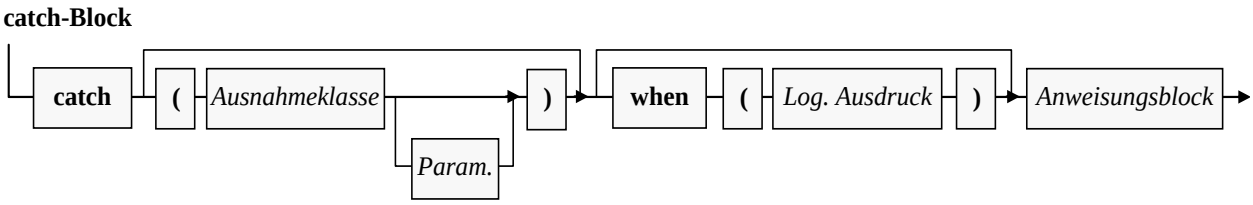
...\BspUeb\WPF\Eurokonverter

Kapitel 13 (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\SimpleStackEx

Kapitel 14 (Attribute und Reflexion)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Attribute\NonsenseAttribute

Kapitel 15 (C# für Fortgeschrittene)

Aufgabe 1

Um die Warnung zu verhindern, sollte die Rückgabe der Methode `Reverse()` das Attribut **NotNullIfNotNull** erhalten:

```
[return: NotNullIfNotNull(nameof(original))]
static string? Reverse(string? original) {
    if (original == null)
        return null;
    var charArray = original.ToCharArray();
    Array.Reverse(charArray);
    return new string(charArray);
}
```

Aufgabe 2

Die Umcodierung per Musterabgleich führt zu einem gut lesbaren Quellcode:

```
static int? AgeToCat(int? age) => age switch {
    < 15           => 1,
    >= 15 and < 30 => 2,
    >= 30 and < 45 => 3,
    >= 45 and < 60 => 4,
    >= 60          => 5,
    _              => null
};
```

Aufgabe 3

Manchmal haben es die Optionen zum Musterabgleich schwer, ihre Überlegenheit gegenüber herkömmlichen Lösungen zu beweisen. Mit einem logischen Ausdruck ist die Aufgabe einfach und schnell zu lösen:

```
public bool Zugelassen() => Passagiere >= 3 || Elektromobil;
```

Literatur

Agafonov, E. & Koryavchenko, A. (2015). *Mastering C# Concurrency*. Birmingham: Packt Publishing.

Albahari, J. (2014). *Threading in C#*. Online-Dokument: http://www.albahari.com/threading/.

Albahari, J. & Johannsen, E. (2020). *C# 8.0 in a Nutshell*. Beijing: O'Reilly.

Albahari, J. (2022). *C# 10.0 in a Nutshell*. Beijing: O'Reilly.

Ashcraft, A. (2023). *Learn WinUI 3: Leverage WinUI and the Windows App SDK to create modern windows applications with C# and XAML*. Birmingham: Packt Publishing.

Baltes, S. & Diehl, S. (2014). *Sketches and Diagrams in Practice*. Paper presented at the International Symposium on the Foundations of Software Engineering (November 2014, Hong Kong).

Baltes-Götz, B. (2009). *Einführung in das Programmieren mit C# 3.0*. Online-Dokument: https://www.uni-trier.de/index.php?id=30454

Baltes-Götz, B. (2021). *Einführung in das Programmieren mit C# 9.0*. Online-Dokument: https://bebagoe.de/csharp/9/

Baltes-Götz, B. & Götz, J. (2023). *Einführung in das Programmieren mit Java 17*. Online-Dokument: https://bebagoe.de/java/

Balzert, H. (2011). *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. Heidelberg: Spektrum.

Bloch, J. (2018). *Effective Java* (3rd. ed.). Upper Saddle River, NJ: Addison-Wesley.

Booch, G. et al. (2007). *Object-Oriented Analysis and Design with Applications* (3rd ed.). Boston, MA: Addison-Wesley.

Brind, M. (2021). *Learn Entity Framework Core*. Online-Dokument: https://www.learnentityframeworkcore.com/

Cleary, S. (2014). *Concurrency in C#. Cookbook*. Sebastopol, CA: O'Reilly.

Conrod, P. & Tylee, L. (2019). *Visual C# and Databases - 2019 Edition*. Kidware Software

Dyer, R. & Chauhan, J. (2022). An Exploratory Study on the Predominant Programming Paradigms in Python Code. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESE/FSE), pp 684-695. Online-Dokument: https://arxiv.org/pdf/2209.01817.pdf

Ebner, M. (2000). *Delphi 5 Datenbankprogrammierung*. München: Addison-Wesley.

ECMA (2022). *C# Language Specification* (6th ed.). Online-Dokument: https://ecma-international.org/publications-and-standards/standards/ecma-334/

Fruja, N.G. & Börger, E. (2005). *Analysis of the .NET CLR Exception Handling Mechanism*. Online-Dokument: https://www.research-collection.ethz.ch/handle/20.500.11850/69594

Gieselmann, H. (2023). Die 80-Prozent-Maschinen. *c't Magazin für Computertechnik*. 2023, Heft 21, 16-19.

Goll, J. & Heinisch, C. (2016). *Java als erste Programmiersprache* (8. Aufl.). Wiesbaden: Springer Vieweg

Griffiths, I. (2013). *Programming C# 5.0*. Beijing: O'Reilly.

Griffiths, I. (2020). *Programming C# 8.0*. Sebastopol, CA: O'Reilly.

Griffiths, I. (2023). *C# 11.0 new features: list pattern matching*. Online-Dokument: https://endjin.com/blog/2023/03/dotnet-csharp-11-pattern-matching-lists

Gunnerson, E. (2002). *C#* (2. Aufl.). Bonn: Galileo.

Hamilton, B. & MacDonald, M (2003). *ADO.NET in a Nutshell*. Beijing: O'Reilly.

Horstmann, C.S. & Cornell, G. (2002). *Core Java. Volume II – Advanved Features*. Palo Alto, CA: Sun Microsystems Press.

Huber, T.C. (2017). *Windows Presentation Foundation* (5. Aufl.). Bonn: Rheinwerk Computing.

Kreft, K & Langer, A. (2014). *Java 8. Default-Methoden und statische Methoden in Interfaces*. Online-Dokument: http://www.angelikalanger.com/Articles/EffectiveJava/72.Java8.DefaultMethods/72.Java8.DefaultMethods.html

Lau, O. (2009). Faites vos jeux! Zufallszahlen erzeugen, erkennen und anwenden. *c't Magazin für Computertechnik*. 2009, Heft 2, 172-178.

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.

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.pdf

Louis, D. & Strasser, S. (2002). *C# in 21 Tagen*. München: Markt + Technik.

MacDonald, M. (2012). *Pro WPF 4.5 in C#* (4rd ed.). New York, NY: Apress.

Marshall, D. & Bruno, J. (2009). *Solid Code*. Redmond: Microsoft Press.

Misner, S. (2007). *Microsoft SQL Server 2005 Express Edition. Start Now!* Redmond, WA: Microsoft Press.

Microsoft (2017). *C# Reference*. Online-Dokument https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/index

Morrison, V. (2005a). *Concurrency: What Every Dev Must Know About Multithreaded Apps*. MSDN Magazine. August 2005. Online-Dokument: https://learn.microsoft.com/en-us/archive/msdn-magazine/2005/august/concurrency-what-every-dev-must-know-about-multithreaded-apps

Morrison, V. (2005b). *Memory Models: Understand the Impact of Low-Lock Techniques in Multithreaded Apps*. MSDN Magazine. October 2005. Online-Dokument: https://learn.microsoft.com/en-us/archive/msdn-magazine/2005/october/understanding-low-lock-techniques-in-multithreaded-apps

Mössenböck, H. (2019). *Kompaktkurs C# 7.0*. Heidelberg: dpunkt.

Nathan, A. (2010). *WPF 4. Unleashed*. Indianapolis, IN: SAMS.

Petre, M. (2013). UML in practice. In: *35th International Conference on Software Engineering (ICSE 2013)*, 18-26 May 2013, San Francisco, CA, USA (forthcoming), pp. 722–731.

Peterson, J. (2014). *Common Pitfalls with IDisposable and the Using Statement*. Online-Dokument: https://superdevelopment.com/2014/03/13/common-pitfalls-with-idisposable-and-the-using-statement/

Petzold, C. (2008). *Foundations: Dependency Properties and Notifications*. MSDN-Magazine. September 2008. Online-Dokument: https://learn.microsoft.com/en-us/archive/msdn-magazine/2008/september/foundations-dependency-properties-and-notifications

Richter, J. (2006). *Microsoft .NET Framework Programmierung in C#* (2. Aufl.). Unterschleißheim: Microsoft Press.

Richter, J. (2012). *CLR via C#* (4. Aufl.). 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.

Schulz, H. (2021). Aufgedreht. Windows 11: Neuer Store für Anwender und Entwickler. *c't Magazin für Computertechnik*. 2021, Heft 16, 24-26.

Simons, R. (2004). *Hardcore Java*. Sebastopol, CA: O'Reilly.

Spurgeon, C. E. (2000). *Ethernet. The Definitive Guide*. Sebastopol, CA: O'Reilly.

Torgersen. M. (2017). *Introducing Nullable Reference Types in C#*. Online-Dokument: https://devblogs.microsoft.com/dotnet/nullable-reference-types-in-csharp/

Wegener, J. (2013). *WPF 4.5 und XAML*. München: Hanser-Verlag.

Stichwortregister

U

V

W

X

Z