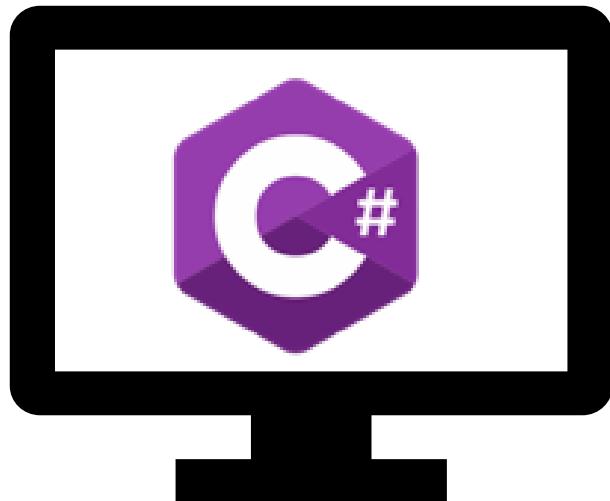




Abteilung für Informatik
htl donaustadt



C# - The Next Generation

von [Thomas Schlägl](#)

Höhere Abteilung für Informatik

HTL-Donaustadt

<mailto:slog@htl-donaustadt.at>

Version: 1.0 im September 2021

Inhaltsverzeichnis

1 EINFÜHRUNG.....	8
1.1 VORWORT DES AUTORS	8
1.2 EINIGE EIGENSCHAFTEN VON C#.....	8
1.2.1 Namensgebung	9
1.2.2 Entwicklungsgeschichte	9
1.2.3 Entwicklungsumgebungen (IDEs).....	9
2 DATENTYPEN	10
2.1 ÜBERSICHT ALLER DATENTYPEN.....	10
2.2 EINFACHE DATENTYPEN.....	11
2.2.1 Ganzzahltypen	11
2.2.2 Fließkommatypen	11
2.2.3 Zeichen- und Boolescher Typ.....	11
2.2.4 Casten von numerischen Datentypen in C#.....	12
2.3 .NET-KLASSEN DER EINFACHEN DATENTYPEN.....	12
2.4 .NET FRAMEWORK UND .NET CORE.....	13
2.4.1 C# Version vs. .NET Version vs. Visual Studio Version	13
2.4.2 Umstieg von .NET Framework auf .NET Core	14
2.5 VOM QUELLCODE ZUM AUSFÜHRBAREN EXE-PROGRAMM	15
3 METHODEN	17
3.1 DRY – DAS "DON'T REPEAT YOURSELF"-PRINZIP.....	17
3.2 METHODEN ZUR CODESTRUKTURIERUNG – EIN BEISPIEL IN C#.....	17
3.2.1 Spaghetti-Code.....	18
3.2.2 Doppelten Code in eine Methode extrahieren	19
3.2.3 Erweiterung der Methode: Input-Parameter Schriftfarbe	20
3.2.4 Noch ein Inputparameter: Es soll piepsen!	21
3.3 WIE METHODEN IN C# WERTE ZURÜCKLIEFERN KÖNNEN	22
3.3.1 Einen einzigen Wert - return	22
3.3.2 Mehrere Werte.....	23
3.3.2.1 Variante 1: Output-Parameter befüllen.....	23
3.3.2.2 Variante 2: Struct als Rückgabewert.....	24
3.3.2.3 Variante 3: Value Tuple als Rückgabewert	25
3.4 WIE MAN EINE METHODE SCHREIBT	25
3.5 SIGNATUR EINER METHODE	26
3.5.1 Beispiele zur Signatur in C#	26
3.6 VALUE TUPLES IN C#	28
3.6.1 Verwendung von Value Tuples.....	29
3.6.2 Value Tuple versus Output Parameter	30
3.7 ÜBUNGEN ZU METHODENSIGNATUREN	31
3.8 VARARG-PARAMETER – SCHLÜSSELWORT PARAMS.....	32
3.8.1 Codebeispiel für Vararg-Parameter	32
4 FORMATIERTE AUSGABEN IN C#.....	34
4.1 INTERPOLATED STRINGS	35
4.2 INDEXBASIERTE FORMATIERUNG	35
4.3 FORMATIERUNG MIT ToString()	37
5 DAS SUMMENZEICHEN UND C#	38
6 AUSGEWÄHLTE METHODEN FÜR COLLECTIONS.....	40
6.1 SUMMIEREN UND ZÄHLEN (AGGREGATSMETHODEN)	40
6.2 EXISTENZMETHODE	41
6.3 VALIDIERUNGSMETHODE	42
6.4 FILTERMETHODE	42

7 STRINGS	44
7.1 UNICODE	44
7.2 NÜTZLICHE METHODEN ZUR STRINGMODIFIKATION	45
7.2.1 Substring und [].....	45
7.2.1.1 Durchlaufen eines Strings mit Substring.....	46
7.2.1.2 Durchlaufen eines Strings mit [].....	46
7.2.1.3 Durchlaufen eines Strings mit foreach.....	46
7.2.2 Suchen in Strings	47
7.2.2.1 Contains.....	47
7.2.2.2 StartsWith und EndsWith	47
7.2.2.3 IndexOf und LastIndexOf	47
7.2.3 Stringoperationen	48
7.2.3.1 Insert.....	49
7.2.3.2 Remove.....	49
7.2.3.3 Replace	50
7.2.3.4 PadLeft und PadRight.....	50
7.2.3.5 Strings bestimmter Länge aus einem Zeichen erzeugen.....	50
7.3 STRINGS SPLITTEN	51
8 ZUFALLSZAHLEN IN C#	52
9 ZWEIDIMENSIONALE ARRAYS IN C#.....	54
9.1 "RECHTECKIGE" ZWEIDIMENSIONALE ARRAYS IN C#	54
9.1.1 Deklaration	54
9.1.2 Zugriff auf Elemente.....	54
9.2 "JAGGED" ARRAYS.....	57
10 FILE-IO	58
10.1 TEXTFILES	58
10.1.1 Textfile einlesen.....	58
10.1.1.1 Wichtige Methoden und Properties	58
10.1.1.2 Beispiel: Einlesen eines CSV-Files.....	59
10.1.2 Textfiles schreiben.....	61
10.1.3 Rezepte für CSV- und Spaltenorientierte Files	61
10.2 BINÄRFILES	63
11 COMMANDLINE PARAMETER IN C#	64
11.1 ÜBERGABE DER PARAMETER AUF DER WINDOWSKONSOLE	64
11.2 FESTLEGUNG IN VISUAL STUDIO.....	65
12 OBJEKTOIENTIERUNG	66
12.1 EIN EINFÜHRENDES BEISPIEL	66
12.2 EINE KLASSE ALS BAUPLAN FÜR OBJEKTE	67
12.3 BESTANDTEILE EINER KLASSE.....	68
12.4 KONSTRUKTOR	70
12.4.1 Constructor Chaining.....	71
12.5 ZERSTÖRUNG VON INSTANZEN ("DESTRUKTOR").....	71
12.6 GRUNDPRINZIPIEN DER OBJEKTOIENTIERUNG	72
12.7 EINE GESCHICHTE VON ROMAN OBJEKTOIENTIERT "ERZÄHLT"	72
12.7.1 Die Geschichte – Daten und Methoden in Klassen	73
12.7.2 Die Klasse Person für Roman und seine Mutter	75
12.7.3 Die Programmierung der Geschichte im Main	76
12.8 ENCAPSULATION (KAPSELUNG)	78
12.9 PROPERTIES	82
12.9.1 Auto-implemented Property mit public get und public set	83
12.9.2 Auto-implemented Property mit public get und private set.....	84
12.9.3 Auto-implemented Property mit public get und keinem set	85
12.9.4 Berechnetes Property mit public get.....	86

12.9.5	<i>Full-implemented Property mit programmiertem get und set</i>	87
12.9.6	<i>Die "Methoden" get und set</i>	88
13	VERERBUNG (INHERITANCE).....	89
13.1	EIN ERSTES EINFACHES BEISPIEL	89
13.2	EIN ZWEITES CODEBEISPIEL	90
13.3	EIN CODEBEISPIEL ZUR VERERBUNG VON METHODEN	91
13.4	ACCESS MODIFIER	92
13.5	KLASSENDESIGN EINES COMPUTERSHOPS	94
13.5.1	<i>Vererbung der Eigenschaften</i>	94
13.5.2	<i>Klassenhierarchie</i>	96
13.5.3	<i>Vererbung von Methoden</i>	97
13.6	POLYMORPHISMUS.....	97
13.7	EINE VIRTUELLE METHODE IM BEISPIEL.....	98
13.8	EINE ABSTRAKTE METHODE IM BEISPIEL	99
13.9	MERKREGEL FÜR VIRTUAL UND ABSTRACT	100
13.9.1	<i>Virtuelle Methoden</i>	100
13.9.2	<i>Abstrakte Methoden</i>	101
13.10	BEISPIEL ZUR VERERBUNG: PIZZAS, KEBAPS UND GETRÄNK.....	102
13.11	BEISPIEL POLYMORPHIE: GEMMA IN DEN SUPERMARKT	106
13.11.1	<i>Supermarktklasse mit einer Liste aller Produkte</i>	108
13.11.2	<i>Befüllung einer Liste mit Instanzen</i>	109
13.11.2.1	<i>Verbesserte Klasse Supermarkt</i>	111
14	COMPILER TYPES UND RUNTIME TYPES.....	113
14.1	BEGRIFFSERKLÄRUNGEN.....	113
14.2	UPCASTS UND DOWNCASTS.....	114
14.2.1	<i>Upcast</i>	115
14.2.2	<i>Downcast.....</i>	115
14.3	VIRTUELLE METHODEN.....	116
14.4	ABSTRAKTE METHODEN	116
14.5	METHODENSUCHE ZUR LAUFZEIT	117
15	NONSTATIC VERSUS STATIC METHODEN	119
15.1	CODEBEISPIEL: KLASSEN VARIABLE ALS INSTANZZÄHLER	120
16	INTERFACES	122
16.1	CODEBEISPIEL 1: INTERFACE ZUR FLÄCHENBERECHNUNG	122
16.2	CODEBEISPIEL 2: INTERFACE FÜR TIERE	124
16.3	INTERFACES FÜR DAS SORTIEREN.....	125
16.3.1	<i>Interface IComparable</i>	125
16.3.1.1	<i>Beispiel: Sortierung von Autos</i>	125
16.3.2	<i>Interface IComparer</i>	126
16.3.2.1	<i>Beispiel: Sortierung von Autos</i>	127
17	EXTENSION METHODS	128
18	INDEXER	130
19	EXPRESSION-BODIED MEMBERS.....	132
20	ENUMS	134
20.1	CODEBEISPIEL FÜR PRIORITÄTEN	135
21	DATUMS- UND ZEITBERECHNUNGEN.....	136
21.1	DATUMSANGABEN UND ZEITPUNKTE: DATETIME	136
21.2	ZEITSPANNEN TIMESPAN	138
22	IMPLICIT OPERATORS – CAST OPERATOREN	139

23	VARIABLEN	141
23.1	LOKALE VARIABLEN	141
23.2	INSTANZVARIABLEN	141
23.3	KLASSENVARIABLEN	141
23.4	INSTANZIERUNG VON OBJEKten UND GARBAGE COLLECTION	141
23.5	VALUE TYPES UND REFERENCE TYPES	142
23.5.1	<i>Merkregel</i>	143
23.5.2	<i>Value Types am Stack</i>	144
23.5.2.1	Value Types bei Methodenaufrufen	145
23.5.3	<i>Reference Types am Heap</i>	145
23.5.3.1	Reference Types bei Methodenaufrufen	146
23.5.3.2	Schlüsselwort <code>ref</code>	147
24	NAMING CONVENTIONS	149
25	EINIGE WEITERE SCHLÜSSELWÖRTER IN C#	150
25.1	VAR	150
25.2	CONST	150
25.3	USING-STATEMENT	151
26	EXCEPTIONS	152
26.1	RÜCKGABEWERTE	152
26.2	CODEBEISPIEL 1: INDEX OUT OF RANGE	152
26.3	CODEBEISPIEL 2: GANZZAHLDIVISION DURCH 0	153
26.4	EXCEPTIONS SELBST AUSLÖSEN	155
26.5	EXCEPTIONKLASSEN SELBST DEFINIEREN	155
27	NAMESPACES	158
27.1	DEFINITION VON NAMESPACES	158
27.2	VERWENDUNG VON KLASSEN AUS NAMESPACES	158
28	GENERICs	160
29	COLLECTIONS	161
29.1	ÜBERSICHT	161
29.2	LIST	162
29.2.1	<i>Wichtige Properties und Methoden</i>	162
29.2.2	<i>Nachteile von List</i>	162
29.2.3	<i>Code Sample</i>	163
29.3	LINKEDLIST UND LINKEDLISTNODE	164
29.3.1	<i>Löschen eines Elements in LinkedList</i>	164
29.3.2	<i>Löschen des ersten Elements in LinkedList</i>	164
29.3.3	<i>Löschen des letzten Elements in LinkedList</i>	165
29.3.4	<i>Einfügen eines Elements innerhalb der LinkedList</i>	165
29.3.5	<i>Einfügen am Beginn einer LinkedList</i>	165
29.3.6	<i>Einfügen am Ende einer LinkedList</i>	165
29.3.7	<i>Wichtige Properties und Methoden</i>	166
29.3.8	<i>Code Sample</i>	166
29.4	ARRAYLIST	167
29.4.1	<i>Wichtige Properties und Methoden</i>	167
29.5	HASHSET	167
29.5.1	<i>Methode GetHashCode()</i>	167
29.5.2	<i>Beispiel: Lottotipps 6 aus 45</i>	168
29.5.3	<i>Wichtige Properties und Methoden</i>	168
29.5.4	<i>Code Sample</i>	169
29.6	SORTEDSET	170
29.6.1	<i>Wichtige Properties und Methoden</i>	170
29.6.2	<i>Beispiel: Lottotipps 6 aus 45</i>	170

29.6.3	<i>Code Sample</i>	172
29.7	DICTIONARY	173
29.7.1	<i>Beispiel: Fußballspieler und Alter</i>	174
29.7.2	<i>Wichtige Properties und Methoden</i>	175
29.7.3	<i>Code Sample</i>	175
29.7.4	<i>SortedDictionary</i>	177
29.8	STACK	178
29.8.1	<i>Wichtige Properties und Methoden</i>	178
29.8.2	<i>Code Sample</i>	179
29.9	QUEUE	180
29.9.1	<i>Wichtige Properties und Methoden</i>	180
29.9.2	<i>Code Sample</i>	181
30	EVENTS UND DELEGATES	182
30.1	OBSERVER PATTERN	182
30.2	DELEGATES	182
30.2.1	<i>Verkettung von Delegates (Multicast Delegates)</i>	183
30.2.2	<i>Func<> und Action<>: Delegates im .NET-Framework</i>	185
30.2.2.1	<i>Beispiel für Func</i>	185
30.2.2.2	<i>Beispiel für Action</i>	186
30.2.3	<i>Übungen zu Delegates</i>	187
30.3	EVENTS IN C#	188
30.4	BEISPIEL ZUR EVENTPROGRAMMIERUNG: AKTIENKURS	188
30.4.1	<i>Kochrezept für das Anlegen eines Events</i>	189
30.4.2	<i>Code Sample in C#</i>	190
30.5	ÜBUNGEN ZU EVENTS	191
31	ASYNCHRONE PROGRAMMIERUNG	192
31.1	EINLEITUNG	192
31.2	KLASSE TASK	192
31.3	ASYNC UND AWAIT	193
31.3.1	<i>Beispiel: Empfang von Daten einer UDP-Kommunikation</i>	193
31.3.2	<i>Änderung eines synchronen in einen asynchronen Aufruf</i>	193
32	LAMBDAS	195
32.1	FAKten ÜBER LAMBDAS	195
33	PROGRAMMBIBLIOTHEKEN	197
33.1	LIBRARY FILES	197
33.2	DYNAMISCHE UND STATISCHE BIBLIOTHEKEN	197
33.3	C# - ERSTELLEN EINER DYNAMIC LINK LIBRARY (DLL)	198
33.4	C# - VERWENDEN EINER DYNAMIC LINK LIBRARY (DLL)	199
34	SPRACHAUSGABE IN C#	201
34.1	WELCHE SPRACHEN SIND UNTER WINDOWS INSTALLIERT?	201
34.2	AUSGABE EINES TEXTES IN EINER BESTIMMTEN SPRACHE	202
34.3	NACHINSTALLIEREN EINER SPRACHE UNTER WINDOWS	203
35	ANHANG	204
35.1	ABBILDUNGSVERZEICHNIS	204
35.2	VIDEOVERZEICHNIS	205
35.3	IDEENLISTE FÜR ERGÄNZUNGEN IN DIESEM SKRIPTUM	205

History of Change

<i>Date</i>	<i>Name</i>	<i>Changes</i>
2021-09-05	V1.0	Initialversion: die allermeisten Grundlagen

1 Einführung

Das vorlegende Skriptum hat das Ziel all jene Jugendlichen, SchülerInnen und sonstige Interessierte die in C# die allerersten Schritte gemacht haben, in die erweiterten Sprachelemente und die Objektorientierung einzuführen.

Voraussetzung für die LeserInnen ist die Kenntnis folgender Themen in C#:

- Variablen: Deklaration, Initialisierung, Zuweisung
- Datentypen: `int`, `double`, `string`, `bool`
- Einlesen von numerischen Werten und Texten mit Hilfe von `int.Parse`, `double.Parse` und `Console.ReadLine()`
- Ausgaben mit `Console.WriteLine`
- `if`-Abfragen
- `switch`-Abfragen
- Schleifen: `for`, `while`, `do-while`
- Numerische Operatoren `+`, `*`, `-`, `/`, `%`
- Logische Operatoren `&&` und `||`
- Anlegen und Ausführen von Konsolenanwendungen in Visual Studio

1.1 Vorwort des Autors

Ich verfolge mit diesem Skriptum das Ziel an das hervorragende Skriptum *POS1 Programmieren und Softwareentwicklung 1 - Skriptum¹* von Dr. Günther Kovaricek und Peter Stengel anzuschließen. Einige Wiederholungen sind allerdings ebenso unumgänglich, wie auch die Ergänzungen zu neuen Entwicklungen von C#, wie beispielsweise die Verwendung von Value Tuples bei Methoden.

Das vorliegende Skriptum bündelt einerseits meine eher losen Unterlagen, die sich in den ersten fünf Jahren meiner Lehrtätigkeit an der Informatik-Abteilung der HTL-Donaustadt angehäuft haben, andererseits wurden zahlreiche Stellen wesentlich ergänzt. Außerdem verweise ich an einigen Stellen auf die Tutorial-Videos meines YouTube-Channels [Coding Kurzgeschichten](#) (früher "Kurzgeschichten in C#").



1.2 Einige Eigenschaften von C#

C# ist eine volltypisierte Sprache, also eine sogenannte Compilersprache. Sourcecode wird in .cs-Files gespeichert, die normale Textfiles sind. Diese werden kompiliert und zu einer Anwendung gebaut. Syntaxfehler werden bereits während dem Kompilieren entdeckt und nicht erst zur Laufzeit, wie bei Skriptsprachen, wie Python oder JavaScript. Die Sprache ist also type-safe.

C# ist selbstverständlich auch eine moderne objektorientierte Sprache. Mit .NET (Framework und Core) unterstützt C# die Entwicklung von Konsolenanwendungen, graphische Anwendungen (Windows Forms und WPF) und serverseitige Komponenten.

C# kennt relativ viele Schlüsselwörter, weil die Sprache im Gegensatz zu Java, bereits auf der Sprachebene sehr viel Unterstützung anbietet.

¹ Das Skriptum lag dem Autor in der Version vom 26. April 2020 vor.

Beispiele für Schlüsselwörter die C# bietet, die in Java und vielen anderen Sprachen nicht zur Verfügung stehen:

- Properties und `get` und `set`: Kapseln den Zugriff auf Instanzvariablen durch automatisch erzeugte Getter und Setter.
- `delegate`: Erlaubt die Definition von Datentypen für Variablen die Referenzen auf Methoden speichern können (Funktionpointer).
- `event`: Definiert ein Ereignis, das ausgelöst werden kann und für das man sich registrieren kann. Implementiert das Listener- bzw. Observer-Design Pattern.

1.2.1 Namensgebung

C# ist nach der Musiknote *cis* benannt. Die Silbe -is erhöht den Notenton um einen Halbton. Das ist eine Anspielung auf die Vorgängersprache C++ und dem ++-Operator mit dem ein numerischer Wert um 1 erhöht werden kann.

1.2.2 Entwicklungsgeschichte

C# ist ebenso wie Java eine plattformunabhängige Sprache und wurde rund um das Jahr 2000 von Microsoft entwickelt. Die Sprache orientierte sich neben Java, das wesentlich früher von Sun Microsystems entwickelt wurde, vor allem an C++ und C. Bei funktionalen Elementen der Sprache orientierte man sich an Haskell.

Der wichtigste Entwickler der Sprache ist [Anders Hejlsberg](#).



Video 1: [Youtube-Video](#) mit Anders Hejlsberg über die Ursprünge von C#

1.2.3 Entwicklungsumgebungen (IDEs)

Neben Visual Studio von Microsoft haben sich einige anderen IDEs für die Entwicklung von C#-Programmen etabliert.

- [Visual Studio 2019](#)
- [Visual Studio Code](#)
- [MonoDevelop](#)
- [Rider](#)

2 Datentypen

In C# gibt es eine ganze Menge an unterschiedlichen Datentypen. Diese ermöglichen eine Spezialisierung je nach den Daten die verarbeitet werden sollen. Beispielsweise stehen für die Speicherung einer Ganzzahl unterschiedliche Integer-Datentypen zur Verfügung.

In C#, wie auch in Java und JavaScript, teilt man alle Datentypen in zwei große Gruppen ein, nämlich **Value Types (Werttypen)** und **Reference Types (Verweistypen)**.

Die Unterscheidung von Value- und Reference Types wird in diesem Skriptum im Kapitel 23.5 Value Types und Reference Types genau erklärt. Zunächst ist es nur wichtig zu wissen, dass es in C# (und vielen volltypisierten Sprachen) diese zwei Arten von Datentypen gibt.

Für Interessierte ist ein ausführliches 14-Minuten-Erklärvideo <https://youtu.be/zif4GXnDGiw> - das allerdings die Kenntnis von Klassen und Objektorientierung voraussetzt² - empfohlen.



Video 2: Coding Kurzgeschichten-Video über Reference Types und Value Types in C#

2.1 Übersicht aller Datentypen

Datentypen in C# werden im Common Type System (CTS) zusammengefasst.

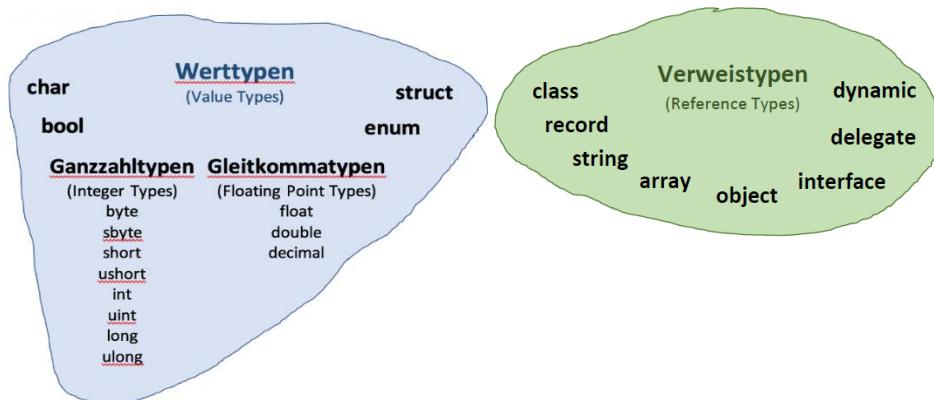


Abbildung 1: Übersicht über die Datentypen in C#

² Klassen und Objektorientierung werden in Kapitel 12 dieses Skriptums erklärt.

2.2 Einfache Datentypen

C# hat bemerkenswert viele einfache Datentypen, bedingt durch die Unterstützung von unsigned-Datentypen, wie sie auch in C und C++ unterstützt werden.

Microsoft hat bei der Implementierung der Sprache aber davon abgesehen, diese unsigned-Datentypen ähnlich systematisch wie in C++ zu verwenden. Die Länge eines Strings hat beispielsweise den Datentyp `int` und nicht `uint`, obwohl die Länge sicher nie negativ wird.

2.2.1 Ganzzahltypen

C#	Bytes	Wertebereich
byte	1	0 .. 255
sbyte		-128 .. 127
short	2	-32768 .. 32767
ushort		0 .. 65535
int	4	$\approx -2 \text{ Mrd} .. \approx 2 \text{ Mrd}$
uint		0 .. $\approx 4 \text{ Mrd}$
long	8	$\approx -9 \cdot 10^{18} .. \approx 9 \cdot 10^{18}$
ulong		0 .. $\approx 18 \cdot 10^{18}$

Abbildung 2: Wertebereich aller Ganzzahltypen

2.2.2 Fließkommatypen

C#	Bytes	Wertebereich	Genauigk.	∞
float	4	$-3,4 \cdot 10^{38} .. -1,5 \cdot 10^{-45}$ und $1,5 \cdot 10^{-45} .. \pm 3,4 \cdot 10^{38}$	7 Stellen	Ja
double	8	$-1,7 \cdot 10^{308} .. -5,0 \cdot 10^{-324}$ und $5,0 \cdot 10^{-324} .. 1,7 \cdot 10^{308}$	15 - 16 St.	Ja
decimal	16	$-7,9 \cdot 10^{28} .. -10^{-28}$ und $10^{-28} .. 7,9 \cdot 10^{28}$	28 - 29 St.	Nein

Abbildung 3: Wertebereich aller Fließkommatypen

2.2.3 Zeichen- und Boolscher Typ

C#	Speicherplatz	Wertebereich
char	2 Bytes	Ein Unicodezeichen ¹ als UTF-16 ² , z. B. 'a', '\u221e'
bool	1 Bit	false, true

Abbildung 4: Wertebereich des Character und Boolschen Datentyps

Die Konstanten zum Abrufen dieser Werte in einem Programm sind [hier](#) angegeben.

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types>

2.2.4 Casten von numerischen Datentypen in C#

Casten bedeutet Typumwandeln. Bei numerischen Datentypen werden kleinere Typen in größere Typen automatisch konvertiert (implicit cast).

Implizites Casten: byte -> short -> int -> long -> float -> double
char -> int

Explizites Casten: Wird von einem größeren Typen auf einen kleineren Typ konvertiert, ist ein Wertverlust möglich. Daher muss explizit gecastet werden.

Beispiel für einen impliziten und einen expliziten Cast:

```
char c1 = 'A';
int n = c1;           // implicit cast
n += 3;
char c2 = (char)n;   // explicit cast c2 = 'D'
```

2.3 .NET-Klassen der einfachen Datentypen

In C# haben die einfachen Datentypen intern eine .NET-Klasse im Hintergrund, die alle Methoden und Properties für den Datentyp zur Verfügung stellen, z. B. System.Int64 für int, System.Boolean für bool.

Eine Übersicht über diese .NET-Klassen (die eigentlich Structs sind!) findet sich auf <https://docs.microsoft.com/en-us/dotnet/standard/class-library-overview>

Beachte, dass in Artikeln und in Blogs im Internet über .NET fast immer von .NET-Klassen gesprochen wird, obwohl es sich eigentlich um Structs - also Value Types - handelt!

Braucht man einfache Datentypen in C#, verwendet man sehr selten die .NET-Klassen. Alle Konstanten und statische Methoden sind über int, double, ... ohnehin zugängig und Collections können diese Datentypen ebenfalls speichern.

2.4 .NET Framework und .NET Core

Microsoft entwickelte verschiedene Frameworks um .NET-Anwendungen zu erstellen und laufen zu lassen.

Von Jänner 2002 bis April 2019 wurde das [.NET-Framework](#) entwickelt und releaset, das vorwiegend auf Windows reduziert war. Der Quellcode des .NET-Frameworks war nicht freizugänglich.

Die neuere Entwicklung seit Juni 2016 ist [.NET-Core](#) bzw. wird es nur noch als .NET bezeichnet, das in Zukunft zu bevorzugen ist und die Betriebssysteme [Linux](#) und [MacOS](#) unterstützt. Diese Plattform ist Open Source und frei.

Beim Anlegen einer neuen Solution in Visual Studio, werden daher zwei Arten von Console Apps angeboten, .NET Core und .NET Framework.

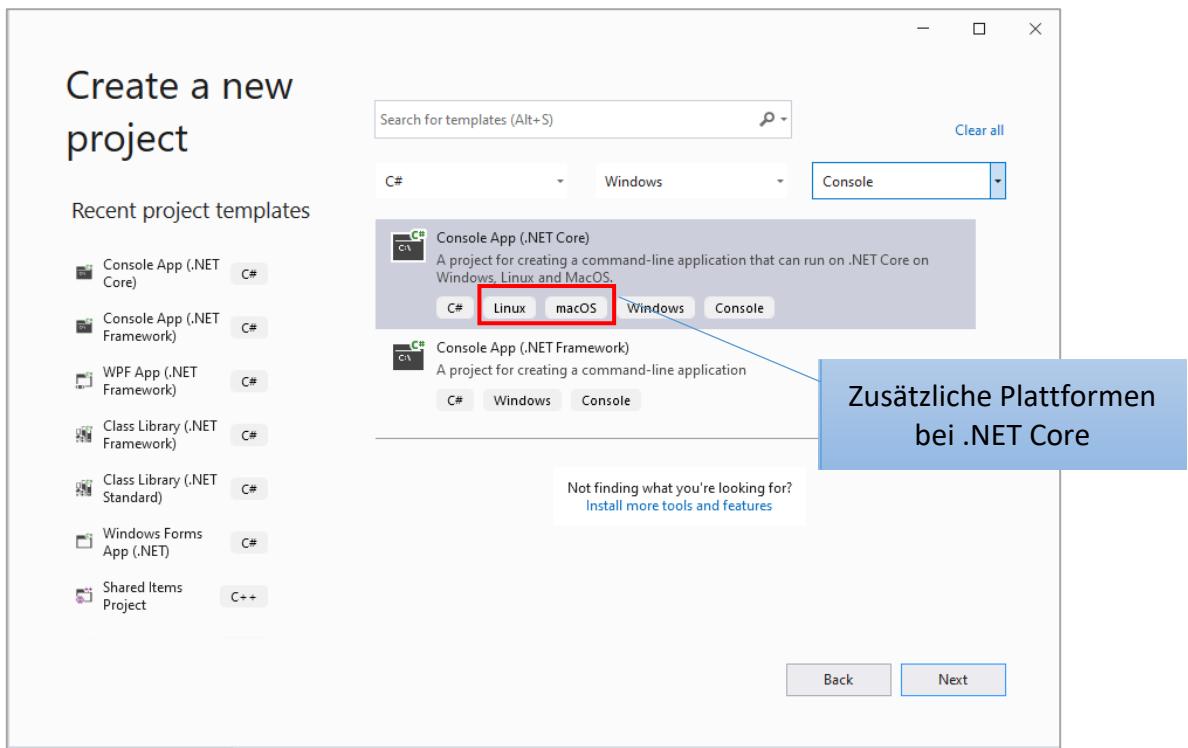


Abbildung 5: Create-Dialog in Visual 2019

2.4.1 C# Version vs. .NET Version vs. Visual Studio Version

C# wird als Sprache selbstverständlich weiterentwickelt und es werden immer mehr Features integriert. Die Version von C# hängt mit der .NET Version nur lose zusammen. Häufig sind die Releases der Entwicklungsumgebung Visual Studio an die .NET Versionen gebunden.

Einen guten Überblick über hier <https://www.tutorialsteacher.com/csharp/csharp-version-history>.

2.4.2 Umstieg von .NET Framework auf .NET Core

Für diejenigen die bisher unter .NET Framework einfache Übungsprogramme entwickelt haben, ist der Umstieg auf .NET Core sehr einfach. Es ändert sich praktisch nichts.

Beim Ausführen einer Konsolenanwendung in Visual Studio 2019 in der Debug-Konfiguration bemerkt man einen Unterschied sehr schnell. In .NET Core wird die Konsole normalerweise nicht geschlossen, was beim Entwickeln sehr angenehm ist.

Zu beachten ist, dass es bei .NET Core NICHT notwendig ist, am Ende einer Konsolenanwendung ein `Console.ReadKey();` oder `Console.ReadLine();`.

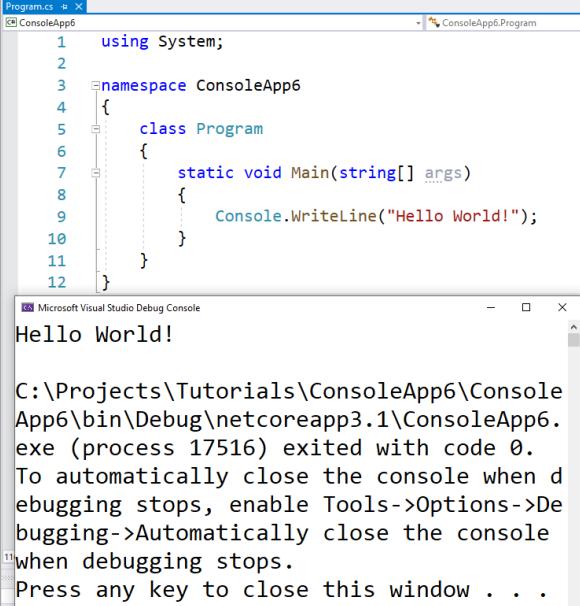
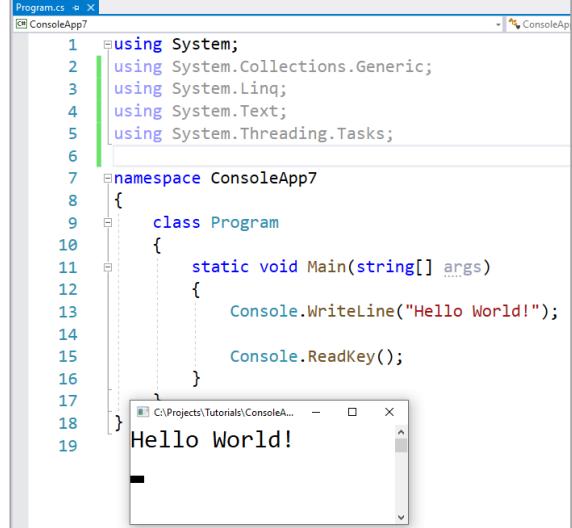
Console App .NET Core	Console App .NET Framework
 <p>Das Konsolenfenster bleibt geöffnet.</p>	 <p>Das Konsolenfenster bleibt nur, aufgrund von <code>Console.ReadKey()</code> geöffnet.</p>

Abbildung 6: Offenhalten des Konsolenfensters bei Visual Studio 2019

Das automatische Schließen des Konsolenfensters kann über Tools->Options angepasst werden.

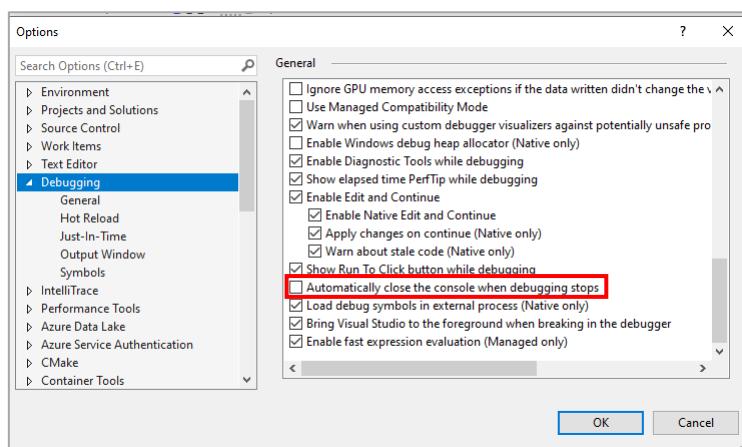


Abbildung 7: Option zum automatischen Schliessen des Debug-Fensters

2.5 Vom Quellcode zum ausführbaren Exe-Programm

Eine Entwicklungsumgebung wie Visual Studio übersetzt den von C#-Programmcode in eine Intermediate Language (IL). Dazu startet Visual Studio das eigentliche Compilerprogramm csc.

Das Compilerprogramm ist unter

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe zu finden. Visual Studio startet dieses Programm automatisch³.

Falls der Programmcode syntaktisch korrekt ist, also keine Kompilierfehler (Compile-Errors) aufweist, erzeugt der Compiler eine Datei, nämlich das sogenannte **Assembly**, das den IL-Code enthält. Das Assembly ist unter dem Betriebssystem Windows entweder eine ausführbare EXE-Datei (Wenn ein Main enthalten ist.) oder eine DLL-Datei⁴.

Eine solche EXE-Datei enthält als ersten Befehl einen Verweis auf die **CLR**⁵ (=Common Language Runtime). Die CLR ist Teil des .NET-Frameworks und führt den IL-Code der in der EXE-Datei enthalten ist aus. Die CLR verwendet dafür ihrerseits einen Compiler, der mit einem Just-In-Time-Compiler (JIT) in nativen Code (=Maschinencode) übersetzt, der dann von der CPU verarbeitet wird.

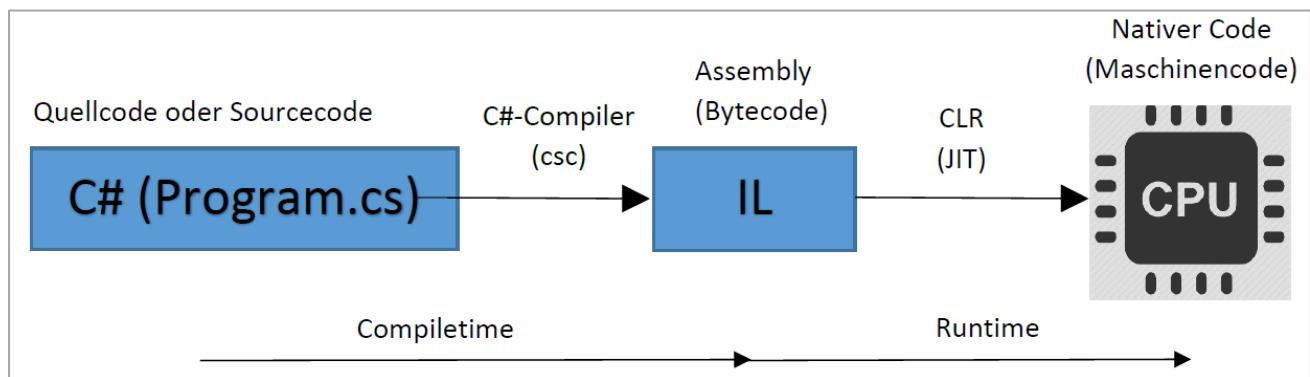


Abbildung 8: Übersicht über Completetime und Runtime bei .NET

Der IL-Code ist nicht einfach lesbar, wenn man Assembler von CPUs nicht beherrscht.



Abbildung 9: Beispiel für die IL (Intermediate Language)

³ Theoretisch könnte man ein eigenes Programm Program.cs auch ohne Visual Studio kompilieren, z. B. C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe /out:Programm.exe Program.cs

⁴ https://de.wikipedia.org/wiki/Dynamic_Link_Library

⁵ https://de.wikipedia.org/wiki/Common_Language_Runtime

Den IL-Code einer bestimmten EXE-Datei kann man sich mit dem Tool IL DASM (=Intermediate Language Disassembly Tool) anschauen, in dem man

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

in einem Windows Command Fenster startet. Für das Entwickeln in C# ist das aber nie notwendig.

Bei anderen Sprachen, wie C++, wird durch den Compiler sofort maschinenabhängiger Code erzeugt. Die EXE-Datei in diesen Sprachen enthält also keinen IL-Code sondern ausschließlich Maschinencode.

Bei C++ wird Maschinencode bereits zur Completetime erstellt, bei **plattformunabhängigen Sprachen**, wie C#, F#, Visual Basic .NET und Java, dagegen erst zur Laufzeit. F#⁶ und Visual Basic .NET⁷ sind andere Sprachen von Microsoft, die IL-Code erzeugen, der dann von der CLR ausgeführt werden kann.

⁶ <https://de.wikipedia.org/wiki/F-Sharp>

⁷ https://de.wikipedia.org/wiki/Visual_Basic_.NET

3 Methoden

Methoden dienen in erster Linie dazu, um Programmteile mehrmals benutzerbar zu machen.

Methodennamen beginnen in C# immer mit großem Anfangsbuchstaben und sollten ein Zeitwort in ihrem Namen enthalten, denn Methoden sind Aktionen und keine Gegenstände oder Objekte.

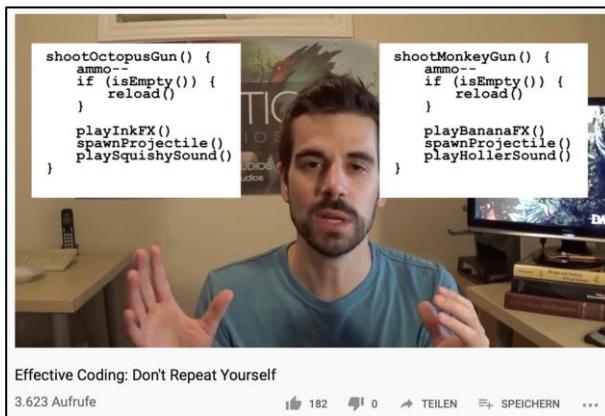
Die folgenden Abschnitte erklären alle wichtigen Teile von Methoden. Das sind:

- Wofür gibt es Methoden überhaupt?
- Was sind Input-Parameter?
- Was sind Output-Parameter?
- Was sind Value Tuples?
- Was ist eine Methodensignatur?

3.1 DRY – Das "Don't repeat yourself"-Prinzip

Das Prinzip sich beim Codieren nicht zu wiederholen - was häufig die Verwendung von Methoden impliziert - ist als "Don't repeat yourself"-Prinzip in der Softwareentwicklung bekannt. Es wird unter anderem im hervorragenden Buch "[Der pragmatische Programmier](#)" von Andrew Hunt und David Thomas beschrieben (siehe <http://www.pragmatischprogrammieren.de/pragprog/>).

Der Blögeintrag <http://tacticstudios.blogspot.com/2015/07/effective-coding-dont-repeat-yourself.html> und das darin enthaltene Video erklärt das DRY-Prinzip.



Video 3: [Youtube Video](#) über DRY in Gaming Software

3.2 Methoden zur Codestrukturierung – Ein Beispiel in C#

Dieser Abschnitt erklärt anhand eines Codestücks im Main, wie dieses durch eine Methode wesentlich übersichtlicher und durch Inputparameter flexibler gestaltet werden kann.

Methoden sind ein wesentliches Konzept um Code wartbar und übersichtlich zu gestalten. Methoden haben den wichtigen Effekt, dass Codestücke in Blöcken zusammengefasst und benannt werden.

3.2.1 Spaghetti-Code

Wir starten mit einem Stück Code im Main, der keinerlei Strukturierung hat.

Beispiel: Das untere Programm rechnet eine Körpergröße in Zentimeter auf die Einheit Fuß⁸ um. Am Beginn wird der Benutzer begrüßt und am Ende verabschiedet. Sowohl am Beginn, als auch am Ende wird eine Copyright-Meldung mit dem aktuellen Datum und der aktuellen Uhrzeit ausgegeben.

Dieses Programm enthält doppelte Codezeilen. Außerdem soll die Farbe in der Willkommensmeldung und die Verabschiedung ausgegeben werden, flexibel geändert werden können.

C# ursprünglicher Code

```
static void Main(string[] args)
{
    Console.OutputEncoding = Encoding.Unicode;

    // Willkommensmeldung
    Console.WriteLine("HALLÖCHEN!");
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine("\u00A9SLOG, Wien 2017"); // U+00A9 ist das Copyrightzeichen
    Console.WriteLine("Datum und Uhrzeit: " + DateTime.Now);
    Console.ResetColor();

    // Eingabe und Berechnung
    Console.Write("Gib deine Körpergröße in Zentimeter ein: ");
    int cm = int.Parse(Console.ReadLine());
    double feet = Math.Round(cm * 3.28084e-2, 2); // gerundet auf zwei Stellen
    Console.WriteLine("Du bist " + feet + " Fuß groß.");

    // Verabschiedung
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine("\u00A9SLOG, Wien 2017");
    Console.WriteLine("Datum und Uhrzeit: " + DateTime.Now);
    Console.ResetColor();
    Console.WriteLine("AUF WIEDERSEHEN!");
}
```

Programmausgabe auf der Konsole:

```
HALLÖCHEN!
©SLOG, Wien 2017
Datum und Uhrzeit: 16.09.2017 23:15:56
Gib deine Körpergröße in Zentimeter ein: 163
Du bist 5,35 Fuß groß.
©SLOG, Wien 2017
Datum und Uhrzeit: 16.09.2017 23:16:02
AUF WIEDERSEHEN!
```

Dieses Programm hat zwei identische Teile.

⁸ [https://de.wikipedia.org/wiki/Fuß_\(Einheit\)](https://de.wikipedia.org/wiki/Fuß_(Einheit))

3.2.2 Doppelten Code in eine Methode extrahieren

In so einem Fall ist es sinnvoll und notwendig eine Methode zu definieren, die die Programmzeilen mit den entsprechenden Zeilen enthält.

```
static void CopyrightMeldungAusgeben()
{
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine("\u00A9SLOG, Wien 2017"); // U+00A9 ist das Copyrightzeichen
    Console.WriteLine("Datum und Uhrzeit: " + DateTime.Now);
    Console.ResetColor();
}
```

An den beiden Stellen, wo bisher die Programmzeilen gestanden sind, schreibt man den Namen der Methode und () ; Eine solche Stelle nennt man *Methodenaufruf* oder *Aufruf der Methode*.

Das Main wird dadurch wesentlich kürzer, weil es keinen doppelten Programmcode mehr gibt und die Anweisungen für das Ausgeben der Copyright-Meldung sind zusammengefasst.

C# Code mit einer Methode

```
static void CopyrightMeldungAusgeben()
{
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine("\u00A9SLOG, Wien 2017"); // U+00A9 ist das Copyrightzeichen
    Console.WriteLine("Datum und Uhrzeit: " + DateTime.Now);
    Console.ResetColor();
}

static void Main(string[] args)
{
    Console.OutputEncoding = Encoding.Unicode;

    // Willkommensmeldung
    Console.WriteLine("HALLÖCHEN!");
    CopyrightMeldungAusgeben();

    // Eingabe und Berechnung
    Console.Write("Gib deine Körpergröße in Zentimeter ein: ");
    int cm = int.Parse(Console.ReadLine());
    double feet = Math.Round(cm * 3.28084e-2, 2); // gerundet auf zwei Stellen
    Console.WriteLine("Du bist " + feet + " Fuß groß.");

    // Verabschiedung
    CopyrightMeldungAusgeben();
    Console.WriteLine("AUF WIEDERSEHEN!");
}
```

Programmausgabe auf der Konsole:

HALLÖCHEN!
©SLOG, Wien 2017
Datum und Uhrzeit: 16.09.2017 23:15:56
Gib deine Körpergröße in Zentimeter ein: 163
Du bist 5,35 Fuß groß.
©SLOG, Wien 2017
Datum und Uhrzeit: 16.09.2017 23:16:02
AUF WIEDERSEHEN!

3.2.3 Erweiterung der Methode: Input-Parameter Schriftfarbe

Die obere CopyrightMeldungAusgeben -Methode macht bei jedem Aufruf exakt das gleiche. Sehr häufig möchte man Methoden aber einen oder mehrere Werte mitgeben, damit sie mehr können.

Beispielsweise wäre es doch nett, wenn wir in unserem Beispiel den Text der Copyright-Meldung in verschiedenen Farben ausgegeben könnten. Dazu muss man einen Inputparameter definieren. Parameter – es gibt auch noch andere als Inputparameter – stehen bei Methoden immer innerhalb der Klammer.

Der Datentyp für die Konsolenfarbe heißt ConsoleColor. Ein Inputparameter bekommt seinen Wert immer erst während der Programmausführung und nicht während dem Kompilieren.

Beim ersten Aufruf wird die Methode für Geld (`ConsoleColor.Yellow`) ausgeführt und für den zweiten Aufruf mit Rot (`ConsoleColor.Red`).

C# Code mit einer Methode und einem Input-Parameter

Datentyp und Name des Inputparameters

```
static void CopyrightMeldungAusgeben( ConsoleColor farbe )
{
    Console.ForegroundColor = farbe;
    Console.WriteLine("\u00A9SLOG, Wien 2017"); // U+00A9 ist das Copyrightzeichen
    Console.WriteLine("Datum und Uhrzeit: " + DateTime.Now);
    Console.ResetColor();
}

static void Main(string[] args)
{
    Console.OutputEncoding = Encoding.Unicode;

    // Willkommensmeldung
    Console.WriteLine("HALLÖCHEN!");
    CopyrightMeldungAusgeben(ConsoleColor.Yellow);

    // Eingabe und Be
    static void CopyrightMeldungAusgeben( ConsoleColor farbe = ConsoleColor.Yellow )
    {
        Console.ForegroundColor = farbe;
        Console.WriteLine("\u00A9SLOG, Wien 2017"); // U+00A9 ist das Copyrightzeichen
        Console.WriteLine("Datum und Uhrzeit: " + DateTime.Now);
        Console.ResetColor();
    }

    // Verabschiedung
    CopyrightMeldungAusgeben(ConsoleColor.Red);
    Console.WriteLine("AUF WIEDERSEHEN!");

    Console.ReadKey();
}
```

Bei der

Programmausgabe auf der Konsole:

```
HALLÖCHEN!  
©SLOG, Wien 2017  
Datum und Uhrzeit: 15.09.2017 23:05:56  
Gib deine Körpergröße in Zentimeter ein: 163  
Du bist 5,35 Fuß groß.  
©SLOG, Wien 2017  
Datum und Uhrzeit: 15.09.2017 23:06:02  
AUF WIEDERSEHEN!
```

3.2.4 Noch ein Inputparameter: Es soll piepsen!

Eine Methode kann zum Glück eine beliebige Anzahl von Inputparameter haben, nämlich gar keine, einen oder auch viel mehr. Die Inputparameter werden einfach durch Beistriche getrennt. Wir erweitern unsere Methode um die Möglichkeit einen 277-Hz-Ton⁹ für eine bestimmte Dauer über den Lautsprecher auszugeben. Das ist über eine Methode `Console.Beep(int, int)`¹⁰ möglich, der man die Frequenz und die Dauer in Millisekunden übergibt.

In unserem Programm wird im Main für die Dauer des Tons einmal 0.3 und einmal 0.5 übergeben. Genauso wie mit dem Inputparameter farbe, wird auch der Inputparameter sekunden während der Programmausführung mit dem Wert des Aufrufs initialisiert.

C# Code mit einer Methode und zwei Input-Parametern

```
static void CopyrightMeldungAusgeben(ConsoleColor farbe, double sekunden)
{
    Console.ForegroundColor = farbe;
    Console.WriteLine("\u00A9SLOG, Wien 2017"); // U+00A9 ist das Copyrightzeichen
    Console.WriteLine("Datum und Uhrzeit: " + DateTime.Now);
    Console.Beep(277 /* in Hz (277 ist die Note C#) ;- */ , (int)(sekunden * 1000));
    Console.ResetColor();
}

static void Main(string[] args)
{
    Console.OutputEncoding = Encoding.Unicode;

    // Willkommensmeldung
    Console.WriteLine("HALLÖCHEN!");
    CopyrightMeldungAusgeben(ConsoleColor.Yellow, 0.3);

    // Eingabe und Berechnung
    Console.Write("Gib deine Körpergröße in Zentimeter ein: ");
    int cm = int.Parse(Console.ReadLine());
    double feet = Math.Round(cm * 3.28084e-2, 2); // gerundet auf zwei Stellen
    Console.WriteLine("Du bist " + feet + " Fuß groß.");

    // Verabschiedung
    CopyrightMeldungAusgeben(ConsoleColor.Red, 0.5);
    Console.WriteLine("AUF WIEDERSEHEN!");
}
```

⁹ 277,18 Hz entspricht genau der Note C # (siehe [https://en.wikipedia.org/wiki/C%23_\(musical_note\)](https://en.wikipedia.org/wiki/C%23_(musical_note)))

¹⁰ [https://msdn.microsoft.com/de-de/library/4fe3hdb1\(v=vs.110\).aspx](https://msdn.microsoft.com/de-de/library/4fe3hdb1(v=vs.110).aspx)

Programmausgabe:

HALLÖCHEN!
 ©SLOG, Wien 2017
 Datum und Uhrzeit: 15.09.2017 23:05:56 C#-Beep!!!
 Gib deine Körpergröße in Zentimeter ein: 163
 Du bist 5,35 Fuß groß.
 ©SLOG, Wien 2017
 Datum und Uhrzeit: 15.09.2017 23:06:02 C#-Beep!!!!
 AUF WIEDERSEHEN!

Man kann eine Methode auch so aufrufen, dass die Werte für die Inputparameter mit dem Namen und in beliebiger Reihenfolge angegeben sind. In unserem Beispiel z. B.:

```
CopyrightMeldungAusgeben(sekunden: 0.3, farbe: ConsoleColor.Yellow);
```

3.3 Wie Methoden in C# Werte zurückliefern können

Methoden können nicht nur durch Input-Parameter Werte erhalten, sondern auch über mehrere Arten Werte zurückliefern oder in übergebenen Parametern Werte so verändern, dass sie an der Programmstelle des Methodenaufrufs verwendet werden können.

3.3.1 Einen einzigen Wert - return

Die einfachste Art Methoden etwas zurückzugeben ist mittels eines Rückgabewerts mit `return`. Und das geht relativ einfach. Statt `void` schreibt man links vom Methodennamen den Datentyp hin, den die Methode zurückgeben soll.

Beispiel: Implementiere eine Methode die zählt, wieviel Selbstlaute in einem Text enthalten sind.

C# Methode die zählt wieviel Selbstlaute in einem Text enthalten sind

```
static int CountVowels(string s)
{
    int counter = 0;
    foreach (char c in s)
    {
        if ("aAeEiIoUu".Contains(c))
        {
            counter++;
        }
    }
    return counter;
}
static void Main(string[] args)
{
    string[] words = { "Kürbiscremesuppe", "Erdäpfelpüree", "Schokomuffins" };
    foreach (string word in words)
    {
        Console.WriteLine("<" + word + "> enthält " + CountVowels(word) + " Selbstlaute.");
    }
}
```

Programmausgabe:

<Kürbiscremesuppe> enthält 5 Selbstlaute.
 <Erdäpfelpüree> enthält 4 Selbstlaute.
 <Schokomuffins> enthält 4 Selbstlaute.

3.3.2 Mehrere Werte

Wenn eine Methode mehr als einen Wert zurückgeben muss, gibt es grundsätzlich drei Arten:

- Variante 1: Mit Output-Parametern
- Variante 2: Mit einer Struct (Aufwendig und unüblich!)
- Variante 3: Mit Value Tuples

Die drei Varianten werden anhand des folgenden Beispiels in den folgenden drei Abschnitten erläutert.

Beispiel: Erstelle eine Methode die alle Selbstlaute und Umlaute zählt und deren Anzahl zurückgibt.

3.3.2.1 Variante 1: Output-Parameter befüllen

Outputparameter sind in einer Methode Variablen, die befüllt werden. Sie werden in der Parameterliste der Methode und beim Aufruf der Methode durch das Wort **out** markiert.

Lösung mit Output-Parametern:

Zwei Output-Parameter werden in der Parameterliste deklariert, counterVowels und counterDigits in der Methode auf berechnet. Output-Parametern muss innerhalb einer Methode immer zumindest einmal ein Wert zugewiesen werden. Beim Aufruf können dann die Werte verwendet werden.

C# Methode mit Output-Parametern

```
static void Count(string s, out int counterVowels, out int counterDigits)
{
    counterVowels = counterDigits = 0;

    foreach (char c in s)
    {
        if ("aAeEiIoOuU".Contains(c))
        {
            counterVowels++;
        }
        else if (char.IsDigit(c) )
        {
            counterDigits++;
        }
    }
}

static void Main(string[] args)
{
    Count("Mondlandung 1969", out int vowels, out int digits);

    Console.WriteLine(vowels + " Selbstlaute und " + digits + " Ziffern");
}
```

Programmausgabe:

3 Selbstlaute und 4 Ziffern

3.3.2.2 Variante 2: Struct als Rückgabewert

Bei dieser Variante werden alle Werte die in der Methode berechnet werden, in einer Struct definiert und in der Methode befüllt. Wie das untere Programm zeigt, ist das allerdings eher aufwendig und mühselig.

Diese Variante ist schon deshalb abzulehnen, weil es notwendig ist, einen zusätzlichen Datentypen, die Struct, zu definieren.

Lösung mit Struct:

Eine Struct mit den zwei Zählern als Komponenten wird definiert, counterVowels und counterDigits. In der Methode wird eine Variable mit dieser Struct als Datentyp definiert, befüllt und zurückgegeben.

Im Main wird die Struct als Returnwert gespeichert und die Komponenten ausgelesen.

C# Methode mit Struct als Rückgabewert

```
struct ReturnValues
{
    public int counterVowels;
    public int counterDigits;
}

static ReturnValues Count(string s)
{
    ReturnValues result;
    result.counterVowels = result.counterDigits = 0;

    foreach (char c in s)
    {
        if ("aAeEiIoOuU".Contains(c))
        {
            result.counterVowels++;
        }
        else if (char.IsDigit(c) )
        {
            result.counterDigits++;
        }
    }

    return result;
}

static void Main(string[] args)
{
    ReturnValues r = Count("Mondlandung 1969");

    Console.WriteLine(r.counterVowels + " Selbstlaute und " + r.counterDigits + " Ziffern");
}
```

Programmausgabe:

3 Selbstlaute und 4 Ziffern

3.3.2.3 Variante 3: Value Tuple als Rückgabewert

Diese Variante ist die modernste Form, weil sie ohne zusätzliche Struct und ohne Output-Parameter auskommt. **Value Tuples werden genau in Abschnitt 3.6 erklärt.**

Lösung mit Value Tuple:

Die zwei Zähler werden als lokale Variablen counterVowels und counterDigits angelegt. Am Ende der Methode werden diese als Value Tuple mit return zurückgegeben. Dafür ist es nur notwendig, alle lokalen Variablen in runden Klammern aufzuzählen.

Im Main wird die Struct als Returnwert gespeichert und die Komponenten ausgelesen.

C# Methode mit Struct als Rückgabewert

```
static (int vowels, int digits) Count(string s)
{
    int counterVowels = 0, counterDigits = 0;

    foreach (char c in s)
    {
        if ("aAeEiIoOuU".Contains(c))
        {
            counterVowels++;
        }
        else if (char.IsDigit(c) )
        {
            counterDigits++;
        }
    }

    return (counterVowels, counterDigits);
}

static void Main(string[] args)
{
    var res = Count("Mondlandung 1969");

    Console.WriteLine(res.vowels + " Selbstlaute und " + res.digits + " Ziffern");
}
```

Programmausgabe:

3 Selbstlaute und 4 Ziffern

3.4 Wie man eine Methode schreibt

Jede Methode in einer Programmiersprache besteht aus zwei Teilen, einem ersten Teil der Methodensignatur¹¹, Methodenkopf oder auch Prototyp einer Methode und einem zweiten Teil, der Methodenimplementierung.

Beispiel:

```
double BerechneGeometrischesMittel(double wert1, double wert2)
{
    return Math.Sqrt(wert1 * wert2);
```

Signatur

Implementierung

¹¹ Beachte, dass der Ausdruck in der Fachliteratur manchmal den Rückgabetyp nicht miteinschließt. Das vernachlässigt der Autor hier allerdings.

Bevor man damit beginnt, eine Methode zu implementieren, ist es wichtig sich zunächst über drei Fragen klar zu werden:

1. Was soll die Methode überhaupt tun?	Implementierung
2. Was kommt in die Methode rein? Also welche Daten.	Signatur
3. Was kommt aus der Methode raus? Welche und wieviel Daten.	

3.5 Signatur einer Methode

Die Erstellung der Signatur ist zunächst wichtiger als die Implementierung selbst, weil eine falsche Signatur, eine korrekte Implementierung unmöglich macht.

Für die Erstellung einer Methode ist das Wichtigste zu klären, was die Methode überhaupt tun soll. In der Praxis zeigt sich, dass dieser Punkt oft weniger trivial ist, als man denken mag.

Wenn geklärt ist, was die Methode tun soll, ist der nächste Schritt die Erstellung der Methodensignatur.

Für die Erstellung der Methodensignatur sollte man sich vor der Implementierung drei Fragen beantworten:

Die GROSSEN drei Fragen!	
1. Welche Werte benötigt die Methode?	-> Inputparameter
2. Soll die Methode etwas zurückliefern und wenn ja, was?	-> Returnwert oder Output-Parameter
3. Was ist der beste Name für die Methode?	

Die Antworten auf diese Fragen bestimmen die Signatur der Methode, also die erste Zeile der Methodendefinition.

3.5.1 Beispiele zur Signatur in C#

In den folgenden Beispielen wird die erste Zeile der Methode durch die Beantwortung der drei Fragen systematisch erstellt.

Beispiel 1:

Erstelle eine Methode die für die Körpergröße in Zentimeter und das Gewicht in Kilogramm den Body-Mass-Index (BMI) berechnet.

Lösung in C#:

1. Welche Werte benötigt die Methode?

Die Körpergröße in Zentimeter ist ein Ganzzahlwert. -> Datentyp `int` height

Das Gewicht in Kilogramm, wobei Waagen meist eine Dezimalzahl anzeigen. -> `double` weight

_____ (`int` height, `double` weight)

2. Was soll die Methode zurückliefern?

Die Methode soll den Body-Mass-Index berechnen. Das ist ein einziger Wert. Eine Google-Suche ergibt, dass der Wert eine Dezimalzahl sein kann <https://www.bmi-online.info>.

-> ein Rückgabewert mit Datentyp **double**

double _____(**int** height, **double** weight)

3. Was ist der beste Name für die Methode?

Die Methode berechnet den BMI, daher sollte das Zeitwort „berechne“ oder „calculate“ enthalten sein. Alle Methoden beginnen in C# mit großem Anfangsbuchstaben. -> CalculateBMI oder BerechneBMI.

Die Methodensignatur ist also **double** CalculateBMI(**int** height, **double** weight) oder auf Deutsch **double** BerechneBMI(**int** hoehe, **double** gewicht).

Beispiel 2:

Eine Methode soll einen Ton einer bestimmten Frequenz in Hertz (Hz) für eine Sekunde am Lautsprecher ausgeben.

Lösung in C#:

1. Welche Werte benötigt die Methode?

Die Frequenz des Tons. -> Datentyp **int** frequency

Die Dauer von einer Sekunde ist fix und ist daher kein Inputparameter!

_____ (**int** frequency)

2. Was soll die Methode zurückliefern?

Nichts. Denn die Ausgabe erfolgt am Lautsprecher. Die Methode muss nichts zurückgeben. -> void

void _____(**int** frequency)

3. Was ist der beste Name für die Methode?

Die Methode lässt den Lautsprecher beepen. -> Beep

Die Methodensignatur ist also **void** Beep(**int** frequency).¹²

Beispiel 3:

Eine Methode soll aus einem Array von Texten die größte und die kleinste Textlänge ermitteln und zurückgeben.

Erstelle die Signatur dieser Methode.

Lösung in C# mit Output-Parameter:

1. Welche Werte benötigt die Methode?

Es wird das Array von Texten benötigt. -> Datentyp **string[]** texts

_____ (**string[]** texts)

¹² Die Methode kann in C# übrigens mithilfe von Console.Beep(**int** frequency, **int** duration) implementiert werden.

2. Was soll die Methode zurückliefern?

Die Methode soll zwei Werte berechnen. Textlängen sind immer ganze Zahlen, also wählen wir den Datentyp `int`. Da es zwei Werte sind, benötigen wir zwei Output-Parameter anstelle eines Rückgabewerts. -> `out double`

```
void _____(string[] texts, out int longest, int double shortest)
```

3. Was ist der beste Name für die Methode?

Die Methode findet die beiden Längen. -> `FindGreatestAndShortestLengths`

Die gesamte Methodensignatur ist in C# also

```
void FindGreatestAndShortestLengths(string[] texts, out int longest,
                                     out int shortest)
```

Lösung in C# mit Value Tuples:

1. Welche Werte benötigt die Methode?

Es wird das Array von Texten benötigt. -> Datentyp `string[] texts`

```
____ _____(string[] texts)
```

2. Was soll die Methode zurückliefern?

Die Methode soll zwei Werte berechnen. Textlängen sind immer ganze Zahlen, also wählen wir den Datentyp `int`. Da es zwei Werte sind, benötigen wir ein Value Tuple mit zwei Werten.

-> `(int longest, int shortest)`

```
(int longest, int shortest) _____(string[] texts)
```

3. Was ist der beste Name für die Methode?

Die Methode findet die beiden Längen. -> `FindGreatestAndShortestLengths`

Die gesamte Methodensignatur ist in C# also

```
(int longest, int shortest) FindGreatestAndShortestLengths(string[] texts)
```

3.6 Value Tuples in C#

Methoden die mehr als einen Wert berechnen oder ermitteln, können in C# mit Hilfe von Output-Parametern implementiert werden.

Das ist aber nicht die einzige Möglichkeit! Output-Parameter gelten in einigen Entwicklungsteams als schlechter Code, d. h. sie haben einen sogenannten [Code Smell](#) und werden in den [Coding Conventions](#) von Entwicklerteams auch verboten.

Viele Sprachen, wie Java oder Dart kennen überhaupt keine Output-Parameter. In C# sind sie vermutlich aufgrund der Vorgängersprache C++ eingebaut worden, in denen die Rückgabe mehrerer Werte mittels Referenzen möglich ist, was sehr ähnlich zu Output-Parameter funktioniert.

Es gibt etwas Besseres als Output-Parameter, nämlich **Tuples**. In C# gibt es gleich zwei Arten von Tuples, [System Tuples](#) (seit .NET 4.0) und [Value Tuples](#) (seit .NET 4.7).

Value Tuples sind weitaus praktikabler und das Skriptum beschränkt sich ausschließlich auf diese Art von Tuplen.

Tuples sind in fast allen Programmiersprachen üblich.

3.6.1 Verwendung von Value Tuples

Bei Value Tuples in C# werden mehrere beliebige Werte "zu einem Ganzen" zusammenzubinden. Das geht ganz einfach durch runde Klammern.

Value Tuples sind sehr ähnlich zu Structs. Sie bestehen aus einzelnen Komponenten, die einen bestimmten Typ haben und denen man Namen geben kann, aber nicht muss.

C#

Anlegen eines Tuples

```
( "Max", 167, 56.9 )
```

Tuples werden einfach durch runde Klammern gebildet. Die Anzahl der Werte ist beliebig. Dieses Tuple besteht aus einem String, einem Integer und einem Double.

Zuweisung zu einer Variablen data

Variante 1:

```
string name, int height, double weight) data = ("Max", 167, 56.9);
Console.WriteLine(data.name +": "+data.height + " cm, " + data.weight + " kg");
```

Variante 2:

```
var data = (name: "Max", height: 167, weight: 56.9);
Console.WriteLine(data.name +": "+data.height + " cm, " + data.weight + " kg");
```

Variante 3:

```
var data = ("Max", 167, 56.9);
Console.WriteLine(data.Item1 +": "+data.Item2 + " cm, " + data.Item3 + " kg");
```

Bemerkungen:

`var` ist eine Art „Faulheitsschlüsselwort“. Mit `var` erspart man sich einen Datentyp vor einer Variablen anzugeben. Der Compiler ermittelt diesen Datentyp selbst.

`var` kann nicht nur bei Value Tuples, sondern bei allen Datentypen in C# verwendet werden.

Die Variante 3 ist aufgrund der schlechten Lesbarkeit von Item1, Item2, usw. keine gute Variante.

Programmausgabe auf der Konsole für alle drei Varianten:

Max: 167 cm, 56.9 kg

3.6.2 Value Tuple versus Output Parameter

Ein Tuple kann von einer Methode auch als Rückgabewert zurückgegeben werden.

Das hat den großen Vorteil, dass alle Werte, die von einer Methode mittels Output-Parameter zurückgeliefert werden würden, nun einfach in ein Value Tuple gepackt werden kann.

In der Methodensignatur kann man die Typen und die Namen angeben und die Werte dann mit return zurückgeben.

Das folgende Beispiel zeigt, den Vergleich zu Output-Parametern.

Beispiel: Minimum- und Maximumssuche in einem Array

Eine Methode soll in einem int-Array den größten und kleinsten Wert suchen.

Erstelle die Methode, einmal mit Output-Parametern und einmal mit einem Value-Tuple.

Lösung:

Methode mit Output-Parameter:	Methode mit Value Tuple:
<pre>static void FindMinMax(int[] numbers, out int min, out int max) { min = int.MaxValue; max = int.MinValue; for (int i = 0; i < numbers.Length; i++) { if (numbers[i] < min) { min = numbers[i]; } if (numbers[i] > max) { max = numbers[i]; } } static void Main(string[] args) { int[] values = { 10, 5, 8, 3, 2, 7 }; FindMinMax(values, out int mi, out int ma); Console.WriteLine(mi + ", " + ma); } }</pre>	<pre>static (int min, int max) FindMinMax(int[] numbers) { int mi = int.MaxValue; int ma = int.MinValue; for (int i = 0; i < numbers.Length; i++) { if (numbers[i] < mi) { mi = numbers[i]; } if (numbers[i] > ma) { ma = numbers[i]; } } return (mi, ma); } static void Main(string[] args) { int[] values = { 10, 5, 8, 3, 2, 7 }; var res = FindMinMax(values); Console.WriteLine(res.min + ", " + res.max); }</pre>
Ausgabe: 2, 10	Ausgabe: 2, 10

3.7 Übungen zu Methodensignaturen

Löse folgende Übungsbeispiele in C#.

Übung 1: Eine Methode soll für einen Vornamen und einen Nachnamen, die Initialen zurückgeben.
z. B. "Arnold" und "Schwarzenegger" -> Rückgabe "A. S."

Erstelle die Methodensignatur.

Übung 2: Eine Methode soll zwei Strings vergleicht und den längeren String zurückgeben.
Erstelle die Methodensignatur.

Übung 3: Eine Methode soll zurückgeben, ob ein String einen Selbstlaut enthält. Selbstlaute sind a, e, i, o und u (Groß- und Kleinbuchstaben).

Erstelle die Methodensignatur.

Übung 4: Eine Methode soll aus einem Array von Texten, alle jene Texte finden, die eine bestimmte Länge haben. Diese sollen in einem Array zurückgegeben werden.

Erstelle die Methodensignatur.

Übung 5: Eine Methode soll in einem String die Anzahl der Ziffern, der Groß- und Kleinbuchstaben zählen.

Erstelle die Methodensignatur.

Übung 6: Eine Methode soll aus einem double-Array alle negativen Werte entfernen.
Erstelle die Methodensignatur.

3.8 Vararg-Parameter – Schlüsselwort params

Man kann einen Parameter auch so definieren, dass man ein Array übergibt, dieses aber beim Aufruf nicht angelegt werden muss! Es reicht, wenn man beim Anruf die Werte durch Beistriche getrennt angibt.

Das ist mit dem Schlüsselwort `params` möglich.

C#

```
double CalcAverage(params double[] a)
{
    ...
}
```

Aufruf im Main:

```
double value = CalcAverage(1.5, 3, 2.1);
Console.WriteLine(value);
```

Programmausgabe:

```
2.2
```

3.8.1 Codebeispiel für Vararg-Parameter

Im folgenden Programm wird die Methode CalcAverage im Main mit einer verschiedenen Anzahl von Parametern aufgerufen.

C# Programm zur Berechnung des Mittelwerts beliebig vieler double Werte

```
using System;

namespace VarArg
{
    class Program
    {
        /// <summary>
        /// Calculates the average of an arbitrary number of double values
        /// </summary>
        /// <param name="values">array of values</param>
        /// <returns>average value</returns>
        static double CalcAverage(params double[] values)
        {
            double average = 0d;

            foreach ( double value in values )
            {
                average += value;
            }

            average /= values.Length;

            return average;

            // Note: convenient way to compute the average of an array
            //return values.ToList().Average();
        }
    }
}
```

```

static void Main(string[] args)
{
    Console.Write("Enter 1st value: ");
    double value1 = double.Parse(Console.ReadLine());

    Console.Write("Enter 2nd value: ");
    double value2 = double.Parse(Console.ReadLine());

    Console.Write("Enter 3rd value: ");
    double value3 = double.Parse(Console.ReadLine());

    double mean1 = CalcAverage(value1, value2);           //return two values
    double mean2 = CalcAverage(value1, value2, value3); //return three values

    Console.WriteLine("The average value of the 1st and 2nd value is {0}.", mean1);
    Console.WriteLine("The average value of all values is {0}.", mean2);

    double[] twovalues = new double[] { value2, value3 };
    double mean3 = CalcAverage(twovalues);

    Console.WriteLine("The average value of the 2nd and 3rd value is {0}.", mean3);
}
}

```

4 Formatierte Ausgaben in C#

Wenn man mit `Console.WriteLine` Werte ausgibt, dann kann man sie einfach mit + aneinanderhängen.

C# Aneinanderhängen mit +

```
static void Main(string[] args)
{
    Console.Write("Geben Sie die °C ein: ");
    double celsius = double.Parse(Console.ReadLine());

    double kelvin = celsius + 273.15;
    Console.WriteLine(celsius + "°C sind " + kelvin + " K");
}
```

Hinweis: Grundsätzlich wird dabei durch den Compiler für jede Variable die nicht vom Typ `string` ist, die Methode `ToString()` aufgerufen, also eigentlich wird

```
Console.WriteLine(celsius.ToString() + "°C sind " + kelvin.ToString() + " K");
```

ausgeführt.

Das Verknüpfen von den Strings mit + ist zur Laufzeit sehr ineffizient. Es werden zahlreiche Teilstrings angelegt, die durch die Garbage Collection erst wieder entfernt werden müssen.

Es gibt zwei Arten Ausgaben zu formatieren die wesentlich performanter sind und in der Praxis daher fast ausschließlich verwendet werden:

1. **Interpolated Strings:** siehe Abschnitt 0
2. **Indexbasierte Formatierung:** siehe Abschnitt 4.2

Das folgende Programm gibt einen einfachen kompakten Überblick.

C# - Drei Arten von Ausgaben

```
static void Main(string[] args)
{
    double celsius = 20;
    double kelvin = celsius + 273.15;

    Console.WriteLine(celsius+ " °C sind "+kelvin+ " K");      // Schlecht!
    Console.WriteLine($"{celsius} °C sind {kelvin} K");          // Interpolated Strings
    Console.WriteLine("{0} °C sind {1} K", celsius, kelvin);     // Indexbasiertes Format
}
```

Programmausgabe:

```
20 °C sind 293.15 K
20 °C sind 293.15 K
20 °C sind 293.15 K
```

4.1 Interpolated Strings

Bei interpolated Strings wird nur ein String verwendet und die Variablen werden in geschwungene Klammern gesetzt. Statt {celsius} wird der Wert von celsius eingefügt.

Jeder beliebige Ausdruck kann in den geschwungenen Klammern verwendet werden:

C# Interpolated Strings

```
static void Main(string[] args)
{
    Console.WriteLine("Geben Sie die °C ein: ");
    double celsius = double.Parse(Console.ReadLine());
    double fahrenheit = celsius * 1.8 + 32;
    Console.WriteLine($"{celsius} °C sind {celsius + 273.15} K und {fahrenheit} °F.");
}
```

Programmausgabe:

```
Geben Sie die °C ein: 20
20 °C sind 293.15 K und 68 °F.
```

Im Internet gibt es zahlreiche Links mit guten Erklärungen von Interpolated Strings.

<https://www.dotnetperls.com/string-interpolation>

<https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/string-interpolation>

Interpolated Strings funktionieren übrigens nicht nur mit Console.WriteLine sondern auch bei allen anderen Fällen, dass ein String zusammengesetzt wird.

C# Interpolated Strings bei einer Zuweisung

```
static void Main(string[] args)
{
    Console.WriteLine("Geben Sie die °C ein: ");
    double celsius = double.Parse(Console.ReadLine());
    double fahrenheit = celsius * 1.8 + 32;
    string s = $"{celsius} °C sind {celsius + 273.15} K und {fahrenheit} °F.";
    Console.WriteLine(s);
}
```

Programmausgabe:

```
Geben Sie die °C ein: 20
20 °C sind 293.15 K und 68 °F.
```

4.2 Indexbasierte Formatierung

Folgendes Beispiel zeigt das Prinzip, das nicht nur für Console.WriteLine, sondern auch mit der Methode String.Format genutzt werden kann.

C# Indexbasierte Formatierung

```
static void Main(string[] args)
{
    double celsius = 5, kelvin = 278.15, fahren = 41;

    double s = String.Format("{0} °C sind {1} K und {2} °F.", celsius, kelvin, fahren);
    Console.WriteLine(s);
}
```

Programmausgabe:

```
5 °C sind 278.15 K und 41 F.
```

Der erste Wert ist immer ein sogenannter Formatstring in dem Platzhalter für einzusetzende Werte stehen: {0}, {1}, {2}, ... sind die Platzhalter. Die Zahl innerhalb der geschwungenen Klammern ist ein Index auf die nachfolgenden Werte

Beispiel:

Formatstring		0 1 2
--------------	--	-----------------

```
Console.WriteLine("{0} °C sind {1} K und {2} F.", celsius, kelvin, fahrenheit);
```

Die Platzhalter werden im Programm durch den Wert des entsprechenden Index ersetzt. Farbig dargestellt: "5 °C sind 278,15 K und 41 F."

Ausgegeben wird daher 5 °C sind 278,15 K und 41 F.

Beispiel: Ausgabe von Quadrat- und Wurzelzahlen in C#

```
for (double x = 1.0; x <= 6.0; x++)
{
    Console.WriteLine("{0}² = {1}", x, x * x);

for (double x = 1.0; x <= 6.0; x++)
{
    Console.WriteLine("\u221A{0} = {1}", x, Math.Sqrt(x));
}
```

Unicode für das Wurzelzeichen¹³

Programmausgabe:

```
1² = 1
2² = 4
3² = 9
4² = 16
5² = 25
6² = 36
√1 = 1
√2 = 1,4142135623731
√3 = 1,73205080756888
√4 = 2
√5 = 2,23606797749979
√6 = 2,44948974278318
```

Es kann auch sinnvoll sein einen Index im Formatierungsstring mehrmals zu verwenden, so wie im folgenden Beispiel den Index 0 in {0}.

Beispiel: Ausgabe von Quadrat- und Wurzelzahlen in einer Zeile in C#

```
for (double x = 1.0; x <= 6.0; x++)
{
    Console.WriteLine("{0}² = {1} \u221A{0} = {2}", x, x * x, Math.Sqrt(x));
```

Programmausgabe:

```
1² = 1 √1 = 1
2² = 4 √2 = 1,4142135623731
3² = 9 √3 = 1,73205080756888
4² = 16 √4 = 2
5² = 25 √5 = 2,23606797749979
6² = 36 √6 = 2,44948974278318
```

¹³ <https://unicode-table.com/de/221A/>

Nicht nur bei Console.Write und Console.WriteLine kann diese Formatierung mit Index und geschwungenen Klammern verwendet werden. Auch die Methode String.Format eignet sich dafür.

Beispiel: ToString() in einer Klasse

```
class Person
{
    public string Vorname { get; }
    public string Nachname { get; }
    public DateTime Geburtstag { get; private set; }

    public Person(string v, string n, DateTime dt)
    {
        Vorname = v;
        Nachname = n;
        Geburtstag = dt;
    }

    public override string ToString()
    {
        return String.Format("{0} {1} ist im Jahr {2} geboren.", Vorname, Nachname,
                             Geburtstag.Year);
    }
}
```

4.3 Formatierung mit ToString()¹⁴

Es gibt eine weitere Art Ausgaben zu formatieren, die vereinzelt verwendet wird. Sie ist eher als Ergänzung zu den beiden vorhergehenden Formatierungsvarianten zu verstehen.

Standardformatierungen, z. B. `ToString("N2")` für zwei Nachkommastellen (siehe [https://msdn.microsoft.com/de-de/library/dwhawy9k\(v=vs.110\).aspx](https://msdn.microsoft.com/de-de/library/dwhawy9k(v=vs.110).aspx)) und benutzerdefinierte Formatierungen, z. B. `ToString("####.##")` für vier Vorkomma- und zwei Nachkommastellen (siehe [https://msdn.microsoft.com/de-de/library/0c899ak8\(v=vs.110\).aspx](https://msdn.microsoft.com/de-de/library/0c899ak8(v=vs.110).aspx)).

C# Ausgabe der Werte mit ToString() und Formatangaben

```
static void Main(string[] args)
{
    Console.WriteLine("Geben Sie die °C ein: ");

    double celsius = double.Parse(Console.ReadLine());
    double kelvin = celsius + 273.15;

    Console.WriteLine(celsius.ToString("N2") + " °C sind " + kelvin.ToString("N2") + " K");
}
```

¹⁴ Die Methode `ToString()` ist überladen, d. h. es gibt eine zweite Methode mit dem Namen `ToString`, die einen Inputparameter hat: `string ToString(string format)`

5 Das Summenzeichen und C#

Wenn viele Zahlen systematisch summiert werden, verwendet man in der Mathematik häufig das Summenzeichen, als abgekürzte Schreibweise.

Das Summenzeichen ist für C#, so etwas wie eine Schleife über einen Index der von einem Start- bis zu einem Endwert läuft.

Der Ausdruck hinter dem Summenzeichen wird jeweils berechnet und zur Summe dazu addiert.

Beispiele:

$1+2+3+4+5 = \sum_{x=1}^5 x$	<pre>int summe = 0; for (int x = 1; x <= 5; x++) { summe = summe + x; } Console.WriteLine("Die Summe ist " + summe);</pre>
$\sum_{x=1}^4 x^2 = 1^2 + 2^2 + 3^2 + 4^2 = 30$	<pre>int summe = 0; for (int x = 1; x <= 4; x++) { summe = summe + x*x; } Console.WriteLine("Die Summe ist " + summe);</pre>
$\sum_{x=0}^4 2^x = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 = 31$	<pre>int summe = 0; for (int x = 0; x <= 4; x++) { // addiert 2 hoch x summe = summe + (int)Math.Pow(2, x); } Console.WriteLine("Die Summe ist " + summe);</pre>

Welcher Buchstabe als Summenvariable verwendet wird, ist übrigens völlig egal. Häufig wird statt x auch i oder j verwendet.

Bei einer größeren Grundgesamtheit wird der Buchstabe im Summenzeichen oft als Index verwendet.

Beispiel: Von fünf Temperaturwerten soll der Durchschnitt berechnet werden.

T ₁	19,4 °C
T ₂	28,0 °C
T ₃	23,1 °C
T ₄	25,8 °C
T ₅	18,7 °C

In C# sind die Werte in einem Array gespeichert. Daher wird das Summenzeichen als eine Schleife über das Array implementiert.

<p>Die allgemeine Formel für n-viele Temperaturen ist:</p> $\bar{x} = \frac{\sum_{j=1}^n T_j}{n}$ <p>Da es fünf Temperaturwerte sind, ist der Wert n = 5.</p> $\bar{x} = \frac{\sum_{j=1}^5 T_j}{5}$	<pre>double[] temp = { 19.4, 28, 23.1, 25.8, 18.7 }; double summe = 0; for (int i = 0; i < temp.Length; i++) { summe = summe + temp[i]; } double mittelwert = summe / temp.Length; Console.WriteLine("Die mittlere Temperatur ist " + + mittelwert + "°C.");</pre>
--	---

Beachte, dass man statt

i < 5

den Ausdruck

i < temp.Length

verwendet, der auch den Wert 5 hat.

Das hat den großen Vorteil, dass bei einer Änderung des Arrays, kein weiterer Code zu ändern ist!

6 Ausgewählte Methoden für Collections

In Übungsbeispielen zu Methoden finden sich Aufgabestellungen, die nach gewissen Mustern gelöst werden können.

Die in diesem Abschnitt angeführten Methodenmuster sind für alle Arten von Collections, also Arrays, Strings (String = verkettete Strings), Listen, usw. anwendbar. Einfachheitshalber werden sie hier für int-Arrays demonstriert.

Beispiel:

Gegeben ist ein int-Array beliebiger Größe.

```
int[] num = { 4, 10, 7, 8, 3, 2, 15, 6, 8, 3, 12 };
```

Gesucht sind nun folgende Methoden, die das Array analysieren.

1. Schreibe eine Methode, die die Summe aller Zahlen im Array zurückgibt.
2. Schreibe eine Methode, die zählt, wie oft eine bestimmte Zahl im Array enthalten ist.
3. Schreibe eine Methode, die zurückgibt, ob eine bestimmte Zahl im Array enthalten ist.
4. Schreibe eine Methode, die zurückgibt, ob alle Zahlen gerade sind.
5. Schreibe eine Methode, die alle geraden Zahlen zurückgibt.

Lösung:

In jeder dieser Methoden befindet sich eine Schleife, die durch das Array durchläuft¹⁵. In dem Beispielcode wird eine for-Schleife verwendet. Sinnvollerweise ist der Einsatz einer foreach-Schleife weitaus einfacher, da die Schleifenvariable i innerhalb der Schleife nur bei numbers[i] verwendet wird. Die Methoden unterscheiden sich aber sehr wesentlich darin, was sie mit den Array-Elementen machen.

6.1 Summieren und Zählen (Aggregatmethoden)

Die ersten beiden Methoden laufen immer über das gesamte Array und sind typische Summation und Zählsituationen, wie man sie aus den allerersten Programmen zur Genüge kennt.

Am Beginn werden die Variablen summe bzw. der Zähler count mit 0 initialisiert und in der Schleife gegebenenfalls erhöht.

C# Methoden

```
// 1. Schreibe eine Methode, die die Summe aller Zahlen im Array zurückgibt.
static int SumUp(int[] numbers)
{
    int summe = 0;

    for (int i = 0; i < numbers.Length; i++)
    {
        summe += numbers[i];
    }
    return summe;
}
```

¹⁵ Hinweis: Mit der Spracherweiterung LINQ können praktisch alle diese Methoden wesentlich gekürzt werden. Dafür sind allerdings Lambdas erforderlich, die funktionale Elemente von C# sind.

```

// 2. Schreibe eine Methode, die zählt, wie oft eine bestimmte Zahl im Array enthalten ist.
static int Count(int[] numbers, int value)
{
    int count = 0;

    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] == value)
        {
            count++;
        }
    }

    return count;
}

static void Main(string[] args)
{
    int[] num = { 4, 10, 7, 8, 3, 2, 15, 6, 8, 3, 12 };

    Console.WriteLine($"Summe = {SumUp(num)}");

    int val = 3;
    Console.WriteLine($"{val} ist {Count(num, val)} mal enthalten.");
}

```

Programmausgabe:

Summe = 78
3 ist 2mal enthalten.

6.2 Existenzmethode

Die 3. Methode könnte man als **Existenzmethode** bezeichnen. Sie gibt zurück, ob zumindest ein Element eine bestimmte Eigenschaft hat. Wenn ein Element gefunden wurde, kann die Methode sofort `true` zurückgeben, am Ende der Methode dagegen `false`, falls das nicht der Fall ist.

C# Code einer Existenzmethode

```

// 3. Schreibe eine Methode, die zurückgibt, ob eine bestimmte Zahl im Array enthalten ist.
// =Existenzmethode
static bool Contains(int[] numbers, int value)
{
    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] == value)
        {
            return true;
        }
    }

    return false;
}

static void Main(string[] args)
{
    int[] num = { 4, 10, 7, 8, 3, 2, 15, 6, 8, 3, 12 };
    Console.WriteLine(Contains(num, 15));
    Console.WriteLine(Contains(num, 5));
}

```

Programmausgabe:

True
False

6.3 Validierungsmethode

Die 4. Methode erfordert dagegen, dass alle Array-Elemente überprüft werden müssen um festzustellen, ob wirklich alle Elemente eine bestimmte Bedingung erfüllen. Dann gibt die Methode `true` zurück. Das sind sogenannte **Validierungsmethoden**.

Also erst wenn alle Elemente in Ordnung sind bzw. eine bestimmte Eigenschaft besitzen, gibt die Methode `true` zurück. Bei nur einem Element, das die Bedingung nicht erfüllt, gibt die Methode `false` zurück.

Es ist nicht notwendig eine lokale Variable für das Resultat anzulegen. Eine Methode kann auch innerhalb einer Schleife die Methode mit `return` verlassen.

C# Code einer Validierungsmethode

```
// 4. Schreibe eine Methode, die zurückgibt, ob alle Zahlen gerade sind.  
// =Validierungsmethode  
static bool AreAllEven(int[] numbers)  
{  
    for (int i = 0; i < numbers.Length; i++)  
    {  
        if (numbers[i] % 2 == 1)  
        { // Zahl ist ungerade  
            return false;  
        }  
    }  
  
    return true;  
}  
  
static void Main(string[] args)  
{  
    int[] num = { 4, 10, 7, 8, 3, 2, 15, 6, 8, 3, 12 };  
  
    Console.WriteLine(AreAllEven(num) ? "Alle Zahlen sind gerade."  
                                : "Nicht alle Zahlen sind gerade.");  
}
```

Programmausgabe:

Nicht alle Zahlen sind gerade.

6.4 Filtermethode

Die 5. Methode ist eine **Filtermethode**, da sie die Array-Elemente gemäß einem bestimmten Kriterium überprüfen und alle jene in einem neuen Array (oder einer anderen Collection) zurückgibt.

Die im folgenden Beispiel gezeigte Methode können genauso auf Strings umgelegt werden, da diese ja aus einzelnen Zeichen bestehen, die als Array verstanden werden können.

C# Code einer Filtermethode

```
// 5. Schreibe eine Methode, die alle geraden Zahlen zurückgibt.
static int[] GetOdd(int[] numbers)
{
    int[] odd = new int[numbers.Length];
    int j = 0;

    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] % 2 == 0)
        {   // Zahl ist gerade
            odd[j] = numbers[i];
            j++;
        }
    }

    Array.Resize(ref odd, j); // Das Resultatsarray auf die richtige Größe bringen.

    return odd;
}

static void Main(string[] args)
{
    int[] num = { 4, 10, 7, 8, 3, 2, 15, 6, 8, 3, 12 };

    int[] result = GetOdd(num);
    Console.WriteLine(String.Join(',', result)); // Rasche Ausgabe eines Arrays.
}
```

Programmausgabe:

4,10,8,2,6,8,12

7 Strings

Ein String ist ein zusammengesetzter Typ der aus einzelnen Zeichen besteht. Jedes dieser Zeichen hat den Typ char.

Durch den Operator [] und der Angabe einer Position kann auf jedes einzelne dieser Zeichen zugegriffen werden.

Beispiel: Der String s `string s = "Hallo!"`; besteht aus den sechs Zeichen 'H', 'a', 'l', 'l', 'o' und '!'.

Jedes dieser Zeichen hat den Datentyp char.

Durch s.Length lässt sich die Länge eines Strings ermitteln.

<code>string s = "Hallo!"</code>					
'H'	'a'	'l'	'l'	'o'	'!'

7.1 Unicode

Jedes einzelne Zeichen entspricht dem sogenannten Unicode.

Der ASCII-Code für ein Zeichen wird durch casten auf den Type int ermittelt.

```
int unicode = (int)'C';
```

Das Zeichen für einen ASCII-Code wird durch casten auf den Type char ermittelt.

```
char zeichen = (char)unicode;
```

Es ist oft nützlich, dass Buchstaben und Zahlen fortlaufende Unicodes haben.

Zeichen	Unicodes
'a', 'b', 'c', ... 'y', 'z'	97, 98, 99, ... 121, 122
'A', 'B', 'C', ... 'Y', 'Z'	65, 66, 67, ... 89, 90
'0', '1', '2', ... '8', '9'	48, 49, 50, ... 56, 57

Beispiel: Durchlaufen eines Strings und Ausgabe des Unicodes jedes Zeichens mit einer for-Schleife und Substring.

```
string s = "Hallo!";  
  
for (int i = 0; i < s.Length; i++)  
{  
    string t = s.Substring(i,1);  
    int unicode = (int)char.Parse(t);  
    Console.WriteLine(t + " = " + unicode);  
}
```

Programmausgabe

```
H = 72  
a = 97  
l = 108  
l = 108  
o = 111  
! = 33
```

7.2 Nützliche Methoden zur Stringmodifikation

String ist in fast allen Programmiersprachen ein sehr wesentlicher Datentyp und es ist nützlich die wichtigsten Methoden zu kennen und anwenden zu können.

Die folgende Übersicht zeigt die wichtigsten Methoden in C#: (Quelle: Skriptum Programmieren und Softwareentwicklung 1)

Eigenschaft / Methode (Auszug)	Typ	Beschreibung
Length	int	liefert die Anzahl der Zeichen
IndexOf(string char val [,int startIndex[, int count]])	int	sucht Position eines übergebenen Zeichens/Strings, liefert -1 wenn erfolglos
Insert(int startIndex, string val)	string	fügt String ab startIndex ein
PadLeft(int totalWidth [, char Füllzeichen])	string	füllt String links mit Füllzeichen auf bis Gesamtbreite totalWidth erreicht ist
PadRight(int totalWidth [, char Füllzeichen])	string	wie PadLeft, nur rechts
Remove(int start [, int count])	string	löscht ab einer bestimmten Position (eine Anzahl an Zeichen)
Replace(char string old, char string new)	string	ersetzt alle Zeichen bzw. Teilstings durch new
Split(char Trenn)	string[]	zerlegt String in Teilstings auf Basis eines Trennzeichens in ein String-Array
Substring(int start [, int len])	string	liefert einen Teilstring ab einer bestimmten Position (in bestimmter Länge)
ToCharArray([int start, int len])	char[]	liefert String (oder Teilstring) als Char-Array
ToLower()	string	konvertiert alle Zeichen in Kleinbuchstaben
ToUpper()	string	konvertiert alle Zeichen in Großbuchstaben
Trim([char zeichen])	string	löscht alle vor- und nachlaufenden Leerzeichen (oder angegebene Zeichen)
TrimStart([char zeichen])	string	löscht alle vorlaufenden Leerzeichen (o. angeg. Z.)
TrimEnd([char zeichen])	string	löscht alle nachlaufende Leerzeichen (o. angeg. Z.)

Abbildung 10: Die wichtigsten String-Methoden in C#

Die folgenden Abschnitte erklären die meisten der angeführten Methoden mehrerer Beispiele.

7.2.1 Substring und []

string Substring(int pos, int length);

Liefert den Teilstring ab Position pos und der Länge length des Strings zurück.

string Substring(int pos);

Liefert den Teilstring ab Position pos bis zum Ende des Strings zurück.

Spezialfall: Nur ein Zeichen wird benötigt. Es gibt zwei Möglichkeiten, Substring oder den []-Operator:

```
string s = "Schnitzel";
string t = s.Substring(2,1);      → t ist "h"
char c = s[2];                  → c ist 'h'
```

7.2.1.1 Durchlaufen eines Strings mit Substring

```
// Durchlaufen eines Strings mit Substring
string s = "Hallo!";

for (int i = 0; i < s.Length; i++)
{
    string t = s.Substring(i,1);
    int unicode = (int)char.Parse(t);
    Console.WriteLine(t + " = " + unicode);
}
```

7.2.1.2 Durchlaufen eines Strings mit []

```
// Durchlaufen eines Strings mit []
string s = "Hallo!";

for (int i = 0; i < s.Length; i++)
{
    char c = s[i];
    int unicode = (int)c;
    Console.WriteLine(t + " = " + unicode);
}
```

7.2.1.3 Durchlaufen eines Strings mit foreach

```
// Durchlaufen eines Strings mit []
string s = "Hallo!";

foreach (char c in s)
{
    int unicode = (int)c;
    Console.WriteLine(t + " = " + unicode);
}
```

Anbei ein Beispiel, das zeigt, wie Substring aus einem String Teile herausschneidet.

string s = "Schnitzel.....8,90 Euro";																						
S	c	h	n	i	t	z	e	l	8	,	9	0	.	E	u	r	o
0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	2	2	2

0 bedeutet 10 (ist aus Platzgründen übereinander)

Beispiele für Substring in C#

```
string s = "Schnitzel.....8,90 Euro";
Console.WriteLine(s.Substring(14,4));
Console.WriteLine(s.Substring(0,9));
Console.WriteLine(s.Substring(0,1));
Console.WriteLine(s.Substring(2,1));
Console.WriteLine(s.Substring(19));
```

Programmausgabe:

```
8,90
Schnitzel
S
H
Euro
```

7.2.2 Suchen in Strings

Grundsätzlich kann in Strings genauso wie in einem Array nach einzelnen Zeichen oder Strings gesucht werden. Strings unterstützen ja wie Arrays den []-Operator. Im Allgemeinen ist es häufig einfacher eine der zahlreichen Suchmethoden zu verwenden.

7.2.2.1 Contains

bool Contains(string r);

Liefert True falls der String r im String enthalten ist, sonst False.

Beispiel zu Contains in C#

```
string s = "Ich esse ein Schnitzel.";
bool sch = s.Contains("Schnitzel"); // liefert zurück ob Schnitzel in s enthalten ist
bool gul = s.Contains("Gulasch"); // liefert zurück ob Gulasch in s enthalten ist
Console.WriteLine(sch + " " + gul);
```

True False

7.2.2.2 StartsWith und EndsWith

bool StartsWith(string r);

Liefert True falls der String mit dem String r beginnt, sonst False.

bool EndsWith(string r);

Liefert True falls der String mit dem String r endet, sonst False.

Beispiel zu StartsWith und EndsWith in C#

```
string s = "Ich esse ein Schnitzel";
bool esse = s.StartsWith("Ich trinke"); // liefert zurück ob ich trinke
Console.WriteLine(esse);

if (s.StartsWith("Ich esse")) // liefert zurück ob ich esse
{
    Console.WriteLine("Ja, ich esse");
}
else
{
    Console.WriteLine("Nein, ich esse nicht.");
}

bool sch = s.EndsWith("Schnitzel");
Console.WriteLine(sch);
```

False

Ja, ich esse

True

7.2.2.3 IndexOf und LastIndexOf

int IndexOf(string r);

Liefert die Position des Strings r im String zurück. Falls r nicht enthalten ist, wird -1 zurückgegeben.

Die Suche beginnt von vorne im String.

```
int LastIndexOf( string r );
```

Liefert die Position des Strings r im String zurück. Falls r nicht enthalten ist, wird -1 zurückgegeben.

Die Suche beginnt von vorne im String.

Das Ergebnis ist 0-basiert, d. h. die Position des ersten Zeichens ist 0 (und nicht 1).

```
string s = "Hallo!";
```

Zeichen	'H'	'a'	'l'	'l'	'o'	!'
Position	0	1	2	3	4	5
IndexOf	startet bei 0 und sucht bis zum Ende des Strings → → → →					
LastIndexOf	← ← ← ← startet bei 5 und sucht bis zum Anfang des Strings					

IndexOf("al") -> 1

IndexOf('l') -> 2

LastIndexOf('l') -> 3

IndexOf("el") -> -1

Beispiel zu IndexOf in C#

```
string s = "Ich esse ein Schnitzel";
int pos = s.IndexOf("Schnitzel"); // liefert die Position von Schnitzel zurück, 13
Console.WriteLine(pos);
```

13

7.2.3 Stringoperationen

Grundsätzlich kann ein String in C# im Speicher NICHT verändert werden. Anders als in Arrays kann man nicht einfach mit dem []-Operator ein einzelnes Zeichen verändern. Die Zeichen in einem String sind schreibgeschützt.

```
class Program
{
    static void Main(string[] args)
    {
        string s = "Hallo!";
        s[1] = 'e';
    }
}
```

struct System.Int32
Stellt eine 32-Bit-Ganzzahl mit Vorzeichen dar.

Für die Eigenschaft oder den Indexer "string.this[int]" ist eine Zuweisung nicht möglich. Sie sind schreibgeschützt.

Wie die Abbildung zeigt, wird s[1] im Visual Studio rot unterstrichen, weil der []-Operator bei Strings keine Zuweisung erlaubt.

Strings werden im .NET-Framework nie verändert. Sie sind **immutable**. string ist der einzige Datentyp in C# der diese Eigenschaft hat.

Wie kann man nun Teile aus einem String entfernen oder Teile hinzufügen? Dies geschieht durch Methoden, die einen neuen String mit der gewünschten Änderung zurückgeben.

7.2.3.1 Insert

```
string Insert( int pos, string r );
```

Liefert einen String zurück, der an der Position pos den String r eingefügt hat.

```
string s = "Hallo!";  
s.Insert(0, "!") → "!Hallo!"  
s.Insert(5, "chen") → "Hallogen!"
```

Beispiel zu Insert in C#

```
string s = "Ich esse ein Schnitzel.";  
int pos = s.IndexOf("Schnitzel"); // liefert die Position von großes zurück, 13  
string g = s.Insert(pos, "großes "); // erzeugt einen neuen String  
Console.WriteLine(s);  
Console.WriteLine(g);
```

Ich esse ein Schnitzel.

Ich esse ein großes Schnitzel.

```
string s = "Ich esse ein Schnitzel.;"
```

I	c	h		e	s	s	e		e	i	n		S	c	h	n	i	t	z	e	l	.
0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	1	2	2

```
int pos = s.IndexOf("Schnitzel"); -> 13
```

```
string g = s.Insert(pos, "großes ");
```

I	c	h		e	s	s	e		e	i	n		g	r	o	ß	e	s		S	c	h	n	i	t	z	e	l	.
0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	

Wie man sieht, ist der String s völlig unverändert. Die Methode Insert legt einen völlig neuen String im Speicher an und liefert einen Verweis darauf zur Variablen g.

7.2.3.2 Remove

```
string Remove( int pos, int length );
```

Entfernt length-viele Zeichen ab Position pos.

```
string Remove( int pos );
```

Entfernt alle Zeichen ab Position pos.

```
string s = "ABCDEF";  
s.Remove(0, 2) → "CDEF"  
s.Remove(3, 1) → "ABCEF"  
s.Remove(1) → "A"
```

Beispiel zu Remove in C#

```
string s = "Ich esse ein kleines Schnitzel.";  
int pos = s.IndexOf("kleines"); // liefert die Position von kleines zurück, 13  
string g = s.Remove(pos, "kleines ".Length); // schneidet etwas aus dem String  
Console.WriteLine(s + "|" + g);  
Console.WriteLine(g);
```

Ich esse ein kleines Schnitzel.

Ich esse ein Schnitzel.

7.2.3.3 Replace

```
string Replace( string old, string new );
string Replace( char old, char new );
```

Liefert einen String zurück, in dem alle Vorkommnisse vom String/char old durch den String/char new ersetzt sind.

```
string s = "Hallo!";
s.Replace("a", "e") → "Hello!"
s.Replace('l', 's') → "Hasso!"
s.Replace("o!", "öchen Jungs") → "Hallöchen Jungs"
```

Beispiel zu Remove in C#

```
string s = "Ich esse ein großes Schnitzel.";
string g = s.Replace("großes", "kleines"); // erzeugt einen neuen String
Console.WriteLine(s);
Console.WriteLine(g);
Ich esse ein großes Schnitzel.
Ich esse ein kleines Schnitzel.
```

7.2.3.4 PadLeft und PadRight

```
string PadLeft( int length );
string PadLeft( int length, char fill );
string PadRight( int length );
string PadRight( int length, char fill );
```

Liefert einen String der Länge length in dem der String links oder rechts ausgerichtet ist.

Beispiel zu PadLeft und PadRight in C#

```
string s = "Hallo!";
string k = s.PadLeft(10);
string g = s.PadLeft(10, '.');
string h = s.PadRight(10, '.');
Console.WriteLine(k);
Console.WriteLine(g);
Console.WriteLine(h);
Hallo!
....Hallo!
Hallo!....
```

7.2.3.5 Strings bestimmter Länge aus einem Zeichen erzeugen

```
String( char c, int length )
```

Liefert einen String der aus einer bestimmten Anzahl eines bestimmten Zeichens besteht.

Diese Methode ist keine normale Funktion, sondern ein Konstruktor. Dieser Begriff wird erst in der Objektorientierten Programmierung erklärt. Daher muss man vorher das Schlüsselwort new verwenden.

Beispiel zum Erzeugen eines Strings mit einem bestimmten Zeichen und Länge

```
string m = new String('*', 5);
Console.WriteLine(m);
*****
```

7.3 Strings splitten

Manchmal ist es günstig einen Text anhand eines bestimmten Zeichens zu zerschneiden, also zu splitten.

Beispiel: Für einen beliebigen String s soll die Anzahl der Wörter bestimmt werden.

```
string s = "Ich esse ein Schnitzel";
```

I	c	h		e	s	s	e		e	i	n		S	c	h	n	i	t	z	e	l
0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	2	2

1. Möglichkeit: Zählen der Leerzeichen. Anzahl der Wörter = Anzahl + 1, aber nur wenn es keine doppelten Leerzeichen gibt.
2. Möglichkeit: Mit IndexOf von einem Leerzeichen zum nächsten springen.
3. Möglichkeit: Den String an allen Stellen zu zerschneiden, an denen ein Leerzeichen steht und dann zu zählen, wieviele Teile entstanden sind.

Dieses Zerschneiden eines Strings leistet die Methode Split.

```
string[] Split( char zeichen );
```

Zerteilt einen String an allen Stellen an denen sich ein bestimmtes Zeichen steht.

```
string s = "Ich esse ein Schnitzel.";
```

I	c	h		e	s	s	e		e	i	n		S	c	h	n	i	t	z	e	l	.
0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	2	2	2

```
string[] teile = s.Split(' ');
```

```
-> s[0] = "Ich"
   s[1] = "esse"
   s[2] = "ein"
   s[3] = "Schnitzel."
```

8 Zufallszahlen in C#

Zufallszahlen können in C# mit der Klasse Random berechnet werden. Zufallszahlen sind oft sehr praktisch, wenn man Spiele programmiert oder sich die Lottozahlen hervorsagen lassen möchte.

<https://docs.microsoft.com/de-de/dotnet/api/system.random?redirectedfrom=MSDN&view=netframework-4.7.2>

Ausgabe einer Zufallszahl in C#

```
static void Main(string[] args)
{
    Random rnd = new Random();
    int zufallszahl = rnd.Next();
    Console.WriteLine(zufallszahl);
}
```

Programmausgabe: (zufällig)

7829634

Die ersten beiden Programmzeilen haben folgende Bedeutung.

Random rnd = new Random();	Ein Zufallszahlengenerator wird angelegt. Und zwar einer der mit einem zufälligen Startwert initialisiert ist.
int zufallszahl = rnd.Next();	Eine Zufallszahl zwischen 0 und int.MaxValue wird berechnet, also zwischen 0 und 2147483647. https://docs.microsoft.com/de-de/dotnet/api/system.random.next?view=netframework-4.7.2

Um mehrere Zufallszahlen auszugeben, kann man Next() einfach öfters aufrufen.

Ausgabe mehrerer Zufallszahlen in C#

```
static void Main(string[] args)
{
    Random rnd = new Random();

    for (int i = 0; i < 8; i++)
    {
        int zufallszahl = rnd.Next();
        Console.WriteLine(zufallszahl);
    }
}
```

Programmausgabe: (zufällig)

1809529634

1947734409

1199532364

461322515

1091468936

1564094953

287213432

Um Zufallszahlen ab 0 und in einem bestimmten Bereich zu erhalten, kann man bei Next eine Obergrenze angeben.

```
Random rnd = new Random();
int zufallszahl1 = rnd.Next(5); // berechnet eine Zufallszahl von 0 bis 4 (nicht 5!!)
int zufallszahl2 = rnd.Next(10); // berechnet eine Zufallszahl von 0 bis 9 (nicht 10!!)
```



Der Höchstwert bei Next muss immer um 1 größer angegeben werden,
als die höchste Zufallszahl die man will!

Zusätzlich kann man auch Zufallszahlen in einem beliebigen Bereich ausgeben lassen, in dem man eine Untergrenze mitgibt.

```
Random rnd = new Random();
int zufallszahl1 = rnd.Next(1, 5); // berechnet eine Zufallszahl von 1 bis 4 (nicht 5!!)
int zufallszahl2 = rnd.Next(5, 10); // berechnet eine Zufallszahl von 5 bis 9 (nicht 10!!)
```

Ausgabe von 10 Zufallszahlen zwischen zwei Grenzen

```
static void Main(string[] args)
{
    Random rnd = new Random();

    Console.Write("Wie groß ist die kleinste Zufallszahl? ");
    int untergrenze = int.Parse(Console.ReadLine());

    Console.Write("Wie groß ist die größte Zufallszahl? ");
    int obergrenze = int.Parse(Console.ReadLine());

    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(rnd.Next(untergrenze, obergrenze + 1));
    }
}
```

Programmausgabe:

Wie groß ist die kleinste Zufallszahl? -2

Wie groß ist die größte Zufallszahl? 2

0

-2

0

2

2

-1

-2

2

-1

-2

1

-1

-2

0

-2

9 Zweidimensionale Arrays in C#

Eindimensionale Arrays sollten hinlänglich bekannt sein.

- Arrays sind Speicherblöcke, die Variablen von einfachen Datentypen, Instanzen oder andere Arrays abspeichern können.
- Arrays sind Instanzen von Klassen, obwohl sie keine explizite Klassendefinition haben.
- Auf einzelne Elemente wird mit dem []-Operator zugegriffen.

9.1 "Rechteckige" zweidimensionale Arrays in C#

Ein solches 2d-Array kann man sich als Rechteck vorstellen, bei dem jedes Element in einer bestimmten Reihe und in einer bestimmten Spalte steht.

Alle Zeilen und alle Spalten werden mit 0 beginnend durchnummeriert. Die jeweilige Nummer bezeichnet man als Index.

Spaltenindex				
Zeilenindex	0,0	0,1	0,2	0,3
	1,0	1,1	1,2	1,3
	2,0	2,1	2,2	2,3

Abbildung 11: Ein 3x4-Array in C#. Jedes Element hat einen Zeilenindex und einen Spaltenindex.

9.1.1 Deklaration

Ein 2d-Array wird ähnlich wie ein 1d-Array deklariert, allerdings ist bei der Deklaration die Anzahl der Zeilen und die Anzahl der Spalten anzugeben.

```
// Ein Array mit 3 Zeilen und 4 Spalten.  
int[,] num = new int[3, 4];
```

9.1.2 Zugriff auf Elemente

Auf Elemente wird wie bei einem 1d-Array durch Angabe der Indices und des []-Operators zugegriffen.

```
// Das Element in der Zeile 1 und der Spalte 2 wird auf 4 gesetzt und dann ausgegeben.  
num[1, 2] = 4;  
Console.WriteLine(num[1, 2]);
```

Auch bei 1d-Arrays werden sehr häufig Schleifen for-Schleifen verwendet, weil man mit den Schleifenvariablen in [] auf einzelne Elemente zugreifen kann.

Das folgende Programm zeigt wie man mehrere Werte setzt im Array setzt und wie man Schleifenvariablen beim Zugriff auf Arrayelemente einsetzt.

Hinweis: Die Werte 3 (Anzahl der Zeilen) und 4 (Anzahl der Spalten) werden nur bei der Deklaration angegeben!

Im Programm ist die Verwendung unbedingt zu vermeiden, sondern stattdessen können die beiden Methoden

- `GetLength(int dimension)`
- `GetUpperBound(int dimension)`

verwendet werden.

C# Zugriff auf die einzelnen Arrayelemente und Ermittlung der Anzahl von Zeilen und Spalten

```
static void Main(string[] args)
{
    // Ein Array mit 3 Zeilen und 4 Spalten.
    int[,] num = new int[3, 4];

    num[1, 0] = 5;
    num[1, 1] = 3;
    num[1, 2] = 4;
    num[1, 3] = 6;

    Console.WriteLine($"{num.GetLength(0)} Zeilen und {num.GetLength(1)} Spalten");
    Console.WriteLine($"Der Zeilenindex läuft von 0 bis {num.GetUpperBound(0)}.");
    Console.WriteLine($"Der Spaltenindex läuft von 0 bis {num.GetUpperBound(1)}.");

    Console.WriteLine("Zeile 1:");

    for (int i = 0; i < num.GetLength(1); i++)
    {
        Console.WriteLine(num[1, i]);
    }
}
```

Programmausgabe:

3 Zeilen und 4 Spalten

Der Zeilenindex läuft von 0 bis 2.

Der Spaltenindex läuft von 0 bis 3.

Zeile 1:

5

3

4

6

Das folgende Übungsbeispiel zeigt einige wichtige Punkte:

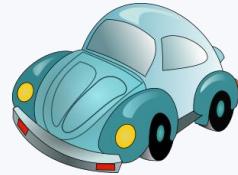
1. Die Initialisierung von 2d-Arrays bei der Deklaration durch die Angabe der Werte in {} (siehe den Beginn des Programms).
2. Zwei verschachtelte for-Schleifen um alle Elemente eines Arrays zu durchlaufen (4. Aufgabe).

Beispiel: Verkaufszahlen von Autos in einem 2d-Array

Der Autohändler "Schnell & Hübsch" verkauft Autos der Marken VW, KIA und Dacia.

Im Jahr 2018 wurden von jeder Marke folgende Fahrzeugtypen verkauft:

	Kleinstwagen	Limousine	SUV	Transporter
VW	30	53	43	8
KIA	12	48	5	9
Dacia	32	65	3	1



Erstelle ein Programm, das diese Werte in einem zweidimensionalen Array direkt speichert (also *hard coded*) und Auswertungen erlaubt.

1. Ausgabe der Anzahl aller verkauften SUV der Marke KIA aus (durch Zugriff auf das Array!)
2. Ausgabe der Anzahl aller verkauften SUV und Transporter der Marke Dacia (durch Zugriff auf das Array!)
3. Nach der Eingabe einer Marke, soll die Gesamtanzahl aller verkauften Autos dieser Marke ausgegeben werden.
z. B. Als Marke wird "VW" eingegeben. Ausgabe: 134 wurden verkauft.
4. Gib den Verkaufsanteil dieser Marke in % an der Gesamtanzahl aller verkauften Autos aus.

Lösung:

```
static void Main(string[] args)
{
    int[,] sales =
    {
        { 30, 53, 43, 8 },
        { 12, 48, 5, 9 },
        { 32, 65, 3, 1 }
    };

    // 1. Ausgabe der Anzahl aller verkauften SUV der Marke KIA aus
    Console.WriteLine($"SUV von KIA wurden { sales[1, 2] } verkauft.");

    // 2. Ausgabe der Summe aller verkauften SUV und Transporter der Marke Dacia
    Console.WriteLine($"Es wurden { sales[2,2]+ sales[2,3] } SUV und Transporter verkauft.");

    // 3. Gesamtanzahl einer Automarke
    int z = 0;
    Console.WriteLine("Welche Marke soll ausgewertet werden? ");
    switch (Console.ReadLine())
    {
        case "VW": z = 0; break;
        case "KIA": z = 1; break;
        case "Dacia": z = 2; break;
        default: Console.WriteLine("Falsche Eingabe"); z = -1; break;
    }

    int summe = 0;

    for (int spalte = 0; spalte <= verkäufe.GetUpperBound(1); spalte++)
    {
        summe += verkäufe[z, spalte];
    }

    Console.WriteLine($"Verkaufte Anzahl: {summe}");
}
```

```

// 4. Verkaufsanteil der Automarke
int gesamtanzahl = 0; // Summation aller Elemente des 2d-Arrays -> Geschachtelte Schleife
for (int zeile = 0; zeile <= sales.GetUpperBound(0); zeile++)
{
    for (int spalte = 0; spalte <= sales.GetUpperBound(1); spalte++)
    {
        gesamtanzahl += sales[zeile, spalte];
    }
}
Console.WriteLine($"Gesamtanteil: {Math.Round((double)summe/gesamtanzahl*100, 2)} %");
}

```

9.2 "Jagged" Arrays

Es ist auch möglich – analog zu Java – zweidimensionale Arrays in C# als eindimensionale Arrays von eindimensionalen Arrays zu definieren. Daher kann jede Zeile unterschiedlich lang sein. Man spricht dann von sogenannten jagged Arrays (=ausgefrazte Arrays).

Im folgenden Beispiel wird ein solches 2d-jagged-Array definiert, bei dem jede Zeile eine andere Länge hat.

Beachte, dass der Zugriff [][] statt [,] ist.

C#

```

static void Main(string[] args)
{
    // jagged array
    string[][] board =
    {
        new string[] {"Max"},
        new string[] {"Hugo", "Paula"},
        new string[] {"Karin", "John", "Mark"}
    };

    Console.WriteLine(board[0][0]);
    Console.WriteLine(board[1][0]);
    Console.WriteLine(board[2][1]);
    Console.WriteLine(board[2][2]);
}

```

Programmausgabe:

Max
Hugo
John
Mark

Row\Column	0	1	2
0	true		
1	false	true	
2	false	false	true

Die Seite <https://www.geeksforgeeks.org/c-sharp-jagged-arrays/> bietet einen guten Überblick.

10 File-IO

File-IO betrifft alle Aktionen, die das Filesystem betreffen, in erster Linie das Lesen und Schreiben von Files.

Es gibt verschiedene Filearten. Dieses Kapitel beschränkt sich auf

1. Textfiles, z. B. CSV-Files
2. Binärfils, z. B. Files mit Bildern.

10.1 Textfiles

Textfiles sind Files die im Allgemeinen für den Menschen lesbar sind und aus Unicode-Zeichen aufgebaut sind.

Ein Unicode-Zeichen ist in C# im Datentyp `char` repräsentiert, viele Unicode-Zeichen daher im Datentyp `string`.

Beim Arbeiten mit Textfiles hat man es also fast ausschließlich mit Strings zu tun.

10.1.1 Textfile einlesen

Um Textfiles einzulesen, benötigt man die Klasse `StreamReader`. Die folgende Tabelle zeigt den Vergleich zum Einlesen eines Strings von der Konsole.

Einlesen eines Strings von der Konsole	Einlesen eines Strings aus einem File in.txt
<pre>string zeile = Console.ReadLine();</pre>	<pre>var reader = new StreamReader("in.txt"); string zeile = reader.ReadLine(); ... reader.Close();</pre>

Im Gegensatz zum Einlesen von der Konsole, muss beim Lesen aus einem File natürlich auch angegeben werden, wie das File heißt, dass eingelesen werden soll. Konsole gibt es schließlich nur eine. Die `Console`-Klasse ist daher statisch.

10.1.1.1 Wichtige Methoden und Properties

Methode/Property	Beschreibung
<code>new StreamReader(file)</code>	Konstruktor der ein File neu anlegt. Falls ein File mit diesem Namen bereits existiert, wird es gelöscht.
<code>string ReadLine()</code>	Einlesen der nächsten Zeile als String
<code>string ReadToEnd()</code>	Liest ab der momentanen Position bis zum Ende das File ein und gibt es als String zurück.
<code>bool EndOfStream</code>	Ist ein bool-Property, das angibt, ob bereits das letzte Zeichen des Files gelesen wurde oder noch nicht.
<code>void Close()</code>	Schließt einen offenen StreamReader.
<code>void Flush()</code>	Schreibt den internen Buffer eines StreamReaders in das File.

10.1.1.2 Beispiel: Einlesen eines CSV-Files

CSV-Files sind Textfiles bei denen jede Zeile einen Datensatz enthält und die einzelnen Teile durch ein bestimmtes Separationszeichen (Separator oder Delimiter) voneinander getrennt sind.

CSV-Files können optional als erste Zeile eine Kopfzeile besitzen, in der steht für den Benutzer steht, welche Daten in den einzelnen Zeilen zu finden sind.

In diesem Beispiel ist das CSV-File Autos.txt gegeben in dem Daten von Autos gespeichert sind.

```
Kennzeichen;Beschreibung;Preis;Treibstoff (Benzin/Diesel);Verbrauch in 1/100km  
W-34546A;BMW Z1;35400;B;7,6  
L-6778BC;Fiat 124 Spider;23990;B;8,5  
MI-17478LI;Opel Zafira Edition;32750;D;6,5  
WU-MAUSI1;VW Passat 1.4;32000;B;5,5  
W-99356K;Ford Mondeo Trend;30100;D;4,9  
OW-7305ZR;Skoda Octavia;30300;D;4,8
```

Lösung:

Da jede Zeile für ein Auto steht wird zunächst eine Klasse angelegt, die ein Auto repräsentiert.

C# Klasse für ein Auto mit einem Aufzählungstyp für die Treibstoffart

```
enum Treibstoff  
{  
    Benzin, Diesel  
}  
  
class Auto  
{  
    public string Kennzeichen { get; private set; }  
    public string Bezeichnung { get; private set; }  
    public double Neupreis { get; private set; }  
    public Treibstoff Treibstoff { get; private set; }  
    public double VerbrauchAuf100Km { get; private set; }  
  
    public Auto(string kennzeichen, string bezeichnung, double neupreis,  
               Treibstoff treibstoff, double verbrauchAuf100Km)  
    {  
        Kennzeichen = kennzeichen;  
        Bezeichnung = bezeichnung;  
        Neupreis = neupreis;  
        Treibstoff = treibstoff;  
        VerbrauchAuf100Km = verbrauchAuf100Km;  
    }  
  
    public override string ToString() =>  
        $"'{Bezeichnung,-20}: {Kennzeichen,-10} {Neupreis,12:C2} ({Treibstoff})";  
}
```

Spaltenbreite ist 20
-20 bedeutet linksbündig

Jetzt wird im Main eine Liste von Autos angelegt, dann das File Autos.txt mit einem StreamReader geöffnet und Zeile für Zeile ausgelesen.

Jede Zeile wird mit `String.Split` in einzelne Teile zerlegt. Jeder Teil wird dann je nach dem Datentyp mit einer der Parse-Methoden umgewandelt und dann dem Konstruktor der Auto-Klasse übergeben.

Anschließend werden alle Instanzen in der Liste gespeichert.

C# Einlesen eines Textfiles und Ausgabe aller eingelesenen Instanzen

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Globalization;
using System.Threading;
using System.Text;

namespace DateiEinlesen
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread.CurrentThread.CurrentCulture = new CultureInfo("de-at");
            Console.OutputEncoding = Encoding.Unicode;

            List<Auto> autos = new List<Auto>(50);

            using (StreamReader reader = new StreamReader("Autos.txt"))
            {
                reader.ReadLine(); // überliest die erste Zeile mit den Beschreibungen

                while (!reader.EndOfStream)
                {
                    // z. B. W-34546A;BMW Z1;35400;B;7,6
                    string zeile = reader.ReadLine();
                    string[] teile = zeile.Split(';');

                    Treibstoff tr = teile[3][0] == 'B' ? Treibstoff.Benzin
                                              : Treibstoff.Diesel;

                    Auto auto = new Auto(
                        teile[0], // Kennzeichen, z. B. "W-34546A"
                        teile[1], // Bezeichnung, z. B. "BMW Z1"
                        double.Parse(teile[2]), // Neupreis, z. B. 35400
                        tr, // Erstes Zeichen des Treibstoffs, z. B. 'B'
                        double.Parse(teile[4])); // Verbrauch, z. B. 7,6 Liter/100km
                    autos.Add(auto);
                }
            }

            // Ausgabe aller Instanzen
            foreach (Auto auto in autos)
            {
                Console.WriteLine(auto);
            }
        }
    }
}
```

Programmausgabe:

BMW Z1	:	W-34546A	€ 35.400,00	(Benzin)
Fiat 124 Spider	:	L-6778BC	€ 23.990,00	(Benzin)
Opel Zafira Edition	:	MI-17478LI	€ 32.750,00	(Diesel)
VW Passat 1.4	:	WU-MAUSI1	€ 32.000,00	(Benzin)
Ford Mondeo Trend	:	W-99356K	€ 30.100,00	(Diesel)
Skoda Octavia	:	OW-7305ZR	€ 30.300,00	(Diesel)

10.1.2 Textfiles schreiben

Um Daten in Textfiles zu schreiben, benötigt man die Klasse `StreamWriter`. Die folgende Tabelle zeigt den Vergleich zur Ausgabe eines Strings auf die Konsole.

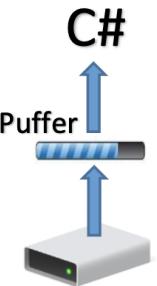
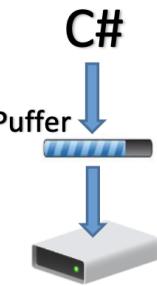
Ausgabe eines Strings auf die Konsole	Ausgabe eines Strings in ein File out.txt
<pre>string zeile = "Hello"; Console.WriteLine(zeile);</pre>	<pre>string zeile = "Hello"; var writer = new StreamWriter("out.txt"); writer.WriteLine(zeile); ... writer.Close();</pre>

Der Aufruf von `Close()` ist wichtig, da sonst das File nicht geschlossen wird und die Daten möglicherweise nicht geflushet werden.

10.1.3 Rezepte für CSV- und Spaltenorientierte Files

Die folgende Aufstellung zeigt Schritt für Schritt das Einlesen aus CSV-Files, sowie die analogen Schritte für spaltenorientierte Textfiles, in denen die Daten untereinanderstehen und die Daten mit `SubString` herausgeschnitten werden müssen.

Bauhausartikel als CSV- und in einem spaltenorientiertem Format			
CSV-Format	Spaltenorientiertes Format		
Kupfernägel 35 Stk;5,11;30	Kupfernägel	35	Stk 5,11 30
Bilderleiste;10,27;8	Bilderleiste		10,27 8
Maurerkelle;19,00;12	Maurerkelle		19,00 12
Scheibtruhe;75,60;2	Scheibtruhe		75,60 2
Senklot;9,23;8	Senklot		9,23 8
Pinsel;2,50;7	Pinsel		2,50 7
Abziehlatte;9,99;5	Abziehlatte		9,99 5
Zement 25 kg;16,49;30	Zement 25 kg		16,49 30
Fugenbunt 5 kg;23,99;15	Fugenbunt 5 kg		23,99 15
Mischmaschine;189;4	Mischmaschine		189 4
Mäh-Roboter;1699;2	Mäh-Roboter		1699 2
Rasensamen 1 kg;5,99;20	Rasensamen 1 kg		5,99 20
Dispersionsfarbe;24,99;25	Dispersionsfarbe		24,99 25

<h2>Dateien</h2>	<h3>CSV (comma separated values)</h3> <p>W-34546A;35.400;BMW Z1;B;7,6 L-6778BC;23.990;Fiat 124 Spider;B;8,5</p>	<h3>Spaltenorientiert (fixed length format)</h3> <p>W-34546A 35.400 BMW Z1 B 7,6 L-6778BC 23.990 Fiat 124 Spider B 8,5 012345678901234567890123456789012345678901 (Raster)</p>
 <h2>Lesen</h2>	<h3>C#</h3> <ol style="list-style-type: none"> 1. StreamReader erzeugen mit StreamReader reader = new StreamReader(<Dateipfad>); 2. Abfragen ob es (noch) Zeichen gibt, die gelesen werden können, z. B. while(!reader.EndOfStream) { <Anweisungen>; ... } 3. Eines oder mehrere Zeichen lesen: z. B. eine ganze Zeile auf einmal einlesen: string zeile = reader.ReadLine(); 4. Eine Zeile in einzelne Teile zerlegen mit der String-Methode Split und dem Trennzeichen (Separator, Delimiter), z. B. string[] teile = zeile.Split(';', StringSplitOptions.RemoveEmptyEntries); 5. Konvertieren der einzelnen Teilstings in Werte des gewünschten Datentyps: double preis = double.Parse(teile[1]) bool benzin = teile[3] == "B"; 6. Weiterverwendung der Daten, z. B. Abspeichern in einem Array, Ausgeben, ... AUFGABENABHÄNGIG! 7. Lesen beenden: Wenn alle gewünschten Daten eingelesen wurden, wird die Verwendung der Datei abgeschlossen reader.Close(); 	<ol style="list-style-type: none"> 1. den String-Methoden SubString, TrimStart, TrimEnd, Trim zum "Ausschneiden" der gewünschten Zeichen, z. B. string kennz = String.Substring(zeile, 0, 12).TrimEnd(); string pr = String.Substring(zeile, 12, 8).Trim(); string b = String.Substring(zeile, 37, 1);
 <h2>Schreiben</h2>	<h3>C#</h3> <ol style="list-style-type: none"> 1. StreamWriter erzeugen mit StreamWriter writer = new StreamWriter(<Dateipfad>); // neues File Anhängen von Daten an eine existierende Datei durch: StreamWriter erzeugen mit StreamWriter writer = new StreamWriter(<Dateipfad>, true); 2. Berechnen oder ermitteln der Daten die in die Datei geschrieben werden sollen: AUFGABENABHÄNGIG! z. B. string name; double preis; name = <Anweisungen>; preis = <Anweisungen>; 3. Werte in die Datei schreiben Entweder zeilenweise writer.WriteLine(name+";"+preis+";"); oder alle Werte nacheinander writer.Write(name+";"); writer.Write(preis+";"); writer.WriteLine(); 4. Schreiben beenden: Wenn alle gewünschten Daten geschrieben wurden, wird die Verwendung der Datei abgeschlossen, d. h. der Puffer wird geflushed (Flush()) und dem Betriebssystem wird mitgeteilt, dass die Datei nicht mehr gebraucht wird. writer.Close(); 	<ol style="list-style-type: none"> 3. Werte in die Datei schreiben writer.WriteLine(name.PadRight(40)); // 40 Zeichen writer.WriteLine(preis.ToString().PadRight(10)); // 10 chars writer.WriteLine(); // Neue Zeile beginnen Bei numerischen, boolschen, ... Werten vor dem PadRight zuerst ToString(). Beim letzten Wert einer Zeile ist PadRight nicht mehr notwendig.

10.2 Binärfiles

Binärdaten aus Files werden im allgemeinem byteweise eingelesen. Ein Byte in C# hat den Wertebereich von 0 bis 255.

Im folgenden Programm wird ein Binärfile input.bin byteweise eingelesen. Es gibt neben der Methode

```
byte ReadByte()
```

auch die Methode

```
byte[] ReadBytes(int count)
```

die gleich ein Array einer gewünschten Größe zurückgibt.

C# Einlesen eines Binärfiles

```
string filepath = "input.bin";

BinaryReader rdr = new BinaryReader(
    File.Open(filepath, FileMode.Open) );

while( rdr.BaseStream.Position != rdr.BaseStream.Length ) // Endeüberprüfung
{
    byte b = rdr.ReadByte(); // single byte
    ...
    // read ten bytes at once
    byte[] bytes = rdr.ReadBytes(10);
    ...
}

rdr.Close();
```

11 Commandline Parameter in C#

Commandline Parameter werden über das String-Array in der Main-Methode übergeben.

Beispiel: Beim folgenden Demoprogramm wird die Größe des Arrays überprüft und dann alle Strings aus dem String-Array args ausgegeben.

C# Commandline Demoprogramm

```
using System;

namespace CommandLineParameter
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine($"Du hast {args.Length} Parameter angegeben.");

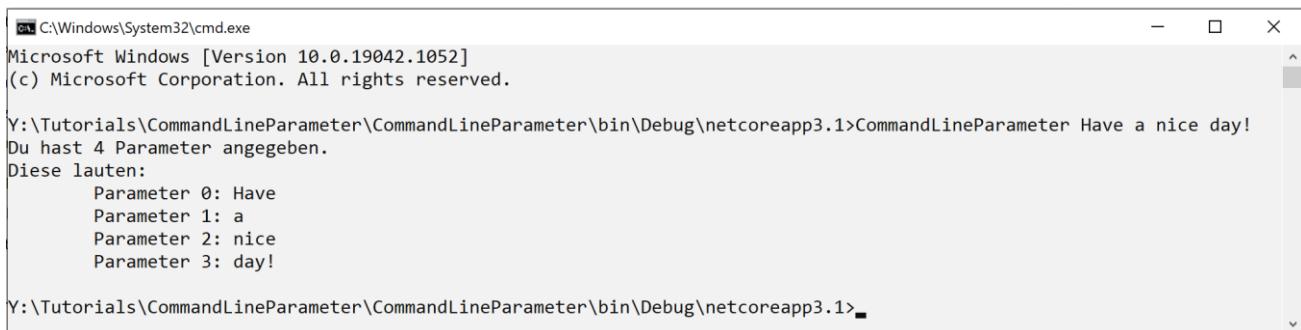
            if (args.Length == 1)
                Console.WriteLine("Dieser lautet:");
            else if (args.Length > 1)
                Console.WriteLine("Diese lauten:");

            for (int i = 0; i < args.Length; i++)
            {
                Console.WriteLine($"Parameter {i}: {args[i]}");
            }
        }
    }
}
```

11.1 Übergabe der Parameter auf der Windowskonsole

Wenn man eine Windowskonsole im Verzeichnis des Executables (.exe-File) öffnet, kann man das Programm starten und dort Parameter übergeben.

Die Ausgabe des oberen Demoprogramms sieht auf der Windowskonsole z. B. so aus:



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The window displays the following output:

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19042.1052]
(c) Microsoft Corporation. All rights reserved.

Y:\Tutorials\CommandLineParameter\CommandLineParameter\bin\Debug\netcoreapp3.1>CommandLineParameter Have a nice day!
Du hast 4 Parameter angegeben.
Diese lauten:
    Parameter 0: Have
    Parameter 1: a
    Parameter 2: nice
    Parameter 3: day!

Y:\Tutorials\CommandLineParameter\CommandLineParameter\bin\Debug\netcoreapp3.1>
```

Abbildung 12: Start des CommandLineParameters Demoprogramms auf der Windowskonsole.

11.2 Festlegung in Visual Studio

In Visual Studio müssen die Properties der Solution geöffnet werden. Hier heißt die Solution *CommandLineParameters*.

Dann sind unter Debug in den *Application arguments* alle Parameterwerte zu übergeben.

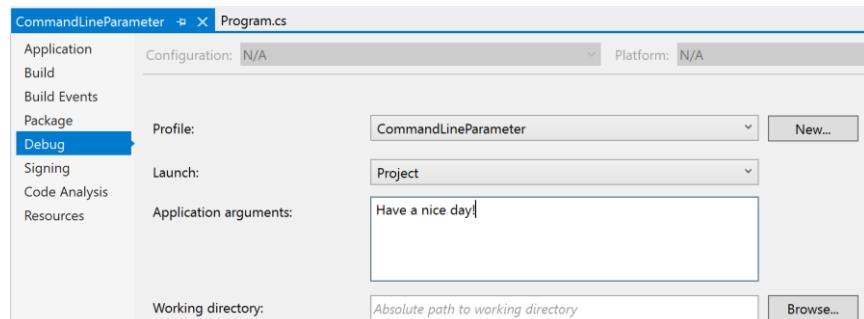


Abbildung 13: Eingabe von Commandline Parametern in den Projekt Properties in Visual Studio

d, werden die eingegebenen Werte als String-Array an das Main übergeben.



Abbildung 14: Start eines Programms in der Debug-Konfiguration

Die Ausgabe erfolgt dann im Konsolenfenster.

A screenshot of the Visual Studio 'Output' window. It displays the following text:

```
Microsoft Visual Studio... — □ X
Du hast 4 Parameter angegeben.
Diese lauten:
    Parameter 0: Have
    Parameter 1: a
    Parameter 2: nice
    Parameter 3: day!
```

The window has a dark background and light-colored text.

12 Objektorientierung

C# ist eine vollkommen objektorientierte Programmiersprache.

Objektorientierung bedeutet, dass beim Ablauf eines Programms, verschiedene Objekte miteinander zusammenarbeiten und Daten austauschen.

Der Begriff Objekt ist sehr unscharf! Ein Objekt in einer Programmiersprache kann gegenständlich sein, z. B. ein Auto, eine Person oder ein Gegenstand oder aber auch nicht gegenständlich, z. B. Konverter, Sortierer, usw.

Ein Objekt ist im Allgemeinen dadurch definiert, dass es einen Zustand hat und ein Verhalten.

Jedes Objekt hat:	Zustand	Verhalten
In Programmiersprachen sind das:	Variablen	Programmcode
Übliche Bezeichnungen in Programmiersprachen:	<ul style="list-style-type: none">EigenschaftenAttributeFelderInstanzvariablenPropertiesData Members	<ul style="list-style-type: none">MethodenOperationenFunction Members

Der Zustand eines Objekts lässt sich in zwei Gruppen aufteilen:	
Stammdaten ...	Bewegungsdaten ...
... ändern sich während dem Programmablauf ...	
... nie oder nur sehr selten.	... häufig.
Beispiele bei einem Auto: <ul style="list-style-type: none">Bezeichnung "BMW Z1" (ändert sich nie)Autokennzeichen W-22HTLDS (ändert sich nur bei einem Fahrzeugverkauf)Höchstgeschwindigkeit 220 km/h oder der Spritverbrauch in Liter auf 100 km (ändert sich sehr selten, nur bei Änderungen am Motor)	Beispiele bei einem Auto: <ul style="list-style-type: none">Kilometerstand in kmMomentangeschwindigkeit in km/hFüllmenge im Tank in Liter

12.1 Ein einführendes Beispiel

Folgendes Beispiel soll die Unterscheidung von Daten (Stammdaten und Bewegungsdaten) und das Verhalten eines Objekts verdeutlichen.

Beispiel: Gegeben sind zwei Autos mit ihren Daten.

Ein Auto ist ein BMW Z1, hat 170 PS und beschleunigt von 0 auf 100 km/h in 7,9 Sekunden. Der BMW ist rot lackiert und hat das Autokennzeichen W-22HTLDS. Die Höchstgeschwindigkeit beträgt 220 km/h.



Das zweite Auto ist ein VW Passat mit 122 PS. Er beschleunigt von 0 auf 100 km/h in 8,9 Sekunden und ist blau. Sein Autokennzeichen ist PL-1337HOT. Höchstgeschwindigkeit beträgt 210 km/h.



Jedes Fahrzeug kann starten, beschleunigen, abbremsen und den Radio ein- und ausschalten.

Bestimme welche Eigenschaften und welche Aktionen jedes Auto hat.

Lösung: Die Stammdaten sind in der Angabe oft explizit herauslesbar, die Bewegungsdaten ergeben sich dagegen erst aus Überlegungen des geforderten Verhaltens.

Stammdaten			
Eigenschaften: Stammdaten = Daten die nie oder selten geändert werden	<i>Bezeichnung</i> als Text	BMW Z1	VW Passat
	<i>Leistung</i> als Ganzzahl	170	122
	<i>Zeit von 0 auf 100 km/h</i> als Dezimalzahl	7,9	8,9
	<i>Farbe</i>	Rot	Blau
	<i>Höchstgeschwindigkeit</i> als Dezimalzahl	220	210
	<i>Autokennzeichen</i> als Text	W-22HTL	PL-1337HOT
Aktionen (Verhalten):	<ul style="list-style-type: none"> • Starten • Beschleunigen (Geschwindigkeit erhöhen) • Abbremsen (Geschwindigkeit reduzieren) • Radio einschalten • Radio ausschalten 		

Bewegungsdaten	Eigenschaft die sich daraus ergibt:
Aktionen (Verhalten):	
1. Starten	<i>Boolische Variable IstMotorGestartet</i> , ob das Auto gestartet ist. True=Das Auto ist bereits gestartet.
2. Beschleunigen	<i>Momentangeschwindigkeit</i> in km/h als Dezimalzahl
3. Abbremsen	keine weitere
4. Radio einschalten	<i>Boolische Variable IstRadioEingeschalten</i> , ob das Radio läuft.
5. Radio ausschalten	keine weitere

In Wahrheit hat ein Auto natürlich noch viel mehr Eigenschaften (Kraftstoff, Gangschaltung, ...) und kann auch weit mehr Aktionen durchführen (Lenken, Auftanken, ...). Je nach der Aufgabe bzw. den Wünschen des Kunden werden aber nur jene Eigenschaften und Aktionen im Programm angebildet, die relevant sind. Die Realität wird immer so eingeschränkt, soweit es nötig ist.

Beispielsweise kann ein Auto auch einen Unfall haben. Falls dieses Verhalten für das Programm nicht notwendig ist, werden einfach keine Eigenschaft *HatteEinenUnfall* und keine Aktion *Verunfallen* vorgesehen.

12.2 Eine Klasse als Bauplan für Objekte

Im oberen Beispiel ist ersichtlich, dass beide Autos dieselben Eigenschaften haben, die unterschiedliche Werte annehmen können. Beide Autos haben ein Autokennzeichen, das aber unterschiedlich ist.

Ebenso haben beide Autos dieselben Methoden für ihr Verhalten.

Um beide Objekte mit dem gleichen Programmcode abbilden zu können, ist es notwendig eine Klasse zu definieren, die die Gemeinsamkeiten als einen Bauplan zur Verfügung stellt.

Eine Klasse...	Ein Objekt (Eine Instanz)...
... wird vom Entwickler programmiert.	.. wird zur Laufzeit aus dem Programmcode der Klasse erstellt.
In C# mit dem <code>class</code> Schlüsselwort. <pre>class Auto { ... }</pre> <p><code>Auto</code> ist eine Klasse.</p>	In C# mit dem <code>new</code> Schlüsselwort. <pre>Auto bmw = new Auto(); Auto vw = new Auto();</pre> <p><code>bmw</code> und <code>vw</code> sind Objekte der Klasse <code>Auto</code>.</p>
Eine Klasse existiert nur als Programmcode!	Objekte existieren während des Programmablaufs und können dort verändert werden!

Zur Laufzeit wird aus einer Klasse ein konkretes Objekt, das im weiteren Programmablauf beliebig verändert und genutzt werden kann.

12.3 Bestandteile einer Klasse

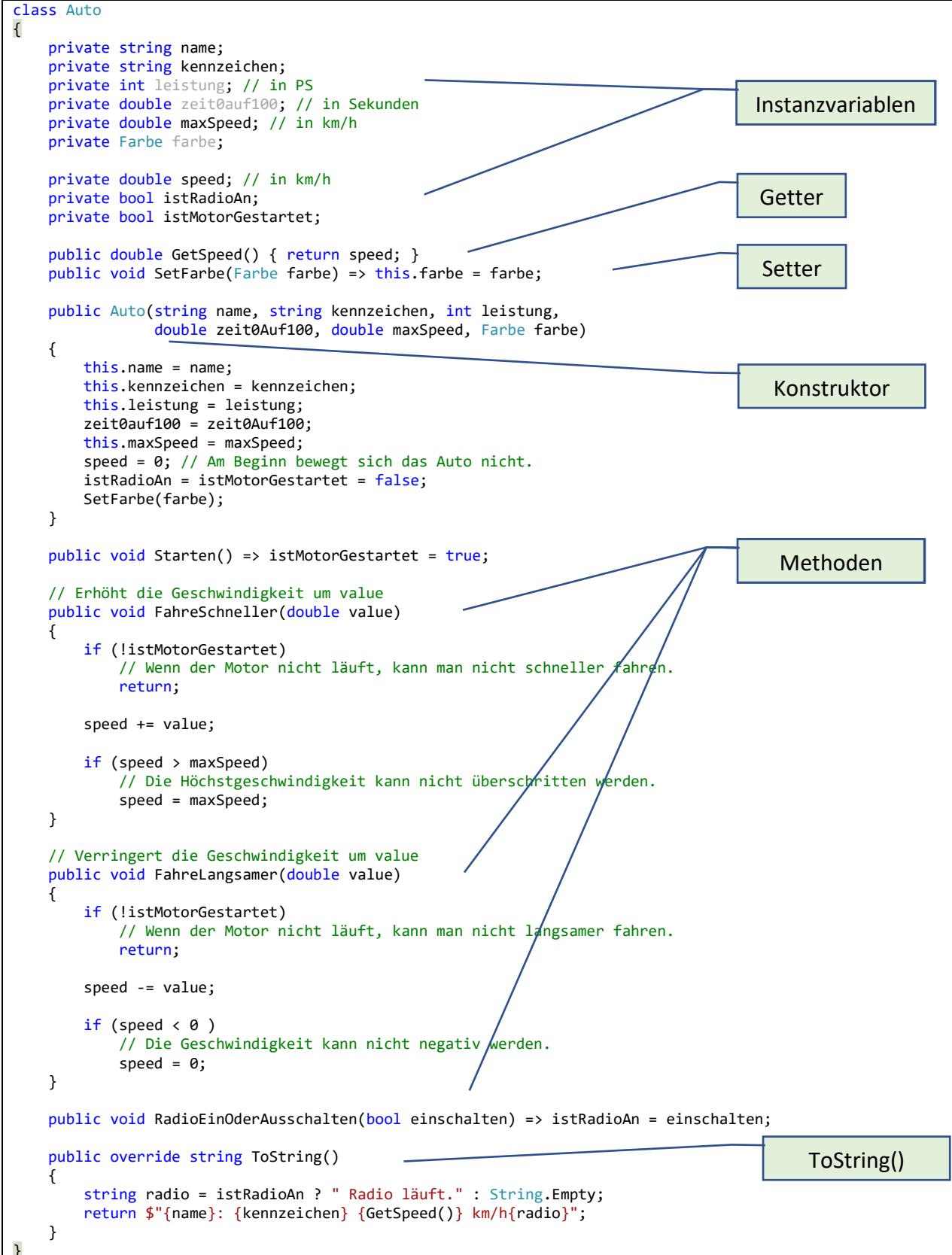
Klassen bestehen in C# neben den Instanzvariablen (Zustand) und den Methoden (Verhalten) noch aus weiteren Teilen.

Bestandteil der Klasse	Beschreibung
Instanzvariablen	Sind Variablen die den Zustand darstellen.
Konstruktor (können mehrere seien)	Ist die Methode, die beim Anlegen eines Objekts mit <code>new</code> aufgerufen wird.
<code>string</code> <code>ToString()</code>	Ist eine Methode, die Instanzvariablen als String formatiert zurückgibt. Ist sehr hilfreich bei <code>Console.WriteLine</code> .
Getter	Sind Methoden, die den Wert von Instanzvariablen zurückgeben.
Setter	Sind Methoden mit denen Instanzvariablen eines Objekts auf bestimmte Werte gesetzt werden können.
Methoden	Stellen das Verhalten da.

C# hat einige Spezialitäten, wie z. B. Properties, Enums, Expression Bodied Members, die nicht alle auf einmal erklärt werden können.

Exemplarisch wird die Klassendefinition für unser Beispiel gezeigt, um einen ersten Eindruck zu geben.

C# Klasse Auto mit einem Aufzählungstyp (Enum) für die Farbe
<pre>enum Farbe { Rot, Blau }</pre>



Um unsere beiden Autos während des Programmablaufs "zum Leben zu erwecken", wird für jedes mit `new` ein Objekt angelegt und dann die diversen Methoden für jedes Objekt aufgerufen.

C# Main das Objekte der beiden Autos erzeugt und einige Aktionen durchführt

```
static void Main(string[] args)
{
    Auto auto1 = new Auto("BMW Z1", "W-22HTLDS", 170, 7.9, 220, Farbe.Rot);

    Auto auto2 = new Auto("VW Passat", "PL-1337HOT", 122, 8.9, 210, Farbe.Blau);

    auto1.Starten();
    auto1.FahreSchneller(30);
    Console.WriteLine(auto1); // BMW fährt 30 km/h

    auto2.Starten();
    auto2.RadioEinOderAusschalten(true);
    Console.WriteLine(auto2); // VW steht noch und Radio läuft

    auto1.FahreSchneller(50); // BMW erhöht um 50 km/h auf 80 km/h
    Console.WriteLine(auto1);

    auto2.FahreSchneller(100); // VW beschleunigt um 100 km/h
    auto2.RadioEinOderAusschalten(false); // VW schaltet den Radio ab
    Console.WriteLine(auto2);

    auto1.FahreLangsamer(30); // BMW bremst um 30 km/h auf 50 km/h herunter
    Console.WriteLine(auto1);

    auto2.FahreLangsamer(20); // VW bremst um 20 km/h auf 80 km/h ab
    Console.WriteLine(auto2);
}
```

Programmausgabe:

```
BMW Z1: W-22HTLDS 30 km/h
VW Passat: PL-1337HOT 0 km/h Radio läuft.
BMW Z1: W-22HTLDS 80 km/h
VW Passat: PL-1337HOT 100 km/h
BMW Z1: W-22HTLDS 50 km/h
VW Passat: PL-1337HOT 80 km/h
```

12.4 Konstruktor

Konstruktoren sind spezielle Methoden die ausschließlich zum Anlegen einer Instanz (eines Objekts) einer Klasse dienen.

Es gelten folgende Regeln, die in den nächsten Abschnitten noch näher erklärt werden:

- Ein Konstruktor ist eine spezielle Methode einer Klasse und zwar eine Instanzmethode.
- Ein Konstruktor hat immer denselben Namen wie die Klasse.
- Ein Konstruktor hat keinen Rückgabetyp, weil er eine Instanz der eigenen Klasse zurückgibt.
- Wenn in einer Klasse kein Konstruktor definiert ist, existiert trotzdem der sogenannte Standardkonstruktor (Defaultkonstruktor) der parameterlos ist.
- Der Standardkonstruktor kann auch explizit implementiert werden.
- Wenn ein Konstruktor implementiert wird, der nicht der Standardkonstruktor ist, hat die Klasse keinen Standardkonstruktor.
- Eine Klasse kann beliebig viele Konstruktoren haben.
- Konstruktoren können sich auch gegenseitig aufrufen (Constructor Chaining).

12.4.1 Constructor Chaining

Konstruktoren können sich während dem Anlegen einer Instanz gegenseitig aufrufen. Sowohl in Java, als auch in C# erfolgt das Chaining durch die Verwendung des Schlüsselwortes `this` in einem Konstruktor.

Die Klasse Person enthält im unteren Beispiel zwei Konstruktoren, wobei der zweite Konstruktor den ersten auruft.

C# Beispiel für Constructor Chaining

```
using System;

namespace AT_HTLDonaustadt_SLOG
{
    class Person
    {
        public string FirstName { get; private set; } // property
        public string LastName { get; private set; } // property

        public Person(string firstname, string lastname)
        {
            FirstName = firstname;
            LastName = lastname;

            identifier = counter;
            counter++;
        }

        public Person(string fullname)
            : this(fullname.Split(' ')[0], fullname.Split(' ')[1])
        {
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person("Anna Komarek");
        Console.WriteLine(p.FirstName + " " + p.LastName);
    }
}
```

Ausgabe: Anna Komarek

Aufruf des anderen Konstruktors durch `this`

12.5 Zerstörung von Instanzen ("Destruktor")

Die Zerstörung von Instanzen kann in Java und C# nicht explizit aufgerufen werden, da die Laufzeitumgebungen beider Sprache eine Garbage Collection besitzen. Trotzdem gibt es die Möglichkeit, eine Methode einzubauen, die bei der Zerstörung der Instanz aufgerufen wird. In C# kann man jede beliebige Klasse vom Interface [IDisposable](#) ableiten und dessen Methode

```
public void Dispose();
```

implementieren. Diese Methode wird immer dann aufgerufen, wenn eine Instanz der Klasse von der Garbage Collection zerstört wird.

12.6 Grundprinzipien der Objektorientierung

Der **Zustand** eines Objekts (Instanz) wird in Daten repräsentiert (Instanzvariablen).

Das **Verhalten** eines Objekts (Instanz) wird in Methoden implementiert (Instanzmethoden).

Der Ausdruck Objekt und der Ausdruck Instanz sind beide gebräuchlich und sind ident!

Eine Klasse stellt einen Bauplan für beliebig viele Objekte (Instanzen) dar.

Moderne objektorientierte Sprachen, wie Java und C#, unterstützen die drei Grundprinzipien der Objektorientierung:

1. **Encapsulation** (Kapselung) von Daten und Methoden: inklusive der Möglichkeit über Schutzstufen (Access Modifier) festzulegen, wie sehr Daten und Methoden geschützt sind (information hiding)
2. **Inheritance** (Vererbung)
3. **Polymorphie** (Vielgestaltigkeit)

Das Beispiel im vorherigen Kapitel soll einen ungefähren Eindruck von Klassen geben und zeigt, wie man aus Klassen Objekte erzeugt, die während dem Programmablauf verändert werden.

12.7 Eine Geschichte von Roman objektorientiert "erzählt"

Dieser Abschnitt beinhaltet die Geschichte von Roman der ins Kino möchte und dafür Knete¹⁶ benötigt. Diese Geschichte wurde vom Autor erfunden um zu demonstrieren, dass auch normale Geschichten des Alltags zu einem gewissen Grad in objektorientierten Programmen "erzählt" werden können.

Die Personen die handeln werden – wie die Autos im vorherigen Beispiel – als Objekte einer Personenklasse angelegt. Ihre Handlungen sind Methoden, ihre Eigenschaften sind Instanzvariablen.

Die Geschichte von Roman sind die beiden allerersten Videos, die vom Autor auf seinem YouTube-Channel veröffentlicht wurden.

Wer nicht gerne liest, sondern lieber ein Video schaut, ist herzlich eingeladen, sich die beiden Videos [Teil 1](#) und [Teil 2](#) anzusehen und anzuhören.

Das Beispiel in diesem Abschnitt soll illustrieren, dass Objektorientierung ein universelles Konzept ist, mit dem man Geschichten mitsamt aller handelnden Personen in einem Computer ablaufen lassen kann.

¹⁶ Knete ist umgangssprachlich für Geld

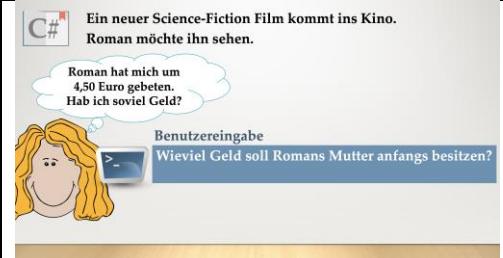
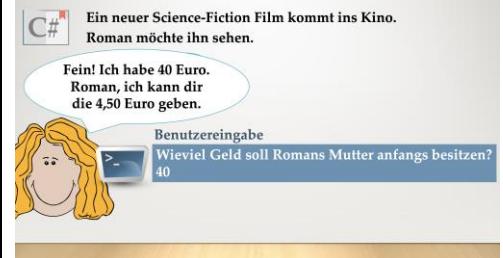
12.7.1 Die Geschichte – Daten und Methoden in Klassen

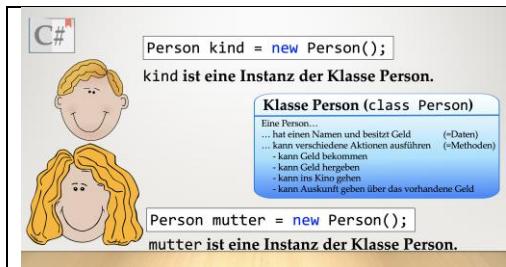
Dieser Abschnitt entspricht dem [Teil 1](#) von "Roman möchte ins Kino gehen und braucht Knete" .

Es geht um eine Geschichte, die drei verschiedene Enden kennt. Ziel ist es, die Geschichte als objektorientiertes Programm in C# zu schreiben.



Folie	Beschreibung
 <p>C# Ein neuer Science-Fiction Film kommt ins Kino. Roman möchte ihn sehen. Kinokarte kostet  10 +  2 +  50 =  12,50 TICKET Hm, habe ich überhaupt soviel Knete?</p>	<p>Ein neuer Science-Fiction Film kommt ins Kino und Roman möchte ihn sich anschauen. Wir nehmen an, dass die Kinokarte 12,50 Euro kostet.</p> <p>Roman überlegt, ob er überhaupt so viel Geld hat.</p>
 <p>C# Ein neuer Science-Fiction Film kommt ins Kino. Roman möchte ihn sehen. Kinokarte kostet  10 +  2 +  50 =  12,50 TICKET Hm, habe ich überhaupt soviel Knete? Benutzereingabe Wieviel Geld soll Roman anfangs besitzen?</p>	<p>Abhängig davon, ob Roman die 12,50 Euro für die Kinokarte hat, kann er ins Kino oder nicht.</p> <p>Im Computerprogramm lassen wir den Benutzer den Geldbetrag, den Roman zur Verfügung hat, eingeben.</p>
 <p>C# Ein neuer Science-Fiction Film kommt ins Kino. Roman möchte ihn sehen. Kinokarte kostet  10 +  2 +  50 =  12,50 TICKET Ich habe 20 Euro. 20 > 12,50 Jubell!!! Benutzereingabe Wieviel Geld soll Roman anfangs besitzen? 20 Roman kauft sich eine Kinokarte und schaut sich den Film an!</p>	<p>Angenommen der Benutzer gibt 20 Euro ein (oder einen anderen Geldbetrag der zumindest 12,50 Euro gross ist). Dann kann Roman jubeln. Denn er hat genügend Geld und er kann sich eine Kinokarte kaufen und ins Kino gehen.</p> <p>Ende 1 der Geschichte! Roman hat genügend Geld.</p>
 <p>C# Ein neuer Science-Fiction Film kommt ins Kino. Roman möchte ihn sehen. Kinokarte kostet  10 +  2 +  50 =  12,50 TICKET Hm, habe ich überhaupt soviel Knete? Benutzereingabe Wieviel Geld soll Roman anfangs besitzen? 8 Roman fragt seine Mutter um die fehlenden 4,50 Euro!</p>	<p>Angenommen der Benutzer gibt Roman am Beginn nur 8 Euro.</p>
	<p>Dann wird Roman feststellen, dass er zuwenig Geld für eine Kinokarte hat.</p> <p>Er fragt seine Mutter, ob sie ihm die fehlenden 4,50 Euro geben kann.</p>

 <p>C# Ein neuer Science-Fiction Film kommt ins Kino. Roman möchte ihn sehen.</p> <p>Roman: Ich hab mich um 4,50 Euro gebeten. Hab ich soviel Geld?</p> <p>Benutzereingabe Wieviel Geld soll Romans Mutter anfangs besitzen?</p>	<p>Die Mutter möchte Roman das Geld geben, aber sie muss vorher bestimmen, ob sie selbst überhaupt soviel Geld hat.</p> <p>Im Computerprogramm lassen wir den Benutzer nun auch den Geldbetrag eingeben, den die Mutter besitzen soll.</p>
 <p>C# Ein neuer Science-Fiction Film kommt ins Kino. Roman möchte ihn sehen.</p> <p>Mutter: Fein! Ich habe 40 Euro. Roman, ich kann dir die 4,50 Euro geben.</p> <p>Benutzereingabe Wieviel Geld soll Romans Mutter anfangs besitzen? 40</p>	<p>Angenommen der Benutzer gibt der Mutter am Beginn 40 Euro. Dann freut sie sich und kann Roman die 4,50 Euro geben.</p> <p>Ende 2 der Geschichte! Roman bekommt von seiner Mutter Geld und hat dann genügend Geld.</p>
 <p>C# Ein neuer Science-Fiction Film kommt ins Kino. Roman möchte ihn sehen.</p> <p>Mutter: Fein! Ich habe 40 Euro. Roman, ich kann dir die 4,50 Euro geben.</p> <p>Benutzereingabe Wieviel Geld soll Romans Mutter anfangs besitzen? 40</p> <p>Roman bekommt die 4,50 Euro, kauft sich eine Karte und geht ins Kino!</p>	<p>Roman bekommt von seiner Mutter die 4,50 Euro, besitzt daher gemeinsam mit seinen 8 Euro insgesamt 12,50 Euro.</p> <p>Er kauft sich damit die Kinokarte und geht ins Kino.</p>
 <p>C# Im Kino wird ein Film in 3D gespielt. Roman möchte ihn sehen.</p> <p>Mutter: Tut mir leid! Ich habe leider nur 2 Euro.</p> <p>Benutzereingabe Wieviel Geld soll Romans Mutter anfangs besitzen? 2</p>	<p>Angenommen Romans Mutter hat gerade nur 2 Euro im Geldbörse. Dann kann sie Roman nicht weiterhelfen. Mit den 2 Euro hätte Roman nur insgesamt 10 Euro, um 2,50 Euro zu wenig für die Kinokarte.</p>
 <p>C# Im Kino wird ein Film in 3D gespielt. Roman möchte ihn sehen.</p> <p>Mutter: Tut mir leid! Ich habe leider nur 2 Euro.</p> <p>Benutzereingabe Wieviel Geld soll Romans Mutter anfangs besitzen? 2</p> <p>Roman kann nicht ins Kino gehen und geht mit Kumpeln Fußballspielen.</p>	<p>Roman geht stattdessen mit seinen Freunden Fußballspielen.</p> <p>Ende 3 der Geschichte! Roman hat nicht genügend Geld.</p>
<p>C# Objektorientierte Sichtweise</p> <p>Klasse Person (class Person)</p> <p>Eine Person...</p> <ul style="list-style-type: none"> ... hat einen Namen und besitzt Geld (=Daten) ... kann verschiedene "Aktionen" ausführen (=Methoden) <ul style="list-style-type: none"> - kann Geld bekommen - kann Geld hergeben - kann ins Kino gehen - kann Auskunft geben über das vorhandene Geld 	<p>Um diese Geschichte in ein Programm umzusetzen erstellen wir für die beiden Handelnden, also Roman und seine Mutter eine gemeinsame Klasse und überlegen, welche Daten und welche Aktionen eine Person ausführen kann.</p> <p>Beachte: Wir vereinigen dabei die Aktionen von Roman und seiner Mutter in einer Klasse. Prinzipiell soll eine Person ins Kino gehen können. Dass es nur Roman bei zwei möglichen Enden der Geschichte macht (und nicht seine Mutter) berücksichtigen wir hier noch nicht.</p> <p>Gehe zur Stelle im Video.</p>



Zur Laufzeit möchten wir für beide Personen ein Objekt dieser Klasse erstellen.

[Gehe zur Stelle im Video.](#)

12.7.2 Die Klasse Person für Roman und seine Mutter

Zunächst erstellen wir die Klasse für eine Person und fügen alle Instanzvariablen (als Daten) und Methoden (für die Aktionen) ein.

C# Die Klasse Person

```
class Person
{
    // Instanzvariablen
    public string name;
    public double geld; // in Euro (kann auch decimal sein)

    // Methoden (Aktionen)
    public void BekommtGeld( double betrag )
    {
        Console.WriteLine(name + " bekommt " + betrag + " Euro.");
        geld += betrag;
        GibtAuskunft();
    }

    public void GibtGeldHer(double betrag)
    {
        Console.WriteLine(name + " gibt " + betrag + " Euro her.");
        geld -= betrag;
        GibtAuskunft();
    }

    public void GehtInsKino(double preis)
    {
        Console.WriteLine(name + " geht ins Kino und kauft sich eine Karte um " + preis+" Euro.");
        GibtGeldHer(preis); // bezahlt mit dem Geld bei der Kassa
        Console.WriteLine(name + " schaut sich den Film an.");
    }

    public void GibtAuskunft()
    {
        Console.WriteLine(name + " hat noch " + geld + " Euro.");
    }
}
```

Beachte, dass die Methoden absichtlich so geschrieben wurden, dass viele Ausgaben mit `Console.WriteLine` enthalten sind.

12.7.3 Die Programmierung der Geschichte im Main

Alle Varianten der Geschichte mit den drei möglichen Enden, können in diesem Entscheidungsbaum zusammengefaßt werden. [Gehe zur Stelle im Video.](#)



Abbildung 15: Der Entscheidungsbaum der Geschichte von Roman der ins Kino möchte

Die Geschichte wird einfach im Main programmiert. Das Geld, das Roman und die Mutter am Beginn besitzen sollen, werden am Beginn des Programms eingelesen.

C# - Die Geschichte ausprogrammiert mit den Fallunterscheidungen

```
static void Main(string[] args)
{
    Console.WriteLine("Wieviel Geld soll Roman anfangs haben? ");
    double anfangsbetrag = double.Parse(Console.ReadLine());

    Person kind = new Person(); //Erzeugung des Objekts für Roman und Setzen des Anfangsbetrags
    kind.name = "Roman";
    kind.geld = anfangsbetrag;

    Console.WriteLine("Wieviel Geld soll Romans Mutter anfangs haben? ");
    anfangsbetrag = double.Parse(Console.ReadLine());

    Person mutter = new Person(); //Erzeugung des Objekts der Mutter und Setzen des Anfangsbetrags
    mutter.name = "Mutter";
    mutter.geld = anfangsbetrag;

    Console.WriteLine("Hier beginnt die eigentliche Geschichte...\n");

    const double preisDerKinokarte = 12.5; // in Euro (als Konstante)

    if ( kind.geld >= preisDerKinokarte )
        { // Roman hat genug Geld = Ende 1 der Geschichte!
            kind.GehtInsKino(preisDerKinokarte);
        }
    else
        { // Roman hat nicht genug Geld
            double fehlendesGeld = preisDerKinokarte - kind.geld;
            Console.WriteLine(kind.name + " fragt " + mutter.name + " um "
                + fehlendesGeld + " Euro.");
            if ( mutter.geld >= fehlendesGeld )
                { // Mutter gibt Roman das fehlende Geld
                    mutter.GibtGeldHer(fehlendesGeld);
                    kind.BekommtGeld(fehlendesGeld);
                    kind.GehtInsKino(preisDerKinokarte); // = Ende 2 der Geschichte!
                }
            else
                { // Roman kann nicht ins Kino = Ende 3 der Geschichte!
                    Console.WriteLine(kind.name + " kann nicht ins Kino gehen.");
                    Console.WriteLine(kind.name + " geht mit den Kumpeln Fußballspielen.");
                }
        }
}
```

Das Programm kann nun alle drei Varianten der Geschichte "erzählen".

Programmablauf: Ende 1 der Geschichte! Roman hat genügend Geld.

Wieviel Geld soll Roman anfangs haben? 20

Wieviel Geld soll Romans Mutter anfangs haben? 10

Hier beginnt die eigentliche Geschichte...

Roman geht ins Kino und kauft sich eine Karte um 12.5 Euro.

Roman gibt 12.5 Euro her.

Roman hat noch 7.5 Euro.

Roman schaut sich den Film an.

Programmablauf: Ende 2 der Geschichte! Roman bekommt von seiner Mutter Geld und hat dann genügend Geld.

Wieviel Geld soll Roman anfangs haben? 8

Wieviel Geld soll Romans Mutter anfangs haben? 40

Hier beginnt die eigentliche Geschichte...

Roman fragt Mutter um 4.5 Euro.

Mutter gibt 4.5 Euro her.

Mutter hat noch 35.5 Euro.

Roman bekommt 4.5 Euro.

Roman hat noch 12.5 Euro.

Roman geht ins Kino und kauft sich eine Karte um 12.5 Euro.

Roman gibt 12.5 Euro her.

Roman hat noch 0 Euro.

Roman schaut sich den Film an.

Programmablauf: Ende 3 der Geschichte! Roman hat nicht genügend Geld.

Wieviel Geld soll Roman anfangs haben? 8

Wieviel Geld soll Romans Mutter anfangs haben? 2

Hier beginnt die eigentliche Geschichte...

Roman fragt Mutter um 4.5 Euro.

Roman kann nicht ins Kino gehen.

Roman geht mit den Kumpeln Fußballspielen.

Je nach dem Ablauf werden verschiedene Methoden aufgerufen, die den Wert der Instanzvariablen der beiden Personen-Objekte, Roman und seiner Mutter, verändern. Bei den Ausgaben werden diese Werte auf der Konsole ausgegeben.

12.8 Encapsulation (Kapselung)

Dieser Abschnitt behandelt den zweiten Teil von "Roman möchte ins Kino gehen und braucht Knete" [Teil 2](#).

In diesem Abschnitt wird erklärt, warum das Programm aus dem ersten Teil noch mangelhaft ist, weil die Instanzvariablen der Objekte vor Zugriff von außerhalb noch nicht geschützt sind.

Kurzgeschichten in C# erzählt und in die Kiste geklopft von Thomas Schlägl
Fortsetzung zu:
Roman möchte ins Kino gehen und braucht Knete
Übung zu
Klassen und Instanzen
Teil 2: Datenkapselung und Information Hiding

Folie	Beschreibung
<pre>class Person { // Instanzvariablen public string Name; public double Geld; // in Euro // Methoden (Aktionen) public void BekommtGeld(double betrag) { Console.WriteLine(Name + " bekommt " + betrag + " Euro."); Geld = Geld + betrag; GibtAuskunft(); } public void GibtGeldHin(double betrag) { Console.WriteLine(Name + " gibt " + betrag + " Euro her."); Geld = Geld - betrag; GibtAuskunft(); } } class Bankkonto { // Instanzvariablen public string IBAN; // z. B. "AT626000000093040950" public double Kontostand; // in Euro // Methoden (Aktionen) public void Gutbuchen(double betrag) { Datenbank.Write(IBAN, betrag, "Gutbuchung"); Kontostand = Kontostand + betrag; DrucktKontostand(); } public void Abbuchen(double betrag) { Datenbank.Write(IBAN, betrag, "Abbuchung"); Kontostand = Kontostand - betrag; DrucktKontostand(); } }</pre>	<p>Gedanklich starten wir bei der Klasse Person des ersten Teils.</p>
	<p>Wir stellen uns nun vor, dass dies keine Klasse für eine Privatperson ist, sondern die Klasse eines Bankkontos auf dem Geld liegt und das einen IBAN-Code hat.</p> <p>Der Kontostand in jedem Objekt dieser Klasse ist natürlich sehr wichtig und es ist wichtig, dass nur wenige ausgewählte Programmstellen den Kontostand verändern dürfen.</p> <p>Gehe zur Stelle im Video.</p>
	<p>Die Situation zwischen einem Bankkonto und einem Geldbörsel einer Person ist durchaus sehr analog.</p> <p>Wenn Roman Geld von seiner Mutter bekommt, entspricht das in etwas, wenn ein Kunde von einer Bank Geld erhält um sich einen coolen Flitzer zu kaufen. Auch hier überweist die Bank dem Kunden Geld, genauso wie die Mutter Roman Geld gibt.</p>
	<p>Der Programmcode der Geld überweist ist sehr wichtig und es ist sehr sinnvoll, dass sie die einzige Methode ist, die tatsächlich die Instanzvariable Kontostand im Objekt verändert.</p>
	<p>Und hier ist das Problem im Programm. Theoretisch könnte ein Programmierer versehentlich (oder wissentlich!) eine Programmzeile schreiben, die den Wert von Kontostand direkt verändert ohne die dafür vorgesehene Methode verwendet.</p>

C# Wo ist das Problem?

```

if ( bankkonto.Kontostand >= kreditsumme &&
    transaktion.IstFrlaucht())
{
    // Bank überweist die Kreditsumme an den Kunden
    bankkonto.ÜberweistBetrag(kreditsumme, kundenkonto);
    bankkonto.Kontostand -= preisEinesCoolenFlitzers;
    kundenkonto.HabtGeldIn(preisEinesCoolenFlitzers);
}

Ausweg: private string IBAN;
private double Kontostand;

```

Ausweg: `private string IBAN;`
`private double Kontostand;`

private bedeutet, dass die Instanzvariablen nur innerhalb der Klasse verwendet werden können und zwar nur in nicht-statischen Methoden!

Das kann verhindert werden!

Verwendet man das Schlüsselwort **private**, statt dem Schlüsselwort **public** vor einer Instanzvariable, kann diese nur von einer Methode innerhalb der Klasse der Methode verändert werden. Und NICHT mehr von außerhalb!

Ändert man diesen sogenannten Zugriffsmodifizierer auf **private** um, zeigt die Entwicklungsumgebung mit einer roten Wellenlinie sofort an, dass die Zuweisung auf die Instanzvariable nicht mehr möglich ist.

Das Programm kompiliert nicht mehr und der ungewünschte Zugriff ist unwiderruflich verhindert.

Fährt man mit der Maus über die rote Wellenlinie, erhält man die Information, dass der Schutzgrad der Variable eine Veränderung unmöglich macht.

C# kind ist eine Instanz der Klasse Person.

```

Person kind = new Person();
kind.Name = "Roman";
kind.Geld = anfangsbetrag;

```

Wir verändern unsere Klassendefinition in der analogen Art und Weise und schreiben daher statt

```

class Person
{
    // Instanzvariablen
    public string Name;
    public double Geld; // in Euro (kann auch decimal sein)
}

```

statt

```

class Person
{
    // Instanzvariablen
    private string Name;
    private double Geld; // in Euro (kann auch decimal sein)
}

```

C# kind ist eine Instanz der Klasse Person.

```

Person kind = new Person();
kind.Name = "Roman";
kind.Geld = anfangsbetrag;

```

Das hat zur Folge, dass die Verwendung im Main rot unterstrichen wird.

```

Console.WriteLine("Wieviel Geld soll Roman anfangs haben? ");
double anfangsbetrag = double.Parse(Console.ReadLine());

Person kind = new Person();
kind.Name = "Roman";
kind.Geld = anfangsbetrag;

Console.WriteLine("Wieviel Geld soll Romans Mutter anfangs haben? ");
anfangsbetrag = double.Parse(Console.ReadLine());

Person mutter = new Person();
mutter.Name = "Mutter";
mutter.Geld = anfangsbetrag;

```

Die Instanzvariablen der beiden Objekte können im Main nicht mehr verändert werden. Sehr gut!

Nur wie können wir jetzt die Werte in den Objekten setzen?

 <p>kind ist eine Instanz der Klasse Person.</p> <pre>Person kind = new Person(); kind.Name = "Roman"; kind.Geld = anfangsbetrag;</pre> <pre>Person kind = new Person("Roman",anfangsbetrag);</pre>	<p>Die Antwort steckt beim Anlegen des Objekts. Person() hinter dem new ist nämlich tatsächlich der Aufruf einer Methode, dem sogenannten Konstruktor.</p>
 <p>kind ist eine Instanz der Klasse Person.</p> <pre>Person kind = new Person(); kind.Name = "Roman"; kind.Geld = anfangsbetrag;</pre> <pre>Person kind = new Person("Roman",anfangsbetrag);</pre> <p>Person(...) ist eine spezielle Methode der Klasse Person, nämlich ein sogenannter Konstruktor. Ein Konstruktor ist eine Methode einer Klasse... <ul style="list-style-type: none"> • die immer denselben Namen hat wie die Klasse, • keinen oder mehrere Parameter haben kann UND • nur beim Erzeugen einer Instanz dieser Klasse mit new aufgerufen </p>	<p>Ein Konstruktor ist eine Methode, die immer beim Anlegen eines Objekts aufgerufen wird. So einer Methode kann man gewünschte Werte für Instanzvariablen als Inputparameter mitgeben, damit die Methode diese Werte dann in den Instanzvariablen setzt.</p>
 <p>kind ist eine Instanz der Klasse Person.</p> <pre>Person kind = new Person(); kind.Name = "Roman"; kind.Geld = anfangsbetrag;</pre> <pre>Person kind = new Person("Roman",anfangsbetrag);</pre> <p>Person(...) ist eine spezielle Methode der Klasse Person, nämlich ein sogenannter Konstruktor.</p> <pre>public Person(string name, double geld) { Name = name; Geld = geld; }</pre>	<p>Der Konstruktor von der Klasse Person führt die Zuweisungen die bisher im Main gestanden sind, in der Methode aus.</p> <pre>class Person { // Instanzvariablen private string Name; private double Geld; // in Euro // Konstruktor public Person(string name, double geld) { Name = name; Geld = geld; } }</pre>
	<p>Die Objekte werden im Main nun so angelegt, dass die gewünschten Werte (Name und verfügbares Geld am Anfang) dem Konstruktor übergeben werden.</p> <pre>Person kind = new Person("Roman", anfangsbetrag); // kind.Name = "Roman"; // kind.Geld = anfangsbetrag; Console.WriteLine("Wieviel Geld soll Romans Mutter anfangs haben? "); anfangsbetrag = double.Parse(Console.ReadLine()); Person mutter = new Person("Mutter", anfangsbetrag); // mutter.Name = "Mutter"; // mutter.Geld = anfangsbetrag;</pre> <p>Sie auskommentierten Zeilen können natürlich vollständig entfernt werden.</p>

Es gibt nun noch einige weitere Stellen, die auf die Instanzvariablen zugreifen um die Werte aus dem Objekt auszulesen.

```

if ( kind.Geld >= preisDerKinokarte )
{
    kind.GehtInsKino(preisDerKinokarte);
}
else
{   // Roman hat nicht genug Geld
    double fehlendesGeld = preisDerKinokarte - kind.Geld;
    Console.WriteLine(kind.Name + " fragt " + mutter.Name + " um "
        + fehlendesGeld + " Euro.");
}

```

Wie können diese Kompilierfehler behoben werden?

Um eine private Instanzvariable aus einem Objekt auszulesen, ist eine Methode notwendig, die man **Getter** nennt. Um eine private Instanzvariable in einem Objekt von außerhalb des Objekts auf einen bestimmten Wert zu setzen, schreibt man eine Methode, die man **Setter** nennt.

```

private string Name;
private double Geld; // in Euro

public double GetGeld() // der Getter der Instanzvariable Geld
{
    return Geld;
}

public void SetGeld(double geld) // der Setter für die Instanzvariable Geld
{
    Geld = geld;
}

public string GetName() // der Getter der Instanzvariable Name
{
    return Name;
}

```

Es ist nicht unbedingt notwendig, diese Methoden zu definieren. Häufig werden **Setter** weggelassen, da die Werte von Instanzvariablen nur innerhalb von anderen Methoden einer Klasse gesetzt werden sollen.

An allen Stellen mit den roten Wellenlinien kann man nun die Getter aufrufen.

```

if ( kind.GetGeld() >= preisDerKinokarte )
{
    kind.GehtInsKino(preisDerKinokarte);
}
else
{   // Roman hat nicht genug Geld
    double fehlendesGeld = preisDerKinokarte - kind.GetGeld();
    Console.WriteLine(kind.GetName() + " fragt " + mutter.GetName() + " um "
        + fehlendesGeld + " Euro.");
}

```

Kapselung bedeutet, dass Daten vor dem Zugriff durch andere Codeteile geschützt sind.

12.9 Properties

Wird in einer Klasse eine Instanzvariable definiert, ist es oft notwendig einen Getter und gelegentlich einen Setter zu definieren.

Instanzvariable mit Getter und Setter	Property mit Getter und Setter
<pre>class Person { private string name; public double GetName() // Getter { return name; } public void SetName(string name) // Setter { this.name = name; } ... }</pre>	<pre>class Person { private string name; public string Name // Property { get // Getter { return name; } set // Setter { name = value; } } ... } kann abgekürzt werden mit: class Person { public string Name { get; set; } }</pre>

Man kann bei einem Property¹⁷ beim Lesen (`get`) und beim Setzen (`set`) beliebigen Programmcode hinzufügen und z. B. beim Setzen eine Überprüfung durchführen. Das ist ein Riesenvorteil gegenüber einer normalen Instanzvariable mit einer eigenen Get- und Set-Methode.

Da dieser Code immer sehr ähnlich ist, werden in modernen Programmiersprachen Features eingebaut um diesen sogenannten "boilerplate code" zu reduzieren. In C# sind das die Properties, die häufiger als Instanzvariablen verwendet werden.

`value` ist ein Schlüsselwort in C# und enthält den Wert dem Property rechts vom `=` einer Zuweisung zugewiesen wird. Der Datentyp von `value` ist derselbe wie der des Properties. Dieser Wert kann in `set` verwendet und beispielsweise überprüft werden.

`get` und `set` sind also eigentlich kleine Methode, die jedes Property zur Verfügung stehen. Sie können implementiert werden, müssen es aber nicht.

Die weiteren Abschnitte beschreiben fünf sehr häufig verwendeten Varianten von Properties, die in C# alle sehr intensiv genutzt werden.

Als Beispiel wird eine Produktklasse verwendet. Jedes Produkt hat einen Preis und ist mit einer gewissen Menge auf Lager

¹⁷ <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties>

12.9.1 Auto-implemented Property mit public get und public set

Es wird beim Property einfach `public int Lagermenge { get; set; }` angegeben. Dann erzeugt C# im Hintergrund selbstständig den Programmcode für einen Getter und Setter einer Instanzvariable, die den Wert eines Properties speichert.

Code mit Instanzvariable	Code mit Property
Ein Property soll innerhalb und außerhalb der Klasse gelesen und gesetzt werden können.	
<pre>class Produkt { private int lagermenge; public int GetLagermenge() { return lagermenge; } public void SetLagermenge(int neu) { lagermenge = neu; } ... public Produkt(..., int lagermenge, ...) { ... this.lagermenge = lagermenge; ... } ... }</pre>	<pre>class Produkt { public int Lagermenge { get; set; } ... public Produkt(..., int lagermenge, ...) { ... Lagermenge = lagermenge; ... } ... }</pre>
Verwendung: <pre>Produkt p = new Produkt(..., 5, ...); Console.WriteLine(p.GetLagermenge()); p.SetLagermenge(10); Console.WriteLine(p.GetLagermenge());</pre>	Verwendung: <pre>Produkt p = new Produkt(..., 5, ...); Console.WriteLine(p.Lagermenge); p.Lagermenge = 10; Console.WriteLine(p.Lagermenge);</pre>
lagermenge ist eine Instanzvariable.	Lagermenge ist ein auto-implemented Property mit einem public Setter.
	Eingabehilfe in Visual Studio: prop[TAB][TAB]

12.9.2 Auto-implemented Property mit public get und private set

Code mit Instanzvariable	Code mit Property
Eine Property soll nur innerhalb der Klasse gesetzt und von innerhalb und außerhalb gelesen werden.	
<pre>class Produkt { private double preis; public double GetPreis(){ return preis; } public Produkt(double preis, ...) { this.preis = preis; ... } public void ErhöhePreis(double prozent) { preis = preis * (1 + prozent / 100); } ... }</pre> <p>Verwendung:</p> <pre>Produkt p = new Produkt(15.95, ...); Console.WriteLine(p.GetPreis().ToString("C2")); p.ErhöhePreis(3.5); Console.WriteLine(p.GetPreis().ToString("C2"));</pre>	<pre>class Produkt { public double Preis { get; private set; } public Produkt(double preis, ...) { Preis = preis; ... } public void ErhöhePreis(double prozent) { Preis = Preis * (1 + prozent / 100); } ... }</pre> <p>Verwendung:</p> <pre>Produkt p = new Produkt(15.95, ...); Console.WriteLine(p.Preis.ToString("C2")); p.ErhöhePreis(3.5); Console.WriteLine(p.Preis.ToString("C2"));</pre>
preis ist eine Instanzvariable.	Preis ist ein auto-implemented Property mit einem private Setter.
Eingabehilfe in Visual Studio: propg[TAB][TAB]	

12.9.3 Auto-implemented Property mit public `get` und keinem `set`

Code mit Instanzvariable	Code mit Property
Eine Instanzvariable soll nur im Konstruktor gesetzt werden können und von innerhalb und außerhalb gelesen .	
<pre>class Produkt { private readonly int bezeichnung; public int GetBezeichnung() { return bezeichnung; } ... public Produkt(..., int bezeichnung, ...) { ... this.bezeichnung = bezeichnung; ... } ... }</pre>	<pre>class Produkt { public int Bezeichnung { get; } public Produkt(..., int bezeichnung, ...) { ... Bezeichnung = bezeichnung; ... } ... }</pre>
Verwendung: <code>Produkt p = new Produkt(..., "Hammer", ...); Console.WriteLine(p.GetBezeichnung());</code> bezeichnung ist ein readonly Instanzvariable, die nur im Konstruktor gesetzt werden kann.	Verwendung: <code>Produkt p = new Produkt(..., "Hammer", ...); Console.WriteLine(p.Bezeichnung);</code> Gesamtwert ist ein readonly Property, das nur im Konstruktor gesetzt werden kann.

12.9.4 Berechnetes Property mit public get

Code mit Methode	Code mit Property
<p>Eine Property soll einen berechneten Wert zurückgeben.</p>	
<pre>class Produkt { private double preis; private int lagermenge; public double GetPreis(){ return preis; } public int GetLagermenge(){ return lagermenge; } public double GetGesamtwert(){ return lagermenge*preis; } public Produkt(string b, double l, int p ...) { ... lagermenge=l; preis = p; ... } ... }</pre>	<pre>class Produkt { public int Lagermenge { get; set; } public double Preis { get; private set; } public double Gesamtwert { get { return Lagermenge*Preis; } } public Produkt(string bezeichnung, int l, double p) { ... Lagermenge = l; Preis = p; ... } ... }</pre>
<p>Verwendung:</p> <pre>Produkt p = new Produkt ("Hammer", ...); Console.WriteLine(p.GetGesamtwert());</pre>	<p>Verwendung:</p> <pre>Produkt p = new Produkt ("Hammer", ...); Console.WriteLine(p.Gesamtwert);</pre>
<p>GetGesamtwert() ist eine Methode, die das Produkt aus lagermenge und preis zurückgibt.</p>	<p>Gesamtwert ist ein berechnetes Property, dass das Produkt aus Lagermenge und Preis zurückgibt.</p>

12.9.5 Full-implemented Property mit programmiertem get und set

Code mit Methode	Code mit Property
<p>Bevor ein Property auf einen bestimmten Wert gesetzt wird, soll der Wert überprüft werden.</p> <pre>class Produkt { private int lagermenge; public int GetLagermenge() { return lagermenge; } public void SetLagermenge(int neu) { if (neu >= 0) { lagermenge = neu; } else { // ungültig -> auf 0 setzen lagermenge = 0; } } ... public Produkt(..., int lagermenge, ...) { this.lagermenge = lagermenge; } }</pre>	<pre>class Produkt { private int lagermenge; public int Lagermenge { get { return lagermenge; } set { if (value >= 0) { lagermenge = value; } else { // ungültig lagermenge = 0; } } } public Produkt(..., int lagermenge, ...) { Lagermenge = lagermenge; } }</pre>
<p>Verwendung:</p> <pre>Produkt p = new Produkt(..., 5, ...); p.SetLagermenge(10); p.SetLagermenge(-5); // sinnlos, lagermenge wird 0</pre>	<p>Verwendung:</p> <pre>Produkt p = new Produkt(..., 5, ...); p.Lagermenge = 10; // ruft set p.Lagermenge = -5; // sinnlos, lagermenge wird auf 0 gesetzt</pre>
	<p>Eingabehilfe in Visual Studio: propfull[TAB][TAB]</p>

12.9.6 Die "Methoden" get und set

Bei einem Property, das full-implemented ist, kann man sich get und set als Methoden vorstellen.

C#

```
class Person
{
    private string name;

    public string Name // Property
    {
        get // Getter
        {
            return name;
        }

        set // Setter
        {
            name = value;
        }
    ...
}
```

```
class Person
{
    private string name;

    public string Name // Property
    {
        int get() // Getter
        {
            return name;
        }

        void set(int value) // Setter
        {
            name = value;
        }
    ...
}
```

Die hellgrauen Teile kursiv geschriebenen Teile sind nur Vorstellungshilfen und werden in C# nicht benötigt.

13 Vererbung (Inheritance)

Vererbung in C# - und allen anderen modernen objekt-orientierten Programmiersprachen – bedeutet:

Eine neue Klasse wird aufbauend auf eine andere bereits existierende Klasse angelegt. Die neue Klasse benutzt Instanzvariablen, Properties und Methoden einer anderen Klasse mit.



Abbildung 16: Alle Eigenschaften der drei Produkte eines Computershops

Das Prinzip der Vererbung ist im [Coding Kurzgeschichten](#)-YouTube-Video ["Vererbung von Klassen in C# - Gemma in den Supermarkt"](#) kurz erklärt.



Vererbung von Klassen in C# - Gemma in den Supermarkt!

Coding Kurzgeschichten

Video 4: Coding Kurzgeschichten-Video „Gemma in den Supermarkt“

Vererbung umfasst drei verschiedene Bereiche:

1. Vererbung von Daten
2. Vererbung von Methoden
3. Polymorphie (Vielgestaltigkeit)

13.1 Ein erstes einfaches Beispiel

Das folgende Beispiel zeigt den einfachsten Fall von Vererbung. Eine einzelne Eigenschaft wird von einer Klasse auf eine weitere vererbt.

C# Einfache Vererbung von Daten	
<pre>class Person { public string Name { get; } public Person(string name) { Name = name; } }</pre>	Person ist eine existierende Klasse, die den Namen einer Person speichern kann.
<pre>class Student : Person { public string Schulkasse { get; } public Student(string name, string klasse) : base(name) { Schulkasse = klasse; } }</pre>	Student ist eine neue Klasse, die die Klasse Person benützt, da nach dem Klassennamen ein : und der Name von Person steht.
<pre>class Program { static void Main(string[] args) { Student s = new Student("Max", "2xHIF"); Console.WriteLine(s.Name + " geht in die " + s.Schulkasse); } }</pre>	<p>Das Objekt s der Klasse Student kann das Property Name verwenden, obwohl es in einer anderen Klasse Person definiert ist.</p> <p>Die Klasse Student erbt das Property Name von seiner Basisklasse Person.</p>

Sprechweisen:

"Die Klasse **Student** ist von der Klasse **Person** abgeleitet."

"Die Klasse **Student** erweitert die Klasse **Person**."

"Die Klasse **Person** ist die Basisklasse der Klasse **Student**."

13.2 Ein zweites Codebeispiel

Auch das folgende Beispiel zeigt, den einfachen Fall einer abgeleiteten Klasse, die von einer Basisklasse abgeleitet wird. Beide Klassen können im Main instanziert werden, d. h. es können Instanzen angelegt werden.

BaseClass.cs
<pre>class BaseClass { private int num; public int GetNum() { return num; } public BaseClass(int num) { this.num = num; } public override string ToString() { return "number=" + num; } }</pre>

DerivedClass.cs

```
class DerivedClass : BaseClass
{
    private double val;

    public DerivedClass(int num, double val)
        : base(num) // Aufruf des Konstruktors der Basisklasse
    {
        this.val = val;
    }

    public override string ToString()
    {
        return "DerivedClass "+base.ToString()
            + " val=" + val;
    }
}
```

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass a = new BaseClass(42);
        DerivedClass b = new DerivedClass(6, 4.5);

        Console.WriteLine(a);
        Console.WriteLine(b);
    }
}
```

Programmausgabe:

```
number=42
DerivedClass number=6 val=4.5
```

13.3 Ein Codebeispiel zur Vererbung von Methoden

Nicht nur Properties die in der Basisklasse definiert sind, werden an die abgeleiteten Klassen vererbt. Auch Methoden der Basisklasse sind automatisch auch als Methoden von Instanzen der abgeleiteten Klassen verfügbar. Das folgende Beispiel zeigt, dass die Methode IncreaseAge und auch das Property IsTeenager vererbt werden.

C# Beispiel zur Vererbung von Methoden

Person.cs

```
class Person
{
    public string Name { get; }
    public int Age { get; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public void IncreaseAge()
    {
        Age++;
    }
}
```

```

public Boolean IsTeenager
{
    get { return Age >= 12 && Age <= 19; }
}

public override string ToString()
{
    return $"{Name} is {Age} years old.";
}
}

```

Pupil.cs

```

class Pupil : Person
{
    public String SchoolClass { get; set; }

    public Pupil(string name, int age, string schoolclass)
        : base(name, age)
    {
        SchoolClass = schoolclass;
    }

    public override string ToString()
    {
        return base.ToString() + $" ({SchoolClass})";
    }
}

```

Program.cs

```

class Program
{
    static void Main(string[] args)
    {
        Person person = new Person("Thomas Schlägl", 50);
        Pupil pupil = new Pupil("Kevin Weber", 17, "4EHIF");

        Console.WriteLine(person);
        Console.WriteLine(pupil);

        Console.WriteLine($"Is {person.Name} a teenager?
                        {(person.IsTeenager ? ":-)" : ":-(")}");

        Console.WriteLine($"Is {pupil.Name} a teenager?
                        {(pupil.IsTeenager ? ":-)" : ":-(")}");
    }
}

```

13.4 Access Modifier

Der Zugriff auf Variablen wird durch die Zugriffsmodifizierer (Access Modifier) **private**, **protected** und **public** gesteuert, die eine Art von Zugriffsrechten festlegen.

Der Vorteil dieser Zugriffsrechte ist, dass man beim Entwickeln einer Klasse genau festlegen und einschränken kann, welche Daten durch welche Methoden geändert werden. Das erleichtert eine Fehlersuche und das Debugging erheblich!

C# Zugriffsmodifizierer für Variablen und Methoden	
Access Modifier	Beschreibung
private	Auf private Variablen und Methoden kann nur innerhalb der eigenen Klasse oder Struct zugegriffen werden.
protected	Auf protected Variablen und Methoden kann innerhalb der eigenen Klasse und jeder abgeleiteten Klasse zugegriffen werden. Die abgeleiteten Klassen können auch außerhalb des Packages (Java) bzw. des Namespaces (C#) liegen.
public	Auf public Variablen und Methoden kann von überall aus zugegriffen werden. Es gibt keine Einschränkung.

Abbildung 17: Übersicht über die drei Arten der Access Modifier in C#

Falls keiner der drei Zugriffsmodifizierer angegeben ist, ist die Variable defaultmäßig private.

Beispieldeklaration	C#
private int identifier;	private
int identifier;	private
protected int identifier;	protected
public int identifier;	public

Abbildung 18: Beispiele aller vier Arten einen Zugriffsmodifizierer anzugeben

13.5 Klassendesign eines Computershops

Für die Software eines Computershops sollen Klassen für die Artikel im Verkaufssortiment erstellt werden. Wir wählen der Einfachheit halber als Beispiel drei Artikel aus dem Waren sortiment aus und auch nur einige wenige ihrer Eigenschaften.

			
Notebook	Tastatur	Monitor	Identische Properties
Artikelnummer Lagerstand Preis (Euro) Bezeichnung	Artikelnummer Lagerstand Preis (Euro) Bezeichnung	Artikelnummer Lagerstand Preis (Euro) Bezeichnung	
CPU RAM (GByte) Anschlüsse	Zehnerblock (J/N) Anschluß	Technologie Größe Anschlüsse	!

Abbildung 19: Alle Eigenschaften der drei Produkte eines Computershops

Man kann grundsätzlich für jeden Artikel eine eigene Klasse definieren, die alle oben angegebenen Properties besitzen. In professionellen Systemen wird das (hoffentlich!) immer vermieden. Das Schlechte an diesen Klassen wäre, dass die Properties Artikelnummer, Lagerstand, Preis und Bezeichnung in allen Klassen definiert sind, d. h. es würde duplizierten Programmcode geben.

Duplizierter Programmcode hat einige schwerwiegende Nachteile:

- Bei späteren Änderungen müssen alle duplizierten Stellen gefunden werden. Wenn eine davon verändert wird, treten in Programmen oft rätselhafte Fehler auf, weil sich Dinge die sich gleich verhalten sollten, es nicht tun...
- Die Klassen werden unnötig groß und dadurch schlecht überblickbar.
- Die Klassen sind weitaus schwieriger zu testen. Man müsste in unserem Beispiel den Code von vier Properties testen und nicht nur ein einziges.

13.5.1 Vererbung der Eigenschaften

Das erklärte Ziel in unserem Beispiel ist:

Die vier Properties sollen nur in einer Klasse programmiert werden.

Dafür legen wir eine Klasse Artikel an, die alle diese Properties enthält. So eine Klasse, die Properties (oder Methoden) von vielen anderen Klassen vereinigt, nennt man Basisklasse oder Superklasse.

```

class Artikel
{
    public string Artikelnummer { get; }
    public int Lagerstand { get; private set; }
    public double Preis { get; set; private set; }
    public string Bezeichnung { get; }

    public Artikel(string artnr, string bez)
    {
        Artikelnummer = artnr;
        Bezeichnung = bez;
    }
}

```

Aus den einzelnen Artikelklassen, wie z. B. Notebook, können diese Properties entfernt werden:

```

class Notebook
{
    public string CPU { ... }
    public int RAM { ... }
    public string Anschlüsse { ... }

    public Notebook(...)
    {
        CPU = cpu;
        RAM = ram;
        Anschlüsse = anschl;
    }
}

```

Die Basisklasse Artikel enthält nun alle gemeinsamen Eigenschaften. Die speziellen Artikelklassen (Notebook, Tastatur, Monitor) enthalten die für sie speziellen Eigenschaften.

Allerdings brauchen diese Klassen jetzt eine Verbindung zu der Klasse Artikel in der die Properties hin verschoben wurden.

Wenn eine Klasse von einer Basisklasse abgeleitet wird, wird bei der Klasse : *Basisklassename* hinzugefügt. Z. B.

Basisklasse	
class Notebook : Artikel	base bedeutet die Basisklasse, also Artikel base(...) ist ein Konstruktoraufruf von Artikel
...	
public Notebook(...) : base(...)	
{	
...	
}	
...	
}	

C# erlaubt nur maximal eine einzige Basisklasse, d. h. die Sprache unterstützt - ebenso wie Java - keine Multiple Inheritance.

13.5.2 Klassenhierarchie

Zusammenhänge von Klassen werden in sogenannten Klassendiagrammen dargestellt. Die Pfeilrichtung zeigt immer von der speziellen Klasse (=abgeleitete Klasse) zur Basisklasse.

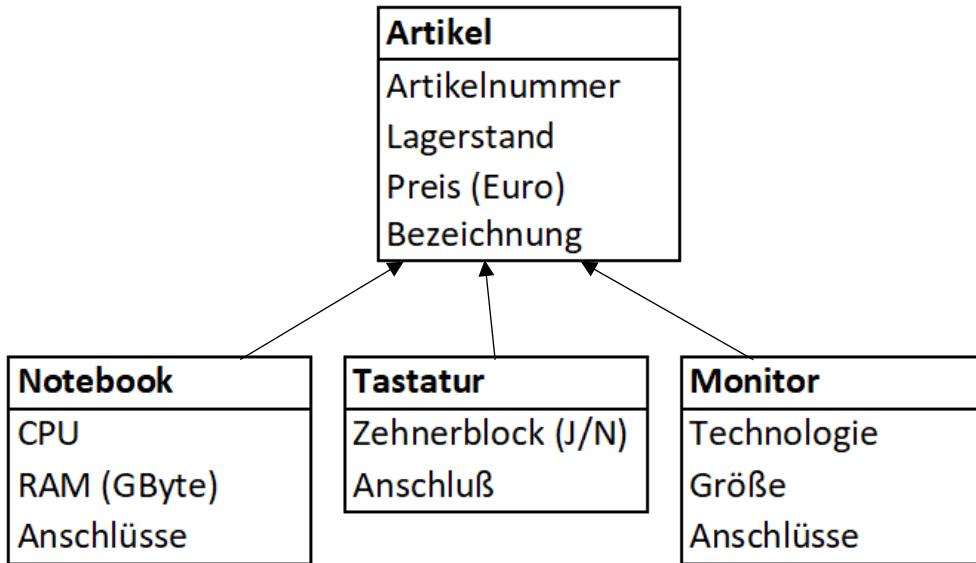


Abbildung 20: Klassenhierarchie der Produkte des Webshops

Da der Konstruktor in der Basisklasse Artikel nur aus den speziellen Produktklassen aufgerufen wird, erhält er den neuen Zugriffsmodifizierer **protected**.

Wenn eine Methode, ein Property oder eine Klassenvariable protected ist, dann ist der Zugriff nur von innerhalb der Klasse oder von einer abgeleiteten Klasse möglich. protected ist also weniger einschränkend wie private aber erlaubt keinen Zugriff wie public von außerhalb des Objekts.

public	Protected	private
Zugriff von... - innerhalb der Klasse - allen abgeleiteten Klassen - von außen	Zugriff von... - innerhalb der Klasse - allen abgeleiteten Klassen	Zugriff von... - innerhalb der Klasse

Steigende Beschränkung des Zugriffs

Abbildung 21: Übersicht über die drei Zugriffsmodifizierer im Beispiel

Speziell bei Basisklassen ist es oft eine gute Strategie die Setter von Properties auf protected oder sogar auf private einzuschränken.

13.5.3 Vererbung von Methoden

Basisklassen sind nicht nur ideal zum Zusammenfassen von gemeinsamen Properties oder Klassenvariablen verschiedener Klassen, sondern auch von Methoden.

In unserem Beispiel hat die Basisklasse Artikel eine eigene Methode Ausgeben um ihre drei Properties auszugeben, die von den speziellen Klassen der einzelnen Produkte verwendet wird. Mit **base.** kann man sicherstellen, dass man auf die Methode (Klassenvariablen, Properties) der Basisklasse zugreift. Falls die Namen aber ohnehin unterschiedlich sind, kann base. auch weggelassen werden.

```
class Artikel
{
    ...
    protected Artikel(...)

    ...
}

public void Ausgeben()
{
    ...
}

public void Sichern(string file)
{
    ...
}

}

class Notebook : Artikel
{
    ...
    public Notebook( ... ) : base(...)

    ...

    public void Ausgeben()
    {
        base.Ausgeben();
        ...
    }

    public void Speichern(string file)
    {
        Sichern( file );
        ...
    }
}
```

Abbildung 22: Artikel-Basisklasse und abgeleitete Notebook-Klasse

13.6 Polymorphismus

Auf Objekte einer abgeleiteten Klasse, z. B. Monitor, können zum einen durch Variablen des Typs der abgeleiteten Klasse zugegriffen werden.

```
Monitor bildschirm = new Monitor( ... );
bildschirm.Speichern( ... );
```

Zusätzlich ist es möglich jede Basisklasse der Klasse des Objekts zu verwenden. Die Zuweisung zu einer Variablen der Basisklasse erfolgt hierbei durch einen impliziten Cast.

```
Monitor bildschirm = new Monitor( ... );
Artikel art = bildschirm;
```

Die Variable art enthält hier das gleiche Objekt bildschirm, hat aber nur den Zugriff auf die Methoden der Basisklasse.

Das Casten einer abgeleiteten Klasse auf eine Basisklasse nennt sich – da man in der Klassenhierarchie nach oben wandert – auch **Up-Cast**. Der Aufruf von Speichern führt lediglich die Methode

Speichern der Basisklasse durch und speichert daher auch nur die Eigenschaften die von `bildschirm` in der Basisklasse gespeichert sind.

```
art.Speichern( ... );
```

Möchte man dagegen die Methode Speichern der abgeleiteten Klasse Monitor aufrufen um auch alle spezifischen Eigenschaften eines Monitors abzuspeichern, dann gibt es die Möglichkeit die Variable `art` mit Hilfe des `as`-Operators auf den Typ der abgeleiteten Klasse Monitor zu casten.

```
Notebook m = art as Notebook;
```

oder (in einer veralteten Schreibweise)

```
Notebook m = (Artikel) art;
```

Da man bei diesem Cast in der Klassenhierarchie nach unten wandert, spricht man auch von einem **Down-Cast**. Ein solcher Down-Cast darf aber nur dann durchgeführt werden, wenn das Objekt tatsächlich von dem entsprechenden Typ ist. Das kann durch den `is`-Operator getestet werden.

```
bool isteinnotebook = art is Notebook;
```

Der `is`-Operator erlaubt es also ein Objekt auf seinen Typ zu testen und dann entsprechend zu casten.

```
if ( art is Notebook )
    ( art as Notebook ).Speichern( ... );
```

Wenn eine Basisklasse viele abgeleiteten Klassen hat, so ist das Herausfinden des Typs eines Objekts mit `is` und das Casten mit `as` aber äußerst mühsam und eine Menge unnötiger Code.

Außerdem muss immer, wenn eine neue Klasse von der Basisklasse abgeleitet wird, der Code an dieser Stelle um eine weitere Abfrage ergänzt werden. Man stelle sich vor, dass es 300 abgeleitete Klassen gibt. Dann müsste man 300 Abfragen einbauen und das für jede Methode, die irgendwo im Programm aufgerufen werden soll.

Daher bieten C# und alle anderen objekt-orientierten Sprachen eine Möglichkeit, dass die „richtigen“ Methoden (jene entsprechend des tatsächlichen Typs des Objekts ganz einfach aufgerufen werden können.

13.7 Eine virtuelle Methode im Beispiel

Wenn eine Methode in einer Basisklasse mit `virtual` deklariert wird, dann gibt die Basisklasse diese Methode für abgeleitete Klassen zum **Überschreiben** frei. D. h. jede abgeleitete Klassen kann ihre eigene Version der Methode enthalten. In der abgeleiteten Klasse muss die Methode mit `override` deklariert werden.

In unserem konkreten Beispiel der Klassen Notebook die von der Basisklasse Artikel abgeleitet ist, sieht die Methode Speichern in der Basisklasse und der abgeleiteten Klasse folgendermassen aus:

```
class Artikel
{
    ...
    public virtual void Speichern(...)
    {
        ...
    }
}

class Notebook : Artikel
{
    ...
    public override void Speichern(...)
    {
        ...
    }
}
```

Abbildung 23: Virtuelle Methode in der Artikel-Basisklasse

Die statische Methode Speichern in Program.cs enthält anstelle der if-else-Kaskade der Typabfragen des Objekts einen einfachen Aufruf der Methode Speichern.

```
art.Speichern();
```

Da die Methode in der Basisklasse mit `virtual` definiert ist, wird beim Aufruf zuerst der Typ des Objekts von `art` ermittelt (dieser ist einer der vier abgeleiteten Klassen) und die entsprechende Methode in der entsprechenden Klasse aufgerufen.

13.8 Eine abstrakte Methode im Beispiel

Dasselbe gilt falls die Methode in der Basisklasse mit `abstract` definiert ist. `virtual` und `abstract` sind nur insofern unterschiedlich, dass Methoden die `abstract` sind, von abgeleiteten Klassen immer überschrieben werden **müssen**.

Die Methoden haben auch gar keine Implementierung. Z. B.

```
abstract class Artikel
{
    ...
    public abstract void Speichern(); // muss überschrieben werden
    ...
}
```

13.9 Merkregel für `virtual` und `abstract`

Es ist essentiell, den Unterschied zwischen virtuellen und abstrakten Methoden zu kennen.

Eine Methode in einer Basisklasse ist <code>virtual</code> .	Eine Methode in einer Basisklasse ist <code>abstract</code> .
Jede abgeleitete Klasse KANN diese Methode überschreiben.	Jede abgeleitete Klasse MUSS diese Methode implementieren.
Überschreiben ist dann sinnvoll, wenn die abgeleitete Klasse eine andere Implementierung als die der Basisklasse benötigt.	Das ist dann sinnvoll, wenn EntwicklerInnen gezwungen werden sollen, bestimmte Methoden in abgeleiteten Klassen zu implementieren.
Vergisst eine EntwicklerIn eine virtuelle Methode zu überschreiben, kompiliert das Programm trotzdem.	Vergisst eine EntwicklerIn eine abstrakte Methode zu überschreiben, kompiliert das Programm nicht.

13.9.1 Virtuelle Methoden

Eine Basisklasse implementiert eine Methode dann `virtual`, wenn es eine Implementierung gibt, die die meisten abgeleiteten Klassen benützen möchten.

Jede abgeleitete Klasse kann die Implementierung der Basisklasse überschreiben, indem sie selbst die Methode mit `override` implementiert.

Beispiel: In einer Klassenhierarchie in einem Webshop haben fast alle Produkte dieselbe Umsatzsteuer von 20 Prozent. Nur einige wenige haben 10 Prozent. Dann ist es sinnvoll, dass die Basisklasse eine Methode virtuell implementiert, die 20 Prozent zurückgibt.

```
class Product
{
    public virtual int GetTaxInPercent()
    {
        return 20;
    }
}
```

Falls ein Produkt nur 10 Prozent Umsatzsteuer benötigt, überschreibt dessen Produktklasse die Methode einfach.

```
class ProductA
{
    public override int GetTaxInPercent()
    {
        return 10;
    }
}
```

und "schaltet" damit die Implementierung der Basisklasse quasi aus.

13.9.2 Abstrakte Methoden

Eine Basisklasse implementiert eine Methode dann `abstract`, wenn sie bewusst keine Implementierung zur Verfügung stellt, die andere Klassen verwenden sollen. Die Methode hat auch gar keinen Codeblock {}.

Jede abgeleitete Klasse muss die Implementierung der Basisklasse unbedingt überschreiben, indem sie selbst die Methode mit `override` implementiert.

Beispiel: In einer Klassenhierarchie in einem Webshop haben alle Produkte eine unterschiedliche Verpackungsgröße. Dann ist es sinnvoll, dass die Basisklasse eine Methode abstrakt definiert und keine Implementierung zur Verfügung stellt.

Beachte, dass jede Klasse die zumindest eine abstrakte Methode hat, selbst abstrakt definiert sein muss.

Eine abstrakte Klasse kann nicht direkt mit `new` instanziiert werden. Das ist ein gewünschtes Verhalten, da die Basisklasse oft nur als Container für Gemeinsamkeiten der abgeleiteten Klassen fungieren soll.

C# Abstrakte Basisklasse

```
abstract class Product
{
    public abstract double GetSize();
}
```

Jedes Produkt muss nur diese Methode überschreiben. Ansonsten würde das Programm nicht kompilieren.

C# ProductB und ProductC

```
class ProductB : Product
{
    public override double GetSize()
    {
        return 1.5;
    }
}

class ProductC : Product
{
    public override double GetSize()
    {
        return 2.0;
    }
}
```

und "schaltet" damit die Implementierung der Basisklasse quasi aus.

13.10 Beispiel zur Vererbung: Pizzas, Kebaps und Getränk

Anmerkung: Dieses Beispiel wird in fünf Videos des YouTube-Channels [Coding Kurzgeschichten](#) in fünf Teilen Schritt für Schritt vorprogrammiert und erklärt.



Die Videos zu diesem Beispiel erklären das gesamte Design der Klassenhierarchie und alle Überlegungen bezüglich virtueller und abstrakter Methoden. Es gibt eine eigene Playlist ([Zur Playlist](#)) dafür.

Der Programmcode zu den einzelnen Teilen ist im Internet auf GitHub frei verfügbar.



Vererbung von Daten - Pizza, Kebap und Getränke (Teil 1)

Coding Kurzgeschichten

9:03

Programmcode: <https://github.com/slogslog/Coding-Kurzgeschichten/tree/master/020%20Pizza%20Kebap%20und%20Getränke%201>



ToString() - Pizza, Kebap und Getränke (Teil 2)

Coding Kurzgeschichten

9:36

Programmcode: <https://github.com/slogslog/Coding-Kurzgeschichten/tree/master/021%20Pizza%20Kebap%20und%20Getränke%202%20ToString>

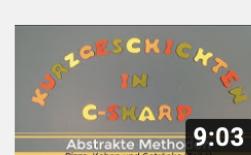


Virtuelle Methoden - Pizza, Kebap und Getränke (Teil 3)

Coding Kurzgeschichten

4:27

Programmcode: <https://github.com/slogslog/Coding-Kurzgeschichten/tree/master/022%20Pizza%20Kebap%20und%20Getränke%203%20Virtuelle%20Methoden>



Abstrakte Methoden - Pizza, Kebap und Getränke (Teil 4)

Coding Kurzgeschichten

9:03

Programmcode: <https://github.com/slogslog/Coding-Kurzgeschichten/tree/master/023%20Pizza%20Kebap%20und%20Getränke%204%20Abstrakte%20Methoden>



Polymorphie - Pizza, Kebap und Getränke (Teil 5)

Coding Kurzgeschichten

6:46

Programmcode: <https://github.com/slogslog/Coding-Kurzgeschichten/tree/master/024%20Pizza%20Kebap%20und%20Getränke%205%20Polymorphie>

Video 5: Coding Kurzgeschichten-Videos des Beispiels "[Pizza, Kebap und Getränke](#)"

Beispiel: Eine Imbissstube verkauft Pizzas, Kebaps und Getränke.

Erstelle ein sinnvolles Klassendesign mit Hilfe von Vererbung (Inheritance), wenn die folgenden Eigenschaften und Methoden zu implementieren sind.

	Pizza	Kebap	Getränk
Eigenschaften:	Name	Name	Name
	Verkaufspreis	Verkaufspreis	Verkaufspreis
	Größe (klein/groß)		Volumen (0,25 oder 0,33 oder 0,5 Liter)
Zubereitungszeit:	12 Minuten	3 Minuten	3 Minuten
Kosten der Zutaten:	Klein: 3 Euro Groß: 4,5 Euro	2 Euro	0,5 l: 1,5 Euro 0,33 l: 1,2 Euro 0,25 l: 0,8 Euro
Gewinn:	$= \frac{\text{Bruttopreis exklusive Umsatzsteuer}}{1 + \frac{\text{Umsatzsteuer(in \%)} }{100}} - \text{Kosten der Zutaten} - \text{Energiekosten}$ <p>Energiekosten = 0,08 * Zubereitungszeit Bruttopreis = Verkaufspreis Umsatzsteuer = 20 %</p>		

Lösung:

1. Schritt:

Erstelle alle drei Klassen mit allen Eigenschaften, den Konstruktoren und ToString()

2. Schritt:

Verwende Vererbung von Daten und erstelle eine Basisklasse mit einem Konstruktor und ToString(). Rufe in den Konstruktoren den Basiskonstruktor auf.

<https://www.youtube.com/watch?v=yWUACeogLOU>

Optimiere ToString().

<https://www.youtube.com/watch?v=fmrZcN9fb4E>

3. Schritt:

Erstelle in der Basisklasse eine Methode für die Zubereitungszeit und gib 3 Min zurück.

Mache die Methode virtuell und überschreibe sie in allen notwendigen Klassen.

<https://www.youtube.com/watch?v=Ew7k0rc8Cgl>

4. Schritt:

Erstelle in der Basisklasse eine Methode für die Kosten der Zutaten.

Mache die Methode abstract und überschreibe sie in allen notwendigen Klassen.

<https://www.youtube.com/watch?v=p0nYmtB7HPw>

5. Schritt:

Implementiere in der Basisklasse eine Methode für den Gewinn. Der fertige Code:

C# Basisklasse Product

```
abstract class Product
{
    public string Name { get; private set; }
    public decimal PurchasePrice { get; private set; }

    public Product(string name, decimal purchasePrice)
    {
        Name = name;
        PurchasePrice = purchasePrice;
    }

    public virtual int GetMinutes()
    {
        return 3;
    }

    public abstract decimal GetIngredientsCosts();

    public override string ToString()
    {
        return $"{Name} {PurchasePrice:C2}";
    }
}
```

C# Klasse Pizza

```
class Pizza : Product
{
    public bool IsLarge { get; private set; }

    public Pizza(string name, decimal purchasePrice, bool isLarge)
        : base(name, purchasePrice)
    {
        IsLarge = isLarge;
    }

    public override int GetMinutes()
    {
        return 12;
    }

    public override string ToString()
    {
        string size = IsLarge ? "large" : "small";
        return $"{base.ToString()} ({size})";
    }

    public override decimal GetIngredientsCosts()
    {
        return IsLarge ? 4.5m : 3;
    }
}
```

C# Klasse Kebap

```
class Kebap : Product
{
    public Kebap(string name, decimal purchasePrice)
        : base(name, purchasePrice)
    {
    }

    public override decimal GetIngredientsCosts() => 2m;
}
```

C# Klasse Drink (für Getränk)

```
enum Volume
{
    _05l,
    _033l,
    _025l
}

class Drink : Product
{
    public Volume Volume { get; private set; }

    public Drink(string name, decimal purchasePrice, Volume volume)
        : base(name, purchasePrice)
    {
        Volume = volume;
    }

    public override string ToString()
    {
        string amount = "";
        switch(Volume)
        {
            case Volume._025l: amount = "0,25 l"; break;
            case Volume._033l: amount = "0,33 l"; break;
            case Volume._05l: amount = "0,5 l"; break;

            default:
                System.Diagnostics.Debug.Assert(false, "Invalid volume!");
                break;
        }

        return $"{base.ToString()} ({amount})";
    }

    public override decimal GetIngredientsCosts()
    {
        switch(Volume)
        {
            case Volume._05l: return 1.5m;
            case Volume._033l: return 1.2m;
            case Volume._025l: return 0.8m;
        }

        System.Diagnostics.Debug.Assert(false, "Invalid volume!");

        return decimal.MinValue;
    }
}
```

C# Main mit drei Instanzen

```
static void Main(string[] args)
{
    Console.OutputEncoding = Encoding.Unicode;

    Product pizza = new Pizza("Venezia", 6.5m, true);
    Product kebap = new Kebap("Ardana Kebap", 4.55m);
    Product drink = new Drink("Cola", 3.5m, Volume._05l);

    List<Product> products = new List<Product>();
    products.Add(pizza);
    products.Add(kebap);
    products.Add(drink);
```

```

foreach ( Product p in products )
{
    Console.WriteLine(p);
    Console.WriteLine($" Preparation time: {p.GetMinutes()} min");
    Console.WriteLine($" Ingredients price: {p.GetIngredientsCosts():C2}");
}
}

```

Programmausgabe:

Venezia €6.50 (large)
 Preparation time: 12 min
 Ingredients price: €4.50

Ardana Kebap €4.55
 Preparation time: 3 min
 Ingredients price: €2.00

Cola €3.50 (0,5 l)
 Preparation time: 3 min
 Ingredients price: €1.50

13.11 Beispiel Polymorphie: Gemma in den Supermarkt

Im vorhergehenden Abschnitt wurde eine Klassenhierarchie angelegt. Mithilfe der Polymorphie ist es möglich, dass mehrere Instanzen unterschiedlicher Produkte in ein und derselben Liste gespeichert werden kann.

Es zeigt, wie man eine Supermarktklasse anlegt, die eine Liste von Produkten speichert, die alle in einer Klassenhierarchie durch Vererbung realisiert sind. Die Klassen sind relativ einfach, wichtiger ist der Aufbau der Klasse Supermarkt.

Auch dieses Beispiel ist in einem Video des YouTube-Channels [Coding Kurzgeschichten](#) vollständig erklärt und vorprogrammiert.



Programmcode: <https://github.com/slogslog/Coding-Kurzgeschichten/tree/master/014%20Vererbung>

Video 6: Coding Kurzgeschichten-Video des Beispiels "[Gemma in den Supermarkt](#)"

Der vollständige Programmcode besteht zunächst aus der Basisklasse Produkt, den (sehr einfachen) Klassen von konkreten Produkten, wie Lebensmittel, Hygieneartikel, Tierfutter und Waschmittel.

Diese Klassen bilden die Klassenhierarchie der Produkte.

C# Basisklasse Produkt

```
abstract class Produkt
{
    public string Name { get; private set; }
    public int Stückzahl { get; private set; }

    public Produkt(string name, int stückzahl)
    {
        Name = name;
        Stückzahl = stückzahl;
    }

    public override string ToString()
    {
        return $"{Stückzahl} Stk. {Name}";
    }
}
```

C# Klasse für Lebensmittel

```
// Alle Produkte die Menschen essen können
// Lebensmittel wird von der Basisklasse Produkt abgeleitet.
// Lebensmittel ist eine abgeleitete Klasse.
class Lebensmittel : Produkt
{
    public Lebensmittel(string name, int stückzahl)
        : base(name, stückzahl)
    {

    }

    public override string ToString()
    {
        return base.ToString() + " (Ist ein Lebensmittel)";
    }
}
```

C# Klasse Hygieneartikel

```
class Hygieneartikel : Produkt
{
    public Hygieneartikel(string name, int stückzahl)
        : base(name, stückzahl)
    {

    }

    public override string ToString()
    {
        return base.ToString() + " (Ist ein Hygieneartikel)";
    }
}
```

C# Klasse Tierfutter

```
class Tierfutter : Produkt
{
    public Tierfutter(string name, int stückzahl)
        : base(name, stückzahl)
    {

    }

    public override string ToString()
    {
        return base.ToString() + " (Ist ein Tierfutter)";
    }
}
```

C# Klasse Waschmittel

```
class Waschmittel : Produkt
{
    public Waschmittel(string name, int stückzahl)
        : base(name, stückzahl)
    {

    }

    public override string ToString()
    {
        return base.ToString() + " (Ist ein Waschmittel)";
    }
}
```

13.11.1 Supermarktklasse mit einer Liste aller Produkte

Die Klasse Supermarkt ist der wesentliche Teil dieses Beispiels. Im Video wird schrittweise erklärt, wie es mit Polymorphie möglich ist, alle Instanzen verschiedener Produkte in einer solchen Liste zu speichern.

Polymorphie ist die Bezeichnung für den Umstand, dass jede Instanz einer der speziell abgeleiteten Produktklassen auch in einer Variable mit dem Typ des Basistyps gespeichert werden kann und nicht nur in einer Variable des Typs der Instanz.

Daher ist es auch möglich, viele solcher Instanzen in einer Liste des Typs `List<Produkt>` abzuspeichern.

C# Klasse Supermarkt

```
class Supermarkt
{
    #region Daten
    public string Name { get; private set; }
    private List<Produkt> Produkte { get; set; } = new List<Produkt>();
    #endregion

    #region Konstruktoren (ctor)
    public Supermarkt(string name) { Name = name; }
    #endregion

    #region Methoden
    // Fügt ein Produkt hinzu
    public void Hinzufügen(Produkt p)
    {
        Produkte.Add(p);
    }

    public override string ToString()
    {
        return $"{Name}:\n{String.Join("\n", Produkte)}";
    }
}
```

```

// Gibt alle Produkte zurück, die Lebensmittel sind.
public List<Lebensmittel> Lebensmittel
{
    get
    {
        List<Lebensmittel> lebensmittel = new List<Lebensmittel>();

        foreach (Produkt p in Produkte)
        {
            if (p is Lebensmittel)
            { // Polymorphie
                lebensmittel.Add(p as Lebensmittel);
                // oder lebensmittel.Add((Lebensmittel)p);
            }
        }

        return lebensmittel;
    }
}

public List<Waschmittel> Waschmittel
{
    get
    {
        List<Waschmittel> lebensmittel = new List<Waschmittel>();

        foreach (Produkt p in Produkte)
        {
            if (p is Waschmittel wm )
            { // Polymorphie
                lebensmittel.Add(wm);
            }
        }

        return lebensmittel;
    }
}
#endregion

```

Ersetzt den as-Operator

Bei konkreten Abfragen nach dem Typ einer Instanz kann der **is**-Operator verwendet werden.

Die Klasse Supermarkt zeigt, wie der **is**- und **as**-Operator eingesetzt werden kann, um den Lautzeittyp (Runtime Type) einer Instanz herauszufinden.

13.11.2 Befüllung einer Liste mit Instanzen

Im Main wird die Liste über eine Methode gefüllt.

Anmerkung: Im nächsten Abschnitt Verbesserte Klasse Supermarkt13.11.2.1 wird gezeigt, wie man sich die zusätzliche Methode Hinzufügen ersparen kann.

C# Main

```
static void Main(string[] args)
{
    // Anlegen des Supermarkts
    Supermarkt sm = new Supermarkt("KaufHierAberZackig");

    // Befüllen des Supermarkts
    sm.Hinzufügen(new Lebensmittel("Ovomaltine 100 g", 2));
    sm.Hinzufügen(new Lebensmittel("Teekanne Lillifee 10 Sackerl", 2));
    sm.Hinzufügen(new Lebensmittel("Teekanne Ostfriesentee 10 Sackerl", 1));
    sm.Hinzufügen(new Lebensmittel("Melitta Harmonie mile 250 g", 1));
    sm.Hinzufügen(new Hygieneartikel("Tempo 100 Stk", 2));
    sm.Hinzufügen(new Waschmittel("Weißer Riese XL", 1));
    sm.Hinzufügen(new Waschmittel("Lenor 5 kg", 1));
    sm.Hinzufügen(new Waschmittel("Ariel 5 kg", 2));
    sm.Hinzufügen(new Lebensmittel("Jodsalz 500 g", 2));
    sm.Hinzufügen(new Lebensmittel("Zwieback Kokos 225 g", 2));
    sm.Hinzufügen(new Lebensmittel("Dr. Oetker Grießbrei", 3));
    sm.Hinzufügen(new Lebensmittel("Dr. Oetker Bistro Baguette Salami", 1));
    sm.Hinzufügen(new Lebensmittel("Iglo Fischstäbchen 15 Stk", 1));
    sm.Hinzufügen(new Lebensmittel("Super Dickmanns", 1));
    sm.Hinzufügen(new Lebensmittel("Brandt Minis Vollmilch Schoko", 1));
    sm.Hinzufügen(new Lebensmittel("Magnum Classic 4 Stk", 1));
    sm.Hinzufügen(new Lebensmittel("Müller's & Mühle Basmati Reis 500g", 1));
    sm.Hinzufügen(new Lebensmittel("Fritt", 2));
    sm.Hinzufügen(new Lebensmittel("Maultaschen 360g", 2)); // Schwäbische Spezialität
    sm.Hinzufügen(new Tierfutter("bunny Kaninchen Traum", 2)); // Kaninchenfutter
    sm.Hinzufügen(new Lebensmittel("Dr. Oetker Extra Gelierzucker 500g", 1));
    sm.Hinzufügen(new Lebensmittel("Oro di Parma Sugo", 2));

    // Ausgabe aller Lebensmittel (=alles was Menschen essen)
    Console.WriteLine(String.Join("\n", sm.Lebensmittel));
    Console.WriteLine();
    Console.WriteLine(String.Join("\n", sm.Waschmittel));
}
```

Programmausgabe

```
2 Stk. Ovomaltine 100 g (Ist ein Lebensmittel)
2 Stk. Teekanne Lillifee 10 Sackerl (Ist ein Lebensmittel)
1 Stk. Teekanne Ostfriesentee 10 Sackerl (Ist ein Lebensmittel)
1 Stk. Melitta Harmonie mile 250 g (Ist ein Lebensmittel)
2 Stk. Jodsalz 500 g (Ist ein Lebensmittel)
2 Stk. Zwieback Kokos 225 g (Ist ein Lebensmittel)
3 Stk. Dr. Oetker Grießbrei (Ist ein Lebensmittel)
1 Stk. Dr. Oetker Bistro Baguette Salami (Ist ein Lebensmittel)
1 Stk. Iglo Fischstäbchen 15 Stk (Ist ein Lebensmittel)
1 Stk. Super Dickmanns (Ist ein Lebensmittel)
1 Stk. Brandt Minis Vollmilch Schoko (Ist ein Lebensmittel)
1 Stk. Magnum Classic 4 Stk (Ist ein Lebensmittel)
1 Stk. Müller's & Mühle Basmati Reis 500g (Ist ein Lebensmittel)
2 Stk. Fritt (Ist ein Lebensmittel)
2 Stk. Maultaschen 360g (Ist ein Lebensmittel)
1 Stk. Dr. Oetker Extra Gelierzucker 500g (Ist ein Lebensmittel)
2 Stk. Oro di Parma Sugo (Ist ein Lebensmittel)

1 Stk. Weißer Riese XL (Ist ein Waschmittel)
1 Stk. Lenor 5 kg (Ist ein Waschmittel)
2 Stk. Ariel 5 kg (Ist ein Waschmittel)
```

13.11.2.1 Verbesserte Klasse Supermarkt

Dass ein Supermarkt eine Liste der Produkte speichert, ist eine mögliche Lösung, allerdings ist es dadurch notwendig viele Methoden, wie z. B. void Hinzufügen(Produkt p) nachzuimplementieren.

Viel einfacher ist es, wenn der Supermarkt selbst eine Liste ist und alle Methoden besitzt die eine Liste besitzt, also auch void Add(Produkt p).

Das ist leicht erreichbar in dem man die Klasse Supermarkt von der Liste ableitet.

C# Klasse Supermarkt abgeleitet von der Liste List<Produkt>

```
class Supermarkt : List<Produkt>
{
    public string Name { get; private set; }

    public Supermarkt(string name) { Name = name; }

    public override string ToString()
    {
        return $"{Name}:\n{String.Join("\n", Produkte)}";
    }
    // Gibt alle Produkte zurück, die Lebensmittel sind.
    public List<Lebensmittel> Lebensmittel
    {
        get
        {
            List<Lebensmittel> lebensmittel = new List<Lebensmittel>();

            foreach (Produkt p in this)
            {
                if (p is Lebensmittel)
                {
                    // Polymorphie
                    lebensmittel.Add(p as Lebensmittel);
                    // oder lebensmittel.Add((Lebensmittel)p);
                }
            }

            return lebensmittel;
        }
    }

    public List<Waschmittel> Waschmittel
    {
        get
        {
            List<Waschmittel> lebensmittel = new List<Waschmittel>();

            foreach (Produkt p in this)
            {
                if (p is Waschmittel wm )
                {
                    // Polymorphie
                    lebensmittel.Add(wm);
                }
            }

            return lebensmittel;
        }
    }
}
```

Ableitung von List<Produkt>
Daher ist kein Property notwendig.

Die Schleife läuft über die eigene Instanz die eine Liste ist.

Die Schleife läuft über die eigene Instanz die eine Liste ist.

Im Main, wird dann einfach die Add-Methode zum Hinzufügen von Instanzen zur Liste – die ja nun der Supermarkt selbst ist.

C# Main

```
static void Main(string[] args)
{
    // Anlegen des Supermarkts
    Supermarkt sm = new Supermarkt("KaufHierAberZackig");

    // Befüllen des Supermarkts
    sm.Add(new Lebensmittel("Ovomaltine 100 g", 2));
    sm.Add(new Lebensmittel("Teekeanne Lillifee 10 Sackerl", 2));
    sm.Add(new Lebensmittel("Teekeanne Ostfriesentee 10 Sackerl", 1));
    sm.Add(new Lebensmittel("Melitta Harmonie mile 250 g", 1));
    sm.Add(new Hygieneartikel("Tempo 100 Stk", 2));
    sm.Add(new Waschmittel("Weißer Riese XL", 1));
    sm.Add(new Waschmittel("Lenor 5 kg", 1));
    sm.Add(new Waschmittel("Ariel 5 kg", 2));
    sm.Add(new Lebensmittel("Jodsalz 500 g", 2));
    sm.Add(new Lebensmittel("Zwieback Kokos 225 g", 2));
    sm.Add(new Lebensmittel("Dr. Oetker Grießbrei", 3));
    sm.Add(new Lebensmittel("Dr. Oetker Bistro Baguette Salami", 1));
    sm.Add(new Lebensmittel("Iglo Fischstäbchen 15 Stk", 1));
    sm.Add(new Lebensmittel("Super Dickmanns", 1));
    sm.Add(new Lebensmittel("Brandt Minis Vollmilch Schoko", 1));
    sm.Add(new Lebensmittel("Magnum Classic 4 Stk", 1));
    sm.Add(new Lebensmittel("Müller's & Mühle Basmati Reis 500g", 1));
    sm.Add(new Lebensmittel("Fritt", 2));
    sm.Add(new Lebensmittel("Maultaschen 360g", 2)); // Schwäbische Spezialität
    sm.Add(new Tierfutter("bunny Kaninchen Traum", 2)); // Kaninchenfutter
    sm.Add(new Lebensmittel("Dr. Oetker Extra Gelierzucker 500g", 1));
    sm.Add(new Lebensmittel("Oro di Parma Sugo", 2));

    // Ausgabe aller Lebensmittel (=alles was Menschen essen)
    Console.WriteLine(String.Join("\n", sm));
    Console.WriteLine();
    Console.WriteLine(String.Join("\n", sm.Waschmittel));
}
```

Auch bei String.Join ist die gesamte Supermarkt-Instanz ausreichend.

Programmausgabe ist wie in Abschnitt 13.11.1

14 Compiler Types und Runtime Types

Dieses Kapitel erläutert die für ProgrammieranfängerInnen nicht ganz trivialen Begriffe. Es ist die Kenntnis von Vererbung und Polymorphie erforderlich.

14.1 Begriffserklärungen

Für das Verständnis was zur Laufzeit durch eine Klassenhierarchie passiert, ist es essentiell zwischen Compilertypes und Runtimetypes zu unterscheiden.

Compiler Types werden auch statische Typen, Runtime Types auch dynamische Typen genannt.

Der Compiler Type ist der Typ der Variable (genaugenommen der Referenzvariable).

Der Runtime Type ist der Typ der Instanz, die von der Variablen zur Laufzeit gespeichert wird.

Es gilt:

<Compiler Type> variable = Instanz mit einem bestimmten <Runtime Type>

Verwendet man die einfache Klassenhierarchie des Beispiels in Abschnitt 13.3 "Ein Codebeispiel zur Vererbung von Methoden", bestimmt sich für jede Zuweisung der Compiler Type und Runtime Type.

Zuweisung	Compiler Type der Variablen	Runtime Type der Variablen
Person a = new Person("Franz Huber", 50);	Person	Person
Person a = new Pupil("Kevin Weber", 17, "4EHIF");	Person	Pupil
		Dieser Cast ist ein Upcast , weil die Instanz einer abgeleiteten Klasse auf eine Basisklasse "hinaufecastet" wird.
Object a = new Person("Franz Huber", 50);	Object	Person
Person a = new Person("Franz Huber", 50);		
Person b = (Person)a;	Person	Person
Person b = new Person("Franz Huber", 50); Pupil c = (Pupil) b; <i>// Kein Compile Error, aber ein Runtime Error!</i>		Dieser Cast ist für den Compiler gültig, da b ein Pupil enthalten sein könnte. Zur Laufzeit kommt es allerdings zu einem Absturz, da der Runtime Type von b Person ist und daher nicht in ein Pupil gecastet werden kann, da in der Person-Instanz hierfür Information fehlt! Dies ist ein Downcast .
Pupil a = new Person("Franz Huber", 50); <i>// Compiler error!</i>		Diese Zuweisung ist ungültig! Da der Compiler Type Pupil nur Instanzen des Typs Pupil oder abgeleitete Klassen speichern kann. Person ist aber ein Basistyp von Pupil und so eine Instanz kann daher nicht in Pupil gespeichert werden.

Pupil a = new Pupil("Kevin Weber", 17, "4EHIF");	Pupil	Pupil
Object a = new Pupil("Kevin Weber", 17, "4EHIF");	Object	Pupil

14.2 Upcasts und Downcasts

Upcasts und Downcasts sind Begriffe, die innerhalb einer Klassenhierarchie gebraucht werden, um Typumwandlungen zu beschreiben.

Ein Upcast ist dabei eine Typumwandlung von einer abgeleiteten Klasse zu einer ihrer Basisklassen.
Ein Downcast eine Typumwandlung eines Basistyps auf eine der abgeleiteten Klassen.

Die Begriffe Up und Down erklären sich hierbei dadurch, dass Basistypen in einer Klassenhierarchie immer oben eingezeichnet werden, während abgeleitete Klassen nach unten hin eingezeichnet werden.

Wir betrachten zur Erklärung eine mehrstufige Klassenhierarchie:

C#

```

class A {
}

class B : A {
}

class C : B {
}

class D : A {
}

public class Main {

    public static void Main(String[] args) {
        // a, b, c and d have all a specific compiler type (static type).
        A a = null;
        B b = null;
        C c = null;
        D d = null;

        // Compiler
        a = b;          // Ok for the compiler, since B is an A.
        b = a;          // Compiler error, since A is not a B!
        b = (B)a;       // Ok for the compiler, since a could be a B.
        c = (C)d;       // Compiler error, since d can never be a C!

        // Runtime system
        a = new C(); // Ok at runtime, since C is an A and can be stored in a.
        d = (D)a;   // Ok for the compiler, since a could be a D
                     // But runtime error, since a stores a C and C is not a D!

        b = (B)a;   // Ok for the compiler, since a could be a B.
                     // Ok at runtime too, since a stores a C and C is a B.
    }
}

```

Die Variable a hat den Compiler Type A.

Zur Laufzeit kann diese Variable a, aber durchaus Instanzen (Objekte) verschiedenen Typs (Runtime Types) speichern.

Der Compiler arbeitet immer mit Compiler Types (Statischen Typen)!

Zur Laufzeit sind die Runtime Types entscheidend. Durch Typinkompatibilitäten kann es während der Programmausführung zu Programmabstürzen kommen!

Wird eine Methode eines Objekts aufgerufen die in der Basisklasse, weder `virtual` noch `abstract` definiert ist, so wird immer die Methode ausgeführt, die dem **Typ der Variable** des Aufrufs entspricht, also dem sogenannten **statischen Typ** oder Compile Type.

Wird eine Methode aufgerufen die in der Basisklasse als `virtual` oder `abstract` deklariert ist, so wird immer die Methode ausgeführt, die dem **Typ des Objekts** entspricht, also dem sogenannten Laufzeittyp oder Runtime Type.

14.2.1 Upcast

Wenn ein abgeleiteter Typ in einer Variablen eines Basisklassentyps gespeichert wird, spricht man von einem **Upcast**.

Beispiel für einen Upcast:

Der Code

```
A a;  
a = new B();  
a.Method();
```

kann nur dann erfolgreich kompiliert werden, wenn die Klasse A eine Methode `Method()` hat, da a den Compiler Type A hat.

In diesem konkreten Code speichert man also in einem Basisklassentyp A eine Instanz des abgeleiteten Typs B (Runtime Type). Es ist also ein **Upcast**.

Man benötigt für Upcasts nie einen expliziten Cast (A) weil zur Laufzeit nie ein Fehler auftreten kann.

```
A a;  
a = (A) new B();
```

14.2.2 Downcast

Wenn ein Basisklassentyp einem abgeleiteten Typ zugewiesen wird, spricht man von einem **Downcast**. Downcasts sind risikanter, da sie zur Laufzeit scheitern können, falls die gespeicherte Instanz gar nicht vom Basisklassentyp ist. Falls ein Downcast scheitert, wird eine Exception geworfen.

Beispiel für einen Downcast:

```
A a;  
B b;
```

```

a = new B(); // Upcast
b = (B) a; // Downcast works fine at runtime, since a stores a B
a = new B();
b = (D) a; // Downcast throws an exception at runtime,
// since a stores no base class of B!

```

14.3 Virtuelle Methoden

Eine Methode kann in C# in einer abgeleiteten Klasse nur überschrieben werden, wenn sie mit dem Schlüsselwort **virtual** deklariert ist.

```

class A
{
    public virtual char getID() { return 'A'; }
}

class B : A
{
    public override char getID() { return 'B'; }

    public char getID(bool select)
    {
        return select ? getID() : base.getID();
    }
}

class Program
{
    static void Main(string[] args)
    {
        A a = new A();
        B b = new B();

        Console.WriteLine(a.getID());
        Console.WriteLine(b.getID());

        Console.WriteLine(b.getID(true));
        Console.WriteLine(b.getID(false));
    }
}

```

Ausgabe:

A
B
B
A

In C# können Klassenmethoden nicht überschrieben werden. Sie können nur mit dem Schlüsselwort **new** neu implementiert werden.

14.4 Abstrakte Methoden

In C# ist es möglich in Basisklassen abstrakte Methoden zu deklarieren, um sie in abgeleiteten Klassen zu implementieren.

C#

```
abstract class A
{
    public abstract char getID();
}

class B : A
{
    public override char getID() { return 'B'; }
}

class C : A
{
    public override char getID() { return 'C'; }
}

class Program
{
    static void Main(string[] args)
    {
        // A a = new A(); Compile error!
        A b = new B();
        A c = new C();

        Console.WriteLine(b.getID());
        Console.WriteLine(c.getID());
    }
}
```

Ausgabe:

B
C

Folgende Punkte sind zu beachten:

- Wenn eine Klasse zumindest eine abstrakte Methode enthält, muss sie selbst abstrakt sein.
- Eine Klasse kann abstrakt sein, ohne dass sie eine abstrakte Methode enthält.
- Eine abstrakte Methode muss in den abgeleiteten Klassen implementiert werden.

14.5 Methodensuche zur Laufzeit

Wir verändern das Codebeispiel von Abschnitt 14.3 Virtuelle Methoden ein wenig und verändern dabei das Main so, dass alle drei erzeugten Instanzen im Compiler Type A gespeichert werden.

Die Überlegungen in diesem Abschnitt gelten ebenso für abstrakte Methoden.

Beispiel:

```
class A
{
    public virtual char GetID() { return 'A'; }
}

class B : A
{
    public override char GetID() { return 'B'; }
}
```

```

class C : A
{
    public override char GetID() { return 'C'; }
}

class Program
{
    static void Main(string[] args)
    {
        A a = new A();
        A b = new B();
        A c = new C();

        Console.WriteLine(a.GetID());
        Console.WriteLine(b.GetID());
        Console.WriteLine(c.GetID());
    }
}

```

Ausgabe:

A
B
C

Der Code im Main kompiliert erfolgreich, da die Klasse A die Methode GetID() besitzt. A ist der Compiler Type (Statische Typ) aller drei Variablen a, b und c.

Während der Programmausführung, wird für den Aufruf b.GetID() zunächst der Runtime Type von b bestimmt. Dieser ist B, da dies der Typ der Instanz ist, die in b gespeichert ist. Daher wird dann die Methode GetID () der Klasse B aufgerufen und der Character 'B' ausgegeben.

Diese Suche nach der richtigen Methode anhand des Runtime Types, kostet bei der Ausführung Zeit.

Der Compile Type (Statischer Typ) bestimmt welche Methoden für eine Instanz aufgerufen werden können.

Die Überprüfung erfolgt zur Compilezeit.

Der Runtime Type (Dynamischer Typ) bestimmt welche Methode konkret ausgeführt wird.

Die Ermittlung dieser Methode erfolgt zur Laufzeit.

15 Nonstatic versus Static Methoden

Statische Methoden können aufgerufen werden, ohne eine Instanz der Klasse in der die Methode implementiert ist, angelegt werden muss.

Statische Methoden sind weitaus seltener, da sie von der Implementierung her nicht an eine bestimmte Klasse gebunden sind. Sie werden in jene Klasse implementiert, in die sie thematisch am besten passen.

Instanzmethoden, also alle Methoden bei denen das Schlüsselwort **static** nicht voransteht, sind dagegen an die Klasse gebunden, in denen sie definiert sind, da sie Instanzvariablen oder Properties der Klasse oder einer Basisklasse verwenden.

Instanzmethoden können nur aufgerufen werden, wenn man eine Instanz dieser Klasse hat.

Statische Methoden können auf Klassenvariablen natürlich zugreifen, da diese Variablen ja selbst statisch definiert sind.

Instanzmethoden	Statisch
Nichtstatische Methoden können auf Instanzvariablen oder Property's zugreifen!	Statische Methoden können NICHT auf Instanzvariablen oder Property's zugreifen!
<pre>class Bruch { public int Z { get; private set; } public int N { get; private set; } public Bruch Multiplizieren(Bruch b) { Bruch result = new Bruch(); result.Set(Z * b.Z, N * b.N); return result; } ... }</pre>	<pre>class Bruch { public int Zähler { get; private set; } public int Nenner { get; private set; } static public Bruch Multiplizieren(Bruch a, Bruch b) { Bruch result = new Bruch(); result.SetZähler(a.Z * b.Z); result.SetNenner(a.N * b.N); return result; } ... }</pre>
Methodenaufruf: Instanz. und Methodename Bruch b1 = = new Bruch(); Bruch b2 = = new Bruch(); ... Bruch ergebnis = b1.Multiplizieren(b2);	Methodenaufruf: Klassenname. und Methodename Bruch b1 = = new Bruch(); Bruch b2 = = new Bruch(); ... Bruch ergebnis = Bruch.Multiplizieren(b1, b2);

Falls eine Methode statisch definiert werden kann, weil sie auf keine Instanzvariablen oder Properties zugreifen, sollte das Schlüsselwort **static** unbedingt vorangestellt werden, weil es für das Verständnis eines Codes sehr zweckmäßig ist.

15.1 Codebeispiel: Klassenvariable als Instanzzähler

Instanzmethoden (instance methods, oft nur als Methoden bezeichnet) können nur über eine Instanz (ein Objekt) der Klasse aufgerufen werden.

Innerhalb der Instanz kann über das Schlüsselwort `this` auf die eigene Instanz zugegriffen werden. Das Schlüsselwort `this` ist nur bei Namenskonflikten (wie im Konstruktor) oder bei Constructor Chaining notwendig.

Klassenmethoden (class methods) können auch ohne eine Instanz, nur durch Angabe des Klassennamens aufgerufen werden.

C# Person.cs

```
using System;

namespace AT_HTLDonaustadt_SLOG
{
    class Person
    {
        private static int counter; // class variable (set to 0) to count the instances
        private int identifier; // instance variable
        public string FirstName { get; private set; } // property
        public string LastName { get; private set; } // property

        // instance method (Konstruktor)
        public Person(string firstname, string lastname)
        {
            FirstName = firstname;
            LastName = lastname;

            identifier = counter;
            counter++;
        }

        public void Rename(String newLastName) // instance method
        {
            LastName = newLastName;
        }

        public static int GetInstanceCount() // class method
        {
            return counter;
        }

        public override string ToString()
        {
            return $"{FirstName} {LastName} ({ID{identifier}})";
        }
    }
}
```

Die Klassenvariable `counter` behält über alle Instanzen der Klasse `Person` ihren Wert. Daher können alle Instanzen im Konstruktor durchnummieriert werden.

Program.cs

```
using System;

namespace AT_HTLDonaustadt_SLOG
{
    class Program
    {
        static void Main(string[] args)
        {
            Person p1 = new Person("Vivan", "Eastwood");
            Console.WriteLine(p1);

            const string NEWNAME = "Westwood";
            Console.WriteLine("Rename {0} to {1}", p1.LastName, NEWNAME);
            p1.Rename(NEWNAME);

            Console.WriteLine(p1);

            Console.WriteLine($"Number of persons: {Person.GetInstanceCount()}");
            Person p2 = new Person("Tom", "Filipovic");
            Console.WriteLine(p2);
            Console.WriteLine($"Number of persons: {Person.GetInstanceCount()}");
        }
    }
}
```

Ausgabe:

```
Vivan Eastwood (ID0)
Rename Eastwood to Westwood
Vivan Westwood (ID0)
Number of persons: 1
Tom Filipovic (ID1)
Number of persons: 2
```

16 Interfaces

Interfaces sind wichtige Hilfsmittel um Klassen um allgemeine Fähigkeiten erweitern zu können.

Gute Links sind <https://www.dotnetperls.com/interface> und
<https://www.codeproject.com/Articles/18743/Interfaces-in-C-For-Beginners>.

Interfaces dienen dazu, Klassen neue einheitliche **Fähigkeiten** zu geben.

In Büchern und Artikeln wird oft beschrieben, dass eine Klasse mit einem Interface einen "Vertrag" abschließt, d. h. dass die Klasse genau die Methoden des Interfaces implementieren muss.

- Interfaces sind in vielerlei Hinsicht ähnlich zu abstrakten Klassen.
Das Interface ist zu der abstrakten Klasse sehr ähnlich.
Auch von Interfaces kann man keine Objekte anlegen, sondern sie dienen nur dazu, dass Klassen sie implementieren. Das ist ganz genauso wie abstrakte Klassen, die nur als Basisklassen verwendet werden können.

<pre>interface IArea { double GetArea();</pre>	ist ähnlich zu	<pre>abstract class Area { abstract double GetArea();</pre>
--	----------------	---

- Eine Klasse kann aber mehrere Interfaces implementieren, aber nur von einer Klasse ableiten.

Die Klasse List im .NET-Framework implementiert beispielsweise drei Interfaces.

```
public class List<T> : IList<T>, IList, IReadOnlyList<T>
```

- Ein Interface wird immer über ein eigenes Keyword definiert, nämlich `interface`.
- Ein Interface kann nie Daten speichern. Es enthält also nie Variablen oder Properties.
Das ist ein Unterschied zu abstrakten Klassen.
- Ein Interface enthält nur Methoden die immer `public` sind. Deshalb wird das Schlüsselwort `public` auch nie hinzugefügt.¹⁸
- Interfaces können keine `static` Methoden und daher keine Operatoren enthalten.

16.1 Codebeispiel 1: Interface zur Flächenberechnung

Gegeben ist ein Interface, das die Fähigkeit der Flächenberechnung darstellt. Die einzige Methode gibt den Wert der Fläche zurück.

C# Ein Interface zur Flächenberechnung

```
// Definiert die "Fähigkeit" eine Fläche zu ermitteln.
interface IArea
{
    double GetArea();
}
```

¹⁸ Ein Interface kann grundsätzlich auch Properties, Events und Indexer enthalten, die dann in der Basisklasse zu implementieren sind (ähnlich zu abstrakten Properties in abstrakten Basisklassen).

Andere Klassen können nur dieses Interface implementieren und die Fähigkeit der Flächenberechnung einbauen. Sie beziehen sich dann alle auf dieselbe Fähigkeit.

C# Eine Dreieck-Klasse und eine Quadrat-Klasse die das Interface IArea implementieren

```
class Triangle : IArea
{
    public double a { get; set; }
    public double b { get; set; }
    public double c { get; set; }

    public Triangle(double a, double b, double c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public double GetArea()
    {   // https://de.wikipedia.org/wiki/Satz_des_Heron
        double s = (a + b + c) / 2;
        return Math.Sqrt(s * (s - a) * (s - b) * (s - c));
    }
}

class Square : IArea
{
    public double a { get; set; }

    public Square(double a) => this.a = a;

    public double GetArea() => a*a;
}
```

Die Methoden können von Instanzen aus ganz normal aufgerufen werden. Das Besondere ist, dass auch für Interfaces Polymorphismus gilt.

In einer Collection können Instanzen von Dreiecken und Quadraten gespeichert werden, weil die beiden Klassen dasselbe Interface implementieren.

C# Main Flächenberechnung von Dreiecken und Rechtecken

```
static void Main(string[] args)
{
    // Anlegen des Supermarkts
    List<IArea> figures = new List<IArea>()
    {
        new Square(2),
        new Triangle(4, 6, 7.2),
        new Square(3.5)
    };

    foreach (var figure in figures)
    {
        Console.WriteLine(figure.GetArea());
    }
}
```

Programmausgabe

```
4
11.999933331481
12.25
```

16.2 Codebeispiel 2: Interface für Tiere

Zwei Klassen, eine für einen Hund und eine für einen Vogel, sollen ein Interface korrekt implementieren.

C#

```
interface Animal
{
    void AnimalSound();
    int GetNumberOfLegs();
}

class Dog : Animal
{
    public void AnimalSound()
    {
        Console.Write("Wau! Wau!");
    }

    public int GetNumberOfLegs()
    {
        return 4;
    }
}

class Bird : Animal
{
    public void AnimalSound()
    {
        Console.Write("Tweet!");
    }

    public int GetNumberOfLegs()
    {
        return 2;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Dog a1 = new Dog();
        Bird a2 = new Bird();
        Console.Write("Animal 1: ");
        a1.AnimalSound();
        Console.WriteLine(" " + a1.GetNumberOfLegs() + " legs");

        Console.Write("Animal 2: ");
        a2.AnimalSound();
        Console.WriteLine(" " + a2.GetNumberOfLegs() + " legs");

        // Assignment to interface type
        Animal a3 = a2;
        a3.AnimalSound();
    }
}
```

Dog implementiert das Interface Animal.

Bird implementiert das Interface Animal ebenfalls.

Programmausgabe:

```
Animal 1: Wau! Wau! 4 legs
Animal 2: Tweet! 2 legs
Tweet!
```

16.3 Interfaces für das Sortieren

Im .NET-Framework sind viele Interfaces definiert, die verwendet werden können. Ein populäres Beispiel sind die beiden Interfaces `IComparable` und `IComparator`, die für die Sortierung von Daten verwendet werden.

16.3.1 Interface `IComparable`

Das Interface `IComparable` stellt genau eine Methode zur Verfügung.

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

Das Interface gibt es auch mit einem Type-Parameter.

```
public interface IComparable<T>
{
    int CompareTo(T obj);
}
```

Der Rückgabewert von `CompareTo` hat folgende Bedeutung:

Rückgabewert < 0, meistens -1	Das Objekt ist kleiner als <code>obj</code> .
Rückgabewert = 0	Das Objekt soll gleich mit <code>obj</code> gereiht werden.
Rückgabewert > 0, meistens +1	Das Objekt ist größer als <code>obj</code> .

Die Methode `CompareTo` wird durch die Sort-Methode sofort aufgerufen, bis Sort die gewünschte Reihenfolge erzeugt hat. Wie oft die Methode aufgerufen wird, hängt von der Reihenfolge in der unsortierten Collection ab.

16.3.1.1 Beispiel: Sortierung von Autos

Um die Autos zu sortieren, ist es notwendig, dass die Klasse deren Instanzen sortiert werden soll, das Interface `IComparable` zu implementieren und die Methode muss den Rückgabewert 0, -1 oder 1 zurückgeben, wie es die Sortierung erfordert.

C# Ausgabe ohne Sortierung	C# Ausgabe mit Sortierung
<pre>class Auto { public string Type { get; private set; } public string Kennzeichen { get; private set; } public int Baujahr { get; private set; } public Auto(string type, string kennz, int jahr) { Type = type; Kennzeichen = kennz; Baujahr = jahr; } public override string ToString() { return \$"{Type,-22} {Kennzeichen,-12} {Baujahr}"; } }</pre>	<pre>class Auto : IComparable<Auto> { ... public int CompareTo(Auto other) { if (other == null) // other ist nicht sinnvoll return 1; // 1. Sortierkriterium: Baujahr (absteigend) int result = -Baujahr.CompareTo(other.Baujahr); if (result == 0) // Beide Baujahre sind gleich. // 2. Sortierkriterium: Type (aufsteigend) result = Type.CompareTo(other.Type); } return result; }</pre>

```

static void Main(string[] args)
{
    List<Auto> autos = new List<Auto>()
    {
        new Auto("VW Passat 1.4", "WU-MAUSI1", 2012),
        new Auto("BMW Z1", "W-34546A", 2010),
        new Auto("Fiat 124 Spider", "L-6778BC", 2012),
        new Auto("Opel Zafira Edition", "MI-17478LI", 2014),
        new Auto("Ford Mondeo Trend", "W-99356K", 2014),
        new Auto("Skoda Octavia Combi", "OW-7305ZR", 2008),
        new Auto("Ford Mondeo Trend", "WU-45513U", 2012),
        new Auto("Alfa Romeo Giulia", "B-ALFA1", 2010)
    };
    Console.WriteLine(String.Join("\n", autos));
}

```

Programmausgabe: (unsortiert)

VW Passat 1.4	WU-MAUSI1	2012
BMW Z1	W-34546A	2010
Fiat 124 Spider	L-6778BC	2012
Opel Zafira Edition	MI-17478LI	2014
Ford Mondeo Trend	W-99356K	2014
Skoda Octavia Combi	OW-7305ZR	2008
Ford Mondeo Trend	WU-45513U	2012
Alfa Romeo Giulia	B-ALFA1	2010

Type
↓

```

static void Main(string[] args)
{
    ...
    autos.Sort();
    Console.WriteLine(String.Join("\n", autos));
}

```

Wenn Sort keinen Parameter hat,
wird CompareTo verwendet.

Programmausgabe: (sortiert)

Opel Zafira Edition	MI-17478LI	2014
Ford Mondeo Trend	W-99356K	2014
Fiat 124 Spider	L-6778BC	2012
Ford Mondeo Trend	WU-45513U	2012
VW Passat 1.4	WU-MAUSI1	2012
Alfa Romeo Giulia	B-ALFA1	2010
BMW Z1	W-34546A	2010
Skoda Octavia Combi	OW-7305ZR	2008

Baujahr
↑

Die Compare-Methode ist so zu implementieren, dass die Sortierkriterien nacheinander angewendet werden. In unserem Beispiel wird zuerst das Baujahr verglichen. Wenn beide gleich groß sind – nur dann liefert CompareTo den Wert 0 zurück – wird das zweite Sortierkriterium angewendet.

Bei einer anderen

16.3.2 Interface IComparer

Das Interface **IComparer** stellt genau eine Methode zur Verfügung, die ähnlich zu **IComparable** ist, allerdings

```

public interface IComparer
{
    int Compare(object x, object y);
}

```

Der Rückgabewert von Compare hat folgende Bedeutung.

Rückgabewert < 0, meistens -1	Die Instanz von x ist kleiner als y.
Rückgabewert = 0	Die beiden Instanzen von x und y sind gleich groß.
Rückgabewert > 0, meistens +1	Die Instanz von y ist kleiner als x.

Die Methode Compare wird durch die Sort-Methode sofort aufgerufen, bis Sort die gewünschte Reihenfolge erzeugt hat. Wie oft die Methode aufgerufen wird, hängt von der Reihenfolge in der unsortierten Collection ab.

16.3.2.1 Beispiel: Sortierung von Autos

Um die Autos zu sortieren, ist es notwendig, dass die Klasse deren Instanzen sortiert werden soll, das Interface `IComparable` zu implementieren und die Methode muss den Rückgabewert 0, -1 oder 1 zurückgeben, wie es die Sortierung erfordert.

C# Ausgabe ohne Sortierung	C# Ausgabe mit Sortierung																																																
<pre>class Auto { public string Type { get; private set; } public string Kennzeichen { get; private set; } public int Baujahr { get; private set; } public Auto(string type, string kennz, int jahr) { Type = type; Kennzeichen = kennz; Baujahr = jahr; } public override string ToString() { return \$"{Type,-22} {Kennzeichen,-12} {Baujahr}"; } } static void Main(string[] args) { List<Auto> autos = new List<Auto>() { new Auto("VW Passat 1.4", "WU-MAUSI1", 2012), new Auto("BMW Z1", "W-34546A", 2010), new Auto("Fiat 124 Spider", "L-6778BC", 2012), new Auto("Opel Zafira Edition", "MI-17478LI", 2014), new Auto("Ford Mondeo Trend", "W-99356K", 2014), new Auto("Skoda Octavia Combi", "OW-7305ZR", 2008), new Auto("Ford Mondeo Trend", "WU-45513U", 2012), new Auto("Alfa Romeo Giulia", "B-ALFA1", 2010) }; Console.WriteLine(String.Join("\n", autos)); }</pre>	<pre>class ComparerBaujahrType : IComparer<Auto> { public int Compare(Auto x, Auto y) { if (x == null) return y == null ? 0 : 1; else if (y == null) return -1; // 1. Sortierkriterium: Baujahr (absteigend) int result = -x.Baujahr.CompareTo(y.Baujahr); if (result == 0) { // Beide Baujahre sind gleich. // 2. Sortierkriterium: Type (aufsteigend) result = x.Type.CompareTo(y.Type); } return result; } } static void Main(string[] args) { ... autos.Sort(new ComparerBaujahrType()); Console.WriteLine(String.Join("\n", autos)); }</pre> <div style="border: 1px solid blue; padding: 5px; width: fit-content;"> <p>Sort kann ein Comparer übergeben werden.</p> </div>																																																
Programmausgabe: (unsortiert) <table style="width: 100%; border-collapse: collapse;"> <tbody> <tr> <td>VW Passat 1.4</td> <td>WU-MAUSI1</td> <td>2012</td> </tr> <tr> <td>BMW Z1</td> <td>W-34546A</td> <td>2010</td> </tr> <tr> <td>Fiat 124 Spider</td> <td>L-6778BC</td> <td>2012</td> </tr> <tr> <td>Opel Zafira Edition</td> <td>MI-17478LI</td> <td>2014</td> </tr> <tr> <td>Ford Mondeo Trend</td> <td>W-99356K</td> <td>2014</td> </tr> <tr> <td>Skoda Octavia Combi</td> <td>OW-7305ZR</td> <td>2008</td> </tr> <tr> <td>Ford Mondeo Trend</td> <td>WU-45513U</td> <td>2012</td> </tr> <tr> <td>Alfa Romeo Giulia</td> <td>B-ALFA1</td> <td>2010</td> </tr> </tbody> </table>	VW Passat 1.4	WU-MAUSI1	2012	BMW Z1	W-34546A	2010	Fiat 124 Spider	L-6778BC	2012	Opel Zafira Edition	MI-17478LI	2014	Ford Mondeo Trend	W-99356K	2014	Skoda Octavia Combi	OW-7305ZR	2008	Ford Mondeo Trend	WU-45513U	2012	Alfa Romeo Giulia	B-ALFA1	2010	Programmausgabe: (sortiert) <table style="width: 100%; border-collapse: collapse;"> <tbody> <tr> <td>Opel Zafira Edition</td> <td>MI-17478LI</td> <td>2014</td> </tr> <tr> <td>Ford Mondeo Trend</td> <td>W-99356K</td> <td>2014</td> </tr> <tr> <td>Fiat 124 Spider</td> <td>L-6778BC</td> <td>2012</td> </tr> <tr> <td>Ford Mondeo Trend</td> <td>WU-45513U</td> <td>2012</td> </tr> <tr> <td>VW Passat 1.4</td> <td>WU-MAUSI1</td> <td>2012</td> </tr> <tr> <td>Alfa Romeo Giulia</td> <td>B-ALFA1</td> <td>2010</td> </tr> <tr> <td>BMW Z1</td> <td>W-34546A</td> <td>2010</td> </tr> <tr> <td>Skoda Octavia Combi</td> <td>OW-7305ZR</td> <td>2008</td> </tr> </tbody> </table>	Opel Zafira Edition	MI-17478LI	2014	Ford Mondeo Trend	W-99356K	2014	Fiat 124 Spider	L-6778BC	2012	Ford Mondeo Trend	WU-45513U	2012	VW Passat 1.4	WU-MAUSI1	2012	Alfa Romeo Giulia	B-ALFA1	2010	BMW Z1	W-34546A	2010	Skoda Octavia Combi	OW-7305ZR	2008
VW Passat 1.4	WU-MAUSI1	2012																																															
BMW Z1	W-34546A	2010																																															
Fiat 124 Spider	L-6778BC	2012																																															
Opel Zafira Edition	MI-17478LI	2014																																															
Ford Mondeo Trend	W-99356K	2014																																															
Skoda Octavia Combi	OW-7305ZR	2008																																															
Ford Mondeo Trend	WU-45513U	2012																																															
Alfa Romeo Giulia	B-ALFA1	2010																																															
Opel Zafira Edition	MI-17478LI	2014																																															
Ford Mondeo Trend	W-99356K	2014																																															
Fiat 124 Spider	L-6778BC	2012																																															
Ford Mondeo Trend	WU-45513U	2012																																															
VW Passat 1.4	WU-MAUSI1	2012																																															
Alfa Romeo Giulia	B-ALFA1	2010																																															
BMW Z1	W-34546A	2010																																															
Skoda Octavia Combi	OW-7305ZR	2008																																															
Type	Baujahr																																																

Die `CompareTo`-Methode ist so zu implementieren, dass die Sortierkriterien nacheinander angewendet werden. In unserem Beispiel wird zuerst das Baujahr verglichen. Wenn beide gleich groß sind – nur dann liefert `CompareTo` den Wert 0 zurück – wird das zweite Sortierkriterium angewendet.

17 Extension Methods

Extensionmethoden erlauben es bestehende Klassen um Methoden zu erweitern, ohne dass deren Programmcode geändert werden muss. Es ist also möglich auch Klassen des .NET-Frameworks wie `string` oder `Random` Methoden hinzuzufügen.

In der Microsoft Doku sind Extension Methods hier beschrieben:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods>

Extensionmethoden werden immer statisch in einer statischen Klasse (mit beliebigen Namen) angelegt werden. Der Typ der erweitert werden soll, muss der erste Parameter der Methode sein muss und diesem muss das Schlüsselwort `this` vorangestellt werden.

Beispiel: Es ist eine String-Methode `MakeTitle` zu schreiben, die einen String in eine Überschrift umwandelt in dem sie alle Buchstaben in Großbuchstaben umwandelt, vorne und hinten mit Spitzklammern einklammert und dann noch Leerzeichen einfügt.

z. B. Der Text "Hello" soll in "< H E L L O >" umgewandelt werden.

Lösung: Die Lösung rechts als Extensionmethode ist im Main weitaus eleganter zu verwenden.

C# MakeTitle als statische Methode	C# MakeTitle als Extension Method
<pre>using System; using System.Text; namespace ExtensionMethod { class Program { // makes a title // e. g. s="Hello" returns "< H E L L O >" public static string MakeTitle(string s) { var builder = new StringBuilder("<"); foreach (char c in s) { builder.Append(" "+c.ToString().ToUpper()); } builder.Append(" >"); return builder.ToString(); } static void Main(string[] args) { Console.Write("Gib einen Namen ein: "); string s = Console.ReadLine(); string title = MakeTitle(s); Console.WriteLine(title); } } }</pre>	<pre>using System; using System.Text; namespace ExtensionMethod { static class ExtensionMethods { // makes a title // e. g. s="Hello" returns "< H E L L O >" public static string MakeTitle(this string s) { var builder = new StringBuilder("<"); foreach (char c in s) { builder.Append(" "+c.ToString().ToUpper()); } builder.Append(" >"); return builder.ToString(); } } class Program { static void Main(string[] args) { Console.Write("Gib einen Namen ein: "); string s = Console.ReadLine(); string title = s.MakeTitle(); Console.WriteLine(title); } } }</pre> <p>1. Erstelle eine statische Klasse</p> <p>2. Erstelle die Methode statisch</p> <p>3. Schreibe <code>this</code> vor den Typ</p> <p>Die Extensionmethode wird als Instanzmethode aufgerufen.</p>
Programmausgabe: Gib einen Namen ein: Hello < H E L L O >	Programmausgabe: Gib einen Namen ein: Hello < H E L L O >

Extensionmethoden werden im [Coding Kurzgeschichten](https://youtu.be/qFS_kDxybTE)-Video https://youtu.be/qFS_kDxybTE erklärt.



Video 7: Coding Kurzgeschichten-Video über Extension Methods

18 Indexer

Eckige Klammern [] sind bei Arrays üblich, um indexbasiert auf Elemente des Arrays zuzugreifen. Diese Zugriffsmöglichkeit steht in C# auch Klassen zur Verfügung.

Es ist keine Besonderheit, dass man bei Listen, z. B. `List<int>`, mit dem []-Operator auf ein bestimmtes Element der Liste zugreifen kann, z. B. mit `liste[0]` auf das erste Element. Listen sind aber Klassen und keine Arrays. Daher muss es offenbar ein Feature in C# geben, dass die Verwendung des []-Operator für Klassen gestattet. Dieses Feature heißt Indexer¹⁹.

- Jede Klasse kann eine beliebige Menge an Indexern implementieren.
- Ein Indexer dient dazu einer Klasse um einen indexbasierten Zugriff zu erweitern, so, dass mit dem []-Operator auf Elemente zugegriffen werden kann.
- Der Index kann ein beliebiger Datentyp sein und muss keine Ganzzahl sein. Bei der Collection-Klasse `Dictionary` ist es sehr üblich beliebige Datentypen zu verwenden.
- Ein Indexer hat die Syntax eines Properties mit einem Inputparameter.

C# Personenklasse mit einem Indexer für den indexbasierten Zugriff auf Vor- und Nachname

```
class Person
{
    public string First { get; private set; }
    public string Last { get; private set; }

    public Person(string first, string last)
    {
        First = first;
        Last = last;
    }

    // indexer
    // index is in fact some sort of "input parameter" which can be used in the get and set
    public string this[int index]
    {
        get
        {
            if (index == 0)
                return First;
            if (index == 1)
                return Last;

            return null;
        }

        set
        {
            if (index == 0)
                First = value;
            else if (index == 1)
                Last = value;
        }
    }
}
```

this[...] ist das Kennzeichen eines
Datentyp des Zugriffparameters (ist völlig beliebig)
Typ des Indexers
(Verwendung wie der Datentyp eines Properties)

¹⁹ <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/>

```
class Program
{
    static void Main(string[] args)
    {
        Person p = new Person("Donald", "Duck");

        Console.WriteLine(p[0] + " " + p[1]);

        p[0] = "Mickey";
        p[1] = "Mouse";

        Console.WriteLine(p[0] + " " + p[1]);
    }
}
```

Programmausgabe:

Donald Duck
Mickey Mouse

19 Expression-Bodied Members

Expression-Bodied Members²⁰²¹ sind eine etwas sperrige Bezeichnung für eine sehr effiziente Kurzschreibweise, die es in C# seit der Version 7.0 gibt.: get =>

Falls die Implementierung in einem der folgenden Codeblöcke

- `get`
- `set`
- Konstruktor
- Methode
- Indexer

nur aus einer einzigen Expression oder einem einzigen Statement besteht, kann man eine => Schreibweise verwenden.

Beispiele für die Verwendung der Expression-Bodied Members:

Herkömmliche Schreibweise	Kurzschreibweise mit =>
<pre>public string Full { get { return \$"{First} {Last}"; } }</pre>	<pre>public string Full => \$"{First} {Last}";</pre>
<pre>public void SetFirst(string first) { First = first; }</pre>	<pre>public void SetFirst(string first) => First = first;</pre>
<pre>public Person(string last) { Last = last; }</pre>	<pre>public Person(string last) => Last = last;</pre>
<pre>public void PrintFullName() { Console.WriteLine(ToString()); }</pre>	<pre>public void PrintFullName() => Console.WriteLine(ToString());</pre>
<pre>public override string ToString() { return \$"{First} {Last}".Trim(); }</pre>	<pre>public override string ToString() => \$"{First} {Last}".Trim();</pre>

Diese Kurzschreibweise kann nur dann verwendet werden, wenn eine einzige Anweisung oder eine einzige Expression vorliegt. Eine Methode mit zwei Zuweisungen, kann nicht abgekürzt werden.

Herkömmliche Schreibweise	Kurzschreibweise mit =>
<pre>public Person(string first, string last) { First = first; Last = last; }</pre>	Nicht möglich!

²⁰ "member" ist ein anderes Wort für "instance variable" oder "function member"

²¹ <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/expression-bodied-members>

Mehrere if-Anweisungen können durch den bedingten Operator (conditional operator ?:) manchmal auch in eine einzelne Expression umgeschrieben werden. Dann kann die Kurzschreibweise ebenfalls verwendet werden.

Programmcode ohne =>	Kurzschreibweise mit =>
<pre>public string this[int index] { get { if (index == 0) return First; if (index == 1) return Last; return null; } }</pre>	<pre>public string this[int index] { get => index == 0 ? First : index == 1 ? Last : null; }</pre> <p>Der conditional Operator wird hier zweimal hintereinander angewendet.</p>

Manchmal wird diese Schreibweise irrtümlich mit Lambdas verwechselt. Expression Bodied Members haben mit Lambdas rein gar nichts zu tun. Dass der Operator => auch bei Lambdas verwendet wird, ist eine andere Geschichte.

<https://davefancher.com/2014/08/25/c-6-0-expression-bodied-members/>

20 Enums

Enum ist die Abkürzung für Enumeration (=Aufzählung, engl. enumeration).

[https://msdn.microsoft.com/de-de/library/sbbt4032\(v=vs.80\).aspx](https://msdn.microsoft.com/de-de/library/sbbt4032(v=vs.80).aspx)

In C# - so wie in vielen anderen modernen Programmiersprachen – gibt es einen eigenen Datentyp für Aufzählungen, nämlich enum.

Enums sind benannte Konstanten.

Beispiel: Jahreszeiten als Enums in C#

```
enum Jahreszeit
{
    Frühling, ← Der erste Enumwert1, also Frühling, hat immer den Wert 0, außer
    Sommer, ← man weist einen anderen Wert zu, z. B. Frühling = 10.
    Herbst,
    Winter   ← Jeder weitere Enumwert hat einen um 1 erhöhten Wert, also
                Sommer = 1, Herbst = 2, Winter = 3.

}

static void Main(string[] args) ← Jahreszeit kann nur einen der vier Enumwerte speichern.
{
    Jahreszeit jahreszeit = Jahreszeit.Herbst; ← Wenn nötig, bekommt
    int wert = (int)jahreszeit; ← man den Zahlenwert
    Console.WriteLine("Der Wert von " + jahreszeit + " ist " + wert + "."); ← durch casten, also (int)

    Console.Write("Gib eine Jahreszeit ein: ");
    Jahreszeit jz = (Jahreszeit)Enum.Parse(typeof(Jahreszeit), Console.ReadLine());

    int pos = (int)jz + 1;
    Console.WriteLine(jz + " ist die " + pos + ". Jahreszeit des Jahres.");

    if (jz <= Jahreszeit.Sommer)
    {
        Console.WriteLine(jz + " ist in der ersten Jahreshälfte.");
    }
    else
    {
        Console.WriteLine(jz + " ist in der zweiten Jahreshälfte.");
    }

    Jahreszeit folgende = jz + 1;
    Console.WriteLine("Nach " + jz + " kommt " + folgende);
}
```

Es ist auch möglich, dass man für einen Zahlenwert, mehrere Namen vergibt, z. B.
Mehrsprachigkeit.

```
enum Jahreszeit
{
    Frühling, Sommer, Herbst, Winter,
    Spring = Frühling, Summer, Autumn
}
```

Frühling, Spring	0
Sommer, Summer	1
Herbst, Autumn	2
Winter	3

Unterschiedliche Groß- und Kleinschreibung bei Benutzereingaben kann so ignoriert werden:

```
Jahreszeit jz = (Jahreszeit)Enum.Parse(typeof(Jahreszeit), Console.ReadLine(), true);
```

²² In der MSDN wird statt "Enumwert" der Ausdruck Enumerator verwendet.

Enumwerte in C# sind immer ein Ganzahltyp (integraler Type), also byte, short, int, long, ...
enum Priority : long ändert den Wertebereich auf long.

Das Coding Kurzgeschichten-Video "[Enums in C# - Die vier Jahreszeiten](https://youtu.be/hvzivzUJjc)" <https://youtu.be/hvzivzUJjc> erklärt Enums anhand der vier Jahreszeiten.



Enums in C# - Die vier Jahreszeiten

Coding Kurzgeschichten

Video 8: Coding Kurzgeschichten-Video über Enums in C#: Die vier Jahreszeiten

20.1 Codebeispiel für Prioritäten

Das folgende Beispiel zeigt ein Enum, das die Priorität repräsentiert.

C# Enum für Prioritäten

```
enum Priority
{
    Low,      // 0
    Medium,   // 1
    High.     // 2
}

class Program
{
    static void Main(string[] args)
    {
        // enum -> int
        int p = (int)Priority.High;
        Console.WriteLine($"{Priority.High} = {p}");
        p--;

        // int -> enum
        Priority prio = (Priority)p;
        Console.WriteLine(prio);
        Console.WriteLine();

        // loop through enum values
        foreach (var pr in Enum.GetValues(typeof(Priority)))
            Console.WriteLine(pr);

        prio = (Priority)Enum.Parse(typeof(Priority), "Low");
        Console.WriteLine("\n" + prio);
    }
}
```

Programmausgabe:

High = 2

Medium

Low

Medium

High

Low

21 Datums- und Zeitberechnungen

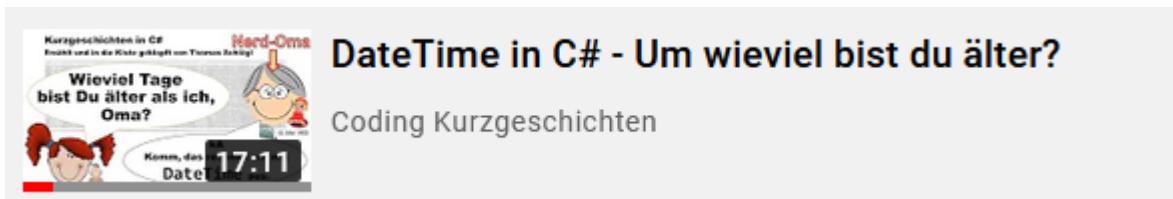
Alle Zeit- und Datumsangaben werden in C# in der Struct `DateTime` gespeichert. Gemeinsam mit der Struct `TimeSpan` für eine Zeitspanne ermöglicht C# einfache Berechnungen mit Datumsangaben und Zeitdauern.

Struct	Beschreibung	Beispiele
DateTime	Zeitpunkt mit Datum und Zeit	Geburtstag, Öffnungszeitpunkt eines Geschäfts, Erstzulassung eines Fahrzeugs, ...
TimeSpan	Zeitdauer in Stunden, Minuten und Sekunden	Öffnungszeitdauer eines Geschäfts, Lebensalter einer Person, ...

Ein [Coding Kurzgeschichten](#)-Video, dass die Verwendung erklärt ist <https://youtu.be/JlbPucuCbGg>.

Der Programmcode ist auf GitHub verfügbar.

<https://github.com/slogslog/Coding-Kurzgeschichten/tree/master/010%20DateTime>



Video 9: Coding Kurzgeschichten-Video über DateTime und TimeSpan in C#

21.1 Datumsangaben und Zeitpunkte: `DateTime`

Diese kann einfach als Datentyp verwendet werden. Z. B. in Klassen oder in Variablen, wie unten im Main.

`DateTime` repräsentiert immer einen Zeitpunkt, also ein Datum mit einer Uhrzeit. Falls die Uhrzeit nicht benötigt wird, kann man diese mit diversen Properties und Methoden ignorieren.

Wenn ein `DateTime` ohne Zeit angelegt wird, z. B. durch

```
DateTime dt1 = new DateTime(2021, 12, 24);
```

ist diese dann immer 0:00 Uhr.

C# Demo für <code>DateTime</code>
<pre>using System; namespace DateTime_Script { class Program { static void Main(string[] args) { // Erstellen eines DateTime mit einem Konstruktor, 24. 12. 2021 0:00 Uhr DateTime dt1 = new DateTime(2021, 12, 24); // 24. 12. 2021 13:37 Uhr und 42 Sekunden } } }</pre>

```

DateTime dt2 = new DateTime(2021, 12, 24, 13, 37, 42);

// Erzeugung eines DateTimes aus einem String.
Console.WriteLine("Gib ein Datum ein: ");
string s = Console.ReadLine();
DateTime dtEingabe = DateTime.Parse(s);

// DateTime hat sehr viele praktische Properties ...
Console.WriteLine(dtEingabe.Day); // Tag ausgeben
Console.WriteLine(dtEingabe.Month); // Monat ausgeben

// ... und sehr viele praktische Methoden.
DateTime dtEineWocheSpäter = dtEingabe.AddDays(7);
Console.WriteLine(dtEineWocheSpäter.ToString("dd-MM-yyyy"));

DateTime dt = dtEingabe.AddMonths(1);
DateTime dtErsterTagdesNächstenMonats = new DateTime(dt.Year, dt.Month, 1);
Console.WriteLine(dtErsterTagdesNächstenMonats.ToString("dd-MM-yyyy"));

DateTime dtLetzterTagdesJetzigenMonats = dtErsterTagdesNächstenMonats.AddDays(-1);
Console.WriteLine(dtLetzterTagdesJetzigenMonats.ToString("dd-MM-yyyy"));

// Der jetzige Zeitpunkt
DateTime jetzt = DateTime.Now;

// Formatierung der Ausgabe
Console.WriteLine(jetzt.ToString("d-M-yyyy hh:mm:ss"));
Console.WriteLine(jetzt.ToString("dd-MM-yyyy"));
Console.WriteLine(jetzt.ToString("d-MMM-yyyy"));
Console.WriteLine(jetzt.ToString("d-MMMM-yyyy"));

// Zeitschleife von jetzt bis in 10 Minuten.
for (DateTime datum = jetzt;
    datum < jetzt.AddMinutes(10);
    datum = datum.AddMinutes(1))
{
    Console.WriteLine($"{datum.ToString("HH:mm")}");
}
}
}

```

Programmabgabe:

Gib ein Datum ein: 2021-05-15

15

5

22.05.2021

01.06.2021

31.05.2021

2021-9-4 09:58:34

2021-09-04

2021-Sep.-4

2021-September-4

09:58 Uhr

09:59 Uhr

10:00 Uhr

10:01 Uhr

10:02 Uhr

10:03 Uhr

10:04 Uhr

10:05 Uhr

10:06 Uhr

10:07 Uhr

21.2 Zeitspanne TimeSpan

Eine Zeitspanne ist der Abstand zweier diskreter Zeitpunkte. In C# erzeugt man eine Instanz oft durch das Subtrahieren von zwei DateTimes. Der Operator - ist für DateTime überladen.

C# Demo für DateTime

```
using System;

namespace DateTime_Script
{
    class Program
    {
        static void Main(string[] args)
        {
            // 24. 12. 2021 18:00 Uhr
            DateTime dt1 = new DateTime(2021, 12, 24, 18, 0, 0);

            // 1. Jänner 2022
            DateTime dt2 = new DateTime(2022, 1, 1);

            TimeSpan duration = dt2 - dt1;
            Console.WriteLine($"Zeitdauer: {Math.Round(duration.TotalHours, 2)} Stunden");

            int tage = (dt1.Date - new DateTime(2021, 1, 1)).Days + 1;
            Console.WriteLine($"{tage} Tage von Neujahr bis Weihnachten");
        }
    }
}
```

Programmabgabe:

Zeitdauer: 174 Stunden
358 Tage von Neujahr bis Weihnachten

Auch das Berechnen des Altersunterschieds in Tagen ist eine nette Anwendung.

Das ist ähnlich zu dem oberen Programm und [hier](#) im [Coding Kurzgeschichten](#)-Video demonstriert <https://youtu.be/JlbPucuCbGg?t=66>.

22 Implicit Operators – Cast Operatoren

Es ist auch möglich, Casten in Klassen einzubauen. Das entspricht dem Überladen des Cast-Operators in C++. Wie Operator Overloading kann das den Code lesbarer machen.

Gegeben ist eine Struct die eine Binärzahl repräsentiert.²³

C# Struct für eine Binärzahl

```
struct Binary
{
    public string binary;

    public int GetDecimal()
    {
        return BinaryToDecimal(binary);
    }

    public Binary(string binary) => this.binary = binary;

    private static int BinaryToDecimal(string binary)
    {
        int summe = 0;

        // höchste 2er-Potenz
        int potenz = (int)Math.Pow(2, binary.Length - 1);

        for (int i = 0; i < binary.Length; i++)
        {
            string zeichen = binary.Substring(i, 1);

            if (zeichen == "1")
                summe += potenz;

            potenz >>= 1; // entspricht potenz = potenz/2;
        }

        return summe;
    }
}
```

Im Main wird die Binärzahl im Konstruktor als String übergeben und der Dezimalwert mit GetDecimal ausgelesen.

C# Anlegen einer Binärzahl und Auslesen des Wertes über einen Getter

```
class Program
{
    static void Main(string[] args)
    {
        Binary binary = new Binary("11101");

        int number = binary.GetDecimal();
        Console.WriteLine(number);
    }
}
```

Programmausgabe:

29

²³ Die Implementierung ist absichtlich sehr einfach gehalten. Eine Struct wurde gewählt, da jede Instanz davon, dann ein Value Type ist und beim Zuweisen und bei Methodenübergaben wirklich als Zahl funktioniert.

Im Main wäre es nun praktisch - da es sich bei Binärzahl um eine Zahl handelt – dass man den Wert ohne Getter (GetDecimal) verwenden könnte, nämlich einfach so:

```
int number = binary;
```

Anders formuliert, es wäre angenehm, jede Instanz in einen anderen Typ casten zu können:

```
int number = (int) binary;
```

Dafür muss für die Instanz (funktioniert auch für Klassen) ein int berechnet werden. Das ist durch das Schlüsselwort `implicit` möglich. Wenn man die Struct um diesen Operator erweitert, ist das oben besprochene Statement im Main möglich.

C# Struct für eine Binärzahl mit einem implicit operator

```
struct Binary
{
    public string binary;

    public static implicit operator int(Binary b)
    {
        return BinaryToDecimal(b.binary);
    }

    public Binary(string binary) => this.binary = binary;

    private static int BinaryToDecimal(string binary)
    {
        int summe = 0;

        // höchste 2er-Potenz
        int potenz = (int)Math.Pow(2, binary.Length - 1);

        for (int i = 0; i < binary.Length; i++)
        {
            string zeichen = binary.Substring(i, 1);

            if (zeichen == "1")
                summe += potenz;

            potenz >>= 1; // entspricht potenz = potenz/2;
        }

        return summe;
    }
}
```

C# Anlegen einer Binärzahl und Auslesen des Wertes über einen Getter

```
class Program
{
    static void Main(string[] args)
    {
        Binary binary = new Binary("11101");

        int number = binary;
        Console.WriteLine(number);
    }
}
```

Programmausgabe:

29

23 Variablen

In C# gibt es dieselben drei Arten von Variablen.

23.1 Lokale Variablen

Existieren innerhalb einer Methode oder eines Blocks. Sie werden immer am Stack (und nicht am Heap) gespeichert. Sie bleiben bis zum Ende der Methode bzw. des Blocks sichtbar und werden, wenn sie out-of-scope gehen, vom Stack gelöscht. Lokale Variablen werden nicht automatisch initialisiert, sondern müssen explizit initialisiert werden.

23.2 Instanzvariablen

Werden in Klassen ohne dem Schlüsselwort `static` definiert. Sie existieren solange es die Instanz (Objekt) gibt, in der sie gespeichert sind. Sie werden mit der Instanz auf dem Heap gespeichert. Instanzvariablen werden mit Defaultwerten initialisiert.

Beachte, dass `public` Instanzvariablen ganz schlechter Programmierstil sind. Sie müssen `private` oder allerhöchstens `protected` sein.

23.3 Klassenvariablen

Ähneln zunächst Instanzvariablen, werden allerdings mit dem Schlüsselwort `static` definiert und sind daher Variablen, die für alle Instanzen (Objekte) einer Klasse gelten. Sie werden mit dem Laden der Klasse angelegt. Klassenvariablen werden mit Defaultwerten initialisiert.

C# Eine Klasse mit allen drei Variabtentypen

```
class Variables {  
  
    private int value;           // instance variable: initialized with 0  
  
    public static bool state;    // class variable: initialized with false  
  
    public static double init() {  
        double local;            // Local variable: uninitialized  
        return local + 0.5;       // Compile error!!!  
    }  
    ...  
}
```

23.4 Instanzierung von Objekten und Garbage Collection

Alle Instanzen (Objekte) werden mit dem Schlüsselwort `new` angelegt und am Heap gespeichert. Wenn es während dem Programmablauf dazu kommt, dass es keine Referenz mehr auf eine Instanz gibt – d. h. sie wird nicht mehr verwendet – kann diese jederzeit durch die Garbage Collection gelöscht werden.

Eine Garbage Collection kann in C# nicht explizit gestartet werden. Es ist nur möglich, sie durch einen Aufruf der Laufzeitumgebung "vorzuschlagen".

23.5 Value Types und Reference Types

C# - und auch Java - unterteilt alle Datentypen in Value Types (Werttypen) oder Reference Types (Verweistypen). Abbildung 1: Übersicht über die Datentypen in C# in Kapitel 2.3 zeigt alle Datentypen.

Zur Laufzeit werden Variablenwerte in zwei verschiedenen Speicherbereichen abgelegt, am **Stack** und am **Heap**, abhängig von dem Datentyp der Variable. Beides sind Speicherbereiche, die von der Laufzeitumgebung von C# bei der Ausführung verwendet wird.



Abbildung 24: Essen auf einem Tisch und Kühlschrank als Analogie zu einem Stack und Heap

Der **Stack** ist ein schneller Speicher, der bei C# defaultmäßig 1 MByte groß ist. Neben lokalen Werten werden auch Rücksprungadressen bei Methodenaufrufen am Stack abgespeichert.

Der **Heap** ist ein Speicher der als Arbeitsspeicher eines Prozesses dient. Seine Größe hängt vom RAM des Computers bzw. dem davon der Laufzeitumgebung von C# zur Verfügung gestellten Größe abhängig. Die Größe ist im Gegensatz zum Stack aber leicht mehrere GByte groß.

Dieser Abschnitt verlinkt an verschiedenen Stellen auf die folgenden zwei [Coding Kurzgeschichten](#)-Videos.

Auch die animierten Powerpoint-Präsentationen können von [GitHub](#) heruntergeladen werden.

 Coding Kurzgeschichten-Video: https://youtu.be/FN5EWXP4QMU Animierte Powerpoint Präsentation auf GitHub: hier	1 Wert- und Verweistypen in C# - Teil 1 - Hinweise auf Essbares Coding Kurzgeschichten Coding Kurzgeschichten-Video: https://youtu.be/FcNvD2-jyCg Animierte Powerpoint Präsentation auf GitHub: hier
---	---

Video 10: Coding Kurzgeschichten-Videos über Value und Reference Types

23.5.1 Merkregel

Variablen die einen **Value Type** als Datentyp haben, speichern ihre Werte **immer am Stack**.

Variablen die einen **Reference Type** als Datentyp haben, also z. B. Array, string oder Instanzen von Klassen, speichern ihre Werte **immer** an einer bestimmten Stelle **am Heap**. Am Stack wird aber zusätzlich für die Variable eine Reference (ein Verweis) gespeichert, wo am Heap der Wert zu finden ist.

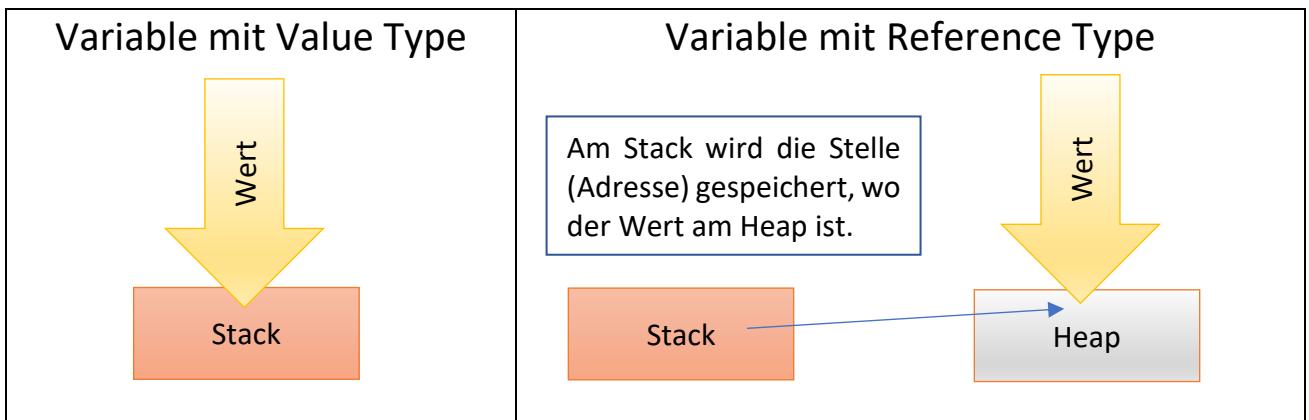


Abbildung 25: Merkregel für Value Types und Reference Types

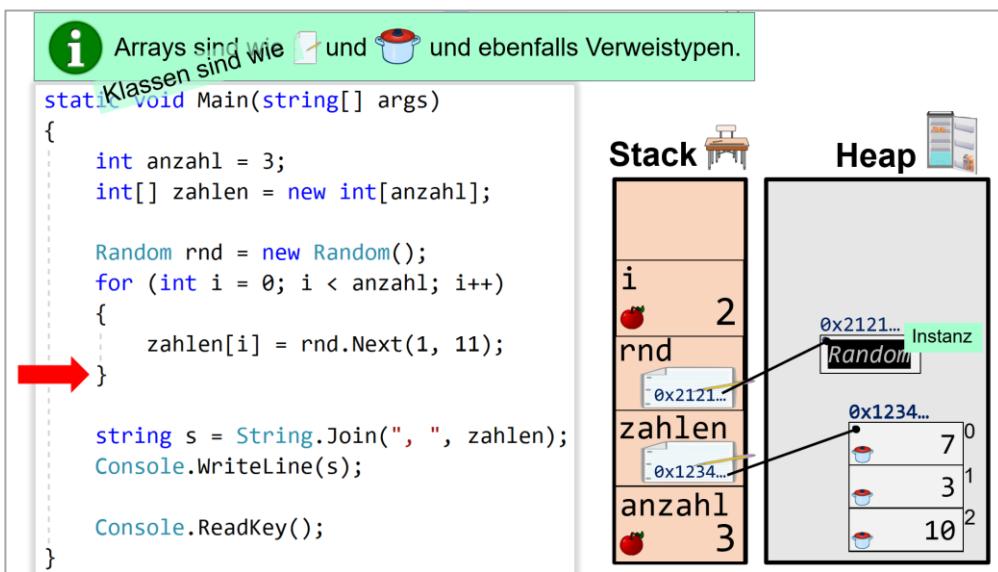


Abbildung 26: Snapshot des Speichers während einer Programmausführung

i und *anzahl* sind **int**-Variablen. **int** ist ein **Value Type**, daher wird der Wert der Variablen am Stack gespeichert.

rnd ist die Instanz einer Klasse, *zahlen* ist ein Array. Klasse und Array sind **Reference Types**, daher werden deren Werte am **Heap** gespeichert. Am **Stack** ist eine **Reference (Verweis)** auf die richtige Stelle im **Heap** gespeichert.

23.5.2 Value Types am Stack

Lokale Variablen von Value Types, also z. B. `int`, `double` oder `char` werden am Stack abgelegt. Dadurch kann auf deren Wert sehr schnell zugegriffen werden.

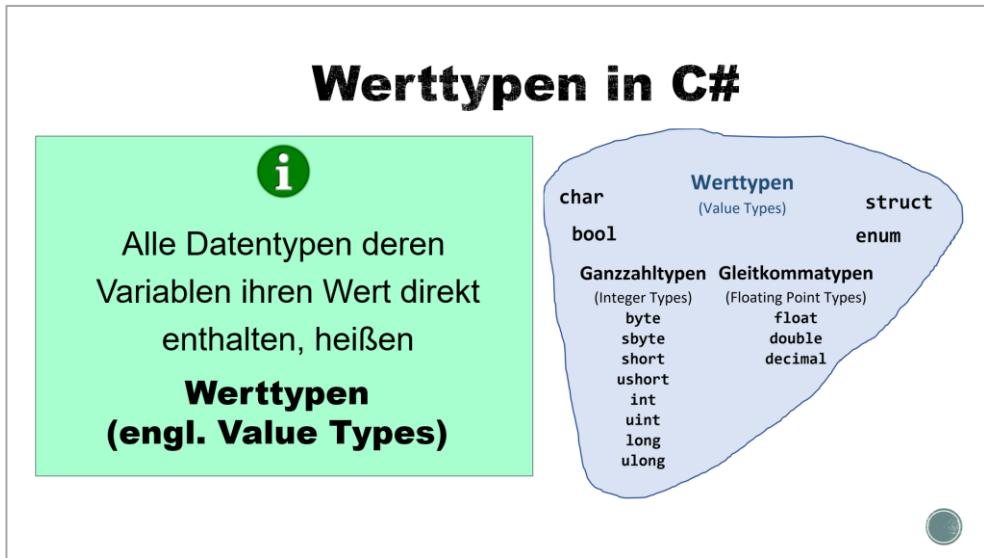


Abbildung 27: Werttypen (Value Types) in C#

Die genauen Abläufe bei den drei unteren Programmen können über diese [Powerpoint-Präsentation](#) nachvollzogen werden.

Im Video werden die Abläufe ab [15 Minuten](#) erklärt.

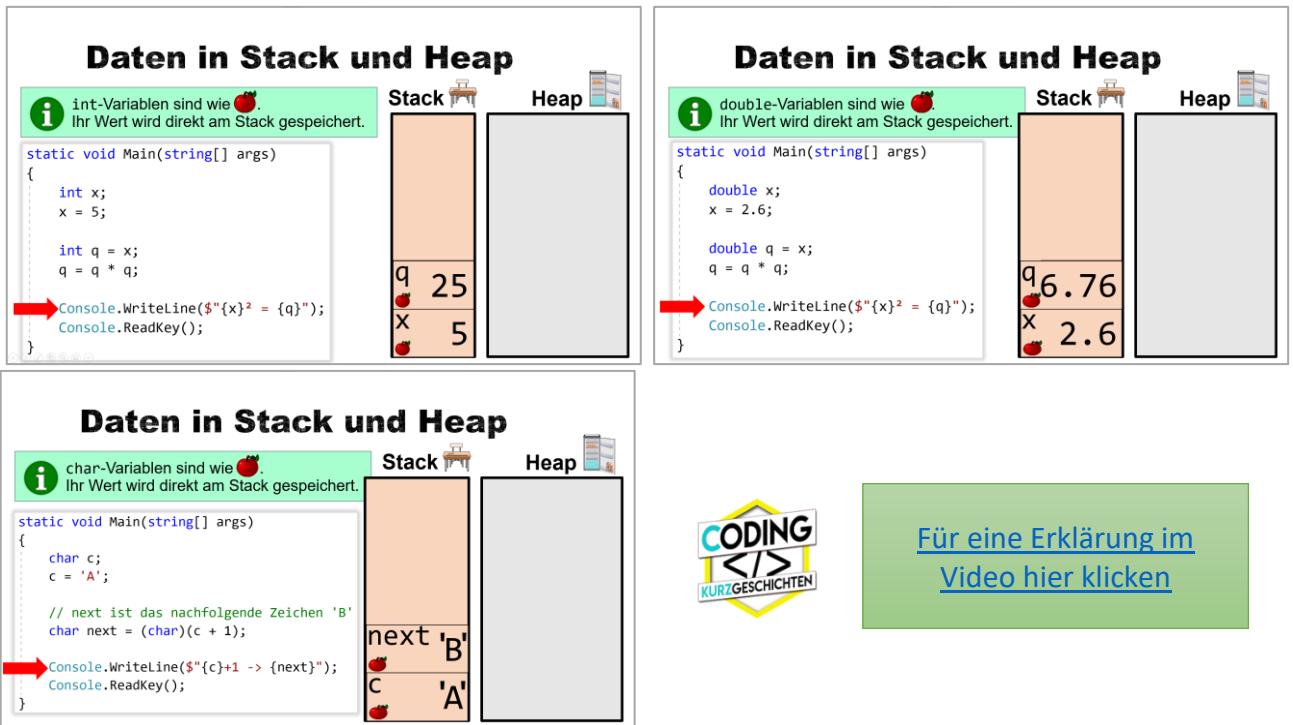


Abbildung 28: Lokale int-, double- oder char-Variablen werden am Stack abgespeichert.

23.5.2.1 Value Types bei Methodenaufrufen

Wenn eine Methode im Main aufgerufen werden, sind die im Main angelegten Variablen in der Methode nicht verfügbar. Dadurch kommt die Stapeleigenschaft des Stacks zu tragen. Am Stack werden – neben der Rücksprungadresse – die Werte der lokalen Variablen der aufgerufenen Methode abgelegt.

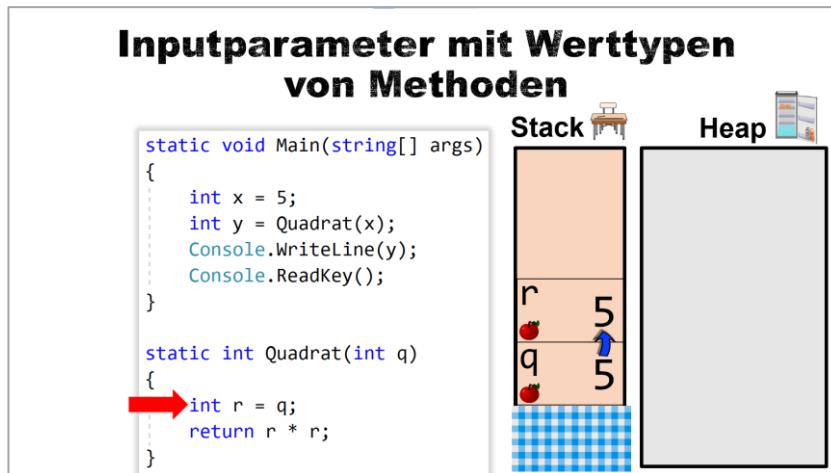


Abbildung 29: int-Variablen am Stack beim Aufruf einer Methode

23.5.3 Reference Types am Heap

Lokale Variablen von Reference Types, also z. B. `string` oder Instanzen von Klassen legen ihren Wert auf den Heap und hinterlassen am Stack einen Verweis (Reference) wo am Heap der Wert zu finden ist.

Der folgende Screenshot aus der Powerpoint Präsentation ([hier klicken](#)) vom [Coding Kurzgeschichten](#)-Video <https://youtu.be/FcNvD2-jyCg> zeigt eine Situation bei der **zwei String-Variablen t und s beide auf denselben String im Heap verweisen**.

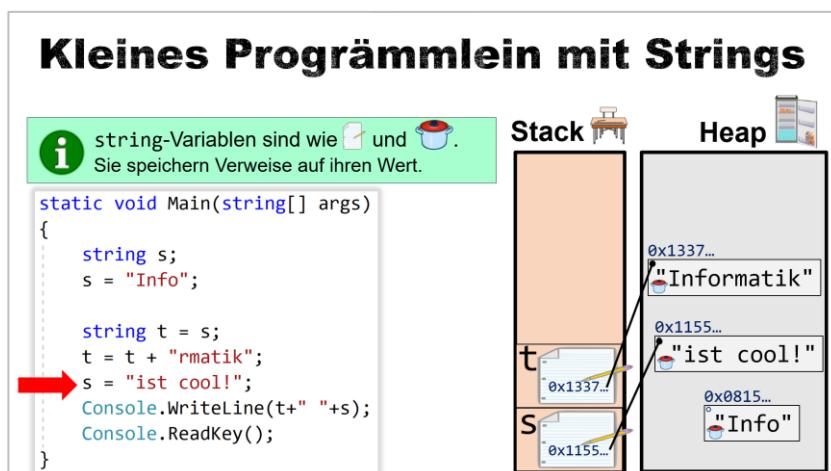


Abbildung 30: Werttypen (Value Types) in C#

Besonders wichtig ist die Kenntnis der Vorgänge bei Reference Types bei der Übergabe von Variablen eines Reference Types an eine Methode.

23.5.3.1 Reference Types bei Methodenaufrufen

Variablen im Main können nämlich in Methoden so verändert werden, dass diese Änderungen im Main bestehen bleiben. Das ist ein wesentlicher Unterschied gegenüber Value Types.



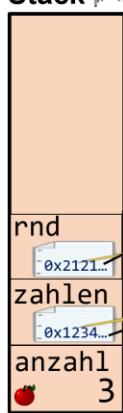
Arrays als Inputparameter

```
static void Main(string[] args)
{
    int anzahl = 3;
    int[] zahlen = new int[anzahl];

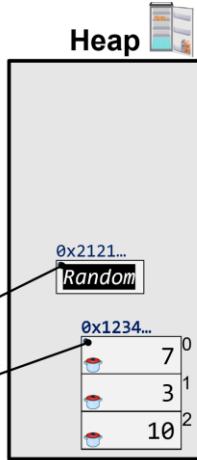
    Random rnd = new Random();
    for (int i = 0; i < anzahl; i++)
    {
        zahlen[i] = rnd.Next(1, 11);
    }

    GeradeMachen(zahlen);
    string s = String.Join(", ", zahlen);
    Console.WriteLine(s);
    Console.ReadKey();
}
```

Stack



Heap



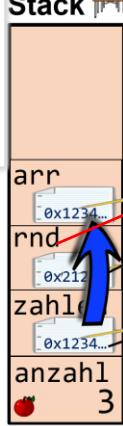
Für eine
Erklärung im
Video hier
klicken

```
AT
static void GeradeMachen(int[] arr)
{
    for (int i = 0; i < arr.Length; i++)
    {
        if (arr[i] % 2 == 1)
            arr[i]++;
    }

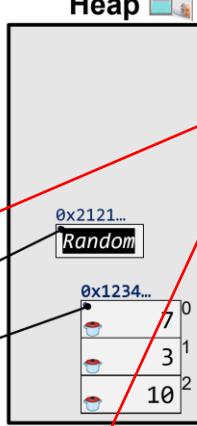
    for (int i = 0; i < anzahl; i++)
    {
        zahlen[i] = rnd.Next(1, 11);
    }

    GeradeMachen(zahlen);
    string s = String.Join(", ", zahlen);
    Console.WriteLine(s);
    Console.ReadKey();
}
```

Stack



Heap



Die Referenz auf
das Array wird
beim Aufruf von
zahlen in arr
kopiert.
Daher verändert
die Methode den
Arrayinhalt!

```
AT
static void GeradeMachen(int[] arr)
{
    for (int i = 0; i < arr.Length; i++)
    {
        if (arr[i] % 2 == 1)
            arr[i]++;
    }

    for (int i = 0; i < anzahl; i++)
    {
        zahlen[i] = rnd.Next(1, 11);
    }

    GeradeMachen(zahlen);
    string s = String.Join(", ", zahlen);
    Console.WriteLine(s);
    Console.ReadKey();
}
```

Stack



Heap



Abbildung 31: int[]-Variable am Heap beim Aufruf einer Methode

Die Screenshots aus der Powerpoint Präsentation ([hier klicken](#)) vom [Coding Kurzgeschichten](#)-Video <https://youtu.be/FcNvD2-jyCg> zeigt wie ein Array an eine Methode übergeben wird.

Da der Array-Inhalt als Reference Type den Wert am Heap speichert und die Methode die Referenz auf diesen Bereich erhält, verändert die Methode den Speicherbereich des Arrays. Nach dem Verlassen der Methode bleibt diese Änderung im Arrayinhalt im Main natürlich bestehen.

Eine sehr ausführliche Erklärung der beiden Speicher und das Ablegen von Variablenwerten während des Programmablaufs werden in den folgenden drei [Coding Kurzgeschichten](#)-Videos erklärt.

23.5.3.2 Schlüsselwort `ref`

Um bei Variablen die einen Value Type haben ebenfalls eine Referenz zu übergeben und so eine Veränderung des Werts in einer Methode zu ermöglichen, ist ein eigenes Schlüsselwort verfügbar.

Das [Coding Kurzgeschichten](#)-Video <https://youtu.be/2ud9My5RNGk> und die Powerpoint Präsentation ([hier klicken](#)) erklären die Zusammenhänge.



Video 11: Coding Kurzgeschichten-Videos über Stack und Heap

Das Schlüsselwort `ref` sorgt dafür, dass der Inputparameter am Stack nicht den Wert der übergebenen Variable speichert, sondern die Referenz auf sie, also die Speicheradresse am Stack. In der Methode wird dann bei jedem Zugriff auf den Inputparameter einfach der Speicherbereich am Stack verändert. Diese Veränderung bewirkt natürlich auch, dass im Main die dort gespeicherte Variable den geänderten Wert hat.

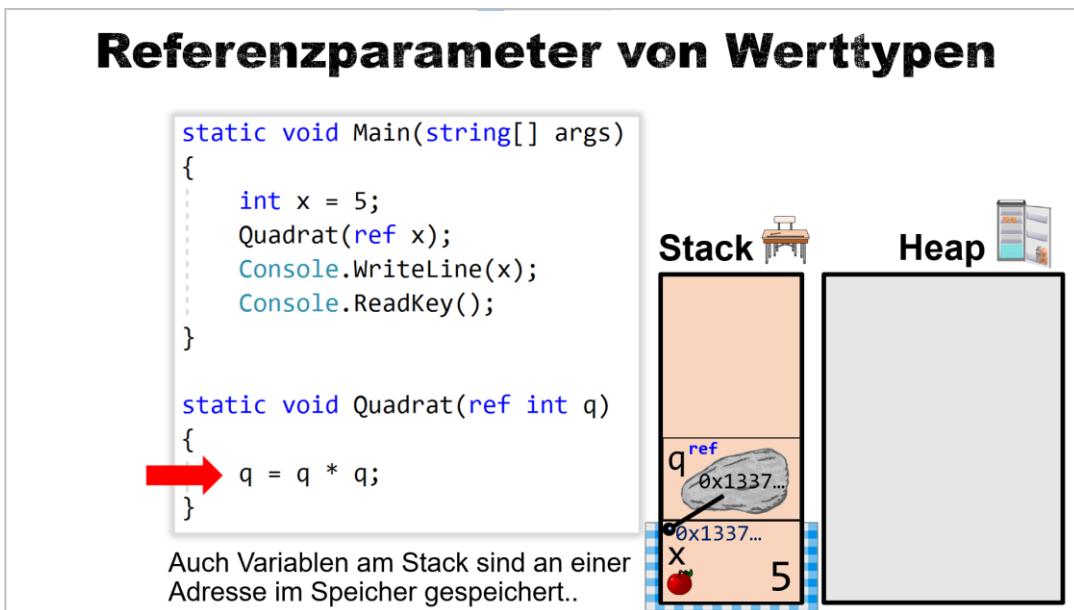


Abbildung 32: `ref int` Parameter bei Methodenaufrufen

Zur genauen Erklärung im Video gelangt man hier <https://youtu.be/2ud9My5RNGk?t=343>.

Natürlich kann ref auch bei Reference Types verwendet werden. Das wäre z. B. beim Austauschen zweier Arrays erforderlich.

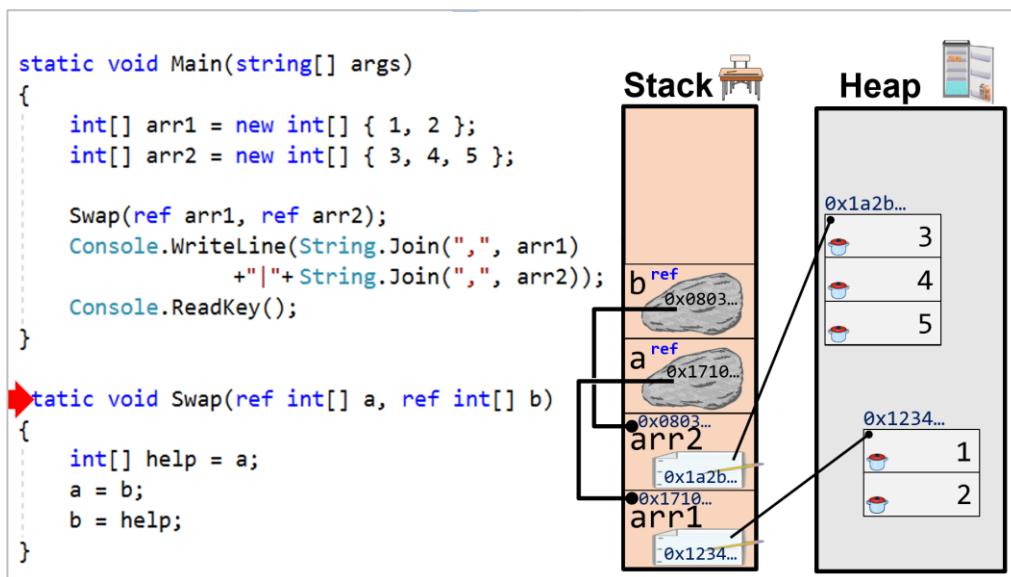


Abbildung 33: ref int[]-Variable bei Methodenaufrufen

Zur genauen Erklärung im Video gelangt man hier <https://youtu.be/2ud9My5RNGk?t=1016>.

24 Naming Conventions

*There are only two hard things in Computer Science:
cache invalidation and naming things. [Phil Karlton]*

Naming Conventions werden in Entwicklungsprojekten meist individuell definiert, es gibt allerdings in C# einige generelle Regeln die allgemein gültig sind und auch im .NET-Framework umgesetzt sind.

In C# beginnen alle Methodenamen ausnahmslos mit Großbuchstaben.

Allgemeine Regeln sind, dass Namen grundsätzlich keine Whitespaces enthalten dürfen und nicht mit speziellen Zeichen, wie \$ oder % beginnen sollen. Weiters ist es angeraten, keine speziellen Zeichen, wie die deutschen Sonderzeichen ä, ö, ü, ß, Ä, Ö, Ü und ß in Namen zu verwenden.

Alle Namen sollen der [Binnenmajuskelschreibweise](#) folgen 😊, besser bekannt unter der englischen Bezeichnung *CamelCase* (siehe auch https://en.wikipedia.org/wiki/Camel_case).

D. h. bei der Kombination von mehreren Wörtern sollten diese aneinandergereiht werden mit dem ersten Buchstaben großgeschrieben, z. B. ActionListener, DatabaseGuard, ComboBox, ListView.

Der Name einer/s ...	C#
Klasse und Struct	<ul style="list-style-type: none">... soll mit einem Großbuchstaben beginnen.... soll ein Hauptwort sein.
Interfaces	<ul style="list-style-type: none">... soll mit einem Großbuchstaben beginnen.... soll ein Eigenschaftwort sein, z. B. Runnable, Remote.
Methode	<ul style="list-style-type: none">... muss mit einem Großbuchstaben beginnen.... soll ein Verb (Zeitwort) sein, z. B. Print, Add.Wenn der Name aus mehreren Wörtern besteht, wird mit einem Kleinbuchstaben
Variablen	<ul style="list-style-type: none">... soll mit einem Kleinbuchstaben beginnen.Wenn der Name aus mehreren Wörtern besteht, wird mit einem Kleinbuchstaben begonnen und dann Camel Case verwendet, z. B. firstName, countOfNumbers, ageOfPerson.
Properties	<ul style="list-style-type: none">... muss mit einem Großbuchstaben beginnen... soll ein Hauptwort sein.
Packages/Namespace	siehe Abschnitt Fehler! Verweisquelle konnte nicht gefunden werden. Fehler! Verweisquelle konnte nicht gefunden werden.
Konstanten	<ul style="list-style-type: none">... soll in Großbuchstaben geschrieben sein, z. B. RED, HIGH.... bei mehreren Wörtern sind Underscores (_) zur Trennung gut, z. B. MAX_VALUE, MIN_RATIO.
Type Parameter	Siehe Abschnitt 28 Generics.

25 Einige weitere Schlüsselwörter in C#

Dieser Abschnitt enthält die Beschreibung einiger nützlicher Schlüsselwörter die zu den anderen Themen nicht zuordenbar sind.

25.1 var

In C# ist die Verwendung des var-Schlüsselworts sehr geläufig. Es ist ein "Faulheitskeyword", das es ermöglicht bei Variablen Deklarationen, den tatsächlichen Datentyp einer Variablen vom Compiler bestimmen zu lassen. Zur Laufzeit macht es überhaupt keinen Unterschied, ob eine Variable mit var oder ihrem tatsächlichen Datentypen deklariert wurde.

C#

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

static void Main(string[] args)
{
    var num = new List<int>() { 8, 12, 4 };

    var primTo10 = new List<int>()
        { 1, 2, 3, 5, 7 };

    var p1 = new Person("Max", "Hammer");

    // 10th March 2002
    var p2 = new DateTime(2002, 3, 10);

    var dic = new Dictionary<Person, int>();

    var counter = 3;
    var g = 9.81;

    ...
}
```

25.2 const

Das Schlüsselwort const kann nur vor einer Variablen Deklaration stehen und zeigt an, dass die so definierte Variable ihren Wert während der Laufzeit nie verändern kann.

C# Instanzvariable mit const

Math.cs (Teil des .NET Frameworks)

```
public static class Math
{
    public const double PI = 3.1415926535897931;
    ...
}
```

In C# ist PI nur formell eine konstante Instanzvariable in einer statischen Klasse. Zu beachten ist, dass const immer bedeutet, dass etwas static ist. Die Kombination const static wäre redundant und wird daher vom Compiler nicht akzeptiert.

Das Schlüsselwort `const` wird in C# verwendet um lokale Variablen und Instanzvariablen unveränderbar statisch anzulegen.

C# Lokale Variable mit `const`

```
static void Main(string[] args)
{
    const int COUNT = 30;
    int[] numbers = new int[COUNT];
    ...
}
```

25.3 using-Statement

C# unterstützt, dass bei resourceintensiven Klassen, wie z. B. dem `StreamReader` abschließende Methodenaufrufe wie `Close()` automatisch aufgerufen werden können, damit man das nicht vergisst. Das erledigt das Statement `using(...){...}`.

Siehe auch <https://www.csharp-examples.net/using/> für ein sehr gute Beschreibung.

C#

```
using System;
using System.IO;

namespace UsingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                using (var reader = new StreamReader("Values.txt"))
                {
                    string line = "";
                    double sum = 0;

                    while ((line = reader.ReadLine()) != null)
                    {
                        sum += double.Parse(line);
                    }

                    Console.WriteLine($"The overall sum is {Math.Round(sum, 2)}");
                }
            catch (IOException e)
            {
                Console.WriteLine(e.Message);
                Console.Error.WriteLine(e.StackTrace);
            }
        }
    }
}
```

Die Klasse muss das Interface `System.IDisposable` implementieren, damit am Ende des try-Blocks die `Dispose()`-Methode automatisch aufgerufen wird.

Schliesst automatisch das File.

Ausgabe:

The overall sum is 35.24.

26 Exceptions

Es ist wichtig in Programmen auf möglicherweise auftretende Fehler des Benutzers zu reagieren. Es gibt zwei wesentliche Arten wie in Programmen auf unvorhergesehene Situationen reagiert wird:

1. **Rückgabewert:** Die Methode in der ein Fehler auftritt, gibt einen Rückgabewert zurück, der anzeigt, dass ein Fehler aufgetreten ist.
2. **Exceptions (Ausnahmen):** Die Programmstelle an der ein Fehler bzw. ein unvorhergesehene Situation auftritt, wirft eine Ausnahme (engl. throws an exception).

26.1 Rückgabewerte

Ein Beispiel für Fehlerbehandlung durch einen Rückgabewert ist die Methode `int.TryParse` (wie auch alle anderen TryParse Methoden der Wertetypen).

```
static void Main(string[] args)
{
    if (int.TryParse(Console.ReadLine(), out int wert))
    {
        Console.WriteLine("{0}*{0}={1}", wert, wert * wert);
    }
    else
    {
        Console.WriteLine("Ungültige Eingabe");
    }
}
```

`TryParse` überprüft, ob der Inputstring eine gültige Zahl enthält. Wenn das der Fall ist, dann gibt `TryParse` true als Return Value zurück und liefert die Zahl als Output Parameter an die Variable `wert` zurück. Wenn der Inputstring keine gültige Zahl enthält, z. B. "34u", dann gibt `TryParse` den Wert false zurück. Dadurch erkennt man beim Aufruf, dass die Variable `wert` keinen gültigen Wert hat.

Im Falle einer unvorhergesehenen Situation kann das Programm auch eine Exception werfen.

26.2 Codebeispiel 1: Index Out Of Range

Wenn ein Codeteil eine Exception auslöst und kein Codeteil fängt diese, so kommt es zum Anhalten des Programms aufgrund einer unhandled Exception.

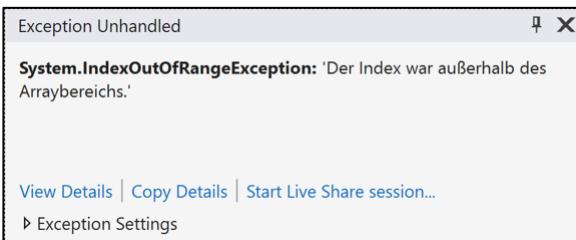
C#

```
class Program
{
    static void Main(string[] args)
    {
        int[] numbers = { 42, 1337, 0xff, 10 };

        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine(numbers[i]);
        }
    }
}
```

Ausgabe:

```
42  
1337  
255  
10
```



Der betroffene Code wird in einen try-Block gepackt. Wenn innerhalb des try-Blocks eine Exception geworfen wird, wird in der Reihenfolge der Implementierung der catch-Blöcke während der Laufzeit nach einem passenden catch-Block gesucht (entlang des Callstacks). Hierbei werden auch Klassenhierarchien der Exceptionklassen berücksichtigt. Abgeleitete Klassen einer Exception im catch-Block werden dort ebenfalls gefangen.

C#

```
class Program  
{  
    static void Main(string[] args)  
    {  
        int[] numbers = { 42, 1337, 0xff, 10 };  
  
        try  
        {  
            for (int i = 0; i < 5; i++)  
            {  
                Console.WriteLine(numbers[i]);  
            }  
        }  
        catch (IndexOutOfRangeException)  
        {  
            Console.WriteLine("Invalid index!");  
        }  
        finally  
        {  
            Console.WriteLine("Final greetings!");  
        }  
    }  
}
```

Ausgabe:

```
42  
1337  
255  
10  
Invalid Index!  
Final greetings!
```

26.3 Codebeispiel 2: Ganzzahldivision durch 0

Das wird beispielsweise dann gemacht, wenn es nicht möglich ist einen Rückgabewert zurückzuliefern, wie bei einer Ganzzahldivision durch 0.

```

try
{
    // Hier ist der Code der einen Fehler wirft, wenn nenner = 0 ist.
    int nenner = int.Parse(Console.ReadLine());
    int x = 7 / nenner;
    Console.WriteLine("7 / " + nenner + "=" + x);
}
catch (Exception)
{
    // Hier ist die Fehlerbehandlung.
    Console.WriteLine("Es wurde durch 0 dividiert.");
}
finally
{
    // Dieser Bereich wird immer aufgerufen.
    Console.WriteLine("finally");
}

```

Jener Teil des Codes der einen Fehler auslösen kann, wird durch den try-Code Block umschlossen.

Falls im oberen Code eine Division durch 0 durchgeführt wird, wird beim Dividieren eine Exception geworfen, die im catch-Teil aufgefangen wird. Es gibt verschiedene Klassen für Exceptions. Alle sind von der Basisklasse `System.Exception` abgeleitet. Im Falle einer Ganzzahldivision durch 0 wird ein Objekt der Exceptionklasse `System.DivideByZeroException` geworfen, das von `System.Exception` abgeleitet ist.

Oft können Methoden auch ganz unterschiedliche Exceptions werfen. In diesem Fall kann ein try-Block auch mehrere catch-Blöcke haben, die nacheinander angegeben werden.

Falls es Programmcode gibt, der sowohl im Fehlerfall, als auch im Nichtfehlerfall ausgeführt werden soll, steht dafür ein finally Block zur Verfügung.

```

string filepath = @"C:\Data\PKW.txt";
try
{
    StreamReader reader = new StreamReader(filepath);
    Console.WriteLine(filepath + " konnte geöffnet werden.");
}
catch (FileNotFoundException)
{
    Console.WriteLine("Das File {0} existiert nicht.", filepath);
}
catch (DirectoryNotFoundException)
{
    string directory = Path.GetDirectoryName(filepath);
    Console.WriteLine("Das Verzeichnis {0} existiert nicht.", directory);
}
catch (UnauthorizedAccessException)
{
    Console.WriteLine("Der Zugriff auf " + filepath + " ist nicht erlaubt.");
}
catch (Exception)
{
    Console.WriteLine("Komisch!");
}
finally
{
    Console.WriteLine("Dieser Teil wird immer ausgeführt.");
}

```

Der Konstruktor der Klasse `StreamReader` kann drei verschiedene Exceptions werfen. Das passiert dann, wenn im oberen Beispiel die Datei C:\Data\PKW.txt nicht geöffnet werden kann.

- Die `FileNotFoundException` Exception wird geworfen, wenn am Verzeichnis C:\Data die Datei PKW.txt nicht existiert.
- Die `DirectoryNotFoundException` Exception wird von Konstruktor dann geworfen, wenn auf C:\ das Verzeichnis Data nicht existiert, d. h. wenn nicht einmal der Dateipfad bis zur Datei vollständig existiert.
- Die `UnauthorizedAccessException` Exception wird ausgelöst, wenn die Datei zwar existiert, aber der Windows Benutzer kein Recht dafür hat, diese zu lesen.

Exceptions werden häufig im Fehlerfall verwendet. Tatsächlich gibt es auch Fälle in denen Exceptions nicht beim Auftreten von Fehlern, sondern beim Auftreten eines speziellen Zustands geworfen werden. Dieser Zustand muss nicht unbedingt ein Fehlerfall sein.

26.4 Exceptions selbst auslösen

In den oberen Beispielen wurden Exceptions gefangen die innerhalb des .NET-Frameworks geworfen wurden.

Falls in selbstgeschriebenem Code unvorhergesehene Fehlersituationen auftreten, müssen ebenfalls Exceptions geworfen werden. Das erfolgt durch die Anweisung `throw`.

Dazu wird nach `throw` mit `new` ein Objekt einer Exceptionklasse erzeugt und "geworfen".

26.5 Exceptionklassen selbst definieren

Das .NET-Framework hat eine ganz Fülle an Exceptionklassen zur Auswahl. Grundsätzlich ist es aber empfehlenswert selbst Klassen zu definieren. Das erfolgt denkbar einfach. Man definiert eine Klasse mit einem passenden Namen und leitet diese von einer bestehenden Exceptionklasse ab.

Die Basisklasse aller Exceptionsklassen ist die Klasse `Exception`, von der eigene Exceptions abgeleitet werden sollen.

Die `ApplicationException` soll dafür nicht verwendet werden.

<https://docs.microsoft.com/en-us/dotnet/api/system.applicationexception?view=net-5.0>

ⓘ Important

You should derive custom exceptions from the `Exception` class rather than the `ApplicationException` class. You should not throw an `ApplicationException` exception in your code, and you should not catch an `ApplicationException` exception unless you intend to re-throw the original exception.

C# Selbstdefinierte Exceptionklasse

```
class InvalidArtikelNameException : Exception
{
}

abstract class Artikel
{
    int Artikelnummer { get; }

    // Jedes Property, das überprüft werden muss, benötigt eine Klassenvariable
    string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (value != null && value != "")
            {
                // Alles ist ok, weil value vernünftig ist.
                name = value;
            }
            else
            {
                // Der Name ist ungültig. Wir werfen eine Exception, die an einer
                // anderen Stelle mit catch aufgefangen werden muß.
                throw new InvalidArtikelNameException();
            }
        }
    }
}
```

In der Praxis gibt man beim Werfen einer Exception häufig Daten mit, die im dazugehörigen catch-Block ausgelesen und verwendet werden können.

Man beachte: Im catch-Block besteht keine Möglichkeit herauszufinden, wo die aufgefangene Exception geworfen wurde! Deshalb muss man alle relevante Information mitgeben, z. B. die Artikelnummer im oberen Beispiel, damit im catch-Block darauf reagiert werden kann.

Der folgende Code zeigt auch, wie ein möglicher catch-Block aussehen könnte.

In Produktionscode müssten die Artikel natürlich einzeln hinzugefügt werden.

C# Selbstdefinierte Exceptionklasse

```
class InvalidArtikelNameException : Exception
{
    public int Artikelnummer { get; }

    public InvalidArtikelNameException( int artnr )
    {
        Artikelnummer = artnr;
    }
}
```

```

class Artikel
{
    int Artikelnummer { get; }

    // Jedes Property, das überprüft werden muss, benötigt eine Klassenvariable
    string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (value != null && value != "")
            {   // Alles ist ok, weil value vernünftig ist.
                name = value;
            }
            else
            {   // Der Name ist ungültig. Wir werfen eine Exception, die an einer
                // anderen Stelle mit catch aufgefangen werden muß.
                throw new InvalidArtikelNameException( Artikelnummer );
            }
        }
    }

    public Artikel(string name, double einkaufspreis)
    {
        Name = name;
        Einkaufspreis = einkaufspreis;
    }
}

// Der dazugehörige try-catch-Block könnte folgendermaßen aussehen.

try
{
    Artikel a1 = new Artikel(44000564, "Blueray Player BP250", 30.5, "LG");
    Artikel a2 = new Artikel(44000565, "", 650d, "Samsung");
    ...
}
catch (InvalidArtikelNameException ex)
{
    Console.WriteLine("Der Name des Artikels {0} ist ungültig.", ex.Artikelnummer);
}

```

Beim Auslösen einer Exception wird die Aufrufhierarchie vollständig aufgelöst und auf jeder Ebene der Aufrufhierarchie (des Callstacks) nach einem catch-Block gesucht, der die geworfene Exceptionklasse behandelt.

27 Namespaces

Alle Klassen einer Anwendung werden logisch throw in Namespaces (deutsch: Namensräume) aufgeteilt.

Jede Klasse (und damit auch deren dazugehöriges Assembly .obj-File) ist genau einem Namespace zugeordnet.

In C# kann ein .cs-File mehrere Klassen enthalten.

27.1 Definition von Namespaces

Ein Namespace wird durch das Schlüsselwort `namespace` definiert.

C# Beispiel

```
namespace AT_HTLDonaustadt_SLOG
{
    class Person
    {
        ...
    }
}
```

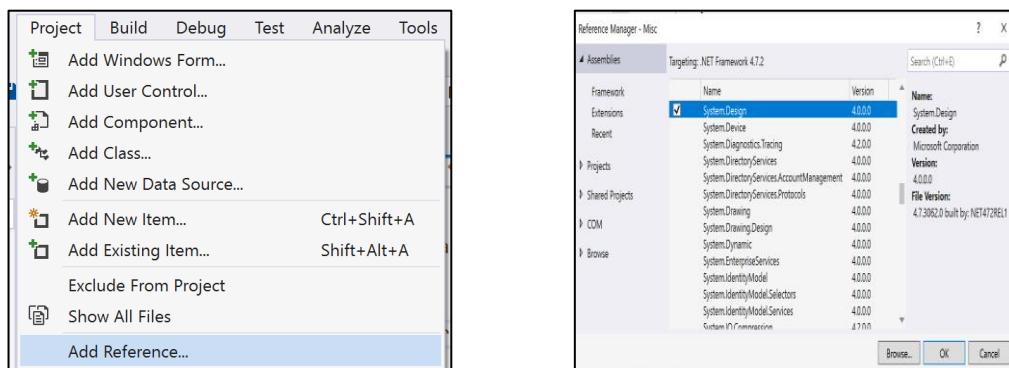
Enthält die Klasse `AT_HTLDonaustadt_SLOG.Person`.

Man kann in C# Namespaces auch nesten (verschachteln).

```
namespace AT
{
    namespace HTLDonaustadt
    {
        namespace SLOG
        {
            class Person
            {
                ...
            }
        }
    }
}
```

27.2 Verwendung von Klassen aus Namespaces

Um die Klassen eines Namespaces in einem .cs-File verwenden zu können, muss man zunächst das Assembly einbinden. Das erfolgt in Visual Studio über das Menü Project -> Add Reference. Dann kann das Assembly im Reference Manager hinzugefügt werden.



Um die Klassen des Assemblies nicht vollqualifiziert aufrufen zu müssen, kann man mit `using` Namespaces angeben, deren Klassen aufgerufen werden, z. B.

```
using System.Text;
using System.IO;
```

28 Generics

Generics dienen dazu, Klassen und Methoden mit Typparametern zu parametrisieren und so Code verallgemeinert verwenden zu können.

Generics gestatten es Klassen und Methoden (generic methods) universell einsetzbar zu machen, aber trotzdem völlige Typsicherheit zu garantieren.

C# Definition einer generischen Klasse

```
class Box<T>
{
    private T content;

    public void Fill(T content)
    {
        this.content = content;
    }

    public T Get()
    {
        return content;
    }

    public Box() { }

    public Box(T content)
    {
        Fill(content);
    }
}
```

Bei einer generischen Klasse (generic class) werden die Typparameter in Spitzklammern < > nach dem Klassennamen geschrieben. Box hat einen Typparameter T.

Die Klasse Box<T> kann nun in der Instanz a
Box<int> a = new Box<int>();
einen Integer speichern und in b
Box<string> b = new Box<string>();
einen String.

Der beim Anlegen der Instanz angegebene Typ wird zur Compilezeit für T eingesetzt.

C# Verwendung der generischen Klasse im Main

```
class Program
{
    static void Main(string[] args)
    {
        Box<int> boxint = new Box<int>();
        boxint.Fill(10);
        int value = boxint.Get();
        Console.WriteLine(value);

        var boxstring = new Box<string>("Pizza");
        string food = boxstring.Get();
        Console.WriteLine(food);
        boxstring.Fill("Cupcake");

        Console.WriteLine(boxstring.Get().ToUpper());
    }
}
```

Programmausgabe:

10
Pizza
CUPCAKE

Die populärste Anwendung von Generics sind die Collectionklassen, die im nächsten Abschnitt beschrieben werden.

29 Collections

Collection Klassen sind immer dann sinnvoll, wenn mehrere Instanzen in gleicher Art und Weise in einem Container gespeichert werden sollen.

- Collections werden auch als Container bezeichnet.
- Alle sind im Namespace `System.Collections.Generic` definiert und hier beschrieben:
[https://msdn.microsoft.com/en-us/library/system.collections.generic\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.collections.generic(v=vs.110).aspx)
- Bei (fast allen) Collections ist es extrem wichtig, die Kapazität (Capacity) möglichst früh zu setzen da spätere Anpassungen, also ein Resizen, sehr ineffizient ist. Viele Containerklassen erlauben es im Konstruktor die Capacity mitzugeben. Nähere Informationen findet man z. B. hier. <https://www.dotnetperls.com/capacity>

29.1 Übersicht

Die Collections lassen sich in C# in vier Gruppen einteilen: Listen, Mengen (Sets), Dictionaries und Queues.

Die folgende Tabelle zeigt eine ganz knappe Übersicht der wichtigsten Collections in C#.

C#	Einige Eigenschaften	
<code>List<T></code>	Lineare Liste (als Array implementiert!): <ul style="list-style-type: none">• Zugriff über Index• Schnelles Lesen +• Langsames Einfügen und Löschen —	
<code>ArrayList</code>	Speichert nur Instanzen des Types <code>Object</code> —	
<code>LinkedList<T></code>	Doppeltverkettete Liste: <ul style="list-style-type: none">• Zugriff über Index• Langsames Lesen —• Schnelles Einfügen und Löschen +	
<code>HashSet<T></code>	Unsortiert	Menge: <ul style="list-style-type: none">• Enthält keine doppelten Elemente• Kein Zugriff über Index
<code>SortedSet<T></code>	Sortiert	
<code>Dictionary< TKey , TValue ></code>	Unsortiert	Speichert Wertepaare (Assoziativspeicher): <ul style="list-style-type: none">• Zugriff über Key• Enthält keine doppelten Schlüssel <code>TKeys</code>
<code>SortedDictionary< TKey , TValue ></code>	Sortiert	
<code>Stack<T></code>	Stapel: <ul style="list-style-type: none">• LIFO (Last in first out)• Push-, Pop-, Peek-Operationen	
<code>Queue<T></code>	Stapel: <ul style="list-style-type: none">• FIFO (First in first out)• Enqueue- und Dequeue-Operationen• Peek-Operation	

29.2 List

Listen werden ähnlich wie Arrays verwendet.

- Ist in C# intern als Array definiert
- Wird im Speicher immer als zusammenhängender Block reserviert.
- Ist die **Kapazität** einer Liste ausgeschöpft, muss für ein weiteres Element ein neuer Block reserviert werden und **alle Elemente** des alten Arrays in das neue umkopiert werden!

29.2.1 Wichtige Properties und Methoden

Der folgende Überblick zeigt die wichtigsten Properties und Methoden.

Aktion	Methode/Property	Beispiel		
		Vorher	Code	Nachher
Erzeugung	Konstruktor		List<int> liste = new List<int>();	{ }
Hinzufügen:	Add(wert)	{ }	liste.Add(2); liste.Add(6); liste.Add(4); liste.Add(1); liste.Add(2);	2 → 6 → 4 → 1 → 2
Länge:	Property Count	2 → 6 → 4 → 1 → 2	int cnt = liste.Count;	cnt = 5
Zugriff:	[]-Operator	2 → 6 → 4 →	int z = liste[1];	z = 6
Entfernen:	Remove(wert)	1 → 2	liste.Remove(4);	2 → 6 → 1 → 2
	RemoveAt(index)		liste.RemoveAt(2);	2 → 6 → 1 → 2
Existenz:	Contains(wert)	2 → 6 → 1 → 2	bool b1 = liste.Contains(4); bool b2 = liste.Contains(6);	b1 = False b2 = True
Positions-suche:	IndexOf(wert)		int pos1 = list.IndexOf(6); int pos2 = list.IndexOf(4);	pos1 = 1 pos2 = -1
Sort:	Sort() oder Sort(Comparer)	2 → 6 → 1	list.Sort(); list.Sort(new Comparer());	1 → 2 → 6
Reverse:	Reverse()	1 → 2 → 6	liste.Reverse();	6 → 2 → 1
Löschen:	Clear()	1 → 2 → 6	liste.Clear();	Liste ist leer

29.2.2 Nachteile von List

Da die Klasse List intern Arrays verwendet, ist bei dieser Klasse Vorsicht geboten:

- **Einfügen** erfordert das Umkopieren der nachfolgenden Elemente
- **Löschen** erfordert das Umkopieren der nachfolgenden Elemente
- Überschreiten der **Kapazität** erfordert das Umkopieren **aller** Elemente

Beispiel:

Im unteren Beispiel, speichert ein Array arr aufsteigend sortierte Ganzzahlwerte. Es soll nun der negative Wert -3 aus dem Array entfernt und anschließend der Wert 7 in das Array an der richtigen Stelle eingefügt werden.

Da ein Array immer einen einzigen Speicherbereich besetzt, kann nicht einfach ein Teil davon entfernt werden, sondern es muss das zu entfernende Element überschrieben werden.

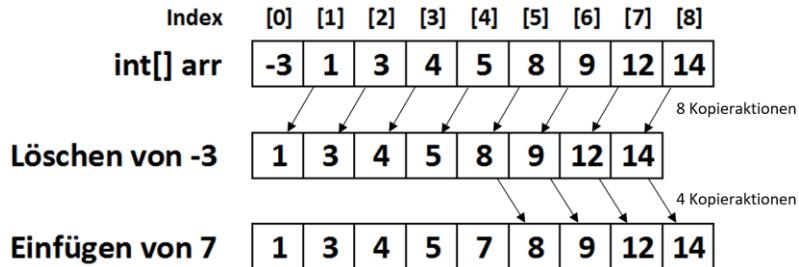


Abbildung 34: Array-Operationen beim Löschen und Einfügen von Elementen

Man sieht, dass es zahlreichen Schreibaktionen bedarf, die recht einfachen Operationen auszuführen.

29.2.3 Code Sample

C# List Sample

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

namespace ListSample
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> list = new List<int>() {4, 6, 8};
            Console.WriteLine(String.Join(',',list));

            list.Add(6);      // single add
            list.Insert(3, 5); // insert 5 at index 3

            Console.WriteLine("numbers : " + String.Join(',',list));

            list.RemoveAt(2); // remove by index

            Console.WriteLine("removed index 2");
            Console.WriteLine("numbers : " + String.Join(',',list));

            list.Remove(6); // remove by value

            Console.WriteLine("removed (first) value 6");
            Console.WriteLine("numbers : " + String.Join(',',list));

            Console.WriteLine("1st element: "+list[0]); // access by index, [] in C#
            Console.WriteLine("2nd element: "+list[1]);
            Console.WriteLine("last element: "+list[list.Count-1]);
        }
    }
}
```

Programmausgabe:

```
4,6,8
numbers : 4,6,8,5,6
removed index 2
numbers : 4,6,5,6
removed (first) value 6
numbers : 4,5,6
1st element: 4
2nd element: 5
last element: 6
```

29.3 LinkedList und LinkedListNode

LinkedList arbeiten intern nicht mit Arrays, sondern mit Knoten (Nodes) die miteinander dynamisch verknüpft sind. Die Klasse dieser Nodes ist `LinkedListNode`.

Jeder Knoten zeigt mit dem Property `Next` auf den Folgeknoten und mit dem Property `Previous` auf den Vorgängerknoten.

Der letzte Knoten hat keinen Nachfolger und `Next` ist daher `null`.

Der erste Knoten hat keinen Vorgänger und `Previous` speichert daher `null`.

Die Klasse `LinkedList` hat zwei Properties, die auf den ersten und letzten Knoten zeigen, `First` zeigt auf den ersten Knoten, `Last` auf den letzten Knoten.

Beispiel: Die folgende `LinkedList` repräsentiert die Folge -3, 1, 3 und 4.

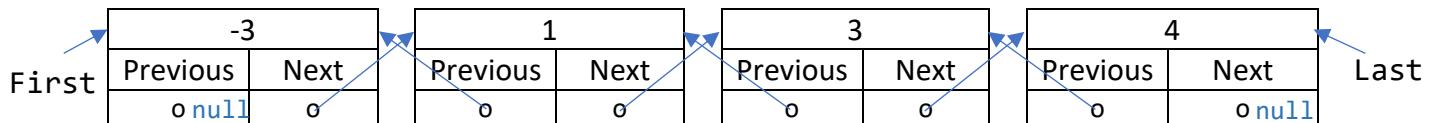


Abbildung 35: Knotenliste einer `LinkedList`

29.3.1 Löschen eines Elements in `LinkedList`

Um das Element 1 zu löschen, muss man intern lediglich den entsprechenden Knoten aus der Liste aushängen.

Diese geschieht dadurch, dass beim Vorgänger von 1, also -3, das Property `Next` auf 3 zeigen muss und das `Previous`-Property von dem Nachfolgeknoten 3 auf -3.

Nur durch diese beiden Schreiboperationen wird das Element 1 gelöscht.

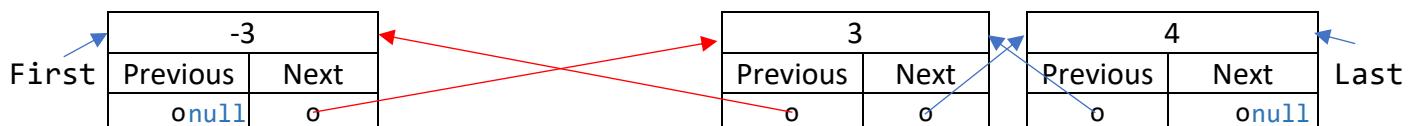


Abbildung 36: Knotenliste einer `LinkedList` beim Löschen eines Knotens

Der zu lösrende Knoten 1 zeigt zwar selbst noch auf seinen Vorgänger und Nachfolger, allerdings wird er von der Garbage Collection gelöscht, da keine Variable auf ihn referenziert.

29.3.2 Löschen des ersten Elements in `LinkedList`

Wenn der zu lösende Knoten der allererste Knoten ist, muss das `LinkedList`-Property `First` auf den neuen ersten Knoten gesetzt werden.

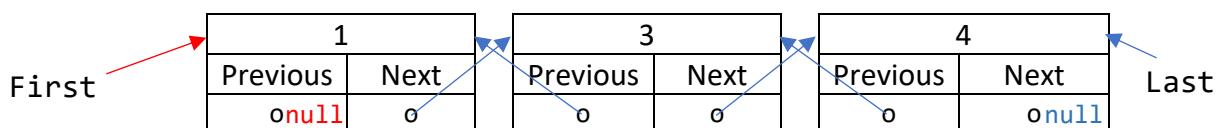


Abbildung 37: Knotenliste einer `LinkedList` beim Löschen des ersten Knotens

29.3.3 Löschen des letzten Elements in LinkedList

Wenn der zu löschenende Knoten der allerletzte Knoten ist, muss das `LinkedList`-Property `Last` auf den vorher vorletzten Knoten gesetzt werden.

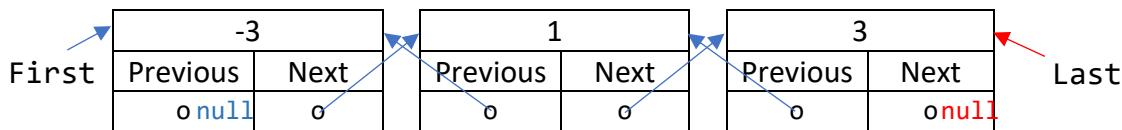


Abbildung 38: Knotenliste einer `LinkedList` beim Löschen des letzten Knotens

29.3.4 Einfügen eines Elements innerhalb der LinkedList

Um ein Element 5 in der `LinkedList` vor vorherigen Abschnitt 0 zwischen 1 und 3 einzufügen, muss ein Knoten mit 5 als Wert angelegt werden und dann dieser mit dem vorderen und hinteren Knoten verknüpft werden.

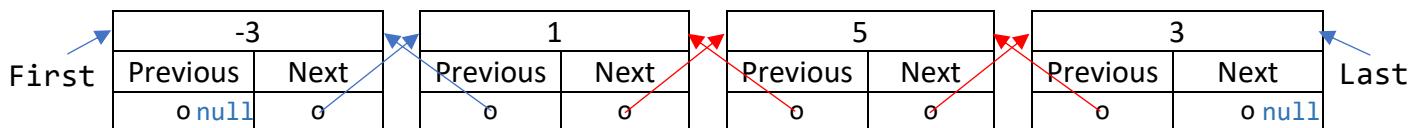


Abbildung 39: Knotenliste einer `LinkedList` beim Einfügen eines Knotens

29.3.5 Einfügen am Beginn einer LinkedList

Um ein Element 5 in der `LinkedList` des Abschnitts 0 am Beginn einzufügen, muss ein Knoten mit 5 als Wert angelegt werden und dann dieser mit dem bisher vorderen Knoten verknüpft werden. Außerdem wird das `LinkedList`-Property `First` verknüpft.

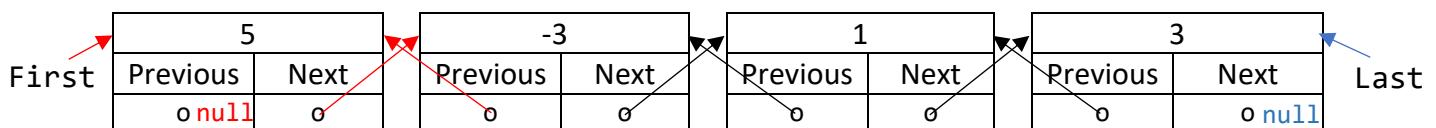


Abbildung 40: Knotenliste einer `LinkedList` beim Einfügen am Beginn der Liste

29.3.6 Einfügen am Ende einer LinkedList

Um ein Element 5 in der `LinkedList` des Abschnitts 0 am Ende einzufügen, muss ein Knoten mit 5 als Wert angelegt werden und dann dieser mit dem bisher hintersten Knoten verknüpft werden. Außerdem wird das `LinkedList`-Property `Last` verknüpft.

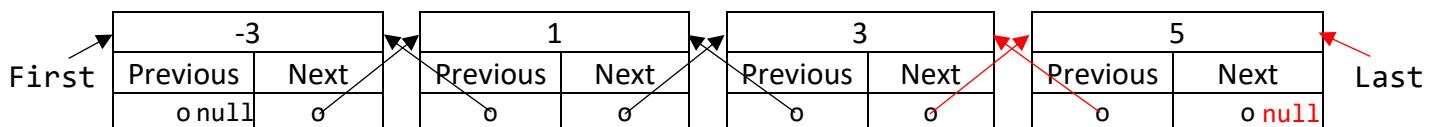


Abbildung 41: Knotenliste einer `LinkedList` beim Einfügen am Ende der Liste

29.3.7 Wichtige Properties und Methoden

Die wichtigsten Properties der `LinkedList` sind genau die oben in den Beispielen erwähnten Properties `First` und `Last`, sowie die Anzahl aller Elemente `Count`.

Die wichtigsten Properties der Knotenklasse `LinkedListNode` sind `Previous`, `Next` und `Value`. `Value` gibt den Wert des Knoten zurück, also in den oberen Beispielen den Zahlenwert.

29.3.8 Code Sample

C# LinkedList Sample

```
using System;
using System.Collections.Generic;
namespace LinkedListSample
{
    class Program
    {
        static void Main(string[] args)
        {
            LinkedList<int> list = new LinkedList<int>(new[] { 4, 6, 8 });
            Console.WriteLine(String.Join(',',list));

            list.AddLast(3); // single add at end
            list.AddFirst(5); // single add at begin

            Console.WriteLine("numbers : " + String.Join(',',list));

            list.Remove(4); // remove by value
            Console.WriteLine("removed element 4");
            Console.WriteLine("numbers : " + String.Join(',',list));
            Console.WriteLine("1st element: "+list.First.Value);
            Console.WriteLine(String.Join(',',list));
            Console.WriteLine("last element: "+list.Last.Value);

            for ( var node = list.First; node != null; node = node.Next) // forward
            {
                Console.Write(node.Value);
                if ( node != list.Last )
                    Console.Write(" - ");
            }
            for ( var node = list.Last; node != null; node = node.Previous) //backward
            {
                Console.Write(node.Value);
                if ( node != list.First )
                    Console.Write(" - ");
            }
        }
    }
}
```

Programmausgabe:

```
4,6,8
numbers : 5,4,6,8,3
removed element 4
numbers : 5,6,8,3
1st element: 5
5,6,8,3
last element: 3
5 - 6 - 8 - 3
3 - 8 - 6 - 5
```

29.4 ArrayList

Ist eine Liste von ganz allgemeinen Objekten. **ArrayList entspricht einer List<object>.**

Eine ArrayList verhält sich zu einer **List<object>**, ähnlich wie ein int-Array **int[]** zu einem **object[]**.

Die Verwendung der Klassen sollte möglichst vermieden werden, da Objekte beliebiger Datentypen enthalten sein können. Beim Hinzufügen wird das konkrete Objekt boxed (Boxing), beim Herauslesen ist Unboxing erforderlich. Das ist unnötig umständlich.

29.4.1 Wichtige Properties und Methoden

Aktion	Methode	Codebeispiel mit Ergebnis	
Erzeugung	Konstruktor	ArrayList liste = new ArrayList();	
Hinzufügen:	Methode Add	liste.Add("Ich"); liste.Add(2); liste.Add(new DateTime(2016,2,1));	"Ich" "Ich" → 2 "Ich" → 2 → (1. 2. 2016)
Alle anderen Methoden und Properties sind sehr ähnlich zur List<T>.			

29.5 HashSet

Die Klasse ist stark typisiert, d. h. dass T ein bestimmter Typ ist. Nur Objekte von diesem Typ sind (oder von diesem Typ abgeleitet sind), können zur Menge hinzugefügt werden.

Ein HashSet entspricht einem einer mathematischen Menge. Jedes Element kann NUR EINMAL enthalten sein bzw. um exakt zu sein: Es können keine zwei Instanzen in einem HashSet sein, die denselben Hash Code haben.

29.5.1 Methode GetHashCode()

Ein Hash Code ist eine Ganzzahl die eine Instanz identifiziert. Jede Instanz hat in C# eine Methode **GetHashCode()** die einen Hash Code zurückgibt.



Erklärt was Coding Kurzgeschichten-Videos die HashSets erklären und anwenden.

	GetHashCode() in C# und warum diese Methode bei HashSets und Dictionaries so wichtig ist Coding Kurzgeschichten https://www.youtube.com/watch?v=8oG-Xs8jFUI
	HashSets in C#: Collections ohne Doppelte Coding Kurzgeschichten https://www.youtube.com/watch?v=l67Ox0f8ACc



Lottoziehung mit HashSet in C#

Coding Kurzgeschichten

Lottoziehung 4:06

https://youtu.be/E_mNsQ50WYI

Zeigt ein Anwendungsbeispiel für HashSets.

Video 12: Coding Kurzgeschichten-Video über HashSets

29.5.2 Beispiel: Lottotipps 6 aus 45

HashSets sind sehr ideal, wenn man eine bestimmte Anzahl von unterschiedlichen Elementen benötigt, wie z. B. eine Anzahl von zufälligen Lottozahlen von 1 bis 45.

C# Fünf zufällige Tipps für 6 aus 45

```
static void Main(string[] args)
{
    const int anzahl = 5;
    var tipp = new HashSet<int>();

    Console.WriteLine("Lottoziehung");

    var rnd = new Random();

    for (int z = 1; z <= anzahl; z++)
    {
        tipp.Clear();

        while (tipp.Count != 6) // wird nur verlassen, wenn sechs unterschiedliche
        {                      // Zahlen von 1 bis 45 in tipp gespeichert sind.
            tipp.Add(rnd.Next(1, 46));
        }

        Console.Write($"{z}. Tipp: ");

        foreach (var zahl in tipp)
        {
            Console.Write($"{zahl} ");
        }

        Console.WriteLine();
    }
}
```

Programmausgabe:

Lottoziehung

1. Tipp: 13 15 3 36 27 9
2. Tipp: 18 34 20 15 12 1
3. Tipp: 19 21 3 35 26 24
4. Tipp: 31 18 33 40 11 15
5. Tipp: 17 26 10 35 7 15

29.5.3 Wichtige Properties und Methoden

HashSets erlauben grundsätzlich keinen indexbasierten Zugriff mit [] oder RemoveAt! HashSets sind schneller im Zugriff als Listen, da sie den Hash-Code von Instanzen verwenden.

Aktion	Methode/Property	Codebeispiel mit Ergebnis		
Erzeugung	Konstruktor	HashSet<int> menge = new HashSet<int>();	{ }	
Hinzufügen:	Methode Add liefert zurück ob das Element hinzugefügt wurde	bool b1 = menge.Add(2); bool b2 = menge.Add(6); bool b3 = menge.Add(2); bool b4 = menge.Add(1); bool b5 = menge.Add(5);	b1 = True b2 = True b3 = False b4 = True b5 = True	{ 2 } { 2, 6 } { 2, 6 } { 2, 6, 1 } { 2, 6, 1, 5 }
Länge:	Property Count	int cnt = liste.Count;	cnt = 4	
Zugriff:	KEIN []	foreach (int i in menge) { .. }		
Entfernen:	Methode Remove	bool b1 = menge.Remove(6); bool b2 = menge.Remove(8);	b1 = True b2 = False	{ 2, 1, 5 } { 2, 1, 5 }
Suche:	Methode Contains	bool b1 = menge.Contains(4); bool b2 = menge.Contains(5);	b1 = False b2 = True	
Löschen:	Methode Clear();	menge.Clear();	{ 2, 1, 5 }	{ }

29.5.4 Code Sample

C# HashSet Sample

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

namespace HashSetSample
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array = { 4, 6, 8, 6, 5, 9, 8 };
            HashSet<int> numbers = new HashSet<int>();
            foreach (int number in array)
            {
                bool inserted = numbers.Add(number);
                Console.WriteLine(number + ( inserted ? " was inserted."
                                              : " was not inserted."));
            }

            Console.WriteLine("\nnumbers : " + String.Join(',', numbers));

            HashSet<int> squares = new HashSet<int>(new[] {4, 9, 16, 25});

            Console.Write("remove from numbers all of : " + String.Join(',', squares));
            numbers.ExceptWith(squares);
            Console.WriteLine(" -> numbers : " + String.Join(',', numbers));

            Console.Write("union numbers with : " + String.Join(',', squares));
            numbers.UnionWith(squares);
            Console.WriteLine(" -> numbers : " + String.Join(',', numbers));

            Console.WriteLine("Even numbers:");

            foreach (var number in numbers.Where(x=> x % 2 == 0))
            {
                Console.WriteLine(number);
            }
        }
    }
}
```

Programmausgabe:

```
4 was inserted.
6 was inserted.
8 was inserted.
6 was not inserted.
5 was inserted.
9 was inserted.
8 was not inserted.
```

```
numbers : 4,6,8,5,9
```

```
remove from numbers all of : 4,9,16,25 -> numbers : 6,8,5
union numbers with : 4,9,16,25 -> numbers : 9,6,8,5,4,16,25
```

```
6
8
4
16
```

29.6 SortedSet

Ein SortedSet entspricht einer sortierten Menge. Die Methoden sind weitgehend gleich wie für ein HashSet<T>, allerdings ComparerKlasse angegeben werden.

Alle Objekte in einem SortedSet sind immer sortiert. Wird ein Objekt mit Add hinzugefügt, wird es SOFORT an die richtige Stelle sortiert (siehe Programm *SortedSet am Share*).

29.6.1 Wichtige Properties und Methoden

Aktion	Methode/Property	Codebeispiel mit Ergebnis		
Erzeugung	Konstruktoren	<pre>SortedSet<int> menge = new SortedSet <int>(); SortedSet<int> menge = new SortedSet <int>(comparer);</pre>	{ }	
Hinzufügen:	Methode Add liefert zurück ob das Element hinzugefügt wurde	<pre>bool b1 = menge.Add(2); bool b2 = menge.Add(6); bool b3 = menge.Add(2); bool b4 = menge.Add(1);</pre>	b1 = True b2 = True b3 = False b4 = True	{ 2 } { 2, 6 } { 2, 6 } { 1, 2, 6 }
Kleinstes/ größtes Element	Property Min Property Max	<pre>int min = menge.Min; int max = menge.Max;</pre>	Min = 1 Max = 6	

29.6.2 Beispiel: Lottotipps 6 aus 45

Um die Lottozahlen des Programms im Abschnitt 0 aufsteigend zu sortieren, reicht es aus, ein SortedSet statt dem HashSet zu verwenden.

C# Fünf zufällige Lottotipps für 6 aus 45 (wie Abschnitt 0) – Aufsteigend sortiert

```
static void Main(string[] args)
{
    ...
    var tipp = new SortedSet<int>();
    ...
}
```

Programmausgabe:

Lottoziehung

1. Tipp: 2 7 8 9 17 19
2. Tipp: 6 13 19 25 28 45
3. Tipp: 12 34 35 39 41 43
4. Tipp: 5 10 20 25 34 37
5. Tipp: 10 14 24 33 36 38

Um die Lottozahlen absteigend zu sortieren, müsste man einen entsprechenden Comparer implementieren und dem Konstruktor des SortedSets übergeben.

C# Fünf zufällige Lottotipps für 6 aus 45 (wie Abschnitt 0) – Absteigend sortiert

```
class DescendingComparer : IComparer<int>
{
    public int Compare( int a, int b )
    {
        return -a.CompareTo(b);
    }
}

static void Main(string[] args)
{
    ...
    var tipp = new SortedSet<int>(new DescendingComparer());
    ...
}
```

Programmausgabe:

Lottoziehung

1. Tipp: 45 42 32 19 11 2
2. Tipp: 39 35 23 17 13 3
3. Tipp: 45 35 25 22 20 17
4. Tipp: 24 18 17 9 4 2
5. Tipp: 39 34 31 23 15 13

29.6.3 Code Sample

C# SortedSet Sample

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

namespace SortedSetSample
{
    class OddBeforeEvenComparer : IComparer<int>
    {
        public int Compare(int o1, int o2)
        {
            if (o1 == o2)
                return 0;

            bool odd1 = o1 % 2 == 1; // o1 is odd
            bool odd2 = o2 % 2 == 1; // o2 is odd

            return odd1 == odd2
                ? o1.CompareTo(o2) // if both numbers are even or odd, order ascending.
                : odd1 && !odd2
                    ? -1 // o1 is odd and o2 is even
                    : 1; // o1 is even and o2 is odd
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            int[] array = { 4, 6, 8, 6, 5, 9, 8 };
            SortedSet<int> numbers = new SortedSet<int>(new OddBeforeEvenComparer() );

            foreach (int number in array)
            {
                bool inserted = numbers.Add(number);
                Console.WriteLine(number + (inserted ? " was inserted."
                                                : " was not inserted."));
            }

            Console.WriteLine("\nnumbers : " + String.Join(',', numbers));

            HashSet<int> squares = new HashSet<int>(new[] { 4, 9, 16, 25 });
            Console.Write("remove from numbers all of : " + String.Join(',', squares));
            numbers.ExceptWith(squares);
            Console.WriteLine(" -> numbers : " + String.Join(',', numbers));

            Console.Write("union numbers with : " + String.Join(',', squares));
            numbers.UnionWith(squares);
            Console.WriteLine(" -> numbers : " + String.Join(',', numbers));

            Console.WriteLine("Even numbers:");
            foreach (var number in numbers.Where(x=> x % 2 == 0))
            {
                Console.WriteLine(number);
            }
        }
    }
}
```

Programmausgabe:

4 was inserted.
6 was inserted.
8 was inserted.

```

6 was not inserted.
5 was inserted.
9 was inserted.
8 was not inserted.

numbers : 5,9,4,6,8
remove from numbers all of : 4,9,16,25 -> numbers : 5,6,8
union numbers with : 4,9,16,25 -> numbers : 5,9,25,4,6,8,16
Even numbers:
4
6
8
16

```

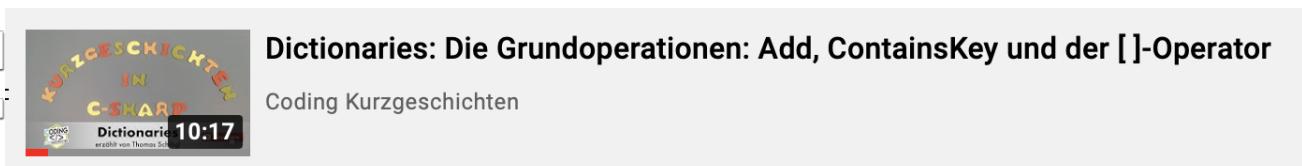
29.7 Dictionary

Ein Dictionary speichert eine beliebige Anzahl von Paaren von Objekten ab. Jedes dieser Paare besteht aus einem Schlüssel (engl. Key) und einem Wert (engl. Value).

Key → Value

"Dictionary" ist das englische Wort für Wörterbuch. Die Datenstruktur heißt so, weil es mit ihr möglich ist, mit einem Schlüssel einen Wert "nachzuschlagen". Der Zugriff auf den Wert, mit Hilfe des Schlüssels ist sehr schnell, weil auch hier wie bei einem HashSet der Hash Code einer Instanz verwendet wird.

Das [Coding Kurzgeschichten](https://youtu.be/3iTUy9GMEbk)-Video <https://youtu.be/3iTUy9GMEbk> erklärt die Grundlagen von Dictionaries und die wichtigsten Methoden.



Video 13: Coding Kurzgeschichten-Video über Dictionaries

Der Screenshot aus dem Video zeigt die Paare von Keys und Values die in einem Dictionary gespeichert sind.

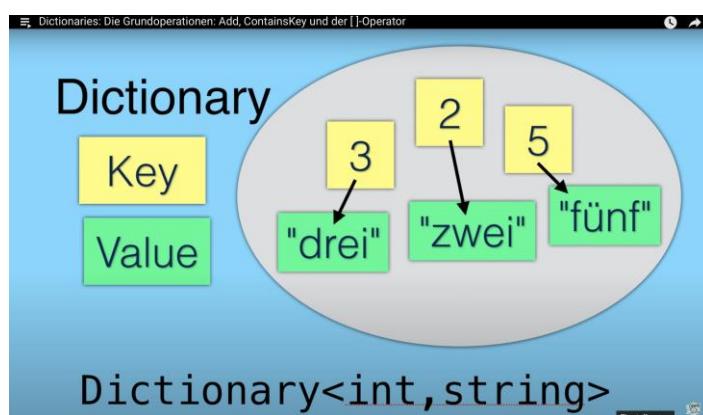


Abbildung 42: KeyValuePairs eines Dictionaries

29.7.1 Beispiel: Fußballspieler und Alter

In einem Programm sollen Namen von Fußballspielern und deren Alter abgespeichert werden. Nach der Benutzereingabe eines Namens soll das Alter des jeweiligen Spielers ausgegeben werden.

Lösung ohne Dictionary:

1. Man definiert eine Klasse `Fußballspieler` mit den Properties `Name` und `Alter`.
 2. Man legt für jeden Spieler ein Objekt an und speichert es in einer Liste `List<Fußballspieler>`.
 3. Das Programm liest mit `Console.ReadLine()` den Namen eines Fußballspielers ein.
 4. Das Programm durchsucht die gesamte Liste aller Fußballspieler bis das Objekt mit dem entsprechenden Name gefunden ist und gibt das dazugehörige Alter aus.
- Diese Lösung ist bei einer großen Anzahl von Spielern viel zu langsam!

Lösung mit einem Dictionary:

1. Im Programm wird ein Dictionary definiert, in das man Namen und Alter abspeichern kann. Der Key des Dictionaries ist der Name, der Wert des Dictionaries das Alter, also ist `Name → Alter`.

```
Dictionary<string, int> d = new Dictionary<string, int>();
```

2. Hinzufügen der Paare ins Dictionary

```
d.Add("Messi", 29);  
d.Add("Ronaldo", 31);  
d["Alaba"] = 24;
```

3. Eingabe des Namens durch den Benutzer: `string name = Console.ReadLine();`

4. Auslesen des Alters durch Angabe des Namens und dem []-Operator: `int alter = d[name];`

Zugriff	Codebeispiel	Beschreibung
Lesen:	<code>int alter = d["Messi"];</code>	weist der Variable <code>alter</code> den Wert 29 zu
Ändern:	<code>d["Messi"] = 30;</code>	setzt das Alter von Messi im Dictionary auf 30
Hinzufügen auf zwei Arten:	<code>d.Add("Alaba", 24);</code> <code>d["Alaba"] = 24</code>	fügt Alaba zum Dictionary hinzu
Löschen:	<code>d.Remove("Ronaldo");</code>	

Dictionaries zählen zu den am häufigsten verwendeten Collections in Programmen, da viele Aufgabenstellungen einen raschen Zugriff auf Werte basierend auf Keys erfordern.

29.7.2 Wichtige Properties und Methoden

Aktion	Methode/Property	Codebeispiel mit Ergebnis	
Erzeugung	Konstruktor	<code>Dictionary<string, int> d = new Dictionary<string, int>();</code>	
Hinzufügen:	Methode Add oder der []-Operator	<code>d.Add("Messi", 29); d.Add("Ronaldo", 31); d["Alaba"] = 24;</code>	
Länge:	Property Count	<code>int cnt = d.Count;</code>	<code>cnt = 3</code>
Zugriff:	[]	<code>d["Alaba"]</code>	<code>24</code>
	TryGetValue	<code>bool b = d.TryGetValue("Ronaldo", out alter); alter ist danach 31</code>	
Entfernen:	Methode Remove	<code>d.Remove("Messi");</code>	
Suche:	ContainsKey ContainsValue	<code>bool b1 = d.ContainsKey("Buffon"); bool b2 = d.ContainsValue(25);</code>	<code>b1 = False b2 = False</code>
Löschen:	Methode Clear();	<code>d.Clear();</code>	

29.7.3 Code Sample

C# Dictionary Sample

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

namespace DictionarySample
{
    class Program
    {
        static void Main(string[] args)
        {
            Dictionary<string, int> name2age = new Dictionary<string, int>()
            {
                { "Messi", 31 },
                { "Ronaldo", 33 },
                { "Ronaldinho", 33 },
                { "Alaba", 26 },
            };

            // loop through a dictionary: using a foreach loop
            foreach (var entry in name2age)
            {
                Console.WriteLine($"{entry.Key} -> {entry.Value}");
            }

            Console.Write("Enter a name: ");
            string name = Console.ReadLine();

            if (name2age.ContainsKey(name))
            {
                Console.WriteLine($"{name} is {name2age[name]} years old.");
            }
            else
            {
                Console.WriteLine($"Cannot find {name} in the dictionary.");
            }
        }
    }
}
```

Programmausgabe:

Messi -> 31

Ronaldo -> 33

Ronaldinho -> 33

Alaba -> 26

Enter a name: Ronaldo

Ronaldo is 33 years old.

29.7.4 SortedDictionary

SortedDictionary ist für ein Dictionary exakt so wie ein SortedSet für ein HashSet. Über einen Comparer können die Keys sortiert werden.

C# SortedDictionary Sample

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

namespace SortedDictionarySample
{
    class LengthComparer : IComparer<string>
    {
        public int Compare(string o1, string o2)
        {
            if (o1 == null)
                return o2 == null ? 0 : -1;

            if (o2 == null)
                return 1;

            int result = o1.Length.CompareTo(o2.Length);

            if (result == 0)
                result = o1.CompareTo(o2);

            return result;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var name2age = new SortedDictionary<string, int>(new LengthComparer())
            {
                { "Messi", 31 }, { "Ronaldo", 33 }, { "Ronaldinho", 33 }, {"Alaba", 26},
            };

            // loop through a dictionary: using a foreach loop
            foreach (var entry in name2age)
            {
                Console.WriteLine($"{entry.Key} -> {entry.Value}");
            }

            Console.Write("Enter a name: ");
            string name = Console.ReadLine();

            Console.WriteLine(name2age.ContainsKey(name)
                ? $"{name} is {name2age[name]} years old."
                : $"Cannot find {name} in the dictionary.");
        }
    }
}
```

Programmausgabe:

```
Alaba -> 26
Messi -> 31
Ronaldo -> 33
Ronaldinho -> 33
Enter a name: Arnautovic
Cannot find Arnautovic in the dictionary.
```

29.8 Stack

Einen Stack kann man sich als Stapel von Elementen vorstellen (oder wie ein Stapel von Spielkarten)

- auf den man ein Element darauflegen kann (Push),
- von dem man ein Element herunterholen kann (Pop) und
- bei dem man das obere Element anschauen kann (Peek).

Diese drei Aktionen sind die Grundfunktionen eines Stacks. Diese Verhaltensweise könnte man auch durch eine `LinkedList` leicht realisieren, allerdings ist die Stärke eines Stacks gerade, dass man sich darauf verlassen kann, dass nur das alleroberste Element zur Verfügung steht. Darunterliegende Elemente können nicht entfernt werden, ohne vorher alle oberen Elemente entfernen zu können.

Stacks werden meistens so dargestellt, dass das zuletzt hinzugefügte Element das oberste ist. Siehe dazu auch <https://de.wikipedia.org/wiki/Stapelspeicher>

- Die Elemente werden also in umgekehrter Reihenfolge entnommen als sie hinzugefügt wurden.
- Aus diesem Grund eignet sich ein Stack zum Beispiel dafür die Reihenfolge von Elementen umzukehren.

Dieses Prinzip nennt man entweder **LIFO** (Last-In, First-Out) oder **FILO** (First-In, Last-Out).

29.8.1 Wichtige Properties und Methoden

Aktion	Methode/Property	Beispiel		
		Vorher	Code	Nachher
Erzeugung	Konstruktor		<code>Stack<int> = new Stack<int>();</code>	leer
Hinzufügen:	Push(wert)	leer	<code>s.Push(2); s.Push(6); s.Push(14);</code>	[14] [6] [2]
Zugriff:	Peek()	[14] [6] [2]	<code>int top = s.Peek();</code>	[14] [6] [2] top = 3
Entfernen:	Pop()	[14] [6] [2]	<code>int top = s.Pop();</code>	[6] [2] top = 3
Länge:	Property Count	[14] [6] [2]	<code>int cnt = s.Count();</code>	cnt = 3
Löschen:	Clear()	[14] [6] [2]	<code>s.Clear();</code>	leer

29.8.2 Code Sample

C# Stack Sample

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

namespace StackSample
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack<string> books = new Stack<string>();
            books.Push("Astrology");      // Hinzufügen zum Stack mit Push
            books.Push("Biology");
            books.Push("Computation");

            // Ausgabe aller Elemente des Stapels (Stacks)
            Console.WriteLine(String.Join(',', books));
            Console.WriteLine($"Es gibt {books.Count} Bücher.");

            // Das oberste Element anzeigen
            string first = books.Peek(); // Gibt das oberste Element Computation zurück
            Console.WriteLine($"Das oberste Buch ist {first}.");
            Console.WriteLine($"Es gibt {books.Count} Bücher.");

            string top = books.Pop();   // Holt das oberste Element vom Stapel.
            Console.WriteLine($"{top} wurde entfernt.");
            Console.WriteLine($"Es gibt nur noch {books.Count} Bücher.");
            Console.WriteLine(String.Join(',', books));

            Console.WriteLine($"Das oberste Buch ist jetzt {books.Peek()}.");
            Console.WriteLine(books.Pop()); // -> Biology wird entfernt
            Console.WriteLine(String.Join(',', books));

            Console.WriteLine($"Die oberste Buch ist jetzt {books.Peek()}.");
            books.Pop(); // -> Astrology wird entfernt

            if (books.Count > 0)
                Console.WriteLine(books.Pop());
            else
                Console.WriteLine("Es gibt kein Buch mehr am Stapel.");
        }
    }
}
```

Programmausgabe:

```
Computation,Biology,Astrology
Es gibt 3 Bücher.
Das oberste Buch ist Computation.
Es gibt 3 Bücher.
Computation wurde entfernt.
Es gibt nur noch 2 Bücher.
Biology,Astrology
Das oberste Buch ist jetzt Biology.
Biology
Astrology
Die oberste Buch ist jetzt Astrology.
Es gibt kein Buch mehr am Stapel.
```

29.9 Queue

Eine Queue kann man sich als Warteschlange von Elementen vorstellen, z. B. eine Warteschlange von Personen vor einer Supermarktkassa,

- bei der Elemente hintereinander gereiht werden (Enqueue),
- wo das „vorderste“ Element entfernt wird (Dequeue) und
- bei dem man nachschauen kann, welches Element als nächstes dran ist (Peek).



Quelle: [Cleanpng: Social Service Background](#)

Diese drei Aktionen sind die Grundfunktionen einer Queue. Diese Verhaltensweise könnte man auch durch eine **List** leicht realisieren, allerdings ist die Stärke einer Queue gerade, dass man sich darauf verlassen kann, dass nur das allernächste Element zur Verfügung steht.

- Die Elemente werden also in der exakt gleichen Reihenfolge entnommen als sie hinzugefügt wurden.
- Aus diesem Grund eignet sich eine Queue zum Beispiel dafür Elemente die der Reihe nach verarbeitet werden müssen, zu sammeln und dann einzeln in der exakt gleichen Reihenfolge zu verarbeiten.

Dieses Prinzip nennt man **FIFO** (First-In, First-Out).

29.9.1 Wichtige Properties und Methoden

Aktion	Methode/Property	Beispiel		
		Vorher	Code	Nachher
Erzeugung:	Konstruktor		Queue<int> q = new Queue<int>;	leer
Hinzufügen:	Enqueue(wert)		q.Enqueue(2); q.Enqueue(6); q.Enqueue(14);	[2] [6] [14]
Zugriff:	Peek()	[2] [6] [14]	int first = q.Peek();	[2] [6] [14] first = 2
Entfernen:	Dequeue()	[2] [6] [14]	int first = q.Dequeue();	[6] [14] first = 2
Länge:	Property Count	[2] [6] [14]	int count = q.Count;	count = 3
Löschen:	Clear()	[2] [6] [14]	q.Clear();	leer

29.9.2 Code Sample

C# Queue Sample

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

namespace QueueSample
{
    class Program
    {
        static void Main(string[] args)
        {
            Queue<string> persons = new Queue<string>();
            persons.Enqueue("Astrid"); // Hinzufügen zur Queue mit Enqueue
            persons.Enqueue("Bernhard");
            persons.Enqueue("Can");

            // Ausgabe aller Elemente
            Console.WriteLine(String.Join(',', persons));
            Console.WriteLine($"Es warten {persons.Count} Personen.");

            // Das vorderste Element anzeigen
            string first = persons.Peek(); // Gibt das vorderste Element zurück. Astrid!
            Console.WriteLine($"Die erste Person ist {first}.");
            Console.WriteLine($"Es warten {persons.Count} Personen.");

            string next = persons.Dequeue(); // Holt das erste Element aus der Queue.
            Console.WriteLine($"{next} ist jetzt dran.");
            Console.WriteLine($"Es warten nur noch {persons.Count} Personen.");
            Console.WriteLine(String.Join(',', persons));

            Console.WriteLine($"Die erste Person ist jetzt {persons.Peek()}.");
            Console.WriteLine(persons.Dequeue()); // -> Bernhard wird entfernt
            Console.WriteLine(String.Join(',', persons));

            Console.WriteLine($"Die erste Person ist jetzt {persons.Peek()}.");
            persons.Dequeue(); // -> Can wird entfernt

            if ( persons.Count > 0 )
                Console.WriteLine(persons.Dequeue());
            else
                Console.WriteLine("Es ist keine Person in der Warteschlange.");
        }
    }
}
```

Programmausgabe:

```
Astrid,Bernhard,Can
Es warten 3 Personen.
Die erste Person ist Astrid.
Es warten 3 Personen.
Astrid ist jetzt dran.
Es warten nur noch 2 Personen.
Bernhard,Can
Die erste Person ist jetzt Bernhard.
Bernhard
Can
Die erste Person ist jetzt Can.
Es ist keine Person in der Warteschlange.
```

30 Events und Delegates

Events dienen in Programmiersprachen dafür, dass eine einzelne Instanz, mehrere andere Instanzen über ein bestimmtes Ereignis informiert und dabei Information verschickt.

30.1 Observer Pattern

Eine Möglichkeit ist, das sogenannte Observer Pattern²⁴ (auch Listener Pattern) zu realisieren.

Man bezeichnet die Instanz, die die Information verschickt, als **Observable**.
Die informierten Instanzen werden als **Observers** bezeichnet.

Bei Events geht es also um die Lösung einer 1:n-Kommunikation. Ein Observable informiert beliebig viele Observers.

Eine Besonderheit dabei ist, dass der Observable nichts über die Natur der Observers weiß. Man sagt auch, dass Observables und Observers voneinander entkoppelt sind.

Eine häufige Anwendung von Events ist die GUI-Programmierung (GUI = Graphical User Interface), in dem UI-Controls, z. B. Buttons, Labels, TextBoxes, ..., bei Veränderungen ihres Zustands über Events die Event Handler-Methoden aufrufen und Informationen mitteilen.

30.2 Delegates

Ein Delegate in C# ermöglicht es Verweise auf Methoden zu speichern. Ein Delegate wird mit dem speziellen Schlüsselwort `delegate` deklariert.

Beispiel:

```
delegate int RechenOperation(int a, int b);
```

`RechenOperation` ist der Name des Delegates.

Die angegebenen Parameter und der Rückgabewert²⁵ legen fest, auf welche Methoden eine Variable mit dem Typ dieses Delegates Verweise speichern kann.

In diesem Fall sind es Methoden die zwei Ganzzahlen als Input Parameter haben und eine Ganzzahl als Rückgabewert zurückliefern. Grundsätzlich unterstützen Delegates auch `out` und `ref` bei Parametern.

Angenommen es existiert eine Methode die zwei Zahlen miteinander addiert

```
int Addieren(int a, int b)
{
    return a + b;
}
```

²⁴ https://en.wikipedia.org/wiki/Observer_pattern

²⁵ Die Gesamtheit aus Rückgabetyp und Parametertypen nennt man auch die Signatur der Methode.

Der Delegate **RechenOperation** hat die gleiche Signatur wie die Methode Addieren. Eine Variable die den Delegate als Typ hat, kann daher einen Verweis auf die Methode Addieren speichern.

Damit ein Verweis auf eine Methode Addieren gespeichert und über den Verweis die Methode aufgerufen werden kann, sind folgende Schritte notwendig:

1. Deklaration des Delegates: Statt dem Methodenamen wird ein Delegatename gewählt.
2. Es muss eine Variable für den Delegate angelegt werden
3. Dieser Variablen kann eine Methode zugewiesen werden. Diese Variable verweist dadurch auf diese Methode.
4. Die Methode die einem Delegate zugewiesen wurde, kann nun über diese Variable aufgerufen werden. Dazu wird statt dem Methodenamen einfach die Variable verwendet.

```
// 1. Deklaration des Delegates passend zu Addition
delegate int RechenOperation(int a, int b);

// 2. Anlegen einer Variablen
RechenOperation op;

// 3. Zuweisung einer Methode
op = Addieren; // new Rechenoperation( Addieren );

// 4. Aufruf der Methode durch die Delegatevariable
int x = 3, y = 5;
int result = op(x, y);
```

Dieses Beispiel zeigt grundsätzlich wie ein Delegate selbst definiert wird und wie man ihn für den Aufruf einer einzelnen Methode nützt.

30.2.1 Verkettung von Delegates (Multicast Delegates)

Ein Delegate in C# kann glücklicherweise nicht nur den Verweis auf eine einzelne Methode speichern, sondern auf mehrere. Das ist durch die Operatoren += und -= möglich.

Angenommen es existieren die beiden folgenden Methoden, die beide dieselbe Signatur haben, d. h. gleiche Input Parameter und gleichen Rückgabetyp (in diesem Fall void).

```
static void Potenzieren(double a, double b)
{
    Console.WriteLine(a + " ^ " + b + " = " + Math.Pow(a, b));
}

static void ArithmetischesMittel(double x, double y)
{
    Console.WriteLine("Mittelwert = " + (x + y) / 2);
}
```

Mit dem Wissen, dass ein Delegate den Verweis auf eine solche Methode speichern kann, definieren wir einen Delegate, legen eine Variable an und weisen dieser zuerst die Methode Potenzieren zu.

```
delegate void Berechnung(double a, double b);

Berechnung op;
op = Potenzieren;

double v1 = 2, v2 = 3;
op(v1, v2);
```

Beim Start dieses Programms werden die beiden Werte 2 und 3 an die Delegatevariable op übergeben. Diese speichert den Verweis auf die Methode Potenzieren, daher werden die beiden Werte 2 und 3 an diese Methode übergeben. Aufgrund des `Console.WriteLine` in der Methode Potenzieren gibt das Programm folgendes aus:

Ausgabe:

2³ = 8

Das ist soweit nichts Neues und entspricht dem vorhergehenden Beispiel.

Man kann nun aber zusätzlich in der Delegatevariablen op eine weitere Methode abspeichern, einfach in dem man sie mit += hinzufügt

```
op += new Berechnung(ArithmetischesMittel);
```

Nach dem Einfügen dieser Zeile in das obere Programm,

```
delegate void Berechnung(double a, double b);

Berechnung op;
op = Potenzieren;
// Fügt den Verweis auf die Methode ArithmetischesMittel hinzu
op += ArithmetischesMittel;

double v1 = 2, v2 = 3;
op(v1, v2);
```

werden durch `op(v1, v2)` beide Methoden hintereinander aufgerufen. op speichert die Verweise auf beide Methoden.

Ausgabe:

2³ = 8
Mittelwert = 2,5

Mit dem Operator -= kann eine bereits hinzugefügte Methode aus einem Delegate wieder entfernt werden:

```
delegate void Berechnung(double a, double b);

Berechnung op;
op = Potenzieren;
op -= ArithmetischesMittel;
```

```

double v1 = 2, v2 = 3;
op(v1, v2);
// Entfernt den Verweis auf die Methode Potenzieren aus dem Delegate
op -= Potenzieren;
op(5, 7);

```

Ausgabe:

```

2^3 = 8
Mittelwert = 2,5
Mittelwert = 6

```

op(5, 7) ruft die Methode ArithmetischesMittel mit den Werten 5 und 7 auf.

Siehe das Verzeichnis *Delegates - Mehrere Methoden in einem Delegate* am Shared für ein weiteres Beispiel.

30.2.2 Func<> und Action<>: Delegates im .NET-Framework

Das .NET-Framework hat einige Delegates vordefiniert die man bei Bedarf einfach verwenden kann. Sehr gebräuchlich sind Action und Func. Beide dienen dazu, Verweise auf Methoden zu speichern, die bis zu 16 Input Parameter haben können.

- Ein Action-Delegate kann nur Verweise auf void-Methoden speichern, also solche Methoden die keinen Rückgabewert haben.
- Ein Func-Delegate dient zum Speichern von Methoden die einen Rückgabewert haben, z. B. double, string oder eine Klasse.

Vordefinierte Delegates können nur verwendet werden, denn die Methoden keinen out oder ref Parameter verwenden!

30.2.2.1 Beispiel für Func

Beim ersten Beispiel wurde ein Delegate definiert, der Methoden speichern kann, die zwei int-Input Parameter und einen int-Rückgabewert erlauben. Das gleiche lässt sich durch den folgenden Delegate Func<int, int, int> erreichen, der im .NET-Framework definiert ist.

In den Spitzklammern werden die Typen der Input Parameter und des Rückgabewerts angegeben. Der Rückgabetyp steht in der Spitzklammer immer ganz zuletzt.
Die beiden unteren Codestücke sind völlig äquivalent.

<i>Selbstdefinierter Delegate</i>	<i>.NET-Delegate Func</i>
<pre> delegate int RechenOperation(int a, int b); RechenOperation op; op = new Rechenoperation(Addieren); int x = 3, y = 5; int result = op(x, y); </pre>	<pre> Func<int, int, int> op; op = new Func<int, int, int>(Addieren); int x = 3, y = 5; int result = op.Invoke(x, y); </pre>

30.2.2.2 Beispiel für Action

Action wird immer dann verwendet, wenn die Methode keinen Rückgabewert hat, also als void definiert ist.

Selbstdefinierter Delegate	.NET-Delegate Func
<pre>delegate void Berechnung(double a, double b); Berechnung op; op = Potenzieren; int x = 2, y = 3; int result = op(x, y);</pre>	<pre>Action<double, double> op; op = Potenzieren; int x = 2, y = 3; op.Invoke(x, y);</pre>

Die folgende Tabelle enthält jeweils eine Methode, den selbstdefinierten Delegate und die vordefinierten Delegates des .NET-Frameworks:

Methode	Selbstdefiniert delegate double Op(int a, int b)
<pre>double Quotient(int a, int b) { return (double)a / b; }</pre>	Code: <pre>delegate double Op(int a, int b); Op v; v = Quotient; int result = v(3, 5);</pre> .NET-Delegate Func<int, int, double> Code: <pre>Func<int, int, double> v; v = Quotient; int result = v.Invoke(3, 5);</pre>
Methode	Selbstdefiniert delegate string Op(string a, int b)
<pre>string Cut(string s, int anz) { return s.Substring(0,anz); }</pre>	Code: <pre>delegate string Op(string s, int b); Op v; v = Cut; string result = v("Laufsport",4);</pre> .NET-Delegate Func<string, int, string> Code: <pre>Func<string, int, string> v; v = Cut; string result = v.Invoke("Laufsport",4);</pre>
Methode	Selbstdefiniert delegate void Op(double a)
	Code: <pre>delegate void Op(double x); Op v; v = KubikAusgeben; v(2);</pre>

<pre>void KubikAusgeben(double w) { Console.WriteLine(w*w*w); }</pre>	.NET-Delegate Action<double> Code: <pre>Action<double> v; v = KubikAusgeben; v.Invoke(2);</pre>
---	---

30.2.3 Übungen zu Delegates

Aufgabe 1:

Welcher der angegebenen Typen (Func, Action) entspricht dieser Methodensignatur?
Person und Car sind Klassen.

```
Car FindCar( Person p, bool b )
    □ Action<Person, bool, Car>
    □ Func<Car, Person, bool>
    □ Func<Person, bool, Car>
    □ Action<Car, Person, bool>
```

Aufgabe 2:

Welchen Typ (Func, Action) entsprechen folgende Methoden? Schreib den Typ in die rechte Spalte.
Person und Car sind Klassen.

Methode	Typ
bool IsGreater(int a, int b)	Func<int, int, bool>
Person Find(string name)	
void Drives(Car c, Person p)	
bool Contains(List<Person> arr, Person p)	
void RaiseEvent(object sender, EventArgs a)	
Car CreateCar()	
void StartEngine(Car c)	
void Exit()	

Aufgabe 3:

Erstelle einen Methodenkopf für die Typen? Der Methodenname ist beliebig.
Person und Car sind Klassen.

Typ	Methode
Func<int,int,bool>	bool XYZ(int a, int b)
Func<Car>	
Action<Car>	
Func< List<Person> >	
Action< List<Person> >	
Func<double, int, string>	
Action<double, int, Car>	
Action<object, EventArgs>	

30.3 Events in C#

Events und Delegates sind in C# oftmalig verwendete Konstrukte, die aus dieser Sprache nicht mehr wegzudenken sind. Events gleichermaßen, können Referenzen auf ihre Event Handler-Methoden speichern. Beim Invoken eines Events, werden Informationen (EventArgs), an alle registrierten Event Handler-Methoden geschickt.

C# kennt die Schlüsselwörter `delegate` und `event`. Die beiden Schlüsselwörter dienen dazu, dass in C# für Eventkommunikation nicht das vollständige Observer Pattern implementiert werden muss.

30.4 Beispiel zur Eventprogrammierung: Aktienkurs

Das Beispiel ist bewusst sehr einfach gehalten um die Codeteile rund um die Auslösung eines Events möglichst gut herauszuarbeiten.

Es gibt eine Klasse Aktie, bei der in einer Methode der Kurs der Aktie mit einem Zufallsgenerator neu berechnet werden kann.

Mit einer 30%igen Wahrscheinlichkeit fällt der Kurs, zu 70 % steigt er. Immer wenn der Kurs fällt, wird in der Aktie ein Event ausgelöst und diese Tatsache, dass der Kurs gefallen ist, an die Instanz einer andere Klasse über ein Event gemeldet. Dabei wird der Name der Aktie und der letzte Kurswert der Aktie übermittelt.

Im Beispiel wird nur eine einzige Instanz einer Aktie erzeugt und auch nur eine einzige Instanz wird von dem Fallen des Kurses informiert. Im Allgemeinen werden viel Instanzen beim Auslösen eines Events informiert.

In einer Schleife im Main wird die Neuberechnung des Aktienkurses alle 0,5 Sekunden angestoßen.

Das folgende Diagramm illustriert die Eventauslösung.

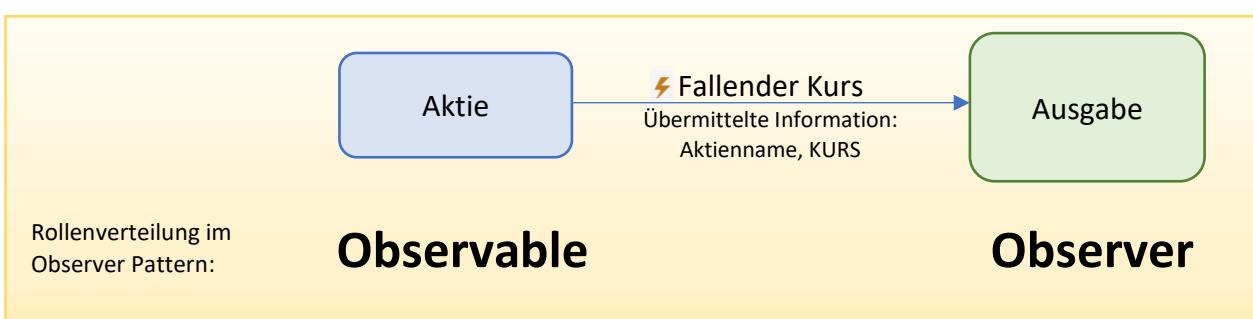


Abbildung 43: Schema des Design Patterns Observer (Listener)

30.4.1 Kochrezept für das Anlegen eines Events

Alle Schritte, die man durchführen muss, um ein Event zu definieren sind folgendermaßen:

Schritt	C#
1. Definition des Events	Definiere im Observable ein Event mit dem Schlüsselwort <code>event</code> und lege einen passenden Delegate für die Informationsübertragung fest. In unserem Beispiel wird dafür der Delegate <code>EventHandler</code> und eine eigene <code>EventArgs</code> -Klasse verwendet. <code>public event EventHandler<KursEventArgs> KursFällt;</code>
Implementierung des EventHandlers im Observer:	Im Observer ist die Eventhandler-Methode mit derselben Signatur des Delegates des Events zu implementieren. <code>void AusgabeWennKursFällt(object sender, KursEventArgs args)</code>
2. Auslösen des Events	Im Observable wird an geeigneter Stelle das Event ausgelöst (<code>invoked</code>). <code>KursFällt?.Invoke(this, new KursEventArgs(Name, Kurs));</code>
3. Registrierung des Events	An geeigneter Stelle fügt der Observer seine EventHandler-Methode dem Delegate des Observables hinzu . <code>// display = Observer, kb = Observable kb.KursFällt += display.AusgabeWennKursFällt;</code>

```
using System;
using System.Diagnostics;
```

30.4.2 Code Sample in C#

```
namespace Events
{
    class Program
    {
        static void Main(string[] args)
        {
            Ausgabe display = new Ausgabe();

            Aktie kb = new Aktie("Code Productions", 110);

            kb.KursFällt += display.AusgabewennKursFällt;

            Stopwatch watch = new Stopwatch();
            watch.Start();

            do
            {
                System.Threading.Thread.Sleep(500); // 0.5s
                kb.BerechneKursNeu();

                display.AusgabeDesKurswertes(kb.Name, kb.Kurs);
            } while (watch.ElapsedMilliseconds < 10000 /* 10 s */);

            kb.KursFällt -= display.AusgabewennKursFällt;

            Console.ReadKey();
        }
    }
}
```

Registrierung des Events

Entregistrierung

```
class Ausgabe // Observer
{
    // Diese Methode gibt einfach den Wert einer Aktie aus.
    public void AusgabeDesKurswertes(string aktienname, int kurswert)
    {
        Console.WriteLine($"{aktienname}: ${kurswert}");
    }

    // Diese Methode soll aufgerufen werden, wenn der Kurs fällt.
    public void AusgabewennKursFällt(object sender, KursEventArgs args)
    {
        Console.WriteLine($"Achtung! Der Kurs von {args.Aktienname} ist bei ${args.Kurs} gefallen.");
    }
}
```

```
namespace Events
{
    class Aktie // Observable
    {
        static Random rnd = new Random();

        public Aktie(string name, int startkurs)
        {
            Name = name;
            Kurs = Kurs;
        }

        public string Name { get; private set; }
        public int Kurs { get; private set; }

        // Definition des Events: Festlegung, was das Event mitteilt,
        // wenn sich jemand für fallende Kurse interessiert.
        public event EventHandler<KursEventArgs> KursFällt;

        public void BerechneKursNeu()
        {
            // Der Kurs fällt mit einer Wahrscheinlichkeit von 30%.
            if (rnd.Next(0, 100) < 30)
            {
                // Hier werden diejenigen die sich für
                // fallende Kurse interessieren, tatsächlich informiert.
                // Auslösen des Events
                KursFällt?.Invoke(this, new KursEventArgs(Name, Kurs));
            }
            Kurs--;
            if (Kurs < 0)
                Kurs = 0;
        }
        else
        {
            // zu 70% steigt der Kurs
            Kurs++;
        }
    }
}

class KursEventArgs : EventArgs
{
    public int Kurs { get; }
    public string Aktienname { get; }

    public KursEventArgs(string aktienname, int kurs)
    {
        Aktienname = aktienname;
        Kurs = kurs;
    }
}
```

30.5 Übungen zu Events

Beispiel: Im 22. Bezirk gibt es drei Schulen, die an das Alarmsystem der Feuerwehr angeschlossen sind.

- HTL-Donaustadt, Deinleinstraße 45
- Bernoulligymnasium, Bernoulligasse 3
- Hertha Firnberg Schule, Firnbergstraße 1

Bei jedem Feueralarm in einer der Schulen, wird die Feuerwehr informiert.

Bei der Alarmsmeldung wird die Schule, der Zeitpunkt, das Stockwerk in dem der Brandmelder angeschlagen hat und die Dringlichkeitsstufe (Vollbrand, Schwellbrand, Unbekannte Stufe) übermittelt.

Es gibt eine Klasse **School** und eine Klasse **FireDepartment**.

Wieviele Instanzen der Klasse Schule gibt es? _____

Wieviele Instanzen der Klasse Feuerwehr gibt es? _____

Wie definiert man das Enum für die Dringlichkeitsstufe? _____

Aufgabe 1: Definiere einen Delegate für die Alarmsmeldung. Verwende Action oder Func:

Definiere ein Event **FireAlarm** mit diesem Delegate.

Aufgabe 2: Leite eine Klasse von **EventArgs** ab, in der man den Zeitpunkt und die Dringlichkeitsstufe speichern kann.

Definiere nun ein Event **FireAlarm** mit dem Delegate **EventHandler<...>**.

Aufgabe 3: In welcher der beiden Klassen muss das Event definiert werden?

Definiere das Event: _____

Schreibe, wie der Eventhandler in der anderen Klasse aussehen muss:

```
public void _____  
{  
    _____  
}
```

31 Asynchrone Programmierung

31.1 Einleitung

To be completed.

31.2 Klasse Task

To be completed.

31.3 `async` und `await`

Methoden, die ein Ergebnis berechnen bzw. auf einen Input von einem anderen Prozess warten, sollten möglichst non-blocking aufgerufen werden.

Blocking (synchroner Aufruf)

Der Aufrufer einer Methode wartet auf das Ergebnis der aufgerufenen Methode.

Nonblocking (asynchroner Aufruf)

Der Aufrufer führt während er ein Ergebnis erwartet, einen anderen Programmcode aus.

31.3.1 Beispiel: Empfang von Daten einer UDP-Kommunikation

```
private static void StartListener()
{
    UdpClient listener = new UdpClient(11000);
    IPEndPoint senderEP = new IPEndPoint(IPAddress.Any, 11000);
    ...
    Console.WriteLine("Waiting for message...");
    byte[] bytes = listener.Receive(ref senderEP); // wird synchron aufgerufen
    ...
}

public static void ComputeSomethingCompletelyDifferent ()
{   // eine unwichtige Berechnung die mit Udp nichts zu tun haben
    ...
}

public static void Main()
{
    StartListener();
    ComputeSomethingCompletelyDifferent(); // wird erst ausgeführt,
                                            // wenn die Daten geschickt wurden
}
```

31.3.2 Änderung eines synchronen in einen asynchronen Aufruf

① Statt dem synchronen Aufruf `Receive`, wird die asynchrone Methode `ReceiveAsync` verwendet.

alt: `byte[] bytes = listener.Receive(ref groupEP); // synchroner Aufruf`

neu: `Task<UdpReceiveResult> result = listener.ReceiveAsync(); // asynchroner Aufruf`

Mit dem Schlüsselwort `await` kann man diesen Aufruf lesbarer machen.

alt: `Task<UdpReceiveResult> result = listener.ReceiveAsync();`

neu: `UdpReceiveResult result = await listener.ReceiveAsync();`

② In `result` findet man die beiden Rückgabewerte des synchronen Calls.

neu: `byte[] bytes = result.Buffer;
 senderEP = result.RemoteEndPoint;`

③ IMMER wenn eine Methode das Schlüsselwort `await` enthält, muss die Methode selbst asynchron gemacht werden, d. h. `async` hinzufügen und an den Methodennamen **Async** anhängen.

```
private static void StartListener()
private static async void StartListenerAsync()
```

```
public static void Main()
{
    StartListenerAsync();
    ComputeSomethingCompletelyDifferent(); // wird ausgeführt wenn die Daten geschickt wurden
}
```

Jetzt wird ComputeSomethingCompletelyDifferent() ausgeführt,
während in ReceiveAsync auf die Daten gewartet wird!

32 Lambdas

Ein Lambda ist etwas in C#, das eine Methode ersetzt. Ein Lambda erkennt man (meist²⁶) daran, dass => vorkommt.

Lambda Expressions dienen dazu um anonyme Methoden erstellen zu können. Sie sind quasi eine Weiterentwicklung von anonymen Methoden.

Lambdas werden vor allem für die Spracherweiterung LINQ verwendet, sowie für die asynchrone Programmierung.

Lambda Expressions nimmt Bezug auf das sogenannte Lambda Kalkül, einen Ausdruck aus der theoretischen Informatik.

Beispiel: Gegeben ist die Methode

```
static double CalcHypo(double a, double b)
{
    return Math.Sqrt(a * a + b * b);
}
```

Diese Methode kann durch dieses Lambda ersetzt werden:

```
(a, b) => Math.Sqrt(a * a + b * b);
```

32.1 Fakten über Lambdas

In diesem Abschnitt sind einige Dinge über Lambdas angegeben.

1. Ein Lambda hat keinen Namen.
2. Die Inputparameter haben meist keinen Datentyp. (Sie können aber auch einen haben.)

```
(double a, double b) => { return Math.Sqrt(a * a + b * b); }
```

oder

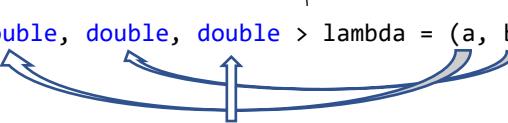
```
(a, b) => { return Math.Sqrt(a * a + b * b); }
```

3. Wenn auf der rechten Seite nur ein Ausdruck berechnet wird, kann man { return ; } auch weglassen.

```
(a, b) => Math.Sqrt(a * a + b * b)
```

4. Ein Lambda kann in einer Variablen "abgespeichert" werden. Genaugenommen wird ein Verweis gespeichert.

Datentyp des Lambdas



```
Func<double, double, double> lambda = (a, b) => Math.Sqrt(a * a + b * b);
```

Rückgabetyp (Ergebnistyp)

²⁶ Auch Bodied Member Expressions verwenden =>, haben mit Lambdas aber rein gar nichts zu tun.

5. Lambdas können aufgerufen (=invoked) werden:

```
Console.WriteLine(lambda(3,5));
```

oder

```
Console.WriteLine(lambda.Invoke(3,5));
```

oder (mit Überprüfung auf null)

```
Console.WriteLine(lambda?.Invoke(3,5));
```

Falls ein Lambda keinen Rückgabetyp hat, kann sein Datentyp mit Action gebildet werden.

```
Action< double, double > lambda2 = (a, b) => Console.WriteLine(a * a + b * b);
```

6. Lambdas die einen Wert berechnen, heißen **Expression-Lambdas**.

7. Lambdas, die keinen Wert berechnen, also nur Anweisungen ausführen haben, heißen **Statement-Lambdas**.

33 Programmbibliotheken

Bibliotheken (engl. Libraries) dienen dazu, Klassen in kompilierter Form zur Laufzeit zur Verfügung zu stellen, so dass andere Methoden diese verwenden können.

Sie erleichtern die Arbeit von Entwicklerinnen und Entwicklern, weil sie bestimmte Funktionalitäten zur Verfügung stellen, die häufig verwendet werden.

Wenn man beispielsweise Daten im JSON-Format schreiben oder lesen möchte, kann man JSON-Libraries verwenden. In C# ist Newtonsoft eine populäre JSON Library

<https://www.nuget.org/packages/Newtonsoft.Json>

Bibliotheken werden beim Entwickeln eines Programms über die Entwicklungsumgebung eingebunden.

33.1 Library Files

Library Files haben in C# unterschiedliche File Extensions.

	C#
<i>Source Files</i>	.cs
<i>Compiled Sources</i>	.obj (Assembly Files)
<i>Executables</i>	.exe
<i>Start eines Programms auf der Commandline</i>	<name>.exe
<i>Dynamische Bibliotheken</i>	.dll
<i>Statische Bibliotheken</i>	.lib
<i>Format von Executables und Bibliotheken</i>	IL (Intermediate Language)

Die Fileendungen sind auch vom Betriebssystem abhängig. In Linux und Windows sind diese unterschiedlich. Auch die Ausdrücke sind betriebssystemabhängig.

In Linux spricht man z. B. von shared libraries.

Bibliotheken dienen im Wesentlichen zur Modularisierung von Funktionalitäten, die in einem Projekt oder in einem Unternehmen entwickelt werden.

33.2 Dynamische und statische Bibliotheken

Grundsätzlich unterscheidet man zwei Arten wie Funktionalität an ein ausführbares Programm gebunden werden kann.

Statische Bibliotheken	Dynamische Bibliotheken
Das Bibliotheksfile wird beim Builden des ausführbaren Programms fest (=statisch) mit dem .exe-File verbunden.	Das Bibliotheksfile ist mit der Applikation nicht fix verbunden. Bei der Ausführung müssen sich die Bibliothekfiles an genau definierten Stellen des Filesystems befinden.

Statische Bibliotheken	Dynamische Bibliotheken
Vorteil: Die Bibliothek ist dadurch fix mit dem .exe verbunden. Es kann nicht sein, dass bei der Ausführung des Programms ein File nicht gefunden wird.	Vorteil: Das Executable ist viel kleiner, weil die Funktionalität der Libraries in anderen Files ausgelagert ist.
Nachteil: Das .exe-File wird dadurch sehr groß.	Nachteil: Bei der Ausführung einer Anwendung ist darauf zu achten, dass alle dynamischen Libraries im Filesystem „an der richtigen Stelle“ zur Verfügung stehen. Java: Die dynamischen Libraries .jar müssen sich im Classpath befinden. Die dynamischen Libraries (.DLLs) sollten sich z. B. am selben Verzeichnis wie das .exe befinden.
Fileextension	
Statische Libraries haben in C# die Fileextension .lib	Dynamische Libraries haben die Endung .dll.

In Softwareprojekten ist es essentiell, dass genau definiert ist, wo sich Library Files beim Entwickeln und beim Ausführen einer Applikation, die Libraries verwenden, befinden.

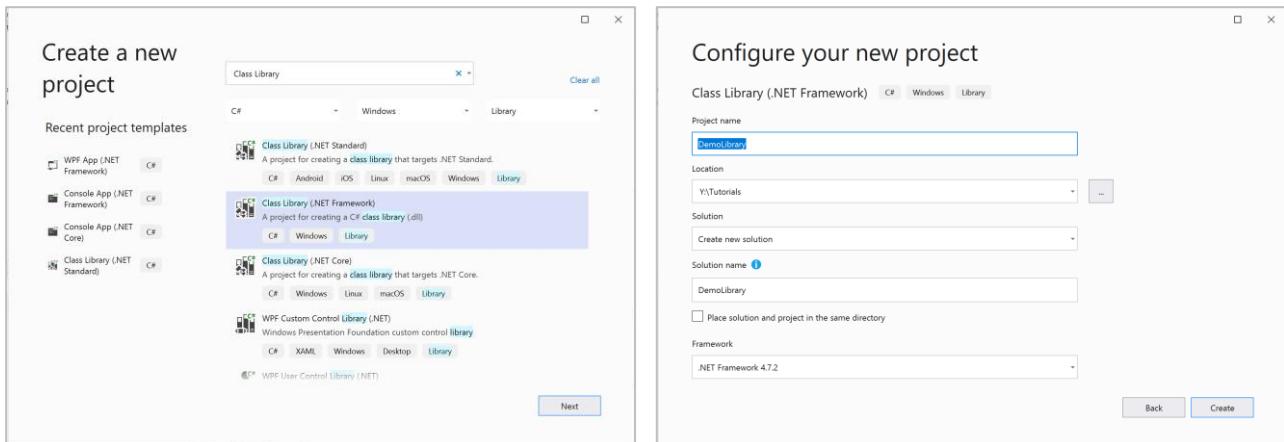
Die korrekte Verteilung von Libraries bei der Installation eines Programms muss in der Praxis genau getestet werden.

In der Praxis ist außerdem die korrekte Versionierung und Release-Planung von Libraries wichtig.

33.3 C# - Erstellen einer Dynamic Link Library (DLL)

Dieser Abschnitt zeigt in Screenshots, wie eine DLL für C# in Visual Studio erstellt wird.

1. Anlegen eines Library Projekts: Class Library (.NET Framework) oder Class Library (.NET Core)

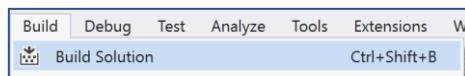


2. Die Klassen der Library implementieren.

The screenshot shows the Microsoft Visual Studio interface. In the Solution Explorer, there is one project named 'DemoLibrary' containing a file 'LibraryClass.cs'. In the code editor, the content of 'LibraryClass.cs' is displayed:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace DemoLibrary
8  {
9      public class LibraryClass
10     {
11         private int n;
12         public void SetN(int n) { this.n = n; }
13
14         public int SumUpToValue(int m)
15         {
16             return SumUpToValue(n, m);
17         }
18
19         public static int SumUpToValue(int from, int to)
20         {
21             int sum = 0;
22
23             for (int i = from; i <= to; i++)
24                 sum += i;
25
26             return sum;
27         }
28     }
29 }
```

3. Am Schluss kann die Library erstellt werden.

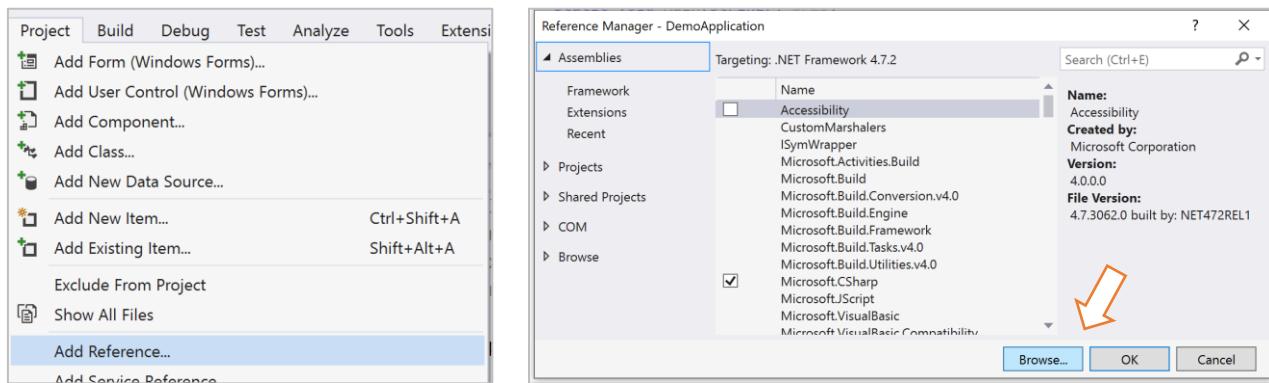


Die Library wird in bin\Debug bzw. bin\Release abgespeichert.



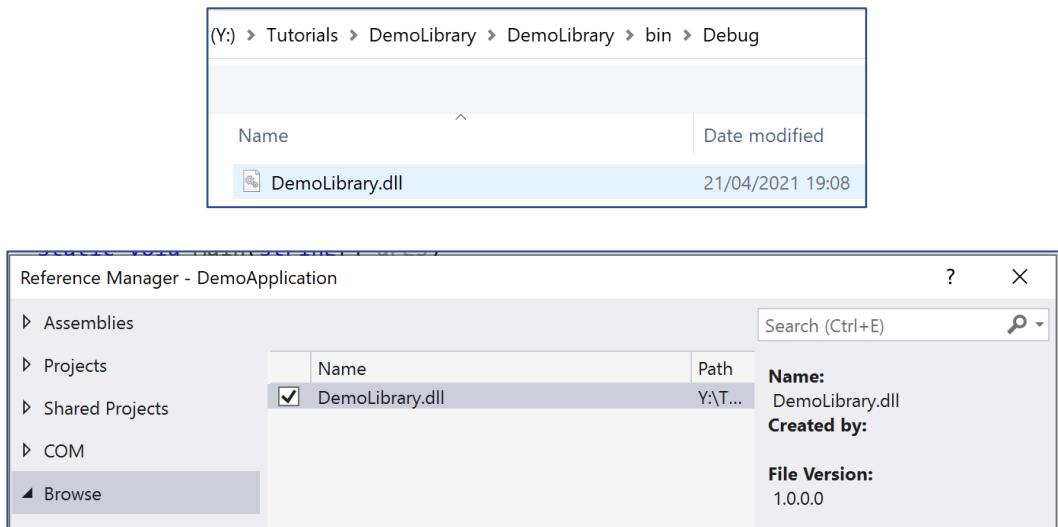
33.4 C# - Verwenden einer Dynamic Link Library (DLL)

Um eine Library in C# in eine Applikation einzubinden, muss man den Reference Manager starten und nach dem dll-File browsen. Das ist über das Projekt-Menü möglich.

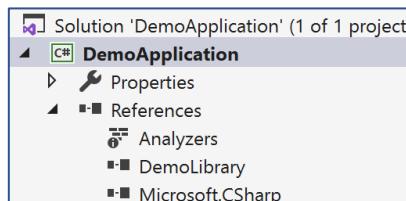


Dann ist das Library File auszuwählen.

Das erzeugte DLL-File im File Manager (Windows Explorer in Windows):



Nach dem Verlassen des Reference Manager-Dialogs erscheint die DLL in den References des Projekts.



Nun kann der Namespace der Library mit einem using-Statement verfügbar gemacht werden.

```
Program.cs  X
DemoApplication
  1  using System;
  2  using DemoLibrary;
  3
  4  namespace DemoApplication
  5  {
  6      class Program
  7      {
  8          static void Main(string[] args)
  9          {
 10              LibraryClass obj = new LibraryClass();
 11              obj.SetN(5);
 12              Console.WriteLine(obj.SumUpToValue(10));
 13              Console.WriteLine(LibraryClass.SumUpToValue(5,10));
 14          }
 15      }
 16  }
```

34 Sprachausgabe in C#

Du hast Lust deinen Computer sprechen zu lassen? Abgesehen von einem Lautsprecher auf deinem Rechner benötigst du in C# angenehmerweise nur sehr wenig Programmcode dazu.

Die Sprachausgabe in einer bestimmten Sprache funktioniert in C# nur dann, wenn das entsprechende Sprachpaket am Computer überhaupt installiert ist.

34.1 Welche Sprachen sind unter Windows installiert?

Auf einem deutschsprachigen Windows sind üblicherweise die Sprachen Deutsch und Englisch verfügbar.

Mit folgendem Programmcode kann man schauen, welche Sprache dein Computer überhaupt sprechen kann, d. h. welche Sprachen sind in C# überhaupt verfügbar.

C# Ausgabe aller installierten Audiosprachen unter Windows (.NET Framework)

```
using System.Speech.Synthesis;
...
static void Main(string[] args)
{
    SpeechSynthesizer synth = new SpeechSynthesizer();

    Console.WriteLine("Installierte Stimmen:");

    var voices = synth.GetInstalledVoices();
    foreach (var voice in voices)
    {
        var vi = voice.VoiceInfo;
        Console.WriteLine(
            $"Name: {vi.Name}\nBeschreibung: {vi.Description} ({vi.Gender}, {vi.Age})");
    }
}
```

Programmausgabe am Windows-Notebooks des Autors:

Installierte Stimmen:

```
Name: Microsoft Hedda Desktop
Beschreibung: Microsoft Hedda Desktop - German (Female, Adult)
Name: Microsoft Hazel Desktop
Beschreibung: Microsoft Hazel Desktop - English (Great Britain) (Female, Adult)
Name: Microsoft Zira Desktop
Beschreibung: Microsoft Zira Desktop - English (United States) (Female, Adult)
Name: Microsoft Helena Desktop
Beschreibung: Microsoft Helena Desktop - Spanish (Spain) (Female, Adult)
```

Beachte, der Namespace

```
using System.Speech.Synthesis;
```

ist nur dann verfügbar, wenn vorher über den Menüpunkt

[Projekt] -> [Verweis hinzufügen] das Assembly `System.Speech` hinzugefügt wurde.

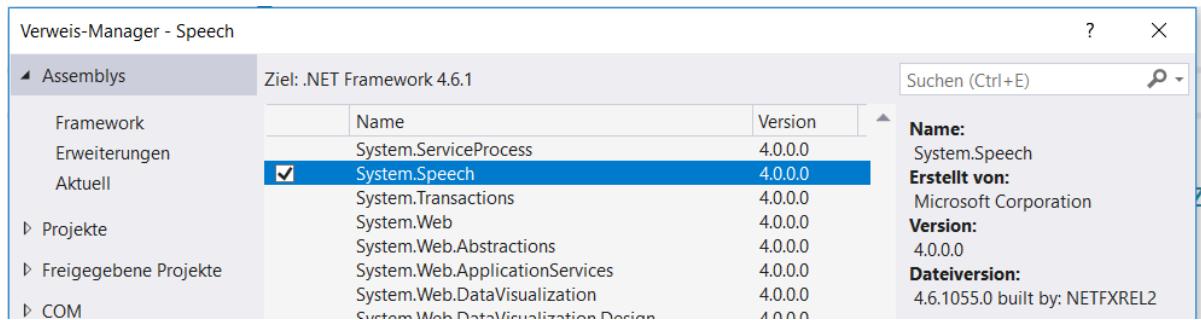


Abbildung 44: Hinzufügen des Assemblies System.Speech zu einer Solution

34.2 Ausgabe eines Textes in einer bestimmten Sprache

Mit folgendem Code kann man die Sätze "Hallo! Wie geht es Dir?" und "Danke! Gar nicht übel." auf Deutsch sprechen lassen:

C# Sprachausgabe auf Deutsch

```
SpeechSynthesizer synth = new SpeechSynthesizer();
synth.SelectVoiceByHints( VoiceGender.NotSet, VoiceAge.NotSet, 0,
                           new System.Globalization.CultureInfo("de") );
synth.Speak("Hallo! Wie geht es Dir?");
synth.Speak("Danke! Gar nicht übel.");
```

SelectVoiceByHints muss nur einmal am Beginn ausgeführt werden und die CultureInfo der jeweiligen Sprache muss übergeben werden.

Für Englisch muss statt "de" einfach "en" übergeben werden. Das ist ein Sprachkürzel.

C# Sprachausgabe auf Englisch

```
SpeechSynthesizer synth = new SpeechSynthesizer();
synth.SelectVoiceByHints( VoiceGender.NotSet, VoiceAge.NotSet, 0,
                           new System.Globalization.CultureInfo("en") );
synth.Speak("Hello! Nice to see you!");
synth.Speak(Console.ReadLine());
```

Das Sprachkürzel, also en für Englisch und de für Deutsch, sind in der internationalen Norm ISO 639-1 festgelegt, https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes

Beispiele für andere Sprachen und weiter Kürzeln:

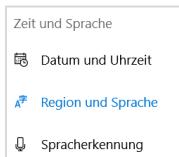
Sprachen	Kürzel
Kroatisch	hr
Rumänisch	rm
Serbisch	sr
Slowakisch	sk
Spanisch	es

34.3 Nachinstallieren einer Sprache unter Windows

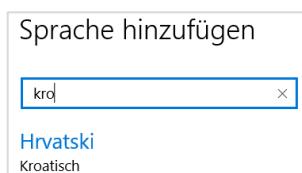
Sprachen können unter Windows auch nachinstalliert werden.

Beispiel: Installation der Sprache Kroatisch

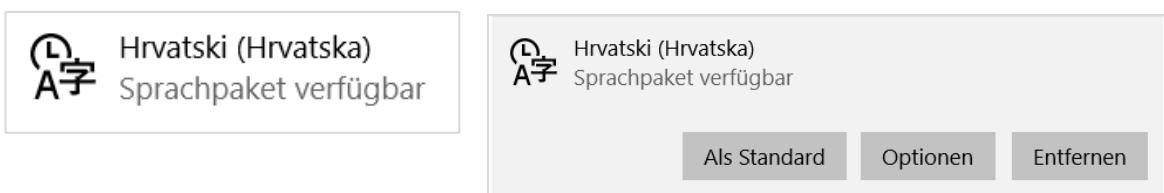
1. [Windows]+I zum Öffnen der Windows-Einstellungen
2. Zeit und Sprache auswählen



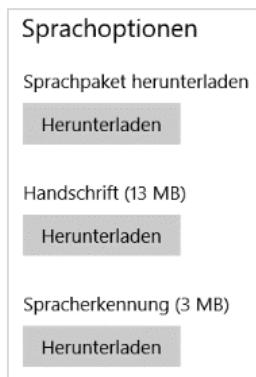
3. Region und Sprache auswählen:



4. Auf Sprache Hinzufügen klicken und die entsprechende Sprache auswählen und anklicken:
5. Das Sprachpaket auswählen und auf Optionen klicken:



6. Auf [Optionen] klicken und dann [Sprachpaket herunterladen] und [Spracherkennung] anklicken.



7. Dann können die Pakete heruntergeladen werden.

35 Anhang

35.1 Abbildungsverzeichnis

Abbildung 1: Übersicht über die Datentypen in C#	10
Abbildung 2: Wertebereich aller Ganzzahltypen.....	11
Abbildung 3: Wertebereich aller Fließkommatypen	11
Abbildung 4: Wertebereich des Character und Boolschen Datentyps.....	11
Abbildung 5: Create-Dialog in Visual 2019	13
Abbildung 6: Offenhalten des Konsolenfensters bei Visual Studio 2019	14
Abbildung 7: Option zum automatischen Schliessens des Debug-Fensters.....	14
Abbildung 8: Übersicht über Completetime und Runtime bei .NET	15
Abbildung 9: Beispiel für die IL (Intermediate Language)	15
Abbildung 10: Die wichtigsten String-Methoden in C#	45
Abbildung 11: Ein 3x4-Array in C#. Jedes Element hat einen Zeilenindex und einen Spaltenindex.	54
Abbildung 12: Start des CommandLineParameters Demoprogramms auf der Windowskonsole.....	64
Abbildung 13: Eingabe von Commandline Parametern in den Projekt Properties in Visual Studio ...	65
Abbildung 14: Start eines Programms in der Debug-Konfiguration	65
Abbildung 15: Der Entscheidungsbaum der Geschichte von Roman der ins Kino möchte.....	76
Abbildung 16: Alle Eigenschaften der drei Produkte eines Computershops.....	89
Abbildung 17: Übersicht über die drei Arten der Access Modifier in C#.....	93
Abbildung 18: Beispiele aller vier Arten einen Zugriffsmodifizierer anzugeben	93
Abbildung 19: Alle Eigenschaften der drei Produkte eines Computershops.....	94
Abbildung 20: Klassenhierarchie der Produkte des Webshops.....	96
Abbildung 21: Übersicht über die drei Zugriffsmodifizierer im Beispiel	96
Abbildung 22: Artikel-Basisklasse und abgeleitete Notebook-Klasse	97
Abbildung 23: Virtuelle Methode in der Artikel-Basisklasse	99
Abbildung 24: Essen auf einem Tisch und Kühlschrank als Analogie zu einem Stack und Heap	142
Abbildung 25: Merkregel für Value Types und Reference Types	143
Abbildung 26: Snapshot des Speichers während einer Programmausführung.....	143
Abbildung 27: Werttypen (Value Types) in C#.....	144
Abbildung 28: Lokale int-, double- oder char-Variablen werden am Stack abgespeichert.....	144
Abbildung 29: int-Variablen am Stack beim Aufruf einer Methode	145
Abbildung 30: Werttypen (Value Types) in C#.....	145
Abbildung 31: int[]-Variable am Heap beim Aufruf einer Methode.....	146
Abbildung 32: ref int Parameter bei Methodenaufrufen	147
Abbildung 33: ref int[]-Variable bei Methodenaufrufen	148
Abbildung 34: Array-Operationen beim Löschen und Einfügen von Elementen	163
Abbildung 35: Knotenliste einer LinkedList	164
Abbildung 36: Knotenliste einer LinkedList beim Löschen eines Knotens	164
Abbildung 37: Knotenliste einer LinkedList beim Löschen des ersten Knotens	164
Abbildung 38: Knotenliste einer LinkedList beim Löschen des letzten Knotens	165
Abbildung 39: Knotenliste einer LinkedList beim Einfügen eines Knotens	165
Abbildung 40: Knotenliste einer LinkedList beim Einfügen am Beginn der Liste	165
Abbildung 41: Knotenliste einer LinkedList beim Einfügen am Ende der Liste	165
Abbildung 42: KeyValueCollection eines Dictionaries	173

Abbildung 43: Schema des Design Patterns Observer (Listener)	188
Abbildung 44: Hinzufügen des Assemblies System.Speech zu einer Solution	202

35.2 Videoverzeichnis

Video 1: Youtube-Video mit Anders Hejlsberg über die Ursprünge von C#.....	9
Video 2: Coding Kurzgeschichten-Video über Reference Types und Value Types in C#	10
Video 3: Youtube Video über DRY in Gaming Software	17
Video 4: Coding Kurzgeschichten-Video „Gemma in den Supermarkt“	89
Video 5: Coding Kurzgeschichten-Videos des Beispiels "Pizza, Kebap und Getränke"	102
Video 6: Coding Kurzgeschichten-Video des Beispiels "Gemma in den Supermarkt"	106
Video 7: Coding Kurzgeschichten-Video über Extension Methods	129
Video 8: Coding Kurzgeschichten-Video über Enums in C#: Die vier Jahreszeiten	135
Video 9: Coding Kurzgeschichten-Video über DateTime und TimeSpan in C#.....	136
Video 10: Coding Kurzgeschichten-Videos über Value und Reference Types.....	142
Video 11: Coding Kurzgeschichten-Videos über Stack und Heap.....	147
Video 12: Coding Kurzgeschichten-Video über HashSets.....	168
Video 13: Coding Kurzgeschichten-Video über Dictionaries	173

35.3 Ideenliste für Ergänzungen in diesem Skriptum

Soll unbedingt einmal rein

- LINQ
- async/await
- Records (ab C#9)
- Interface Enumerable
- Nullable Types
- WPF