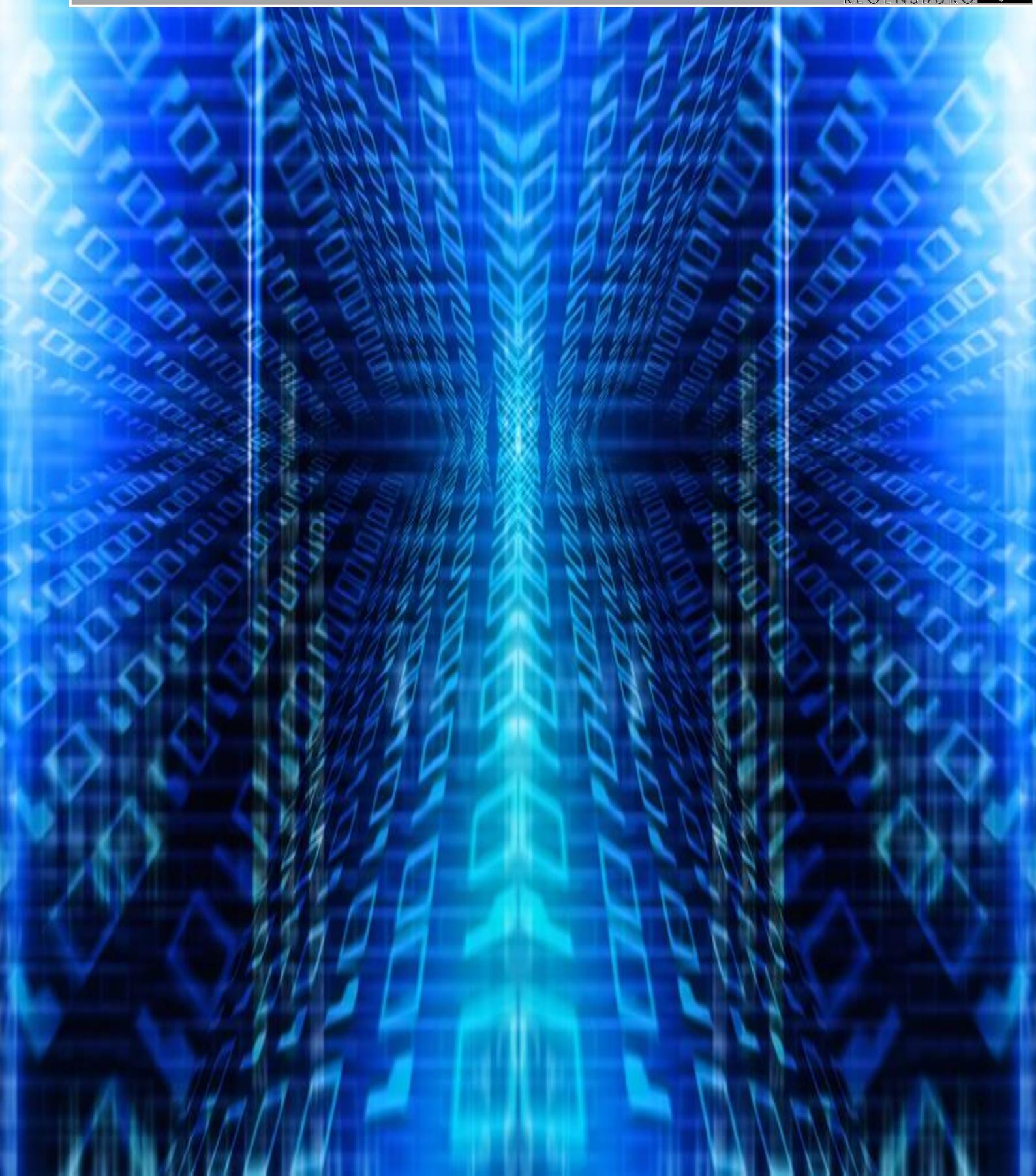


Programmieren 2

Technische Informatik
Ostbayerische Technische Hochschule Regensburg
Sommersemester 2015



1 Inhaltsverzeichnis

1	Inhaltsverzeichnis	1
2	Einleitung	10
3	Setup	11
3.1	Ein Bild sagt mehr als tausend Worte	11
3.2	Visual Studio und Windows installieren	11
3.2.1	Version und Ausführung von Visual Studio	12
3.2.2	Windows unter Mac OS X und Linux installieren	12
3.3	Erste Schritte mit Visual Studio	12
3.3.1	Hello World in C#	12
3.3.2	Weitere wichtige Grundlagen in Visual Studio	13
3.3.3	Was ist .NET?	13
3.3.4	Updates und Erweiterungen zu Visual Studio	14
4	C# - die Grundlagen	16
4.1	Anatomie von Funktionen	16
4.2	Die Main Methode	17
4.3	Imperativen Code in Funktionen schreiben	18
4.3.1	Variablendefinition und Wertzuweisung	18
4.3.2	Methoden aufrufen	19
4.3.3	Parameter in Funktionen einsetzen	20
4.3.4	Anweisungen und Ausdrücke	21
4.3.5	Primitive Datentypen	22
4.3.5.1	Ganzzahltypen	23
4.3.5.2	Gleitkommazahltypen	24
4.3.5.3	Boolesche Werte	26
4.3.5.4	Zeichenketten und Buchstaben	26
4.3.5.5	Referenztypen	27
4.3.6	Typumwandlung und –Konvertierungen	27
4.3.6.1	Implizite Konvertierungen	28
4.3.6.2	Explizite Konvertierungen	29
4.3.6.3	Konvertierungen mit Helfer-Klassen	30
4.3.7	Arrays	30
4.3.8	Kontrollflussstrukturen	32
4.3.8.1	If – Else Blöcke	32
4.3.8.2	Switch Anweisungen	34
4.3.8.3	While und Do-While Schleife	35

4.3.8.4	For und Foreach Schleife	37
4.3.8.5	Die Schlüsselwörter continue und break für Schleifen	39
4.3.9	Operatoren	40
4.3.9.1	Arithmetische Operatoren für numerische Ausdrücke	40
4.3.9.2	Vergleichsoperatoren.....	41
4.3.9.3	Logische Operatoren für boolesche Ausdrücke	41
4.3.9.4	Bitweise Operatoren für numerische Ausdrücke	42
4.3.9.5	Zuweisungsoperatoren.....	42
4.3.9.6	Weitere Operatoren.....	43
4.3.9.7	Aufrufreihenfolge von Operatoren	43
4.4	Datum und Zeit.....	44
5	Klassen, Kapselung und Vererbung	45
5.1	Klassendefinition und –scope.....	45
5.2	Methoden als Klassenmitglieder.....	46
5.3	Felder als Klassenmitglieder.....	46
5.4	Von Klassen zu Objekten	47
5.5	Die Modifizierer public und private und die this Referenz	49
5.6	Der Modifizierer static.....	51
5.7	Kapselung – welche Mitglieder mache ich private, welche public?.....	52
5.8	Zugriff auf Felder über Eigenschaften gewähren	55
5.9	Namensräume	59
5.10	Konstruktoren.....	61
5.10.1	Grundsätzliches zu Konstruktoren	61
5.10.2	Inline-Initialisierung von Feldern.....	63
5.10.3	Statische Konstruktoren	64
5.10.4	Der Modifizierer readonly	65
5.10.5	Konstruktoraufrufe verketten	66
5.10.6	Objektinitialisierungssyntax für Konstruktoraufrufe.....	67
5.10.7	Destruktoren	68
5.11	Konstanten mit dem Schlüsselwort const.....	69
5.12	Vererbung.....	70
5.12.1	Grundlagen der Klassenvererbung	70
5.12.2	Die Ist-Eine-Beziehung zwischen Basis- und Subklasse	71
5.12.3	Casting von Basisklasse zu Subklasse	72
5.12.4	Klassenhierarchien	74
5.12.5	Der Modifizierer protected	75

5.12.6	Virtuelle Methoden	76
5.12.6.1	Grundsätzliches zu virtuellen Methoden	76
5.12.6.2	Dynamische Bindung bei virtuellen Methoden	77
5.12.6.3	Die Funktionalität von virtuellen Methoden erweitern	78
5.12.7	Abstrakte Methoden und abstrakte Klassen.....	79
5.12.8	Object – die ultimative Basisklasse	81
5.12.9	Mit Modifizierer sealed	83
5.12.10	Statische Klassen, statische Mitglieder und Vererbung	83
5.12.11	Vererbung mit Interfaces	83
5.12.11.1	Grundsätzliches zur Interfacedefinition	83
5.12.11.2	Interfaces in Klassen implementieren	84
5.12.11.3	Die Ist-Eine-Beziehung bei Interfaces.....	85
5.12.11.4	Mehrere Interfaces in eine Klassen implementieren	85
5.12.11.5	Mehrfachvererbung mit Interfaces	86
5.13	Weitere interessante C# Features.....	87
5.13.1	Das Schlüsselwort var	87
5.13.2	Die Modifizierer ref und out für Parameter	88
5.13.3	Attribute	89
5.13.4	Operatorenüberladung	90
5.13.5	Implizite und explizite nutzerdefinierte Typumwandlungen	91
5.13.6	Alle Modifizierer im Überblick.....	93
5.14	Exceptions	94
5.14.1	Exceptions werfen mit dem Schlüsselwort throw.....	94
5.14.2	Exceptions mit try-catch Blöcken behandeln	96
5.14.3	System.Exception – die Basisklasse aller Exceptions	97
5.14.4	Eigene Exceptionklassen implementieren.....	98
5.14.5	Mehrere catch-Blöcke für unterschiedliche Exceptiontypen verwenden.....	100
5.14.6	Wann sollte man Exceptions auslösen?	101
5.14.7	Wann sollte man Exceptions behandeln?	102
5.15	Collections und Generics	104
5.15.1	List<T> – die am häufigsten benutzte Collection	104
5.15.2	Was sind Generics?.....	105
5.15.2.1	Ein Stack für Strings	106
5.15.2.2	Don't repeat yourself – ein Stack für int Werte	107
5.15.2.3	Alles ist object – der Stack für object	108
5.15.2.4	Der generische Stack – typsicher und DRY	110

5.15.3	Weitere Klassen für Collections.....	112
5.15.4	Collections über Interfaces ansprechen.....	113
5.15.4.1	Die Interfaces <code>IEnumerable<T></code> , <code>ICollection<T></code> und <code>IList<T></code>	113
5.15.4.2	<code>IEnumerable<T></code> und <code>IEnumerator<T></code> im Detail (Level 200)	114
5.15.4.3	Die Funktionsweise der <code>foreach</code> Schleife (Level 200).....	119
5.15.4.4	Endlosschleifen mit <code>foreach</code> (Level 200).....	119
5.15.4.5	Nur lesend auf Collections via Interfaces zugreifen	121
5.15.5	Wann sollte man Collections einsetzen?.....	122
5.16	Fortgeschrittene Möglichkeiten mit Generics und Reflection	123
5.16.1	Immutable Objects und Value Objects – Die Klasse Farbe.....	123
5.16.2	Das Interface <code>IEquatable<T></code>	126
5.16.3	Verschiedene Value Objects und damit verbundene DRY-Probleme	128
5.16.4	Reflection.....	129
5.16.4.1	Grundsätzliches zu Reflection	129
5.16.4.2	Die Enumeration <code>BindingFlags</code>	131
5.16.4.3	Die Vererbungshierarchie zu den wichtigsten Reflection-Klassen.....	133
5.16.5	Eine generische Basisklasse für Value Objects (Level 200)	133
5.16.6	Zusammenfassung für Generics	136
5.17	Das Typsystem von C#	138
5.17.1	Welche Typen unterstützt C#?	138
5.17.2	Enumerationen.....	138
5.17.2.1	Enumerationen definieren	138
5.17.2.2	Enumerationen einsetzen	140
5.17.2.3	Enumerationen als Bitfelder verwenden.....	141
5.17.2.4	Enum – die Basisklasse für Enumerationen.....	142
5.17.2.5	Ein komplexeres Beispiel für den Einsatz von Enumerationen	143
5.17.2.6	Wann sollte man Enumerationen einsetzen?	144
5.17.3	Delegates	144
5.17.3.1	Delegates definieren und instanziieren.....	145
5.17.3.2	Delegates aufrufen	146
5.17.3.3	Das Hollywood-Prinzip anhand von <code>System.Threading.Timer</code>	147
5.17.3.4	Die Vererbungshierarchie für Delegates	151
5.17.3.5	Die generischen Delegates <code>Action</code> und <code>Func</code> von .NET	153
5.17.3.6	Events	154
5.17.3.6.1	Events definieren	154
5.17.3.6.2	Methoden an Events registrieren	155

5.17.3.6.3	Die Kurzschreibweise für Events	156
5.17.3.6.4	Wann und wie sollte man Events einsetzen?.....	157
5.17.3.7	Anonyme Methoden und Lambdas	158
5.17.3.7.1	Anonyme Methoden mit dem Schlüsselwort delegate.....	158
5.17.3.7.2	Closure bei anonymen Methoden.....	159
5.17.3.7.3	Lambdas – die kürzere Schreibweise für anonyme Methoden.....	160
5.17.3.7.4	LINQ als Beispiel für den Einsatz von Lambdas	161
5.17.3.7.5	Wie LINQ intern funktioniert (Level 200)	164
5.17.3.8	Zusammenfassung für Delegates	166
5.17.4	Strukturen.....	167
5.17.4.1	Definition von Strukturen.....	167
5.17.4.2	Unterschiede zwischen Strukturen und Klassen	168
5.17.4.3	Strukturen einsetzen	168
5.17.4.4	Wann sollte man Strukturen einsetzen?	169
5.18	Die Common Language Runtime.....	171
5.18.1	Just-In-Time Compilation der CLR	171
5.18.2	Speichermanagement in der CLR	173
5.18.2.1	Wofür ist der Stack zuständig, wofür der Heap?.....	173
5.18.2.2	Laufzeitverhalten von Stack und Heap visualisieren.....	174
5.18.2.2.1	Grundsätzlicher Aufbau der Visualisierung.....	174
5.18.2.2.2	Objektreferenzen innerhalb des Heaps	176
5.18.2.2.3	Statische Felder und Delegates im Speicher	177
5.18.2.3	Garbage Collection in der CLR	177
5.18.2.4	Unterschiede zwischen Beispiel und Wirklichkeit.....	178
5.19	Persistenz mit .NET.....	180
5.19.1	Dateizugriffe	180
5.19.1.1	Überblick über Speichern und Laden	180
5.19.1.2	Einfacher Zugriff auf Binär- und Textdateien	182
5.19.1.2.1	Die Klassen File und FileInfo.....	182
5.19.1.2.2	Zugriff auf Ordner im Dateisystem.....	183
5.19.1.2.3	Pfade zu Dateien und Ordnern.....	184
5.19.1.2.4	Vereinfachte Pfadangabe im Code durch Verbatim-Stringliterale	185
5.19.1.2.5	Die Klasse DriveInfo.....	186
5.19.1.2.6	Dateisystemexplorer als Beispiel	186
5.19.1.3	Dateistreaming	189
5.19.1.3.1	Übersicht über Streams in .NET	189

5.19.1.3.2	Die Klasse FileStream im Beispiel	190
5.19.1.4	Serialisierung und Deserialisierung	191
5.19.1.5	Das Serialisierungsformat XML.....	193
5.19.1.6	Objekte automatisch zu XML umformen in .NET	196
5.19.1.7	DataContractSerializer zum Laden und Speichern einsetzen.....	197
5.19.1.8	Zusammenfassung für Zugriff auf Dateien	199
5.19.2	Ein kurzer Einblick in den Zugriff auf Datenbanken (Level 200).....	200
5.19.2.1	Was sind Datenbanken?	200
5.19.2.2	Zugriff auf relationale Datenbanken	201
5.19.2.3	Der Objektrelationale Mapper Entity Framework im Beispiel	201
5.19.2.3.1	Wie bekomme ich Entity Framework?	201
5.19.2.3.2	Das zu speichernde Klassenmodell	203
5.19.2.3.3	Die Klasse DbContext	204
5.19.2.3.4	Moment – in welches Datenbanksystem schreibe ich eigentlich gerade?	206
5.19.2.3.5	Verbindung zum Datenbanksystem in Visual Studio aufnehmen.....	207
5.19.2.3.6	Zusammenfassung für Objektrelationale Mapper	209
5.19.3	Die Unterscheidung zwischen Entities und Value Objects	210
5.20	Oberflächenprogrammierung mit WPF in .NET	212
5.20.1	Hello World mit WPF	212
5.20.1.1	Ein WPF Projekt erstellen	212
5.20.1.2	XAML und Code-Behind-Datei modifizieren.....	213
5.20.1.3	Was haben wir gelernt?.....	215
5.20.2	Die verschiedenen Elemente in WPF	215
5.20.2.1	Visuelle Elemente	216
5.20.2.2	Controls	216
5.20.2.3	Panels	217
5.20.2.4	Items Controls	218
5.20.2.5	Fenster und User Controls.....	219
5.20.2.6	Wo finde ich weitere Dokumentationen zu WPF Elementen?.....	220
5.20.3	Aufbau üblicher Ansichten	220
5.20.3.1	Master-Detail-Ansichten	220
5.20.3.2	Formularansichten.....	224
5.20.4	Die Klasse App	227
5.20.5	Der Renderprozess des User Interfaces und Threading.....	229
5.20.5.1	Der User Interface Thread	229
5.20.5.2	Methoden auf Hintergrundthreads auslagern	231

5.20.5.3	Threadaffinität von UI-Elementen.....	234
5.20.5.4	Die Schlüsselwörter <code>async</code> und <code>await</code>	235
5.20.5.5	Zusammenfassung für GUI und Threading.....	237
5.20.6	Wichtige Klassen in der WPF Vererbungshierarchie (Level 200)	237
5.20.7	Data Binding und Dependency Properties in WPF (Level 200)	239
5.20.8	Zusammenfassung für WPF und Oberflächenprogrammierung	242
6	Objektorientiertes Design (OOD)	244
6.1	Die Abgrenzung zwischen objektorientiert und nicht-objektorientiert.....	244
6.2	SOLID Principles of Object-Oriented Design.....	245
6.2.1	Dependency Inversion Principle (DIP)	246
6.2.1.1	Kopierprogramm als Ausgangsbeispiel für das DIP	246
6.2.1.2	Das Problem der direkten Abhängigkeit	247
6.2.1.3	Programmieren gegen Abstraktionen als Lösung des Problems.....	248
6.2.1.4	Zusammenfassung für DIP	252
6.2.2	Single Responsibility Principle (SRP).....	252
6.2.3	Open / Closed Principle (OCP)	253
6.2.4	Liskov Substitution Principle (LSP).....	254
6.2.5	Interface Segregation Principle (ISP)	256
6.2.6	Zusammenfassung für die SOLID-Prinzipien.....	257
6.3	Weitere bekannte Prinzipien.....	257
6.3.1	Don't repeat yourself (DRY).....	257
6.3.2	Favor Composition over Inheritance	258
6.3.3	Hollywood Prinzip (Inversion of Control)	258
6.3.4	Tell Don't Ask (Level 200)	258
6.3.5	Keep it simple and sweet (KISS)	258
6.3.6	Zusammenfassung für Designprinzipien	258
6.4	Objektorientierte Analyse (OOA)	259
6.4.1	Die Schritte der objektorientierten Analyse.....	259
6.4.2	Ein Beispiel zur objektorientierten Analyse	259
6.4.2.1	Schritt 1: Substantive erfassen	260
6.4.2.2	Schritt 2: Relationen erfassen	260
6.4.2.3	Schritt 3: Verben erfassen	260
6.4.2.4	Schritt 4: Adjektive und Adverbiale erfassen	261
6.4.2.5	Schritt 5: Klassenmodell aufbauen und verfeinern	261
6.4.3	Zusammenfassung für die objektorientierte Analyse	262
6.5	Objektorientierte Design Patterns	262

6.5.1	Historie zu Design Patterns (Level 200).....	263
6.5.2	Organisation von Design Patterns.....	263
6.5.3	Warum Design Patterns?.....	264
6.5.4	Factory	265
6.5.5	Command	267
6.5.6	Decorator.....	268
6.5.7	Adapter.....	269
6.5.8	Strategy	270
6.5.9	Memento (Level 200)	271
6.5.10	Repository und Unit of Work (Level 200).....	272
6.5.11	Observer (Level 200)	274
6.5.12	Model-View-View Model (MVVM) (Level 200)	274
6.5.13	Zusammenfassung für Design Patterns.....	275
6.6	Eine Einführung in automatisiertes Testen (Level 200)	276
6.6.1	Was genau bedeutet automatisiertes Testen?	276
6.6.2	Automatisierte Tests in Visual Studio.....	277
6.6.3	Die Struktur von automatisierten Tests	279
6.6.4	Was sollte ich testen?.....	281
6.6.5	Testgetriebene Entwicklung	281
6.6.6	Zusammenfassung für automatisiertes Testing	282
7	Objektorientierte Programmierung in C++	283
7.1	Hello World in C++.....	283
7.2	Die wichtigsten Unterschiede zwischen C# und C++ auf einen Blick	287
7.3	Modern C++.....	288
7.4	Primitive Datentypen und Flow Control in C++.....	288
7.5	Zeiger	288
7.6	Konstante Werte und Zeiger	290
7.7	Referenzen in C++.....	291
7.7.1	Ivalue Referenzen	291
7.7.2	Ivalue Referenzen und die Range-Based-For-Loop	292
7.7.3	Konstante Ivalue Referenzen.....	294
7.7.4	rvalue Referenzen (Level 200)	295
7.8	Wann setze ich Pointer und Referenzen ein?	296
7.9	Klassen in C++	297
7.9.1	Klassendeklaration und –Definition	297
7.9.2	Klassen auf Stack oder Heap instanzieren.....	300

7.9.3	Konstruktoren in C++.....	302
7.9.3.1	Explizite Konstruktoren	302
7.9.3.2	Copy-Konstruktoren	304
7.9.4	Destruktoren in C++.....	305
7.9.5	Statische Klassenmethoden	306
7.9.6	Konstante Klassenmethoden.....	307
7.9.7	Klassenvererbung in C++	308
7.9.7.1	Von einer Basisklasse ableiten	308
7.9.7.2	Die Ist-Eine-Beziehung zwischen Basis- und Subklasse in C++	310
7.9.7.3	Abstraktionen in C++	312
7.9.7.4	Abstraktionen implementieren	313
7.9.7.5	Mehrfachvererbung mit Klassen	314
7.9.7.6	Das Diamond Problem und virtuelle Vererbung in C++ (Level 200).....	315
7.9.7.7	Vererbung verhindern	318
7.10	Templates.....	319
7.10.1	Template-Funktionen	319
7.10.2	Template-Klassen	320
7.11	Operatorenüberladung in C++.....	322
7.11.1	Operatorenüberladungen deklarieren und definieren	322
7.11.2	Functors (Level 200)	323
7.12	Wichtige Klassen der Standard Library.....	324
7.12.1	Wichtige Headerdateien der Standard Library.....	324
7.12.2	Smart Pointers	324
7.13	Große C++ Softwarelösungen aufbauen	326

2 Einleitung

Willkommen bei Programmieren 2. In diesem Kurs lernen Sie die objektorientierte Programmierung kennen. Dazu schauen wir uns zwei objektorientierte Sprachen an: C#, eine sog. „Managed Programming Language“, sowie C++, eine native objektorientierte Programmiersprache. Neben den Programmiersprachen an sich werfen wir auch einen Blick in die dazugehörigen Frameworks, .NET 4.5 bei C# sowie die Standard Library, die bei jedem vollwertigen C++ Compiler mitgeliefert wird.

Ebenso betrachten wird die Theorie hinter Objektorientierter Programmierung (abgekürzt OOP): angefangen bei den grundlegenden Prinzipien Kapselung, Vererbung und Polymorphie bis hin zu Design Prinzipien und Design Patterns, die beim Strukturieren von Code helfen, werden objektorientierte Konzepte und Hilfsmittel besprochen, die unabhängig von der jeweiligen Programmiersprache eingesetzt werden können.

Denn das wichtigste Ziel in der Softwareentwicklung ist auch heutzutage noch gut lesbarer Code, der flexibel ist, leicht gepflegt werden kann und leicht testbar ist. Viele Softwareprojekte scheitern bei zunehmender Komplexität an unflexiblen Code, dessen Bestandteile zu sehr gekoppelt und funktionell zu groß geschnitten sind, sodass subtile Bugs schnell Einzug halten. Design Patterns und Prinzipien können hier einen großen Schritt weiterhelfen, auch wenn sie für Programmieranfänger wahrscheinlich zunächst nicht intuitiv wirken.

Am Ende dieses Kurses sollten Sie zu den oben genannten Themen eine fundierte Aussage treffen und die gelernten Konzepte auch im Programmieralltag einsetzen können. Auf diesem Weg möchte ich Ihnen noch folgenden, meines Erachtens wichtigen Hinweis geben: bitte programmieren Sie. Es wird nicht reichen, den Code der Vorlesung nur zu lesen. Die Prüfung wird zu einem Großteil aus Programmieraufgaben bestehen und aus Erfahrung kann ich Ihnen nur bestätigen, dass es sehr schwer wird, diese Aufgaben zu lösen, wenn Sie vorher nicht aktiv selbst programmiert haben. Ich kann Ihnen auch nur empfehlen, nicht alleine zu programmieren, sondern paarweise und sich dabei beständig abzuwechseln.

Und mein letzter Tipp: wenn Sie ein guter Programmierer werden möchten, sollten Sie auf jeden Fall der englischen Sprache mächtig sein. Die besten Lernmittel und die Dokumentation der neuesten Frameworks erscheinen hauptsächlich auf Englisch. Ich persönlich werde den Code auf Deutsch schreiben und bei Fachbegriffen sowohl den englischen als auch die deutsche Variante aufführen, da sonst Personen, die zum jetzigen Zeitpunkt nicht gut Englisch können, einen deutlichen Nachteil hätten. Dennoch: Englisch ist in der Programmierung meines Erachtens nach ein Muss.

In diesem Sinne möchte ich mit einem Zitat (auf Englisch ☺) von Programmierlegende Martin Fowler diese Einleitung abschließen: „Any fool can write code that a computer can understand. Good programmers write code that humans can understand.“

3 Setup

In dieser Kapitel schauen wir uns den Aufbau dieses Scripts an. Ich gebe einige Hinweise, wie Sie an die unterschiedlichen Lerninhalte herangehen sollten. Des Weiteren zeige ich Ihnen, wie sie Visual Studio, das Programm, mit dem wir Code schreiben und kompilieren, installieren und verwenden.

3.1 Ein Bild sagt mehr als tausend Worte...

...und bewegte Bilder meines Erachtens noch viel mehr. Sie werden zu den einzelnen Lehrinhalten in diesem Script sowohl Text als auch Verweise auf Videos finden. In den Videos erkläre ich Ihnen die Inhalte direkt in der Entwicklungsumgebung, in der ich vorprogrammiere, was im jeweiligen Abschnitt besprochen wurde.

Sie haben so die Möglichkeit, sich die Lerninhalte auf zwei Arten zu erschließen: lesend über das Script oder per YouTube-Video. Meinen Channel finden Sie unter
https://www.youtube.com/channel/UCfS1gb_r6GUfPd1EV82aBzg.

Mein Tipp: schauen Sie sich die Videos in 1,5-facher Geschwindigkeit auf YouTube an. Man kann meinen Ausführungen noch gut folgen und spart sich so ein Drittel der Zeit.

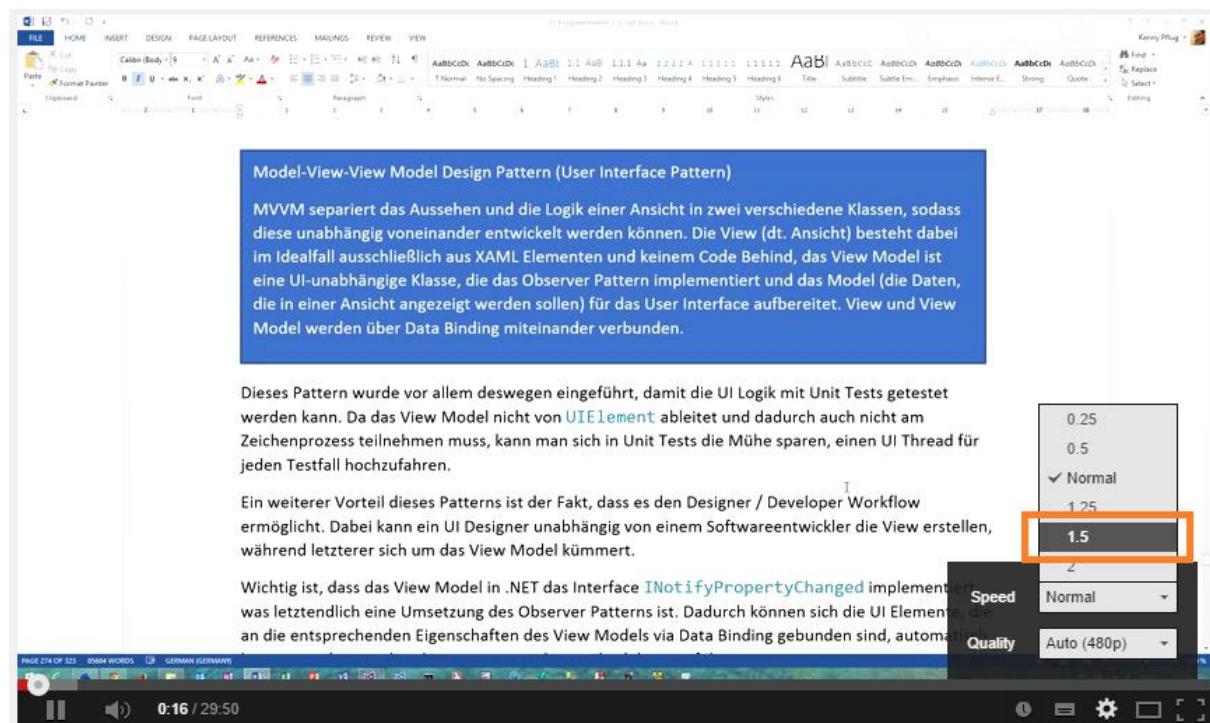


Abbildung 1: Videos auf YouTube schneller abspielen

3.2 Visual Studio und Windows installieren

Visual Studio ist Microsofts primäre Entwicklungsumgebung (englisch Integrated Development Environment, abgekürzt IDE) und läuft nur unter Windows. Beide Softwareprodukte sind für Studenten kostenlos erhältlich bei [DreamSpark](#), genauso wie viele andere Produkte von Microsoft. Des Weiteren ist Visual Studio auf jedem CIP-Pool Rechner der Fakultät Informatik und Mathematik installiert und kann dort genutzt werden.

3.2.1 Version und Ausführung von Visual Studio

Zurzeit ist Visual Studio 2013 die aktuellste Version, die im November 2013 erschienen ist. Visual Studio wird hauptsächlich in drei verschiedenen Ausführungen angeboten: Professional, Premium und Ultimate. Für die Vorlesung reicht die Professional-Variante, ich würde aber empfehlen, Ultimate zu installieren, da dort einige interessante Tools mit integriert sind, die in Professional nicht enthalten sind, wie z.B. Code Coverage Analysetools oder Architekturtools, die aus Code UML Diagramme erstellen können.

Visual Studio 2013 läuft nur unter Windows 7 bzw. Windows 8 und belegt in der Standardinstallation knappe 7 GB auf der Festplatte. Bei der Installation ist nichts weiter zu beachten. Ein Video zum Download der Software findet man unter http://youtu.be/Q_3L4IgZ4M4.

Mittlerweile ist auch eine sog. Community Edition von Visual Studio erhältlich (<https://www.visualstudio.com/products/visual-studio-community-vs>), die an sich identisch im Funktionsumfang mit der Professional Edition ist, mit der Sie aber auch kommerzielle Produkte erstellen können. Die Community Edition ist für jedermann frei erhältlich (sogar für Unternehmen mit bis zu fünf Entwicklern).

3.2.2 Windows unter Mac OS X und Linux installieren

Wer OS X oder Linux als Betriebssystem sein eigen nennt, dem empfehle ich, eine virtuelle Maschine oder eine parallele Betriebssysteminstallation mit Windows 7 / 8 aufzusetzen und auf dieser Visual Studio zu installieren. Visual Studio über Wine unter Linux auszuführen kann ich nicht empfehlen.

Eine kostenlose Software für virtuelle Maschinen ist Virtual Box. Unter OS X kann man Boot Camp nutzen, um eine parallele Windowsinstallation einzurichten. Der virtuellen Maschine sollten ca. 50 GB Festplattenspeicher und ca. 2, besser 4 GB Arbeitsspeicher zugewiesen werden.

3.3 Erste Schritte mit Visual Studio

3.3.1 Hello World in C#

In diesem Abschnitt schauen wir uns das einfachste aller C# Programme an: Hello World. Dabei sehen wir auch, wie man mit Visual Studio Projekte erstellt, Code schreibt und debuggt. Das dazugehörige Video können Sie hier finden: <http://youtu.be/uI1Xnlh5SuQ>

Ich empfehle Ihnen, sich direkt jetzt das Video anzusehen. Hier sind nochmal die wichtigsten Schritte:

- Wenn man Visual Studio gestartet hat, kann man über File -> New Project... den Dialog öffnen, über den neue Projekte erstellt werden können.
- Nachdem das Projekt erstellt wurde, sieht man mehrere Dateien im Solution Explorer, die man auch im Dateiexplorer wiederfindet, darunter
 - eine Solution- / Projektmappendatei mit der Endung *.sln
 - eine Projektdatei mit der Endung *.csproj
 - mehrere zum Projekt dazugehörige Dateien, wobei die wichtigste Program.cs ist.
- In der Datei Program.cs findet man eine Klasse **Program**, in der die Funktion Main definiert ist, der Einstiegspunkt eines C# Programms. In diese schreibt man folgenden Code:

```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
    Console.ReadLine();
}
```

- Mit dem Startbutton in der obersten Leiste wird das Programm kompiliert und ausgeführt. Alternativ können Sie auch einfach F5 drücken.
- Falls Sie nur das Projekt erstellen möchten, ohne es auszuführen, können Sie F6 oder STRG + Shift + B drücken.
- Wer sein Konsolenprogramm mit STRG + F5 statt nur F5 ausführt, kann sich den letzten `Console.ReadLine` Befehl sparen (das ist aber nicht hilfreich, wenn man das Programm direkt über die *.exe Datei startet). Mit STRG + F5 ist kein Debugging möglich, da sich hier Visual Studio nicht an den Prozess anheftet.
- Über die linke Leiste neben dem Codeeditor oder über F9 lassen sich Break Points setzen, an denen bei Ausführung eines Programms in Visual Studio angehalten wird. Dadurch kann man schrittweise den Code ausführen mit den folgenden Kommandos:
 - Mit F10 kann man die aktuelle Anweisung ausführen lassen.
 - Mit F11 springt man in die Funktion, die als nächstes ausgeführt werden würde.
 - Mit F5 nimmt man die normale Abarbeitung der Anweisungen wieder auf.
 - Wenn man Anweisungen mit mehreren Funktionsaufrufen hat, kann man mit Shift + Alt + F11 direkt die Funktion auswählen, in die gesprungen werden soll. Weitere Infos zu Anweisungen und Ausdrücken finden Sie in Kapitel 4.3.4.

3.3.2 Weitere wichtige Grundlagen in Visual Studio

Im Video http://youtu.be/UceUJk_nSz4 werden weitere wichtige Grundlagen gezeigt. Visual Studio besitzt ein flexibles Fenstersystem, sodass beliebige Fenster an bestimmten Bereichen der IDE angedockt werden oder als eigenes Fenster frei über dem Hauptfenster liegen können. Wenn ein Fenster geschlossen wurde (meistens wohl aus Versehen), dann kann man die wichtigsten Fenster unter dem Menüpunkt „View“ bzw. andere Fenster wie den Test Explorer unter den anderen entsprechenden Menüpunkten in der Menüleiste finden und wieder sichtbar machen.

3.3.3 Was ist .NET?

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=ty1Of5RA22I>.

.NET ist ein sog. Framework (dt. Software-Bibliothek, wörtlich Rahmenwerk), das von Microsoft erstellt, gepflegt und mit Visual Studio automatisch ausgeliefert wird. Ein Framework stellt Software-Artefakte zur Verfügung, die bestimmte Funktionalitäten mitbringen und die man in seinen eigenen Softwareprojekten einsetzen kann, um bestimmte Probleme zu lösen (Wiederverwendung).

Das .NET Framework ist in unterschiedliche sog. Assemblies aufgeteilt, in dem die wiederverwendbaren Artefakte je nach Anwendungsgebiet (bspw. Konsolenausgabe, Laden- und Speichern von Dateien, Kommunikation mit anderen Prozessen oder Oberflächenprogrammierung) zu finden sind. Assemblies beinhalten vorkompilierten Code, sodass dieser beim Erstellen des eigenen Projekts nicht erneut kompiliert werden muss. Ein Konsolenprojekt nutzt dabei folgende Assemblies automatisch, wie man im Punkt References unterhalb eines Projekts im Solution Explorer (dt. Projektmappen-Explorer) sehen kann:

- Microsoft.CSharp
- System
- System.Core
- System.Data
- System.Data.DataSetExtensions
- System.Xml
- System.Xml.Linq

Mit einem Rechtsklick auf den Punkt References innerhalb eines Projekts kann man Assemblies über einen Dialog hinzufügen oder entfernen. Dabei kann man nicht nur Teile des .NET Frameworks referenzieren, sondern auch eigene Assemblies, die bspw. im Dateisystem liegen oder eigene Projekte in derselben Solution.

Im Microsoft Developer Network (MSDN) gibt es eine Online-Lernquelle (MSDN Library), in der sämtliche Typen des .NET Frameworks ausführlich dokumentiert sind. Nebenbei gibt es viele Artikel, die eine Zusammenfassung zu einem jeweiligen Themengebiet, wie bspw. Dateisystemzugriffe oder Oberflächenprogrammierung, bieten. Viele dieser Artikel und Dokumentation sind auch auf Deutsch verfügbar, leider allerdings nicht alle. Die MSDN Library ist auf jeden Fall der erste Nachschlagpunkt, wenn man wissen möchte, wie bspw. eine bestimmte Klasse des .NET Frameworks funktioniert. Zu finden ist die Website unter <http://msdn.microsoft.com/de-de/library> für die deutsche Version, die englischsprachige findet man unter <http://msdn.microsoft.com/en-us/library>.

Daneben gibt es mit NuGet einen Paketmanager, mit dem man sich kostenfreie Assemblies vom www.nuget.org Feed besorgen und verwalten kann. Microsoft selbst ist zurzeit auf dem Weg, einige Teile des .NET Frameworks direkt auf NuGet auszulagern, um die verschiedenen Projekte unabhängiger voneinander veröffentlichen zu können. So sind bspw. ASP.NET und Entity Framework als Open Source Projekte auch über NuGet erhältlich.

3.3.4 Updates und Erweiterungen zu Visual Studio

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=WCrKCtSI1gc>.

Im rechten oberen Bereich links neben den Minimieren-, Maximieren- und Schließen-Buttons befindet sich ein kleines FahnenSymbol, welches das Notification- / Benachrichtigungsfenster öffnet. In diesem wird man über Updates zu Visual Studio informiert. Aktuell ist Update 4 die neueste Version für Visual Studio. Diese allgemeinen Updates kann man bedenkenlos installieren.

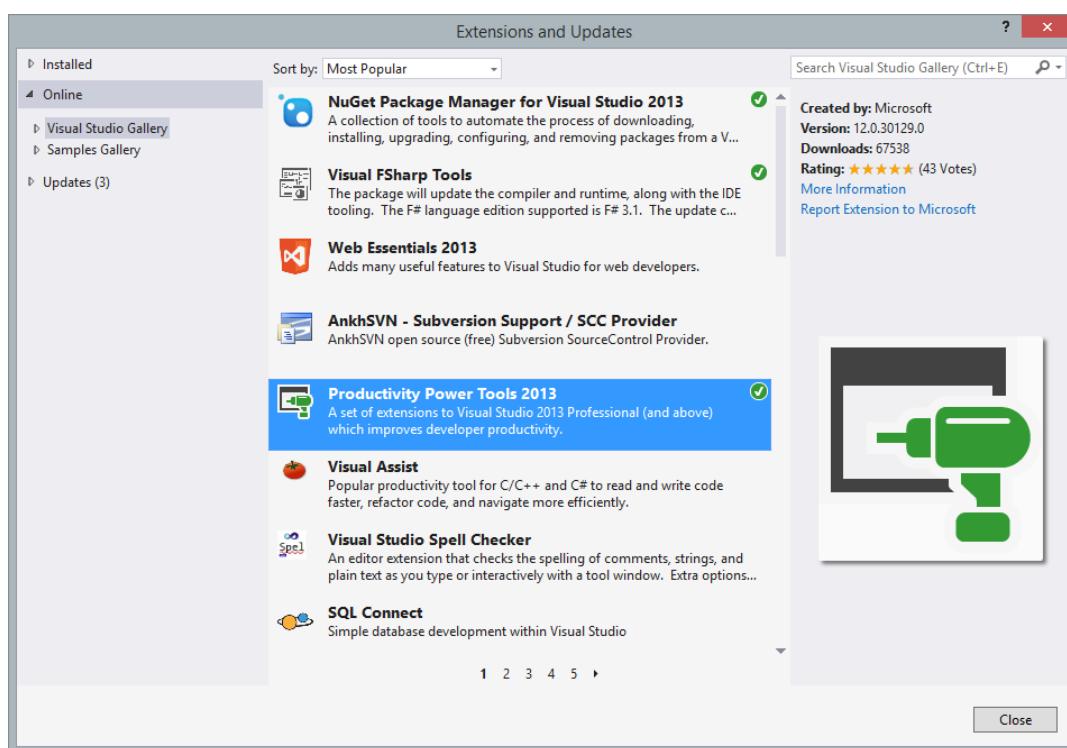


Abbildung 2: Der Extensions and Updates Dialog in Visual Studio 2013

Des Weiteren gibt es unter dem Menüpunkt Tools -> Extensions and Updates einen Dialog, mit dem sich kostenfreie und ebenfalls kostenpflichtige Erweiterungen zu Visual Studio hinzufügen lassen können. Was ich auf jeden Fall empfehlen kann, sind die Productivity Power Tools 2013, die einige angenehme Funktionalitäten zu Visual Studio hinzufügen. Um nicht vorhandene Erweiterungen zu installieren, kann man in der Baumstruktur links den Punkt Online auswählen und im Suchfenster oben rechts den Begriff „Power Productivity Tools“ eingeben. Dann einfach den Eintrag in der Mitte auswählen und installieren drücken.

Ebenfalls links in der Baumstruktur findet man den Punkt „Updates“, mit dem man bereits installierten Erweiterungen auf den neuesten Stand bringen kann.

Die Productivity Power Tools kann man unter Tools -> Options konfigurieren. Ich persönlich habe die folgenden Punkte deaktiviert, da sie meines Erachtens nicht notwendig sind bzw. das Erscheinungsbild des Codes stören:

- CTRL + Click Go To Definition
- Custom Document Well
- Structure Visualizer

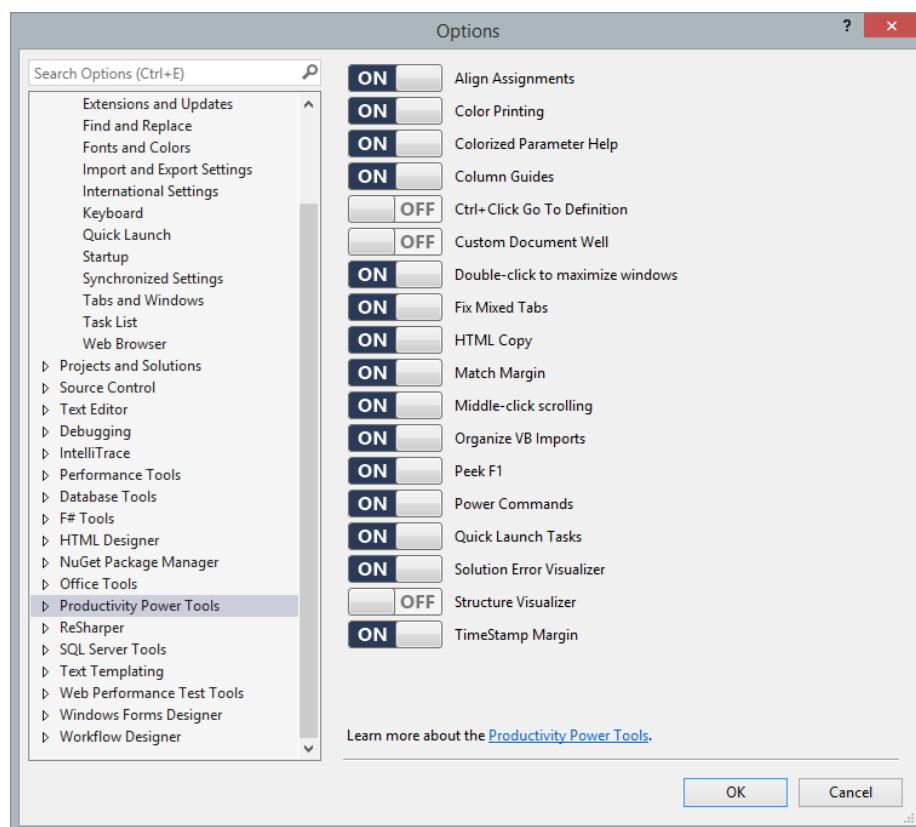


Abbildung 3: Der Options Dialog zur Konfiguration der Einstellungen von Visual Studio

Nach den Änderungen muss Visual Studio einmal neugestartet werden, damit diese aktiv werden.

In meinen Videos verwende ich zum großen Teil eine Visual Studio Version, bei der ein Plug-In namens ReSharper aktiviert ist. ReSharper ist ein sehr umfangreiches Tool, das die bei Visual Studio mittgelieferte Codevervollständigung namens IntelliSense nahezu vollständig ersetzt und darüber hinaus weitere sinnvolle Funktionen anbietet. Wenn Sie ReSharper ebenfalls nutzen möchten, können Sie sich als Student die Software kostenlos hier runterladen:

<https://www.jetbrains.com/student/>

4 C# - die Grundlagen

In diesem Kapitel beschäftigen wir uns mit den Grundlagen der Programmiersprache C#, wozu Funktionen, Variablen, Parameter, imperativer Code und die Struktur von C#-Programmen gehören. C# ist eine Programmiersprache, deren Syntax sehr ähnlich der von C / C++ ist, weswegen Sie ihr Wissen, das Sie in PG1 mit C gelernt haben, auch in C# anwenden können.

4.1 Anatomie von Funktionen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=M0sK437Avyc>.

Funktionen sind Ihnen als Programmierkonstrukt bestimmt noch aus PG1 bekannt. Auch in C# gibt es Funktionen, allerdings hat man bei deren Definition noch einige Zusatzmöglichkeiten, die es in C so nicht gibt. Als erstes Beispiel einer Funktion nehmen wir die **Main** Funktion, die wir bereits in unserem ersten Programm Hello World gesehen haben:



Abbildung 4: Anatomie einer Funktion in C#

Wie in Abbildung 4 zu sehen ist, setzt sich eine Funktion in C# wie folgt zusammen: zunächst schreibt man den Funktionskopf, gefolgt von einem Paar geschweifter Klammern, die den Funktionsscope vorgeben – innerhalb dieser Klammern kann man prozeduralen Code schreiben. Der Funktionskopf selbst setzt sich dabei aus mehreren Bestandteilen zusammen, deren Reihenfolge eingehalten werden muss:

- Modifizierer statten eine Funktion mit Zusatzinformationen aus, im Normalfall mit solchen, die etwas darüber aussagen, über welche Konstrukte eine Funktion aufgerufen werden darf. Wir werden uns die verschiedenen Modifizierer im Detail später anschauen. Im oberen Beispiel sieht man die beiden Modifizierer `public` und `static` im Einsatz. Setzen sie keine Modifizierer, wird implizit vom Compiler bei Methoden der Modifizierer `private` gesetzt.
- Mit dem Rückgabetyper verdeutlicht man, welcher Typ der Wert hat, den man zurückgeben möchte. Wer in einer Funktion nichts zurückgeben möchte, nutzt als Rückgabetyper das Schlüsselwort `void`.
- Nach dem Rückgabetyper folgt der Name der Funktion. Dieser muss mit einem Buchstaben oder einem Unterstrich beginnen. Üblicherweise versucht man Ziffern in Funktionsnamen zu vermeiden. In diesen Bezeichnern sind keine Sonderzeichen wie bspw. % oder / erlaubt.
- Danach kommt die Parameterliste, die in zwei runde Klammern eingeschlossen ist. In ihr werden alle Parameter kommagetrennt aufgezählt. Ein Parameter besteht dabei aus einem

Typ gefolgt von einem Leerzeichen und dem Bezeichner des Parameters. Der Bezeichner unterliegt dabei den gleichen namentlichen Einschränkungen wie der Funktionsname. Es ist auch möglich, keine Parameter anzugeben, dann steht zwischen den zwei runden Klammern nichts. Im obigen Beispiel gibt es nur einen Parameter mit dem Bezeichner `args`, dessen Typ `string[]` ein Array von Strings ist.

Die Funktionssignatur ist das Eindeutigkeitsmerkmal einer Funktion. Sie setzt sich zusammen aus dem Namen und der Parameterliste. Innerhalb einer Klasse darf es nicht zwei oder mehr Funktionen mit derselben Signatur geben (also Funktionen mit demselben Namen und gleichen Parametern, wichtig ist hier bei den Parametern der Typ und die Reihenfolge).

In Abbildung 5 sehen Sie eine weitere Funktion `Addiere`, bei der wie oben bei der Funktion `Main` die unterschiedlichen Teile der Funktion farblich markiert sind. Beachten Sie, dass diese Funktion mehrere Parameter entgegen nimmt, die mit Komma getrennt sind. Weiterhin besitzt diese Funktion nicht den Modifizierer `static`. Als Rückgabetyp wurde hier nicht `void` (wie oben), sondern `int` angegeben, weswegen der Code mit dem Schlüsselwort `return` einen Wert zurückgeben muss, der zu genau diesem Typ passt. In dieser Funktion sehen Sie auch bereits einen alten Bekannten aus der Programmiersprache C: den Plusoperator, der die beiden Parameter `zahl1` und `zahl2` addiert.

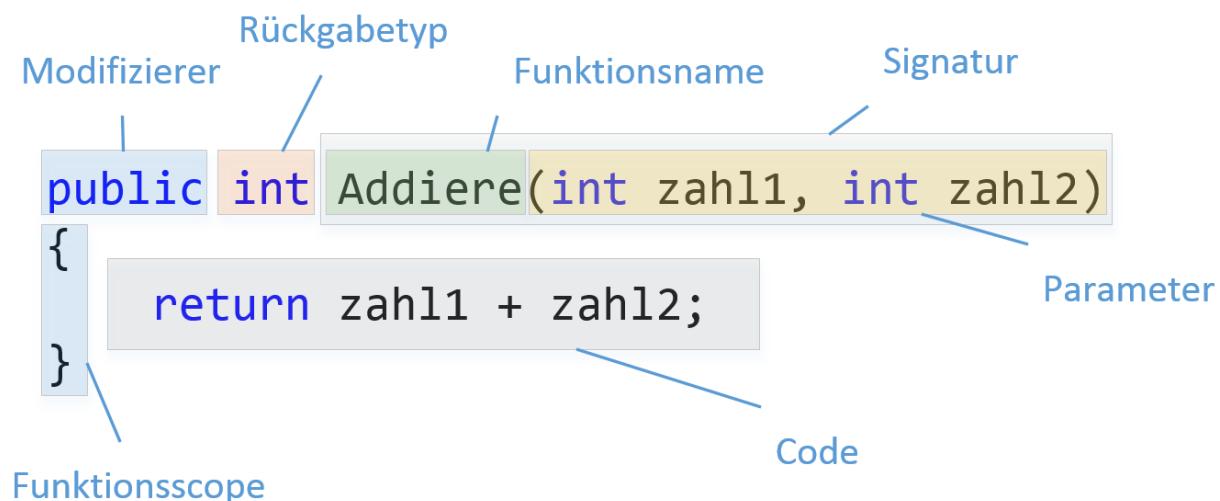


Abbildung 5: Ein weiteres Beispiel einer Funktion in C#

Ein wichtiger Unterschied besteht noch zwischen C und C#: Funktionen können nicht einfach frei in einer Codedatei aufgeführt werden, sondern müssen, wie schon im Hello World Beispiel erwähnt, innerhalb einer Klasse deklariert sein. Funktionen werden weiterhin in objektorientierten Sprachen sehr häufig als **Methoden** bezeichnet, weswegen wir diesen Begriff ab jetzt auch vorrangig benutzen.

4.2 Die Main Methode

Das Video zu diesem Abschnitt findet man unter https://www.youtube.com/watch?v=GCIXEHJ_wKY.

Die Methode `Main` bildet wie in der Programmiersprache C auch in C# den Einstiegspunkt in ein Programm. Sie wird automatisch aufgerufen, wenn das Programm über die *.exe Datei gestartet wird. Dabei gibt es aber einige Unterschiede, die beachtet werden müssen:

- Die Funktion muss genau `Main` heißen. C# beachtet Groß- / Kleinschreibung (ist damit also case-sensitive), weswegen man bspw. nicht `main` als Methodenbezeichner wählen darf.

- Es darf nur eine einzige Main Methode geben. In welcher Klasse diese definiert ist, ist jedoch egal. Bei Konsolenprojekten wird standardmäßig die Klasse **Program** erstellt, in der die Main Methode zu finden ist.
- Sie muss mit dem Modifizierer **static** gekennzeichnet sein.
- Als Rückgabetypen kann man entweder **void** oder **int** verwenden. Üblicherweise wird **void** verwendet, da **int** als Programmiererlekt gilt. Über diesen Wert hatte man früher Statuscodes an das Betriebssystem übermittelt. Ein Wert ungleich von 0 bedeutete im Normalfall, dass ein Fehler beim Ausführen des Programms aufgetreten ist. Diese Funktionalität wird aber heutzutage kaum noch eingesetzt und kann deshalb in C# auch ignoriert werden.
- Weiterhin optional ist die Angabe des Parameters **string[] args**: über diesen kann man Argumente beim Aufruf des Programms übergeben lassen, die dann in diesem Array zur Verfügung stehen. Dies wird je nach Anwendung auch heutzutage noch benutzt, in unseren Beispielen wird man diese Funktionalität aber kaum brauchen.

In Abbildung 6 sehen Sie die verschiedenen Möglichkeiten zur Definition der Main Methode, wobei die zweite Variante die von Visual Studio generierte ist, wenn ein neues Konsolenprojekt erstellt wird.

```
static void Main()
{
    /* Code hier schreiben */
}

static void Main(string[] args)
{
    /* Code hier schreiben */
}

static int Main()
{
    /* Code hier schreiben */
    return 0;
}

static int Main(string[] args)
{
    /* Code hier schreiben */
    return 0;
}
```

Abbildung 6: Mögliche Varianten der Deklarierung der Main-Methode

4.3 Imperativen Code in Funktionen schreiben

Nachdem wir uns in den letzten beiden Abschnitten mit dem Äußeren von Methoden beschäftigt haben, schauen wir uns jetzt an, wie man Code in Funktionen schreibt und welche Möglichkeiten hier bestehen.

4.3.1 Variablendefinition und Wertzuweisung

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=yx9m4nO2BKA>.

Innerhalb von Funktionen lassen sich beliebig viele Variablen erstellen. Jede Variable innerhalb des jeweiligen Funktionsscopes muss dabei einen eindeutigen Namen haben. In Abbildung 7 sehen Sie dazu ein Beispiel:

```
public int GibFünf()
{
    int a; — Variablendefinition
    a = 3; — Wertzuweisung
    int b = 2;— Definition und Zuweisung
    return a + b;
}
```

Abbildung 7: Beispiel von Variablen-deklarationen in einer Funktion

In der ersten Zeile der Funktion `GibFünf` wird eine Variable namens `a` vom Typ `int` definiert. Variablen-deklarationen werden in C# allgemein nach dem Schema „`Typ variablenName`“ erstellt. Bei einer Variablen-deklaration wird zur Laufzeit im Arbeitsspeicher genau so viel Platz allokiert, wie der angegebene Typ Speicher braucht (im Beispiel hier also 32 Bit für einen `int`). Variablen landen in C# grundsätzlich auf dem sog. Stack. Wie der Arbeitsspeicher logisch in Stack und Heap aufgeteilt ist und warum das gemacht wird, sehen wir uns in einem späteren Kapitel an.

In der nächsten Zeile wird der Variable `a` der Wert drei zugewiesen. Dazu wird in einer Anweisung das Literal „3“ mit dem Ist-Gleich-Operator `=` der Variablen zugewiesen (die Operatorauswertung funktioniert hier genau wie in C). Zur Laufzeit passiert nichts anderes, als das in den Speicherbereich, der für `a` allokiert wurde, die Bitrepräsentation des Ganzahlwerts „3“ geschrieben wird. Variablen haben deshalb auch ihren Namen: während des Funktionsablaufs können ihnen unterschiedliche Werte zugewiesen werden.

In der dritten Zeile der Funktion wird das, was wir in den beiden Zeilen vorher gemacht haben, innerhalb einer Anweisung ausgedrückt: die Variable `b` wird definiert und sofort mit dem Literal „2“ initialisiert. Das ist auch die übliche Schreibweise, wenn man Variablen definieren und ihnen sofort einen Initialwert zuweisen möchte. Am Ende der Funktion werden die Werte der beiden Variablen `a` und `b` addiert und zurückgegeben.

Im Gegensatz zu C müssen in C# nicht alle Variablen zu Beginn einer Methode definiert werden. Wie im obigen Beispiel zu sehen wird Variable `b` erst definiert, nachdem man schon mit `a = 3;` eine Anweisung in der Zeile darüber geschrieben hat.

4.3.2 Methoden aufrufen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=DfEFc5Df9D4>.

Innerhalb von Methoden kann man auch andere Methoden aufrufen. Ebenfalls ist es auch möglich, die eigene Methode aufzurufen, was man als Rekursion bezeichnet. Im folgenden Beispiel werden in der `Main` Methode zwei Funktionen aufgerufen:

```

class Program
{
    static void Main()      Methodenaufruf
    {
        Zuweisung des int rückgabewert = GibFünf();
        Rückgabewert   Console.WriteLine(rückgabewert);
    }
    static int GibFünf()    Methodenaufruf
    {
        int a;
        a = 3;
        int b = 2;
        return a + b;
    }
}

```

Abbildung 8: Aufruf von anderen Methoden innerhalb einer Methode

In Abbildung 8 wird zunächst die Methode `GibFünf` aufgerufen. Dies geschieht, indem man den Namen der Funktion schreibt gefolgt von zwei runden Klammern, in denen die Parameter angegeben werden. Da die Funktion `GibFünf` im oberen Fall keine Parameter entgegen nimmt, bleiben diese leer. Die Funktion gibt aber stattdessen einen Ganzahlwert zurück, der über den Zuweisungsoperator in der Variable `rückgabewert` gespeichert wird. Bitte beachten Sie, dass im Vergleich zum vorherigen Beispiel in Abbildung 7 die `GibFünf` Methode mit dem Modifizierer `static` und nicht `public` ausgestattet ist, damit diese Methode problemlos aus der `Main` Methode heraus aufgerufen werden kann. Anschließend wird über den Aufruf von `Console.WriteLine` der gespeicherte Wert auf der Konsole ausgegeben. Auch dies ist wiederum ein Methodenaufruf einer statischen Funktion, allerdings liegt diese Methode nicht in derselben Klasse wie `Main`, weswegen wir den entsprechenden Klassennamen `Console` mit dem Punktoperator vor den Methodennamen schreiben müssen. Wie die verschiedenen Modifizierer genau funktionieren und was beim Punktoperator zu beachten ist, schauen wir uns in einem späteren Kapitel an.

4.3.3 Parameter in Funktionen einsetzen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=HGh1F-sPJ4E>.

Man kann wie auch in C beliebig viele Parameter zu einer Funktion hinzufügen. Beim Aufruf dieser Funktion muss man diese Parameter angeben, sonst gibt der Compiler beim Erstellvorgang einen Fehler an der betreffenden Codestelle an.

In Abbildung 9 wird in der Methode `Main` die Methode `Addiere`, die zwei Parameter jeweils vom Typ `int` erwartet, insgesamt dreimal aufgerufen. Dabei werden die Parameter auf unterschiedliche Weise angegeben: beim ersten Aufruf werden die Literale für Ganzahlwerte 3 und 5 übergeben, sodass die mit dem Rückgabewert die Variable `ersteZahl` mit dem Wert 8 initialisiert wird. Beim zweiten Aufruf wird dieselbe Variable und das Literal 2 übergeben und der Rückgabewert von 10 in der Variablen `zweiteZahl` gespeichert. Beim dritten Aufruf werden die beiden zuvor erstellten Variablen übergeben, sodass danach der Wert 18 der Variablen `dritteZahl` zugewiesen wird.

```

class Program
{
    static void Main()
    {
        int ersteZahl = Addiere(3, 5);
        int zweiteZahl = Addiere(ersteZahl, 2);
        int dritteZahl = Addiere(ersteZahl, zweiteZahl);

        Console.WriteLine(ersteZahl);
        Console.WriteLine(zweiteZahl);
        Console.WriteLine(dritteZahl);
    }
    static int Addiere(int ersteZahl, int zweiteZahl)
    {
        return ersteZahl + zweiteZahl;
    }
}

```

Abbildung 9: Einsatz von Parametern in Methoden

Bitte beachten Sie in diesem Beispiel, dass die Variablen `ersteZahl` und `zweiteZahl` in der `Main` Methode unabhängig sind von den Parametern der Methode `Addiere`, auch wenn sie jeweils denselben Namen haben. Es gilt folgender wichtiger Grundsatz:

Variablen und Parameter sind nur innerhalb ihres jeweiligen Funktionsscopes gültig. Es ist dabei unerheblich, ob in anderen Methoden Variablen oder Parameter existieren, die denselben Namen haben und vom selben Typ sind.

Letztendlich kann man in C# Parameter wie Variablen innerhalb einer Funktion betrachten, denen der Aufrufer der Funktion einen Wert zuweisen muss. Außerdem sollte man versuchen, die Anzahl von Parametern in Funktionen so gering wie möglich zu halten: mehr als drei Parametern sollten eine Ausnahme darstellen. Dies ist meistens ein Hinweis darauf, dass eine Methode zu viel macht – überlegen Sie, ob es nicht sinnvoller ist, die Methode in mehrere Methoden aufzuteilen.

4.3.4 Anweisungen und Ausdrücke

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=iYkTvUnSQI>.

Der prozedurale Code in Methoden besteht im Normalfall aus mehreren Anweisungen, die hintereinander geschrieben werden, damit sie zur Laufzeit in genau dieser Reihenfolge ausgeführt werden. Einzelne Anweisungen werden dabei durch ein Semikolon abgeschlossen. Eine Anweisung besteht in C# aus einem oder mehreren Ausdrücken, die in einer gewissen Reihenfolge abgearbeitet werden. Ausdrücke können dabei wie in C beliebig geschachtelt werden.

In Abbildung 10 sind in der Methode `Main` vier Anweisungen (engl. Statements) sichtbar. Die erste Anweisung besteht dabei aus nur einem Ausdruck (engl. Expression), bei der mit dem Operator `=` der Variable `a` der Wert 42 zugewiesen wird. Die zweite Anweisung besteht aus bereits aus drei Ausdrücken:

1. Der Ausdruck `a + 3`, dessen Wert als erster Parameter für den Funktionsaufruf von `Addiere` genutzt wird.

2. Der Ausdruck `Addiere(a + 3, a)` ist ein Funktionsaufruf, dessen Rückgabewert für den Zuweisungsoperator genutzt wird.
3. Der Ausdruck `int b = Addiere(a + 3, a);` ist der oberste Ausdruck, der den Wert, den der vorherige Ausdruck berechnet hat, der Variable b zuweist. Das Semikolon am Ende bedeutet, dass die Anweisung damit abgeschlossen ist.

Zur Laufzeit wird diese Anweisung in genau derselben Reihenfolge wie gerade eben beschrieben abgearbeitet.

```
class Program
{
    static void Main()
    {
        int a = 42;                                Anweisung mit
                                                einem Ausdruck
        int b = Addiere(a + 3, a);                  Anweisung mit drei
                                                    Ausdrücken
        int c = Addiere(Addiere(a + b, a), Addiere(a, a + b));   Anweisung mit
                                                                sechs Ausdrücken
        int d = (a + Addiere(b, c)) / (a - c);      Anweisung mit fünf
                                                    Ausdrücken
    }
    static int Addiere(int ersteZahl, int zweiteZahl)
    {
        return ersteZahl + zweiteZahl;
    }
}
```

Abbildung 10: Anweisungen mit einem oder mehreren Ausdrücken

Die dritte Anweisung ist aus sechs Ausdrücken aufgebaut und letztendlich nur eine komplexere Variante der vorherigen Anweisung. Insgesamt wird dreimal Addiere aufgerufen, wobei die Rückgabewerte der ersten beiden Aufrufe die Parameter für den dritten Aufruf bereitstellen. Bei der vierten Anweisung sieht man, dass man genau wie in C Klammern nutzen kann, um Ausdrücke anders zu priorisieren, sodass deren Abarbeitungsreihenfolge geändert wird:

1. Zunächst wird der Ausdruck `Addiere(b, c)` ausgewertet.
2. Danach wird der Ausdruck `a + Addiere(b, c)` ausgeführt, da dieser zusätzlich geklammert ist. Ansonsten würde der Ausdruck des Operators `/` Vorrang bekommen.
3. Im Anschluss wird `a - c` ausgewertet, ebenfalls weil dieser Ausdruck geklammert ist.
4. Danach wird der Divisionsausdruck `(a + Addiere(b, c)) / (a - c)` ausgewertet.
5. Zum Schluss wird das Ergebnis im letzten Ausdruck der Variable `d` zugewiesen.

Wie man bereits sieht, wird die der Code deutlich schlechter lesbar, wenn man viele Ausdrücke in einer einzigen Anweisung verwendet. Die Lesbarkeit ist aber wichtig für die Codequalität, weswegen ich Ihnen empfehle, sich wirklich Gedanken zu machen, ob es notwendig ist, ein Statement mit mehr als drei Ausdrücken zu schreiben.

4.3.5 Primitive Datentypen

Das Video zu diesem Abschnitt finden Sie unter https://www.youtube.com/watch?v=g_w1hKzAL2Y.

C# / .NET bringt einige primitive Datentypen mit. Alle diese Datentypen besitzen ein C# Schlüsselwort, mit dem sie ebenfalls angesprochen werden können. In der folgenden Tabelle sind alle primitiven Datentypen aufgelistet:

C# Alias	.NET Datentyp	Wertebereiche
<code>int</code>	<code>System.Int32</code>	-2^{31} bis $2^{31} - 1$ (Ganzzahl)

<code>double</code>	<code>System.Double</code>	$5,0 \cdot 10^{-324}$ bis $1,7 \cdot 10^{308}$ (Gleitkommazahl)
<code>bool</code>	<code>System.Boolean</code>	<code>true</code> oder <code>false</code>
<code>string</code>	<code>System.String</code>	2^{31} Unicodezeichen (Referenztyp)
<code>char</code>	<code>System.Char</code>	Unicodezeichen zwischen 0 und 65535
<code>long</code>	<code>System.Int64</code>	-2^{63} bis $2^{63} - 1$ (Ganzzahl)
<code>short</code>	<code>System.Int16</code>	-2^{15} bis $2^{15} - 1$ (Ganzzahl)
<code>uint</code>	<code>System.UInt32</code>	0 bis $2^{32} - 1$ (Ganzzahl)
<code>ulong</code>	<code>System.UInt64</code>	0 bis $2^{64} - 1$ (Ganzzahl)
<code>ushort</code>	<code>System.UInt16</code>	0 bis $2^{16} - 1$ (Ganzzahl)
<code>float</code>	<code>System.Single</code>	$1,4 \cdot 10^{-45}$ bis $3,4 \cdot 10^{38}$ (Gleitkommazahl)
<code>decimal</code>	<code>System.Decimal</code>	$\frac{-7,9 \cdot 10^{28} \text{ bis } 7,9 \cdot 10^{28}}{10^0 \text{ bis } 28}$ (Gleitkommazahl)
<code>byte</code>	<code>System.Byte</code>	0 bis 255
<code>sbyte</code>	<code>System.SByte</code>	-128 bis 127
<code>object</code>	<code>System.Object</code>	Universell (Referenztyp)

Abbildung 11: Die primitiven Datentypen von C#

In C# wird zwischen sogenannten Wertetypen und Referenztypen unterschieden. Wertetypen sind Strukturen, deren Werte direkt im allokierten Speicherbereich gehalten werden: erstellt man bspw. eine Variable vom Typ `int`, werden auf dem Stack 32 Bits (4 Bytes) für diese reserviert. Wird dieser Variable ein Wert zugewiesen, wird er direkt in diesen allokierten Speicher geschrieben. Bei Referenztypen verhält sich das anders: wird bspw. eine Variable vom Typ `object` erstellt, dann werden für diese Variable ebenfalls 32 Bit allokiert, allerdings wird ihr bei Wertzuweisung eine Adresse eines Objekts zugewiesen (die Variable zeigt also auf das Objekt, weswegen man von Referenztypen spricht). Referenztypen müssen üblicherweise mit dem `new` Operator instanziiert werden. Wir werden mehr zum Thema Speicherverwaltung und den Unterschied zwischen Referenz- und Wertetypen in einem späteren Kapitel lernen.

Alle primitiven Datentypen sind Wertetypen außer `object` und `string`, diese sind Referenztypen. Obwohl es möglich ist, diese Typen über ihren .NET Namen anzusprechen (zu sehen in der zweiten Spalte oben), schreibt man im Normalfall das entsprechende Schlüsselwort in C#. Die am häufigsten benutzten primitiven Datentypen sind die obersten fünf der Tabelle: `int` für Ganzzahlen, `double` für Gleitkommazahlen, `bool` für boolesche Werte, `string` für Zeichenketten und `char` für Buchstaben.

In den folgenden Abschnitten werden wir noch etwas genauer auf die verschiedenen Gruppen von primitiven Datentypen eingehen:

4.3.5.1 Ganzzahltypen

Die Typen `int`, `byte`, `long`, `short`, `sbyte`, `uint`, `ushort`, und `ulong` sind Ganzzahltypen (engl. Integer) mit unterschiedlichen Wertebereichen. Ganzzahlliterale sind grundsätzlich erst einmal vom Typ `int`, außer man nutzt bestimmte Suffixnotationen für das Literal. Am einfachsten lässt sich dies natürlich an einem Beispiel erklären:

```

static void Main()
{
    int integer1 = 41; ----- int Literal ohne Suffix

    uint integer2 = 41u; ----- uint Literal mit Suffix u

    ushort integer3 = 41; ----- ushort hat kein entsprechendes Literal,
                                sondern verwendet das von int

    long integer4 = 41l; ----- long Literal mit Suffix l

    ulong integer5 = 9223372036854775808ul;
}

```

ulong Literal mit Suffix ul

Abbildung 12: Literale für Ganzzahltypen

In Abbildung 12 sieht man eine `Main` Methode, in der fünf Variablen mit jeweils unterschiedlichen Ganzzahltypen erstellt und mit den entsprechenden Literalen initialisiert werden. Dabei kann man allgemein folgende Punkte beachten:

- Ganzzahlliterale ohne Suffix geben immer einen `int` Wert zurück
- Literale mit Suffix `u` werden zu `uint`
- Literale mit Suffix `l` werden zu `long`
- Literale mit `ul` wird zu `ulong`
- Alle anderen Ganzzahltypen haben keine expliziten Literale, sondern nutzen die von `int`

Neben der Kleinschreibweise für diese Suffixe kann man auch Großbuchstaben verwenden. Für alle diese Datentypen ist 0 der Initialwert, wenn bei Variablen Deklaration kein Wert zugewiesen wird. Im Programmieralltag nutzt man von diesen Typen hauptsächlich `int`, `byte` und ab und zu `long`.

4.3.5.2 Gleitkommazahltypen

Die Typen `double`, `float` und `decimal` sind die primitiven Typen, die numerische Werte im Gleitkommaformat (engl. floating point format) abspeichern. Dabei besteht ein großer Unterschied: `double` und `float` speichern ihre Werte im Binärsystem (Basis 2) ab, wohingegen `decimal` den Wert in einer anderen Darstellung im Dezimalsystem ablegt. Deswegen wird der Typ `decimal` häufig in Finanzsoftware eingesetzt, da durch die nähere Verwandtschaft mit dem uns üblicheren Rechensystem eine größere Präzision erreicht wird. Der Nachteil ist, dass dieser Typ bei den verschiedenen Rechenoperationen deutlich langsamer ist als `double` und `float`.

Auch für Gleitkommaliterale gibt es bestimmte Schreibweisen, die wir uns in der folgenden Abbildung anschauen:

```

static void Main()
{
    double gleitkommazahl1 = 42.25; —— double Literal mit PunktSchreibweise

    double gleitkommazahl2 = 42d; —— double Literal mit Suffix d

    float gleitkommazahl3 = 42.25f; —— float Literal mit Suffix f

    decimal gleitkommazahl4 = 42.25m; —— decimal Literal mit Suffix m
}

```

Abbildung 13: Literale für Gleitkommazahltypen

Ähnlich wie auch schon im vorherigen Beispiel, werden in Abbildung 13 mehrere Gleitkommavariablen mit den unterschiedlichen Typen und ihren dazugehörigen Literalen erstellt. Hier kann man zusammenfassend folgende Aussagen treffen:

- Literale mit Dezimalpunkt (nicht Komma wie in der deutschen Schreibweise) und darauffolgenden Ziffern werden als `double` zurückgegeben
- Einen `double` erhält man ebenfalls, wenn man hinter numerischen Wert das Suffix `d` setzt
- Hinter numerische Werte, die als `float` zurückgegeben werden sollen, muss das Suffix `f` stehen
- Das gleiche gilt für `decimal` Werte, allerdings nutzt man hier das Suffix `m`

Wie bei Ganzahlsuffixen ist es egal, ob man diese groß oder klein schreibt. Auch für die hier besprochen Typen ist 0 der Standardwert, wenn beim Erstellen einer Variable kein Wert zugewiesen wird. Im Programmieralltag benutzt man `double` für Gleitkommawerte, in Finanzsoftware setzt man `decimal` ein. `float` sollte man vermeiden, da seine Präzision nur bei sechs bis sieben Stellen liegt, was nicht sehr viel ist.

Eine wichtige Sache muss ich noch im Zusammenhang mit Gleitkommazahlen erwähnen: Bei Überprüfung zweier `double` oder zweier `float` Werte auf Gleichheit muss Vorsicht angewandt werden. Da diese Gleitkommatypen im Binärsystem abgespeichert werden, können sie auch nur die Bruchteile $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$ usw. akkurat darstellen. Alle anderen reellen Zahlen sind Annäherungen, keine exakten Werte. Die sieht man an folgendem Codebeispiel:

```

static void Main()
{
    double gleitkommazahl1 = 1.001 - 0.001;
    double gleitkommazahl2 = 0.999 + 0.001;

    if (gleitkommazahl1 == gleitkommazahl2)
        Console.WriteLine("Die Zahlen sind gleich");
    else
        Console.WriteLine("Die Zahlen sind verschieden");
}

```

Vergleich von zwei double Werten gibt nicht immer wahr zurück

Abbildung 14: Gleichheit von zwei Gleitkommawerten ist nicht immer gegeben

In Abbildung 14 sieht man zwei `double` Variablen, die intuitiv gesehen beide den Wert 1.0 haben sollten. Allerdings kommt es bei der Auswertung im `if` Block nicht immer zu einem wahren Ausdruck, weswegen in diesem Falle dann der `else` Block ausgeführt wird. `if` und `else` verhalten sich in C# genauso wie in C und wir werden in einem kommenden Abschnitt noch genauer darüber sprechen. Das Problem liegt daran, dass die Teilwerte 0.001 und 0.999 nicht akkurat gespeichert werden können im Binärsystem und deshalb die Berechnung mit dem - bzw. + Operator auf manchen Prozessoren (das Ganze ist tatsächlich prozessorabhängig) zu leicht unterschiedlichen Ergebnissen führen kann. Dieses Phänomen tritt übrigens mit `decimal` nicht auf, da, wie schon eben erwähnt, dieser Typ Gleitkommazahlen im Dezimalsystem abspeichert und damit auch 0.001 und 0.999 akkurat abbilden kann im Speicher.

Um dieses Dilemma zu lösen, muss man statt dem `==` Operator einen anderen Ausdruck verwenden, dem man eine gewisse Toleranz mitgibt. Im Beispiel oben könnte man bspw. folgenden Ausdruck in der `if` Abfrage verwenden:

```
if (Math.Abs(gleitkommazahl1 - gleitkommazahl2) < toleranzwert)
```

Zunächst zieht man beide `double` Werte voneinander ab und nimmt davon den Betrag. Ist dieser kleiner als ein gewisser toleranzwert (den man hier bspw. auf 0.0001 setzen könnte), dann gelten die Zahlen als gleich. Mehr zum Speicherbild von Ganzzahl- und Gleitkommazahlformaten lernen Sie in der Vorlesung Computerarithmetik und Rechenverfahren (CR).

4.3.5.3 Boolesche Werte

Der Typ `bool` ist für Werte gedacht, die entweder wahr oder falsch sein können, was durch die Literale `true` und `false` repräsentiert ist. Der Standardwert für eine erstellte Variable ist `false`. Weiterhin ist anzumerken, dass in C# numerische Werte nicht als Wahrheitswerte gelten, d.h. aus 0 wird nicht automatisch `false`, aus Werten ungleich 0 wird nicht automatisch `true`. Viel mehr gibt es zu diesem Datentyp an dieser Stelle nicht zu sagen.

4.3.5.4 Zeichenketten und Buchstaben

Mit den Typen `string` und `char` werden in C# Zeichenketten (engl. Strings) bzw. Buchstaben (engl. Character) abgebildet. Dabei ist zu beachten, dass `char` ein Wertetyp ist, `string` jedoch ein Referenztyp. Erklären kann man sich das ganz einfach so: ein `char` Wert ist letztendlich nichts anderes als ein Ganzahlwert, der über die Unicode Umformungstabelle als (Schrift-)Zeichen interpretiert wird. Ein `string` kann man als ein Array von `char` Werten sehen, die hintereinander stehen und für uns als Nutzer lesbar sind.

Auch für diese Typen gibt es unterschiedliche Literalschreibweisen, die man im folgenden Beispiel sehen kann:

```
static void Main()
{
    char buchstabe = 'a'; —— char Literal mit einfachen Anführungszeichen

    string zeichenkette1 = "A"; —— string Literal mit nur einem Buchstaben

    string zeichenkette2 = "Das ist aber schönes Wetter heute.";
}
```

—————
string Literal mit doppelten Anführungszeichen

Abbildung 15: Char und String Literale

Zu beachten ist in Abbildung 15 nur, dass `char` Literale einfache Anführungszeichen nutzt. Alles, was zwischen zwei Anführungszeichen steht, wird automatisch ein `string` (auch wenn dieser String nur

einen einzigen Buchstaben enthält, wie in der zweiten Anweisung zu sehen). Der Standardwert für `char` ist `\0`, also der Nullcharakter, der in C / C++ dem Abschluss eines Zeichenketten andeutete. Dieses `\0` entspricht dem Ganzahlwert 0. Der Standardwert für `string` ist, wie bei allen Referenztypen, `null`. Im Programmieralltag nutzt man hauptsächlich `string`, `char` wird nur in seltenen Fällen direkt genutzt.

4.3.5.5 Referenztypen

Die primitiven Datentypen `object` und `string` sind beide Referenztypen, wie wir oben schon erwähnt haben. Referenztypen muss man üblicherweise mit `new` initialisieren, der Typ `string` bietet hieron die einzige Ausnahme, ihn kann man ja, wie im vorherigen Abschnitt beschrieben, auch über ein Literal erzeugen. In Abschnitt 4.3.5 Primitive Datentypen habe ich bereits erwähnt, dass Referenztypen und Wertetypen unterschiedlich behandelt werden. Dazu schauen wir uns nochmals ein Beispiel an:

```
static void Main()
{
    object objekt = new object(); — Neue Instanz von object erstellen

    objekt = null; — Nullreferenz Literal für Referenztypen
}
```

Abbildung 16: new Operator und null Literal für Referenztypen

In Abbildung 16 wird in der `Main` Methode zunächst ein neues Objekt erstellt mit dem `new` Operator. Das geht folgendermaßen: zunächst wird im sog. Heap (engl. Free Store oder ebenfalls Heap, auf Deutsch heißt Heap Haufen) das entsprechende Objekt initialisiert und dessen Speicheradresse zurückgegeben. Diese wird in der Variablen `objekt` gespeichert, sodass man über diese Referenz auf das Objekt zugreifen kann. Damit ist die Variable `objekt` funktional gesehen sehr ähnlich wie ein Zeiger in C. Möchte man jedoch, dass diese Referenz zerstört wird, d.h. `objekt` soll nicht mehr auf das zuvor zugewiesene Objekt im Speicher verweisen, dann nutzt man das `null` Literal. Durch diese Zuweisung sagt man aus, dass `objekt` auf nichts verweist. Und das ist ein triftiger Unterschied zwischen Referenz- und Wertetypen: ersteren kann `null` zugewiesen werden, letzteren jedoch nicht.

Bitte beachten Sie auch, dass nach dem `new` Operator der Typname samt einem Paar runde Klammern aufgeführt ist (und das Ganze ist tatsächlich ein Funktionsaufruf). Was hinter dieser Anweisung tatsächlich steckt, schauen wir uns genauer beim Thema Klassen an.

4.3.6 Typumwandlung und –Konvertierungen

C# ist eine sog. stark-typisierte Programmiersprache. Das bedeutet, dass man einer Variablen eines bestimmten Typs nicht ohne weiteres einen Wert eines anderen Typs zuweisen darf. Im folgenden Beispiel ist das zu sehen:

```

static void Main()
{
    int ganzzahl;
    ganzzahl = "Hello World!";
}

```

Zuweisung von Stringwert an int Variable nicht möglich

Abbildung 17: Starke Typisierung – Zuweisung an Variable mit unterschiedlichen Typ nicht möglich

Dennoch gibt es die Möglichkeit, Werte eines Typs in denselben Wert eines anderen Typs umzuwandeln. Diese Konvertierungen sind in vier Gruppen eingeteilt:

- Implizite Konvertierung (engl. implicit conversion)
- Explizite Konvertierung (engl. cast oder explicit conversion)
- Konvertierung mit Helfer-Klasse
- Nutzerdefinierte Konvertierung (engl. user-defined conversion)

Die ersten drei dieser Konvertierungen schauen wir uns jetzt im Detail an. Die letzte Konvertierung betrachten wir genauer, wenn wir mehr Wissen über Klassen und Strukturen in C# haben, da dazu sog. Operatorüberladungen notwendig sind.

4.3.6.1 Implizite Konvertierungen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=CKjGAQaqgIU>.

Für implizite Konvertierungen ist keine besondere Syntax notwendig. Sie wird automatisch vom Compiler eingesetzt, wenn ein Wert einwandfrei zum Zieltyp zugewiesen werden kann, ohne dass der Wert dabei abgeschnitten oder gerundet werden muss. Dies ist v.a. bei numerischen Typen der Fall. Dies können wir uns an einem Beispiel verdeutlichen:

```

static void Main()
{
    int i = 103;
    long l = i;      Implizite Konvertierung von  
int zu long
}

```

Abbildung 18: Implizite Konvertierung von int zu long

In Abbildung 18 sieht man eine implizite Konvertierung von **int** zu **long**. Diese ist möglich, da **int** ein 32 Bit und **long** ein 64 Bit Ganzahltyp ist. **long** beinhaltet damit den gesamten Wertebereich von **int**, weshalb die Konvertierung ohne Probleme durchgeführt werden kann.

Implizite Konvertierung kann nicht nur beim Zuweisungsoperator passieren, sondern bspw. auch beim Funktionsaufruf:

```

static void Main()
{
    float f = 103;
    double quadrat = BerechneQuadrat(f);
}

static double BerechneQuadrat(double zahl)
{
    return zahl * zahl;
}

```

Abbildung 19: Implizite Konvertierung von float zu double

In Abbildung 19 wird die Variable `f` beim Funktionsaufruf `BerechneQuadrat` als Parameter angeben. Bevor die Methode tatsächlich ausgeführt wird, wird eine implizite Konvertierung von `float` zu `double` beim Parameter durchgeführt. Auch dies ist wiederum möglich, weil `double` den gesamten Wertebereich von `float` abdeckt (`float` besitzt 32 Bit, `double` 64 Bit).

In der folgenden Tabelle finden Sie die möglichen impliziten Konvertierungen zwischen den verschiedenen numerischen Datentypen. Eine komplette Übersicht dazu finden Sie im [MSDN](#). Es ist nicht notwendig, diese Tabelle auswendig zu kennen, interessant ist aber folgendes: es gibt keine impliziten Konvertierungen von `string`, `bool`, `double` und `decimal` hin zu anderen Typen.

Von	Nach
<code>byte</code>	<code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> oder <code>decimal</code>
<code>int</code>	<code>long</code> , <code>float</code> , <code>double</code> oder <code>decimal</code>
<code>long</code>	<code>float</code> , <code>double</code> oder <code>decimal</code>
<code>char</code>	<code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> oder <code>decimal</code>
<code>float</code>	<code>double</code>

Abbildung 20: Mögliche Implizite Typkonvertierungen

4.3.6.2 Explizite Konvertierungen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=raz38qkXUJE>.

Wenn eine implizite Konvertierung nicht möglich ist, kann man dennoch zwischen bestimmten Typen konvertieren, indem man einen sog. Cast (explizite Konvertierung) durchführt. Diese sind im Normalfall dann notwendig, wenn eine implizite Konvertierung nicht möglich ist. Mit einem Cast macht man dem Compiler deutlich, dass man sich beim Konvertieren eines möglichen Datenverlusts oder Rundungsfehlern bewusst ist. Auch können Exceptions bei solchen Casts auftreten.

```

static void Main()
{
    double gleitkommazahl = 42.7;
    int ganzzahl = (int)gleitkommazahl;
}

Expliziter Cast von
double zu int

```

Abbildung 21: Expliziter Cast von double zu int

In Abbildung 21 wird der Wert der Variable `gleitkommazahl` per Cast zu einem `int` Wert konvertiert und der Variablen `ganzzahl` zugewiesen. Der Cast wird dabei mit zwei runden Klammern, in denen der Zieltyp angegeben wird, eingeleitet. Hinter den Klammern kann ein beliebiger Ausdruck stehen, der gecastet werden soll. Wie im Beispiel oben zu sehen, wird bei Casts von Gleitkommazahltypen zu Ganzahltypen sämtliche Nachkommastellen abgeschnitten, es wird nicht gerundet. `ganzzahl` hat deswegen am Ende den Wert 42.

Genau wie bei impliziten Konvertierungen sind explizite Konvertierungen nur zwischen bestimmten primitiven Datentypen möglich. Für die wichtigsten Datentypen ist dies in der folgenden Tabelle aufgelistet. Eine komplette Übersicht finden Sie im [MSDN](#).

Von	Nach
<code>byte</code>	<code>sbyte</code> oder <code>char</code>
<code>int</code>	<code>sbyte, byte, short, ushort, uint, ulong</code> oder <code>char</code>
<code>long</code>	<code>sbyte, byte, short, ushort, int, uint, ulong</code> oder <code>char</code>
<code>char</code>	<code>sbyte, byte</code> oder <code>short</code>
<code>float</code>	<code>sbyte, byte, short, ushort, int, uint, long, ulong, char</code> oder <code>decimal</code>
<code>double</code>	<code>sbyte, byte, short, ushort, int, uint, long, ulong, char, float</code> oder <code>decimal</code>

Abbildung 22: Mögliche explizite Typkonvertierungen

4.3.6.3 Konvertierungen mit Helfer-Klassen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=VxeLk1qKnSw>.

Wenn sowohl implizite als auch explizite Konvertierung nicht möglich ist, dann gibt es standardmäßig noch eine dritte Möglichkeit, einen Wert in einen anderen Typ zu konvertieren: die Klasse `System.Convert`. Sie bietet mehrere statische Methoden an, mit denen Konvertierungen durchgeführt werden können.

```
static void Main()
{
    string zahlenstring = „2“;
    int zahl = Convert.ToInt32(zahlenstring);
}
```

Funktionsaufruf zur Konvertierung von
string zu int

Abbildung 23: Konvertierung mit der Klasse `Convert` von `string` zu `int`

In Abbildung 23 wird die Methode `Convert.ToInt32` genutzt, um eine Zeichenkette zu einer Ganzzahl zu konvertieren. Die Funktionen der Klasse `Convert` sind überladen, d.h. es gibt mehrere Methoden, die den gleichen Namen haben, aber eine unterschiedliche Parameterliste. Somit gibt es nicht nur eine `ToInt32` Methode, die einen `string` Parameter entgegen nimmt, sondern auch noch weitere, die andere Typen wie bspw. `decimal` konvertieren. Eine Gesamtübersicht der Klasse `Convert` findet man im [MSDN](#).

4.3.7 Arrays

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=UQzd-xQjI0A>.

Zu jedem beliebigen Typ lässt sich ein Array erstellen. Arrays verwalten wie auch in C mehrere Objekte bzw. Strukturen eines Typs. Dabei ist die Syntax etwas anders als in C, was sich am besten an einem Beispiel verdeutlichen lässt.

```

        static void Main()
    {
        int[] ganzzahlArray = new int[3];
        ganzzahlArray[0] = 42;
        ganzzahlArray[1] = 17;
        ganzzahlArray[2] = 23;

        Console.WriteLine("Größe des Arrays: ");
        Console.WriteLine(ganzzahlArray.Length); - Gibt Größe des Array zurück
        Console.WriteLine("Die Einträge sind:");
        Console.WriteLine(ganzzahlArray[0]);
        Console.WriteLine(ganzzahlArray[1]); - Auslesen der Elemente im
        Console.WriteLine(ganzzahlArray[2]);
    }

```

Abbildung 24: Einsatz eines Arrays

In Abbildung 24 sieht man eine `Main` Methode, in der in der ersten Zeile ein Array für `int` Werte initialisiert wird. Wenn man hinter den Variablen Typen leere eckige Klammern wie bei `int[]` setzt, so interpretiert der Compiler dies nicht als `int` Variable, sondern als Array von `int`-Werten. Diese Variable wird auf der rechten Seite des Zuweisungsoperators mit dem Ausdruck `new int[3]` initialisiert. Der `new` Operator erstellt dabei ein neues Objekt, `int[3]` gibt an, dass es sich um einen `int` Array mit drei Stellen handelt. Arrays sind immer Referenztypen, auch wenn der verwaltete Typ ein Wertetyp ist, weswegen auch der `new` Operator zur Initialisierung eingesetzt wird.

In den folgenden drei Anweisungen werden dem Array an seine drei möglichen Stellen die Werte 42, 17 und 23 zugewiesen. Dies geschieht mit dem Indexoperator `[]`, der hinter dem Variablen Namen `ganzzahlArray` angewendet wird. Arrays sind wie in C nullbasiert, d.h. um auf das n -te Element im Array zuzugreifen, muss man als Indexwert $n - 1$ verwenden. Im Beispiel oben wird deshalb auch das erste Element mit `ganzzahlArray[0]`, das zweite mit `ganzzahlArray[1]` usw. gesetzt.

In den restlichen Statements wird mit `Console.WriteLine` Befehlen Ausgaben auf die Konsole gemacht. Die erste Ausgabe ist die Größe des Arrays: diese kann man über `ganzzahlArray.Length` abfragen. Anders als in C weiß ein Array in C#, wie viele Elemente er umfassen kann, sodass dieser Wert nicht in einer gesonderten Variable mitgeführt werden muss, sondern über eben genannten Ausdruck abgefragt werden kann. Danach wird auf die einzelnen Elemente des Arrays nullbasiert mit dem Indexoperator zugegriffen, um sie auszugeben. Bitte beachten Sie, dass man für die Ausgabe aller Elemente eines Arrays eigentlich Schleifen verwenden sollte (diese lernen wir im nächsten Abschnitt kennen).

In den ersten vier Anweisungen des gerade besprochenen Beispiels haben wir einen Array initialisiert und die einzelnen Stellen mit Werten belegt. Dafür gibt es in C# auch noch kürzere Schreibweisen:

- Mit der Anweisung `int[] ganzzahlArray = new int[] { 42, 17, 23 };` lässt sich dasselbe Anstellen. Beachten Sie die Änderungen der Schreibweise bei der Initialisierung mit dem `new` Operator: die eckigen Klammern bleiben leer, stattdessen gibt man in zwei weiteren geschweiften Klammern die Elemente kommagetrennt an, die der Array nach der Initialisierung enthalten soll. Der Compiler leitet die Größe des Arrays aus der Anzahl dieser Elemente ab.
- Es gibt sogar eine noch kürzere Schreibweise: wenn in den geschweiften Klammern alle Elemente denselben konkreten Typ haben, dann kann man nach dem `new` Operator sogar

```
den Arraytyp weglassen: int[] ganzzahlArray = new [] { 42, 17, 23 };
```

Der Compiler leitet den Typen des Arrays in diesem Fall vom Typ der Elemente ab.

Die eben besprochene Syntax heißt Objekt-Initialisierungssyntax.

Auch wenn wir jetzt sehr viel über Arrays gesprochen haben, so werden sie im objektorientierten Programmieralltag doch eher selten eingesetzt: das Hauptproblem ist, dass sie nach der Initialisierung eine feste Größe haben und durch einen neuen Array ersetzt werden müssen, wenn diese Größe überschritten wird. Genau dieses Problem lösen Collections: diese sind für den Nutzer genauso komfortabel einzusetzen wie Arrays, man muss sich aber nicht über deren Größenlimit Gedanken machen. Collections lernen wir in einem späteren Kapitel kennen.

4.3.8 Kontrollflussstrukturen

Wie in C gibt es auch in C# Kontrollflussstrukturen (engl. Flow Control), mit denen man den Ablauf einer Methode steuern kann. Dabei werden grundsätzlich folgende Gruppen unterschieden:

- Abfrageblöcke: zu ihnen gehören **if-else** Blöcke bzw. **switch** Statements. Sie prüfen einen booleschen Ausdruck oder einen Wert und verzweigen anhand dieser.
- Schleifen: zu ihnen gehören die **while**, **for** und **foreach** Schleife. Sie wiederholen einen bestimmten Codeblock, bis die Schleife beendet wird.
- Exceptions (dt. Ausnahmen): sie werden genutzt um fehlerhafte Zustände im Programm anzuzeigen und das Programm zum Absturz zu bringen, wenn die Exception nicht behandelt wird. Ein fehlerhafter Zustand kann bspw. das Aufrufen einer Methode mit einem falschen Parameterwert sein.

Exceptions schauen wir uns in einem späteren Kapitel an, da wir dazu Grundwissen über Vererbung brauchen. Abfragen und Schleifen werden in den kommenden Abschnitten beleuchtet.

4.3.8.1 If – Else Blöcke

Das Video zu diesem Abschnitt finden Sie unter https://www.youtube.com/watch?v=PA_UJmRe_aM.

Wenn man bestimmte Teile des Codes nur unter bestimmten Bedingungen ausführen möchte, so bieten sich **if** Blöcke an. **if** Blöcke funktionieren im Prinzip genau gleich wie in der Programmiersprache C. Schauen wir uns dazu folgendes Beispiel an:

```
public void GibZahlAusUndPrüfeObGerade(int zahl)
{
    Console.WriteLine(zahl);

    if Schlüsselwort ————— if (rest == 0) ————— Boolescher Ausdruck
        {
            if Block ————— Console.WriteLine(„Die Zahl ist gerade.“);
        }
}
```

Abbildung 25: if Block für Verzweigung unter bestimmten Bedingungen

In Abbildung 25 sieht man die Methode **GibZahlAusUndPrüfeObGerade**. Der übergebene Parameter **zahl** wird zunächst auf der Konsole ausgegeben. Im Anschluss wird mit **zahl % 2** der Rest von einer Division durch zwei berechnet. Mit diesem Rest lässt sich feststellen, ob die Zahl

gerade oder ungerade ist (bei einer `% 2` Operation kommt entweder 0 oder 1 raus). Genau dies wird im nächsten Codeteil überprüft: mit dem Schlüsselwort `if` wird eine Abfrage eingeleitet. Der dazugehörige boolesche Ausdruck `rest == 0` muss dabei in den runden Klammern stehen, die dem Schlüsselwort `if` folgen. Wenn der boolesche Ausdruck wahr ist, wird der Code im `if` Block ausgeführt. Falls nicht, wird dieser einfach übersprungen und die nächste Anweisung ausgeführt. Zu beachten ist, dass bei `if` Abfragen in C# im Gegensatz zu C immer ein boolescher Wert verwendet werden muss, d.h. der Ausdruck zwischen den Klammern muss einen Wert vom Typ `bool` zurückgeben.

Wenn man anhand einer Bedingung in unterschiedliche Codeteile verzweigen möchte, dann bieten sich `if-else` Blöcke an. Dabei wird direkt nach einer `if` Abfrage ein dazugehöriger `else` Block angehangen, wie im folgenden Beispiel zu sehen:

```
public void PrüfeObZahlGeradeOderUngeradeUndGibErgebnisAus(int zahl)
{
    int rest = zahl % 2;
    if Schlüsselwort if (rest == 0) Boolescher Ausdruck
    {
        if Block Console.WriteLine("Die Zahl ist gerade.");
    }
    else Schlüsselwort else
    {
        else Block Console.WriteLine("Die Zahl ist ungerade");
    }
}
```

Abbildung 26: `if-else` in Kombination

In Abbildung 26 sieht man eine Funktion, die etwas anders strukturiert ist als die aus dem vorherigen Beispiel. Auch in ihr wird zunächst der Rest berechnet. Anhand diesem wird in der `if` Abfrage überprüft, ob die Zahl gerade war. Trifft dies zu, wird der Code im `if` Block ausgeführt und der `else` Block übersprungen. Wenn der Wert der Variablen `rest` jedoch nicht 0 ist, wird der Code im `else` Block ausgeführt. `if-else` verwendet man also für Verzweigungen nach dem Konzept Entweder-Oder.

Dieses Konzept kann man noch etwas komplexer aufbauen, indem man mehrere `if-else` Blöcke hintereinander reiht. Dies wird häufig gemacht, wenn man von einem Datentyp mehrere valide Ausdrücke hat, nach denen man in bestimmte Codeteile verzweigt. Sehen Sie sich dazu folgendes Beispiel an:

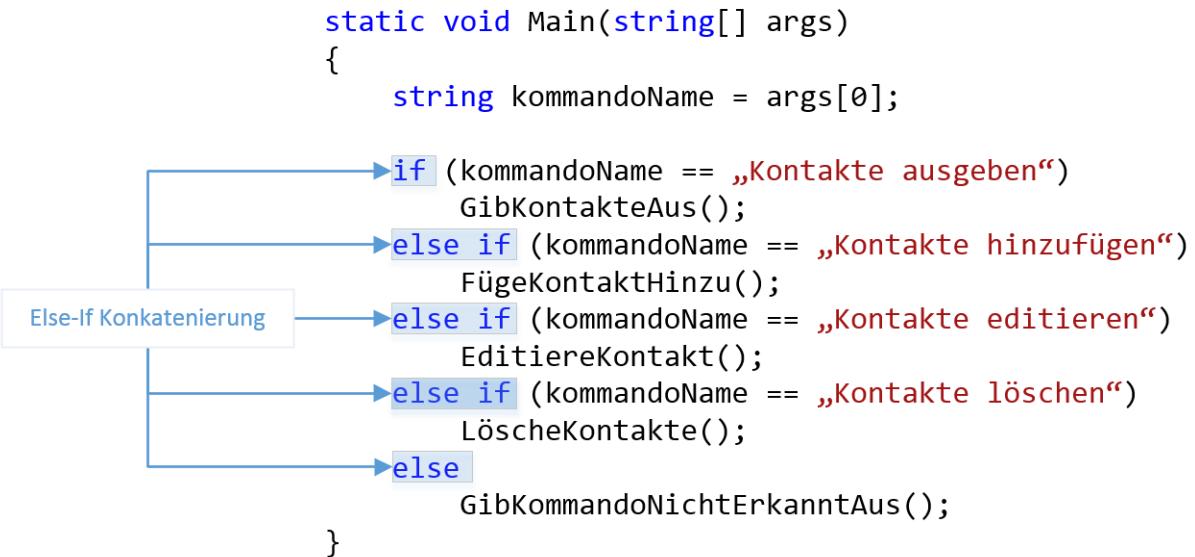


Abbildung 27: Else-If Aneinanderreihung

In Abbildung 27 sehen Sie eine Main Methode, in der der Array, der die Startparameter für das Konsolenprogramm enthält, ausgewertet wird. Dazu wird der erste Eintrag des Arrays in der Variable kommandoName gespeichert. Anhand des Wertes in dieser Variable wird mit einer Aneinanderreihung von **if-else** Blöcken zu verschiedenen Funktionen verzweigt. Sobald der Kommandoname mit einem der vier vorgegeben Werte „Kontakte ausgeben“, „Kontakte hinzufügen“, „Kontakte editieren“ oder „Kontakte löschen“ übereinstimmt, wird der entsprechende Block ausgeführt und danach das gesamte Konstrukt verlassen. Stimmt der Wert von kommandoName nicht mit einem dieser Stringwerte überein, dann wird der Code im letzten **else** Block ausgeführt, der die Funktion GibKommandoNichtErkanntAus aufruft. In diesem Beispiel sehen sie ebenfalls, dass man die geschweiften Klammern eines **if** Blocks weglassen kann, wenn sich innerhalb dieses Blocks nur eine einzige Anweisung befindet.

Für genau diesen Einsatzzweck gibt es jedoch auch das Schlüsselwort **switch**, das im nächsten Abschnitt erklärt wird.

4.3.8.2 Switch Anweisungen

Das Video zu diesem Abschnitt findet man unter <https://www.youtube.com/watch?v=DwdEA7jrRKg>.

Switch Anweisungen sind eine andere Möglichkeit, eine **if-else** Aneinanderreihung auszudrücken. In Abbildung 28 wird genauso wie im vorherigen Beispiel aus den Konsolenparametern der Kommandoname einer Variablen zugewiesen. Danach wird eine **switch** Verzweigung auf diese Variable ausgeführt. Dazu schreibt man das Schlüsselwort **switch** gefolgt von einem Paar runde Klammern, in denen man einen Ausdruck angibt, dessen Wert zur Überprüfung verwendet werden soll (anzumerken ist hier, dass der Typ dieses Ausdrucks nicht vom Typ **bool** sein muss wie bei der **if** Abfrage). Nach den runden Klammern kommen geschweifte Klammern, die den **switch** Block öffnen, in dem die einzelnen Verzweigungsfälle mit dem Schlüsselwort **case** aufgelistet sind. Hinter **case** gibt man direkt den Wert an, mit dem der zu überprüfende Wert übereinstimmen soll, gefolgt von einem Doppelpunkt. Gibt die Überprüfung wahr zurück, so wird der Code, der nach dem Doppelpunkt folgt, ausgeführt. Stimmt der Wert mit keinem der Fälle überein, so wird der Code im optionalen **default** Case ausgeführt. Wichtig ist hierbei auch, dass in jedem **case** (auch bei **default**) jeweils am Ende eine **break** Anweisung stehen muss, sonst wirft der Compiler einen Fehler. **break** sorgt dafür, dass beim Ausführen dieser Anweisung der **switch** Block verlassen wird.

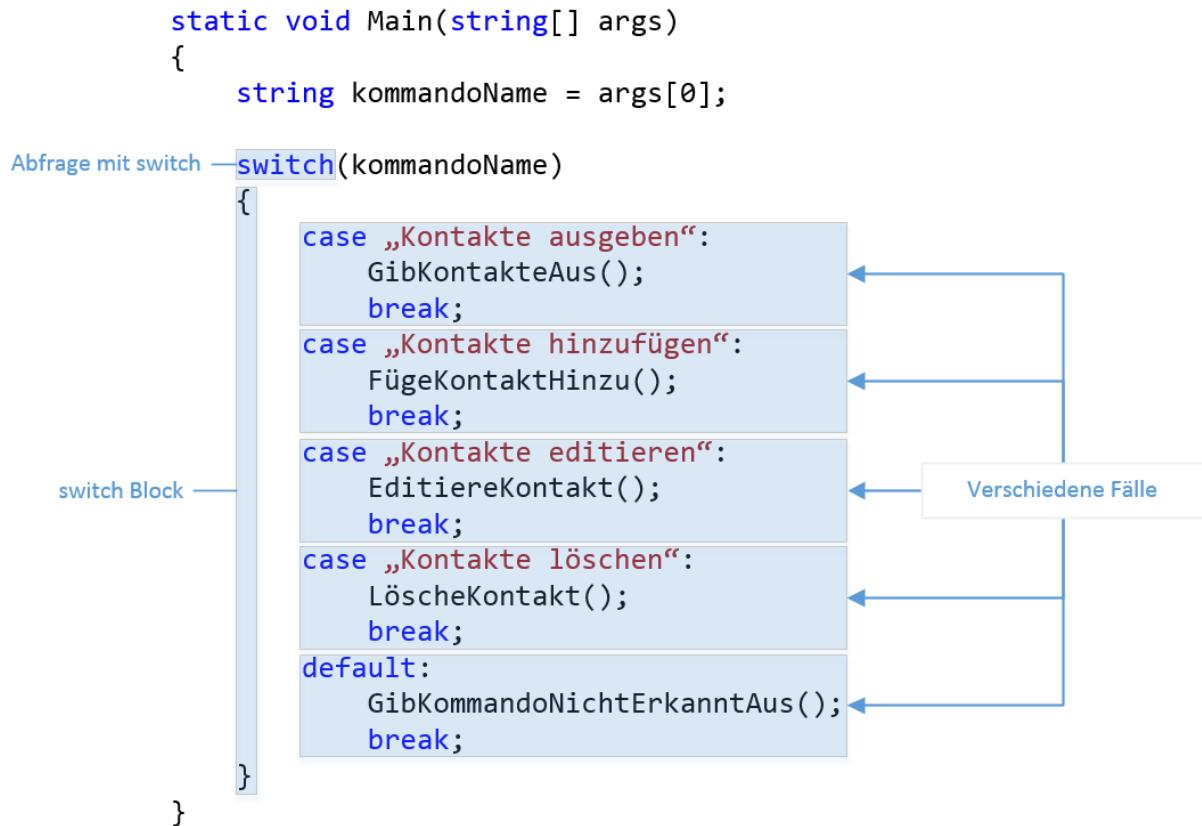


Abbildung 28: Switch Anweisung mit selber Funktionalität wie Abbildung 27

Ob man **if-else** Aneinanderreihungen oder **switch** Anweisungen bevorzugt, ist letztendlich die Vorliebe des jeweiligen Programmierers. Ich möchte an dieser Stelle jedoch hinzufügen, dass der häufige Einsatz von diesen Konstrukten keinen guten objektorientierten Design entspricht, da es gegen das sog. Open-Closed Prinzip verstößt. „Gute Programmierer“ sollten deshalb versuchen, **if-else** Aneinanderreihungen bzw. **switch** Anweisungen durch polymorphe Aufrufe zu ersetzen. Wie dies genau funktioniert, werden wir in einem späteren Kapitel lernen. Mit unserem bisherigen Wissen können wir aber guten Mutes diese Konstrukte einsetzen.

4.3.8.3 While und Do-While Schleife

Das Video zu diesem Abschnitt findet man unter

<https://www.youtube.com/watch?v=3XHCM7jCOGs>.

Die **while** Schleife wird genutzt, um einen Codeteil mehrmals hintereinander auszuführen, bis sich eine gewisse Bedingung ändert. Sie funktioniert genauso wie in C, bis auf den Fakt dass die Bedingung in C# zwingend ein boolescher Ausdruck sein muss (wie bei der **if** Abfrage). Am besten verdeutlicht man sich die Funktionsweise anhand eines Beispiels:

In Abbildung 29 sieht man die Funktion **EssenBisIchSattBin**, in der eine **while** Schleife eingesetzt wird. Zunächst werden zwei Variablen angelegt, **binIchHungry** und **anzahlSchnitzel** und jeweils initialisiert. Dann startet der interessante Teil: mit dem Schlüsselwort **while** wird eine Schleife erzeugt, die den Code innerhalb des Schleifenblocks ausführt, solange die Schleifenbedingung den Wert **true** zurückgibt. Innerhalb der Schleife wird eine andere Methode aufgerufen und die Variable **anzahlSchnitzel** inkrementiert (der **++** Operator funktioniert genauso wie in der Sprache C). Wenn diese Variable den Wert 7 erreicht hat, wird im **if** Block **binIchHungry** auf **false** gesetzt. Damit ist beim nächsten Schleifendurchlauf die

Schleifenbedingung `false`, weswegen die Schleife verlassen wird. Insgesamt wird der Schleifenblock also sieben Mal ausgeführt.

```
public void EssenBisIchSattBin()
{
    bool binIchHungrig = true;
    int anzahlSchnitzel = 0;

    while Schlüsselwort —— while (binIchHungrig) —— Schleifenbedingung
    {
        Schleifenblock ——
        EsseSchnitzel();
        anzahlSchnitzel++;
        if (anzahlSchnitzel == 7)
            binIchHungrig = false;
    }
}
```

Abbildung 29: While Schleife

Bei diesem Schleifentyp ist zu beachten, dass Code innerhalb der `while` Schleife nicht ausgeführt wird, wenn beim ersten Eintritt in die Schleife die Bedingung `false` zurückgibt. Dann wird mit dem nächsten Statement nach der Schleife weitergemacht. Um dies zu umgehen, gibt es eine Variation der `while` Schleife, die `do-while` Schleife:

```
public void NascheSüssigkeiten(bool mehrAlsEineSüssigkeit)
{
    int anzahlSüssigkeiten = 0;

    do —— do Schlüsselwort
    {
        Schleifenblock ——
        NascheSüssigkeit();
        anzahlSüssigkeiten++;
        if (anzahlSüssigkeiten == 5)
            mehrAlsEineSüssigkeit = false;
    } while (mehrAlsEineSüssigkeit); —— Schleifenbedingung
    while Schlüsselwort —— Boolescher Ausdruck
```

Abbildung 30: Do-While Schleife

In Abbildung 30 sieht man die Methode `NascheSüssigkeiten`, die den Parameter `mehrAlsEineSüssigkeit` hat, der vom Aufrufer entweder auf `true` oder `false` gesetzt werden kann. Bei `true` verhält sich die Schleife ähnlich wie im vorherigen Beispiel, wird jedoch `mehrAlsEineSüssigkeit` auf `false` gesetzt, dann sorgt die `do-while` Schleife dafür, dass der Schleifenblock zumindest einmal ausgeführt wird. Hätte man in diesem Fall eine `while` Schleife verwendet, würde der Schleifenblock nicht ausgeführt, da die Bedingung bereits zu Beginn `false` ist. Allgemein lässt sich sagen, dass bei dieser Schleife die Bedingung erst nach dem Ausführen des Schleifenblocks überprüft wird.

Beachten Sie bitte auch den kleinen Syntaxunterschied bei der **do-while** Schleife: nach den Klammern der Schleifenbedingung folgt noch ein Semikolon.

4.3.8.4 For und Foreach Schleife

Das Video zu diesem Abschnitt findet man unter <https://www.youtube.com/watch?v=p8BvBFpzdJY>.

Auch die schon aus C bekannte **for** Schleife ist in C# vertreten. Sie wird hauptsächlich genutzt, um einen Index zum Zugriff auf Arrays oder Collections bereitzustellen. Pro Schleifendurchlauf wird der Index dabei üblicherweise inkrementiert. Sehen wir uns dazu folgendes Beispiel an:

```

public void GibStringsAus(string[] strings)
{
    for (int i = 0; i < strings.Length; i++) {
        string @string = strings[i];
        Console.WriteLine(@string);
    }
}

```

Abbildung 31: For Schleife, die Index für Array bereitstellt

In Abbildung 31 ist die Funktion **GibStringsAus** zu sehen, die als Parameter einen Array von Strings erhält. Sie soll alle diese Strings auf der Konsole ausgeben. Dies wird gemacht mit einer **for** Schleife: nach dem Schlüsselwort kommt wie bei der **while** Schleife ein Paar runde Klammern. In diesen wird allerdings nicht einfach ein boolescher Ausdruck zu Abbruchbedingung angegeben, sondern insgesamt drei Ausdrücke:

- Der Initialausdruck (engl. Initial Expression) ist der erste dieser Ausdrücke. Dieser wird einmalig beim Eintritt in die Schleife ausgeführt. In ihm erstellt und initialisiert man üblicherweise die sog. Zählvariable (engl. Counter) für die **for** Schleife. Standardmäßig heißt diese Variable wie auch im obigen Beispiel **i**. Üblicherweise initialisiert man diese auf den Wert 0, da Arrays und Collections wie in C nullbasiert sind (das erste Element liegt an Index 0, das n-te Element an der Stelle $n - 1$). Auf die Zählvariable kann nur im Schleifenkopf und im Schleifenblock zugegriffen werden.
- Der Testausdruck (engl. Test Expression) ist der Ausdruck, der vor jedem Schleifendurchlauf überprüft wird. Nur wenn dieser Ausdruck **true** zurückgibt, wird der Schleifenblock erneut ausgeführt. Bei der standardmäßigen Nutzung der **for** Schleife gibt man hier die üblicherweise den booleschen Ausdruck **i < array.Length** (bzw. **i < collection.Count**) an, sodass mit **i** auf alle Elemente des Arrays oder der Collection zugegriffen werden kann.
- Der Aktualisierungsausdruck (engl. Update Expression) wird am Ende des Schleifenblocks ausgeführt, üblicherweise um die Zählvariable zu inkrementieren. Dies wird mit **i++** auch im obigen Beispiel gemacht.

Innerhalb der Schleifenblocks wird dann mit dem Indexoperator **[]** auf das jeweilige Element des Arrays zugegriffen und in einer gesonderten Variable namens **@string** gespeichert. Diese heißt so, da **string** ein Schlüsselwort in C# ist und deshalb nicht als Variablenname verwendet werden darf. In solchen Fällen ist es Konvention, dass man einfach ein **@** vor den entsprechenden Bezeichner stellt, wenn dieser mit einem Schlüsselwort kollidiert.

Bitte beachten Sie auch, dass Semikolons und nicht Kommas verwendet werden, um die einzelnen Ausdrücke im Schleifenkopf zu separieren. Intuitiv könnte man vermuten, dass eher letzteres der Fall ist. Verstehen Sie diese Semikolons auch nicht als Kennzeichnung einer Anweisung, sie sind wirklich nur dazu da, die Ausdrücke voneinander zu trennen (das sieht man u.a. auch daran, dass hinter dem Aktualisierungsausdruck kein Semikolon steht).

Des Weiteren möchte ich darauf hinweisen, dass Sie die drei besprochenen Ausdrücke im Schleifenkopf letztendlich beliebig kompliziert aufbauen können durch Ausdrucksschachtelung wie in 4.3.4 Anweisungen und Ausdrücke beschrieben. Machen Sie dies bitte nicht, es schadet ihrer Codequalität deutlich. Andere Softwareentwickler werden wesentlich länger brauchen, um ihren Code zu verstehen, wenn Sie die oben gezeigte Konvention für **for** Schleifen nicht anwenden.

Wer auf alle Elemente eines Arrays oder einer Collection zugreifen möchte, ohne dabei einen Index zu brauchen, dem bietet sich die **foreach** Schleife an. Zu ihr gibt es kein Pendant in C. Ihre Syntax ist deutlich einfacher als der der **for** Schleife. Im folgenden Beispiel erledigt die **foreach** Schleife das gleiche wie die **for** Schleife in Abbildung 31:

```
public void GibStringsAus(string[] strings)
{
    foreach Schlüsselwort — foreach (string @string in strings)
    {
        Schleifenblock — {
            }
    }
}
```

The diagram shows the C# code for a foreach loop. It highlights the 'foreach' keyword, the loop body enclosed in curly braces, and the array 'strings' used in the 'in' clause. Annotations explain the components: 'foreach Schlüsselwort' for the keyword, 'Schleifenblock' for the loop body, 'Elementvariable' for the variable '@string', 'in Schlüsselwort' for the 'in' keyword, and 'Aufzählung' for the array 'strings'.

Abbildung 32: Foreach Schleife

In Abbildung 32 wird mit dem Schlüsselwort **foreach** die gleichnamige Schleife erzeugt. Sie läuft über alle Elemente der angegebenen Aufzählung und weist das jeweils aktuelle Element der Elementvariablen zu. Innerhalb der Klammern gibt man drei Dinge an:

- Als erstes folgt die Angabe der Elementvariablen, sie enthält das aktuelle Element. Im obigen Beispiel heißt sie wie im Beispiel davor auch **@string**.
- Danach folgt das Schlüsselwort **in**
- Als letztes gibt man die Aufzählung an, die durchlaufen werden soll. Im obigen Beispiel ist das der als Parameter übergebene Array **strings**.

Wenn man den kompletten Schleifenkopf betrachtet, kann man ihn wie folgt in Prosa lesen: „Für jeden String in Strings, führe den Code im Schleifenblock aus“. Das ergibt die deutlich bessere Lesbarkeit gegenüber der **for** Schleife.

Bitte beachten Sie noch einen wichtigen Unterschied: Sie dürfen beim Einsatz der **foreach** Schleife im Schleifenblock nicht die Aufzählung manipulieren, bspw. indem Sie Elemente aus dem Array entfernen. Dies wird mit einem Fehler quittiert. Wenn Sie eine Aufzählung innerhalb eines Schleifenblocks manipulieren möchten, müssen Sie eine andere Schleifenart wählen.

Sie können als Aufzählung in einer **foreach** Schleife nicht nur Arrays und Collections verwenden, sondern mit jedem Objekt, das entweder das Interface **IEnumerable** oder das Interface **IEnumerable<T>** implementiert. Was dies genau bedeutet, lernen wir in einem späteren Kapitel.

4.3.8.5 Die Schlüsselwörter *continue* und *break* für Schleifen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=B1-5H5iZYOE>.

Innerhalb eines Schleifenblockes können bei allen Schleifentypen die Schlüsselwörter **continue** und **break** verwendet werden, um mit ersterem den nächsten Schleifendurchlauf vorzeitig anzustoßen oder mit letzterem die Schleife komplett zu verlassen. Dazu können wir folgendes Beispiel betrachten:

```
public bool ÜberprüfeObArrayZeichenfolgeEnthält(string[] array, string wert)
{
    for (int i = 0; i < array.Length; i++)
    {
        if (array[i] != wert)
            continue; Bei continue wird sofort zum nächsten Schleifendurchlauf gesprungen, nachfolgender Code wird nicht mehr ausgeführt
        return true;
    }

    return false;
}
```

Abbildung 33: Einsatz des Schlüsselworts *continue* in einer Schleife

In Abbildung 33 ist die Methode `ÜberprüfeObArrayZeichenfolgeEnthält` zu sehen, welche die zwei Parameter `array` und `wert` entgegen nimmt. Ihre Aufgabe ist es, festzustellen, ob der Array einen `string` mit derselben Zeichenkette, wie im Parameter `wert` angegeben, enthält. Dazu nutzt die Funktion eine `for` Schleife, die als Index für den Array bereitgestellt wird. Innerhalb des Schleifenblocks wird in der ersten `if` Abfrage überprüft, ob das aktuelle Element des Arrays `array[i]` denselben Wert hat wie der mitgelieferte Parameter. Ist dies nicht der Fall, wird sofort die `continue` Anweisung ausgeführt, die zum nächsten Schleifendurchlauf springt. Das kann in zwei möglichen Varianten enden:

1. Im Array befindet sich kein Element, das mit `wert` übereinstimmt. Damit wird ständig das `continue` Statement ausgeführt und die Anweisung `return true` wird niemals erreicht. Nachdem die Schleife beendet ist, wird `false` über die letzte Anweisung der Methode zurückgegeben.
2. Eines der Arrayelemente ist gleich dem mitgelieferten Wert, sodass der `if` Block nicht ausgeführt wird und deshalb mit der Anweisung `return true` die Funktion beendet wird.

Ebenso ist es möglich, diese Funktionalität mit dem `break` umzusetzen, wie folgendes Beispiel zeigt:

```

public bool ÜberprüfeObArrayZeichenfolgeEnthält(string[] array, string wert)
{
    bool wurdeÜbereinstimmungGefunden = false;
    foreach (string element in array)
    {
        if (element == wert)
        {
            wurdeÜbereinstimmungGefunden = true;
            break; ----- Mit break wird der aktuelle Schleifenblock komplett verlassen
        }
    }

    return wurdeÜbereinstimmungGefunden;
}

```

Abbildung 34: Einsatz des Schlüsselworts `break` in einer Schleife

Der in Abbildung 34 eingesetzte Code erfüllt genau denselben Zweck wie der im Beispiel vorher: auch hier wird ein Array nach einem bestimmten Wert durchsucht. Allerdings wird im Schleifenblock zunächst überprüft, ob das aktuelle Element mit dem mitgelieferten Wert übereinstimmt. Falls das der Fall ist, wird die Variable `wurdeÜbereinstimmungGefunden` auf `true` gesetzt und die Schleife mit der `break` Anweisung komplett verlassen. Nach der Schleife wird einfach der Wert der variablen zurückgegeben.

Bitte beachten Sie, dass `continue` und `break` sich jeweils nur auf eine Schleife beziehen, d.h. wenn Sie mehrere Schleifen schachteln, ist nur die Schleife von den Schlüsselwörtern betroffen, in dessen Schleifenblock sie stehen. Des Weiteren würde ich Ihnen raten, nicht zu viele Schleifen ineinander zu schachteln, da dies die Lesbarkeit Ihres Codes deutlich einschränkt. Lagern Sie Schleifen, die innerhalb anderer Schleifen liegen, möglichst in eigene Methoden aus und rufen Sie diese dann innerhalb der äußeren Schleife auf. Als Faustregel gilt: eine Schleife ist gut, zwei geschachtelte Schleifen sind teilweise ok, ab drei ineinander geschachtelten Schleifen ist der Zustand inakzeptabel.

4.3.9 Operatoren

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=SaAQ0nwkg3Y>.

Die Operatoren in C# sind weitestgehend dieselben, die man bereits aus der Sprache C kennt. Sie sind in den folgenden Tabellen aufgeführt.

4.3.9.1 Arithmetische Operatoren für numerische Ausdrücke

Arithmetische Operatoren werden mit zwei numerischen Operanden, d.h. Ganzzahl- oder Gleitkommawerten, genutzt.

Operator	Beschreibung
+	Hat zwei Funktionalitäten <ul style="list-style-type: none"> Als Zuweisungsoperator bildet er die Summe zweier Operanden ($x + y$) Als unärer Vorzeichenoperator beschreibt er eine positive Zahl ($+x$)
-	Hat wie der Plusoperator zwei Funktionalitäten <ul style="list-style-type: none"> Als Subtraktionsoperator bildet er die Differenz zweier Operanden ($x - y$) Als unärer Vorzeichenoperator beschreibt er eine negative Zahl ($-x$)
*	Multiplikationsoperator für zwei Operanden ($x * y$)
/	Divisionsoperator für zwei Operanden (x / y) Vorsicht, wenn er auf Ganzahltypen angewendet wird, schneidet er Nachkommastellen einfach ab (kein Runden)

%	Modulooperator für zwei Ganzahloperanden, der den Restwert einer Division zurückgibt ($x \% y$)
++	Erhöht den Inhalt eines Operanden um 1 (++x gibt Wert nach Erhöhung zurück, x++ den vor der Erhöhung)
--	Verringert den Inhalt eines Operanden um 1 (gleiche Funktionsweise wie ++ Operator)

Abbildung 35: Arithmetische Operatoren in C#

4.3.9.2 Vergleichsoperatoren

Vergleichsoperatoren werden mit zwei Operanden genutzt, die miteinander vergleichbar sind. Bspw. sind das numerische Werte oder Strings. Im Allgemeinen muss der jeweilige Typ die Interface `IComparable<T>` und `IEquatable<T>` implementieren und die entsprechenden Operatoren überladen (was das genau heißt, klären wir in einem späteren Kapitel).

Operator	Beschreibung
<code>a == b</code>	Gibt <code>true</code> zurück, wenn a gleich b ist
<code>a != b</code>	Gibt <code>false</code> zurück, wenn a gleich b ist
<code>a > b</code>	Gibt <code>true</code> zurück, wenn a größer als b ist
<code>a < b</code>	Gibt <code>true</code> zurück, wenn a kleiner als b ist
<code>a <= b</code>	Gibt <code>true</code> zurück, wenn a kleiner als oder gleich b ist
<code>a >= b</code>	Gibt <code>true</code> zurück, wenn a größer als oder gleich b ist

Abbildung 36: Vergleichsoperatoren in C#

4.3.9.3 Logische Operatoren für boolesche Ausdrücke

Diese Operatoren verknüpfen boolesche Ausdrücke. Dabei müssen die Operanden vom Typ `bool` sein.

Operator	Beschreibung
!	Unärer Negationsoperator, der einen booleschen Ausdruck umdreht (<code>!a</code>)
&&	And-Operator, verknüpft zwei boolesche Ausdrücke und gibt <code>true</code> zurück, wenn beide wahr sind (<code>a && b</code>)
	Or-Operator, verknüpft zwei boolesche Ausdrücke und gibt <code>true</code> zurück, wenn mindestens eins der beiden wahr ist (<code>a b</code>)
^	XOr-Operator, verknüpft zwei boolesche Ausdrücke und gibt <code>true</code> zurück, wenn beide Ausdrücke einen unterschiedlichen Wahrheitswert haben (<code>a ^ b</code>)

Abbildung 37: Logische Operatoren für boolesche Ausdrücke in C#

Zu den logischen Operatoren `&&` und `||` muss gesagt werden, dass diese auch jeweils in einer anderen Ausführung mit nur jeweils einem Zeichen existieren. Diese Varianten sollten aber im allgemeinen Programmieralltag vermieden werden, da sie zwei logische Ausdrücke nicht so schnell abarbeiten wie die Operatoren `&&` und `||`. Sehen wir uns dazu folgendes Beispiel an:

In Abbildung 38 wird in der `Main` Methode zwei boolesche Ausdrücke abgefragt. Beim ersten `if` Block wird der einfache `&` Operator eingesetzt. Bei diesem wird zunächst der Ausdruck links des Operators evaluiert, d.h. es wird die Methode `GibFalschZurück` aufgerufen. Danach wird der Ausdruck rechts des Operators (`GibWahrZurück`) evaluiert. Sind beide Ergebnisse vorhanden, werden diese logisch Und verknüpft und das Ergebnis für die `if` Überprüfung bereitgestellt.

Beim zweiten `if` Block wird der `&&` Operator eingesetzt (mit zwei kaufmännischen Und). Dieser geht anders als der einfache Und-Operator vor: er evaluiert zunächst den linken Ausdruck (`GibFalschZurück`) und merkt dabei, dass dieser `false` zurückgegeben hat. Bei einer logischen Und-Verknüpfung müssen aber beide Ausdrücke wahr sein, damit der Gesamtausdruck wahr wird. Da dies nicht mehr möglich ist, beendet der `&&` Operator sofort die Überprüfung und gibt `false` an die `if` Überprüfung weiter.

In Abbildung 38 wird deshalb keiner der beiden `Console.WriteLine` Aufrufe ausgeführt. Von außen gesehen verhalten sich der & und der && Operator gleich. Allerdings ist der zweite schneller zu seinem Ergebnis gekommen.

```

class Program
{
    static void Main()
    {
        if (GibFalschZurück() & GibWahrZurück())
            Console.WriteLine("Ich bin im ersten if Block");

        if (GibFalschZurück() && GibWahrZurück())
            Console.WriteLine("Ich bin im zweiten if Block");
    }
    static bool GibFalschZurück()
    {
        return false;
    }
    static bool GibWahrZurück()
    {
        return true;
    }
}

```

Abbildung 38: Unterschied zwischen dem & und dem && Operator

Für die logischen Oder-Operatoren | und || gilt analog zu & und && dasselbe.

4.3.9.4 Bitweise Operatoren für numerische Ausdrücke

Diese Operatoren manipulieren Werte auf Bitebene.

Operator	Beschreibung
<code>~</code>	Invertiert jedes Bit eines Ausdrucks (<code>~a</code>)
<code> </code>	Verknüpft jede Bitstelle von zwei Ausdrücken mit logischem Oder (<code>a b</code>)
<code>&</code>	Verknüpft jede Bitstelle von zwei Ausdrücken mit logischem Und (<code>a & b</code>)
<code>^</code>	Verknüpft jede Bitstelle von zwei Ausdrücken mit XOr (<code>a ^ b</code>)
<code><<</code>	Verschiebt alle Bits von <code>a</code> um die in <code>b</code> angegebene Anzahl von Stellen nach links, dabei werden die freien Bitstellen mit 0 aufgefüllt (<code>a << b</code>)
<code>>></code>	Wie <code><<</code> Operator, nur wird nach rechts verschoben

Abbildung 39: Bitweise Operatoren für numerische Ausdrücke in C#

Die Bitweise Operatoren werden kaum noch in der alltäglichen Programmierung genutzt. Bitte beachten Sie, dass das & und das | Zeichen sowohl für Bitweise als auch für logische Operatoren stehen (je nachdem, welche Operanden ihnen übergeben werden).

4.3.9.5 Zuweisungsoperatoren

Diese Operatoren weisen einer bestimmten Speicherstelle einen Wert zu.

Operator	Beschreibung
<code>a = b</code>	Weist <code>a</code> den Wert von <code>b</code> zu
<code>a += b</code>	Weist <code>a</code> den Wert von <code>a + b</code> zu
<code>a -= b</code>	Weist <code>a</code> den Wert von <code>a - b</code> zu
<code>a *= b</code>	Weist <code>a</code> den Wert von <code>a * b</code> zu

<code>a /= b</code>	Weist a den Wert von <code>a / b</code> zu
<code>a %= b</code>	Weist a den Wert von <code>a % b</code> zu
<code>a &= b</code>	Weist a den Wert von <code>a & b</code> zu (Bitweise)
<code>a = b</code>	Weist a den Wert von <code>a b</code> zu (Bitweise)
<code>a ^= b</code>	Weist a den Wert von <code>a ^ b</code> zu (Bitweise)
<code>a <= b</code>	Weist a den Wert von <code>a < b</code> zu
<code>a >= b</code>	Weist a den Wert von <code>a > b</code> zu

Abbildung 40: Zuweisungsoperatoren in C#

4.3.9.6 Weitere Operatoren

Operator	Beschreibung
<code>.</code>	Punktoperator, wird zum Zugriff auf die Members eines Objekts oder Struktur verwendet
<code>[]</code>	Indexoperator, wird für den Zugriff auf einzelne Elemente in Arrays und Collections genutzt
<code>()</code>	Hat zwei Funktionen <ul style="list-style-type: none"> Zur Ordnung der Ablaufreihenfolge von Ausdrücken in einem Statement <code>((a + b) / (c + d))</code> Zum expliziten Casten eines Ausdrucks <code>(double x = (double)42;)</code>
<code>?:</code>	Gibt einen von zwei Werten in Abhängigkeit eines booleschen Wertes zurück
<code>new</code>	Instanziert eine Klasse, eine Struktur oder einen Array
<code>is</code>	Überprüft, ob zur Laufzeit ein Objekt von einem bestimmten Typ ist
<code>as</code>	Versucht ein Objekt zur Laufzeit in einen anderen Typ zu casten, gibt <code>null</code> zurück bei Fehler
<code>typeof</code>	Gibt den <code>System.Type</code> eines Objekts oder einer Struktur zur Laufzeit zurück (notwendig für Reflection)

Abbildung 41: Weitere Operatoren in C#

Die Funktionalität, die diese Operatoren bieten, wird sich uns erschließen, wenn wir mit Objekten und Klassen in den späteren Kapiteln zu tun bekommen.

4.3.9.7 Aufrufreihenfolge von Operatoren

Ausdrücke sind, wie oben gesehen, entweder Operatoren oder Funktionsaufrufe. Wenn mehrere Operatoren in einer Anweisung eingesetzt werden, dann werden diese nach einer gewissen Reihenfolge abgearbeitet, die in der folgenden Tabelle abgebildet ist. Dabei binden die Operatoren in Gruppe 1 stärker als die in Gruppe 2 usw. Man muss diese Tabelle in keinem Fall auswendig können, da man prinzipiell darauf achten sollte, nur wenige Ausdrücke in einer Anweisung zu nutzen, dennoch sollte man sich diese Tabelle einmal angesehen haben. Zu beachten ist, dass sich alle Operatoren, die in C ebenso vorhanden sind, gleich verhalten wie in dieser Sprache.

Gruppe	Operatoren
1	<code>a.b, a[x], x++, x--, new, typeof</code>
2	<code>+(unär), -(unär), !, ~, ++x, --x</code>
3	<code>*, /, %</code>
4	<code>+(additiv), -(subtraktiv)</code>
5	<code><<, >></code>
6	<code><, >, <=, >=, is, as</code>
7	<code>==, !=</code>
8	<code>&</code>
9	<code>^</code>
10	<code> </code>

11	&&
12	
13	?:
14	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =

Abbildung 42: Abarbeitungsreihenfolge der Operatoren in C#

4.4 Datum und Zeit

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=dS64zcvDmEA>.

Für die Verarbeitung von Zeitpunkten und Zeitspannen werden die Strukturen `System.DateTime` und `System.TimeSpan` des .NET Frameworks genutzt. Auch wenn diese Typen keine primitiven Typen sind, gehören sie zu den wichtigsten und am häufigsten eingesetzten im Programmieralltag. Am besten erklärt man diese Typen anhand eines kurzen Beispiels.

```
static void Main()
{
    DateTime meinGeburtstag = new DateTime(1987, 2, 12);
    DateTime jetzt = DateTime.Now;
    TimeSpan alter = jetzt - meinGeburtstag;
    Console.WriteLine(alter.Days);
}
```

Abbildung 43: DateTime und TimeSpan verwenden

In Abbildung 43 kann man vier Anweisungen in der `Main` Methode sehen. In der ersten wird zunächst mit dem `new` Operator eine neue `DateTime` Struktur erstellt, die das Datum 12.02.1987 darstellt und in der Variable `meinGeburtstag` gespeichert wird. In der nächsten Anweisung wird das aktuelle Datum (und Uhrzeit) über die statische Eigenschaft `DateTime.Now` abgegriffen und der Variable `jetzt` zugewiesen. Im dritten Statement wird der Minusoperator auf die beiden erstellten Variablen angewendet, mit dem Resultat einer `TimeSpan` Struktur, deren Wert in der Variable `alter` festgehalten wird. Dieser Wert repräsentiert die Zeitspanne von meinem Geburtstag bis hin zum jetzigen Zeitpunkt. In der letzten Anweisung wird die Anzahl an Tagen, die ich seitdem auf der Welt bin, auf der Konsole ausgegeben.

Die beiden Strukturen `DateTime` und `TimeSpan` bieten noch viele andere Funktionalitäten an, die ich hier nicht aufzählen werde. Informieren können Sie sich zu diesen unter den folgenden Links:

- `TimeSpan`: [http://msdn.microsoft.com/de-de/library/system.timespan\(v=vs.110\).aspx](http://msdn.microsoft.com/de-de/library/system.timespan(v=vs.110).aspx)
- `DateTime`: [http://msdn.microsoft.com/de-de/library/system.datetime\(v=vs.110\).aspx](http://msdn.microsoft.com/de-de/library/system.datetime(v=vs.110).aspx)

5 Klassen, Kapselung und Vererbung

In diesem Kapitel beginnen wir mit den Neuheiten, die uns C# für die objektorientierte Programmierung bietet. Hier lernen Sie auch die grundlegenden Mechanismen der Kapselung und der Vererbung kennen.

5.1 Klassendefinition und -scope

Das Video zu diesem Abschnitt findet man unter <https://www.youtube.com/watch?v=UDvmbqTfh7o>.

Klassen sind der Grundstein zur objektorientierten Programmierung in C#. Wir schauen uns zunächst einmal an, wie man eine Klasse aufbaut und was diese enthalten.

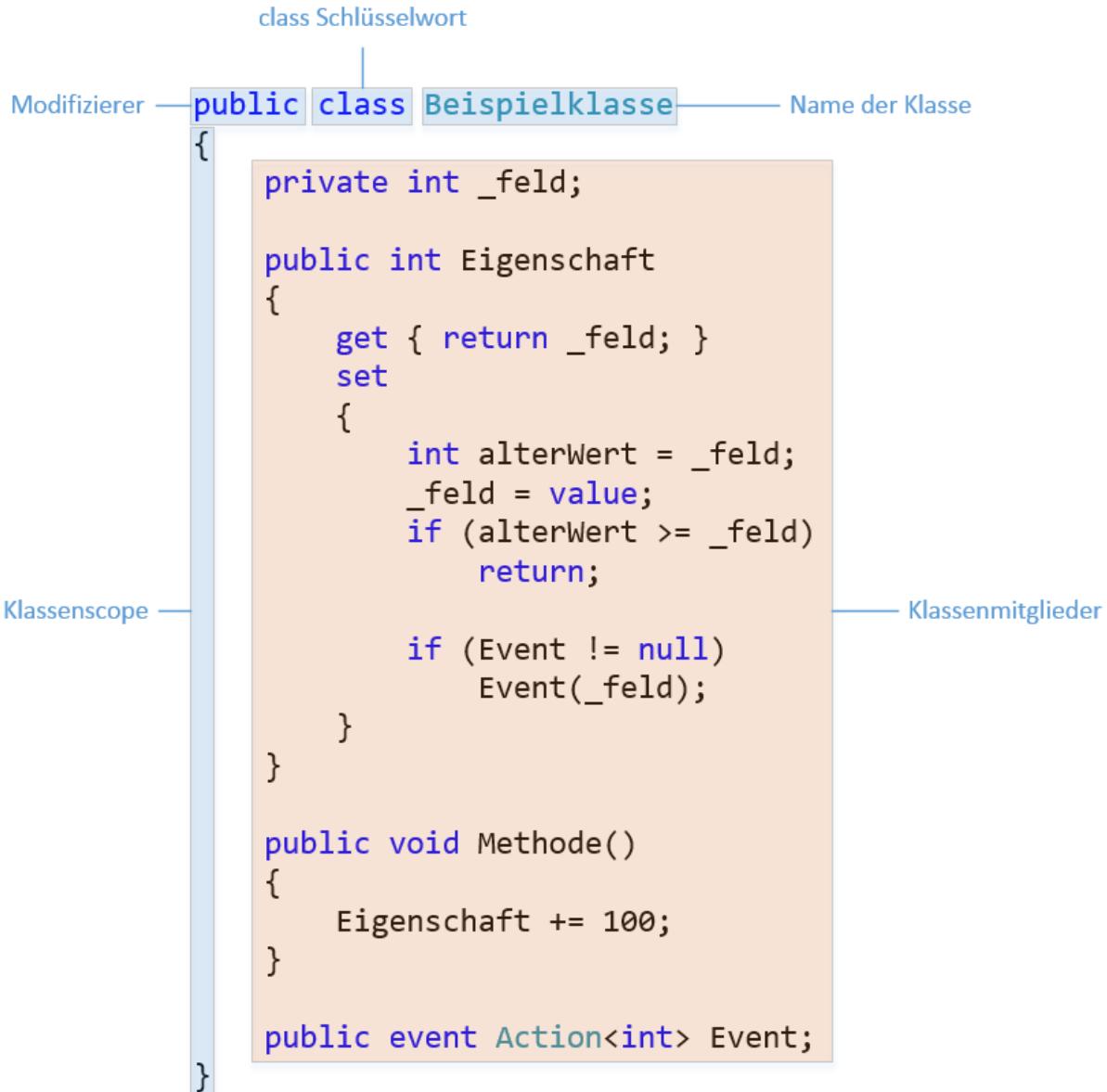


Abbildung 44: Beispielklasse in C#

In Abbildung 44 sieht man die Grundstruktur einer Klasse: sie besteht aus einem Klassenkopf und dem Klassenscope. Im Klassenkopf gibt man folgende Dinge in genau dieser Reihenfolge an:

- Die Modifizierer: mit diesen legt man mehrere Dinge über die Klasse wie z.B. den Zugriff fest. Im obigen Beispiel sehen sie nur den Modifizierer `public`, allerdings sind auch noch andere Modifizierer möglich. Sofern sie keine Modifizierer angeben, wird implizit der Modifizierer

`internal` für Klassen gesetzt. Wie Modifizierer genau funktionieren, lernen Sie in einem der nächsten Abschnitte.

- Das Schlüsselwort `class`: damit zeigt man dem Compiler an, dass der hier definierte Typ eine Klasse ist. Welche anderen Typen es noch in C# gibt und wie man diese einsetzt, lernen wir in einem späteren Abschnitt.
- Der Name der Klasse: über diesen kann die Klasse später angesprochen werden.

Im Anschluss folgen zwei geschweifte Klammern, die den Klassenscope vorgeben. Innerhalb dieses Scopes wird eine beliebige Anzahl von sog. Klassenmitgliedern (engl. Class Members) aufgeführt. Die verschiedenen Klassenmitglieder werden wir in den folgenden Abschnitten genauer betrachten. Sie müssen zum jetzigen Zeitpunkt noch nicht verstehen, was im roten Kästchen im Beispiel passiert. Dennoch sieht man jeweils eines der vier unterschiedlichen möglichen Members bei Klassen:

- Felder
- Eigenschaften
- Methoden
- Events

5.2 Methoden als Klassenmitglieder

In Kapitel 4 haben wir uns bereits ausführlich mit Methoden beschäftigt. Da diese in C# immer innerhalb des Scopes einer Klasse liegen müssen, sind sie auch automatisch immer Klassenmitglieder. Bisher haben wir allerdings hauptsächlich statische Methoden eingesetzt, d.h. alle Methoden, die aufgerufen wurden, hatten den Modifizierer `static`.

5.3 Felder als Klassenmitglieder

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=SGYKC9uxBwg>.

Den nächsten Typ von Klassenmitgliedern, den wir uns anschauen, sind Felder: sie sind dazu da, um unterschiedlichen Werte, die innerhalb eines Objekts gespeichert werden sollen, festzulegen.

Betrachten wir dazu folgendes Beispiel:

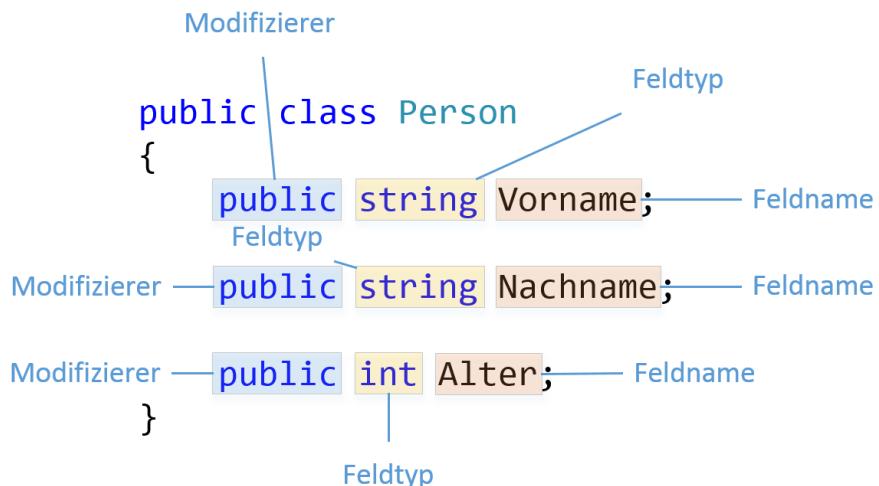


Abbildung 45: Felddefinitionen in einer Klasse

In Abbildung 45 werden in der Klasse `Person` die drei Felder `Vorname`, `Nachname` und `Alter` definiert. Ein Feld definiert man dabei wie folgt:

- Zunächst gibt man die Modifizierer für das Feld an. Im obigen Beispiel ist nur der Modifizierer **public** bei jedem der Felder angegeben, damit von außerhalb des Klassenscopes auf diese zugegriffen werden kann.
- Danach folgt der Typ des Feldes. Bei Vorname und Nachname ist dies jeweils **string**, bei Alter **int**.
- Danach folgt der Name des Feldes, über den es nachher angesprochen werden kann.
- Abgeschlossen wird die Felddefinition mit einem Semikolon.

Im jetzigen Zustand verhält sich die Klasse Person noch wie eine Struktur in C, die mit denselben Strukturvariablen ausgestattet sind. Wie man diese Klasse instanziert und einsetzt, schauen wir uns im nächsten Abschnitt an.

5.4 Von Klassen zu Objekten

Das Video zu diesem Abschnitt findet man unter

<https://www.youtube.com/watch?v=mW9HmSyXZbA>.

Da im Namen „objektorientierte Programmierung“ mehr Objekt als Klasse steckt, ist es natürlich wichtig, auf den Zusammenhang zwischen diesen beiden Komponenten einzugehen. Aus einer Klasse wird ein Objekt, indem man die Klasse mit dem **new** Operator instanziert. Dazu schauen wir uns folgende Abbildung an, das die im vorigen Beispiel erstellte Klasse Person einsetzt:

```
static void Main()
{
    Person walter = new Person();           Intanziierung eines Person
                                            Objekts über new Operator
    walter.Vorname = „Walter“;             Wertzuweisung an Felder über
    walter.Nachname = „White“;            Punktoperator
    walter.Alter = 52;
}
```

Abbildung 46: Instanziierung eines Objekts und Wertzuweisung an dessen Felder

In Abbildung 46 wird in der Main Methode als erstes ein neues Objekt vom Typ **Person** erstellt mit dem Ausdruck **new Person()**. In diesem Ausdruck wird zunächst das Schlüsselwort **new** verwendet, gefolgt vom Name der Klasse, die wir instanziieren wollen, gefolgt von einem Paar runde Klammern. Dabei sieht **Person()** wie ein parameterloser Funktionsaufruf aus und genau das ist es auch, obwohl hier kein Funktionsname, sondern ein Klassenname verwendet wird: hier wird ein sog. Konstruktor aufgerufen. Konstruktoren sind spezielle Methoden in Klassen, die ausschließlich für die Initialisierung zuständig ist. Es wird bei Klassen immer ein Standardkonstruktor vom Compiler erstellt, wenn wir, wie in Abbildung 45 offensichtlich zu sehen ist, keinen eigenen erstellt haben. Wie genau Konstruktoren funktionieren und was man bei ihnen beachten muss, schauen wir uns in einem späteren Abschnitt an.

Der **new** Operator gibt die Adresse des neuen Objekts zurück, die der Variablen **walter** zugewiesen wird. Auch wenn sich die Schreibweise von Variablen zu Wertetypen wie **int** oder **bool** nicht unterscheidet, sollte man sich bewusst machen, dass **walter** nur eine Referenz zu einem Objekt und nicht das Objekt selber hält, denn:

Alle Klassen sind in C# sog. Referenztypen. Wenn also Klassen zu Objekten mit dem new Operator instanziert werden, wird das Objekt auf dem sog. Heap im Arbeitsspeicher erstellt. Die Adresse des neuen Objekts im Heap wird vom new Operator zurückgegeben und im Normalfall einer Variablen oder einem Feld zugewiesen. Da diese in diesem Fall nur Adressen auf Objekte enthalten, kann man sich ihre Funktionsweise ähnlich wie Zeiger in C vorstellen.

In den letzten drei Anweisungen des obigen Beispiels wird der Punktoperator genutzt, um auf die Felder des in der Variable `walter` referenzierten Objekts zuzugreifen. Nacheinander werden bei diesen Feldern die Werte „Walter“ und „White“ für Vornamen und Nachnamen sowie 52 für das Alter gesetzt. Die Aufgabe des Punktoperators ist hierbei die Dereferenzierung der Adresse, die in der Variable `walter` gespeichert ist.

Im letzten Beispiel haben wir nur ein Objekt aus einer Klasse erstellt. Es ist jedoch möglich, beliebig viele Objekte aus einer Klasse zu instanzieren, indem man einfach den Konstruktor einer Klasse mehrmals mit dem new Operator aufruft, wie in der folgenden Abbildung zu sehen ist:

```
static void Main()
{
    Person walter = new Person(); — Instanziierung des 1. Objekts
    walter.Vorname = „Walter“;
    walter.Nachname = „White“;
    walter.Alter = 52;

    Person jesse = new Person(); — Instanziierung des 2. Objekts
    jesse.Vorname = „Jesse“;
    jesse.Nachname = „Pinkman“;
    jesse.Alter = 27;

    Person hank = new Person(); — Instanziierung des 3. Objekts
    hank.Vorname = „Hank“;
    hank.Nachname = „Schrader“;
    hank.Alter = 49;
}
```

Abbildung 47: Mehrere Objekte initialisieren

In Abbildung 47 werden nicht nur eines, sondern drei Objekte der Klasse `Person` erstellt. Dies wird jeweils durch einen Konstruktorauftrag mit dem new Operator bewerkstelligt, wie schon im Beispiel vorher. Man kann dazu folgende Überlegung machen:

Klassen kann man als Baupläne für Objekte betrachten. Dabei gibt eine Klasse durch ihre Mitgliederdefinitionen vor, wie die Speicherstruktur des tatsächlichen Objekts aussieht, d.h. welche Felder und Methoden vorhanden sind. Über Konstruktoraufträge mit dem new Operator lassen sich beliebig viele Objekte aus einer Klasse instanzieren. Erst über diese Objekte kann man die Funktionalität der Klasse tatsächlich einsetzen, d.h. auf deren Werte zugreifen und deren Methoden aufrufen.

Die Klasse `Person` aus Abbildung 45 besitzt ausschließlich Felder und keine Methoden, weswegen sie in ihrer Funktionalität einer Struktur in C gleicht. So eine Klasse hat mit objektorientierter Design

auch nichts zu tun. Wie man das besser macht, schauen wir uns im nächsten Abschnitt an, in dem wir über die Modifizierer **public** und **private** reden.

5.5 Die Modifizierer **public** und **private** und die **this** Referenz

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=ZABqg1e-3mU>.

In diesem Abschnitt schauen wir uns ein neues Klassenbeispiel an, in dem das Zusammenspiel von Feldern und Methoden sowie die Modifizierer **private** und **public** erklärt werden. Die Klasse stellt Identifikationsnummern bereit, die allgemein mit ID abgekürzt werden. Das wichtige an einer Identifikationsnummer ist ihre Eindeutigkeit, d.h. beim Generieren darf es niemals vorkommen, dass eine schon generierte ID nochmals erzeugt wird. Schauen wir uns dazu die folgende Abbildung an:

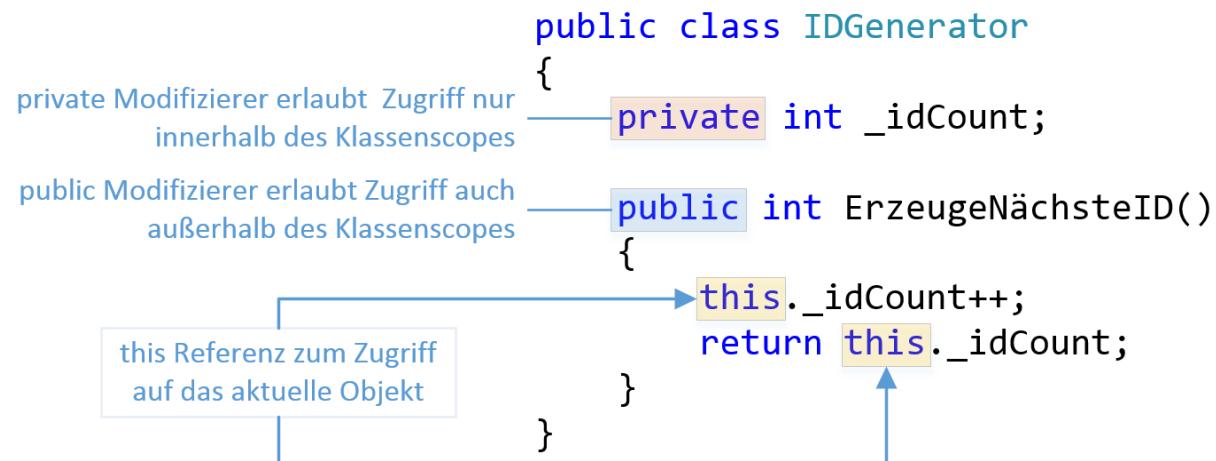


Abbildung 48: Die Modifizierer **public** und **private** sowie die **this** Referenz im Einsatz

In Abbildung 48 sehen Sie die Klasse **IDGenerator**, die zwei Klassenmitglieder hat: das Feld **_idCount** vom Typ **int** sowie die Funktion **ErzeugeNächsteID**, die keine Parameter entgegennimmt, aber dafür einen Wert vom Typ **int** zurückgibt, eben die nächste ID. Wenn wir den Code dieser Funktion betrachten, sehen wir, dass zunächst das Feld inkrementiert wird und im Anschluss der Wert des Feldes zurückgegeben wird. Dabei wird die **this** Referenz benutzt, um auf Instanzmitglieder der aktuellen Klasse zuzugreifen. Dies funktioniert wie folgt:

- wenn eine Methode auf einem Objekt aufgerufen wird, wird nebenbei auch die **this** Referenz auf die Referenz dieses Objekts gesetzt.
- Im Code der aufgerufenen Funktion kann diese dann verwendet werden, um auf andere Instanzmitglieder des Objekts zuzugreifen.

Deswegen ist die **this** Referenz, auch wenn sie auf den ersten Blick so aussieht, kein Literal. Ihr Wert ändert sich nämlich über den Programmablauf hinweg. Wichtig ist auch, dass die **this** Referenz nur in Methoden verwendet werden kann, die nicht statisch sind (die also nicht mit dem Modifizierer **static** gekennzeichnet sind).

Das Schlüsselwort **this** kann auch weggelassen werden, wenn der Bezeichner des Klassenmitglieds, dass mit **this** angesteuert wird, eindeutig ist. Diese Eindeutigkeit wäre bspw. nicht gegeben, wenn innerhalb der Methode ein Parameter oder eine Variable mit demselben Bezeichner definiert wäre. Deshalb kann man die Klasse **IDGenerator** auch wie folgt schreiben (die **this** Referenz wird dann automatisch vom Compiler beim Erstellungsvorgang gesetzt):

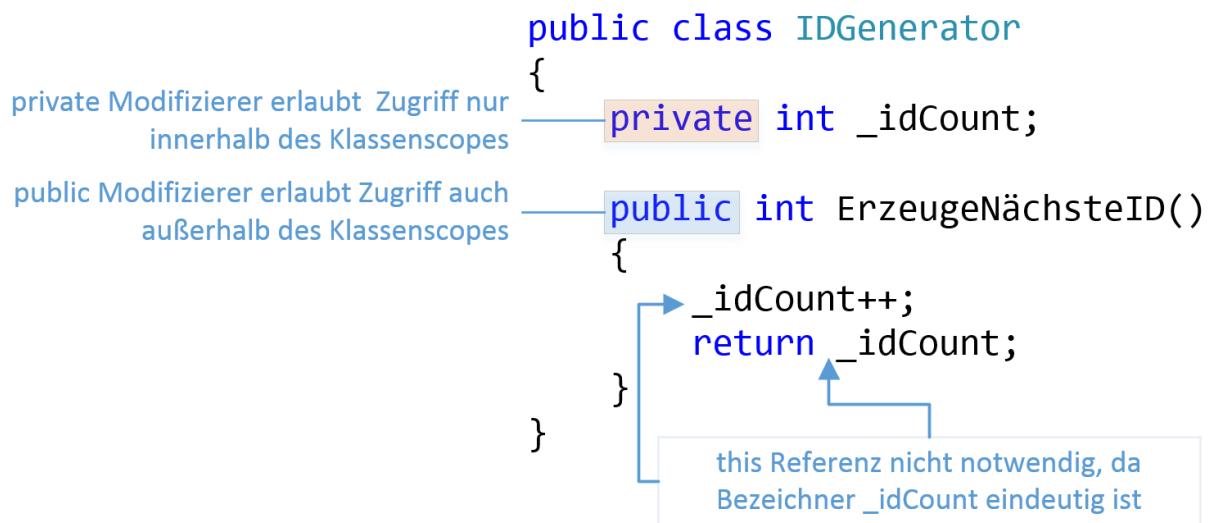


Abbildung 49: Die `this` Referenz kann bei eindeutigen Klassenmitgliederbezeichnern weggelassen werden

Als nächstes möchte ich auf die unterschiedlichen Modifizierer `private` und `public` eingehen. Ersterer sagt aus, dass auf Klassenmitglieder nur innerhalb des Klassenscopes, in dem sie definiert wurden, zugegriffen werden kann. Dadurch hat die Klasse Hoheit über dieses Mitglied: nur in ihr kann das entsprechende Feld gesetzt oder ausgelesen bzw. die entsprechende Funktion aufgerufen werden. Im Gegensatz dazu steht `public`: Klassenmitglieder, die als `public` gekennzeichnet sind, können auch außerhalb des Klassenscopes angesprochen werden. Ist ein Klassenmitglied mit keinem Modifizierer gekennzeichnet, dann ist es standardmäßig `private`.

Die Klasse `IDGenerator` kann man dann wie in der folgenden Abbildung zu sehen einsetzen.

```

static void Main()
{
    IDGenerator generator = new IDGenerator(); // Instanziierung des IDGenerators
    int ersteID = generator.ErzeugeNächsteID(); // Zugriff auf public Members des Objekts
    int zweiteID = generator.ErzeugeNächsteID();

    Console.WriteLine(erdeID);
    Console.WriteLine(zweiteID);
}

```

Abbildung 50: Die Klasse `IDGenerator` einsetzen

In Abbildung 50 wird zunächst ein Objekt der Klasse `IDGenerator` erzeugt mit dem `new` Operator und dem Aufruf des Standardkonstruktors der Klasse. In den folgenden beiden Anweisungen, im Beispiel oben rot gekennzeichnet, wird jeweils die Methode `ErzeugeNächsteID` der Klasse aufgerufen. Dafür wird zunächst die Variable `generator` mit dem Punktoperator dereferenziert, um auf die Klassenmitglieder, die `public` sind, zuzugreifen (bitte beachten Sie, dass sich die `Main` Methode nicht im Klassenscope der Klasse `IDGenerator` befindet). In der `Main` Methode ist es deshalb auch nicht möglich, über `generator` das Feld `_idCount` zuzugreifen, da dieses `private` gekennzeichnet ist. Daraus kann man sich auch den Fachbegriff API ableiten:

Die API (Application Programming Interface, dt. Programmierschnittstelle) einer Klasse sind alle Klassenmitglieder, die `public` sind. Damit beschreibt der Begriff API die Menge aller Methoden, die (andere) Programmierer nutzen können, wenn sie eine Klasse einsetzen.

Zurück zur `this` Referenz: jedes Mal, wenn eine Objektreferenz dereferenziert und eine Methode auf dem betreffenden Objekt aufgerufen wird, wird die `this` Referenz aktualisiert. Im Beispiel oben passiert das bei den zwei Aufrufen `generator.ErzeugeNächsteID()`: bevor man in die Methode eintritt, wird `this` auf die Adresse von `generator` aktualisiert, danach läuft die Funktion durch und nach deren Abschluss erhält `this` die Adresse, des Objekts, das vorher referenziert wurde.

Man könnte jetzt natürlich fragen, warum überhaupt das Feld `_idCount` in der Klasse eingesetzt wurde, anstatt einfach in der Methode `ErzeugeNächsteID` eine Variable zu verwenden. Die Antwort darauf ist einfach: wie bereits in Abschnitt 4.3.3 beschrieben, sind Variablen nur in ihrer Funktionsscope gültig. Ist die Funktion vorbei, werden sie deallokiert, weswegen auch ihr letzter gültiger Wert verloren geht. Felder hingegen bleiben solange bestehen, bis das komplette Objekt deallokiert wird. Man kann Felder ähnlich betrachten wie globale Variablen in C, die aber eben nicht frei verfügbar, sondern einem bestimmten Objekt zugeordnet sind. Jede Instanz hat dabei ihr eigenes Feld und damit auch ihren eigenen Wert für dieses Feld. Bitte beachten Sie, dass das Feld dazu nicht mit dem Modifizierer `static` ausgezeichnet sein darf – was dieser genau macht, schauen wir uns im nächsten Abschnitt an.

5.6 Der Modifizierer static

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=U-kqutMKoBs>.

Jedes Mitglied einer Klasse kann mit dem Modifizierer `static` ausgestattet werden, was folgende Konsequenzen hat:

- Die Methode wird nicht mehr über ein Objekt, sondern über den Klassennamen aufgerufen, wie bspw. bei `Console.WriteLine` oder bei `Convert.ToInt32`. Hier ist jeweils das Türkis gedruckte Wort der Klassenname, mit dem Punktoperator kann man dann auf dessen statische Mitglieder zugreifen. Bitte beachten Sie, dass der Punktoperator hier nicht dereferenziert.
- Innerhalb einer statischen Methode kann die `this` Referenz nicht genutzt werden (eine statische Methode ist, wie eben ja erklärt, nicht an ein Objekt gebunden). Das hat zur Folge, dass man innerhalb einer statischen Methode auch nicht auf andere Instanzmitglieder der Klasse zugreifen kann.
- Als Instanzmitglieder bezeichnet man alle Members einer Klasse, die nicht `static` sind – sie können nur angesprochen werden, wenn man ein Objekt aus der Klasse erstellt und dieses nutzt.
- Auf statische Mitglieder kann jederzeit zugegriffen werden, auch in Instanzmethoden.
- Man kann auch eine Klasse mit dem Modifizierer `static` ausstatten. Dies bedeutet, dass sämtliche Mitglieder in der Klasse statisch sein müssen.

Statische Felder, die ebenfalls `public` sind, verhalten sich deshalb genauso wie globale Variablen in C: aus jedem beliebigen Kontext kann man auf diese zugreifen. Ähnlich ist es für Methoden, die mit `public` und `static` gekennzeichnet sind: sie verhalten sich wie normale Funktionen in C, die von überall her aufgerufen werden können. Statische Mitglieder haben allerdings einen erheblichen Nachteil: sie sind nicht mit dem Konzept der Polymorphie, einem wichtigen Grundsatz, der Code ungeheuer flexibel machen kann, vereinbar. Deswegen lassen Sie bitte die Finger von statischen Klassenmitgliedern und befolgen folgenden Grundsatz:

Statische Felder und Methoden sind globalen Variablen und freien Funktionen in C sehr ähnlich, haben allerdings einen großen Nachteil: sie unterstützen keine Polymorphie. Deswegen sollte jede neue Klasse so weit wie möglich mit Instanzmitgliedern ausgestattet werden, die über ein entsprechendes Objekt der Klasse genutzt werden können. Statische Mitglieder sollten wirklich nur in ganz bestimmten Situation Anwendung finden – und das so selten wie möglich.

Mit unserem jetzigen Wissen können wir noch nicht verstehen, warum Polymorphie flexibleren Code ermöglicht. Ich bitte sie trotzdem, den eben genannten Grundsatz zu beachten.

5.7 Kapselung – welche Mitglieder mache ich private, welche public?

Das Video zu diesem Abschnitt finden Sie unter https://www.youtube.com/watch?v=pxlyoxsbU_8.

Nachdem wir im vorigen Abschnitt geklärt haben, dass statische Mitglieder möglichst vermieden werden sollten, schauen wir uns jetzt das erste große Grundprinzip der objektorientierten Programmierung an: die Kapselung.

Dabei sollte man folgende Punkte zu diesem Grundprinzip zunächst beachten:

- Als **Kapselung** bezeichnet man die **Bündelung von Daten und Methoden**, die auf diesen Daten operieren. Dies kann mit Klassen gemacht werden: in ihnen können Felder und Methoden definiert werden, wobei die Felder innerhalb der Methoden genutzt werden.
- Ein Objekt dieser Klasse hat dabei **Hoheit über seine Daten**: es ist nicht möglich, die Werte von Feldern außerhalb des Klassenscopes einfach zu ändern. Dies ist nur durch einen kontrollierten Zugriff über Methoden möglich. Als Konsequenz heißt das: die Felder einer Klasse sollten immer als **private** gekennzeichnet werden. Über spezielle Methoden, die **public** gekennzeichnet sind, können Feldwerte gelesen bzw. geschrieben werden.
- Eine Klasse sollte immer **genau ein Teilproblem lösen**: durch das Aufrufen einer oder mehrerer Methoden können bestimmte Funktionalitäten ausgeführt werden. Um das jeweilige Ziel innerhalb einer Methode zu erreichen, kann das Objekt die Werte von Feldern und von mitgegebenen Parametern nutzen. Dabei sollte man darauf achten, dass der Nutzer der Klasse auch nur die Parameter angeben muss, die für ihn zur Erfüllung des Problems notwendig sind. Andere Daten, von denen die Klasse abhängt, sollten über Felder bezogen werden und für den Nutzer verborgen bleiben (**Information Hiding**). Dies sagt letztendlich nichts anderes aus, als dass die API einer Klasse möglichst einfach verständlich und leicht einsetzbar sein sollte für den Nutzer der Klasse.

Dazu schauen wir uns nochmals die Klasse **IDGenerator** an und überlegen, ob sie diesen drei Anforderungen an Kapselung genügt:

```

public class IDGenerator
{
    private int _idCount; Das private Feld kann nur innerhalb des  
Klassenscopes verändert werden  
(Datenhoheit)

    public int ErzeugeNächsteID()
    {
        _idCount++;
        return _idCount;
    }
}

```

Die public Methode kann vom Nutzer der Klasse aufgerufen werden, um das Teilproblem der Generierung einer ID zu lösen. Implementierungsdetails bleiben dabei vor dem Nutzer verborgen (Information Hiding)

Abbildung 51: Datenhoheit und Information Hiding des Kapselungsprinzips verdeutlicht

Wie in Abbildung 51 zu sehen ist, nutzt die Klasse ein privates Feld und hat damit Datenhoheit über dessen Wert. Über die Funktion `ErzeugeNächsteID` wird das Problem der ID-Generierung für den Nutzer der Klasse gelöst, dabei wird auf das eben erwähnte Feld zugegriffen. Das ist dieser Funktion beim Aufruf allerdings nicht anzusehen, weswegen die Algorithmus zur Erstellung der ID sowie die genutzten Werte dazu vor dem Nutzer verborgen bleiben.

Bitte beachten Sie, dass man als Felder (und Parameter) nicht nur primitive Wertetypen nutzen kann, sondern auch Referenzen auf andere Objekte, deren Funktionalität man zur Erfüllung der eigenen nutzt. Schauen Sie sich dazu folgendes Beispiel an:

```

public class LottozahlenGenerator
{
    private Random _zufallszahlenGenerator = new Random(); Privates Feld, das eine Referenz auf ein  
anderes Objekt hält (Datenhoheit)

    public int[] GeneriereLottozahlen() API für Nutzer dieser Klasse
    {
        int[] lottozahlen = new int[6];
        int anzahlEindeutigerLottozahlen; Einsatz der Funktionalität des Feldes

        while(anzahlEindeutigerLottozahlen < 6)
        {
            int neueZufallszahl = _zufallszahlenGenerator.Next(1, 50);
            if (ÜberprüfeObZahlInArrayVorhanden(lottozahlen, neueZufallszahl))
                continue;

            lottozahlen[anzahlEindeutigerLottozahlen] = neueZufallszahl;
            anzahlEindeutigerLottozahlen++;
        }

        return lottozahlen;
    }

    private bool ÜberprüfeObZahlInArrayVorhanden(int[] array, int zahl) Private Hilfsfunktion (Information Hiding)
    {
        foreach (int element in array)
        {
            if (element == zahl)
                return true;
        }
        return false;
    }
}

```

Abbildung 52: Kapselung am Beispiel eines Lottozahlengenerators

Die in Abbildung 52 zu sehende Klasse **LottozahlenGenerator** hat die Aufgabe, sechs eindeutige Zufallszahlen zwischen 1 und 49 zu generieren, diese in einem Array zu speichern und diesen Array letztendlich dem Nutzer der Klasse zurückzugeben. Die Klasse hat drei Mitglieder, allerdings ist nur die Methode **GeneriereLottozahlen** für den Nutzer der Klasse sichtbar. In dieser wird zunächst ein Array erstellt und eine Variable, in der die Anzahl zugewiesener Lottozahlen mitgezählt wird. Danach wird eine **while** Schleife gestartet: solange die Anzahl der Lottozahlen kleiner als sechs ist, soll zunächst eine neue Zufallszahl generiert werden. Dies passiert über die Methode **Next** der Klasse **Random**, von der ein Objekt über das Feld **_zufallszahlenGenerator** referenziert wird. Ihr wird per Parameter übergeben, dass sich die Zufallszahl innerhalb des Bereichs 1 bis exklusive 50 befinden soll. Danach wird eine Überprüfung gestartet, ob die generierte Zufallszahl im Array schon vorhanden ist (es könnte ja sein, dass einer der **Next** Aufrufe eine bereits existierende Zufallszahl zurückgibt). Ist dies nicht der Fall, wie die neue Zahl dem Array an der entsprechenden Stelle zugewiesen und im Anschluss der Wert der Zählvariable erhöht.

Auch in diesem Beispiel findet man Datenhoheit wieder, allerdings wird nicht einfach ein Ganzzahlwert wie im vorherigen Beispiel in einem Feld gespeichert, sondern ein anderes Objekt referenziert. Genauer genommen wird dieses Objekt bei Erstellung des Feldes mit dem Ausdruck **new Random()** initialisiert (dieser Ausdruck landet implizit im Konstruktor der Klasse **LottozahlenGenerator**, dazu später mehr). Innerhalb der Methode **GeneriereLottozahlen** wird die Funktionalität dieses Objekts eingesetzt, um neue Zufallszahlen zu generieren. Ebenfalls ist

die Methode ÜberprüfeObZahlInArrayVorhanden **private**, was allerdings nicht auf Datenhoheit, sondern auf Verbergen interner Funktionalität, also Information Hiding, hinausläuft.

Wir haben nun mehrere Beispiele gesehen, in denen Klassen jeweils eine bestimmte Aufgabe erfüllen und dabei die Grundrichtlinien der Kapselung einhalten. Ich möchte Ihnen noch folgende Ratschläge an die Hand geben, wenn Sie selber Klassen schreiben:

- Denken Sie daran, dass ihre Klasse erst dann Nutzen bringt, wenn an einer anderen Codestelle ihre Klasse zu einem Objekt instanziert und eingesetzt wird. Stellen Sie deshalb sicher, dass die API ihrer Klasse unmissverständlich und einfach zu nutzen ist. Konkret heißt das zunächst, dass der Nutzer möglichst wenige Parameter angeben muss, um die Methoden ihres Objekts zu nutzen. Das erhöht nicht nur die allgemeine Lesbarkeit von Code, sondern freut vor allem auch den Entwickler, der Ihre Klasse einsetzen möchte (oder muss).
- Behalten Sie Datenhoheit im Blick: auf die Daten, die Sie in ihren Methoden brauchen und die nicht per Parameter übergeben werden, haben nur Sie Zugriff innerhalb ihrer Klasse. Das heißt: alle Felder sind **private**. Wenn Nutzer ihrer Klasse auf diese Werte zugreifen sollen, dann nur über spezielle Methoden, die das Auslesen oder Schreiben von Werten ermöglichen. Das macht ihren Code insgesamt robuster, da es von außen nicht möglich ist, die Daten für ihre Operationen „aus Versehen“ zu ändern.
- Eine Klasse löst immer genau **ein Teilproblem (Single Responsibility Principle)**: gerade Programmieranfänger in der Objektorientierung tendieren dazu, große Klassen zu schreiben, die zu viel Funktionalität umfassen und damit unflexibel und schlecht wartbar werden. Bitte tun sie das nicht. Versuchen Sie, Klassen möglichst klein zu schneiden und achten Sie auch darauf, dass die Funktionen innerhalb der Klassen nicht zu lang werden. Häufig ist das ein Anzeichen für zu viel Funktionalität, die an genau einer Stelle implementiert liegt. Betrachten Sie bspw. nochmals die Klasse **LottozahlenGenerator** aus dem letzten Beispiel: diese Klasse hat keine Ahnung, wie man Zufallszahlen erstellt, sondern delegiert diese Arbeit an die Klasse **Random** weiter. **LottozahlenGenerator** weiß aber, wie man erstellte Zufallszahlen auf ihre Eindeutigkeit überprüft und zu einem Array hinzufügt. Als Faustregel gilt: lieber viele kleine Klassen als wenige große.

Kapselung beseitigt letztendlich das in der Programmiersprache C häufig auftretende Problem von Variablen, dessen Werte in mehreren Funktionen genutzt wurden: entweder hat man dies geschafft, indem man sie als Parameter mitgeliefert (was häufig die Parameterliste aufblättert und damit die Lesbarkeit und Wartbarkeit des Codes beeinträchtigt) oder als globale Variable zur Verfügung gestellt hat. Durch die freie Zugänglichkeit von globalen Variablen ist es aber auch zu Programmfehlern gekommen, wenn diese an der falschen Stelle zum falschen Zeitpunkt auf einen invaliden Wert gesetzt wurden. Da in objektorientierten Programmiersprachen Objekte so ausgestattet sind, dass sie nur kontrollierten Zugriff auf die Daten, die sie benötigen, zulassen, kann es nicht mehr zu einem fehlerhaften Zustand einer globalen Variable kommen, die von mehreren Funktionen genutzt wird. In der objektorientierten Programmierung ist jedes Objekt selbst für seinen Status verantwortlich.

5.8 Zugriff auf Felder über Eigenschaften gewähren

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=x5gvnxANJ8Q>.

Im letzten Kapitel haben wir uns ausgiebig mit dem objektorientierten Grundkonzept der Kapselung auseinandergesetzt – unter anderem mit dem Ergebnis, dass sämtliche Felder einer Klasse als **private** gekennzeichnet werden sollten, damit das letztendliche Objekt Hoheit über diese Daten hat. Um dem Nutzer dennoch eine gewisse Bemächtigung auf diese Daten zu gewähren, können Methoden eingesetzt werden, die lesenden oder schreibenden Zugriff gewähren. Dazu sehen wir uns

das folgende modifizierte Beispiel der Klasse `Person` an, die wir schon vorher einmal betrachtet haben:

```
public class Person
{
    private string _vorname;
    private string _nachname; ← Felder sind private
                                (Datenhoheit)

    public string GetVorname() ← Getter Methoden geben
    {                                jeweils Wert eines Felds zurück
        return _vorname;
    }

    public void SetVorname(string vorname) ← Setter Methoden setzen jeweils Wert
    {                                eines Felds nach Überprüfung
        if (vorname == null)
            return;
        _vorname = vorname;
    }

    public string GetNachname() ←
    {                                Getter Methoden geben
        return _nachname;          jeweils Wert eines Felds zurück
    }

    public void SetNachname(string nachname) ←
    {                                Setter Methoden setzen jeweils Wert
        if (nachname == null)
            return;
        _nachname = nachname;
    }
}
```

Abbildung 53: Getter und Setter Methoden für Felder

In Abbildung 53 sieht man, dass die `Person` Klasse im Gegensatz zu vorher `private` Felder hat. Damit Nutzer die Werte dieser dennoch ändern können, werden die Methoden `SetVorname` und `SetNachname` bereitgestellt, die jeweils einen Parameter entgegen nehmen. Innerhalb dieser Methoden wird der Parameter auf `null` überprüft und, falls dies nicht der Fall ist, dem entsprechenden Feld zugewiesen. Damit wird verhindert, dass das Objekt falsche Werte enthält. Um die gesetzten Werte wieder auszulesen, existieren die Methoden `GetVorname` und `GetNachname`, die den Wert des entsprechenden Feldes zurückgeben.

Die folgende Abbildung zeigt, wie die neue Klasse `Person` angewendet werden kann.

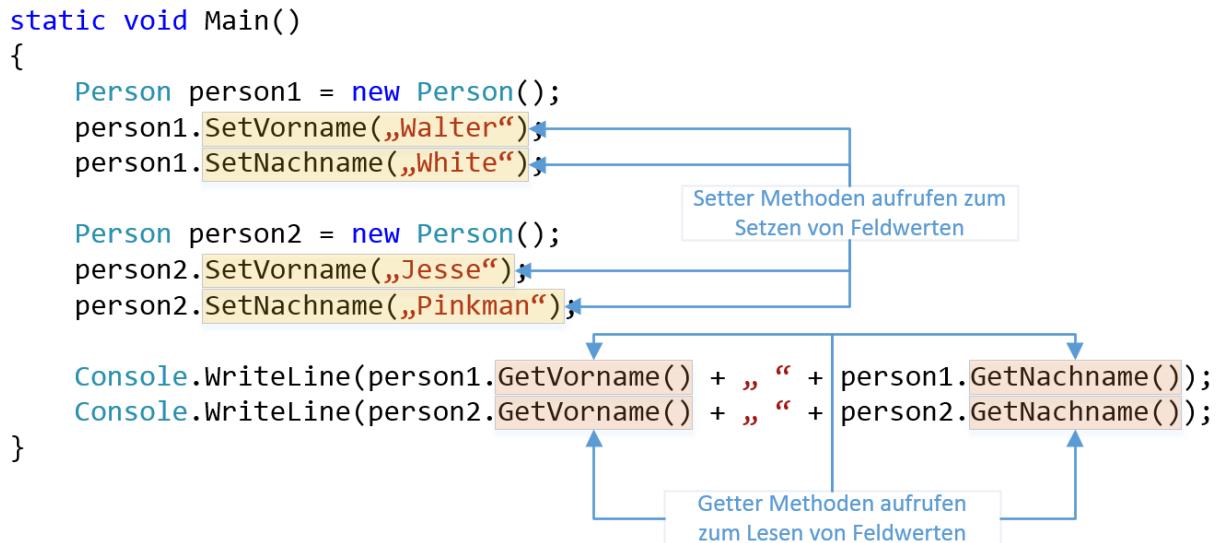


Abbildung 54: Getter und Setter Methoden aufrufen

In Abbildung 54 sieht man, wie die Set-Methoden genutzt werden, um dem Feldern des Objekts einen Wert zuzuweisen, und die Get-Methoden, um den gesetzten Wert wiederum auszulesen. Deshalb kann man auch folgende verallgemeinernde Aussage treffen: zu jedem Feld, das **private** ist, wird...

- ...eine dazugehörige **public** Get-Methode erstellt, um den Nutzer lesenden Zugriff auf den Feldwert zu gestatten.
- ...eine dazugehörige **public** Set-Methode erstellt, um den Nutzer schreibenden Zugriff auf den Feldwert zu gestatten. Innerhalb dieser Set-Methode sollten neben der Wertzuweisung zum Feld auch zusätzliche Überprüfungen durchgeführt werden.

Offensichtlich bilden die optionalen Get- und Set-Methoden eine Einheit, die Zugriff auf ein Feld gewährt. Vergessen Sie bitte jedoch sofort die Beispiele, die ich Ihnen in den letzten beiden Abbildungen suggeriert habe, denn in C# gibt es das Konzept der Eigenschaften (engl. Properties), mit denen dieselbe Funktionalität mit weniger Code implementiert werden kann, wie in folgender Abbildung zu sehen ist: in dieser ist die Klasse Person nochmals abgebildet, allerdings hat sie statt vier Get- und Set-Methoden zwei Eigenschaften, die wie gewohnt farbig markiert sind. Dabei ist der Kopf einer Eigenschaft wie schon gewohnt mit Modifizierer, Eigenschaftstyp und Eigenschaftsname deklariert, nachdem zwei geschweifte Klammern folgen, in denen die **get** und **set** Methoden enthalten sind. Dabei sind diese Methoden nicht der normalen Methodensyntax unterworfen, sondern werden nur mit den eben genannten Schlüsselworten definiert, gefolgt von geschweiften Klammern, die den Funktionsscope vorgeben. Dabei muss folgendes beachtet werden:

- In der **get** Methode muss ein Wert mit **return** zurückgegeben werden, der zum Eigenschaftstyp passt.
- In der **set** Methode kann mit dem Schlüsselwort **value** auf den Wert zugegriffen werden, der dem Feld zugewiesen werden soll.

Ansonsten verhalten sich diese Methoden genauso wie andere Methoden auch, d.h. man kann in ihren Funktionsscopes beliebig viele sequentielle Anweisungen schreiben (obwohl man sich natürlich auf das Zurückgeben oder Setzen der Feldwerte beschränken sollte).

```

public class Person
{
    private string _vorname;
    private string _nachname;
    Eigenschaftstyp

    Modifizierer public string Vorname Bezeichner der Eigenschaft
    {
        get { return _vorname; } Get Methode (optional)
        set
        {
            if (value == null)
                return;
            _vorname = value;
        }
    Eigenschaftstyp

    Modifizierer public string Nachname Bezeichner der Eigenschaft
    {
        get { return _nachname; } Get Methode (optional)
        set
        {
            if (value == null)
                return;
            _nachname = value;
        }
    }
}

```

Abbildung 55: Eigenschaften in C# im Einsatz

Nachdem Sie nun wissen, wie man Eigenschaften definiert, bleibt noch die Frage, wie man sie als Nutzer der Klasse aufrufen kann. Sehen Sie sich dieses abgewandelte Beispiel von Abbildung 54 an:

```

static void Main()
{
    Person person1 = new Person();
    person1.Vorname = „Walter“;
    person1.Nachname = „White“;

    Person person2 = new Person();
    person2.Vorname = „Jesse“;
    person2.Nachname = „Pinkman“;

    Console.WriteLine(person1.Vorname + „, “ + person1.Nachname);
    Console.WriteLine(person2.Vorname + „, “ + person2.Nachname);
}

```

Abbildung 56: Get- und Set-Methoden einer Eigenschaft aufrufen

In Abbildung 56 sehen Sie, dass man die `set` Methode einer Eigenschaft aufruft, indem man erst das dazugehörige Objekt mit dem Punktoperator dereferenziert und dann über den Namen der Eigenschaft eine Zuweisung macht. Der zugewiesene Wert wird dann implizit in `value` geschrieben, sodass man innerhalb der `set` Methode über dieses Schlüsselwort auf den Wert zugreifen kann. Auf die `get` Methode einer Eigenschaft greift man ebenfalls nur über den Eigenschaftsnamen zu. Dabei kann man sich folgenden Grundsatz merken:

Eigenschaften in C# vereinen eine Get- und eine Set-Methode unter einem Bezeichner. Dabei kann eine der Methoden optional sein und folglich weggelassen werden. Für den Nutzer der Klasse unterscheidet sich die Nutzung einer Eigenschaft jedoch nicht von der Nutzung eines Feldes: über den Bezeichner lassen sich einfach der dahinterliegende Wert auslesen oder setzen (über den Zuweisungsoperator).

Leider besitzt nur C# das Konzept der Eigenschaft. Bei anderen Programmiersprachen wie bspw. Java und C++ müssen deshalb, wie in Abbildung 53 und Abbildung 54 gezeigt, entsprechende Getter und Setter Methoden geschrieben werden. In C# verwenden wir jedoch ab jetzt konsequent Eigenschaften.

5.9 Namensräume

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=BUzS4YEI7io>.

Namensräume (engl. Namespaces) sind ein Konzept, dass Klassen und anderen Typen einen eindeutigen Namen verschafft. Wir haben bereits in vorherigen Codebeispielen gesehen, dass sich die von Visual Studio automatisch erstellte Klasse `Program` und die von uns erstellten Klassen im Scope eines ebenfalls von Visual Studio erstellten Namespaces befanden, wussten aber noch nicht um dessen Bedeutung. In folgender Abbildung sehen wir uns die Bestandteile eines Namensraums bei seiner Definition an:

```

namespace
Schlüsselwort      namespace HelloWorld      Bezeichner des Namespace
{
    public class Person
    {
        // Code der Klasse Person hier
        // nicht angegeben
    }
}

```

Abbildung 57: Namensraumdefinition

Wenn man einen Namensraum manuell erstellt, muss man zunächst das Schlüsselwort `namespace` anführen, gefolgt vom Bezeichner des Namespaces. Im Anschluss folgen zwei geschweifte Klammern, in denen Klassen oder andere Typdefinitionen vorgenommen werden können. Dies hat dann Auswirkungen auf den Namen des Typs: der sog. vollqualifizierte Name (engl. full qualified name) der in Klasse in Abbildung 57 ist `HelloWorld.Person`, also die Zusammensetzung aus Namespace und Klassenbezeichner, wobei beide mit einem Punkt voneinander getrennt sind. Dies macht man, um Namensübereinstimmungen von Klassen, die von unterschiedlichen Programmierern stammen,

auflösen zu können (unter der Annahme, dass diese Klassen in unterschiedlichen Namensräumen liegen).

Folgende Dinge müssen bei Namensräumen beachtet werden:

- Innerhalb eines Namensraums muss der Name einer Klasse oder anderer Typen eindeutig sein. Wenn mehrere Klassen im selben Namenraum den gleichen Bezeichner haben, gibt der Compiler einen Fehler aus.
- Alle Typen, die innerhalb eines Namespaces definiert werden, müssen nicht in einer Datei liegen, sondern können über mehrere Dateien verteilt werden. Dazu macht man in der jeweiligen Datei einen entsprechenden Namensraum mit demselben Bezeichner auf.
- Es ist zwar möglich, Namespaces zu schachteln, d.h. einen weiteren Namensraum innerhalb des Scopes eines anderen zu erstellen, allerdings ist das nicht die übliche Herangehensweise. Wenn man Unternamensräume erstellen möchte, schließt man einfach den dazugehörigen Obernamensraum mit in den Bezeichner mit ein, abgetrennt durch einen Punkt. So würden bspw. alle Klassen, die zum Unternamenraum „Persistenz“ gehören sollen, in folgendem Namespace definiert werden:
`namespace HelloWorld.Persistenz { }`
- Es ist auch möglich, Typen außerhalb des Scopes eines Namensraums zu definieren, allerdings rate ich Ihnen auch hier sehr davon ab. Ihre Klassen sind dann implizit dem sog. globalen Namensraum zugeordnet, wodurch die ganzen Vorteile von Namespaces ausgesetzt werden und es wieder leichter zu Überschneidungen bei Bezeichnern unterschiedlicher Programmierer kommt.

Im Allgemeinen sollte man darauf achten, dass die Namensräume, zu denen die verschiedenen Typen gehören, sich in der Projekt- und Projektunterordnerstruktur widerspiegeln, die Sie im Solution Explorer angelegt haben. Des Weiteren empfehle ich Ihnen, pro Datei in Ihrem Projekt genau einen Typ zu definieren.

Auch das .NET Framework nutzt zur Strukturierung Namensräume: der wichtigste ist dabei der **Namespace System**. In ihm liegen nicht nur alle primitiven Datentypen, sondern auch viele andere Grundfunktionalitäten, die häufig genutzt werden. In der MSDN Library können Sie deshalb nicht nur Dokumentationen zu einzelnen Typen finden, sondern auch von Namensräumen, wie bspw. hier zu **System**: [http://msdn.microsoft.com/en-us/library/system\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/system(v=vs.110).aspx)

Als Abschluss dieses Abschnitts zeige ich Ihnen noch, wie man die vollqualifizierten Namen von Klassen mit **using** Direktiven leichter auflösen kann (denn offensichtlich haben wir auf unsere Klassen bisher auch zugegriffen, ohne dabei den vollqualifizierten Namen anzugeben). Dazu betrachten wir folgendes Beispiel:

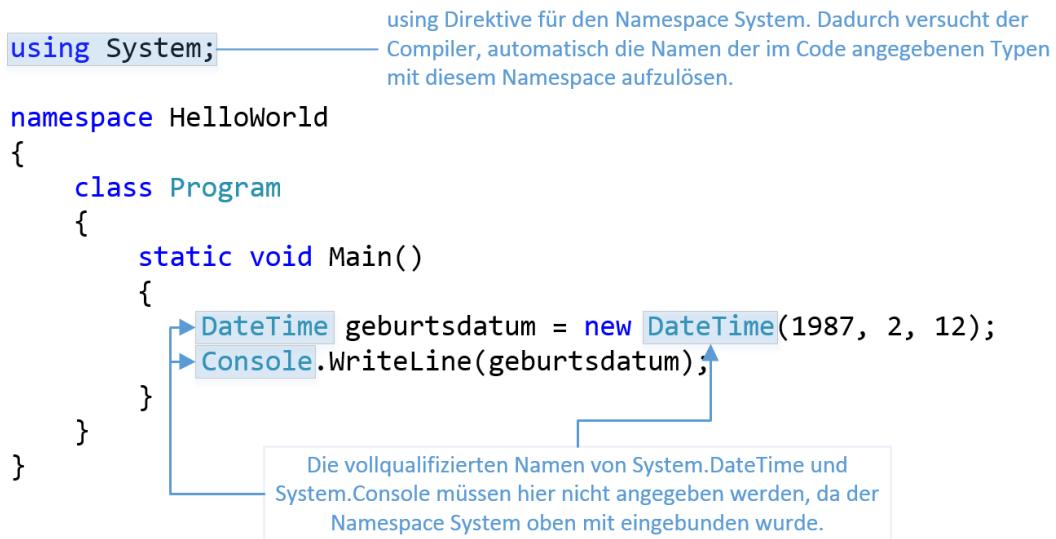


Abbildung 58: Using Direktive bindet den Namensraum System des .NET Frameworks mit ein

In Abbildung 58 sehen Sie, wie zu Beginn mit dem Schlüsselwort **using** gefolgt von **System** und einem Semikolon der gleichnamige Namensraum eingebunden wird. Eingebunden heißt in diesem Fall, dass der Compiler beim Erstellvorgang versucht, Typenangaben wie **DateTime** und **Console** (deren vollqualifizierter Name jeweils **System.DateTime** und **System.Console** ist) im oberen Beispiel durch die Einbeziehung des Namensraum **System** aufzulösen. Dabei müssen sie keine **using** Direktiven angeben, wenn Sie Typen einsetzen, die sich im selben Namensraum befinden wie der aktuelle Code. Man kann auch mehrere **using** Direktiven zu Beginn einer Datei angeben, um mehrere Namensräume anzugeben, dann darf es aber innerhalb des Codes nicht zu Namensüberschneidungen kommen. In diesem Fall müssen Sie auch im Code den vollqualifizierten Namen eines Typs angeben oder andere Möglichkeiten anwenden, die im Video oder in der MSDN Library unter <http://msdn.microsoft.com/de-de/library/sf0df423.aspx> beschrieben sind.

Als Faustregel gilt: verwenden Sie **using** Direktiven. Nur in Ausnahmefällen sollten Sie den vollqualifizierten Namen eines Typs im Code angeben müssen.

5.10 Konstruktoren

Das Video zu den nächsten vier Unterabschnitten finden Sie unter

<https://www.youtube.com/watch?v=AO4HNZuY8ms>.

5.10.1 Grundsätzliches zu Konstruktoren

Wir haben in mehreren vorigen Abschnitten bereits die Existenz von Konstruktoren angesprochen, in diesem Abschnitt schauen wir uns genauer an, wie diese aufgebaut sind und wie man sie einsetzt.

Konstruktoren sind genauso wie die **get** und **set** Methoden in Eigenschaften spezielle Methoden, die letztendlich Klassenmitglieder repräsentieren. Dabei hat jede Klasse einen Standardkonstruktor, der automatisch vom Compiler erstellt wird, wenn er nicht explizit in der Klasse angegeben wird. In Abbildung 59 sehen Sie eine modifizierte Version der Klasse **Person**, in der der Standardkonstruktor explizit aufgeführt wird als erstes Member der Klasse. Wir sehen bei diesem Konstruktor einige Besonderheiten im Vergleich zu normalen Methoden:

- Konstruktoren haben keinen Rückgabetypen (auch nicht **void**).
- Ihr Bezeichner bzw. Name muss identisch sein mit dem Klassennamen, in dem sie stehen.

- Sie besitzen einen Modifizierer, allerdings funktioniert hier keine Kombinationen mehrerer Modifizierer wie bspw. `public static`. Der Modifizierer muss entweder `public`, `private` oder `static` sein.
- Innerhalb des Funktionsscopes gelten die gleichen Regeln wie für normale Methoden. Üblicherweise nutzt man allerdings in Konstruktoren nur Code, der Felder initialisiert, entweder mit vorgegebenen Werten oder, was eher der Fall ist, mit Parametern des Konstruktor.

```

Konstruktorbezeichner
(muss gleich dem Klassenbezeichner sein)

public class Person
{
    Modifizierer public Person()
    {
        Funktionsscope {
    }
}

private string _name;

public string Name
{
    get { return _name; }
    set { if (value != null) _name = value; }
}
}

```

Abbildung 59: Standardkonstruktor in Klasse Person

Wie man am Standardkonstruktor auch sehen kann, enthält er standardmäßig keinen Code. Im folgenden Beispiel ändern wir den Konstruktor nun ab, indem wir ihm einen Parameter hinzufügen.

```

Konstruktorbezeichner

public class Person
{
    Modifizierer public Person(string name)
    {
        Funktionsscope {
            if (name != null)
                _name = name;
        }
    }

    private string _name;

    public string Name
    {
        get { return _name; }
        set { if (value != null) _name = value; }
    }
}

```

Abbildung 60: Parametrierter Konstruktor in Klasse Person

In Abbildung 60 sehen Sie, wie der Konstruktor jetzt den Parameter `name` entgegen nimmt. Im Code wird dieser auf `null` überprüft und falls dies nicht zutrifft, dem Feld `_name` zugewiesen. Bitte beachten Sie, dass es sich in diesem Fall nicht mehr um den Standardkonstruktor handelt: dieser ist

nämlich immer der, der keine Parameter entgegen nimmt. Eine Klasse kann beliebig viele Konstruktoren haben, wobei wie bei normalen Methoden die Signatur für die Eindeutigkeit eines Konstruktor entscheidend ist. Ich würde Ihnen jedoch den Rat mit auf den Weg geben, dass Sie versuchen sollten, genau einen Konstruktor pro Klasse anzustreben. Wichtig ist ebenfalls folgenden Fakt: wenn Sie einer Klasse einen Konstruktor mit Parameter hinzufügen, erstellt der Compiler keinen Standardkonstruktor für diese Klasse beim Kompilieren. Wenn Ihre Klasse dann ebenso mit einem Standardkonstruktor ausgestattet sein soll, müssen Sie diesen explizit hinzufügen.

Wenn wir die modifizierte Klasse Person aus Abbildung 60 instanziieren und nutzen möchten (also die Version ohne Standardkonstruktor), ändert sich dabei der Einsatz des `new` Operators wie folgt:

```

static void Main()
{
    Person meinePerson = new Person("Walter White");
    Console.WriteLine(meinePerson.Name);
}
```

Parameterangabe bei
Konstruktorauf调用 notwendig

Abbildung 61: Parametisierte Konstruktoren aufrufen

In Abbildung 61 sehen Sie, wie nach dem `new` Operator weiterhin der Name der Klasse steht, die man instanzieren möchte, jedoch muss man innerhalb der Klammern jetzt den geforderten Parameter des Konstruktor angeben. Und genau hier wird verdeutlicht, was der Vorgang der Instanzierung eines Objekts genau ist: der `new` Operator sorgt dafür, dass ein Speicherbereich passend zur verlangten Größe des Typs Person allokiert wird. Im Anschluss wird der entsprechende Konstruktor aufgerufen, dessen Aufgabe es im Normalfall ist, die Werte der Felder des Objekts zu initialisieren (im Allgemeinen: sämtlichen Code ausführen, der zur Initialisierung eines Objekts notwendig ist). Nachdem dieser Prozess abgeschlossen ist, wird die Adresse des neuen Objekts zurückgegeben und kann in eine Variable, im oberen Beispiel `meinePerson`, gespeichert werden.

5.10.2 Inline-Initialisierung von Feldern

Sie können in C# Felder nicht nur in Konstruktoren, sondern auch inline in einer Anweisung nach der Felddefinition. Wir haben dies auch schon einmal beim Beispiel des Lottozahlengenerators gemacht, ohne dies explizit zu erwähnen:

```

public class LottozahlenGenerator
{
    private Random _zufallszahlenGenerator = new Random();

    // Weitere Code ist hier aus Platzgründen entfernt worden
}
```

Inline-Initialisierung eines Felds

Abbildung 62: Inline-Initialisierung eines Felds

In Abbildung 62 sehen Sie eine auf das wesentliche gekürzte Variante der Klasse `LottozahlenGenerator`. In ihr wird das Feld `_zufallszahlenGenerator` definiert und sofort initialisiert, indem man nach dem Feldbezeichner den Zuweisungsoperator mit dem Initialisierungsausdruck angibt. Erst danach folgt das Semikolon zum Abschluss der Felddefinition. Bitte beachten Sie, dass der Inline-Initialisierung eine Anweisung mit beliebig komplexer Ausdrucksschachtelung sein kann, allerdings eben nur genau eine Anweisung. Wenn Sie mehrere

Anweisungen benötigen, um ein Feld zu initialisieren, müssen Sie dies im Funktionsscope eines Konstruktors tun.

Die Inline-Initialisierungen werden vom Compiler in die jeweiligen Konstruktoren verschoben, und zwar nach folgenden Regeln:

- Wenn kein Konstruktor in der Klasse existiert, wird die Anweisung in den vom Compiler implizit erstellten Standardkonstruktor verschoben.
- Wenn ein oder mehrere Konstruktoren in der Klasse existieren, dann verschiebt der Compiler die Anweisung in alle diese.

Folglich kann man sich sicher sein, dass eine Inline-Initialisierung in jedem Fall ausgeführt wird, egal welchen Konstruktor man auuft.

5.10.3 Statische Konstruktoren

Wir haben bereits im vorletzten Abschnitt erklärt, dass man bei einem Konstruktor keine Modifizierer-Kombination wie bspw. `public static` angeben darf. Der Grund dafür ist die besondere Stellung des sog. statischen Konstruktors, den sie im nächsten Beispiel kennen lernen:

```
public class Person
{
    static Person()
    {
        _anzahlInstanzen = 0;
    }

    private static int _anzahlErstellteInstanzen;
    public static int AnzahlErstellteInstanzen
    {
        get { return _anzahlErstellteInstanzen; }
    }

    public Person(string name)
    {
        _anzahlInstanzen++;
        _name = name;
    }

    private string _name;
    public string Name
    {
        get { return _name; }
    }
}
```

Abbildung 63: Statischer Konstruktor in Klasse Person

In Abbildung 63 wurde die bereits bekannt Klasse `Person` wieder einmal abgeändert zu einer Funktionalität, mit der automatisch die Anzahl der erstellten Instanzen dieser Klasse mitgezählt werden. Dazu sieht man als erstes Klassenmitglied den statischen Konstruktor, der das statische Feld

`_anzahlErstellteInstanzen` initialisiert. Bei statischen Konstruktoren muss man folgende Regeln beachten:

- Sie dürfen nur mit dem Modifizierer `static` gekennzeichnet sein
- Sie dürfen kein Parameter entgegen nehmen
- In ihrem Scope kann nur auf statische Mitglieder der Klasse zugegriffen werden (wie bei anderen statischen Methoden nicht auf Instanzmember, also keine `this` Referenz verfügbar)

Der Grund, warum der statische Konstruktor keine Parameter entgegen nehmen darf, ist, dass er nicht explizit aufgerufen werden kann, sondern nur ein einziges Mal implizit aufgerufen wird, sobald die Klasse im Code genutzt wird.

```
static void Main()
{
    Person walter = new Person("Walter White");
}
```

Bei der ersten Nutzung einer Klasse wird automatisch der statische Konstruktor aufgerufen (falls vorhanden)

Abbildung 64: Impliziter Aufruf des statischen Konstruktors bei Erstgebrauch einer Klasse

Neben dem statischen Konstruktor ist der Code der letzten beiden Abbildungen auch nochmals ein Beispiel für den Unterschied zwischen statischen Feldern und Instanzfeldern. Der Speicherbereich für ein statisches Feld wird genau einmal erstellt (und eben nicht pro Instanz einmal) und wenn man aus einer Instanzmethode darauf zugreift wie im obigen Beispiel aus dem Konstruktor, wird letztendlich nur derselbe Speicherbereich adressiert (die Instanzen teilen sich sozusagen das Feld). Dies wird auch nochmal mit folgendem Code deutlich:

```
static void Main()
{
    Person walter = new Person("Walter White");
    Person jesse = new Person("Jesse Pinkman");
    Person hank = new Person("Hank Schrader");

    Console.WriteLine(Person.AnzahlErstellteInstanzen);
}
```

Anzahl erstellte Instanzen ist 3, da das dazugehörige Feld statisch ist und innerhalb der Konstruktorenaufrufe jeweils inkrementiert werden

Abbildung 65: Unterschied zwischen statischen und Instanzfeldern

Wie allerdings schon im Abschnitt „5.6 Der Modifizierer static“ erwähnt, rate ich Ihnen vom umfangreichen Einsatz statischer Klassenmitglieder ab, da sie keine Polymorphie ermöglichen.

5.10.4 Der Modifizierer `readonly`

Felder können mit dem Modifizierer `readonly` ausgestattet werden, der aussagt, dass die entsprechenden Felder nur im Konstruktor oder inline gesetzt werden können. Dies macht man häufig, um sicherzugehen, dass sich ein Feldwert während der Lebenszeit des Objektes nicht mehr ändert.

```

public class LottozahlenGenerator
{
    private readonly Random _zufallszahlenGenerator = new Random();

    // Weitere Code ist hier aus Platzgründen entfernt worden
}



Mit readonly geht man sicher, dass ein Feldwert ausschließlich im Konstruktor gesetzt werden kann


```

Abbildung 66: Feld mit Modifizierer `readonly`

In Abbildung 66 sehen Sie eine modifizierte Variante der Klasse `LottozahlenGenerator`. Das Feld ist nun zusätzlich mit dem Modifizierer `readonly` ausgestattet, sodass dieses Feld nicht außerhalb eines Konstruktors gesetzt werden kann. Ich persönlich verwende `readonly` bei Feldern, wo es nur geht, um den Lesern meiner Klasse die kognitive Last wegzunehmen (immerhin ist das ja eine prägnante Aussage, dass ein Feldwert sich zur Lebenszeit nicht verändern kann). Und nebenbei: natürlich können auch statische Felder mit `readonly` gekennzeichnet werden.

5.10.5 Konstruktoraufrufe verketten

Das Video zu diesem und zum nächsten Abschnitt finden Sie unter <https://www.youtube.com/watch?v=o8eJFPTU77U>.

Wie schon in den vorherigen Abschnitten erwähnt, kann man einer Klasse mehrere Konstruktoren spendieren, deren Parameterliste natürlich jeweils unterschiedlich sein muss. Des Weiteren kann man beim Aufruf eines Konstruktors einen anderen Konstruktor aufrufen – das macht man häufig, um für einen der Parameter des Zielkonstruktors einen Standardwert bereitzustellen. Dieses Aufrufen eines Konstruktors aus dem anderen heraus nennt man Verkettung von Konstruktoraufrufen.

```

public class LottozahlenGenerator
{
    private readonly Random _zufallszahlenGenerator;
    Doppelpunkt nach Parameter
    public LottozahlenGenerator(): this(new Random()) { }

    public LottozahlenGenerator(Random zufallszahlenGenerator)
    {
        if (zufallszahlenGenerator == null)
            return;
        _zufallszahlenGenerator = zufallszahlenGenerator
    }

    // Weitere Code ist hier aus Platzgründen entfernt worden
}



this Schlüsselwort zum Aufruf eines anderen Konstruktors innerhalb der Klasse



Anhand der angegebenen Parameter wird der Zielkonstruktor bestimmt


```

Abbildung 67: Konstruktorkettung anwenden

In Abbildung 67 sehen Sie eine modifizierte Version der Klasse `LottozahlenGenerator`. Sie besitzt wie gehabt das `readonly` Feld `_zufallszahlenGenerator`, welches aber nicht inline, wie im vorherigen Beispiel, sondern innerhalb des Konstruktors, der einen Parameter entgegennimmt, initialisiert wird. Zusätzlich findet sich ein Standardkonstruktor in der Klasse, bei dem der parametrisierte Konstruktor aufgerufen wird. Dies wird gemacht, indem man nach den Klammern für die Parameter im Funktionskopf einen Doppelpunkt schreibt gefolgt vom Schlüsselwort `this`: damit

ruft man einen anderen Konstruktor innerhalb der Klasse auf. Welcher Konstruktor das genau ist, wird anhand der Parameter, die man dem Aufruf übergibt, entschieden. Bitte beachten Sie, dass man mit dem `this` Schlüsselwort hier nicht auf alle Mitglieder des Objekts zugreifen kann wie in einem Funktionsscope, sondern nur auf die Konstruktoren der Klasse.

Konstruktorverkettung wende ich ab und zu im realen Programmieralltag an, jedoch sollte man versuchen, nur einen Konstruktor für Klassen zu schreiben. Es gibt andere Design Patterns wie das Factory Pattern und das Builder Pattern, die sich mit dem Erstellen von Objekten mit Parametern kümmern, die man in solchen Situation besser anwenden kann. Was Design Patterns genau sind, schauen wir uns in einem späteren Kapitel genauer an.

5.10.6 Objektinitialisierungssyntax für Konstruktoraufrufe

In C# gibt es eine weitere Möglichkeit, etwas Code zu sparen, wenn Objekte über einen Konstruktoraufruf erstellt werden und ihnen gleichzeitig Feldwerte des neuen Objekts initialisiert werden sollen. Dazu verwenden wir folgende Beispielklasse:

```
public class Beispielklasse
{
    public string WertA;

    public int WertB { get; set; }
}
```

Syntax für automatische Eigenschaften
(dazugehöriges Feld wird vom Compiler erstellt)

Abbildung 68: Beispielklasse mit automatischer Eigenschaft

Die Klasse aus Abbildung 68 hat ein Feld und eine Eigenschaft, wobei letztere in einem besonderen Syntax geschrieben ist, nämlich dem für automatische Eigenschaften. Innerhalb des Scopes der Eigenschaften werden die `get` und `set` Methode nicht ausformuliert, sondern nur mit einem Semikolon abgeschlossen. Der Compiler erstellt in diesem Fall automatisch ein Feld für diese Eigenschaft, zugreifen kann man auf dieses aber nur über die automatische Eigenschaft (ein Direktzugriff ist nicht möglich). Die `get` und `set` Methoden werden jeweils mit einer Standardimplementierung ausgestattet, die jeweils eine Anweisung lang sind: in ersterer wird der Feldwert zurückgegeben, in letzterer wird der Wert auf das Feld gesetzt. Damit ist eine automatische Eigenschaft gleichbedeutend mit der folgenden Definition:

```
private int _wertB;
public int WertB
{
    get { return _wertB; }
    set { _wertB = value; }
}
```

Wie man sieht, hat das nicht wirklich etwas mit Kapselung zu tun, da beim Setzen des Feldwertes der Wert nicht überprüft wird. Deswegen sollte man automatische Eigenschaften auch in Klassen, die den objektorientierten Paradigmen folgen, nicht einsetzen. Bei Datenklassen, die nur Werte zusammenfassen (die also keine Methoden zu diesen Daten anbieten), kann man automatische Eigenschaften aber durchaus nutzen.

Im folgenden Beispiel können Sie sehen, wie Sie diese Klasse instanzieren und die Felder des neuen Objekts mit Werten belegen können, einmal in konventioneller Weise und einmal mit der sog. Objektinitialisierungssyntax:

```

static void Main()
{
    Beispielklasse objekt1 = new Beispielklasse();
    objekt1.WertA = „Foo“;
    objekt1.WertB = 42;

    Beispielklasse objekt2 = new Beispielklasse()
    {
        WertA = „Foo“,
        WertB = 42
    };
}

```

Abbildung 69: Objektinitialisierungssyntax im Einsatz

In Abbildung 69 ist die Objektinitialisierungssyntax beim Erstellen von `objekt2` zu sehen: nach den Parameterklammern für den Konstruktorauftrag, der wie gewohnt vonstattengeht, wird ein weiteres Paar geschweifte Klammern eingesetzt, in denen man Eigenschaften oder Feldern, die `public` sind, Werte zuweisen kann. Die einzelnen Ausdrücke für die einzelnen Zuweisungen sind dabei voneinander mit einem Komma getrennt. Nach der geschweiften Klammer folgt noch ein Semikolon, das den Abschluss der Anweisung darstellt. Mit der Objektinitialisierungssyntax ist es also möglich, ein Objekt zu instanzieren und in derselben Anweisung dessen Felder mit Werten zu befüllen. Bitte beachten Sie auch, dass bei dieser Syntax die Parameterklammern weggelassen werden können, wenn der Standardkonstruktor aufgerufen wird.

Bitte vergleichen Sie diese Syntax auch mit der Syntax zur Arrayinitialisierung, die wir im Abschnitt 4.3.7 kennengelernt haben, da beide sich sehr ähnlich sind. Die Objektinitialisierungssyntax werde ich in den kommenden Abschnitten häufiger verwenden.

5.10.7 Destruktoren

Wie wir in den letzten Abschnitten gesehen haben, werden Konstruktoren als spezielle Methoden genutzt, die bestimmten Code bei der Instanziierung eines Objekts (bzw. der statischen Mitglieder beim statischen Konstruktor) ausführen, üblicherweise eben das Zuweisen von Werten an Felder. Ein Destruktor sitzt am anderen Ende der Lebenszeit eines Objekts: diese spezielle Methode wird aufgerufen, wenn der Speicher für ein Objekt deallokiert wird. Im nächsten Beispiel sehen wir die Definition eines Destruktors:

```

public class Beispielklasse
{
    ~Beispielklasse() — Destruktor, Tildezeichen vorangestellt,
                        keine Parameter erlaubt
    {
        // Code zur Freigabe von nativen Ressourcen hier
    }
}

```

Abbildung 70: Definition eines Destruktor in C#

Wie in Abbildung 70 zu sehen ist, unterscheidet sich die Definition eines Destruktors im Vergleich zu Konstruktoren nochmals. Dabei müssen folgende Regeln eingehalten werden:

- Ein Destruktor hat keine Modifizierer (somit gibt es auch keine Unterscheidung zwischen Instanz- und statischen Destruktor, ein Destruktor arbeitet immer auf einer Instanz).
- Der Destruktor hat wie der Konstruktor keinen Rückgabetypen (auch nicht `void`).
- Der Bezeichner der Funktion ist wie der Name der Klasse, in der er liegt, mit einem vorangestellten ~ (Tilde Zeichen).
- Der Destruktor darf keine Parameter entgegen nehmen.
- Folglich kann jede Klasse auch nur maximal einen Destruktor haben.

Innerhalb eines Destruktors sollte nur Code stehen, der mit der Freigabe von nativen Ressourcen zu tun hat. Dazu muss man erwähnen, dass C# Source Code nicht zu direkt ausführbaren (nativen) Maschinencode, sondern in eine sog. Zwischensprache (engl. Intermediate Language) kompiliert wird, die bei C# MSIL (Microsoft Intermediate Language) heißt. Dieser Code ist nicht direkt ausführbar, sondern wird in der sog. Common Language Runtime (CLR) ausgeführt. Diese Laufzeit ist ein Prozess, der einige interessanten Eigenschaften bereitstellt, bspw. automatische Speicherverwaltung über den sog. Garbage Collector (GC), der Teil der CLR ist. Ebenfalls übersetzt diese Laufzeit den MSIL Code direkt in Maschinenanweisungen. Mithilfe dieser beiden Eigenschaften ist es für Programmierer nicht notwendig, Objekte, die auf dem Heap der CLR erstellt wurden, selbstständig zu deallozieren (in C war dazu der Aufruf der Funktion `free` notwendig). In diesem Fall spricht man von CLR Ressourcen oder Managed Resources.

Es ist jedoch möglich, nativ implementierte Funktionen, d.h. Methoden die bspw. in C / C++ geschrieben und kompiliert sind, aus C# heraus aufzurufen. Da man in diesem Fall die Grenze der CLR überschreitet, ist es in diesem Fall üblich, bei der Deallokation eines CLR Objekts, das native Funktionalitäten nutzt, ebenfalls den nativen Speicherbereich zu bereinigen. Und genau hierin liegt die Aufgabe des Destruktors: in ihm kann man native Methoden aufrufen, die für die Freigabe von nativen Speicherbereichen zuständig sind.

Wir werden uns genauer mit der CLR und ihren einzelnen Eigenschaften in einem späteren Kapitel beschäftigen, in dem wir auch noch genauer auf das Thema Speicherverwaltung eingehen. Für Destruktoren befolgen Sie bitte folgende Grundregeln:

- Schreiben Sie keinen Destruktor in C#, wenn Sie nicht genau wissen, was Sie tun.
- Schreiben Sie einen Destruktor in C# nur dann, wenn Ihre Klasse direkt mit nativen Ressourcen interoperiert. In diesem Fall müssen Sie sich auch noch mit der korrekten Implementierung des Interfaces `System.IDisposable` beschäftigen. Was Interfaces im Allgemeinen sind und `System.IDisposable` im Besonderen ist, sehen wir uns in späteren Kapiteln an.

Destruktoren bei C# Klassen können einen beträchtlichen Performancenachteil haben, da Sie sog. Finalizer Pressure auf den Garbage Collector ausüben. Deswegen nochmals: normale Klassen brauchen keinen Destruktor.

5.11 Konstanten mit dem Schlüsselwort `const`

In Abschnitt 5.10.4 haben wir gesehen, wie man ein Feld mit dem Modifizierer `readonly` ausstatten kann, um zu gewährleisten, dass dieses Feld nur bei der Speicherinitialisierung, also im Konstruktor, gesetzt werden kann. Eine ähnliche Funktionalität bietet das Schlüsselwort `const`, mit dem sich Konstanten in Methoden und Klassen definieren lassen. Im folgenden Beispiel sehen Sie, wie beide Varianten erstellt werden:

```

Mit dem Schlüsselwort const wird der Wert
des Feldes hartcodiert und unveränderlich

Das Schlüsselwort const fixiert
den Wert für diese Variable

public class Beispielklasse
{
    public const double Pi = 3.141; ← Die Werte für Konstanten
                                    müssen als Literal angegeben werden

    public void Beispielmethode()
    {
        const string text = „Die Zahl Pi ist ungefähr“;

        Console.WriteLine(text); — Zugriff auf Konstante wie auf Variable in Methode
        Console.WriteLine(Beispielklasse.Pi);
    }
}

Zugriff auf konstantes Feld wie auf statisches Feld
(nicht über this Referenz)

```

Abbildung 71: Konstanten in Klassen und Methoden definieren

In Abbildung 71 werden zwei Konstanten mit dem Schlüsselwort `const` definiert, einmal als Klassenmitglied und einmal in einer Methode. Genauso wie Felder, die `readonly` gekennzeichnet sind, können Sie nach der Initialisierung nicht mehr verändert werden. Allerdings kann man die Werte für Konstanten ausschließlich mit Literalen festlegen (es sind keine anderen Ausdrücke erlaubt, insbesondere keine Methodenaufrufe). Der Wert einer Konstanten wird auch nicht im Konstruktor gesetzt, sondern direkt vom Compiler beim Erstellvorgang in den IL-Code eingebettet. Beim Zugriff sieht man auch, dass konstante Felder über den Klassennamen angesprochen werden und sich in dieser Hinsicht wie statische Felder verhalten.

Konstanten sieht man im Programmieralltag eher selten. Am ehesten wird diese Funktionalität noch in Test-Code verwendet, wenn man bspw. eine Methode mit festgelegten Parameterwerten testen möchte.

5.12 Vererbung

In diesem Abschnitt möchte ich mich dem zweiten großen Paradigma der Objektorientierung widmen: der Vererbung. Dieses Thema ist insbesondere deshalb brisant, weil Vererbung einerseits Polymorphie ermöglicht, andererseits kann falsch eingesetzte Vererbung den Code auch sehr unflexibel und schlecht wartbar machen. Deshalb rate ich Ihnen gleich zu Beginn: setzen Sie Vererbung nur dort ein, wo Sie es für wirklich notwendig erachten. Auch werden die Beispiele, die Sie in den folgenden Abschnitten sehen, abstrakt gehalten sein, damit Sie sich wirklich auf die Möglichkeiten und Grenzen, die Vererbung bietet, konzentrieren können.

5.12.1 Grundlagen der Klassenvererbung

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=OKfiw6iPTsk>.

C# bietet die Möglichkeit, eine Klasse von genau einer anderen Klasse abzuleiten. Diese als Einfachvererbung (engl. Single Inheritance) bezeichnete Funktionalität wird umgesetzt, indem man hinter dem Klassenbezeichner einen Doppelpunkt schreibt und danach den Namen der Klasse setzt, von der man ableiten möchte. Dies kann man gut im nächsten Beispiel betrachten:

```

public class A
{
    private int _zähler = 1;

    public int Zähler
    {
        get { return _zähler; }
    }

    public void ErhöheZähler()
    {
        _zähler++;
    }
}

public class B : A
{
    public void ErhöheZählerUmDrei()
    {
        for (int i = 0; i < 3; i++)
            ErhöheZähler();
    }
}

```

Klasse B leitet von A ab und erbt dadurch alle in A definierten Klassenmitglieder (und damit auch die API von A)

Im Klassenscope von B kann man nun auf Mitglieder von A zugreifen, als wären sie in B definiert

Abbildung 72: Grundlagen der Klassenvererbung

In Abbildung 72 sind die Klassen **A** und **B** zu sehen, wobei letztere von ersterer ableitet. Dies erkennt man an der eben angesprochenen Syntax. Prinzipiell hat das Ableiten von einer anderen Klasse die Auswirkung, dass sämtliche Klassenmitglieder der Basisklasse (im obigen Fall **A**) auch in die Subklasse (im obigen Fall **B**) zur Verfügung stehen, sofern diese nicht mit dem Modifizierer **private** gekennzeichnet sind. Zu sehen ist das in der Methode **B .ErhöheZählerUmDrei**: in dieser wird die Methode **ErhöheZähler** aufgerufen, obwohl sich diese in der Basisklasse **A** befindet. Wichtig ist auch zu bedenken, dass die Subklasse damit die gleiche API bereitstellt wie die Basisklasse.

Indem man im Scope von Subklassen neue Mitglieder hinzufügt, kann man so also die Funktionalität der Basisklasse erweitern. Im obigen Beispiel ergibt das noch nicht sehr viel Sinn, da letztendlich nur eine Methode angeboten wird, mit der man den Wert von **_zähler** um drei statt wie in der Basisklasse um eins erhöhen kann. Prinzipiell kann dies der Nutzer von A auch direkt machen, ohne dafür B einsetzen zu müssen. Sobald wir uns komplexere Klassen anschauen, werden wir sehen, wie man durch Vererbung Vorgaben der Basisklasse in verschiedenen Subklassen unterschiedlich implementieren kann. Das obige Beispiel soll nur verdeutlichen, dass der Zugriff auf nicht **private** Mitglieder der Basisklasse möglich ist.

5.12.2 Die Ist-Eine-Beziehung zwischen Basis- und Subklasse

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=owVzafXa2e0>.

Subklassen werden wie ihre jeweiligen Basisklassen mit dem **new** Operator und einem Konstruktorauftruf instanziert. Wie man die Klassen A und B aus dem vorherigen Beispiel einsetzen kann, sieht man in folgendem Beispiel:

```

static void Main()
{
    A objekt1 = new A();
    B objekt2 = new B();

    objekt1.ErhöheZähler();
    Console.WriteLine(objekt1.Zähler);

    objekt2.ErhöheZähler();
    objekt2.ErhöheZählerUmDrei();
    Console.WriteLine(objekt2.Zähler);

    A objekt3 = new B(); Ist-Eine-Beziehung: Instanzen von Subklasse B können als  
Referenz von Basisklasse A angesprochen werden
    objekt3.ErhöheZähler();
    Console.WriteLine(objekt3.Zähler);
}

```

Bei objekt3 kann nur die API der Basisklasse A genutzt
werden, obwohl tatsächlich eine Instanz von B
dahintersteckt (Typ der Referenz ist ausschlaggebend)

Abbildung 73: Die Ist-Eine-Beziehung zwischen Basisklasse und Subklasse

In Abbildung 73 werden insgesamt drei Objekte erstellt. `objekt1` ist dabei eine Instanz der Klasse `A`, `objekt2` der Klasse `B`. Beide Objekte werden über Variablen ihres jeweiligen Typs referenziert. Im Anschluss wird auf `objekt1` die Methode `ErhöheZähler` aufgerufen und der Wert des Zählers auf der Konsole ausgegeben. Das Gleiche wird danach mit `objekt2` gemacht, da Klasse `B` allerdings von Klasse `A` ableitet, wird außerdem die Methode `ErhöheZählerUmDrei` aufgerufen, die ausschließlich in `B` implementiert ist.

Der große Clou passiert am Ende der `Main` Methode: hier wird ein Objekt von Klasse `B` instanziert, allerdings von einer Variable von Typ `A` referenziert. Genau dies nennt man die Ist-Eine-Beziehung zwischen Subklasse und Basisklasse: Objekte von einer Subklasse können über eine Referenz vom Typ einer Basisklasse angesprochen werden. Zu beachten ist außerdem, dass für `objekt3` nur die API der Klasse `A` zur Verfügung steht (es ist also nicht möglich, die Methode `ErhöheZählerUmDrei` aufzurufen, wie das bei `objekt2` möglich ist, obwohl das von der Variablen `objekt3` referenziert Objekt tatsächlich eine Instanz der Klasse `B` ist). Der Typ der Referenz, die auf das Objekt verweist, ist dabei ausschlaggebend. Ebenfalls möchte ich darauf hinweisen, dass bei der Zuweisung des neu instanzierten Objekts zu `objekt3` eine implizite Konvertierung von `B` nach `A` durchgeführt wird, wie wir sie in Abschnitt 4.3.6.1 kennengelernt haben.

Wenn wir den obigen Code betrachten, fällt es zunächst schwer, den Sinn der Ist-Eine-Beziehung zu verstehen. Warum sollte man schließlich ein Objekt einer Subklasse über seine Basisklasse ansprechen? Im Normalfall gehen dadurch ja die Mitglieder der Subklasse verloren (man kann sie über die Basisklassenreferenz nicht aufrufen). Dennoch ist die Ist-Eine-Beziehung ein wichtiges Fundament für flexiblen Code, da sie den Grundstein für die leichte Austauschbarkeit von Objekten bildet, der für das Konzept der Polymorphie unabdingbar ist. Wenn wir uns genauer mit der Polymorphie beschäftigt haben, werden Sie den Sinn erkennen.

5.12.3 Casting von Basisklasse zu Subklasse

Das Video zu diesem Abschnitt finden Sie unter https://www.youtube.com/watch?v=zig_DkBmW10.

Nachdem wir im letzten Abschnitt besprochen hatten, dass eine Zuweisung eines Objekts vom Typ einer Subklasse zu einer Basisklassenreferenz eine implizite Konvertierung miteinschließt, bleibt natürlich auch die Frage offen, ob man von Basisklassenreferenzen auch zu Subklassenreferenzen casten kann. Genau dies ist ebenfalls möglich, zwar nicht durch implizite Konvertierung, sondern durch explizite Casts und durch den Operator `as`. Schauen wir uns dazu folgendes Beispiel an:

```

public class A { // Implementierung weggelassen }
public class B : A { // Implementierung weggelassen }
public class C : A { // Implementierung weggelassen }

public class Program
{
    public static void Main()
    {
        A objekt1 = new B();
        A objekt2 = new C();

        B objekt3 = (B) objekt1; — Cast von objekt1 von A zu B, gibt Referenz auf B-Objekt zurück
        C objekt4 = (B) objekt2; — Cast von objekt2 von A zu B, wirft Exception

        C objekt5 = objekt2 as C; — Cast mit as Operator, gibt Referenz auf C-Objekt zurück
        C objekt6 = objekt1 as C; — Cast mit as Operator, gibt null zurück

        bool Objekt1IstB = objekt1 is B; — Überprüft, ob objekt1 von Typ B ist, gibt wahr zurück
        bool Objekt2IstB = objekt2 is B; — Überprüft, ob objekt2 von Typ B ist, gibt falsch zurück
    }
}

```

Abbildung 74: Casting von Basisklasse zu Subklasse

In Abbildung 74 sehen Sie, dass zunächst die drei Klassen `A`, `B` und `C` definiert sind, wobei `B` und `C` jeweils von `A` ableiten. Die Implementierung dieser Klassen wurde entfernt, da sie in diesem Beispiel nicht von Belang ist. Innerhalb der Main Methode wird dann die Ist-Eine-Beziehung ausgenutzt, um eine Instanz von `B` und eine Instanz von `C` jeweils in einer Variablen zu referenzieren, die vom Basisklassentyp `A` ist.

In den darauffolgenden vier Anweisungen wird jeweils versucht, mit zwei unterschiedlichen Methoden die A-Referenzen zu casten: zunächst wird dazu der Klammeroperator `()` genutzt, um jeweils `objekt1` und `objekt2` in eine Referenz von `B` explizit zu konvertieren. Die Syntax ist dabei genauso, wie wir sie schon in Abschnitt 4.3.6.2 kennengelernt haben: innerhalb der beiden runden Klammern wird der Zieltyp angegeben. Wichtig ist hier zu beachten, dass eine `System.InvalidCastException` geschmissen wird, wenn der Cast nicht durchgeführt werden kann. Dies ist der Fall, wenn man versucht, die Referenz auf `objekt2`, das ja eine Instanz von `C` ist, auf eine Referenz von `B` zu casten. Diese Exception kann man umgehen, indem man statt des Klammeroperators den `as` Operator einsetzt: dieser gibt entweder eine valide Objektreferenz zurück, wenn der Cast funktioniert, oder `null`, falls ein Cast nicht möglich ist. Die Syntax ist für den `as` Operator wie folgt: nach einem beliebigen Ausdruck schreibt man `as` und danach den Zieltyp, in den man konvertieren möchte, wie in den Anweisungen für `objekt5` und `objekt6` zu sehen ist.

In den letzten beiden Anweisungen wird der `is` Operator eingesetzt, mit dem man zwar nicht casten, aber überprüfen kann, ob das Objekt hinter einer Referenz einer bestimmten Subklasse angehört. Der Syntax für den `is` Operator ist identisch mit dem für `as`, allerdings gibt dieser Operator natürlich einen booleschen Wert zurück.

Auch wenn wir uns jetzt ausgiebig mit Casts von Basisklassen zu Subklassen beschäftigt haben, möchte ich Sie darauf hinweisen, dass sich in guten objektorientiertem Code in C# nahezu keine expliziten Konvertierungen zwischen Klassenreferenzen befinden. Diese expliziten Konvertierungen sollten so selten wie möglich eingesetzt werden. Bei häufiger Anwendung ist das meistens ein Hinweis darauf, dass Klassen und Abstraktionen nicht richtig design sind. Bitte beachten Sie, dass die expliziten Konvertierungsoperatoren () und `as` jeweils nur angewandt werden können, wenn der Quell- und der Zieltyp jeweils in einer Vererbungslinie stehen. Was das genau bedeutet, schauen wir uns im kommenden Abschnitt an.

5.12.4 Klassenhierarchien

Das Video zu diesem Abschnitt finden Sie unter https://www.youtube.com/watch?v=mecB_Zb4WVk.

In den letzten Abschnitten wurde dargestellt, dass eine Klasse von einer anderen Klasse ableiten kann und welche Konsequenzen dies hat. Wir haben dabei jeweils nur eine Klasse von einer anderen erben lassen, in diesem Beispiel werden wir uns allerdings ein abstraktes Beispiel von mehreren Klassen anschauen, die in einer gewissen Vererbungsbeziehung zueinander stehen.

```
public class A { // Implementierung weggelassen }
public class B : A { // Implementierung weggelassen }
public class C : A { // Implementierung weggelassen }
public class D : B { // Implementierung weggelassen }
public class E : B { // Implementierung weggelassen }
public class F : C { // Implementierung weggelassen }
public class G : C { // Implementierung weggelassen }
public class H : C { // Implementierung weggelassen }
public class I : F { // Implementierung weggelassen }
public class J : F { // Implementierung weggelassen }
public class K : H { // Implementierung weggelassen }
```

Abbildung 75: Verschiedene Klassen, die eine Klassenhierarchie bilden

In Abbildung 75 sehen Sie mehrere Klassen, die voneinander in einer gewissen Reihenfolge ableiten. Diese Vererbung geht über mehrere Ebenen hinweg, bspw. leitet die Klasse C von A, F wiederrum von C und I wiederum von F. Ebenfalls sieht man, dass einige Klassen dieselbe Elternklasse haben. Diese Vererbung über mehrere Levels hinweg lässt sich auch in einer sog. Klassenhierarchie darstellen, wie sie im folgenden Bild dargestellt ist.

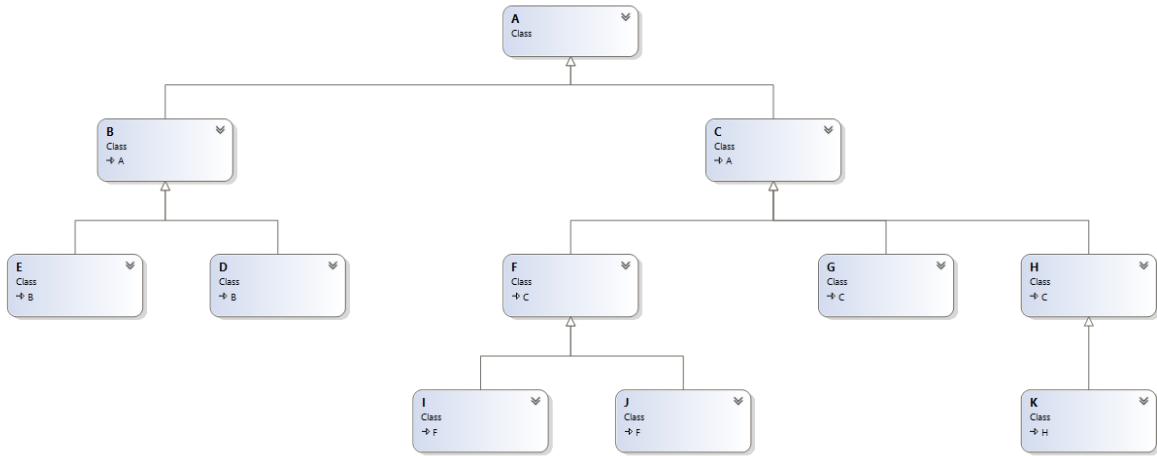


Abbildung 76: Klassenhierarchie als Baumdiagramm dargestellt

In Abbildung 76 sieht man die eben angesprochene Hierarchie in einem Klassendiagramm dargestellt. Dabei symbolisiert ein Knoten in diesem Diagramm eine Klasse. Wenn zwischen zwei Knoten ein Pfeil ist, dann heißt das, dass die Klasse, von der der Pfeil weg weist, von der Klasse ableitet, zu der der Pfeil hinzeigt. So sieht man bspw. unten im Diagramm, dass die Klassen I und J beide von F ableiten.

Klassenhierarchien zweigen also zwei oder mehr Klassen gleichzeitig an und beschreiben, in welchem Ableitungsverhältnis sich diese untereinander befinden. Im folgenden Bild schauen wir uns nochmals eine reale Klassenhierarchie aus dem .NET Framework an: die Items Controls der Windows Presentation Foundation (WPF).

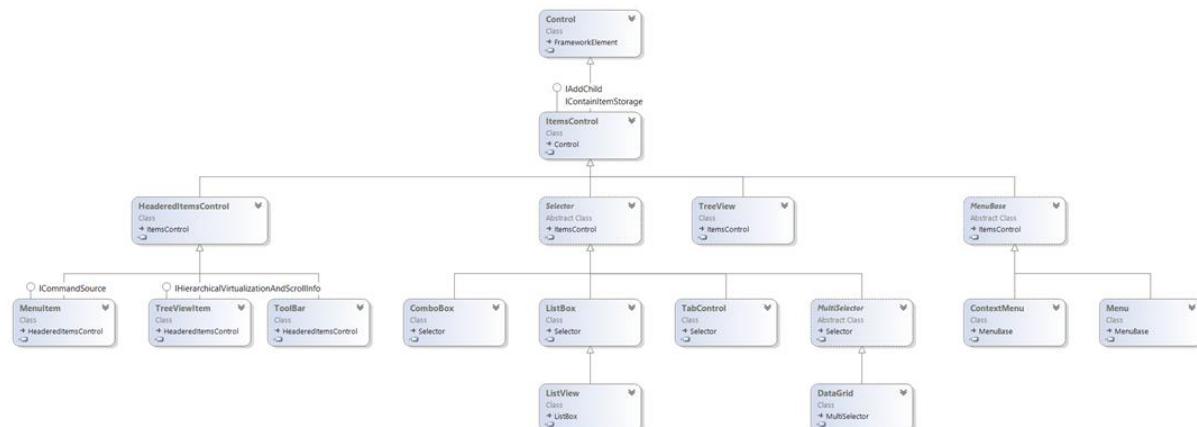


Abbildung 77: Klassenhierarchie der Items Controls in WPF

Klassenhierarchien nutzt man häufig, um sich einen Überblick über mehrere Klassen zu schaffen und zu schauen, ob man zwei verschiedene Klassen mit demselben Code nutzen kann, da sie von einer gemeinsamen Basisklasse ableiten. Generell sollte man aber bei seinem Code darauf achten, nicht zu viel Vererbung zwischen den Klassen einzusetzen. Als Faustregeln gelten:

- Klassenhierarchien sollten so flach wie möglich sein.
- Favor Composition Over Inheritance: man sollte versuchen, die Funktionalität einer Klasse zu erweitern, indem man diese in einer anderen Klasse kapselt, anstatt von ihr abzuleiten.

5.12.5 Der Modifizierer protected

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=AwssbDIEd78>.

Bis jetzt haben wir die Zugriffsmodifizierer `public` und `private`, mit denen man die Sichtbarkeit von Klassenmitgliedern festlegen kann. Ein weiterer Zugriffsmodifizierer ist `protected` – mit ihm kann man bestimmen, dass auf ein Klassenmitglied in der eigenen Klasse und in abgeleiteten Klassen zugegriffen werden kann, jedoch nicht außerhalb dieser Klassenscopes (wie bspw. mit `public`) – `protected` Members können also ohne weiteres in abgeleiteten Klassen genutzt werden.

5.12.6 Virtuelle Methoden

5.12.6.1 Grundsätzliches zu virtuellen Methoden

Das Video zu diesem Abschnitt finden Sie unter https://www.youtube.com/watch?v=ZTvgTBF_Jas.

Wir haben bereits gelernt, dass man durch Ableiten neue Klassenmitglieder zur API der Basisklasse hinzufügen kann. Durch virtuelle Methoden ist sogar möglich, in der Subklasse die Implementierung einer Methode aus der Basisklasse durch eine eigene Implementierung zu ersetzen. Dazu schauen wir uns nochmals eine modifizierte Variante der Klassen an, die einen Feldwert hochzählen.

```
public class A
{
    private int _zähler;

    public int Zähler
    {
        get { return _zähler; }
        protected set { if (_zähler < value) _zähler = value; }
    }

    public virtual void ErhöheZähler()
    {
        _zähler++;                         Modifizierer virtual macht eine Methode virtuell.
                                            Diese Methode kann jetzt in abgeleiteten Klassen
                                            überschrieben werden.
    }
}

public class B : A
{
    public override void ErhöheZähler()
    {
        Zähler += 3;                      Mit dem Modifizierer override setzt man die
                                            Implementierung der Basisklasse außer Kraft.
                                            Stattdessen wird der Code in dieser Methode
                                            ausgeführt.
    }
}
```

Abbildung 78: Virtuelle Methoden definieren und überschreiben

In Abbildung 78 sehen Sie, dass in Klasse `A` die Methode `ErhöheZähler` mit dem Modifizierer `virtual` gekennzeichnet wurde. Damit ist diese Methode als virtuell deklariert, wodurch ableitende Klassen diese überschreiben können. Genau das wird in Klasse `B`, die von `A` ableitet, gemacht: eine Methode mit demselben Funktionskopf und dem Modifizierer `override` (dt. überschreiben, wörtlich „außer Kraft setzen“) wird genutzt, um die Implementierung von `ErhöheZähler` zu ersetzen. In dieser wird das Feld `_zähler`, das über die Eigenschaft `Zähler` angesprochen wird, nicht nur um eins, sondern um drei erhöht. Bitte bedenken Sie auch, dass Eigenschaften und Events jeweils nur

Beim Einsatz von `virtual` und `override` müssen folgende Regeln beachtet werden:

- Eine mit `virtual` oder `override` gekennzeichnete Methode darf nicht `static` sein – statische Funktionen können nicht überschrieben werden.
- Einmal mit `override` überschriebene Methoden können in weiteren Subklassen ebenfalls wieder überschrieben werden, außer man nutzt die Kombination `sealed override` – der Modifizierer `sealed` sorgt dafür, dass die überschriebene Methode nicht mehr zum Überschreiben zur Verfügung steht. `sealed override` nutzt man aber im Allgemeinen nicht (warum sollte man weiteren ableitenden Kindklassen verbieten, die Methode ebenfalls zu überschreiben?).
- Konstruktoren und Destruktoren können nicht als virtuell deklariert werden.

Wenn man virtuelle Methoden aufruft, tritt die sog. Dynamische Bindung in Kraft. Was das genau ist, schauen wir uns im nächsten Abschnitt an.

5.12.6.2 Dynamische Bindung bei virtuellen Methoden

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=DN8pbNW-C4M>.

Sobald man eine virtuelle Methode über ein Objekt aufruft, wird die sog. dynamische Bindung (auch häufig als späte Bindung bezeichnet, engl. Dynamic Binding bzw. Late Binding) eingesetzt. Was das genau bedeutet, schauen wir uns anhand des folgenden Beispiels an:

```
static void Main()
{
    A objekt1 = new A();
    B objekt2 = new B();

    objekt1.ErhöheZähler();
    objekt2.ErhöheZähler();

    Ist-Eine-Beziehung — A objekt3 = new B();           Dieser Aufruf landet aufgrund der dynamischen
    objekt3.ErhöheZähler();                            Bindung von virtuellen Methoden in der
                                                       Implementierung von B
    Console.WriteLine(objekt1.Zähler);
    Console.WriteLine(objekt2.Zähler);
    Console.WriteLine(objekt3.Zähler);
}
```

Abbildung 79: Dynamische Bindung bei virtuellen Methoden

In Abbildung 79 werden drei Objekte instanziert und jeweils die virtuelle Methode `ErhöheZähler` aus dem vorherigen Beispiel aufgerufen. Das funktioniert bei `objekt1` und `objekt2` auch wie erwartet: bei ersterem wird der Zähler in der Methode der Basisklasse `A` um eins erhöht, bei letzterem wird in der Implementierung der Subklasse `B` der Zähler um drei erhöht. Bei `objekt3` wird jedoch die Ist-Eine-Beziehung eingesetzt, d.h. eine Instanz von `B` wird über eine Variable vom Typ `A` referenziert. Intuitiv würde man hier wahrscheinlich vermuten, dass in der Anweisung `objekt3.ErhöheZähler();` die Implementierung von `A` aufgerufen wird – dies ist aber nicht der Fall. Man landet in der überschriebenen Methode von `B` wegen der dynamischen Bindung.

Beim Aufruf von virtuellen Methoden tritt die sog. dynamische Bindung in Kraft. Beim Aufruf wird zunächst der tatsächliche Typ des Objekts, das über eine Referenz angesprochen wird, festgestellt und in dessen vTable nachgeschlagen, ob eine virtuelle Methode innerhalb der Klassenhierarchie überschrieben wurde. Die nächste überschriebene Methode wird dann aufgerufen – auch wenn das Objekt über eine Basisklasse referenziert wurde.

Durch dieses spezielle Vorgehen ist natürlich auch die Performance beim Aufruf virtueller Methoden etwas langsamer als beim Aufruf nicht-virtueller Methoden. Im Normalfall ist dieser Unterschied nicht erheblich, kann jedoch ins Gewicht fallen, wenn man eine Methode mehrere tausend Mal pro Sekunde aufgerufen werden soll.

Zum Schluss dieses Abschnitts bleibt die Frage, ob man sich beim Schreiben einer Klasse direkt Gedanken machen sollte, diese Klasse für Ableitung fit zu machen, bspw. indem man bestimmte (oder sogar alle) Methoden virtuell macht und bestimmte (oder sogar alle) Setter von Eigenschaften oder gar Felder **protected**. Die Antwort ist: Nein. Setzen Sie Vererbung, wie schon oben erwähnt, erst dann ein, wenn Sie sie wirklich brauchen.

5.12.6.3 Die Funktionalität von virtuellen Methoden erweitern

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=BSmotyrkq5U>.

Neben der Möglichkeit, virtuelle Methoden komplett außer Kraft zu setzen, indem man sie in einer Subklasse überschreibt, gibt es ebenfalls die Möglichkeit, ihre Funktionalität durch überschreiben zu erweitern. Dazu wird die **base** Referenz eingesetzt, mit der man auf Mitglieder der (direkten) Basisklasse zugreifen kann, welche nicht **private** sind. Dazu schauen wir uns folgendes Beispiel an, bei dem wir verfolgen möchten, wie oft eine bestimmte Funktionalität aufgerufen wird.

In Abbildung 80 wird zunächst die Klasse **Gehaltsrechner** definiert, welche die Methode **BerechneGehaltserhöhung** enthält, mit der das Gehalt von Mitarbeitern anhand ihres Mitarbeiterverhältnisses im Unternehmen berechnet werden kann. Die Methode ist virtuell und kann damit von ableitenden Klassen überschrieben werden. Die Klasse **GehaltsrechnerMitTracking** tut dies, ruft aber, und das ist der Clou, die Implementierung der Basisklasse auf. Dies wird mit der **base** Referenz gemacht, die mit dem Punktoperator dereferenziert wird, um auf die Mitglieder der Basisklasse zuzugreifen. Im Anschluss folgt der Name der Methode, die aufgerufen werden soll, in diesem Fall eben genau die Funktion, die wir überschrieben haben. Durch dieses Vorgehen kann man die Funktionalität von virtuellen Methoden erweitern, indem man vor den Aufruf der Basisklassenimplementierung beliebige andere Anweisungen schreibt. Im Beispiel unten wird dabei mitverfolgt, wie oft die Methode aufgerufen wird. Dieser Wert kann später über die Eigenschaft **AnzahlAufrufe** abgefragt werden. Gehaltserhöhungen dürfen schließlich nicht zu oft ausgeführt werden ;-)

Dadurch, dass man Objekte von Subklassen über ihre Basisklassen ansprechen kann, bei virtuellen Methoden aber die dynamische Bindung eingesetzt wird, kann man Nutzern, die ein **Gehaltsrechner**-Objekt erwarten, eine Instanz von **GehaltsrechnerMitTracking** übergeben, bspw. über einen Parameter.

```

public class Gehaltsrechner
{
    public virtual decimal BerechneGehaltserhöhung(Mitarbeiter mitarbeiter)
    {
        if (mitarbeiter.Mitarbeiterverhältnis == Mitarbeiter.Management)
            return mitarbeiter.AktuellesGehalt * 1.5m;
        else if (mitarbeiter.Mitarbeiterverhältnis == Mitarbeiter.Teamleiter)
            return mitarbeiter.AktuellesGehalt * 1.1m;
        else if (mitarbeiter.Mitarbeiterverhältnis == Mitarbeiter.Angestellter)
            return mitarbeiter.AktuellesGehalt * 1.05m;
        else
            return mitarbeiter.AktuellesGehalt * 0.8m;
    }
}

public class GehaltsrechnerMitTracking : Gehaltsrechner
{
    public int AnzahlAufrufe { get; private set; }
    public override decimal BerechneGehaltserhöhung(Mitarbeiter mitarbeiter)
    {
        AnzahlAufrufe++;
        return base.BerechneGehaltserhöhung(mitarbeiter);
    }
}

public class Mitarbeiter
{
    public const string Management = „Management“;
    public const string Teamleiter = „Teamleiter“;
    public const string Angestellter = „Angestellter“;

    public string Name { get; set; }
    public string Mitarbeiterverhältnis { get; set; }
    public decimal AktuellesGehalt { get; set; }
}

```

Mit der `base` Referenz kann man die Implementierung der Basisklasse aufrufen. Vor und nach diesem Aufruf lassen sich beliebige andere Anweisungen schreiben, die so die Funktionalität erweitern.

Abbildung 80: Instanzmitglieder der Basisklasse mit der `base` Referenz aufrufen

Auch wenn diese Art der Erweiterung von Funktionalität von Klassen sehr elegant auszusehen scheint, gibt es ein deutlich besseres Muster, um dies zu erreichen. Wenn man mehrere sog. Cross-Cutting-Concerns zur Hauptfunktionalität von Klassen hinzufügen möchte, sollte man das sog. Decorator-Pattern einsetzen, bei dem eine neue Klasse geschrieben wird, die die Funktionalität der ursprünglichen Klasse kapselt und durch Bereitstellen derselben API erweitert. Wie dieses Designmuster genau funktioniert, sehen wir uns in einem späteren Kapitel an.

Der Grundsatz lautet: Vererbung ist deutlich unflexibler als Komposition. Und deshalb gilt auch hier: Favor Composition Over Inheritance.

5.12.7 Abstrakte Methoden und abstrakte Klassen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=aam6oQM1uPo>.

Wir haben in den letzten Abschnitten gesehen, dass mit dem Modifizierer `virtual` Methoden als virtuell gekennzeichnet werden können, sodass diese in Subklassen überschrieben werden können. Mit dem Modifizierer `abstract` lässt sich in Basisklassen sogar die Implementierung der entsprechenden Methode weglassen – ableitende Klassen wird damit vorgeschrieben, welche Methoden sie zu implementieren bzw. überschreiben haben. Das ganze lässt sich an einem Beispiel verdeutlichen.

```

Dieser Modifizierer zeichnet die Klasse
als abstrakt. Abstrakte Klassen lassen
sich nicht instanzieren.

Mit dem Modifizierer
abstract wird eine
Methode definiert,
die von ableitenden
Klassen implementiert
werden muss.

public abstract class MitarbeiterVerwalter
{
    public abstract Mitarbeiter[] LadeMitarbeiter();
}

public class DatenbankMitarbeiterVerwalter : MitarbeiterVerwalter
{
    public override Mitarbeiter[] LadeMitarbeiter()
    {
        // Hier werden Kontakte aus einer Datenbank geladen
        // Code ist aus Platz- und Verständnisgründen weggelassen
    }
}

public class DateiMitarbeiterVerwalter : MitarbeiterVerwalter
{
    public override Mitarbeiter[] LadeMitarbeiter()
    {
        // Hier werden Kontakte aus einer Datei geladen
        // Code ist aus Platz- und Verständnisgründen weggelassen
    }
}

```

Abstrakte Methoden werden nicht implementiert, sondern nur mit einem Semikolon abgeschlossen.

Subklassen müssen die abstrakten Methoden von Basisklassen implementieren oder selber abstrakt sein.

Abbildung 81: Abstrakte Methoden und abstrakte Klassen

In Abbildung 81 sehen Sie u.a. die Klasse `MitarbeiterVerwalter`, welche die abstrakte Methode `LadeMitarbeiter` definiert. Sobald eine Methode mit `abstract` modifiziert wurde, darf diese keine Implementierung mehr enthalten. Stattdessen wird der Methodenkopf einfach mit einem Semikolon abgeschlossen. Bitte beachten Sie, dass nicht nur Methoden, sondern auch Eigenschaften und Events als `abstract` definiert werden können.

Eine Klasse mit mindestens einem abstrakten Mitglied ist nicht komplett ausimplementiert und muss deshalb ebenfalls mit dem Modifizierer `abstract` gekennzeichnet werden. Abstrakte Klassen können nicht mit dem `new` Operator instanziert werden. Stattdessen kann die in ihnen erstellte Funktionalität erst genutzt werden, wenn andere Klassen davon ableiten. Ebenfalls ist es möglich, Klassen als abstrakt zu kennzeichnen, wenn in ihr keine abstrakten Mitglieder definiert sind. Das macht man häufig, wenn zwei Klassen teilweise dieselben Methoden und Felder definieren: diese werden dann in eine abstrakte Basisklasse ausgelagert, von der die beiden eben erwähnten Klassen ableiten. Die Basisklasse wird mit `abstract` gekennzeichnet, um potentiellen Nutzern direkt anzudeuten, dass man diese Klasse nicht direkt benutzen kann.

Subklassen müssen sämtliche abstrakte Mitglieder der Basisklasse, von der Sie ableiten, implementieren oder sich selbst als `abstract` kennzeichnen. Abstrakte Methoden implementiert man einfach, indem man sie wie virtuelle Methoden überschreibt, wie es jeweils in den Klassen `DatenbankMitarbeiterVerwalter` und `DateiMitarbeiterVerwalter` zu sehen ist. Wenn abstrakte Methoden aufgerufen werden, tritt ebenso die dynamische Bindung in Kraft wie bei virtuellen Methoden.

Folgende Regeln müssen bei der Benutzung von `abstract` eingehalten werden:

- Abstrakte Methoden dürfen wie virtuelle Methoden nicht statisch sein.
- Einmal überschriebene abstrakte Methoden können in weiteren Subklassen ebenfalls wieder überschrieben werden, außer man zeichnet diese Methode mit dem Modifizierer `sealed override` aus.
- Konstruktoren und Destruktoren können nicht als abstrakt gekennzeichnet werden.

Mit abstrakten Mitgliedern gibt man letztendlich vor, wie bestimmte Teile der API von Subklassen auszusehen haben. Diese Bestandteile der API lassen sich auch über die abstrakte Basisklasse auslösen, da sie die entsprechende Methode deklarieren, aber nicht implementieren.

Abstrakte Klassen, die ausschließlich abstrakte Mitglieder enthalten, sind Abstraktionen.
Abstraktionen sind ein wichtiges Fundament für Polymorphie und Entkopplung von Klassen.
Abstraktionen kann man ebenfalls über C# Interfaces erstellen.

5.12.8 Object – die ultimative Basisklasse

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=vezp3xFWzDI>.

Nachdem wir in den letzten Abschnitten Vererbung im Detail besprochen haben, schauen wir uns in diesem Abschnitt die Klasse `object` bzw. `System.Object` an, welche die allgemeine Basisklasse für sämtliche Klassen in C# ist. Von `object` leiten alle anderen Typen implizit oder explizit ab und folglich kann man auch sämtliche Klasseninstanzen implizit zu `object`-Referenzen konvertieren, wie wir es im Abschnitt 5.12.2 kennengelernt haben. Das folgende Beispiel veranschaulicht dies:

```
public class A
{
}

public class B : object
{
}

public class Program
{
    static void Main()
    {
        object objekt1 = new A();
        object objekt2 = new B();
    }
}
```

Klasse A leitet implizit von object ab, da keine andere Basisklasse angegeben ist

Klasse B leitet explizit von object ab
(diese Schreibweise ist nicht üblich)

Ist-Eine-Beziehung mit Instanz von A und B

Abbildung 82: `object` ist Basisklasse aller Klassen

Da also jede Klasse in der Vererbungshierarchie mit `object` steht, ist es natürlich interessant zu erfahren, welche Mitglieder von dieser Klasse definiert werden. In der folgenden Abbildung sind diese zu sehen:

```

namespace System
{
    public class Object
    {
        public virtual bool Equals(object obj);
        public static bool Equals(object objA, object objB);
        public virtual int GetHashCode();
        public Type GetType();
        protected object MemberwiseClone();
        public static bool ReferenceEquals(object objA, object objB);
        public virtual string ToString();
    }
}

```

Abbildung 83: Die Mitglieder der Klasse object

Zu den in Abbildung 83 zu sehenden Klassenmitglieder sind die jeweiligen Implementierungen nicht abgebildet, sondern nur jeweils der Methodenkopf angegeben. Auf diese möchte ich dennoch kurz eingehen:

- Die virtuelle Methode `ToString` gibt einen `string` zurück, der das jeweilige Objekt beschreibt. Standardmäßig gibt diese Methode den vollqualifizierten Namen der Klasse, die instanziert wurde, zurück. Üblicherweise wird diese Methode in Subklassen überschrieben, wenn man anderen Informationen zu einem Objekt herausgeben möchte. `ToString` wird bspw. auch dann genutzt, wenn man ein Objekt über `Console.WriteLine` ausgibt.
- Die Funktionen `Equals` und `GetHashCode` sind zwei wichtige Methoden, die für die Gleichheit von zwei Objekten ausschlaggebend sind. Mit der Instanzmethode `Equals` kann ein Objekt sich mit dem im Parameter übergebenen Objekt auf Gleichheit überprüfen. Die Standardimplementierung nutzt dafür Referenzgleichheit, d.h. wenn das im Parameter referenzierte Objekt an derselben Stelle im Speicher zu finden ist (letztendlich also dasselbe Objekt referenziert wird), gelten beide Objekte als gleich. Zwei Referenzen auf dieselbe Speicherstelle hin überprüfen kann man auch direkt, wenn man den `==` Operator oder die statische Funktion `ReferenceEqual` von `object` auruft. `Equals` wird häufig in Subklassen überschrieben, um bspw. Funktionalität zu implementieren, dass zwei Objekte als gleich gelten, wenn sie dieselbe ID haben (Entities) oder alle Feldwerte gleich sind (Value Objects). `GetHashCode` wird ebenfalls zur Gleichheitsüberprüfung eingesetzt, allerdings gibt diese Methode jeweils einen 32 Bit Hash-Wert zurück, der mit den Hashwerten anderer Objekte verglichen werden kann. Wenn zwei Hashwerte unterschiedlich sind, können die jeweiligen Objekte nicht gleich sein. `GetHashCode` wird bspw. bei Suchen nach Objekten in Arrays oder Collections genutzt.
Wer `Equals` überschreibt, sollte deshalb auch `GetHashCode` in seiner Klasse überschreiben. Weitere Information zu diesem Methodenpaar finden Sie in der [MSDN Library](#). Die statische `Equals` Methode kann man ignorieren, da Sie nur die Instanzvariante aufruft und dabei die gegebenen Parameter verwendet.
- Mit `GetType` kann man sich Typinformationen zum aktuellen Objekt holen, die man für Reflection einsetzen kann. Was Reflection genau ist, schauen wir uns in einem späteren Kapitel genauer an.
- Mit `MemberwiseClone` kann man sich eine sog. flache Kopie (engl. Shallow Copy) des aktuellen Objekts erstellen. Flache Kopien sind neue Instanzen von bestehenden Objekten, wobei die Feldwerte des Ursprungsobjekts einfach in die Feldwerte des neuen Objekts zugewiesen werden. Dies steht im Gegensatz zu sog. tiefen Kopien (engl. Deep Copy), die bei Feldern, die andere Objekte referenzieren, diese ebenfalls klonen. Somit wird nicht nur ein

Objekt kopiert, sondern der gesamte Objektgraph, der hinter diesem Objekt steht. MemberwiseClone ist **protected**, sodass diese Methode nur in abgeleiteten Klassen aufgerufen werden kann. Allzu häufig wird diese Methode im Programmieralltag aber nicht eingesetzt.

Den wichtigsten Punkt, den man aus diesem Abschnitt mitnehmen sollte, ist der, dass man jedes Objekt als **object** interpretieren kann.

5.12.9 Mit Modifizierer sealed

Den Modifizierer **sealed** haben wir schon in Zusammenhang mit dem Modifizierer **override** kennengelernt. Bei überschriebenen Methoden hat er dort verhindert, dass diese in weiteren Subklassen nochmals überschrieben werden können. **sealed** kann man aber auch auf Klassen anwenden – mit der Bedeutung, dass keine andere Klasse mehr von ihr ableiten kann. Mehr ist bei diesem Modifizierer nicht zu beachten.

5.12.10 Statische Klassen, statische Mitglieder und Vererbung

Auch statische Mitglieder werden an Subklassen vererbt, d.h. man kann auch über den Bezeichner der Subklasse auf statische Mitglieder zugreifen. Wie schon in den entsprechenden Abschnitten oben erwähnt können statische Methoden nicht virtuell oder abstrakt gekennzeichnet werden und schließen sich damit für Polymorphie aus. Auch ist das Ableiten von Klassen, die mit dem Modifizierer **static** gekennzeichnet sind, nicht möglich. Deswegen hier nochmals der Tipp: verwenden Sie statische Mitglieder oder statische Klassen nur da, wo es wirklich notwendig ist und setzen Sie ansonsten auf Instanzmitglieder und nicht-statische Klassen.

5.12.11 Vererbung mit Interfaces

In Abschnitt 5.12.7 haben wir bereits Abstraktionen angesprochen. Diese können einerseits über abstrakte Klassen, die ausschließlich abstrakte Methoden, Eigenschaften und Events definieren erzeugt werden, oder eben über Interfaces. Letztere schauen wir uns in den folgenden Abschnitten genauer an und stellen dabei die Unterschiede zu abstrakten Basisklassen fest.

5.12.11.1 Grundsätzliches zur Interfacedefinition

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=LUIGa1bpJqo>.

Interfaces werden genauso wie Klassen innerhalb von Namespaces deklariert und formen neben Klassen den zweiten C#-Typ, den wir kennenlernen. Im Folgenden Beispiel sehen Sie eine Definition eines Beispielinterfaces:



Abbildung 84: Definition eines Interface in C#

Wie in Abbildung 84 zu erkennen ist, sind Interfacedefinitionen sehr ähnlich zu Klassendefinitionen. Beide besitzen einen Bezeichner und einen Scope, in dem Mitglieder definiert sind. Dennoch muss man folgende Punkte bei der Interfacedefinition beachten:

- Ein Interface kann nicht mit Modifizierern wie `static` oder `sealed` ausgestattet werden wie Klassen. Nur die Zugriffsmodifizierer `public`, `protected`, `internal` oder `private` sind gestattet, wobei ersterer der übliche ist.
- Der Bezeichner eines Interfaces sollte immer mit einem großen I (wie Ida) beginnen. Diese Konvention wird zwar nicht vom Compiler erzwungen, ist aber in .NET sehr weit verbreitet.
- Sämtliche Interfacemitglieder sind abstrakt. Dabei werden sie jedoch nicht mit dem Modifizierer `abstract` gekennzeichnet, den wir in Abschnitt 5.12.7 kennengelernt haben, sondern sie besitzen eine eigene Syntax: keine Modifizierer und jede Methode wird mit einem Semikolon abgeschlossen.
- Interfaces können keine Feld-, Konstruktor- oder Destruktorddefinitionen enthalten.

Genauso wie komplett abstrakte Klassen legen Interfaces also die API des Objekts fest, die das Interface implementieren. Wie das genau geht, schauen wir uns im nächsten Abschnitt an.

5.12.11.2 Interfaces in Klassen implementieren

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=ggB3Mc7sKGU>.

Genauso wie man von einer anderen Klasse ableiten kann, ist es möglich, von einem Interface abzuleiten. Dabei muss man die Mitglieder, die das Interface vorgibt, in die Klasse implementieren. Dazu sehen wir uns ein modifiziertes Beispiel an, das wir bereits in Abschnitt 5.12.7 kennengelernt haben.

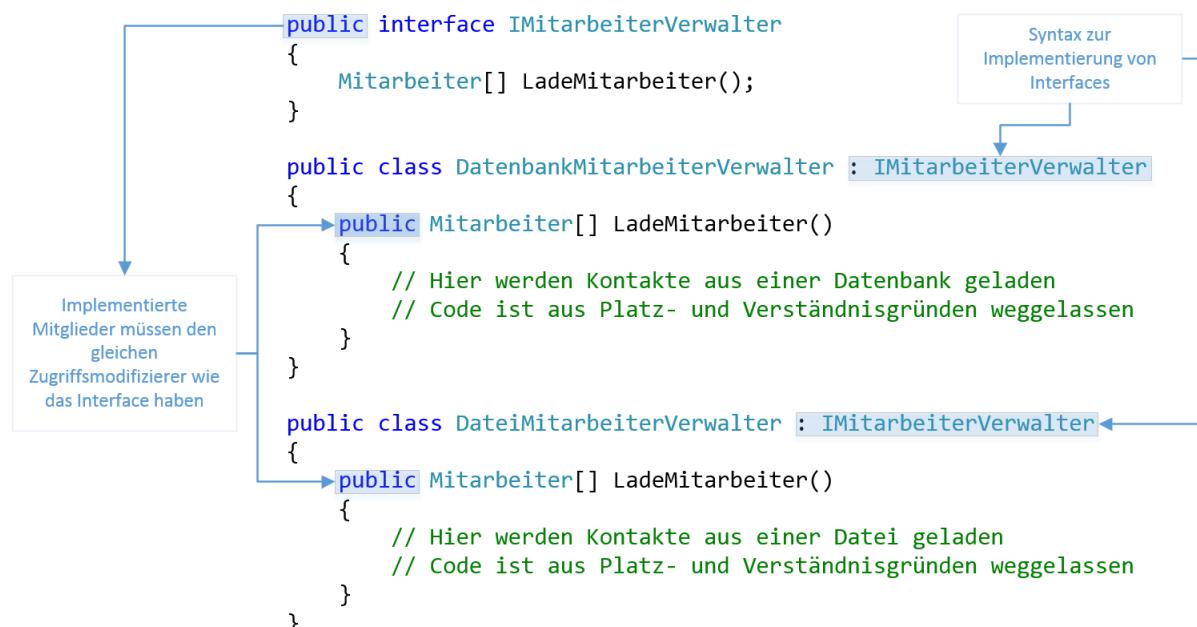


Abbildung 85: Interfaces in Klassen implementieren

In Abbildung 85 sieht man, dass statt einer abstrakten Klasse ein Interface namens `IMitarbeiterVerwalter` verwendet wird, welches ein Mitglied zum Laden von Mitarbeiter-Objekten vorgibt. Von dieser Abstraktion leiten wie schon im damaligen Beispiel die beiden Klassen `DatenbankMitarbeiterVerwalter` und `DateiMitarbeiterVerwalter` ab, die den Rückgabewert jeweils auf unterschiedliche Art und Weise beschaffen, was im Beispiel aber nicht

gezeigt wird. Beide Klassen implementieren die Methode, die das Interface vorgibt, hier werden jedoch Unterschiede zur abstrakten Klasse aus dem vorigen Beispiel deutlich:

- Interfacemitglieder werden nicht überschrieben, sondern ganz gewohnt definiert. Deshalb sind Klassenmitglieder, die aus Interfacesvorgaben stammen, auch nicht virtuell – folglich wird auch keine dynamische Bindung beim Aufruf angewandt. Es ist allerdings erlaubt, diese Klassenmitglieder als virtuell zu kennzeichnen und es damit weiteren Subklassen zu erlauben, diese Mitglieder zu überschreiben.
- Der Modifizierer der implementierten Mitglieder muss gleich sein mit dem Modifizierer, der vor dem Interface zu finden ist (üblicherweise verwendet man hier aber sowieso nur `public`).
- Interfacemitglieder können nicht mit dem Modifizierer `static` in Klassen implementiert werden (genauso wie bei virtuelle Methoden).

Die Syntax zum Ableiten von Interfaces ist dabei identisch mit der Syntax zum Ableiten von Klassen. Interfaces verhalten sich dabei grundsätzlich wie abstrakte Klassen mit ausschließlich abstrakten Mitgliedern – Interfaces stellen damit ebenfalls Abstraktionen dar. Dennoch gibt es einen gewichtigen Unterschied: mit Interfaces ist Mehrfachvererbung möglich. Wie dies genau funktioniert, sehen wir uns in den nächsten Abschnitten an.

5.12.11.3 Die Ist-Eine-Beziehung bei Interfaces

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=596zBf3rpAY>.

Genauso wie bei Basis- und Subklassen lässt sich bei Interfaces und implementierenden Klassen die Ist-Eine-Beziehung anwenden, d.h. man kann über den Interfacetyp ein Objekt der entsprechenden Klasse referenzieren. In folgender Abbildung finden Sie dazu nochmals ein Beispiel:

```
static void Main()
{
    IMitarbeiterVerwalter dbVerwalter = new DatenbankMitarbeiterVerwalter();
    IMitarbeiterVerwalter dateiVerwalter = new DateiMitarbeiterVerwalter();

    Mitarbeiter[] mitarbeiter = dbVerwalter.LadeMitarbeiter();
    mitarbeiter = dateiVerwalter.LadeMitarbeiter();
}
```

Ist-Eine-Beziehung zwischen Interface
und implementierender Klasse

Abbildung 86: Ist-Eine-Beziehung zwischen Interface und implementierender Klasse

Genauso wie bei der Ist-Eine-Beziehung bei Basis- und Subklasse kann man auch nur die Mitglieder eines Objekts aufrufen, die im Interface definiert sind. Ebenso müsste man explizite Casts einsetzen, um zurück zum eigentlichen Objekttypen zu konvertieren.

5.12.11.4 Mehrere Interfaces in eine Klassen implementieren

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=lC-3VYlbVDk>.

Im Gegensatz zu Klassen ist mit Interfaces Mehrfachvererbung möglich, d.h. eine Klasse kann von mehreren Interfaces ableiten. Dabei werden die verschiedenen Mitglieder der Interfaces wie gewohnt in die Klasse implementiert. Ein Problem kann genau dann entstehen, wenn zwei Interfaces identische Member besitzen: dann muss zumindest eines von ihnen explizit in der Klasse implementiert werden. Wie man das macht, sieht man im folgenden Beispiel:

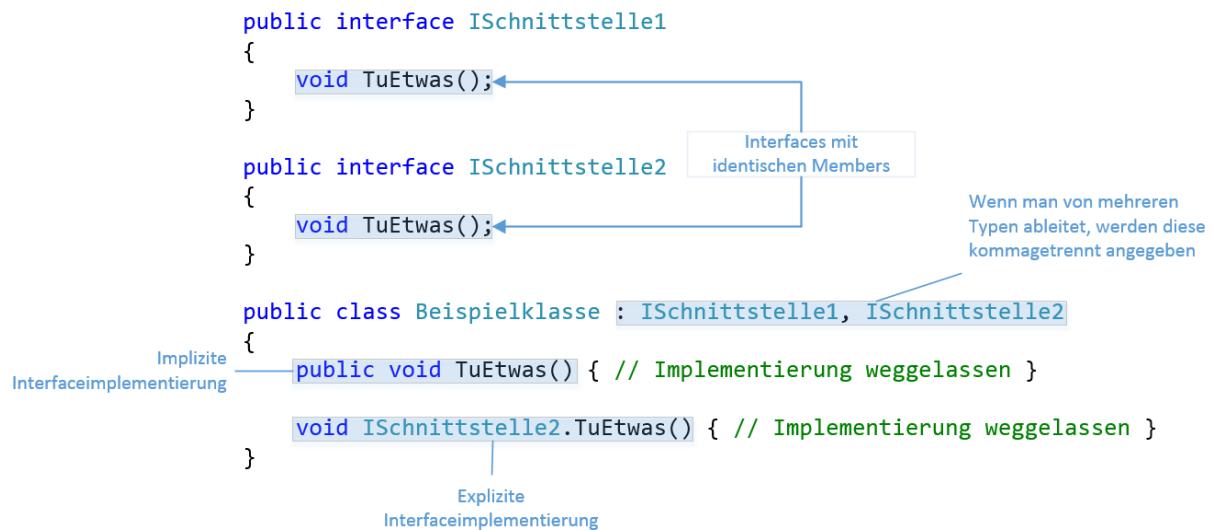


Abbildung 87: Implizite und explizite Interfaceimplementierungen

In Abbildung 87 sieht man die beiden Interfaces `ISchnittstelle1` und `ISchnittstelle2`, die beide jeweils eine Methode `TuEtwas` vorgeben. Die darunter stehende Klasse `Beispielklasse` möchte von beiden Interfaces ableiten. Wenn man von mehreren Typen ableiten möchte (sowohl Klassen als auch Interfaces), gibt man diese einfach kummagetrennt an. Um den Konflikt bei den Mitgliedern der Interfaces aufzuheben, wird die `TuEtwas` Methode von `ISchnittstelle1` implizit implementiert, d.h. nach dem uns bereits bekannten Vorgehen. Bei `TuEtwas` von `ISchnittstelle2` wird jedoch die sog. explizite Interfaceimplementierung angewandt, wobei folgende Regeln dabei beachtet werden müssen:

- Explizite Interfaceimplementierungen dürfen nicht mit Modifizierern ausgestattet werden
- Stattdessen integriert man in den Bezeichner des Klassenmitglieds den Namen des Interfaces gefolgt von einem Punkt, im Beispiel oben also `ISchnittstelle2.TuEtwas`
- Ansonsten verhalten sich diese Mitglieder wie ihre normalen Pendants

Auch wenn es zu keinen Konflikten bei Interfaceimplementierungen kommt, kann man sich entscheiden, bestimmte oder alle Mitglieder eines Interfaces explizit zu implementieren. Das hat aber einen deutlichen Nachteil: explizit implementierte Mitglieder sind nicht in der API der Klasse enthalten. Stattdessen muss man ein Objekt der Klasse erst in den entsprechenden Interfacetyp überführen (im obigen Beispiel `ISchnittstelle2`), um das entsprechende Mitglied als Nutzer aufrufen zu können. Diesen Nachteil ist im Normalfall nicht gewünscht, weswegen man sich an folgenden Grundsatz halten sollte:

Explizite Interfaceimplementierungen sollten so weit wie möglich vermieden werden. Man sollte sie nur in seltenen Fällen einsetzen, bspw. um Mitgliederkonflikte aufzulösen, wenn eine Klasse mehrere Interfaces implementiert.

5.12.11.5 Mehrfachvererbung mit Interfaces

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=Y5iWhlg01zE>.

Neben der Möglichkeit, mehrere Interfaces in einer Klasse zu implementieren, gibt es auch die Möglichkeit, bei der Definition von einem Interface von beliebig vielen anderen Interfaces abzuleiten.

```

public interface ISchnittstelle1
{
    void MethodeInSchnittstelle1();
}

public interface ISchnittstelle2
{
    void MethodeInSchnittstelle2();
}

public interface ISchnittstelle3 : ISchnittstelle1, ISchnittstelle2
{
    void MethodeInSchnittstelle3();
}

```

Mehrfachvererbung bei Interfacesdefinitionen

Abbildung 88: Mehrfachvererbung bei Interfaces

In Abbildung 88 sehen Sie drei Interfaces, wobei das letzte von `ISchnittstelle1` und `ISchnittstelle2` ableitet. Bei der Interfacedefinition gibt man dazu kummagetrennt die Interfaces nach dem Doppelpunkt an, von denen man ableiten möchte. Ableiten heißt bei Interfaces, dass die Mitglieder der anderen Interfaces übernommen werden. Dabei kann es zwar zu Namenskonflikten bei den Mitgliedern kommen, bei diesen wird vom Compiler aber kein Fehler erzeugt, denn diese können durch explizite Implementierungen innerhalb einer Klasse aufgelöst werden. Dennoch sollte man versuchen, solche Namenskonflikte zu vermeiden.

Generell sollte man jedoch vermeiden, Interfaces voneinander ableiten zu lassen. Stattdessen sollte man auch das sog. Interface Segregation Principle achten, dass wir in einem späteren Kapitel bei den SOLID Prinzipien genauer besprechen.

5.13 Weitere interessante C# Features

5.13.1 Das Schlüsselwort var

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=3oGC0NWr930>.

Mit dem Schlüsselwort `var` lässt sich die Schreibweise für Variablenarten deutlich verkürzen. Dazu schauen wir uns wie gewohnt ein Beispiel an:

```

static void Main()
{
    IDGenerator generator1 = new IDGenerator();

    var generator2 = new IDGenerator();
}

```

Schlüsselwort var

Tatsächlicher Typ für var wird aus Zuweisungsausdruck bestimmt

Abbildung 89: Das Schlüsselwort var

In Abbildung 89 sehen wir zwei Variablen, die erstellt werden. Die erste wird dabei auf gewohnte Weise definiert. Bei der zweiten wird das Schlüsselwort `var` eingesetzt: hier setzt der Compiler den

Typ der Variablen automatisch beim Erstellvorgang. Der Typ wird dabei aus dem Ausdruck bestimmt, der der Variablen zugewiesen wird. Beim **var** müssen folgende Dinge beachtet werden:

- **var** kann nur für Variablen eingesetzt werden – nicht für Parameter, Felder oder sonstige Typangaben.
- Wenn **var** eingesetzt wird, muss bei der Variablendefinition ein Wert zugewiesen werden (sonst könnte der Compiler keinen Typen automatisch ableiten).

Persönlich nutze ich nahezu ausschließlich **var**, um Variablen zu definieren, weswegen wir es ab jetzt auch hauptsächlich einsetzen.

5.13.2 Die Modifizierer ref und out für Parameter

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=EBfhWoU4PcY>.

Wir haben bereits in mehreren vorherigen Abschnitten über die Unterschiede zwischen Werte- und Referenztypen unterhalten. Dabei haben wir herausgestellt, dass bei Wertetypen der Wert direkt in einer Variablen gehalten, bei Referenztypen jedoch die Speicheradresse zu einem Objekt verwaltet wird. Dieses unterschiedliche Verhalten hat Auswirkungen bei Funktionsaufrufen, wenn ein Parameter ein Wertetyp ist. Sehen Sie sich dazu folgendes Beispiel an:

```
static void Main()
{
    int wert = 42;

    ManipuliereWert1(wert);
    Console.WriteLine(wert);

    ManipuliereWert2(ref wert);
    Console.WriteLine(wert);
}

static void ManipuliereWert1(int wert)
{
    wert += 5;
    Console.WriteLine("Manipulierter Wert: " + wert.ToString());
}

static void ManipuliereWert2(ref int wert)
{
    wert += 5;
    Console.WriteLine("Manipulierter Wert: " + wert.ToString());
}
```

Beim Aufruf muss **ref** (oder **out**)
beim Wertetypparameter explizit
mitangegeben werden.

Der Modifizierer **ref** bewirkt bei einem
Parameter, dass ein Wertetyp als Referenz
übergeben wird.

Abbildung 90: ref Modifizierer zur Übergabe von Wertetypen als Referenz

In Abbildung 90 werden neben der Main Methode zwei weitere Methoden definiert. In ManipuliereWert1 wird wie in bisher gewohnter Weise der Parameter **int** wert angegeben. Wenn diese Funktion in der Methode Main aufgerufen wird, wird der Wert der Variablen zum Parameter kopiert, was als Call-By-Value bezeichnet wird. Dies ist nicht der Fall bei ManipuliereWert2, da hier beim Parameter zusätzlich der Modifizierer **ref** angegeben wird. Dies bedeutet, dass beim Aufruf nicht der Wert in den Parameter kopiert wird, sondern wie bei Referenztypen eine Adresse auf die angegebene Variable zugewiesen wird (Call-By-Reference). Damit hat die Manipulation innerhalb der Methode auch den Wert der ursprünglichen Variablen aus der Main Methode geändert.

Statt `ref` kann man auch den Modifizierer `out` verwenden. Dann muss innerhalb der Methode der Parameter einmal ein Wert zugewiesen werden (was bei `ref` nicht der Fall sein muss). Bitte beachten Sie, dass man `ref` und `out` jedoch nur bei Parametern, die Wertetypen sind, anwenden darf (also nicht Klassen und Interfaces). Im Programmieralltag sieht man diese beiden Modifizierer eher selten, aber wenn man bspw. die verschiedenen TryParse Methoden der primitiven Datentypen nutzt, bekommt man Sie zu Gesicht.

5.13.3 Attribute

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=9BtM4ax-eUA>.

Attribute sind Zusatzinformationen, mit denen man bestimmte Codeteile ausstatten kann. Am häufigsten werden Attribute auf Klassen, Interfaces und Methoden angewendet, allerdings können sie auch bspw. auf Parameter, Rückgabewerttypen, Eigenschaften oder Events angewandt werden. Diese Zusatzinformationen können von entsprechenden Komponenten mittels des Reflection-Mechanismus ausgelesen und verarbeitet werden – sie steuern beispielsweise bestimmte Aufrufeigenschaften von Methoden (v.a. wenn es um das Thema Security geht) oder kennzeichnen bspw. bestimmte Codeteile für eine besondere Benutzung. Genau dazu sehen wir uns jetzt ein Beispiel für Attribute bei Unit Tests an.

```

TestClass Attribut kennzeichnet, dass
diese Klasse Testmethoden enthält

[TestClass]
public class RoverTests
{
    [TestMethod]
    public void RoverMussKorrektPositioniertSeinNachInitialisierung()
    {
        var testTarget = new RoverBuilder().MitXPosition(2)
            .MitYPosition(3)
            .Erstelle();

        Assert.IsTrue(testTarget.XPosition == 2 &&
                      testTarget.Yposition == 3);
    }
}

```

Abbildung 91: Einsatz von Attributen bei Unit Tests

In Abbildung 91 sehen Sie die Klasse `RoverTests`, die genau eine Methode enthält. Die Klasse ist mit dem Attribut `[TestClass]` ausgestattet, die Methode mit dem `[TestMethod]`. Diese Attribute werden von einer sog. Test Execution Engine genutzt, um die Klassen innerhalb einer Assembly zu finden, die Testmethoden enthalten und die entsprechenden Testmethoden auszuführen. Diese sog. Unit Tests sind einfache Methoden, die jeweils ein bestimmtes Verhalten einer Klasse auf Korrektheit überprüfen. Wie man Unit Tests genau einsetzt, um seinen Code sukzessive aufzubauen, beleuchten wir in einem späteren Kapitel.

Zurück zum Thema Attribute, für Sie gelten folgende Regeln:

- Attribute werden immer in eckigen Klammern vor dem Codeteil angegeben, den sie kennzeichnen sollen (im obigen Beispiel vor der Klasse und vor der Methode).

- Attribute sind selber Klassen, die von der Klasse Attribute ableiten und dem Attribut `AttributeUsage` gekennzeichnet sind. Folglich kann man auch eigene Attribute erstellen, häufig fördern sie aber kein gutes objektorientiertes Design. Eine Anleitung zum Erstellen eigener Attribute finden Sie unter <http://msdn.microsoft.com/de-de/library/sw480ze8.aspx>
- Die Bezeichner von Attributen enden konventionsgemäß immer mit dem Suffix `Attribute`, bspw. bei `TestMethodAttribute`. Dieses Suffix muss man beim Kennzeichnen nicht angeben, wie man im obigen Beispiel sehen kann.

Im Programmieralltag sollte man versuchen, Attribute so selten einzusetzen wie möglich. Bei der Nutzung einiger Frameworks, bspw. bei Unit Tests oder der Serialisierung von Objektgraphen bekommt man sie jedoch häufiger zu Gesicht.

5.13.4 Operatorenüberladung

Das Video zu diesem Abschnitt finden Sie hier: https://www.youtube.com/watch?v=abn3zbT_mUA.

In Abschnitt 4.3.9 haben wir uns mit Operatoren auseinandergesetzt. Es gibt in C# die Möglichkeit, Operatoren zu überladen, sodass bestimmte Methoden aufgerufen werden, wenn Operatoren auf die entsprechenden Typen angewandt werden. Sehen wir uns dazu das folgende Beispiel an:

```
public class Adresse
{
    public Adresse(string straße, string postleitzahl, string ort)
    {
        Straße = straße;
        Postleitzahl = postleitzahl;
        Ort = ort;
    }

    public string Straße { get; private set; }
    public string Postleitzahl { get; private set; }
    public string Ort { get; private set; }

    public static bool operator ==(Adresse ersteAdresse, Adresse zweiteAdresse)
    {
        return ersteAdresse.Straße == zweiteAdresse.Straße &&
               ersteAdresse.Postleitzahl == zweiteAdresse.Postleitzahl &&
               ersteAdresse.Ort == zweiteAdresse.Ort;                                Überladung für Operator ==
    }

    public static bool operator !=(Adresse ersteAdresse, Adresse zweiteAdresse)
    {
        return (ersteAdresse == zweiteAdresse) == false;                         Überladung für Operator !=
    }
}
```

Abbildung 92: Operatorenüberladung

In Abbildung 92 sehen Sie die Klasse Adresse, in der mithilfe der beiden unteren Methoden die Operatoren `==` und `!=` überladen wurden. Dabei müssen folgende Regeln beachtet werden:

- Operatorenüberladungen sind immer statische Methoden, bei denen der Bezeichner durch das Schema `operator Operatorzeichen` ersetzt wird (oben bspw. `operator ==`).
- Der Rückgabetyp wird anhand des Operators bestimmt (bspw. bei Vergleichsoperatoren `bool`, bei arithmetischen Operatoren der Typ der Parameter)

- Überladungen für binäre Operatoren erhalten zwei Parameter, Überladungen für unäre Operatoren logischerweise nur einen Parameter.

Die geschriebenen Funktionen werden genau dann aufgerufen, wenn ein Operator auf die entsprechenden Operanden, die zu den Parametern passen müssen, aufgerufen wird:

```
static void Main()
{
    var adresse1 = new Adresse(„Universitätsstr. 31“, „93053“, „Regensburg“);
    var adresse2 = new Adresse(„Universitätsstr. 31“, „93053“, „Regensburg“);

    Console.WriteLine(adresse1 == adresse2); Bei dieser Überprüfung wird die  
Operatorüberladung aufgerufen
}
```

Abbildung 93: Operatorenüberladung aufrufen

Nicht alle Operatoren in C# lassen sich überladen. Weiterhin muss bei den Vergleichsoperatoren in Paaren überladen werden, d.h. wenn bspw. der Kleiner-Operator < überladen wird, muss ebenfalls der Größer-Operator > überladen werden. Folgende Tabelle zeigt Operatoren, die überladen werden können:

Unäre Operatoren	+, -, !, ~, ++, --, true, false
Binäre Operatoren	+, -, *, /, %, &, , ^, <<, >>, (== und !=), (< und >), (>= und <=)

Abbildung 94: Überladbare Operatoren in C#

Damit ist es in C# nicht möglich, Operatoren wie den Punktoperator ., Methodenaufrufe (), den Zuweisungsoperator =, die logischen Operatoren &&, ||, ?: sowie new, typeof, as und is zu überladen. Wenn ein binärer Operator überladen wird, wird ebenfalls automatisch auch der dazugehörige Zuweisungsoperator überladen (bspw. + und +=). Operatorenüberladungen sieht man eher selten im Programmieralltag, je nach Einsatzgebiet kommen sie aber vor.

5.13.5 Implizite und explizite nutzerdefinierte Typumwandlungen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=gbrVN-pBWVI>.

Neben der Operatorenüberladung kann man als Programmierer ebenfalls Einfluss auf implizite und explizite Konvertierungen nehmen. Diese besitzen eine ähnliche Syntax wie Operatorüberladungen, wie wir im folgenden Beispiel sehen:

```

public class Person
{
    public Person(string vorname, string nachname)
    {
        Vorname = vorname;
        Nachname = nachname;
    }

    public string Vorname { get; private set; }
    public string Nachname { get; private set; }

    public static implicit operator string(Person person)
    {
        return person.Vorname + „“ + person.Nachname;
    }

    public static explicit operator Person(string @string)
    {
        var namenArray = @string.Split(',');
        if (namenArray.Length != 2)
            throw new ArgumentException(„Parameter entspricht keinem Vor- und Nachnamen“);

        return new Person(namenArray[0], namenArray[1]);
    }
}

```

Abbildung 95: Implizite und explizite nutzerdefinierte Konvertierungen

Die uns bereits bekannte Klasse Person wurde in Abbildung 95 so modifiziert, dass jeweils eine statische Methode für die implizite Konvertierung von `Person` nach `string` und eine für die explizite Konvertierung von `string` nach `Person` definiert wurde. Dabei gilt es folgende Dinge zu beachten:

- Genauso wie bei Operatorenüberladungen wird bei nutzerdefinierten Konvertierungsmethoden eine statische Methode eingesetzt, gefolgt vom Schlüsselwort `implicit` für implizite Konvertierungen bzw. `explicit` für Casts.
- Der Rückgabetyp wird nicht nach den Modifizierern, sondern im Bezeichner nach `operator` angegeben.

Ebenfalls sehen wir zum ersten Mal im Code der untersten Methode, dass eine Exception geworfen wird. Wie wir bereits in vorherigen Abschnitten angedeutet haben, werden Exceptions eingesetzt, um bspw. bei Methodenaufrufen mit fehlerhaften Parameterwerten das Programm notfalls zum Absturz zu bringen. Genau dasselbe wird im obigen Beispiel gemacht, wenn der übergebene `string` nicht zwei Worte enthält, die voneinander durch ein Leerzeichen getrennt sind. Wie genau Exceptions funktionieren, sehen wir uns in einem späteren Kapitel an.

Die Konvertierungen kann man dann wie folgt einsetzen:

```

static void Main()
{
    Person person = (Person) „Walter White“; — Cast von string zu Person
    string name = person; — Implizite Konvertierung von Person zu string
}

```

Abbildung 96: Nutzerdefinierte Konvertierungen aufrufen

In Abbildung 96 sehen Sie, wie bei der ersten Anweisung ein `string` explizit in den Typ `Person` gecastet wird. In der zweiten Anweisung wird das erstellte Objekt implizit in einen `string` konvertiert.

Allgemein werden implizite und explizite nutzerdefinierte Konvertierungen eher selten im Programmieralltag angewandt. Wenn man sich doch dazu entschließt, sollte man beachten, dass bei Konvertierungen, bei denen leicht ein Fehler auftreten kann, man zu expliziten Casts tendieren sollte. In diesem Fall muss der Nutzer durch Angabe des () Operators sein Einverständnis geben, dass er sich eines potenziellen Fehlers bewusst ist (genau wie das im obigen Beispiel der Fall ist). In allen anderen Fällen kann man implizite Konvertierungen verwenden.

5.13.6 Alle Modifizierer im Überblick

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=OjNaSLcB0es>.

Wir haben über verschiedene Abschnitte hinweg uns immer wieder unterschiedliche Modifizierer für Klassen, Methoden und Felder und ihre Auswirkungen auf den Gesamtcode angeschaut. In diesem Abschnitt wollen wir diese Zusammenführen und wenn möglich kategorisieren. Des Weiteren werden Sie einige neue Modifizierer kennenlernen, die man im Programmieralltag eher selten einsetzt oder mit denen wir uns in einem späteren Kapitel beschäftigen.

Modifizierer	Bemerkung
<code>public</code> <code>private</code> <code>protected</code> <code>internal</code>	Diese sog. Zugriffsmodifizierer (engl. Access Modifiers) werden genutzt, um bei Typen und Typmitgliedern die Sichtbarkeit festzulegen. <code>public</code> , <code>private</code> und <code>protected</code> sind uns bereits bekannt, <code>internal</code> jedoch nicht: mit diesem legt man fest, dass auf einen Typ oder ein Mitglied ausschließlich innerhalb der Assembly (bzw. dem Projekt), in dem er definierte wurde, zugegriffen werden kann. Prinzipiell rate ich nicht dazu, diesen Zugriffsmodifizierer einzusetzen, dennoch trifft man im .NET Framework relativ häufig darauf. Weiterhin ist es möglich, <code>internal protected</code> als Kombination einzusetzen, sodass so ausgezeichnete Mitglieder ebenfalls in ableitenden Klassen eingesetzt werden können, die innerhalb derselben Assembly liegen.
<code>abstract</code>	Zeigt an, dass eine Methode, eine Eigenschaft oder ein Event nicht implementiert ist, sondern nur eine API-Vorgabe für ableitende Klassen ist. Bei Klassen zeigt <code>abstract</code> an, dass diese nicht instanziiert werden können und man deshalb von ihnen ableiten muss, um einen effektiven Nutzen aus ihnen ziehen zu können. Klassen, die wenigstens ein abstraktes Mitglied besitzen, müssen natürlich selbst als abstrakt gekennzeichnet sein.
<code>async</code>	Zeigt an, dass eine Methode asynchron ausgeführt werden soll (genaueres dazu im Abschnitt über Threading)
<code>const</code>	Zeigt an, dass der Wert eines Feldes oder einer Variablen nicht modifiziert werden kann, da dieser vom Compiler beim Erstellvorgang durch das angegebene Literal festgesetzt wurde.
<code>event</code>	Deklariert einen Event innerhalb eines Typs.
<code>extern</code>	Zeigt an, dass eine Methode nicht in derselben Assembly implementiert ist. Üblicherweise wird dieser Modifizierer genutzt, um Methodenköpfe von nativen Funktionen in C# zu deklarieren und die native Funktion darüber aufzurufen. <code>extern</code> kann nicht in Kombination mit <code>abstract</code> eingesetzt werden.
<code>new</code>	Verbirgt explizit eine Methode einer Basisklasse mit einer neuen Implementierung. Dies hat nichts mit virtuellen Methoden und später Bindung zu tun. Wird das Objekt über seine Basisklasse referenziert und ruft man die in der Subklasse mit <code>new</code> modifizierte Methode auf, landet man trotzdem in der Implementierung der Basisklasse. Ich rate Ihnen davon ab, diesen Modifizierer

	einzusetzen. Vorsicht: verwechseln sie diesen Modifizierer nicht mit dem <code>new</code> Operator zum Instanziieren von Klassen und Strukturen.
<code>override</code>	Zeigt an, dass eine Methode, eine Eigenschaft oder ein Event, welche entweder virtuell oder abstrakt sind, der Basisklasse überschrieben wird.
<code>partial</code>	Mit <code>partial</code> lassen sich Klassen oder Strukturen auf mehrere Dateien hinweg verteilen. Der Compiler fügt dann alle partiellen Dateien zu einem Typen zusammen. Das wird v.a. dann gemacht, wenn der Code für eine Klasse teils automatisch generiert wird. Dann kommen generierter Code und vom Entwickler geschriebener Code für bspw. eine Klasse in unterschiedliche Dateien, damit beim erneuten automatischen Generieren (was üblicherweise beim Erstellvorgang gemacht wird) nicht der Code des Entwicklers überschrieben wird. Ebenfalls kann man Methoden als partial deklarieren, das macht man aber eher selten im Programmieralltag.
<code>readonly</code>	Zeigt an, dass der Wert eines Feldes nur innerhalb des Konstruktors gesetzt werden und nach der Instanziierung des Objekts nicht mehr geändert werden kann. Bitte beachten Sie den Unterschied zwischen <code>const</code> und <code>readonly</code> – bei letzterem kann der Wert für ein Feld durchaus berechnet werden, allerdings eben nur im Konstruktor. Bei Konstanten ist der Wert über ein Literal hartcodiert. Die beiden können auch nicht in Kombination verwendet werden.
<code>sealed</code>	Zeigt an, dass von einer Klasse nicht abgeleitet werden darf oder dass eine Methode, eine Eigenschaft oder ein Event, die jeweils überschrieben werden, nicht in weiteren Subklassen überschrieben werden darf. Bei letzterem Fall kann man <code>sealed</code> nur in Kombination mit <code>override</code> nutzen.
<code>static</code>	Deklariert ein Klassen- oder Strukturmitglied als statisch – damit gehört das Mitglied zum Typ selbst und nicht zu einer spezifischen Instanz.
<code>unsafe</code>	Deklariert eine Methode oder einen Block mit geschweiften Klammern als nicht-sicher, d.h. man kann innerhalb dieses Scopes Zeiger in der von C / C++ gewohnten Art und Weise einsetzen. Ich rate Ihnen vom Einsatz dieser Möglichkeit ab, da die Common Language Runtime nicht-sichere Codeabschnitte nicht auf Typsicherheit verifizieren kann.
<code>virtual</code>	Deklariert eine Methode, eine Eigenschaft oder ein Event als virtuell, sodass ableitende Klassen die Implementierung dieser Mitglieder durch eigene Implementierungen ersetzen können.
<code>volatile</code>	Deklariert ein Feld als volatile (unbeständig). Diese Funktionalität setzt man ein, wenn man beim Schreiben weiß, dass ein Feldwert von mehreren Threads ausgelesen und gesetzt wird, ohne dass die lock Anweisung eingesetzt wird. <code>volatile</code> stellt hier sicher, dass jederzeit der aktuellste Wert für dieses Feld zur Verfügung steht. Dieses Schlüsselwort setzt man im Programmieralltag sehr selten ein.

Abbildung 97: Alle C# Modifizierer im Überblick

5.14 Exceptions

In mehreren Fällen haben wir Exceptions (dt. Ausnahmen) in den vorherigen Abschnitten angesprochen. Wir wissen, dass Exceptions Fehler im Programmstatus anzeigen, am häufigsten verursacht durch falsch gesetzte Parameterwerte oder Fehlern beim Zugriff auf Elemente, welche die Prozessgrenze überschreiten (bspw. Festplatten-, Datenbank- oder Netzwerkzugriffe). Weiterhin wissen wir, dass bei Auftreten einer Exception der Ablauf sofort gestoppt und das Programm zum Absturz gebracht wird. In den folgenden Abschnitten sehen wir uns genauer an, was beim Thema Exceptions alles zu beachten ist.

5.14.1 Exceptions werfen mit dem Schlüsselwort `throw`

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=xgD6JDWymqw>.

Exceptions werden mit dem Schlüsselwort `throw` ausgelöst. Danach gibt man einen Ausdruck an, der vom Typ `System.Exception` ist. Üblicherweise instanziert man dabei gleich das entsprechende Exception -Objekt mit `new`. Ausnahmen sind in dieser Hinsicht nichts anderes als Objekte. Das Interessante beim Werfen von Exceptions ist der Kontrollfluss, der im nächsten Beispiel verdeutlicht wird:

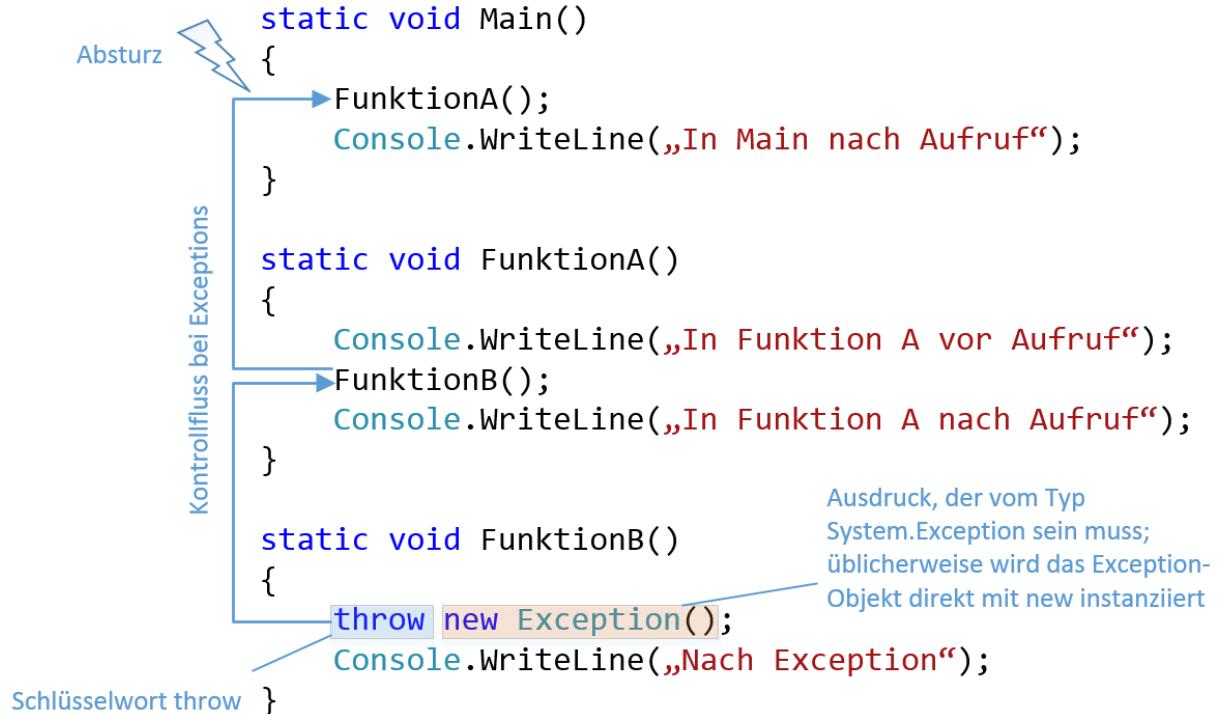


Abbildung 98: Exceptions mit throw auslösen

In Abbildung 98 sehen Sie die `Main` Methode, die `FunktionA` aufruft, welche wiederum `FunktionB` auruft. In letzterer Methode wird direkt in der ersten Anweisung eine Exception mit `throw` nach dem eben beschriebenen Schema geworfen. Dabei wird direkt die Basisklasse `Exception` instanziert, was man üblicherweise nicht tut, sondern eine ihrer Subklassen nimmt. Hier wird es nur als abstraktes Beispiel genutzt, um die Funktionalität zu verdeutlichen.

Das wichtigste, was man bei Exceptions beachten muss, ist, dass sie Einfluss auf den Kontrollfluss des Programms nehmen, wie an den Pfeilen links verdeutlicht: sobald eine Exception geworfen wird, wird die Codeausführung der aktuellen Anweisung abgebrochen und überprüft, ob die Exception innerhalb eines sog. `try-catch` Blocks geworfen wurde. Nach einem solchen Block wird erst innerhalb der gerade ausgeführten Methode gesucht, dann in der aufrufenden Methode, dann in dessen aufrufender Methode usw. bis man in der `Main` Methode angekommen ist. Wird auch hier nichts gefunden, stürzt der Prozess (bzw. der Thread) ab. Dies kann man auch kürzer ausdrücken:

Wenn eine Exception geworfen wird, wird ein passender try-catch Block zur Ausnahmebehandlung entlang des Call Stack gesucht. Wird die Exception bis zur `Main` Methode nicht behandelt, wird der Prozess (bzw. der Thread) durch die Ausnahme beendet.

Als Call Stack bezeichnet man die Aufrufreihenfolge der Methoden, in der die aktuelle Anweisung aufgeführt wird. Beim Werfen der Exception im obigen Beispiel befinden sich die Methoden in folgender Weise auf dem Call Stack:

- FunktionB
- FunktionA
- Main

Deshalb bekommt man im obigen Beispiel auch nur den String „In Funktion A vor Aufruf“ auf der Konsole zu Gesicht, bevor es zum Absturz kommt.

5.14.2 Exceptions mit try-catch Blöcken behandeln

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=34ATBWEKLXs>.

Durch **try-catch** Blöcke können Exceptions behandelt und damit ein Absturz verhindert werden. Dabei ist dieses Konstrukt so zu verstehen, dass Exceptions, die innerhalb eines **try** Blocks geworfen werden, den Kontrollfluss so ändern, das als nächstes die Anweisungen im **catch** Block ausgeführt werden, der sich entlang des Call Stacks am nächsten zur ausgelösten Exception befindet und zum deren Typ passt. Sehen Sie sich dazu folgendes Beispiel an:

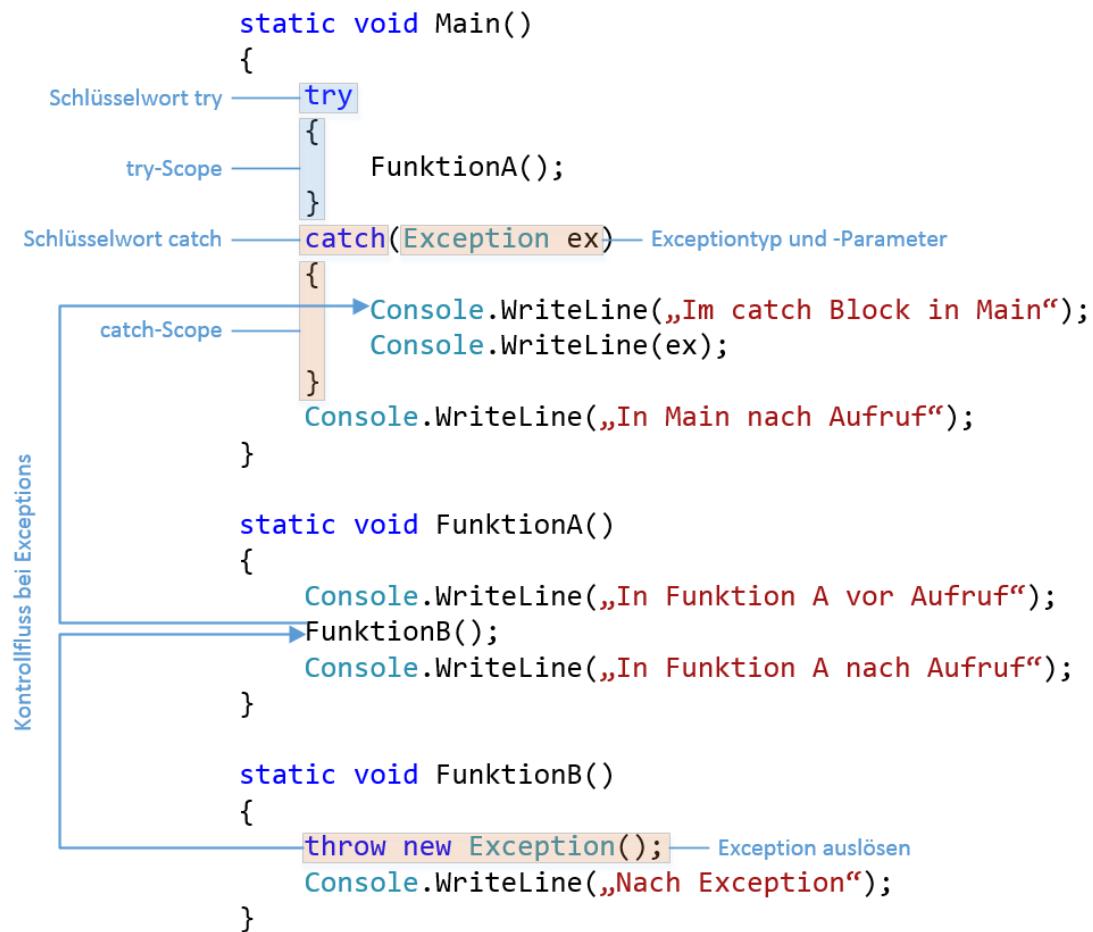


Abbildung 99: Verwendung von try-catch Block zur Exceptionbehandlung

In Abbildung 99 sehen Sie eine modifizierte Variante des vorherigen Beispiels: der Aufruf von FunktionA findet jetzt innerhalb des Scopes eines **try** Blocks statt. Nach einem **try** Block können beliebig viele **catch** Blöcke folgen, im obigen Beispiel wird nur ein einziger verwendet, dessen Ausnahmetyp System.Exception ist. Diesen gibt man beim **catch** Block an wie einen Parameter und genauso kann man auch innerhalb des **catch**-Scopes auf das Exceptionobjekt zugreifen.

Durch den Einsatz des **try-catch** Blocks wird die Applikation nicht beendet, da es nach dem **try** Block einen zum Ausnahmetyp passenden **catch** Block gibt, durch den die Exception behandelt werden kann. Wichtig ist aber dennoch zu beachten, dass in FunktionA die Anweisung

`Console.WriteLine(„In Funktion A nach Aufruf“);` nicht ausgeführt wird, da beim Suchen nach einem passenden `catch` Block in dieser Methode nichts gefunden wird. Der Kontrollfluss geht sofort weiter bis zur Main Methode.

5.14.3 System.Exception – die Basisklasse aller Exceptions

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=zcc0j2aYtpA>.

In den vorherigen beiden Abschnitten haben wir die Klasse `System.Exception` direkt instanziert und ausgelöst. Dies sollte man allerdings in tatsächlichen Code unterlassen und stattdessen andere Exceptionklassen nutzen, die anhand ihres Typnamens und der am Objekt angehafteten Information den aufgetretenen Fehler besser spezifizieren. Das .NET Framework bietet eine ganze Reihe an vorgefertigter Exceptionklassen mit, von denen einige wichtige in folgender Tabelle zu sehen sind:

Klasse	Bemerkung
<code>System.ArgumentException</code>	Zeigt den Fehler an, dass ein bestimmter Parameter mit einem falschen Wert übergeben wurde.
<code>System.ArgumentNullException</code>	Zeigt den Fehler an, dass ein übergebener Referenztyp-Parameter als <code>null</code> übergeben wurde.
<code>System.ArgumentOutOfRangeException</code>	Zeigt den Fehler an, dass sich ein bestimmter Parameterwert nicht innerhalb eines bestimmten Intervalls befindet (bspw. ein Index für einen Array).
<code>System.FormatException</code>	Zeigt den Fehler an, dass ein bestimmter Parameter nicht einem vorgegebenen Format entspricht (bspw. wenn ein <code>string</code> zu einem <code>int</code> konvertiert werden soll).
<code>System.NotImplementedException</code>	Zeigt an, dass eine bestimmte Methode noch nicht implementiert ist (wird standardmäßig in eine von Visual Studio erstellte Methode miteingefügt).
<code>System.NotSupportedException</code>	Wird im Normalfall bei implementierten Mitgliedern von Interfaces verwendet und zeigt, dass genau dieses Mitglied von der Klasse nicht unterstützt wird.
<code>System.InvalidOperationException</code>	Zeigt an, dass ein bestimmter Methodenaufruf nicht valide ist, da sich das dazugehörige Objekt (oder allgemein der dazugehörige Kontext) in einem falschen Zustand befindet.
<code>System.NullReferenceException</code>	Zeigt an, dass versucht wurde, eine Referenz, die <code>null</code> ist, mit dem Punktoperator zu dereferenzieren (diese Exception sollte nicht vom Entwickler ausgelöst werden).
<code>System.OutOfMemoryException</code>	Zeigt an, dass nicht mehr genügend dynamischer Speicher zum Allokieren neuer Objekte zur Verfügung steht (diese Exception sollte nicht vom Entwickler ausgelöst werden).
<code>System.InvalidCastException</code>	Zeigt an, dass ein expliziter Cast zu einem anderen Datentyp nicht durchgeführt werden kann (diese Exception sollte nicht vom Entwickler ausgelöst werden).

<code>System.DivideByZeroException</code>	Zeigt an, dass eine Division durch 0 durchgeführt wurde (diese Exception sollte nicht vom Entwickler ausgelöst werden).
<code>System.IO.IOException</code>	Basisklasse für alle Input-Output bezogenen Exceptions, bspw. Konsolen-, Datei- oder Datenbankzugriffe.

Abbildung 100: Einige wichtige Exceptionklassen

Wie man in der obigen Tabelle sieht, gibt es auch Exceptionklassen, die man nicht selbst instanzieren und auslösen sollte, wie bspw. `NullReferenceException` und `OutOfMemoryException`. Diese werden von der Common Language Runtime automatisch ausgelöst, wenn die entsprechende Bedingung eintritt, bspw. wenn der Heap voll ist oder eine Division durch 0 durchgeführt wird. Dennoch sollte man dazu tendieren, bestehende Exceptionklassen so weit wie möglich wiederzuverwenden anstatt neue Exceptionklassen zu schreiben.

Der wichtigste Punkt ist, dass sämtliche Exceptionklassen direkt oder indirekt von `System.Exception` ableiten – Objekte anderen Typs darf man nämlich nicht hinter dem Schlüsselwort `throw` zum Auslösen von Exceptions angeben. Auch wenn Sie eigene Exceptionklassen implementieren möchten, müssen Sie von `System.Exception` ableiten – dazu mehr im folgenden Abschnitt. Die Basisklasse `Exception` bietet als wichtigste Mitglieder die Eigenschaften `Message` und `StackTrace`, über die eine Fehlermeldung in Prosa und der Call Stack, also die Aufrufe der Methoden, die zur Exception geführt haben, aufgelistet ist.

5.14.4 Eigene Exceptionklassen implementieren

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=45j6BhS-ids>.

Wenn Sie eigene Exceptionklassen implementieren möchten, sollten Sie folgende Punkte beachten:

- Ihre Klasse muss direkt oder indirekt von `System.Exception` ableiten.
- Ausnahmen sollten keine Methoden enthalten, die Logik ausführen, die für das jeweilige Programm relevant ist, sondern ausschließlich Informationen zum Fehler für den entsprechenden `catch` Block bereitstellen. Diese Informationen sollten über den Konstruktor angenommen werden und den Nutzern des Exception-Objekts via Eigenschaften lesend zur Verfügung gestellt werden.
- Die Basisklasse `Exception` enthält eine Eigenschaft `Message` vom Typ `string`, in der mit Prosa beschrieben werden sollte, warum der Fehler aufgetreten ist (diesen Prosatext sieht man dann auch in Visual Studio, wenn die Exception ausgelöst und nicht behandelt wird). Das dazugehörige Feld zu dieser Eigenschaft kann nur über den Konstruktor von `Exception` gesetzt werden.

In folgendem Beispiel werden diese drei Punkte nochmals verdeutlicht:

```

public class FügeMitarbeiterHinzuException : Exception
{
    private readonly Mitarbeiter _fehlerhafterMitarbeiter;

    public FügeMitarbeiterHinzuException(string message, Mitarbeiter fehlerhafterMitarbeiter)
        : base(message) — Exceptionnachricht an Konstruktor der Basisklasse weiterreichen
    {
        _fehlerhafterMitarbeiter = fehlerhafterMitarbeiter;
    }

    public Mitarbeiter FehlerhafterMitarbeiter
    {
        get { return _fehlerhafterMitarbeiter; }
    }
}

```

Abbildung 101: Nutzerdefinierte Exceptionklasse

Die in Abbildung 101 zu sehende Klasse `FügeMitarbeiterHinzuException` könnte bspw. ausgelöst werden, wenn versucht wird, einen neuen Mitarbeiter, dem noch kein Mitarbeiterverhältnis zugewiesen wurde, zu registrieren. Die Exception könnte dann wie in folgendem Codeabschnitt genutzt werden:

```

public void FügeMitarbeiterHinzu(Mitarbeiter mitarbeiter)
{
    if (mitarbeiter.Mitarbeiterverhältnis != Mitarbeiter.Management || 
        mitarbeiter.Mitarbeiterverhältnis != Mitarbeiter.Teamleiter || 
        mitarbeiter.Mitarbeiterverhältnis != Mitarbeiter.Angestellter)
    {
        string message = „Dem Mitarbeiter wurde noch kein Mitarbeiterverhältnis zugewiesen.“;
        throw new FügeMitarbeiterHinzuException(message, mitarbeiter);
    }
    mitarbeiter.ID = _idGenerator.ErzeugeNächsteID();
    SpeichereNeuenMitarbeiter(mitarbeiter);
}

private void SpeichereNeuenMitarbeiter(Mitarbeiter mitarbeiter)
{
    // Implementierung aus Verständnis- und Platzgründen weggelassen
}

```

Abbildung 102: Nutzerdefinierte Exception auslösen

In Abbildung 102 sehen Sie, wie zu Beginn der Methode `FügeMitarbeiterHinzu` eine Überprüfung stattfindet, ob das Mitarbeiterverhältnis auf einen validen Wert gesetzt wurde. Ist dies nicht der Fall, wird die im vorigen Beispiel erstellte Exception ausgelöst mit einer Nachricht und dem Mitarbeiter als Information, die beide dem Konstruktor übergeben werden. Bei einem direkten oder indirekten Aufrufer der Funktion kann dann innerhalb eines `try-catch` Blocks die `FügeMitarbeiterHinzuException` gefangen und mit entsprechenden Schritten reagiert werden, bspw. indem man eine Fehlermeldung ausgibt und den Nutzer bittet, das Mitarbeiterverhältnis für den entsprechenden Mitarbeiter einzutragen.

Wie im letzten Abschnitt schon erwähnt, sollten Sie eigene Exceptionklassen soweit wie möglich vermeiden. Nutzen Sie stattdessen die bereits bestehenden Klassen des .NET Frameworks, außer Sie benötigen für Ihren Kontext spezifische Informationen.

5.14.5 Mehrere catch-Blöcke für unterschiedliche Exceptiontypen verwenden

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=NK787jCe43I>.

Im Beispiel in Abschnitt 5.14.2 haben wir einen `try` Block mit nur einem einzigen `catch` Block verwendet. Prinzipiell ist es aber möglich (und teilweise auch notwendig), beliebig viele `catch` Blöcke nach einem `try` Block aufzuführen, solange sich die entsprechend angegeben Exceptiontypen unterscheiden. Hierzu sehen wir uns nochmals die Methode `System.Convert.ToInt32` an, jedoch diesmal unter dem Aspekt der Ausnahmen, die beim Aufruf dieser Funktion auftreten können. Diese sind laut [MSDN](#):

- `System.FormatException`: Diese Exception tritt auf, wenn der übergebene String keiner Zahl entspricht, d.h. nicht aus einem optionalen Vorzeichen und einer beliebigen Anzahl von Ziffern zwischen 0 und 9 besteht.
- `System.OverflowException`: Diese Exception tritt auf, wenn die übergebene Zahl nicht in den Wertebereich von `int` passt, d.h. kleiner als -2^{31} oder größer als $2^{31} - 1$ ist.

Wenn wir eine Methode `LeseZahlVonKonsoleEin` sicher gegen jegliche Art von Absturz schreiben wollen, müsste Sie deshalb wie folgt aussehen:

```
public int LeseZahlVonKonsoleEin()
{
    while (true)
    {
        Console.WriteLine("Ihre Zahl: ");
        var eingabe = Console.ReadLine();
        Console.WriteLine();

        try
        {
            return Convert.ToInt32(eingabe);
        }
        catch (FormatException) — catch Block für FormatException
        {
            Console.WriteLine("Sie haben keine Zahl eingegeben. Bitte versuchen Sie es erneut.");
        }
        catch (OverflowException) — catch Block für OverflowException
        {
            Console.WriteLine("Ihre Zahl ist zu groß oder zu klein. Bitte versuchen Sie es erneut.");
        }
    }
}
```

Abbildung 103: Mehrere catch-Blöcke mit unterschiedlichen Exceptiontypen

In Abbildung 103 wird nach dem `try` Block, der den Aufruf von `Convert.ToInt32` umschließt, zwei `catch` Blöcke aufgelistet, die als Exceptiontypen einmal `FormatException` und einmal `OverflowException` angeben (hier sieht man auch, dass man den Bezeichner weglassen kann, wenn man nicht direkt auf das Exception-Objekt zugreifen möchte). Tritt nun beim Aufruf der Methode `ToInt32` eine Exception auf, wird anhand des Typs der Exception der passende `catch` Block gesucht und dieser ausgeführt. Sind dessen Anweisungen vorüber, wird aus der gesamten `try-catch` Struktur herausgesprungen. Beim Suchen nach dem passenden `catch` Block wird dabei wie folgt vorgegangen:

Beim Suchen nach passenden Exceptiontypen werden die catch Blöcke der Reihe nach abgearbeitet. Ein catch Block gilt als passend, wenn der angegebene Typ mit dem des Exception-Objekts übereinstimmt oder eine Basisklasse von ihm darstellt. Im Anschluss wird ausschließlich der ausgewählte Block ausgeführt und im Anschluss das try-catch Konstrukt verlassen.

Folglich ist es möglich, jede mögliche Exception zu fangen, indem man im catch Block den Typ `System.Exception` angibt – tun Sie das jedoch standardmäßig nicht. Damit drücken Sie in Ihrem Code aus, dass Ihnen der spezielle Grund einer Exception (die ja einen Fehler anzeigt) egal ist und sie keine spezielle Fehlerbehandlung dafür anbieten. Exceptions sollten dennoch nie dazu führen, dass Ihr Programm abstürzt. Deswegen ist es genau an einer Stelle sinnvoll, eine einen catch Block mit `System.Exception` als Typ zu verwenden: in der Main Methode. Sollte hier eine Exception gefangen werden, heißt das, dass Ihr Fehlerbehandlungscode in später aufgerufenen Methoden noch unvollständig ist. Geben Sie in diesem Fall den Nutzer in einer Meldung an, dass ein unerwarteter Fehler aufgetreten ist, loggen Sie die Exception inklusive aller Informationen und Stack Trace und beenden Sie das Programm.

In der folgenden Abbildung sehen Sie einen Auszug aus der Klassenhierarchie der im obigen Beispiel eingesetzten Klassen `FormatException` und `OverflowException`. Wie eben schon gesagt, könnten Sie beide auch über ihre jeweiligen Basisklassen im `catch` Block ansprechen.

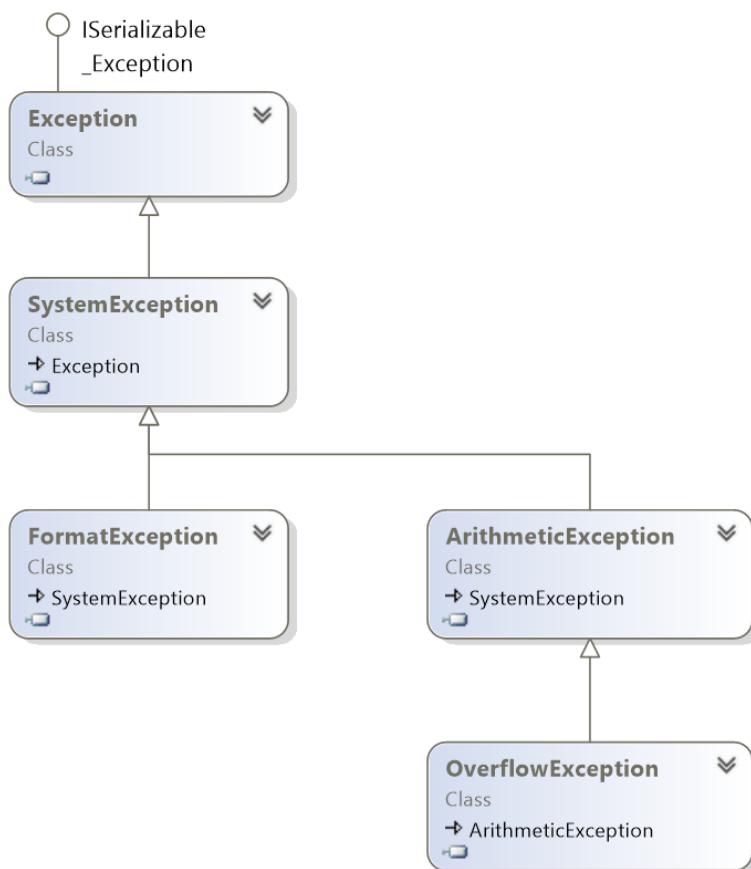


Abbildung 104: Auszug aus Klassenhierarchie von `FormatException` und `OverflowException`

5.14.6 Wann sollte man Exceptions auslösen?

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ND1uaRewg1E>.

Im Wesentlichen gibt es zwei Fälle, bei denen man Exceptions werfen sollte:

- Guard Clauses (dt. Schutzabfragen) – dies sind `if` Blöcke, die zu Beginn einer Methode, die Parameter entgegennimmt, Exceptions auslösen. Diese `if` Abfragen werden genutzt, um die Parameterwerte auf Richtigkeit zu überprüfen. Ist dies nicht der Fall, beispielsweise weil ein Index über die Größe des Arrays hinausgeht oder weil eine Objektreferenz mit `null` übergeben wurde, fliegt eine Exception. Dies ist der häufigste Einsatz für Exceptions.

- Ansonsten löst man häufig dann eine Ausnahme aus, wenn man erkennt, dass ein Objekt in einem fehlerhaften Zustand übergehen würde – dies ist meistens durch bestimmte Nebenbedingungen geschuldet (bspw. durch den Zustand eines anderen Objekts). Dies ist meistens ein Hinweis auf schlechtes Klassendesign, wenn solche Schritte notwendig sind. Manchmal ist das aber auch nicht vermeidbar, v.a. wenn man gegen Peripherie programmiert oder die Prozessgrenze verlässt.

In der folgenden Abbildung sehen Sie, wie Guard Clauses aussehen können:

```

public class Gehaltsrechner
{
    public decimal BerechneGehaltserhöhung(Mitarbeiter mitarbeiter)
    {
        Guard Clause verhindert  
invalide Werte bei Parametern
        if (mitarbeiter == null) throw new ArgumentNullException("mitarbeiter");

        if (mitarbeiter.Mitarbeiterverhältnis == Mitarbeiter.Management)
            return mitarbeiter.AktuellesGehalt * 1.5m;
        else if (mitarbeiter.Mitarbeiterverhältnis == Mitarbeiter.Teamleiter)
            return mitarbeiter.AktuellesGehalt * 1.1m;
        else if (mitarbeiter.Mitarbeiterverhältnis == Mitarbeiter.Angestellter)
            return mitarbeiter.AktuellesGehalt * 1.05m;
        else
            return mitarbeiter.AktuellesGehalt * 0.8m;
    }
}

```

Abbildung 105: Ein Guard Clause zu Beginn einer Methode

In Abbildung 105 sehen Sie, dass zu Beginn der Methode `Gehaltsrechner.BerechneGehaltserhöhung` ein Guard Clause eingefügt würde. Dieser überprüft den Parameter auf null und löst eine `ArgumentNullException` aus, wenn die Methode genauso aufgerufen wurde. Dies sorgt dafür, dass die eigentliche Funktionalität der Methode nicht ausgeführt wird und entspricht damit dem sog. Fail-Fast-Prinzip:

Das Fail-Fast-Prinzip besagt, dass eine Methode schnellstmöglich eine Exception werfen sollte, wenn einer oder mehrere Parameter der Methode einen fehlerhaften Wert haben. Dies wird erreicht, indem man Guard Clauses zu Beginn der Methode einsetzt – üblicherweise einen pro Parameter. Nach den Guard Clauses wird die eigentliche Funktionalität der Methode aufgeführt.

Das Fail-Fast-Prinzip ist besonders wichtig, da in der eigentlich wichtigen Logik der Funktion Seiteneffekte ausgelöst werden könnten. Wird erst danach eine Exception aufgrund eines fehlerhaften Parameters ausgelöst, ist es wahrscheinlich, dass sich das Objekt bedingt durch den ausgelösten Seiteneffekt in einem fehlerhaften Zustand befindet.

5.14.7 Wann sollte man Exceptions behandeln?

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=0aVo87L-kpk>.

Auch zur Anwendung von `try-catch` Blöcken gibt es einige Richtlinien, die man einhalten sollte:

- Behandeln Sie niemals durch Guard Clauses ausgelöste Exceptions – sorgen Sie stattdessen dafür, dass diese Exceptions niemals auftreten, indem Sie ihre Parameterwerte richtig angeben, wenn Sie die entsprechende Methode aufrufen.
- Behandeln Sie niemals Exceptions in derselben Methode, in der sie ausgelöst werden. Schreiben Sie stattdessen ihre Klassen so, dass sie mehrere kleine Klassen haben, die Grundfunktionalitäten durchführen und gegebenenfalls Exceptions auslösen. Diese Klassen

werden in einer sog. orchestrierenden Klasse gekapselt, welche die einzelnen Grundfunktionalitäten in bestimmter Reihenfolge ausführt, Exceptions fängt und die Fehler an bestimmte andere Objekte weiterleitet – bspw. zum Logging und für Hinweisnachrichten an den Nutzer.

- Setzen Sie eben genanntes Schema vor allem dort ein, wo die Prozessgrenze überschritten wird: bspw. bei Interaktion mit dem Nutzer sowie bei Festplatten-, Datenbank- oder Netzwerkzugriffen. All Input is Evil – schützen Sie ihr Programm vor falsch entgegengenommenen Werten.

Grundsätzlich gilt, dass man **try-catch** Blöcke so häufig wie nötig, aber so selten wie möglich einsetzen sollte, da sie einen beträchtlichen Einfluss auf die Performance haben können und die Lesbarkeit des Codes verschlechtern. Verwenden Sie niemals **try-catch** ohne einen gewissen Grund, erst recht nicht, damit eine unerwartete Exception „geschluckt“ wird und ihr Programm nicht abstürzt.

5.15 Collections und Generics

5.15.1 List<T> – die am häufigsten benutzte Collection

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=8fzrR2cD2uw>.

In Abschnitt 4.3.7 haben wir bei der Betrachtung von Arrays auch aufgeführt, dass diese im objektorientierten Programmieralltag kaum noch eingesetzt werden. Ihre größten Nachteile sind:

- Die statische Größe, d.h. einmal initialisierte Arrays halten eine feste Anzahl von Stellen und können nicht mehr verändert werden.
- C# Arrays wissen im Vergleich zu Arrays in C, wie viele Stellen sie umfassen, allerdings kann man nicht ohne weiteres herausfinden, welche dieser Stellen tatsächlich belegt sind (d.h. wie viele Elemente im Array gehalten werden).
- Wenn ein Element an einer bestimmten Stelle im Array entfernt wird, bleibt diese Stelle einfach leer, da nachfolgende Elemente nicht nachrutschen. Üblicherweise ist dieses Verhalten aber gewünscht, d.h. wenn über alle Elemente eines Arrays iteriert werden soll, sind die leeren Stellen im Normalfall nicht von Belang.

Sog. Collections (dt. Ansammlungen oder Kollektionen) lösen diese Probleme. Sie bieten eine sehr einfache API an und kapseln üblicherweise einen Array, in dem sie Elemente verwalten. Diesen Array bekommt der Nutzer der Klasse allerdings nie zu Gesicht (Information Hiding, Datenhoheit). Der bekannteste und im Programmieralltag wohl am häufigsten angewandte Collectiontyp ist `System.Collections.Generic.List<T>`. Wie man diese Klasse benutzt, schauen wir uns im folgenden Beispiel an:

```
static void Main()
{
    List<int> zahlen = new List<int>();

    zahlen.Add(47);
    zahlen.Add(5);
    zahlen.Add(21);
    zahlen.Add(5);

    zahlen.Insert(2, 11);

    GibZahlenAus(zahlen);

    zahlen.Remove(21);
    zahlen.Remove(5);
    zahlen.RemoveAt(2);

    GibZahlenAus(zahlen);

    static void GibZahlenAus(List<int> zahlen)
    {
        Console.WriteLine("Anzahl Elemente: {0}", zahlen.Count);
        foreach (int zahl in zahlen)
        {
            Console.WriteLine(zahl);
        }
        Console.WriteLine();
    }
}
```

Neue Liste instanziieren

Mit `List<T>.Add` werden Elemente ans Ende der Collection angefügt

Mit `List<T>.Insert` können Elemente an eine bestimmte Position eingefügt werden

Mit `List<T>.Remove` und `List<T>.RemoveAt` werden Elemente analog zu `Add` und `Insert` aus der Collection entfernt

Mit `List<T>.Count` kann man die Anzahl der Elemente in einer Collection abfragen

`List<T>` kann genauso wie ein Array in `foreach` und `for` Schleifen verwendet werden

Abbildung 106: `List<T>` im Einsatz

In Abbildung 106 sehen Sie, wie eine Liste instanziert und genutzt wird. Dabei kommen die Methoden Add, Insert, Remove, RemoveAt sowie die Eigenschaft Count zum Einsatz. In der folgenden Tabelle finden sie die wichtigsten Klassenmitglieder von `List<T>` nochmals aufgelistet:

Klassenmitglied	Bemerkung
Add	Fügt ein Element ans Ende der Collection hinzu.
Capacity	Gibt an, wie groß der von der Collection gekapselte Array ist. Der interne Array wird automatisch durch einen größeren Array ausgetauscht, wenn die Kapazität überschritten wird.
Clear	Entfernt alle Elemente aus der Collection, die im Anschluss leer ist.
Contains	Überprüft, ob ein bestimmtes Mitglied mindestens einmal in der Collection vorkommt.
Count	Gibt an, wie viele Elemente in der Collection enthalten sind.
Insert	Fügt ein Element an einer bestimmten Stelle in der Collection ein.
IndexOf	Sucht nach einem bestimmten Element in der Collection und gibt dessen Index zurück.
Item (Indexoperator)	Mit der Indexschreibweise kann man genauso wie bei Arrays auf Elemente an bestimmten Positionen zugreifen.
Remove	Sucht nach einem bestimmten Element und entfernt dieses beim ersten Auftreten aus der Collection.
RemoveAt	Entfernt ein Element an einer bestimmten Stelle.
Reverse	Dreht die Reihenfolge der Elemente in der Collection um.
Sort	Sortiert die Elemente der Collection.

Abbildung 107: Wichtige Members der Klasse `List<T>`

Die in Abbildung 107 sind die Members, mit denen man im Programmieralltag mit Listen am häufigsten in Berührung kommt. `List<T>` hat aber noch viele weitere Mitglieder, die Sie u.a. in der [MSDN Library](#) anschauen können.

`List<T>` löst dabei alle Probleme, die wir bei Arrays vorhin angesprochen haben:

- Zu einer Liste können beliebig viele Elemente hinzugefügt werden, ohne dass man sich über die Größe Gedanken machen muss.
- Listen kennen nicht nur ihre derzeitige Kapazität, sondern auch, wie viele Elemente tatsächlich in ihnen enthalten sind.
- Wenn ein Element aus der Liste entfernt wird, entsteht keine leere Stelle, sondern die nachfolgenden Elemente werden um eine Stelle nach vorne verschoben.

Ein interessanter Punkt, den wir bis jetzt noch nicht angesprochen haben, ist das `<T>` in `List<T>`: dies bezeichnet man als Generic, was letztendlich ein Platzhalter für einen bestimmten Typen ist. Generics muss man beim Instanziieren auflösen, im obigen Beispiel wurde das getan, indem wir statt `List<T>` eben `List<int>` beim Variabtentyp, beim Konstruktorauftrag und beim Parametertyp geschrieben haben. Letztendlich bedeutet dies nichts anderes als dass diese Liste Integerwerte verwaltet. Wollte man eine Liste von Mitarbeitern nutzen, müsste man beim Instanziieren `List<Mitarbeiter>` angeben. Was Generics genau sind und worin ihre Vorteile liegen, schauen wir uns in den kommenden Abschnitten genauer an, in denen wir eine eigene Datenstruktur implementieren, die ebenfalls mehrere Elemente eines Typs verwaltet: die Klasse `Stack<T>`.

5.15.2 Was sind Generics?

Die im Titel gestellte Frage möchte ich klären, indem wir schrittweise eine Klasse `Stack<T>` aufbauen. Ein Stack (dt. Stapel) ist eine Datenstruktur ähnlich einer Liste, die aber einen eingeschränkteren Funktionsumfang hat:

- Wenn man ein Element in den Stapel einfügt, wird dieses immer hinten angefügt (ein Einfügen an einer beliebigen Stelle ist nicht möglich).
- Beim Entfernen wird immer das hinterste Element des Stacks zurückgegeben und aus der Datenstruktur entfernt.

Damit ist ein Stack eine Datenstruktur, die nach dem LIFO (Last In First Out) Prinzip arbeitet – das letzte Element, das hinzugefügt wurde, ist das erste, was beim Entfernen zurückgegeben wird (im Gegensatz dazu arbeitet eine Schlange (engl. Queue) nach dem FIFO (First In First Out) Prinzip).

5.15.2.1 Ein Stack für Strings

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=bJcu3HKYZVk>.

Die eben aufgeführte Funktionalität könnte man wie folgt in einer Klasse umsetzen, die **string** Werte verwaltet:

```
public class StringStack
{
    private readonly string[] _array; ----- StringStack kapselt intern einen Array, in dem die einzelnen Werte, die man
                                         zum Stack hinzufügt, eingetragen werden

    public StringStack(int kapazität) ----- Die Größe des Arrays wird über den Konstruktor entgegen genommen
    {
        if (kapazität < 2)
        {
            var message = string.Format("Der Kapazitätswert {0} muss größer als 2 sein", kapazität);
            throw new ArgumentOutOfRangeException("kapazität", message);
        }

        _array = new string[kapazität];
    }

    public void FügeElementHinzu(string element)
    {
        if (AnzahlElemente > _array.Length)
            throw new InvalidOperationException("Der Stack ist voll");

        _array[AnzahlElemente] = element;
        AnzahlElemente++;
    }

    public string EntferneElement()
    {
        if (AnzahlElemente == 0)
            throw new InvalidOperationException("Der Stack ist leer");

        AnzahlElemente--;
        var element = _array[AnzahlElemente];
        _array[AnzahlElemente] = null;
        return element;
    }

    public int AnzahlElemente { get; private set; } ----- AnzahlElemente wird sowohl zur Info für den Nutzer als auch als
                                         Laufvariable eingesetzt
}
```

Das Diagramm zeigt die Klasse StringStack mit ihren Methoden und Eigenschaften. Es gibt drei hervorgehobene Bereiche, die durch Pfeile auf die entsprechenden Zeilen im Code zeigen:

- Guard Clauses:** Ein Kasten rechts beschriftet mit "Guard Clauses schützen das Objekt vor einem fehlerhaften Status" umschließt die if-Bedingungen in den Methoden FügeElementHinzu, EntferneElement und der Konstruktor. Ein Pfeil weist von diesem Kasten auf die entsprechenden Zeilen im Code.
- Eigenschaftsdokumentation:** Ein Kasten rechts beschriftet mit "AnzahlElemente wird sowohl zur Info für den Nutzer als auch als Laufvariable eingesetzt" umschließt die Zeile "public int AnzahlElemente { get; private set; }". Ein Pfeil weist von diesem Kasten auf die entsprechende Zeile im Code.
- Kommentare:** Ein Kasten rechts beschriftet mit "StringStack kapselt intern einen Array, in dem die einzelnen Werte, die man zum Stack hinzufügt, eingetragen werden" umschließt den Kommentar im Konstruktor. Ein Pfeil weist von diesem Kasten auf die entsprechende Zeile im Code.

Abbildung 108: Eine Stackklasse für Strings

In Abbildung 108 sehen Sie die Klasse **StringStack**, deren zwei Methoden **FügeElementHinzu** und **EntferneElement** und die Eigenschaft **AnzahlElemente** die oben genannte API für einen Stack darstellen. Intern verwendet die Klasse einen String-Array, in dem die Werte, die der Stack halten soll, abgelegt werden. Dieser Array wird im Konstruktor initialisiert, wobei der Parameter **kapazität** die Größe des Arrays festlegt (anders ausgedrückt: beim Instanziieren kann der Nutzer angeben, wie groß der intern verwendete Array sein soll).

Sämtliche Methoden, die von außen aufgerufen werden können (die also **public** sind), sind mit Guard Clauses geschützt, damit der Stack nicht in einen falschen Zustand übergeht. Dies könnte

beispielsweise passieren, wenn alle Stellen im Array belegt sind und ein weiteres Element zum Stack hinzugefügt wird oder eine falsche Kapazität beim Konstruktorauftruf angegeben wird. Beim Guard Clause im Konstruktor wird zusätzlich die `string.Format` Methode benutzt, bei der man im String Platzhalter mit `{0}` definieren kann, die dann mit dem Wert von kapazität zur Laufzeit ersetzt werden.

Die Klasse kann dann wie folgt eingesetzt werden:

```
static void Main()
{
    StringStack stringStack = new StringStack(5);

    for (int i = 0; i < 5; i++)
    {
        stringStack.FügeElementHinzu("A" + (i + 1));
    }

    while(stringStack.AnzahlElemente > 0)
        Console.WriteLine(stringStack.EntferneElement());
}
```

Abbildung 109: Die Klasse `StringStack` einsetzen

In Abbildung 109 wird ein `StringStack`-Objekt erstellt, die Werte „A1“ bis „A5“ zum Stack hinzugefügt und im Anschluss alle Elemente wieder aus dem Stack entfernt. Dabei wird die komplette API eingesetzt, die wir vorher angesprochen haben. Ein wichtiges Problem bleibt hier aber noch ungelöst: wir können diese Stackklasse ausschließlich für den Typ `string` einsetzen – wenn wir Ganz- oder Gleitkommazahlen, boolesche Werte oder beliebige andere Typen einsetzen möchten, müssen wir eine weitere Stackklasse für den entsprechenden Typ erstellen, was wir im folgenden Abschnitt tun werden.

5.15.2.2 *Don't repeat yourself – ein Stack für int Werte*

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=VG07E-cL2Ws>.

Im letzten Abschnitt haben wir uns den Stack für `string` angeschaut und festgestellt, dass dieser bspw. für `int` Werte nicht eingesetzt werden kann. In diesem Abschnitt schauen wir uns deshalb die Klasse `IntStack` an und betrachten, was sich im Vergleich zu `StringStack` verändert hat. In Abbildung 110 sehen Sie, dass die Unterschiede sich ausschließlich auf die Typangaben beruhen, ansonsten sind die beiden Klassen vollkommen identisch. Bei dieser Konstellation sollte man aufpassen, denn hier wurde gegen das sog. DRY (Don't Repeat Yourself) Prinzip verstößen.

Das DRY Prinzip besagt, dass Codeduplikierungen vermieden werden sollten. Stattdessen sollte jedes Stück Information in einem System an genau einer Stelle unmissverständlich und verbindlich beschrieben sein. Wenn Änderungen notwendig werden, so muss man diese nur an einer Stelle durchführen.

Mit diesen Informationen können wir folgende Frage stellen: können wir eine Stackklasse schreiben, die unabhängig von genutzten Typ ist? In den beiden folgenden Abschnitten werden wir dieser Frage nachgehen und uns zwei Möglichkeiten ansehen, wie man dieses Ziel erreichen kann.

```

public class IntStack
{
    private readonly int[] _array;

    public IntStack(int kapazität)
    {
        if (kapazität < 2)
        {
            var message = string.Format("Der Kapazitätswert {0} muss größer als 2 sein", kapazität);
            throw new ArgumentOutOfRangeException("kapazität", message);
        }

        _array = new int[kapazität];
    }

    public void FügeElementHinzu(int element)
    {
        if (AnzahlElemente > _array.Length)
            throw new InvalidOperationException("Der Stack ist voll");

        _array[AnzahlElemente] = element;
        AnzahlElemente++;
    }

    public int EntferneElement()
    {
        if (AnzahlElemente == 0)
            throw new InvalidOperationException("Der Stack ist leer");

        AnzahlElemente--;
        var element = _array[AnzahlElemente];
        _array[AnzahlElemente] = 0;
        return element;
    }

    public int AnzahlElemente { get; private set; }
}

```

Unterschied zur Klasse StringStack: der Typ string wurde durch int ausgetauscht

Abbildung 110: Eine Stackklasse für int Werte

5.15.2.3 Alles ist object – der Stack für object

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=oCrOmbeTKv8>.

In Abschnitt 5.12.8 haben wir uns mit der Basisklasse **object** beschäftigt und festgestellt, dass diese Klasse implizit oder explizit die Basisklasse für alle anderen Klassen darstellt. Das gilt nicht nur für Referenztypen (zu denen alle Klassen zählen), sondern auch für Wertetypen, also bspw. die primitiven Datentypen **int**, **double**, **bool** oder **char**. Folglich könnte man eine allgemeine Klasse **Stack** mit dem Typen **object** bauen, genau nach dem gleichen Muster, wie wir beim **IntStack** vorgegangen sind: wir nehmen den Source Code von **StringStack** und tauschen an den vier entsprechenden Stellen den Typ **string** durch **object** aus.

In Abbildung 111 sehen Sie diese Klasse. Mit ihr können nun Stack-Objekte instanziert werden, die für die unterschiedlichsten Datentypen funktionieren, womit wir das DRY-Prinzip eingehalten haben und nicht für jeden Typ, den wir in einem Stack verarbeiten wollen, eine eigene Stackklasse implementieren müssen. Allerdings stellt sich ein neues Problem ein, wenn wir Stack einsetzen, wie man in Abbildung 112 sehen kann.

```

public class Stack
{
    private readonly object[] _array;

    public Stack(int kapazität)
    {
        if (kapazität < 2)
        {
            var message = string.Format("Der Kapazitätswert {0} muss größer als 2 sein", kapazität);
            throw new ArgumentOutOfRangeException("kapazität", message);
        }

        _array = new object[kapazität];
    }

    public void FügeElementHinzu(object element)
    {
        if (AnzahlElemente > _array.Length)
            throw new InvalidOperationException("Der Stack ist voll");

        _array[AnzahlElemente] = element;
        AnzahlElemente++;
    }

    public object EntferneElement()
    {
        if (AnzahlElemente == 0)
            throw new InvalidOperationException("Der Stack ist leer");

        AnzahlElemente--;
        var element = _array[AnzahlElemente];
        _array[AnzahlElemente] = null;
        return element;
    }

    public int AnzahlElemente { get; private set; }
}

```

Unterschied zur Klasse
StringStack: der Typ string wurde
durch object ausgetauscht

Abbildung 111: Stackklasse, die den Typ object verwendet

```

static void Main()
{
    Stack intStack = new Stack(5);
    intStack.FügeElementHinzu(1);
    intStack.FügeElementHinzu(42);

    int element1 = intStack.EntferneElement(); ←

    Stack stringStack = new Stack(5);
    stringStack.FügeElementHinzu("Hallo");
    stringStack.FügeElementHinzu("Welt");

    string element2 = intStack.EntferneElement(); ←

    intStack.FügeElementHinzu("String im Int Stack");
}

```

EntferneElement hat
Rückgabetyp object,
allerdings wird jeweils int
oder string erwartet

Es ist möglich, andere
Datentypen zum IntStack
hinzuzufügen

Abbildung 112: Fehlende Typsicherheit bei Klasse Stack

In Abbildung 112 sehen Sie, dass beim Einsatz der Klasse Stack Probleme auftreten, die man intuitiv nicht erwartet hätte:

- In der Variablen `intStack` soll ein Stack für `int` Werte, in der Variablen `stringStack` ein Stack für `string` Werte angelegt werden. Wir sehen ebenfalls, dass beim Einfügen von

Elementen mithilfe der Methode `FügeElementHinzu` in beiden Fällen keine Probleme auftreten. Das ist nicht der Fall beim Entfernen von Elementen: die Methode `EntferneElement` gibt einen Wert vom Typ `object` zurück, wir als Nutzer erwarten allerdings (intuitiv) einen Wert vom Typ `int`. Wie wir aus Abschnitt 4.3.6 wissen, ist hier eine implizite Konvertierung nicht möglich. Wir müssten hier einen expliziten Cast durchführen, um wieder zurück zum Typ `int` zu kommen.

- Des Weiteren ist es möglich, in den `intStack` Strings einzufügen, wie die unterste markierte Anweisung zeigt. Auch dieses Verhalten erwartet man als Nutzer (intuitiv) nicht, die Möglichkeit dazu besteht aber, da `FügeElementHinzu` einen Parameter vom Typ `object` erwartet und somit jedes beliebige Objekt oder jede beliebige Struktur hinzugefügt werden kann.

Diese beiden Punkte kann man wie folgt zusammenfassen:

Bei der Klasse `Stack` ist die Typsicherheit verloren gegangen. Da Parameter- und Rückgabetypen `object` nutzen, muss der Nutzer sich bewusst sein, dass beliebige Objekte zum Stack hinzugefügt werden können und entfernte Elemente erst wieder in den entsprechenden Typen gecastet werden müssen – beides kann leicht zu Fehlern führen, die der Compiler beim Erstellvorgang nicht erkennen kann.

Ein weiterer Punkt ist, dass bei Wertetypen, die als `object` interpretiert werden, sog. Boxing und Unboxing auftritt. Wie wir bereits wissen, sind die Variablenwerte von Referenztypen immer auf den Heap zu finden, die Variablenwerte von Wertetypen direkt auf dem Stack. Wird dann bspw. ein `int` Wert als `object` interpretiert, so wird dieser Wert in ein Objekt verpackt und auf dem Heap verschoben, was man als Boxing bezeichnet. Bei einem expliziten Cast zurück zum Wertetyp tritt genau das Gegenteil auf: der Wert wird aus dem vorher genutzten Objekt ausgepackt und wieder auf den Stack verschoben. Wie dies genau funktioniert, schauen wir uns in einem späteren Kapitel genauer an, allerdings sei so viel gesagt: Boxing und Unboxing kann einen erheblichen Einfluss auf die Performance haben. Dies sollte man wo es geht vermeiden.

Es bleibt die Frage, wie man eine Stackklasse schreiben kann, die sowohl für beliebige Datentypen einsetzbar ist, aber auch Typsicherheit bietet, und genau für diesen Fall kann man Generics einsetzen, wie wir im nächsten Abschnitt sehen werden.

5.15.2.4 Der generische Stack – typsicher und DRY

Das Video zu diesem Abschnitt gibt es unter <https://www.youtube.com/watch?v=jCFVClioW9g>.

Was wir für die Stackklasse, die wir schrittweise in den letzten Abschnitten aufgebaut haben, eigentlich brauchen, ist ein Platzhalter für den Typen, dessen Werte vom Stack veraltet werden sollen. Und genau dies bieten Generics: sie sind Platzhalter für bestimmte Typen, die beim Einsatz (also dem Instanziieren der Klasse) festgelegt werden.

In Abbildung 113 sehen Sie, wie die Klasse `Stack` generisch geschrieben werden kann. Dazu wurde ein klassenweites Generic definiert, indem man nach dem Klassenbezeichner `<T>` schreibt, was nichts anderes ist als der Bezeichner für den Generic innerhalb von spitzen Klammern. Dieses T kann dann an genau den Stellen eingesetzt werden, wo wir sonst mit einem konkreten Typen agiert haben. Wie man sieht, wird an diesen Stellen einfach nur der Typ ausgetauscht mit dem Generic. Eine weitere Besonderheit ist die Angabe von `default(T)` in der Methode `EntferneElement`: hiermit holt man sich den Standardwert für einen bestimmten Typen, also bspw. 0 bei `int`, 0.0 bei `double`, `false` bei `bool` oder `null` bei allen Referenztypen. Diese Angabe ist hier notwendig, da wir nicht wissen, ob für T ein Werte- oder Referenztyp eingesetzt wird.

```

public class Stack<T> — Definition eines Generic, der über die ganze Klasse hinweg genutzt werden kann
{
    private readonly T[] _array;

    public Stack(int kapazität)
    {
        if (kapazität < 2)
        {
            var message = string.Format("Die Kapazitätswert {0} muss größer als 2 sein", kapazität);
            throw new ArgumentOutOfRangeException("kapazität", message);
        }

        _array = new T[kapazität];
    }

    public void FügeElementHinzu(T element)
    {
        if (AnzahlElemente > _array.Length)
            throw new InvalidOperationException("Der Stack ist voll");

        _array[AnzahlElemente] = element;
        AnzahlElemente++;
    }

    public T EntferneElement()
    {
        if (AnzahlElemente == 0)
            throw new InvalidOperationException("Der Stack ist leer");

        AnzahlElemente--;
        var element = _array[AnzahlElemente];
        _array[AnzahlElemente] = default(T);
        return element;
    }

    public int AnzahlElemente { get; private set; }
}

```

T wird jetzt anstatt eines konkreten Typen eingesetzt

Abbildung 113: Die generische Stackklasse

```

static void Main()
{
    Stack<int> intStack = new Stack<int>(5);

    intStack.FügeElementHinzu(1);
    intStack.FügeElementHinzu(42);

    int intElement = intStack.EntferneElement(); ← Rückgabetyp muss nicht mehr gecastet werden

    → Stack<string> stringStack = new Stack<string>(5);
    stringStack.FügeElementHinzu("Hallo");
    stringStack.FügeElementHinzu("Welt");

    string stringElement = stringStack.EntferneElement(); ←

    intStack.FügeElementHinzu(stringElement); ← In intStack können keine Werte von anderen Typen hinzugefügt werden
}

```

Bei klassenweiten Generics muss dieser bei Typangaben aufgelöst werden

Rückgabetyp muss nicht mehr gecastet werden

In intStack können keine Werte von anderen Typen hinzugefügt werden

Abbildung 114: Einsatz der generischen Klasse Stack

In Abbildung 114 ist zu sehen, dass der Einsatz des Generic all unsere Probleme der nicht-generischen Klasse `Stack` löst. Bei Instanziieren gibt man den jeweiligen Typ, den der Stack verwalten soll an, indem man nach dem Bezeichner für die Klassen den eigentlichen Typen für den

Generic setzt (in unserem Fall `Stack<int>` sowie `Stack<string>`). Hierdurch muss man Rückgabetypen nicht mehr in den eigentlichen Typ casten und erhält die Sicherheit, dass ein `Stack<string>` (sprich „Stack of String“) tatsächlich auch nur `string` Werte verwaltet. Jedweder Versuch, dies zu umgehen, führt zu einem Compilerfehler, wie im Beispiel zu sehen ist.

Auch wenn `Stack<T>` nur ein Generic benutzt, können prinzipiell beliebig viele Generics auf einem Typ definiert werden. Diese müssen dann unterschiedliche Bezeichner haben, die man konventionsgemäß alle mit großem T beginnt, bspw. beim Typen `Dictionary< TKey, TValue >`, das ein Objekt über einen Schlüssel (Key) ansprechen lässt. Wird nur ein Generic bei einem Typen verwendet, so nennt man ihn meistens einfach T.

Generics sieht man sehr häufig bei Collectionstypen, allerdings können Sie bei jeder beliebigen Klasse eingesetzt werden. Im Allgemeinen bieten Sie folgende Vorteile:

- Sie erlauben das Schreiben von Typen oder Methoden, die für mehrere Typen anwendbar sind und dennoch Typsicherheit bieten (bspw. sind in einer `List<string>` ausschließlich `string` Werte zu finden).
- Durch den Einsatz von Generics kann der Compiler beim Erstellvorgang überprüfen, ob der angegebene Code korrekt ist (Compilerfehler sind Laufzeitfehlern ganz allgemein immer vorzuziehen).
- Bei Wertetypen tritt bei Generics kein Boxing bzw. Unboxing auf, was die Performance erhöht.
- Generics fördern das DRY-Prinzip, da man denselben Code für beliebig viele unterschiedliche Typen nutzen kann (bspw. wird für eine `List<DateTime>` die gleiche Klasse genutzt wie für `List<Mitarbeiter>`).

Es gibt noch weitere Möglichkeiten, Generics zu konfigurieren, diese schauen wir uns aber in einem späteren Kapitel an. Nachdem wir jetzt aber wissen, was Generics sind und wie sie funktionieren, schauen wir uns weitere Collectionklassen an.

5.15.3 Weitere Klassen für Collections

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=teakRWyIJLk>.

Im Namespace `System.Collections.Generic` findet man neben `List<T>` auch andere Klassen, die mehrere Objekte desselben Typs verwalten. Diese sind in der folgenden Tabelle aufgelistet.

Klasse	Bemerkung
<code>Dictionary< TKey, TValue ></code>	Repräsentiert ein Mapping zwischen einem Schlüssel und einem dazugehörigen Wert. Dabei müssen die Schlüsselwerte eindeutig sein. Beispielsweise könnte ein Mitarbeiter anhand seiner ID in einem <code>Dictionary<int, Mitarbeiter></code> verwaltet werden. Der Zugriff mittels Index erfolgt dann über den Schlüsselwert und ist deutlich schneller als das Suchen nach einem bestimmten Objekt mithilfe einer Schleife bei einer Liste.
<code>HashSet<T></code>	Ein sog. Set (dt. Menge) ist eine Collection, die keine Duplikate enthält. Ein <code>HashSet<int></code> darf bspw. nicht zweimal den Wert 42 enthalten. Beim Einfügen nutzt diese Collection deswegen die <code>GetHashCode</code> Methode eines Objekts, um schneller zwei Objekte miteinander zu vergleichen.

<code>LinkedList<T></code>	Repräsentiert eine verkettete Liste, d.h. eine Collection, in der die Elemente jeweils den jeweiligen Vorgänger und den Nachfolger kennen.
<code>List<T></code>	Dieser Typ repräsentiert eine Liste und ist der am häufigsten eingesetzte Collectiontyp.
<code>Queue<T></code>	Repräsentiert eine Queue (dt. (Warte-)Schlange), die umgedreht analog zum Stack funktioniert: auch hier ist nur das Einfügen am Ende möglich, wenn aber ein Element entfernt wird, ist es immer das Erste. Die Queue funktioniert deshalb nach dem FIFO (First In First Out) Prinzip.
<code>SortedDictionary<TKey, TValue></code>	Repräsentiert wie <code>Dictionary<TKey, TValue></code> ein Mapping, dessen Schlüsselwerte aber zusätzlich sortiert sind.
<code>SortedSet<T></code>	Repräsentiert wie <code>HashSet<T></code> eine Menge ohne Duplikate, wobei dessen Werte aber sortiert sind.
<code>Stack<T></code>	Repräsentiert eine LIFO (Last In First Out) Datenstruktur, die wir ausgiebig in den letzten Abschnitten besprochen haben.

Abbildung 115: Wichtige Collectionstypen in `System.Collections.Generic`

Auch wenn in Abbildung 115 viele Typen aufgelistet sind, nutzt man im Programmieralltag hauptsächlich `List<T>` und `Dictionary<TKey, TValue>`. Die weiteren Typen können je nach Bedarf für bestimmte Probleme eingesetzt werden.

Neben diesen Klassen, die alle Generics nutzen, gibt es auch noch die Klassen im Namespace `System.Collections`, welche allerdings keine Generics benutzen. Dies ist historisch bedingt: .NET unterstützt erst seit Version 2.0 Generics, vorher arbeiteten Collections immer mit dem Typ `object`, nach dem gleichen Muster wie in Abschnitt 5.15.2.3 gezeigt. Da diese Collections keine Typsicherheit bieten, ist es heutzutage nicht mehr sinnvoll, diese `System.Collections` Klassen einzusetzen. Der bekannteste Vertreter dieses Namensraums ist die Klasse `ArrayList`.

5.15.4 Collections über Interfaces ansprechen

5.15.4.1 Die Interfaces `IEnumerable<T>`, `ICollection<T>` und `IList<T>`

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=EaYs2Y06VAw>.

Wir haben in vorherigen Abschnitten bereits das Prinzip „Programmieren gegen Abstraktionen statt konkrete Typen“ angesprochen und auch für Collections gibt es bestimmte Interfaces, die anstatt der eigentlichen Klassen für Referenzen verwendet werden können. Dabei sind die Interfaces für Collections in drei Vererbungsstufen unterteilt:

1. Aufzählungen werden durch die Interfaces `IEnumerable<T>` und `IEnumerable` repräsentiert. Aufzählung heißt in diesem Sinne, dass man sich über Objekte dieser Collections einen sog. Enumerator (dt. Aufzähler) besorgen kann, mit dem man alle Elemente der Collection durchlaufen kann. Vorsicht: prinzipiell können Aufzählungen unendlich lang sein (sind Sie aber meistens nicht).
2. Als nächstes in der Hierarchie folgen die Interfaces `ICollection<T>` und `ICollection`. Ihre Aufgabe ist es, die Anzahl der Elemente in einer Collection abfragbar zu machen über die Eigenschaft `Count`. `ICollection<T>` bietet weiterhin die Möglichkeit, Elemente in eine Collection hinzuzufügen mit `Add`, zu entfernen mit `Remove` und die Existenz von Objekten in der Collection abzufragen mit `Contains`. Diese Methoden sind im nicht-generischen Fall erst bei `IList` zu finden.

3. Zuletzt gibt es die Interfaces `IList<T>` und `IList`. Diese erweitern Collections um die Möglichkeit, Elemente an bestimmten Indexstellen zu setzen, einzufügen oder auszulesen.

Üblicherweise verwendet man natürlich die generischen Varianten dieser Interfaces, um Collection-Objekte zu referenzieren. Die nicht-generischen Varianten sind hier nur der Vollständigkeit halber erwähnt. Die genannten Interfaces werden von den in Abschnitt 5.15.3 angegebenen Klassen natürlich so weit wie möglich implementiert. In Abbildung 116 sehen Sie diese Interfaces in ihrer Vererbungshierarchie.

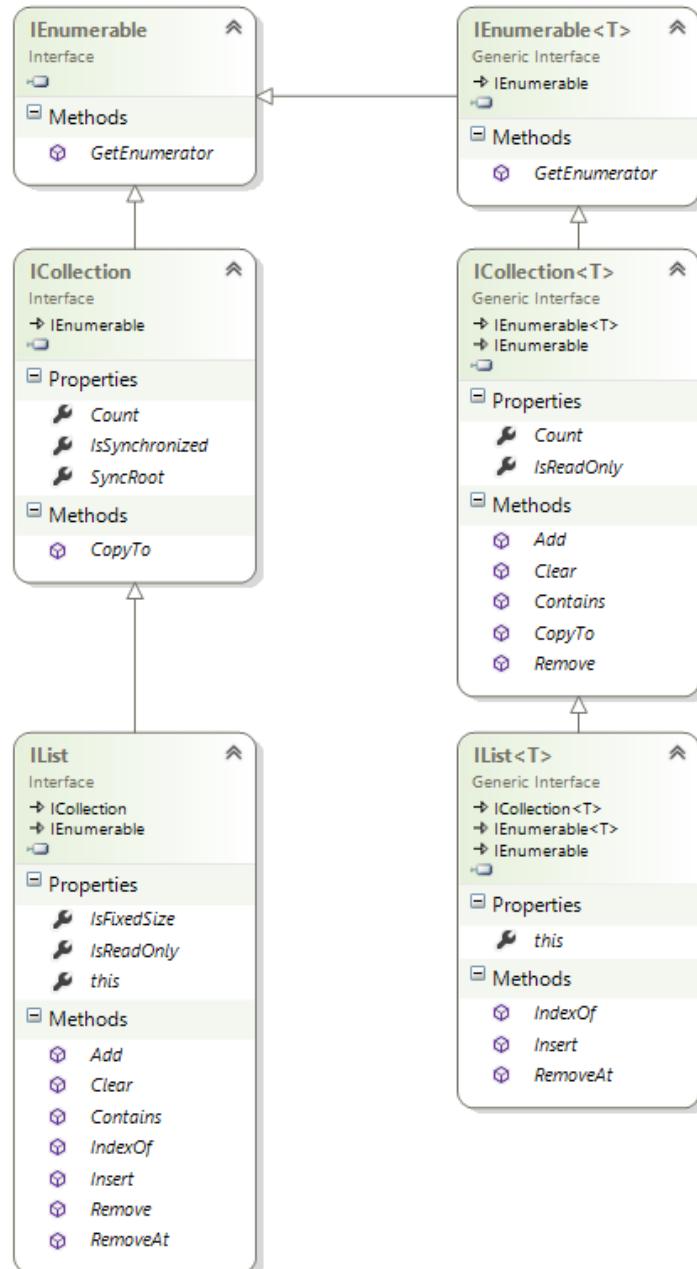


Abbildung 116: Die drei wichtigen Collectioninterfaces

5.15.4.2 `IEnumerable<T>` und `IEnumerable` im Detail (Level 200)

Das Video zu diesem Abschnitt finden Sie hier: <https://www.youtube.com/watch?v=qp9M7uq2n9M>.

Das Interface `IEnumerable<T>` möchte ich dabei nochmals in Betracht ziehen. Genau dieses Interface (und sein nicht-generisches Pendant) ist notwendig, damit eine Aufzählung in einer `foreach` Schleife durchlaufen werden kann. Dies wird in folgender Abbildung nochmals deutlich:

```
static void Main()                                Initialisierungssyntax für Collections
{
    var strings = new List<string> { "Hallo", "Welt" };
    GibStringsAus(strings);
}

static void GibStringsAus(IEnumerable<string> strings)
{
    foreach (var @string in strings)              Nach dem Schlüsselwort in muss bei
                                                foreach Schleifen ein Objekt stehen, das
                                                IEnumerable<T> oder IEnumerable
                                                implementiert
    {
        Console.WriteLine(@string);
    }
}
```

Abbildung 117: `foreach` Schleife benötigt `IEnumerable<T>`

In Abbildung 117 sehen Sie, dass es eine arrayähnliche Initialisierungssyntax für Collections gibt, mit der man Collection-Objekte mit einer Anweisung erstellen und gleichzeitig mit Werten befüllen kann, so wie es bei `strings` gemacht wird. Der springende Punkt des Beispiels ist jedoch der Fakt, dass beim Aufruf der Methode `GibStringsAus` die zuvor erstellte Liste über ihr Interface `IEnumerable<string>` angesprochen wird. Zumindest dieses Interface ist notwendig, damit ein Objekt nach dem `in` Schlüsselwort in einer `foreach` Schleife eingesetzt werden kann.

Wenn wir das Interface `IEnumerable<T>` genauer betrachten, dann ist auf ihm nur eine einzige Methode namens `GetEnumerator` definiert, bei der man als Rückgabewert einen `IEnumerator<T>` bekommt. Das ist der bereits oben erwähnte Aufzähler, mit dem man die Collection schrittweise durchlaufen kann. In folgender Abbildung sehen Sie die genaue Mitgliederdefinition von `IEnumerable<T>`:

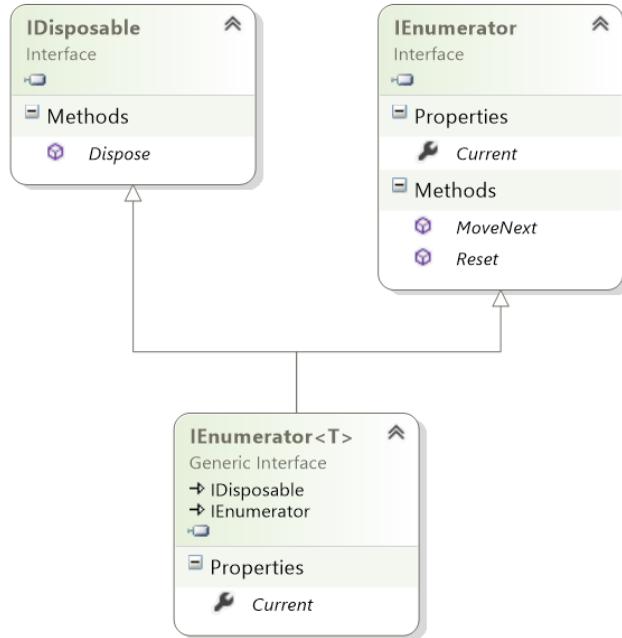


Abbildung 118: Das Interface `IEnumarator<T>`

Wenn man die gesamte Interfacevererbungshierarchie in Abbildung 118 betrachtet, setzt sich ein `IEnumarator<T>`-Objekt im Wesentlichen aus folgenden Klassenmitgliedern zusammen:

- Mit der Methode `MoveNext` wird der nächste Wert einer Auflistung geholt (anders ausgedrückt: der Enumerator geht eine Position in der Collection weiter). Standardmäßig muss man diese Methode auch einmal aufrufen, nachdem der Aufzähler instanziert wurde, damit er auf das erste Element einer Aufzählung schaut. `MoveNext` hat als Rückgabewert einen `bool`, der angibt, ob das Ende der Aufzählung erreicht wurde.
- Mit der Eigenschaft `Current` kann man auf das aktuelle Element der Aufzählung zugreifen. Dieses ist zu Beginn auf `default(T)` gesetzt, wenn `MoveNext` noch nicht aufgerufen wurde. Dasselbe passiert, wenn mit `MoveNext` das Ende der Aufzählung erreicht wurde.
- Mit der Methode `Reset` kann man den Enumerator zurücksetzen auf seinen Initialzustand. Man muss also nochmals `MoveNext` aufrufen, um wieder auf das erste Element zu schauen.

Zusätzlich dazu leitet `IEnumarator<T>` noch vom Interface `IDisposable` ab, auf das ich hier jetzt aber nicht weiter eingehen möchte. Kurz zusammengefasst kann man sagen, dass dieses Interface für das vorzeitige Freigeben von Ressourcen gedacht ist, noch bevor der Garbage Collector aktiv wird. Näheres zu diesem Thema besprechen wir beim Speichermanagement der CLR und beim Dateizugriff.

Mit diesem Hintergrundwissen möchten wir jetzt `IEnumarable<T>` in unsere bestehende Klasse `Stack<T>` implementieren. In Abbildung 119 sehen Sie dazu die modifizierte Variante der Klasse. Da `IEnumarator<T>` von der nicht-generischen Variante `IEnumerator` ableitet und beide die Methode `GetEnumerator` definieren, muss eine der beiden explizit implementiert werden, um den Namenskonflikt aufzulösen, was in `IEnumerator.Get.GetEnumerator` passiert. Bei der Methode `GetEnumerator` wird letztendlich nur eine neue Instanz der Klasse `ArrayEnumerator<T>` erstellt, die den Enumerator repräsentiert und die eigentliche Arbeit des Aufzählens durchführt. Diesem Enumerator wird beim Konstruktoraufdruff der Array übergeben, der intern von der Klasse `Stack<T>` verwendet wird.

```

public class Stack<T> : IEnumerable<T>
{
    private readonly T[] _array;           Stack<T> implementiert jetzt zusätzlich IEnumerable<T>

    public Stack(int kapazität)
    {
        if (kapazität < 2)
        {
            var message = string.Format("Die Kapazitätswert {0} muss größer als 2 sein", kapazität);
            throw new ArgumentOutOfRangeException("kapazität", message);
        }

        _array = new T[kapazität];
    }

    public void FügeElementHinzu(T element)
    {
        if (AnzahlElemente > _array.Length)
            throw new InvalidOperationException("Der Stack ist voll");

        _array[AnzahlElemente] = element;
        AnzahlElemente++;
    }

    public T EntferneElement()
    {
        if (AnzahlElemente == 0)
            throw new InvalidOperationException("Der Stack ist leer");

        AnzahlElemente--;
        var element = _array[AnzahlElemente];
        _array[AnzahlElemente] = default(T);
        return element;
    }

    public int AnzahlElemente { get; private set; }

    public IEnumarator<T> GetEnumarator()
    {
        return new ArrayEnumarator<T>(_array, AnzahlElemente);
    }

    IEnumarator IEnumerable.GetEnumarator()
    {
        return GetEnumarator();
    }
}

```

Abbildung 119: **IEnumerable<T>** in **Stack<T>** implementieren

Da, wie in Abbildung 119 zu sehen ist, nur minimale Änderungen an der Klasse **Stack<T>** notwendig sind, um **IEnumerable<T>** zu implementieren, liegt der Fokus auf der Klasse **ArrayEnumarator<T>**, in der das Interface **IEnumarator<T>** implementiert ist. In Abbildung 120 sieht man diese Klasse. Dabei erfüllen die Methoden **MoveNext** und **Reset** die geforderte Funktionalität, mit **Current** kann der Nutzer den aktuellen Wert, bei dem der Enumerator steht, auslesen. Intern merkt sich der Enumerator über das Feld **_aktuellePosition**, an welcher Stelle er sich gerade befindet. Über den Konstruktorparameter **anzahlElemente** kann der Ersteller des Enumerators angeben, wie viele Elemente des Arrays ausgegeben werden sollen. Die **Dispose** Methode bleibt leer, da wir keine nativen Ressourcen in dieser Klasse verwalten, die wir in dieser Methode freigeben müssten.

```

public class ArrayEnumerator<T> : IEnumerator<T>
{
    private readonly T[] _array;
    private readonly int _anzahlElemente;
    private T _current;
    private int _aktuellePosition = -1;

    public ArrayEnumerator(T[] array, int anzahlElemente)
    {
        if (array == null) throw new ArgumentNullException("array");

        _array = array;
        _anzahlElemente = anzahlElemente;
    }

    public bool MoveNext() MoveNext setzt die aktuelle Position um eins
    {                                              nach vorne, bis das Arrayende erreicht ist
        if (_aktuellePosition + 1 >= _anzahlElemente)
        {
            _current = default(T);
            return false;
        }

        _aktuellePosition++;
        _current = _array[_aktuellePosition];
        return true;
    }

    public void Reset() Reset setzt die aktuelle Position vor das erste
    {                                              Element des Arrays
        _aktuellePosition = -1;
        _current = default(T);
    }

    public void Dispose() { }

    public T Current { get { return _current; } }

    object Ienumerator.Current { get { return Current; } }
}

```

Abbildung 120: Die Klasse `ArrayEnumerator<T>`

Nachdem wir die Klasse `Stack<T>` um diese Funktionalität erweitert haben, können wir sie in `foreach` Schleifen einsetzen, um durch alle Elemente, die der Stack verwaltet, zu iterieren.

```

static void Main()
{
    var stack = new Stack<string>(5);
    stack.FügeElementHinzu("Hallo");
    stack.FügeElementHinzu("Welt");

    foreach (var element in stack) Da Stack<T> das Interface
    {                                     Ienumerable<T> implementiert,
        Console.WriteLine(element);      können wir mit einer foreach
                                         Schleife durch diese Collection
                                         iterieren
    }
}

```

Abbildung 121: `Stack<T>` im Einsatz mit `foreach` Schleife

5.15.4.3 Die Funktionsweise der foreach Schleife (Level 200)

Das Video für diesen Abschnitt finden Sie unter <https://www.youtube.com/watch?v=X60-JRX2pTI>.

Wenn Sie den Code, der in Abbildung 121 zu sehen ist, im Einzelschrittmodus debuggen, werden Sie sehen, dass beim Durchlaufen der Schleife zunächst genau einmal GetEnumerator und dann im Wechsel MoveNext und die **get** Methode der Eigenschaft Current aufgerufen wird, bis die Collection komplett durchlaufen worden ist. Zum Abschluss wird einmal Dispose aufgerufen, um mögliche Ressourcen nach der Nutzung des Enumerators wieder freizugeben. Tatsächlich kann man die **foreach** Schleife auch in einer langen Form schreiben, da sie letztendlich nichts anderes macht als der Code in der folgenden Abbildung, der genau dasselbe macht wie der Code aus Abbildung 121:

```
static void Main()
{
    var stack = new Stack<string>(5);
    stack.FügeElementHinzu("Hallo");
    stack.FügeElementHinzu("Welt");

    IEnumator<string> enumerator = stack.GetEnumerator();
    string element; —— foreach Laufvariable
    try
    {
        while (enumerator.MoveNext())
        {
            element = enumerator.Current;
            Console.WriteLine(element); —— Code des foreach-Schleifenblocks
        }
    }
    finally
    {
        enumerator.Dispose();
    }
}
```

Abbildung 122: Die interne Funktionsweise von foreach

Wie in Abbildung 122 zu sehen ist, wird zunächst der Enumerator für die Aufzählung erstellt, wenn eine **foreach** Schleife begonnen wird. Innerhalb eines **try-finally** Blocks wird dann solange MoveNext auf dem Enumerator aufgerufen, bis dieser am Ende der Aufzählung angekommen ist. Innerhalb der dazu verwendeten **while** Schleife wird der **foreach** Laufvariablen, die im obigen Beispiel **element** heißt, jeweils das aktuelle Element des Enumerators zugewiesen. Der **try-finally** Block wird übrigens genutzt, damit auf dem Enumerator auf jeden Fall Dispose aufgerufen wird, egal ob im **try** Block eine Exception ausgelöst wird oder nicht. Da kein **catch** Block mitangegeben ist, wird eine eventuelle Exception entlang des Call Stacks weitergeleitet.

5.15.4.4 Endlosschleifen mit foreach (Level 200)

Das Video für diesen Abschnitt finden Sie hier: <https://www.youtube.com/watch?v=fNsgO1MWeeY>.

Durch die im vorherigen Abschnitt aufgezeigte interne Funktionalität der **foreach** Schleife ist es aber auch möglich, eine Endlosschleife mit **foreach** zu produzieren, nämlich genau dann, wenn **IEnumerator<T>.MoveNext** immer **true** zurückgibt und jeweils einen neuen Wert bereitstellt (anders ausgedrückt: die Aufzählung ist unendlich lang). Dies kann man relativ leicht nachbauen, indem man bspw. folgenden Code schreibt:

```

public class EndloseZufallszahlenAufzählung : IEnumerable<int>
{
    public IEnumerator<int> GetEnumerator()
    {
        return new EndloserEnumerator(new Random());
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

private class EndloserEnumerator : IEnumerator<int>
{
    private readonly Random _zufallszahlenGenerator;

    public EndloserEnumerator(Random zufallszahlenGenerator)
    {
        if (zufallszahlenGenerator == null)
            throw new ArgumentNullException("zufallszahlenGenerator");

        _zufallszahlenGenerator = zufallszahlenGenerator;
    }

    public void Dispose() { }

    public bool MoveNext()
    {
        Current = _zufallszahlenGenerator.Next();
        return true;
    }

    public void Reset()
    {
        throw new NotSupportedException();
    }

    public int Current { get; private set; }

    object IEnumerator.Current { get { return Current; } }
}
}

```

Klassen können ineinander geschachtelt werden, richtig sinnvoll ist das in den meisten Fällen nicht

MoveNext erstellt immer eine neue Zufallszahl und gibt true zurück

Abbildung 123: Eine unendlich große Aufzählung

Wenn die in Abbildung 123 zu sehende Klasse `EndloseZufallszahlenAufzählung` instanziert und in einer `foreach` Schleife eingesetzt wird, kommt es zu einer Endlosschleife, da `MoveNext` nie `false` zurückgibt, was das Ende der `foreach` Schleife auslösen würde. Wenn man wirklich sicher gehen möchte, dass eine Aufzählung endlich ist, muss man die Interfaces `ICollection<T>` oder `IList<T>` als Referenztypen verwenden, da die dort definierte Eigenschaft `Count` vorgibt, dass eine Collection eine gewisse Anzahl von Elementen hat und damit endlich ist.

Neben diesem interessanten Fakt findet man noch zwei weitere Nebenpunkte für das Allgemeinwissen im letzten Beispiel:

- Klassen können ineinander geschachtelt werden. In diesem Fall wurde das gemacht, damit die innere Klasse `EndloserEnumerator` mit dem Modifizierer `private` ausgestattet werden konnte und somit nur innerhalb der Klasse `EndloseZufallszahlenAufzählung`

instanziert und referenziert werden kann. Jeder weitere Zugriff außerhalb ist nur über das Interface `IEnumerator<int>` möglich. Nutzen Sie diese Fähigkeit nicht in Ihrem Produktivcode – ich wollte Ihnen nur zeigen, dass es prinzipiell möglich ist.

- Wenn man bestimmte Member einer Abstraktion nicht unterstützen möchte, wie das bei der Methode `Reset` im Enumerator der Fall ist, wirft man in diesen konventionsgemäß eine `NotSupportedException`. Grundsätzlich sollten Sie aber versuchen, es zu vermeiden, bestimmte Mitglieder nicht zu unterstützen.

5.15.4.5 Nur lesend auf Collections via Interfaces zugreifen

Das Video zu diesem Abschnitt finden Sie hier: <https://www.youtube.com/watch?v=rP0RL0pfWtI>.

Wenn Sie in Ihren Klassen nur lesend auf Collections via Interfaces zugreifen möchten, können Sie alternativ zu `ICollection<T>` und `IList<T>` die Interfaces `IReadOnlyCollection<T>` oder `IReadOnlyList<T>` einsetzen. Im Gegensatz zu den ersten beiden haben die letzten beiden nur die `Count` Eigenschaft bzw. den Indexzugriff definiert. Alle Methoden zum Manipulieren von Collections wie `Add`, `Clear`, `Insert`, `Remove` oder `RemoveAt` fehlen.



Abbildung 124: Die Interfaces `IReadOnlyCollection<T>` und `IReadOnlyList<T>` in der Vererbungshierarchie

Prinzipiell kann ich Ihnen nur raten, diese Interfaces überall wo möglich einzusetzen. Einerseits schützen Sie sich davor, ein Collection aus Versehen an Codestellen zu manipulieren, an denen das nicht vorgesehen ist, andererseits wird jedem beim Lesen ihrer Klasse klar, dass eine referenzierte Collection hier nur Objekte bereitstellt und nicht verändert wird. `IReadOnlyCollection<T>` und `IReadOnlyList<T>` sind leider erst seit .NET 4.5 im Framework enthalten – wenn Sie also gegen eine frühere Version programmieren, stehen Ihnen diese nicht zur Verfügung.

5.15.5 Wann sollte man Collections einsetzen?

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=cEZSu1Oh90>.

Collections sollten immer dann eingesetzt werden, wenn mehrere Objekte eines Typs verarbeitet werden müssen, und das passiert im Programmieralltag sehr häufig. Prinzipiell kann man nahezu blind eine Collection nutzen, wenn man in einer Anforderung eines der Worte viele, mehrere, einzelne oder ähnliche Pluralwörter hört. Weiterhin sollten Sie aufhören, Arrays einzusetzen, da diese im Vergleich zur Liste deutlich unflexibler sind und eine umständlichere API besitzen. Arrays haben im Vergleich zu Listen nur einen leichten Geschwindigkeitsvorteil, den Sie aber in nahezu allen Fällen ignorieren können.

5.16 Fortgeschrittene Möglichkeiten mit Generics und Reflection

Damit wir die weiteren Möglichkeiten mit Generics kennenlernen, nehmen wir uns ein weiteres Beispiel vor und bauen dies schrittweise aus. Bei diesem Beispiel geht es um die Gleichheit von Objekten der Klasse **Farbe**, deren Source Code wir im folgenden Abschnitt sehen. Neben den weiteren Möglichkeiten, die Generics bieten, lernen wir in diesem Abschnitt Reflection kennen. Mit dieser .NET API kann man die Mitglieder eines Typs dynamisch zur Laufzeit auslesen und aufrufen.

5.16.1 Immutable Objects und Value Objects – Die Klasse Farbe

Das Video zu diesem Abschnitt finden Sie unter https://www.youtube.com/watch?v=i4oY_zR3wUk.

In der folgenden Abbildung sehen Sie die Klasse **Farbe**, die ein sog. Immutable Object (dt. unveränderliches Objekt) ist. Unveränderliche Objekte werden durch Klassen repräsentiert, deren Felder nur beim Initialisieren gesetzt werden können. Nach dem Konstruktorauftrag ist nur noch lesender Zugriff auf diese Felder über **get** Methoden gestattet. Wenn man eine Farbe ändern möchte, kann dazu nicht der Rot-, Grün- oder Blauwert eines Farbobjekts angepasst werden, sondern man muss ein komplett neues Farbobjekt erstellen.

```
public class Farbe
{
    private readonly byte _rot;
    private readonly byte _grün;
    private readonly byte _blau;

    public Farbe(byte rot, byte grün, byte blau)
    {
        _rot = rot;
        _grün = grün;
        _blau = blau;
    }

    public byte Rot { get { return _rot; } }
    public byte Grün { get { return _grün; } }
    public byte Blau { get { return _blau; } }
}
```

Abbildung 125: Die Klasse Farbe ist ein Immutable Object

Instanzen der Klasse Farbe sind ebenfalls sog. Value Objects (dt. Wertobjekte), da sie ihre Identität ausschließlich über die drei Werte für Rot, Grün und Blau definieren: zwei Farben gelten dann als gleich, wenn Sie über genau dieselben RGB-Werte verfügen. Wir möchten im nächsten Beispiel Farbobjekte in Verbindung mit einem **HashSet**<T> einsetzen. Diese Collection lässt keine Duplikate zu, wie wir in Abschnitt 5.15.3 gesehen haben. Sehen Sie sich jedoch an, welches Problem bei folgendem Codebeispiel auftritt:

```

    static void Main()
    {
        var intHashSet = new HashSet<int>();
        intHashSet.Add(3);
        intHashSet.Add(5);
        intHashSet.Add(3);

        GibAufzählungAus(intHashSet); — intHashSet hat bei diesem
                                         Aufruf zwei Elemente

        var farbenHashSet = new HashSet<Farbe>();
        farbenHashSet.Add(new Farbe(128, 0, 0));
        farbenHashSet.Add(new Farbe(0, 128, 0));
        farbenHashSet.Add(new Farbe(128, 0, 0));

        GibAufzählungAus(farbenHashSet); — farbenHashSet hat bei diesem
                                         Aufruf drei Elemente, obwohl
                                         nur zwei erwartet sind
    }

    static void GibAufzählungAus<T>(IEnumerable<T> aufzählung)
    {
        foreach (var element in aufzählung)
        {
            Console.WriteLine(element);
        }
    }

```

Abbildung 126: Gleichheitsproblem bei mehreren Instanzen, die dieselbe Farbe repräsentieren

In Abbildung 126 sehen sie zwei Hash Sets, eines für `int` Werte und eines für Objekte der Klasse `Farbe`. Zum `intHashSet` werden die Werte drei, fünf und drei hinzugefügt, wobei natürlich das Hash Set die letzte drei nicht einfügt, da keine Duplikate möglich sind. Deshalb enthält `intHashSet` beim ersten Aufruf von `GibAufzählungAus` auch nur zwei Elemente. (In diesem Beispiel sieht man als Nebensächlichkeit auch, dass man Generics nicht nur auf Klassen und Interfaces, sondern auch direkt auf Methoden anwenden kann).

Genau nach demselben Prinzip möchten wir auch das zweite Hash Set `farbenHashSet` aufbauen. Dazu fügen wir einmal rot, einmal grün und nochmals rot zum Hash Set hinzu, allerdings hat `farbenHashSet` beim zweiten Aufruf von `GibAufzählungAus` drei statt zwei Elemente: das zweite Rot-Objekt `new Farbe(128, 0, 0)` wurde offensichtlich nicht als Duplikat erkannt.

Dies ist dadurch bedingt, dass die Klasse ihre Gleichheitsfunktionalität von der allgemeinen Basisklasse `object` ableitet, genauer gesagt von den Methoden `object.Equals` und `object.GetHashCode`. Wie wir bereits in Abschnitt 5.12.8 angemerkt haben, überprüft diese Standardimplementierung die Gleichheit anhand der Speicheradresse des Objekts im Heap – anders ausgedrückt: zwei Objektreferenzen gelten als gleich, wenn sie auf dasselbe Objekt zeigen.

Diese beiden Methoden sind jedoch virtuell, sodass wir sie in abgeleiteten Klassen überschreiben können. Genau das möchten wir jetzt für die Klasse `Farbe` tun, damit zwei Farbobjekte, welche dieselbe Farbe repräsentieren, als gleich gelten. Dabei müssen `Equals` und `GetHashCode` derselben Logik folgen:

- `Equals` gibt einen `bool` zurück, der aussagt, ob der übergeben Parameter mit diesem Objekt übereinstimmt oder nicht.

- GetHashCode gibt einen sog. Hash Code (der nur ein `int` Wert ist) zurück für einen schnellen Vergleich zwischen zwei Instanzen. Dabei gilt: zwei Objekte mit unterschiedlichen Hash Codes sind niemals als gleich anzusehen, zwei Objekte mit identischen Hash Codes können gleich sein.

Wichtig: wenn Sie die `Equals` Methode einer Klasse überschreiben, müssen Sie auch jeweils die `GetHashCode` Methode überschreiben, damit es in Collections wie `HashSet<T>` nicht zu inkonsistenten Verhalten kommt. Genau das wurde für die Klasse Farbe in folgendem Beispiel gemacht:

```
public class Farbe
{
    private readonly byte _rot;
    private readonly byte _grün;
    private readonly byte _blau;

    public Farbe(byte rot, byte grün, byte blau)
    {
        _rot = rot;
        _grün = grün;
        _blau = blau;
    }

    public override bool Equals(object obj)
    {
        if (obj == this) // Über die Kombination der Werte
            return true; // Rot, Grün und Blau (RGB) wollen
                           // wir die Gleichheit für diese
                           // Klasse definieren

        var anderesFarbobjekt = obj as Farbe; // Zunächst wird überprüft, ob es
        if (anderesFarbobjekt == null) // sich bei der anderen Referenz
            return false; // um dasselbe Objekt handelt

        return anderesFarbobjekt._rot == _rot && // Dann wird überprüft, ob es sich
            anderesFarbobjekt._grün == _grün && // bei der anderen Referenz um ein
            anderesFarbobjekt._blau == _blau; // Farbobjekt handelt

    }

    public override int GetHashCode()
    {
        return _rot.GetHashCode() ^
               _grün.GetHashCode() ^
               _blau.GetHashCode(); // Zum Schluss werden die
                           // tatsächlichen RGB Werte
                           // verglichen
    }

    public byte Rot { get { return _rot; } }

    public byte Grün { get { return _grün; } }

    public byte Blau { get { return _blau; } }
}
```

Abbildung 127: `Equals` und `GetHashCode` für Value Objects überschreiben

In Abbildung 127 sehen Sie, wie die Methode `Equals` aufgebaut wurde. Zunächst wird direkt überprüft, ob die Referenzen `this` und `obj` identisch sind – also auf dasselbe Objekt im Speicher zeigen. Wenn das der Fall ist, können wir natürlich sofort `true` zurückgeben. Trifft dies jedoch nicht

zu, so muss `obj` zunächst in ein Farbobjekt gecastet werden, um an die Rot-Grün-Blau (RGB) Werte von `obj` zu kommen. Ist das übergeben Objekt jedoch nicht vom Typ `Farbe`, gibt der `as` Operator null zurück und wir können in diesem Fall `false` an den Aufrufer zurückgeben. Der letzte Schritt in `Equals` ist der tatsächlich interessante: wenn es sich bei `obj` um ein Objekt von `Farbe` handelt, können wir jetzt die RGB-Feldwerte miteinander vergleichen. Stimmen diese alle überein, so geben wir `true` zurück, ansonsten `false`.

Die `GetHashCode` Funktion bildet gemäß dem letzten Schritt von `Equals` den Hash Code aus genau den drei RGB Werten. Wenn Hash Codes aus Feldwerten gebildet werden wie in diesem Beispiel, ist es Konvention, dass man einfach auf den Feldern jeweils `GetHashCode` aufruft und die resultierenden Werte mit XOR verknüpft. Genau das wird oben gemacht. Eine bessere Implementierung wäre hier jedoch möglich, da es sich um `byte` Werte handelt, die man bspw. auf verschiedene Positionen des `int` Rückgabewerts legen könnte (bspw. mit dem Shift-Operatoren `>>` und `<<`). Dadurch würde das zufällige Zusammentreffen von identischen Hash Codes bei unterschiedlichen RGB Werten vermieden werden. Ich habe hier jedoch diese Form gewählt, da sie die übliche Implementierungsweise von `GetHashCode` darstellt.

Wenn wir diese modifizierte Klasse jetzt mit dem Code aus Abbildung 126 einsetzen, dann erhalten wir das erwartete Verhalten, da nun zwei Farbe-Instanzen, die mit denselben Werten ausgestattet sind, auch als gleich gelten. Das ist in diesem Fall genau das Verhalten, dass man intuitiv von Value Objects erwartet.

5.16.2 Das Interface `IEquatable<T>`

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=vIBXN1yGNq8>.

Wir haben im vorherigen Abschnitt zwar das Problem der Gleichheit bei Farbe gelöst, ein Punkt ist allerdings noch nicht optimal: die Methode `object.Equals` hat als Parametertypen `object`. Das sorgt dafür, dass bei jeder Überschreibung erst der Parameter zum eigentlichen Zieltypen gecastet werden muss, so wie das in Schritt 2 der `Equals` Methode in Abbildung 127 der Fall ist. Dem kann man durch Implementieren des Interfaces `IEquatable<T>` entgegenwirken (`equatable` heißt auf Deutsch vergleichbar).

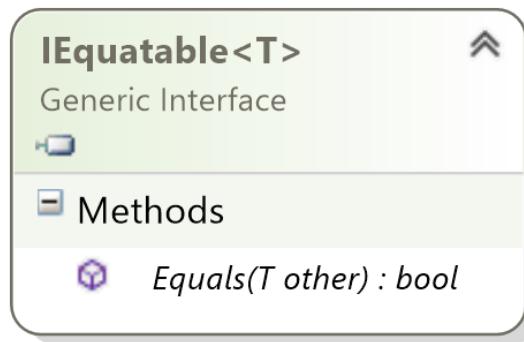


Abbildung 128: Das Interface `IEquatable<T>`

`IEquatable<T>` besitzt genauso wie die Basisklasse `object` eine Methode `Equals`, die allerdings generisch ist, d.h. der Typparameter nutzt den Generic T. Dadurch entfällt der vorher angesprochene Cast bei des Parameterwertes. Einige .NET Collections wie `List<T>`, `Dictionary<TKey, TValue>` oder `LinkedList<T>` unterstützen `IEquatable<T>`, wenn auf Gleichheit überprüft wird bei Methoden wie `Contains`, `IndexOf`, oder `Remove`. Das heißt, dass beim Vergleich zweier Objekte `Equatable<T>.Equals` aufgerufen wird anstatt von `object.Equals`, um eine bessere Performance zu erhalten (einerseits durch den verhinderten Cast,

andererseits durch die fehlende dynamische Bindung beim Aufruf). Dies ist im Einzelfall natürlich kaum relevant, wenn man aber bspw. Collections mit mehreren tausend Objekten durchsucht, dann kann dies einen deutlichen Performanceeinfluss haben. Dieses Interface möchten wir nun in der Klasse Farbe implementieren.

```
public class Farbe : IEquatable<Farbe>
{
    private readonly byte _rot;
    private readonly byte _grün;
    private readonly byte _blau;

    public Farbe(byte rot, byte grün, byte blau)
    {
        _rot = rot;
        _grün = grün;
        _blau = blau;
    }

    public bool Equals(Farbe other) ————— Die Equals Methode erhält jetzt
    {                                         ein Objekt von Typ Farbe, wie es
        if (other == this) return true;      das Interface vorgibt
        if (other == null) return false;

        return other._rot == _rot &&
               other._grün == _grün &&
               other._blau == _blau;
    }

    public override bool Equals(object obj)
    {
        return Equals(obj as Farbe); ————— Die object.Equals Methode leitet ihren
    }                                         Aufruf weiter

    public override int GetHashCode()
    {
        return _rot.GetHashCode() ^ _grün.GetHashCode() ^ _blau.GetHashCode();
    }

    public byte Rot { get { return _rot; } }

    public byte Grün { get { return _grün; } }

    public byte Blau { get { return _blau; } }
}
```

Abbildung 129: `IEquatable<T>` implementieren

Beim Beispiel in Abbildung 129 sehen wir gleich mehrere interessante Sachen:

- Beim Ableiten von generischen Typen kann man den Generic auflösen. In den vorherigen Beispielen haben wir dies meist nicht getan und stattdessen auf der Klasse einen eigenen Generic definiert.
- Die `Equals` Methode, die durch das Interface vorgegeben ist, hat als Parametertyp `Farbe` statt `object`. Damit entfällt der vorher noch notwendige Cast.

- Die überschriebene Methode `object.Equals` leitet seinen Aufruf einfach an die eigentliche `Equals` Methode weiter. Dabei muss der Parameter mit dem `as` Operator gecastet werden.

Genau nach diesem Muster sollten Sie Klassen, welche die Gleichheitsfunktionalität von `object` ändern, aufbauen:

1. Implementieren Sie `IEquatable<T>`
2. Überschreiben Sie `object.Equals` und leiten Sie den Aufruf an die `IEquatable<T>.Equals` Implementierung weiter. Dabei muss der Parameterwert zum Zieltyp gecastet werden.
3. Der Hash Code muss nach den gleichen Maßstäben aufgebaut werden wie die für den Vergleich in der `Equals` Methode geltenden Maßstäbe (bei Value Objects eben bspw. die Feldwerte).
4. Überladen Sie die Operatoren `==` und `!=` wie in 5.13.4 gezeigt. Die Überladungen sollten aber intern die `Equals` Methode aufrufen.

5.16.3 Verschiedene Value Objects und damit verbundene DRY-Probleme

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=nkoQ37Csmtl>.

Wenn Sie verschiedene Klassen nach dem im vorherigen Abschnitt gezeigten Muster erstellen, die Value Objects repräsentieren, wird Ihnen sehr schnell auffallen, dass die Implementierung dieser sehr ähnlich ist. Sehen Sie sich bspw. die bereits bekannt Klasse `Adresse` an, die ebenfalls ein Value Object darstellt und in Abbildung 130 zu sehen ist. Alle Methode, die mit der Überprüfung der Gleichheit zu tun haben, also die beiden `Equals` Methoden, `GetHashCode` und die Operatorenüberladungen `==` und `!=` sind funktional identisch mit denen der Klasse `Farbe`, nutzen allerdings die jeweiligen Felder der Klasse. Deswegen kann man diesen Code nicht mit den uns bekannten Mitteln in eine andere Klasse auslagern (entweder über Komposition oder Vererbung), da wir eben die Felder der jeweiligen Klasse kennen müssen, um zu bestimmen, ob zwei Objekte gleich sind. Dazu muss man den jeweiligen Typ kennen und das ist insofern ärgerlich, da wir eigentlich die Schritte zur Erstellung für Value Objects wie folgt verallgemeinern können:

- Zu Beginn werden zunächst die Felder für die Klasse festgelegt und auf Sie lesender Zugriff über Eigenschaften gestattet. Die Werte für diese Felder werden ausschließlich im Konstruktor gesetzt über Parameter.
- Im Anschluss implementiert man das Interface `IEquatable<T>`. In dessen `Equals` Methode wird das zu vergleichende Objekt auf `null` und auf `this` überprüft. Wenn diese Überprüfungen zu keinem Ergebnis führen, vergleicht man sämtliche Feldwerte. Sind alle diese gleich, gilt das andere Objekt als identisch.
- Im Anschluss überschreibt man `object.Equals` und leitet den Aufruf an die eben implementierte `Equals` Methode weiter. Der Parameter wird dabei zum jeweiligen Klassentyp gecastet.
- Als nächstes überschreibt man `object.GetHashCode`. Der Hash Code setzt sich aus allen Feldwerten zusammen, deren Hash Codes mit XOR zum Rückgabewert verknüpft werden.
- Zuletzt überlädt man die Operatoren `==` und `!=`, damit auch diese vom Nutzer für den Vergleich zweier Value Objects eingesetzt werden können. Die `==` Überladung ruft dabei die `Equals` Methode auf, nachdem die beiden Parameterwerte auf `null` überprüft wurden. Die `!=` Überladung ruft im Gegensatz dazu einfach die `==` Überladung auf und negiert deren Rückgabewert.

Diese Schritte müssen beständig wiederholt werden für jedes neue Value Object, das implementiert werden soll, was eindeutig gegen das DRY-Prinzip verstößt. Um dies zu umgehen, brauchen wir eine Möglichkeit, uns alle Felder einer Klasse zu besorgen und diese dann dynamisch auszulesen.

```

public class Adresse : IEquatable<Adresse>
{
    public Adresse(string straße, string postleitzahl, string ort)
    {
        if (straße == null) throw new ArgumentNullException("straße");
        if (postleitzahl == null) throw new ArgumentNullException("postleitzahl");
        if (ort == null) throw new ArgumentNullException("ort");

        Straße = straße;
        Postleitzahl = postleitzahl;
        Ort = ort;
    }

    public string Straße { get; private set; }
    public string Postleitzahl { get; private set; }
    public string Ort { get; private set; }

    public bool Equals(Adresse other)
    {
        if (other == null) return false;
        if (other == this) return true;

        return other.Straße == Straße &&
               other.Postleitzahl == Postleitzahl &&
               other.Ort == Ort;
    }

    public static bool operator ==(Adresse adresse1, Adresse adresse2)
    {
        if (adresse1 == null) return adresse2 == null;

        return adresse1.Equals(adresse2);
    }

    public static bool operator !=(Adresse adresse1, Adresse adresse2)
    {
        return !(adresse1 == adresse2);
    }

    public override bool Equals(object obj)
    {
        return Equals(obj as Adresse);
    }

    public override int GetHashCode()
    {
        return Straße.GetHashCode() ^ Postleitzahl.GetHashCode() ^ Ort.GetHashCode();
    }
}

```

Diese Methoden sind funktional
identisch mit den Methoden der
Klasse Farbe, hängen aber von
den jeweiligen Feldwerten der
Klasse ab

Abbildung 130: DRY Problem bei Value Objects

5.16.4 Reflection

5.16.4.1 Grundsätzliches zu Reflection

Das Video zu diesem Abschnitt finden Sie unter https://www.youtube.com/watch?v=t2_OrLF8uo.

Das im vorigen Abschnitt angesprochene Problem des Zugriffs auf alle Felder eines Objekts, obwohl wir den entsprechenden Typ nicht kennen, kann beseitigt werden, indem wir Reflection einsetzen. Reflection ist der Mechanismus des Auslesens von Typinformationen zur Laufzeit. Eine Typinformation beinhaltet dabei alle Informationen, die auch über den regulären Zugriff auf Objekte oder Klassen erhältlich sind, also alle Klassenmitglieder. Zusätzlich dazu kann man auch noch alle Attribute, die wir uns im Abschnitt 5.13.3 angesehen haben, zu einem Typ auslesen. All diese Informationen können wiederum genutzt werden, um bestimmte Klassenmitglieder aufzulisten, zu filtern und (dynamisch) aufzurufen. Damit das Ganze besser verständlich wird, sehen wir uns im Folgenden die Klasse **KonsolenTypdarsteller** an, der alle Methoden, die **public** sind, zu einem beliebigen Objekt oder zu einem über einen Generic festgelegten Typ via Reflection ausgibt:

```

public class KonsolenTypdarsteller
{
    public void GibTypinfosAus(object @object)
    {
        if (@object == null) throw new ArgumentNullException("object");
        GibTypinfosAus(@object.GetType());
```



```

    }

    public void GibTypinfosAus<T>()
    {
        GibTypinfosAus(typeof(T));
```



```

    }

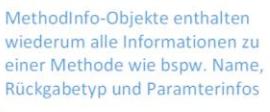
    private void GibTypinfosAus(Type typ)
    {
        GibTypArtAus(typ);
        GibMethodenAus(typ);
```



```

    }

    private void GibTypArtAus(Type typ)
    {
        var typArt = typ.IsClass ? "Klasse" : typ.IsInterface ? "Interface" : "Struktur";
        Console.WriteLine("{0} ({1})", typ.Name, typArt);
        Console.WriteLine("Namespace: {0}", typ.Namespace);
        Console.WriteLine("-----");
        Console.WriteLine();
    }
```



```

    }

    private void GibMethodenAus(Type typ)
    {
        MethodInfo[] methodenInfos = typ.GetMethods();
        if (methodenInfos.Length == 0)
        {
            Console.WriteLine("Dieser Typ enthält keine Methoden");
            return;
        }
        foreach (MethodInfo methodenInfo in methodenInfos)
        {
            Console.WriteLine(methodenInfo.Name);
            string istStatischText = methodenInfo.IsStatic ? " (statisch)" : "";
            Console.WriteLine("Rückgabetyp: " + methodenInfo.ReturnType.Name + istStatischText);
            ParameterInfo[] parameterInfos = methodenInfo.GetParameters();
            if (parameterInfos.Length == 0)
            {
                Console.WriteLine("Diese Methode besitzt keine Parameter");
            }
            else
            {
                Console.Write("Parameter: ");
                foreach (ParameterInfo parameterInfo in parameterInfos)
                {
                    Console.Write("{0} ({1}), ", parameterInfo.Name, parameterInfo.ParameterType.Name);
                }
                Console.WriteLine();
            }
            Console.WriteLine();
        }
        Console.WriteLine();
    }
}
```

Abbildung 131: Über Reflection Typinformationen dynamisch auslesen

In Abbildung 131 sehen Sie eben genannte Klasse, welche die Methode `GibTypinfosAus` in zwei Überladungen besitzt:

- Bei der ersten Überladung kann man ein beliebiges Objekt übergeben, über den ein `System.Type`-Objekt mit der Methode `GetType` ausgelesen wird.
- Bei der zweiten Überladung löst man beim Aufruf den in der Methode angegeben Generic auf, über den dann mit dem `typeof` Operator das `Type`-Objekt besorgt wird.

Dies sind die beiden Arten, um sie ein Type-Objekt zu besorgen, in dem die oben angesprochenen Typinformationen (auch Typmetadaten genannt) festgehalten sind. Egal auf welchem Wege: letztendlich wird die `private` Methode `GibTypinfosAus` aufgerufen, in der besagtes `Type`-Objekt verwendet wird, um direkt Infos zur Typdefinition auszugeben (siehe Methode `GibTypartAus`) bzw. um alle `MethodInfo`-Objekte zum entsprechenden Typ zu besorgen, die wiederum Informationen zu den Methoden des Typs halten, also bspw. Bezeichner der Methode, Rückgabetyppinfos, Parameterinfos usw.

Die Klasse `KonsolenTypdarsteller` kann bspw. wie folgt eingesetzt werden:

```
static void Main()
{
    var typdarsteller = new KonsolenTypdarsteller();
    var objekt = new object();
    typdarsteller.GibTypinfosAus(objekt); ————— Aufruf über Objekt
    typdarsteller.GibTypinfosAus<int>(); ————— Aufruf über Auflösen eines Generic
    typdarsteller.GibTypinfosAus<IList<string>>();
```

Abbildung 132: Einsatz der Klasse `KonsolenTypdarsteller`

Natürlich kann man sich über ein `Type`-Objekt nicht nur Infos zu Methoden, sondern zu allen Mitgliedern eines Typs wie Feldern, Eigenschaften oder Events holen. Die entsprechenden Funktionen dazu sind `Type.GetFields`, `Type.GetProperties` und `Type.GetEvents`, die jeweils `FieldInfo`-, `PropertyInfo`- und `EventInfo`-Objekte in einem Array zurückgeben. Die API dieser Klassen ist spezifisch auf die jeweilige Mitgliederart zugeschnitten, d.h. bspw. das man über eine `PropertyInfo` die jeweiligen `get` und `set` Methoden einer Eigenschaft auslesen kann. In der Tat ist die Reflection API sehr umfangreich, weswegen wir hier nicht auf alle Details eingehen können. Detaillierte Informationen zu den einzelnen Klassen entnehmen Sie bitte der [MSDN Library](#). Für sie wichtig ist folgender Punkt:

Die Klasse `System.Type` ist der Einstiegspunkt für den Reflection-Mechanismus. Mit einem Objekt dieser Klasse können alle Metadaten zu einem bestimmten Typ ausgelesen werden. Diese Metadaten sind so geschachtelt, wie man es beim Schreiben von Typen gewohnt ist. Bspw. erhält man über das `Type`-Objekt alle Methodeninfos einer Klasse, eine einzelne Methodeninfo wiederum gibt Infos zu Bezeichner, Rückgabetyp und Parameter, usw.

5.16.4.2 Die Enumeration `BindingFlags`

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=Mvgi9oCvG3E>.

Beim Auslesen von Mitgliederinformationen über Reflection kann man bei den Methoden `GetMethods`, `GetFields` usw. auch jeweils eine Überladung aufrufen, die einen Wert vom Typ `System.Reflection.BindingFlags` entgegennimmt. `BindingFlags` ist weder eine Klasse noch ein Interface (die beiden Typen, mit denen wir uns bis jetzt ausgiebig auseinandergesetzt haben), sondern eine sog. Enumeration (Vorsicht: Enumerationen haben rein gar nichts mit Collections und

`IEnumerable<T>` zu tun). Enumerationen kann man sich am ehesten als eine zur Compilezeit festgelegte Ansammlung von konstanten Werten, die thematisch zusammengehören, vorstellen. Genaueres zu Enumerationen lernen wir in einem späteren Kapitel kennen. In der folgenden Abbildung sehen Sie, wie Binding Flags definiert ist:

```
// Summary:
//     Specifies flags that control binding and the way in which the search for
//     members and types is conducted by reflection.
[Serializable]
[ComVisible(true)]
[Flags]
public enum BindingFlags
{
    ...Default = 0,
    ...IgnoreCase = 1,
    ...DeclaredOnly = 2,
    ...Instance = 4,
    ...Static = 8,
    ...Public = 16,
    ...NonPublic = 32,
    ...FlattenHierarchy = 64,
    ...InvokeMethod = 256,
    ...CreateInstance = 512,
    ...GetField = 1024,
    ...SetField = 2048,
    ...GetProperty = 4096,
    ... SetProperty = 8192,
    ... PutDispProperty = 16384,
    ... PutRefDispProperty = 32768,
    ... ExactBinding = 65536,
    ... SuppressChangeType = 131072,
    ... OptionalParamBinding = 262144,
    ... IgnoreReturn = 16777216,
}
```

Abbildung 133: Definition der Enumeration `BindingFlags`

Mit den einzelnen Werten von `BindingFlags` kann man spezifizieren, welche Mitglieder genau von einem Typ zurückgegeben werden sollen. Standardmäßig bekommt man bspw. alle Methoden, die als `public` deklariert wurden. Da auf `BindingFlags` ebenfalls das Attribut `[Flags]` definiert ist, kann man die einzelnen Werte von `BindingFlags` mit dem Bitweise Operator `|` zusammenfügen. Um bspw. ausschließlich Methodeninformationen von privaten Instanzmethoden zu bekommen, kann man den folgenden Code einsetzen:

```
public MethodInfo[] GibPrivateInstanzMethodenZurück(object objekt)
{
    var typ = objekt.GetType();
    const BindingFlags privatUndInstanz = BindingFlags.NonPublic | BindingFlags.Instance;
    return typ.GetMethods(privatUndInstanz);
}

Diese Anweisung gibt jetzt
ausschließlich Infos zu
privaten, nicht-staticischen
Methoden zurück
```

Die Konstanten Werte können
mit `|` verknüpft werden, wenn
auf der Enumeration das
`FlagsAttribute` gesetzt ist

Abbildung 134: Mit `BindingFlags` die über Reflection beschafften Infos filtern

In Abbildung 134 wird zunächst das Type-Objekt geholt und im Anschluss die Enumerationswerte `BindingFlags`.`NonPublic` und `BindingFlags`.`Instance` mit dem `|` Operator kombiniert,

sodass beim Aufruf im letzten Statement alle `MethodInfo`-Objekte zurückgegeben werden, die private und nicht-statisch sind. Mit diesem Mechanismus kann man die zurückgegebenen Methodeninformationen filtern.

5.16.4.3 Die Vererbungshierarchie zu den wichtigsten Reflection-Klassen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=lePeDfXDik8>.

Die Klassen `Type`, `FieldInfo`, `MethodInfo`, `PropertyInfo` und `EventInfo` stehen in einer Vererbungshierarchie zueinander, wie man in folgender Abbildung sehen kann:

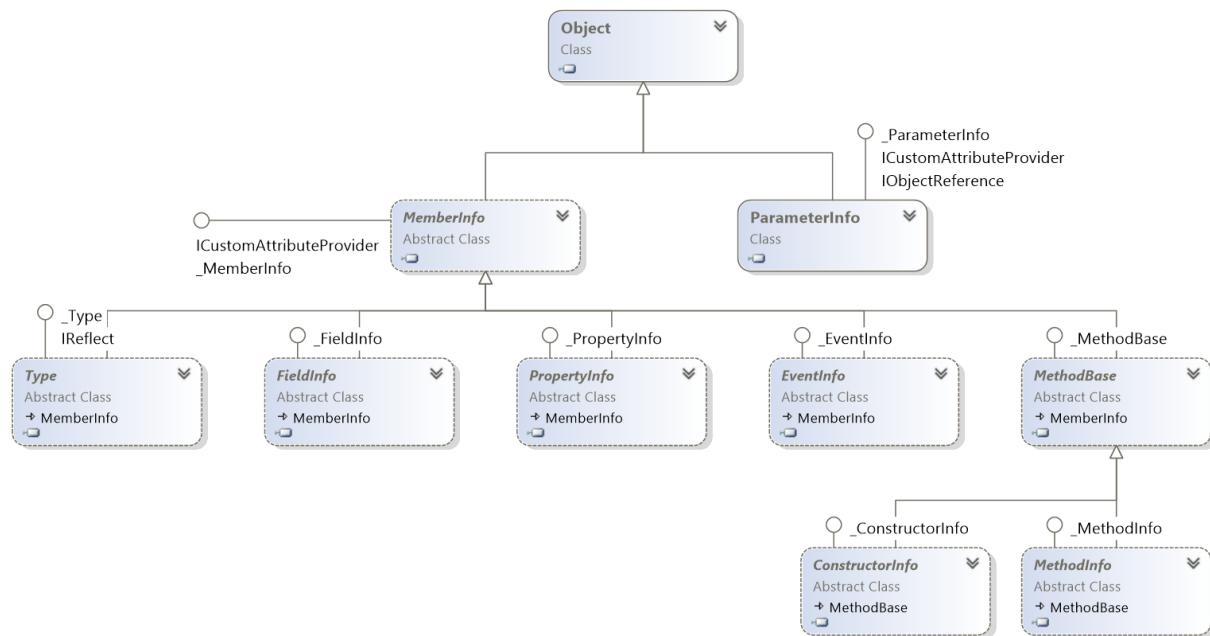


Abbildung 135: Die Klassenhierarchie der Reflection-API

Wie man in Abbildung 135 sehen kann, leiten alle eben genannten Klassen von der Basisklasse `MemberInfo` ab, einzig `ParameterInfo` nutzt als Basisklasse `object`. Weiterhin interessant ist, dass zwischen einer Metadateninformation für Methoden und Konstruktoren durch die Klassen `MethodInfo` und `ConstructorInfo` unterschieden wird. Dies zeigt nochmals die besondere Stellung von Konstruktoren. Man sollte sie nicht mit regulären Methoden gleichsetzen.

Ein wichtiger Punkt ist ebenfalls, dass diese Klassen alle abstrakt sind. Wenn wir gegen diese Klassen programmieren, wie das bei der Klasse `KonsolenTypdarsteller` der Fall ist, dann sind die Objekte hinter diesen Referenzen eigentlich Instanzen abgeleitete Klassen, die ebenfalls im .NET Framework implementiert sind, allerdings mit dem Modifizierer `internal` gekennzeichnet wurden, sodass wir auf diese außerhalb der Assembly System nicht zugreifen können.

Über all diese gezeigten Klassen kann man nicht nur lesend auf die Metadaten zu einem Typen zugreifen, sondern man kann auch dynamisch Feldwerte setzen und auslesen sowie Methoden, Eigenschaften und Events aufrufen. Und genau dies hilft uns bei der Beseitigung des DRY-Problems bei Value Objects, das wir im Abschnitt 5.16.3 identifiziert haben, weiter.

5.16.5 Eine generische Basisklasse für Value Objects (Level 200)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=XjahoXgkik0>.

In Abbildung 130 haben wir gesehen, dass bei Klassen, die Value Objects repräsentieren, die Methoden einen großen Teil an repetitiven Code darstellen, der aber von den jeweiligen Feldern abhängt. Durch Reflection haben wir die Möglichkeit, zur Laufzeit uns alle Informationen über die

privaten Instanzfelder eines Objekts zu holen, sodass wir diesen repetitiven Code, der gegen das DRY-Prinzip verstößt, in einer Basisklasse auslagern können. Diese Klasse nennen wir `ValueObject` und ihr Code sieht wie folgt aus:

```

public abstract class ValueObject<T> : IEquatable<T> where T : class
{
    private readonly IReadOnlyList<FieldInfo> _feldInfos;
    private Nullable<int> _hashCode;
    private const BindingFlags InstanzUndPrivat = BindingFlags.NonPublic | BindingFlags.Instance;

    protected ValueObject() : this(typeof(T).GetFields(InstanzUndPrivat)) { }

    protected ValueObject(IReadOnlyList<FieldInfo> feldInfos) — Generics können in ihren möglichen Typen eingeschränkt werden mit dem where Schlüsselwort
    {
        if (feldInfos == null) throw new ArgumentNullException("feldInfos");
        _feldInfos = feldInfos;
    }

    public bool Equals(T other)
    {
        if (other == null) return false;
        return ReferenceEquals(this, other) || VergleicheFeldwerteMitReflection(other);
    }

    private bool VergleicheFeldwerteMitReflection(T other)
    {
        foreach (var feldInfo in _feldInfos)
        {
            object wertDiesesObjects = feldInfo.GetValue(this);
            object wertDesAnderenObjects = feldInfo.GetValue(other);
            if (wertDiesesObjects.Equals(wertDesAnderenObjects) == false)
                return false;
        }
        return true;
    } — In dieser Methode werden die Feldwerte zweier Objekte via Reflektion ausgelesen und verglichen
}

public override bool Equals(object obj)
{
    return Equals(obj as T);
}

public override int GetHashCode()
{
    if (_hashCode == null)
        _hashCode = BerechneHashCode();

    return _hashCode.Value;
}

private int BerechneHashCode()
{
    var hashCode = 0;
    foreach (var fieldInfo in _feldInfos)
    {
        object wert = fieldInfo.GetValue(this); — In dieser Methode wird der Hash Code über Reflection berechnet
        if (wert != null)
            hashCode ^= wert.GetHashCode();
    }
    return hashCode;
}

public static bool operator ==(ValueObject<T> first, ValueObject<T> second)
{
    return ReferenceEquals(first, null) == false ? first.Equals(second) : ReferenceEquals(second, null);
}

public static bool operator !=(ValueObject<T> first, ValueObject<T> second)
{
    return (first == second) == false;
}
}

```

Abbildung 136: Die Basisklasse `ValueObject<T>`

In Abbildung 136 sehen wir, dass `ValueObject`<T> generisch ist. Der Generic wurde hier genutzt, da diese Klasse `IEquatable`<T> implementieren soll, jedoch den Generic dieses Interfaces nicht auflösen soll. Wir sehen nach dem Interface jedoch eine uns unbekannte Angabe

`where T : class`. Mit `where` Angaben nach den Angaben der abgeleiteten Klasse bzw. der implementierten Interfaces kann man Beschränkungen für Generics festlegen. Hier wurde die Angabe `class` gewählt, sodass der Generic nur mit einem Typen aufgelöst werden kann, der eine Klasse ist (bei anderen Typangaben wie Interfaces, Strukturen oder Enumerationen wirft der Compiler einen Fehler). Weitere Details zur Einschränkung von Generics besprechen wir in einem späteren Abschnitt.

Die beiden wichtigen Methoden von `ValueObject`<T> heißen

`VergleicheFeldwerteMitReflection` und `BerechneHashCode`. In der ersten der beiden werden über die festgehaltenen Feldinfos iteriert und die Methode `fieldInfo.GetValue` zweimal aufgerufen. Dieser Methode übergibt man die Instanz, deren Feldwert ausgelesen werden soll, was einmal mit `this` und einmal mit `other` geschieht. Und genau dies bewirkt das tatsächliche Auslesen des entsprechenden Feldes bei beiden Objekten. Wir sehen hier auch, dass eventuell Boxing auftritt, wenn es sich beim Feldtypen um einen Wertetyp handelt (Reflection hat in der Tat einen relativ hohen Einfluss auf die Performance, nicht nur durch Boxing / Unboxing, sondern auch durch das dynamische Aufrufen von Mitgliedern über die Reflection-Metadaten). Die beiden dynamisch ausgelesenen Feldwerte werden im Anschluss auf Gleichheit überprüft. Sind alle Feldwerte gleich, so gilt das andere Objekt identisch zu `this`.

`BerechneHashCode` nutzt ebenfalls die Feldinfos, um die Werte dynamisch auszulesen und diese mit XOR zu verknüpfen. Dabei wird zusätzlich noch überprüft, ob der Wert ungleich null ist, um eine `NullReferenceException` zu vermeiden. Bitte beachten Sie auch, dass diese Funktion nur ein einziges Mal aufgerufen wird, genau dann, wenn der Hash Code zum ersten Mal angefordert wird. Der berechnete Wert wird im Feld `_hashCode` vom Typ `Nullable<int>` abgelegt, um bei weiteren Aufrufen von `GetHashCode` den Wert direkt zu liefern, ohne ihn neu berechnen zu müssen. Ein `Nullable`<T> ist dabei ein sog. Wrapper für einen Wertetyp (in diesem Fall eben `int`), damit man diesen Wert auch auf `null` setzen kann (was ja bei Wertetypen standardmäßig nicht geht). `Nullable`<T> bietet auch implizite Konvertierungen vom jeweiligen Zieltyp an, sodass bei der Anweisung `_hashCode = BerechneHashCode();` keine Cast notwendig ist. `Nullable`<T> wird so häufig eingesetzt, dass es dafür eine Kurzschreibweise in C# gibt: man setzt hinter einen Wertetyp einfach ein Fragezeichen. Statt `Nullable<int>` hätten wir also auch `int?` schreiben können.

Die Feldinfos, die in den beiden eben beschriebenen Methoden eingesetzt werden, werden der Klasse via Konstruktor mitgegeben. Der Standardkonstruktor leitet an diesen Konstruktor weiter und übergibt dabei alle Feldinfos, die privat und nicht-statisch sind. Genau das sind die Felder, die man üblicherweise braucht, prinzipiell kann eine ableitende Klasse aber beliebige `FieldInfo`-Objekte übergeben, wenn sie den Konstruktor mit Parameter direkt ansteuert. Bitte beachten Sie auch, dass diese `FieldInfo`-Collection über das Interface `IReadOnlyList<FieldInfo>` angesprochen wird, da wir in dieser Klasse diese Collection nicht manipulieren, sondern einfach nur die Feldinfos einsetzen möchten.

Die in den letzten drei Absätzen beschriebene Funktionalität wird dabei im uns bereits bekannten Gerüst von `Equals`- und `GetHashCode`-Methoden sowie Operatorüberladungen eingesetzt. Wenn wir unsere beiden bereits bekannten Klassen `Farbe` und `Adresse` so modifizieren, dass Sie von `ValueObject`<T> ableiten, verkürzt sich deren Implementierung deutlich und wir haben das DRY-Problem mit diesen Klassen gelöst, wie in folgender Abbildung zu sehen ist:

```

public class Farbe : ValueObject<Farbe> ←
{
    private readonly byte _rot;
    private readonly byte _grün;
    private readonly byte _blau;

    public Farbe(byte rot, byte grün, byte blau)
    {
        _rot = rot;
        _grün = grün;
        _blau = blau;
    }

    public byte Rot { get { return _rot; } }
    public byte Grün { get { return _grün; } }
    public byte Blau { get { return _blau; } }
}

public class Adresse : ValueObject<Adresse> ←
{
    public Adresse(string straße, string postleitzahl, string ort)
    {
        if (straße == null) throw new ArgumentNullException("straße");
        if (postleitzahl == null) throw new ArgumentNullException("postleitzahl");
        if (ort == null) throw new ArgumentNullException("ort");

        Straße = straße;
        Postleitzahl = postleitzahl;
        Ort = ort;
    }

    public string Straße { get; private set; }
    public string Postleitzahl { get; private set; }
    public string Ort { get; private set; }
}

```

Anstatt alle Members für die Gleichheitsüberprüfung selbst zu implementieren, können Farbe und Adresse von ValueObject<T> ableiten

Abbildung 137: Klassen von ValueObject<T> ableiten

Anhand dieses Beispiels haben wir gesehen, wie man ein relativ komplexes Problem durch Kombination von zwei Mechanismen, Generics und Reflection, lösen kann.

5.16.6 Zusammenfassung für Generics

Das Video hierzu finden Sie unter https://www.youtube.com/watch?v=ftmrlnN_zyU.

Wir haben in den letzten Abschnitten im Detail betrachtet, dass es sich bei Generics um Platzhalter für Typen handelt, sodass eine bestimmte Funktionalität nach dem DRY-Prinzip und trotzdem typsicher implementiert werden kann. Wir haben festgestellt, dass man Generics auf Klassen und Interfaces als auch direkt auf Methoden definieren kann. Dabei können beliebig viele Generics definiert werden, allerdings müssen diese einen unterschiedlichen Bezeichner haben, damit man diese unterscheiden kann.

Wenn man einen generischen Typ instanziiieren will, muss man alle Generics durch konkrete Typen auflösen. Beim Ableiten oder Implementieren von generischen Typen kann man sich entscheiden, ob man den Generic auflösen oder weiterführen möchte. Dasselbe gilt für den Aufruf von generischen Methoden – entweder man löst alle Generics auf, oder man macht die aufrufende Methode selber generisch. Beim Auflösen von generischen Parametern oder Rückgabetypen kann der Compiler

unterstützen, da er den Genericotyp durch die Typen der angegebenen Werte bzw. Variablen auflöst. Dies bezeichnet man als Typinferenz (engl. Type Inference).

Weiterhin haben wir am Beispiel von `ValueObject<T>` gesehen, dass man die möglichen Typen für Generics durch die `where` Angabe beschränken kann – es dürfen dann beim Auflösen von Generics ausschließlich Typen angegeben werden, die alle `where` Beschränkungen enthalten. Dabei darf man auch mehrere `where` Beschränkungen hintereinander schreiben, wie in folgendem abstrakten Codebeispiel zu sehen ist.

```
public class Beispielklasse<T, U> where T : IComparable<T>, new()
    where U : class
{
    // Implementierung weglassen
}
```

Für jeden Generic, der eingeschränkt werden soll, macht man mehrere `where` Angaben.
Mehrere Restriktionen bei einer `where` Angabe werden kommagetrennt aufgeführt.

Abbildung 138: Mehrere Beschränkungen für Generics festlegen

Dabei sind folgende Beschränkungen zulässig:

Beschränkung	Bemerkung
<code>where T : struct</code>	Das Typargument muss ein Wertetyp sein.
<code>where T : class</code>	Das Typargument muss ein Referenztyp sein (dies trifft für all Klassen, Interfaces, Delegates oder Arraytypen zu).
<code>where T : new()</code>	Das Typargument muss einen öffentlichen Standardkonstruktor besitzen. Diese Angabe muss am Ende stehen, wenn sie in Kombination mit anderen genutzt wird.
<code>where T : Basisklassename</code>	Das Typargument muss die spezifizierten Klasse sein oder von ihr ableiten.
<code>where T : Interfacename</code>	Das Typargument muss das spezifizierte Interface sein oder davon ableiten. Es können auch mehrere Interfaces kommagetrennt hintereinander aufgeführt werden.
<code>where T : U</code>	Das Typargument für T muss derselbe Typ wie der Generic U sein oder davon ableiten.

Abbildung 139: Mögliche Beschränkungen für Generics

Generics, die keinen Beschränkungen unterliegen, nennt man unbeschränkte Generics. Für sie gelten die folgenden Regeln:

- Die Operatoren `!=` und `==` können auf sie nicht angewendet werden, da es keine Garantie gibt, dass der konkrete Typ diese Operatoren unterstützt.
- Sie können zu `object` implizit konvertiert oder zu Interfacetypen gecastet werden.
- Sie können mit `null` verglichen werden. Falls der konkrete Typ ein Wertetyp ist, gibt dieser Vergleich immer `false` zurück.

Des Weiteren bringen Einschränkungen den Vorteil, dass die entsprechende API von Interfaces und/oder Klassen zur Verfügung steht, wenn ein Generic auf diese Typen beschränkt wurde. Ansonsten hat man für T standardmäßig nur die API von `object` zur Verfügung.

5.17 Das Typsystem von C#

5.17.1 Welche Typen unterstützt C#?

Mit Klassen und Interfaces haben wir uns mittlerweile ausgiebig beschäftigt. Doch dies sind nur zwei von insgesamt fünf möglichen Typen, die man in C# definieren kann, wie folgende Abbildung zeigt:

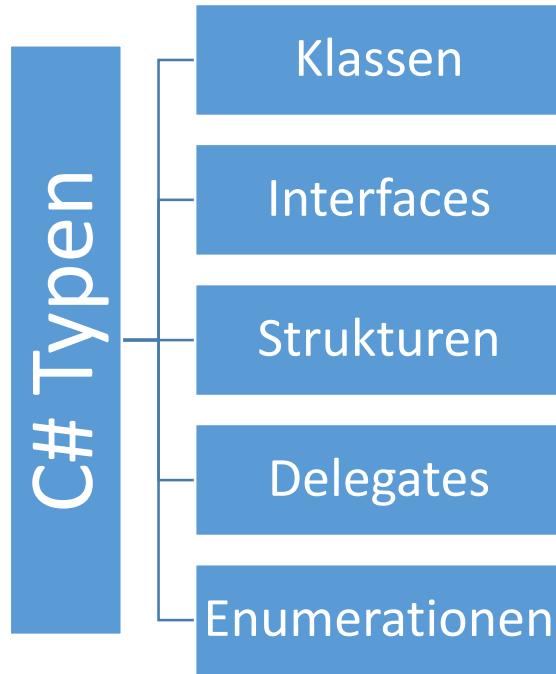


Abbildung 140: Das Typsystem von C#

Auch Strukturen und Enumerationen haben wir bereits in einigen Abschnitten erwähnt, auf die genaue Funktionsweise von ihnen und Delegates gehen wir aber in den nächsten Abschnitten ein.

5.17.2 Enumerationen

5.17.2.1 Enumerationen definieren

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=AMiW4wRK3xQ>.

Eine Enumeration ist eine Menge an konstanten Werten, die thematisch zusammengehören. Bitte beachten Sie, dass Enumerationen rein gar nichts mit dem bereits besprochenen Thema Collections und `IEnumerable<T>` zu tun haben. Wenn der Enumerationstyp eingesetzt wird, kann dann genau einer dieser konstanten Werte zugewiesen werden. In der folgenden Abbildung sehen Sie, wie Enumerationen definiert werden können:



Abbildung 141: Enumerationen definieren

In Abbildung 141 wird die Enumeration `Kartenfarbe` definiert. Dies wird gemacht, indem zuerst ein Zugriffsmodifizierer festgelegt wird gefolgt vom Schlüsselwort `enum` und dem Bezeichner einer Enumeration. Wie bei den anderen Typen auch folgen darauf geschweift Klammern, die den Scope der Enumeration festlegen. Innerhalb dieses Scopes können Konstanten definiert werden, im obigen Beispiel Kreuz, Pik, Herz und Karo. Die konstanten Werte werden dabei kommagetrennt angegeben.

Intern verhalten sich diese Konstanten genauso wie `int` Werte, d.h. der Konstanten Kreuz wird der Wert 0 vergeben, Pik bekommt die 1, Herz die 2 usw. Es ist jedoch auch möglich, manuell Werte für Konstanten zu vergeben, wie im Folgenden modifiziertem Beispiel zu sehen ist:

```
public enum Kartenfarbe
{
    Kreuz = 12,
    Pik = 11,
    Herz = 10,
    Karo = 9
}
```

Den Enumerationswerten kann auch manuell ein beliebiger Ganzahlwert zugewiesen werden

Abbildung 142: Werte für Enumerationsmitglieder manuell festlegen

Wie in Abbildung 142, weist man dazu den Enumerationsmitgliedern einfach bei der Definition mit dem `=` Operator einen Wert zu. Anstatt jeder Konstanten aber einen eigenen Wert zuzuweisen, beherrscht der C# Compiler kann aber auch noch eine andere Art der Beschreibung von Werten für Enumerationskonstanten:

Enumerationen können statt als int auch
als ein beliebig andere Ganzahltyp
interpretiert werden, allerdings nutzt man
dies im Normalfall nicht

```
public enum Kartenwert : byte
{
    Sieben = 7,
    Acht,
    Neun,
    Zehn,
    Bube,
    Dame,
    König,
    Ass
}
```

Sieben = 7, Acht, Neun, Zehn, Bube, Dame, König, Ass

Wenn der Wert einer Konstante manuell festgelegt wird, werden die nachfolgenden Elemente automatisch nach diesem Wert angepasst

Abbildung 143: Automatische Fortfhrung von Werten fr Enumerationskonstanten

In Abbildung 143 sehen Sie, dass bei der Enumeration `Kartenwert` nur das Mitglied `Sieben` auf einen tatsächlichen Wert festgelegt wurde. Alle darauffolgenden Konstanten werden automatisch (anhand des Wertes der vorherigen Konstante) angepasst, d.h. `Acht` bekommt den Wert 8, `Neun` den Wert 9, `Bube` den Wert 11, `Dame` den Wert 12 usw. Diese Möglichkeit ist letztendlich aber nur eine Programmiererleichterung (Syntactic Sugar).

Des Weiteren sieht man die Möglichkeit, dass Enumerationen nicht nur als `int` Werte, sondern auch als beliebig andere Ganzahltypen interpretiert werden können. Dazu setzt man hinter den Bezeichner einen Doppelpunkt und gibt den entsprechenden Typ an, im obigen Beispiel `byte`. Üblicherweise wird dies aber nicht eingesetzt, da heutige Prozessorarchitekturen auf 32 Bit Datentypen optimiert sind. Die Angabe von `byte` im obigen Fall würde also keinen Performancezuwachs bringen. Und auch wenn dieser Syntax etwas an Vererbung bei Klassen und Interfaces erinnert, hat dies damit rein gar nichts zu tun.

5.17.2.2 Enumerationen einsetzen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ST4EwczVCJE>.

Enumerationen können genauso wie andere Typen über Variablen, Parameter oder Felder eingesetzt werden. Sehen wir uns hierzu folgendes Beispiel an:

```

static void Main()
{
    Kartenfarbe farbe1 = Kartenfarbe.Herz;
    Kartenwert wert1 = Kartenwert.König;           Variablen, Parameter oder Konstanten
                                                    halten genau einen Wert einer
                                                    Enumeration

    int wertAlsZahl = (int) wert1;
    Kartenwert wert2 = (Kartenwert) wertAlsZahl;   Enumerationswerte können zu den
                                                    numerischen Datentypen explizit
                                                    konvertiert werden (und umgekehrt)

    Kartenwert wert3 = Kartenfarbe.Pik;           Enumerationen sind typsicher, bspw.
    Kartenfarbe farbe2 = wert1;                   kann man ihnen nicht einfach Werte
                                                    einer anderen Enumeration zuweisen
}

```

Abbildung 144: Enumerationen einsetzen

In Abbildung 144 sehen Sie, dass innerhalb der Main Methode zunächst die beiden Variablen farbe1 und wert1 definiert werden, die von den uns bereits bekannten Typen **Kartenfarbe** und **Kartenwert** sind. Diese Variablen können dann genau einen Wert der Enumeration halten. Im obigen Beispiel ist das **Kartenfarbe.Herz** und **Kartenwert.König**. Man greift über diese Syntax auf die einzelnen Konstanten einer Enumeration zu, wobei die Schreibweise identisch ist mit dem Zugriff auf bspw. statische Felder einer Klasse.

In den nächsten beiden Anweisungen sehen Sie, dass man den Wert einer Enumerationskonstante auch explizit in einen numerischen Datentyp konvertieren kann. Das ist bspw. hier hilfreich, wenn man mit dem Kartenwert rechnen möchte, da Enumerationen nur eingeschränkt numerische Operatoren unterstützen. Im Prinzip ist das nur der + und - Operator, mit denen man aus einem bestehenden Enumerationswert einen anderen berechnen kann, der um eine bestimmte Zahl verschoben ist (Operatoren werden in Zusammenhang mit Enumerationen im Programmieralltag nahezu gar nicht eingesetzt).

Natürlich kann man auch versuchen, eine Zahl als Enumerationswert zu interpretieren. Auch hierfür ist ein Cast notwendig. Vorsicht: wird hier eine Zahl verwendet, die keiner Konstanten der Enumeration entspricht, wird zur Laufzeit keine Exception geworfen. Der Wert der entsprechenden Variablen ist dann invalide.

Ansonsten sehen wir aber in den letzten beiden Anweisungen des obigen Beispiels, dass Enumerationen typsicher sind, d.h. man kann nicht Werte einer anderen Enumeration zu einer Variablen zuweisen oder Zahlen, ohne dass man Casts verwendet.

5.17.2.3 Enumerationen als Bitfelder verwenden

Das Video hierzu finden Sie unter https://www.youtube.com/watch?v=Jp9IB-xD_el.

In Abbildung 133 haben wir bereits die Enumeration **BindingFlags** gesehen, die mit dem **FlagsAttribute** ausgestattet ist. Wird dieses Attribut auf eine Enumeration angewendet, so können ihre Einzelkonstanten mit dem Bitweise | Operator kombiniert werden, wie wir es bereits in den Beispielen nach dieser Abbildung eingesetzt haben. Die Konstanten werden bei Anwendung dieses Attributs automatisch so gesetzt, dass ihre Werte ein Vielfaches von 2 sind, sodass nur ein einzelnes der 32 Integerbits für einen Wert gesetzt ist. Diese Funktionalität nimmt man häufig dann her, wenn man bei einer API bestimmte Auswahlmöglichkeiten anbieten möchte über die einzelnen Konstanten und diese auch noch kombinierbar sein sollen.

Wenn sie bei Bitfeld-Enumerationen die Werte für die Konstanten selber definieren möchten, passen Sie auf, dass ihre Werte ausschließlich Vielfache von 2 sind, sonst kann es beim Verknüpfen zu einer

Kollision kommen. Wenn Sie einen Bitfeld-Enumerationswert auf eine bestimmte Option hin überprüfen möchten, setzen Sie dazu den & Operator ein.

5.17.2.4 Enum – die Basisklasse für Enumerationen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=NvUnZytlX8>.

Enumerationen leiten implizit immer von der Klasse `System.Enum` ab. Auf dieser Klasse sind auch noch einige interessante Methoden definiert, die man im Zusammenhang mit Enumerationen einsetzen kann:

Methode	Bemerkung
<code>GetName</code>	Gibt den Namen einer Konstanten einer Enumeration als <code>string</code> zurück.
<code>GetNames</code>	Gibt alle Namen eines Enumerationstyps als Array zurück.
<code>GetValues</code>	Gibt alle Konstanten einer Enumeration als Array zurück.
<code>HasFlag</code>	Überprüft, ob ein oder mehrere Bitfelder eines Enumerationswerts gesetzt sind.
<code>IsDefined</code>	Überprüft, ob eine gewisse Zahl als Konstante für eine bestimmte Enumeration definiert ist.
<code>Parse</code>	Konvertiert einen <code>string</code> Wert zu einem Enumerationswert.
<code>TryParse</code>	Wie Parse, allerdings wird dabei keine Exception geworfen.

Abbildung 145: Hilfreiche Methoden der Klasse `Enum`

Es ist jedoch nicht möglich, manuell von `Enum` abzuleiten. Der Compiler wirft einen Fehler beim Erstellvorgang, wenn dies versucht wird. In der Vererbungshierarchie steht `Enum` unterhalb der Klasse `System.ValueType`, die Wertetypen repräsentiert. Auch von dieser kann nicht direkt abgeleitet werden.

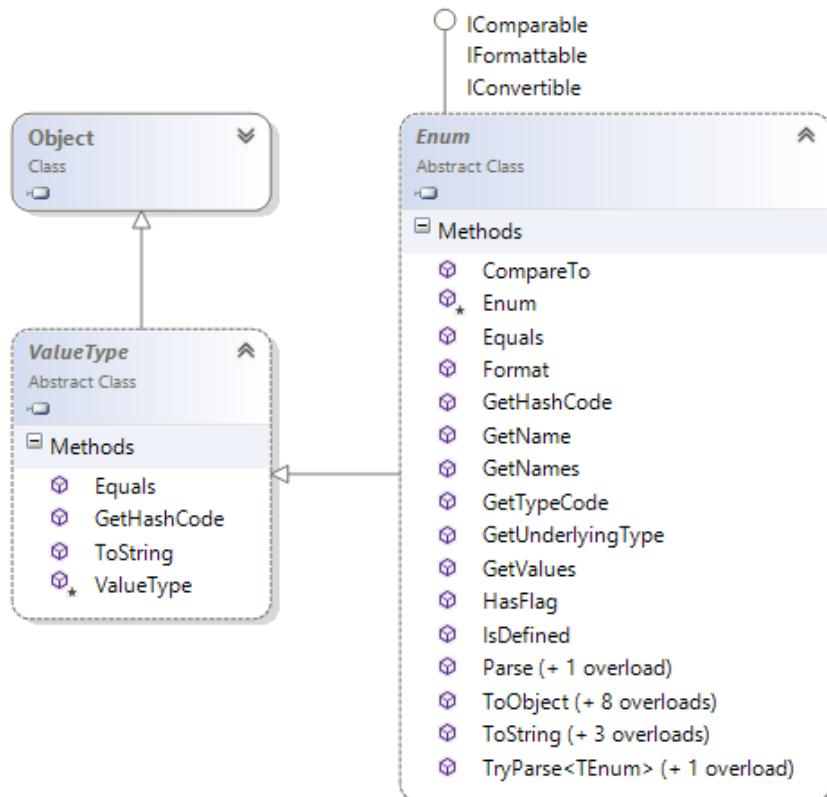


Abbildung 146: Klassenhierarchie zu `Enum`

Wie in Abbildung 146 zu sehen ist, lassen sich diese Klassen auch nicht direkt instanzieren, da sowohl **Enum** als auch **ValueType** abstrakt sind.

5.17.2.5 Ein komplexeres Beispiel für den Einsatz von Enumerationen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=KE5qi4CAJtU>.

Nachdem wir uns in den vorherigen Abschnitten genauer mit Enumerationen auseinandergesetzt, möchte ich hier nochmals ein größeres Beispiel machen, in dem Enumerationen eingesetzt werden. Das sehen wir uns zunächst die Klasse **Spielkarte** an:

```
public class Spielkarte : ValueObject<Spielkarte>
{
    private readonly Kartenfarbe _farbe;
    private readonly Kartenwert _wert; Spielkarte ist ein Value Object

    public Spielkarte(Kartenfarbe farbe, Kartenwert wert)
    {
        _farbe = farbe;
        _wert = wert;
    }

    public Kartenfarbe Farbe { get { return _farbe; } }
    public Kartenwert Wert { get { return _wert; } }
}
```

Abbildung 147: Die Klasse Spielkarte

Die in Abbildung 147 zu sehende Klasse nutzt zwei Enumerationswerte, **farbe** und **wert**, um eine Spielkarte zu repräsentieren. Sie ist dabei ein Value Object, ein nach der Instanziierung unveränderliches Objekt, das identisch mit anderen Instanzen der Klasse ist, wenn deren Werte gleich sind. Dazu leiten wir von der Basisklasse **ValueObject**<T> ab, die wir bereits besprochen haben.

Damit man nun beispielsweise einen Kartenstapel erstellen möchte, in dem alle möglichen Karten enthalten sind, könnte man folgende Methode schreiben:

```

public IList<Spielkarte> ErstelleKartenstapel()
{
    var kartenStapel = new List<Spielkarte>();
    var alleFarben = Enum.GetValues(typeof(Kartenfarbe));
    var alleKartenwerte = Enum.GetValues(typeof(Kartenwert));

    foreach (Kartenfarbe kartenfarbe in alleFarben)
    {
        foreach (Kartenwert kartenwert in alleKartenwerte)
        {
            kartenStapel.Add(new Spielkarte(kartenfarbe, kartenwert));
        }
    }

    return kartenStapel;
}

```

Abbildung 148: Eine Collection mit allen möglichen Spielkarten erstellen

In der in Abbildung 148 dargestellten Methode `ErstelleKartenstapel` wird zunächst eine neue Instanz von `List<T>` erstellt. Über `Enum.GetValues` werden alle möglichen Werte der Enumerationen `Kartenfarbe` und `Kartenwert` geholt. Über diese beiden Arrays wird jeweils in einer `foreach` Schleife iteriert, sodass zu jeder möglichen Kombination eine neue Spielkarte erstellt werden und in den Kartenstapel eingefügt werden kann. Diese API könnte so als Fundament für ein Kartenspiel genutzt werden.

5.17.2.6 Wann sollte man Enumerationen einsetzen?

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=rQwygXlzVuo>.

Enumerationen sind ein relativ einfach einzusetzendes Programmierwerkzeug, mit der man thematisch zusammengehörende feste Werte anlegen kann. Dies kann man nutzen, wenn diese Konstanten sehr fest sind und man nicht in die Gefahr läuft, dass sich diese Konstanten während des Entwicklungs- oder gar während des Lebenszyklus der Applikation ändern. Das ist bspw. bei `Kartenfarbe` und `Kartenwert` der Fall: die entsprechenden Werte sind in weiten Regionen der Welt bekannt und standardisiert. Es ist sehr unwahrscheinlich, dass sich diese Werte in absehbarer Zukunft ändern werden.

Das muss aber nicht bei jedem festen Wert so sein. Gerade wenn sich bestimmte feste Werte ändern könnten oder wenn bspw. für verschiedene Kunden andere feste Werte verwendet werden, sollte man Konstanten nicht im Code festlegen, sondern dann lieber beim Programmstart diese Werte aus einer Konfigurationsdatei laden und diese dann bspw. als `string` oder `int` interpretieren. Wenn Sie im Zweifel sind, sollten Sie sich lieber gegen Enumerationen entscheiden und stattdessen diese Werte über einen gewissen Mechanismus konfigurierbar machen.

5.17.3 Delegates

Delegates (dt. Delegat, Stellvertreter) werden in C# genutzt, um Referenzen auf Methoden festzuhalten. Beim Erstellen eines Delegate gibt man eine Methode an, auf die dieser Delegate zeigen soll. Im Anschluss kann man den Delegate wie eine Methode aufrufen, um die an ihn gebundene Methode auszulösen. Im Wesentlichen verhalten sich Delegates also wie Funktionszeiger in C. In den kommenden Abschnitten sehen wir uns alle Details zu diesem Typ an.

5.17.3.1 Delegates definieren und instanziieren

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=A5snl1VnrzQ>.

Die Definition eines Delegates ist im Vergleich zu anderen Typen wie Interfaces und Klassen sehr kurz gehalten, denn bei dessen Definition gibt man nur an, wie der Funktionskopf einer Methode aussehen muss, damit sie konform zum Delegate ist. Sehen wir uns dazu ein Beispiel an:



Abbildung 149: Definition eines Delegate

In Abbildung 149 sehen Sie, dass ein Delegate sehr ähnlich wie ein Funktionskopf aussieht. Zunächst wird ein Zugriffsmodifizierer angegeben, gefolgt vom Schlüsselwort **delegate**. Im Anschluss kommt der Rückgabetyp, der Bezeichner des Delegate sowie die Parameter, die der Delegate entgegennimmt. Anders als bei anderen Typen wird danach kein Scope mit geschweiften Klammern aufgemacht, sondern die Typdefinition direkt mit einem Semikolon abgeschlossen.

Wie bereits erwähnt, können Delegates Methoden referenzieren, welche denselben Rückgabetypen und dieselbe Parameterliste besitzen. Im Beispiel oben wären das also alle Methoden, die als Parameter einen **int** Wert entgegen nehmen und **int** als Rückgabetyp haben. Dies wird im folgenden Beispiel nochmals verdeutlicht:

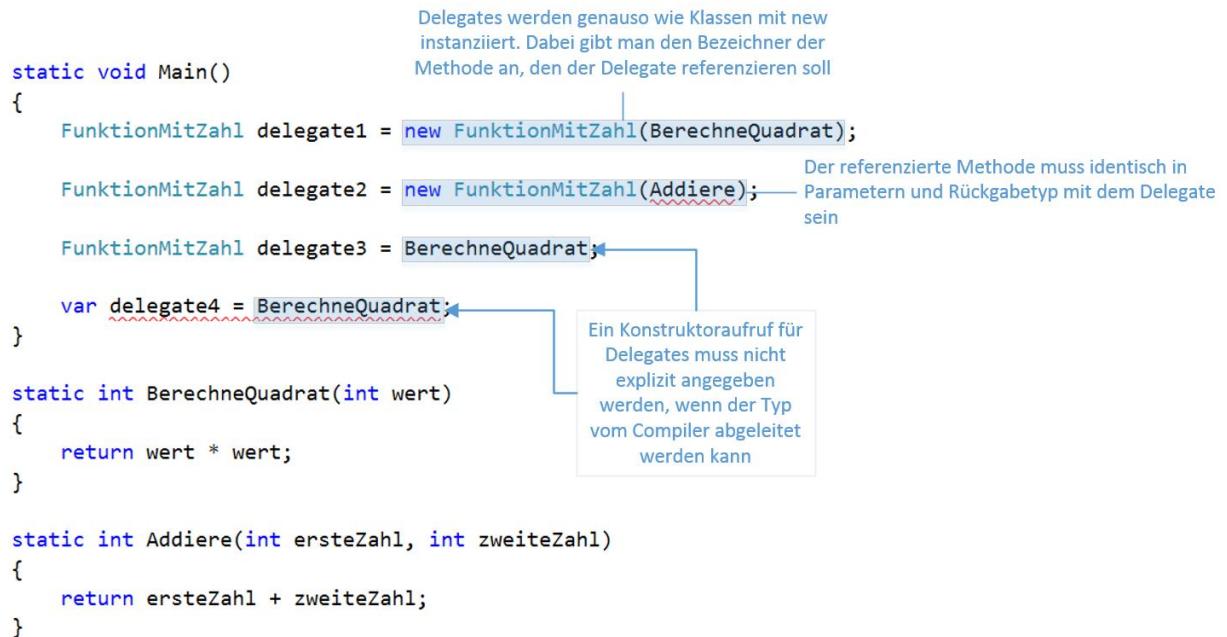


Abbildung 150: Delegates instanziieren

In Abbildung 150 sehen Sie, dass innerhalb der Main Methode versucht wird, vier Delegates vom Typ **FunktionMitZahl** zu instanziieren. Bei der ersten Anweisung sehen wir, dass das prinzipielle Vorgehen dabei genauso wie bei Klassen ist: mit dem **new** Aufruf auf den Typbezeichner wird ein Objekt von **FunktionMitZahl** instanziert, als Konstruktorparameter muss man die Methode angeben, die beim Beispiel oben **BerechneQuadrat** heißt. Bitte beachten Sie, dass bei der Angabe

von BerechneQuadrat keine runden Klammern eingesetzt werden wie beim Aufruf einer Funktion, sondern ausschließlich der Bezeichner der Methode.

Bei delegate2 sehen wir, dass der Delegate nicht instanziert werden kann, wenn die referenzierte Methode in Parametern und Rückgabetyp dem Delegate entspricht: die Methode Addiere, die hier angegeben wird, hat zwar den gleichen Rückgabetypen, allerdings zwei Parameter, sodass sie nicht von einem Delegate vom Typen FunktionMitZahl referenziert werden kann. Hierzu müsste ein anderer Delegatetyp verwendet werden.

Bei delegate3 sehen wir, dass die Angabe des new Operators gefolgt vom Typbezeichner nicht notwendig ist: dies wird automatisch vom Compiler erledigt, wenn er wie in diesem Fall aus der Variablendefinition den Typ des Delegates ableiten kann. Genau das geht bei Anweisung vier nicht, da dort var als Typ für die Variable gewählt wurde und der Compiler deshalb nicht weiß, welchen Delegatetyp er hier zur Kapselung von BerechneQuadrat verwenden soll (es könnte ja auch einen anderen Delegatetypen geben, der die gleichen Parameter und Rückgabetyp vorgibt).

Nachdem wir uns in diesem Abschnitt mit der Definition und Instanziierung von Delegates beschäftigt haben, schauen wir uns im kommenden Abschnitt an, was passiert, wenn man einen Delegate aufruft.

5.17.3.2 Delegates aufrufen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=wc4mQfU4Hvg>.

Sobald ein Delegate instanziert ist und dadurch eine andere Methode referenziert, kann man letztere über den Delegate aufrufen. Die Syntax dazu ist identisch mit der zum direkten Aufruf von Methoden, d.h. man nutzt runde Klammern nach dem Bezeichner für den Delegate, in welche die Parameter gesetzt werden. Wenn der Aufruf vorbei ist und dabei ein Wert zurückgegeben wurde, kann dieser bspw. durch eine Zuweisung in einer Variablen festgehalten werden. Diese wird auch im folgenden Beispiel verdeutlicht:

```
static void Main()
{
    FunktionMitZahl @delegate = BerechneQuadrat;

    var ergebnis = @delegate(42);
    Console.WriteLine(ergebnis);
}

static int BerechneQuadrat(int zahl)
{
    return zahl * zahl;
```

Hier wird die Methode BerechneQuadrat über einen Delegate ausgeführt

Abbildung 151: Methoden via Delegates ausführen

In Abbildung 151 sehen Sie, dass zunächst ein Delegate instanziert wird, der die Methode BerechneQuadrat referenziert. In der zweiten Anweisung wird der Delegate aufgerufen, genauso wie wir es bei Methoden schon gewohnt sind. Der Rückgabewert wird nach dem Aufruf in der Variablen ergebnis gespeichert und im letzten Statement auf der Konsole ausgegeben.

Wir sehen also, dass es keinen Unterschied macht, ob wir Methoden direkt aufrufen oder dies über einen Delegaten machen. Die Frage, die bleibt, ist natürlich der Vorteil von Delegates: warum kann man nicht direkt die Methode aufrufen, die man braucht und sollte stattdessen einen Delegate einsetzen?

5.17.3.3 Das Hollywood-Prinzip anhand von System.Threading.Timer

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=w1Ny8QRZoyQ>.

Zugegeben: das in Abbildung 151 gezeigt Beispiel für Delegates macht keinen Sinn – es soll nur die Arbeitsweise von Delegates zeigen. Mit Delegates ist es jedoch möglich, dass man Referenzen auf bestimmte Methoden übergibt – das sorgt dafür, dass bspw. bestimmte Klassen Methoden aufrufen können, die sie per Parameter erhalten haben. Damit wir das anhand eines Beispiels sehen, werfen wir jetzt einen Blick auf die Klasse `System.Threading.Timer`. Ein Timer ist eine Klasse, welche bestimmte Funktionalität nach einer gewissen Zeitspanne immer wieder aufruft. Dieser Aufruf wird auch als Tick bezeichnet.

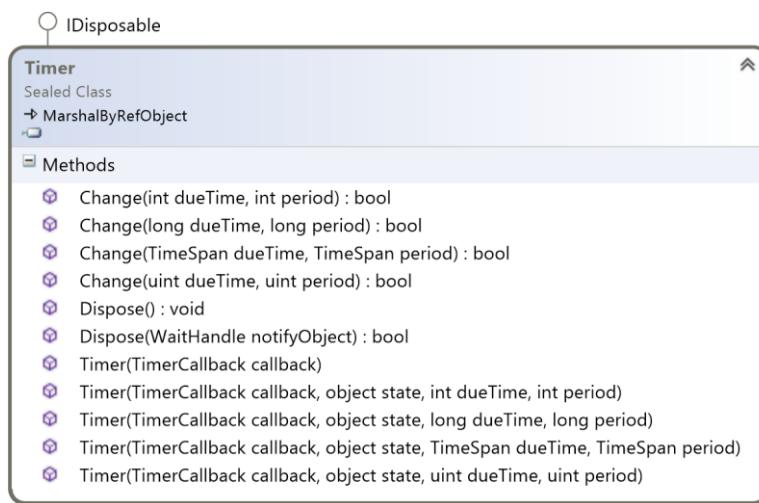


Abbildung 152: Aufbau der Klasse `System.Threading.Timer`

In Abbildung 152 sehen Sie, wie diese Klasse prinzipiell aufgebaut ist. Im Besonderen sind die fünf Konstruktoren interessant, welche den Timer initialisieren. Alle nutzen einen Delegate, der die Methode referenziert, die vom Timer aufgerufen werden soll, wenn ein Tick stattfindet. Der Delegatentyp heißt `System.Threading.TickerCallback` und ist wie folgt definiert:

```
public delegate void TimerCallback(object state);
```

Dieser Delegate kann also Methoden referenzieren, deren Rückgabetyp `void` ist und die genau einen Parameter vom Typ `object` entgegennehmen. Mit diesem Parameter kann man sich zusätzliche Statusobjekte in die aufgerufene Methode übergeben lassen, was aber nicht notwendig ist. In unseren kommenden Beispielen werden wir diesen Parameter auch nicht aktiv einsetzen.

Nachdem der Timer initialisiert ist, beginnt er sofort zu laufen (außer man nutzt den Konstruktor mit nur einem Parameter). Über die Parameter `dueTime` und `period` kann man steuern, nach welchem Zeitintervall der erste Tick kommt und wie lange die folgenden Zeitintervalle sind, die zwischen den Ticks vergehen. Möchte man das Verhalten eines laufenden Timers ändern, so muss man eine der `Change` Überladungen aufrufen. Soll der Timer komplett beendet werden, dann reicht ein Aufruf auf `Dispose`. Genauere Infos zu dieser API gibt es in der [MSDN Library](#).

Wir möchten in unserem Beispiel über den Timer Zeile für Zeile einen ASCII-Zeichenbaum ausgeben. Nach dem Ausgeben einer Zeile soll etwas Zeit verstreichen (bspw. 300 Millisekunden), bevor die

nächste Zeile ausgegeben werden soll, was wir über den Timer steuern möchten. Die Klasse, die das Ausgeben eines ASCII-Zeichenbaums übernimmt, sieht wie folgt aus:

```
public class AsciiBaum
{
    private readonly char _zeichen;
    private int _anzahlLeerstellen;
    private int _anzahlZeichen = 1;

    public AsciiBaum(int anzahlZeilen, char zeichen = '*')
    {
        if (anzahlZeilen < 2)
            throw new ArgumentOutOfRangeException("anzahlZeilen");
        _zeichen = zeichen;
        _anzahlLeerstellen = anzahlZeilen - 1;
    }

    public void GibNächsteZeileAus()
    {
        if (IstFertigMitAusgabe)
            return;

        var leerstellen = new string(' ', _anzahlLeerstellen);
        var zeichen = new string(_zeichen, _anzahlZeichen);
        Console.WriteLine(leerstellen + zeichen);

        _anzahlLeerstellen--;
        _anzahlZeichen += 2;
    }

    public bool IstFertigMitAusgabe{ get { return _anzahlLeerstellen == -1; } }
}
```

Welches ASCII-Zeichen genutzt wird, kann über den Konstruktor definiert werden. Standardmäßig wird das Sternchen (*) genutzt. Über diese Syntax kann man Standardwerte für Parameter festlegen, deren Angabe dann beim Aufruf optional ist

In dieser Methode wird die neue Zeile als string zusammengestellt und ausgegeben sowie die Felder für den nächsten Aufruf aktualisiert

Abbildung 153: Die Klasse AsciiBaum

In Abbildung 153 sehen Sie die Klasse `AsciiBaum`, die ein ASCII-Zeichen über eine vorgegebene Anzahl an Zeilen ausgibt. Das Wesentliche passiert in der Funktion `GibNächsteZeileAus`, in der nach einer Überprüfung, ob noch Zeilen ausgegeben werden müssen, jeweils zwei Strings erstellt werden: der erste enthält die richtige Anzahl Leerzeichen, der zweite die richtige Anzahl des ASCII-Zeichens. Beide Strings werden dann zusammengefügt und über die Konsole ausgegeben. Zuletzt werden die Feldwerte für die Anzahl an Leerstellen und Zeichen für den nächsten Aufruf angepasst.

Ebenfalls sehen wir im Konstruktor nebenbei eine kleine Neuerung: man kann Parameter mit Standardwerten vorbelegen, hier geschehen über die Angabe `char zeichen = '*'`. Eine Wertangabe für den Aufrufer ist dann optional – er kann den Parameter entweder angeben oder weglassen. Genau aus diesem Grund müssen auch alle Parameter mit Standardwert am Ende der Parameterliste liegen.

Möchte man diese beiden Bausteine zusammen einsetzen, könnte das in etwa wie folgt aussehen:

```

class Program
{
    private static Timer _timer;
    private static AsciiBaum _asciiBaum;

    static void Main()
    {
        _asciiBaum = new AsciiBaum(15);
        var interval = TimeSpan.FromMilliseconds(300);
        _timer = new Timer(WennTimerTickt, null, interval, interval);

        Console.ReadLine();
    }

    static void WennTimerTickt(object state)
    {
        _asciiBaum.GibNächsteZeileAus();
        if (_asciiBaum.IstFertigMitAusgabe)
        {
            _timer.Dispose();
        }
    }
}

```

Als ersten Parameter erhält der Timer einen Delegate auf die Methode, die er bei jedem Tick aufrufen soll. Nach der Initialisierung läuft er sofort los.

Diese Methode wird bei jedem Tick des Timers über den Delegate aufgerufen

Abbildung 154: Timer und AsciiBaum in Kombination einsetzen

In Abbildung 154 wird zunächst ein `AsciiBaum`-Objekt instanziert mit der Angabe, dass insgesamt fünfzehn Zeilen ausgegeben werden sollen (den zweiten Konstruktorparameter können wir wie gesagt weglassen, da dafür ein Standardwert existiert). Als nächstes wird eine `TimeSpan` Struktur initialisiert, welche das Intervall für den Timer spezifiziert. Zuletzt wird der Timer gestartet: und hier passieren zwei interessante Sachen:

- Über einen Delegate wird die Methode `WennTimerTickt` an den Timer übergeben. Somit kann der Timer diese Methode bei jedem Tick aufrufen.
- Sobald der Timer nach der Initialisierung losläuft, ändert sich der Kontrollfluss vehement: der Timer wartet das angegebene Zeitintervall ab und ruft dann selbstständig die `WennTimerTickt` Methode über den Delegate auf. Sehen Sie genau hin: diese Methode wird nicht ein einziges Mal von uns direkt aufgerufen.

Diese Änderung des Kontrollflusses bezeichnet man als Hollywood-Prinzip, dass aus der Filmbranche bekannt ist: "Don't call us, we call you". Ursprünglich war damit gemeint, dass ein Schauspielbewerber sich nach dem Vorstellungsgespräch nicht um ein Telefonat bemühen braucht, um den Status der Bewerbung (wurde er/sie genommen oder nicht) abzufragen – die Filmproduktion meldet sich selbstständig, wahrscheinlich um nicht hunderten Bewerbern auf die verschiedenen Rollen jeweils eine Absage erteilen zu müssen. Aus diesem Klischee wurde ein Prinzip in der Programmierung: nämlich das man bestimmte Methoden nicht aktiv selbst aufruft, was als linearer Kontrollfluss bezeichnet wird, sondern das bestimmte Komponenten bei bestimmten Ereignissen (in unserem Beispiel eben ein Tick des Timers) bestimmte Funktionalität aufrufen, die ihnen vorher als Referenz übergeben wird – diese Komponenten sind also in dieser Hinsicht konfigurierbar.

Dieses Prinzip kann man sich auch in einem Sequenzdiagramm verdeutlichen, dass wir in folgender Abbildung sehen:

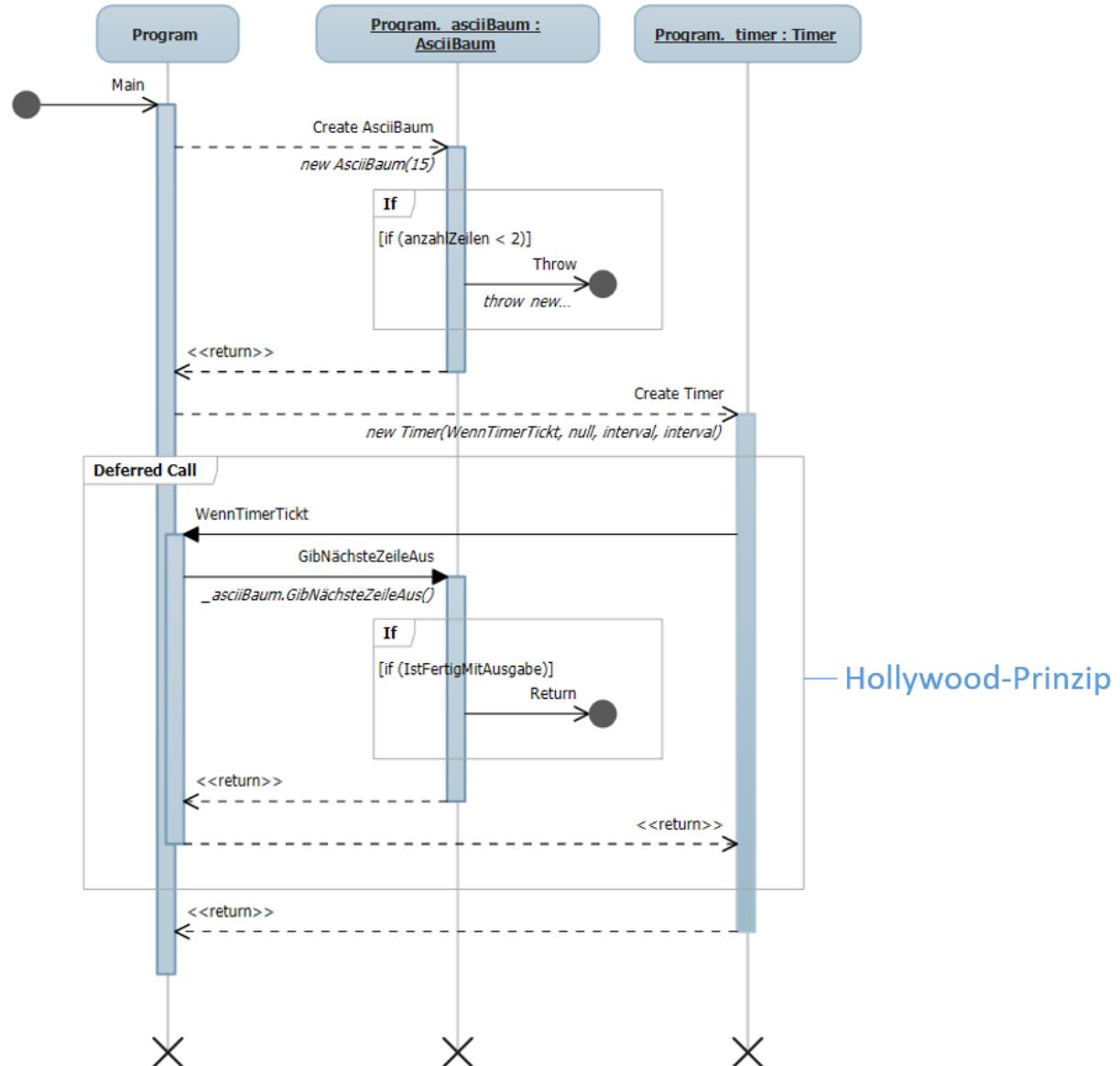


Abbildung 155: Hollywood-Prinzip in Sequenzdiagramm dargestellt

In Abbildung 155 sehen Sie ein sog. Sequenzdiagramm, welches die Aufrufe zwischen verschiedenen Objekten darstellt. Dabei sind die verschiedenen beteiligten Objekte als vertikale Lebenslinien nebeneinander gekennzeichnet. Pfeile zwischen den Lebenslinien zeigen die verschiedenen Methodenaufrufe zwischen den einzelnen Klassen an – zu Beginn bspw. der Konstruktoraufruf für **AsciiBaum**. Die dicke blaue Linie auf einer Lebenslinie zeigt an, dass eine bestimmte Funktion ausgeführt wird. Wird diese beendet, dann geht man mit einem **return** zum Aufrufer zurück.

Bei reiner linearer Programmierung würden also alle weiteren Aufrufe direkt oder indirekt über die **Main** Methode passieren – wenn diese beendet wird, sorgt das ja auch für das Programmende. In Abbildung 155 sehen wir allerdings auch ein Kästchen mit der Überschrift **Deferred Call** eingezeichnet. Und innerhalb dieses ruft der Timer bei einem Tick über einen Delegate die Methode **WennTimertickt** der Klasse **Program** auf. Genau dieser Kasten stellt das Hollywood-Prinzip dar, da wir dieser Aufruf keinen linearen, von uns als Entwickler initiierten Aufruf darstellen, sondern eine andere Komponente diesen Aufruf unserer Methode für uns erledigt.

Bitte bedenken Sie auch, dass wir in Abbildung 154 nach dem Starten des Timers den Thread, auf dem wir arbeiten, mit `Console.ReadLine` blockieren. Dieser Thread läuft also genau dann erst weiter, sobald der Nutzer Enter drückt. Im Gegenzug heißt das, dass der Timer die Methode `WennTimerTickt` nicht auf diesem ursprünglichen Thread auslöst, sondern auf einem sog. Hintergrundthread. Was dies genau bedeutet, schauen wir uns im Detail in einem späteren Kapitel an. Sie sollten für jetzt nur mitnehmen, dass es bei Einsatz eines Timers zu Multithreading kommt.

Weiterhin sollten Sie aus diesem Beispiel mitnehmen, dass man mit Delegates Methoden wie Werte in Variablen, Parametern oder Feldern referenzieren und einsetzen kann – durch Delegates wird damit erst das Hollywoodprinzip möglich, denn nur so können beliebige Methoden, die eine gewisse Parameterliste und Rückgabetyp haben von einer bestimmten Komponenten aufgerufen werden, ohne dass diese eben genannte Methoden zur Entwicklungszeit kannte. Ob es dabei um statische Methoden wie in unserem Beispiel oder um Instanzmethoden handelt, ist völlig egal.

5.17.3.4 Die Vererbungshierarchie für Delegates

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=qXhwe1fG4Z8>.

Wir haben bereits in Abschnitt 5.17.3.1 gesehen, dass die Definition von Delegates relativ kurz im Vergleich bspw. zu Klassen- oder Interfacedefinitionen ist und wir dort auch keinen Einfluss auf Vererbung haben, wie das bei den letzteren beiden der Fall ist. Delegates leiten dennoch implizit von Basisklassen genauso wie Enumerationen ab – dies sind die abstrakten Klassen `MulticastDelegate` und `Delegate`. Im folgenden Diagramm sehen wir diese Klassen.

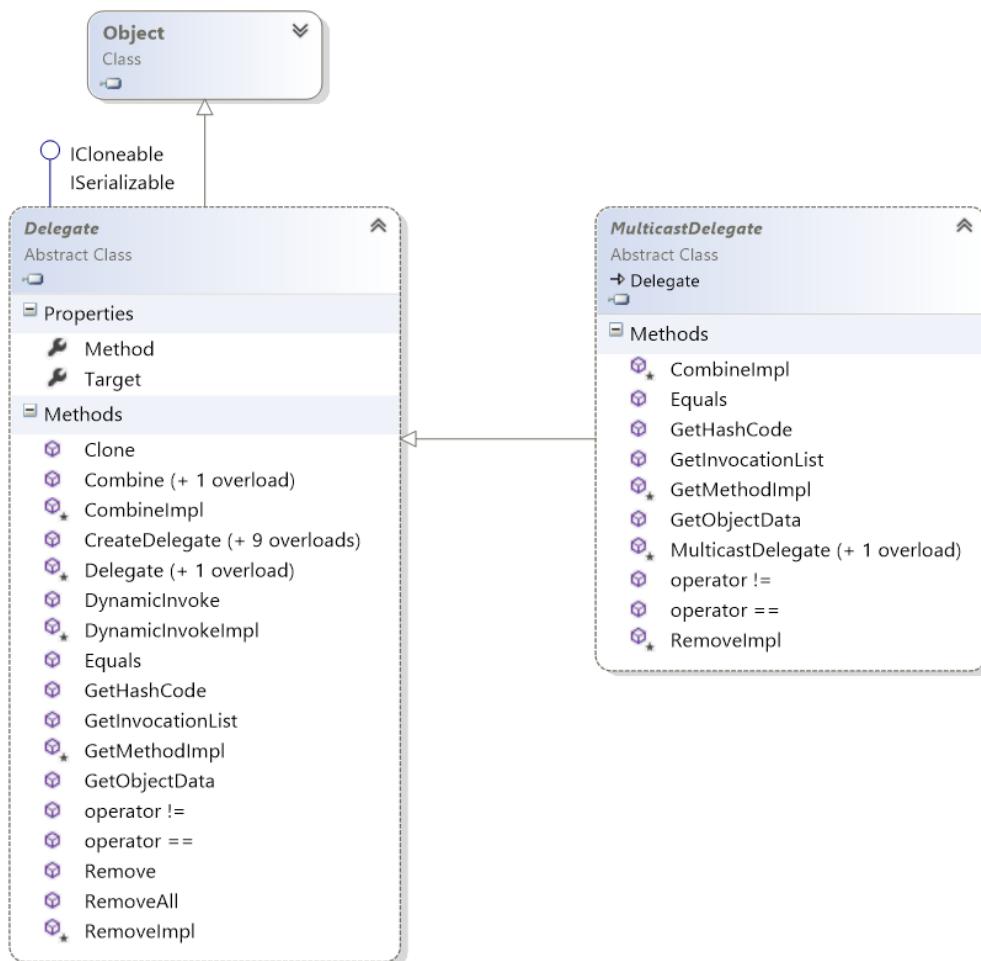


Abbildung 156: Die implizite Vererbungshierarchie von Delegates

Wie man in Abbildung 156 sehen kann, sind einige Members auf **Delegate** definiert, die teilweise in **MulticastDelegate** überschrieben werden. Die wichtigsten sehen wir uns in der folgenden Tabelle an:

Mitglied	Bemerkung
Combine	Mit dieser Methode lassen sich zwei oder mehr existierende Delegate-Objekte zu einem Delegate zusammenfassen, der alle referenzierten Methoden mit einem Aufruf auslöst (dies nennt man Multicast). Alternativ kann man hier auf den + = Operator verwenden.
DynamicInvoke	Mit DynamicInvoke lassen sich Delegate-Objekte über Reflection auslösen, sofern man diese über ihre Basisklasse Delegate oder MulticastDelegate anspricht und nicht über ihren eigentlichen Typ.
GetInvocationList	Mit dieser Methode kann man sich bei einem Multicast-Delegate die einzelnen Methoden holen, die von ihm referenziert werden.
operator == und !=	Delegates sind Value Objects, d.h. zwei Delegate-Objekte, welche auf dieselbe Methode zeigen, gelten als identisch.
Remove	Die Methode ist das Gegenteil von Combine: mit ihr kann man eine bereits existierende Methodenreferenz aus einem Multicast-Delegate entfernen.

Abbildung 157: Die wichtigsten Mitglieder der Basisklassen **Delegate** und **MulticastDelegate**

Wie man aus Abbildung 157 herauslesen kann, können Delegates nicht nur eine Methode referenzieren, sondern beliebig viele Methoden (die jeweils über dieselbe Parameterliste und Rückgabetypen verfügen müssen). Dabei baut man diese Multicast-Delegates (Multicast heißt auf Deutsch in etwa Mehrfachauslösen) sukzessive auf, indem man auf zwei (oder mehr) Delegates, die jeweils eine Methode referenzieren, die Methode Combine aufruft oder (und das ist einfacher) den **+ =** Operator einsetzt. Mit **- =** bzw. der Methode Remove kann man umgedreht einen Delegate aus einem Multicast-Delegate wieder entfernen. Sehen wir uns dazu folgendes Beispiel an:

```
public delegate void VerarbeiteZahlDelegate(int zahl);

public class Program
{
    static void Main()
    {
        var delegate1 = new VerarbeiteZahlDelegate(BildeQuadratUndGibAus);
        delegate1 += ZieheWurzelUndGibAus;
        delegate1(81); // Hier werden die beiden Methoden BildeQuadratundGibAus sowie ZieheWurzelUndGibAus aufgerufen (Multicast)
    }

    static void BildeQuadratUndGibAus(int zahl)
    {
        Console.WriteLine(Math.Pow(zahl, 2));
    }

    static void ZieheWurzelUndGibAus(int zahl)
    {
        Console.WriteLine(Math.Sqrt(zahl));
    }
}
```

Mit **+ =** werden zwei Delegates zu einem Multicast-Delegate vereinigt und wieder der Variablen **delegate1** zugewiesen

Abbildung 158: Multicast-Delegates aufrufen

In Abbildung 158 sehen Sie, dass die Variable **delegate1** mit einem **VerarbeiteZahlDelegate** instanziert wird, welcher auf die Methode **BildeQuadratUndGibAus** zeigt. Der wichtige Punkt folgt in der nächsten Anweisung, bei der mit **+ =** Operator der eben erstellte Delegate und ein neuer,

der auf ZieheWurzelUndGibAus zeigt, zu einem Multicast-Delegate kombiniert werden. Ist das getan, wird dieser über die gewohnte Syntax aufgerufen – und dabei werden natürlich die beiden referenzierten Methoden aufgerufen. Hiermit wird deutlich, dass ein Delegate nicht nur ein Zeiger auf eine Funktion ist, sondern auch ein Zeiger auf mehrere Methoden darstellen kann.

5.17.3.5 Die generischen Delegates Action und Func von .NET

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=isbh6zMzx0Q>.

Genauso wie bei Klassen und Interfaces können auch Delegates mit Generics ausgestattet werden. Das hat übrigens den praktischen Nebeneffekt, dass Sie nie wieder eigene Delegatetypen definieren müssen – sie können einfach die generischen Delegates **Action** und **Func** aus .NET wiederverwenden und die Generics nach genau den Typen auflösen, die Sie als Parameter- und Rückgabetypr brauchen. In der folgenden Abbildung sehen Sie, dass es die Delegates **Action** und **Func** nicht nur einmal gibt, sondern jeweils sechzehn Mal, allerdings jeweils mit einer unterschiedlichen Anzahl an Generics (man kann also zwei Typen mit dem gleichen Bezeichner ausstatten; solange beide Typen ein unterschiedliche Anzahl an Generics haben, gelten Sie nicht als gleich und der Compiler wirft keinen Fehler):

```
// Generische Action Delegates
// -----
public delegate void Action();
public delegate void Action<in T>(T obj);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
public delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
// usw. Action funktioniert mit bis zu 16 Parameter

// Generische Func Delegates
// -----
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3);
// usw. Func funktioniert ebenfalls mit bis zu 16 Parameter
```

Abbildung 159: Die generischen Delegates Action und Func des .NET Frameworks

Wie man in Abbildung 159 sehen kann, sind die beiden Delegatetypen wie folgt aufgeteilt:

- **Action** kann Methoden referenzieren, die als Rückgabetypr **void** haben und bis zu sechzehn Parametern haben.
- **Func** hingegen hat einen generischen Rückgabetypr und bis zu sechzehn Parameter.

Mit diesen Delegatetypen können Sie jede Methode referenzieren, die Sie schreiben werden (außer Sie haben mehr als 16 Parameter – wovon ich Ihnen ernsthaft abrate; sobald Sie mehr als drei haben, sollten Sie sich schon Gedanken machen, ob Sie Ihren Code nicht besser strukturieren können).

Ebenfalls sehen wir im obigen Beispiel die Schlüsselwörter `in` und `out` bei der Definition der Generics. Dies hat mit dem Thema Kontra- und Kovarianz bei Generics zu tun, über das Sie hier mehr lesen können: [http://msdn.microsoft.com/en-us/library/dd799517\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/dd799517(v=vs.110).aspx). Für unser Verständnis ist dieses Thema nicht weiter wichtig, weswegen ich hier auch nicht näher darauf eingehen werde.

Wie wir **Action** und **Func** sinnvoll einsetzen können, sehen wir uns im kommenden Abschnitt an, in dem wir uns mit Events beschäftigen.

5.17.3.6 Events

Wir haben in Abschnitt 5.1 bereits darüber gesprochen, dass eine Klasse vier verschiedene Typen an Mitgliedern haben kann, wobei wir bis jetzt nur Felder, Methoden und Eigenschaften besprochen haben. In den folgenden Abschnitten schauen wir uns Events genauer an.

5.17.3.6.1 Events definieren

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=Ka0nGCarOxM>.

Ein Event ist nichts anderes als ein spezielles Paar Methoden, ähnlich einer Eigenschaft. Die Methodenpaare heißen bei Events `add` und `remove` – das hat den Hintergrund, dass Events ausschließlich mit Delegates als Typen arbeiten und mit der `add` Methode ein Delegate zu einem Multicast-Delegate hinzugefügt wird, mit der `remove` Methode wird analog umgekehrt ein Delegate vom Multicast-Delegate entfernt. Sehen wir uns dazu gleich ein Beispiel an:

```
public class Metronom
{
    private Action _tick; ----- Privates Feld, das Delegate referenziert
    public event Action Tick ----- Event mit Add und Remove Methode.
    {
        add { _tick += value; } ----- Dabei werden eingehende Delegates mit dem
        remove { _tick -= value; } ----- im Feld _tick referenzierten Delegate zu einem
                                         Multicast-Delegate verarbeitet.
    }
    private bool _shouldStop;
    private int _beatsPerMinute = 120;

    public void Start()
    {
        _shouldStop = false;
        var task = new Task(ErzeugeTicks); ----- Mit Task können Methoden auf einem
        task.Start(); ----- Hintergrundthread ausgeführt werden.
    }

    private void ErzeugeTicks()
    {
        var intervalInMilliseconds = (60.0 / _beatsPerMinute) * 1000.0;
        var interval = TimeSpan.FromMilliseconds(intervalInMilliseconds);
        while (_shouldStop == false)
        {
            if (_tick != null)
                _tick();
            Thread.Sleep(interval); ----- Mit Thread.Sleep kann man einen Thread eine
                                         gewisse Zeit schlafen legen – erst danach wird die
                                         nächste Anweisung ausgeführt
        }
    }

    public void Stop()
    {
        _shouldStop = true;
    }

    public int BeatsPerMinute
    {
        get { return _beatsPerMinute; }
        set
        {
            if (value < 1)
                throw new ArgumentOutOfRangeException("value", "BPM muss größer als 0 sein");
            _beatsPerMinute = value;
        }
    }
}
```

Abbildung 160: Events in Klassen definieren

In Abbildung 160 sehen wir wieder mehrere neue Dinge in dort definierten Klasse `Metronom`, deren Aufgabe es ist, nach einer gewissen Zeit einen Event auszulösen, der `Tick` heißt. Sehen wir uns dazu gleich einmal an, wie man Events definiert:

- Zunächst gibt man Modifizierer an, wie das bei anderen Mitgliedern auch üblich ist.
- Danach folgt das Schlüsselwort `event`.
- Im Anschluss gibt man einen Delegatentypen an (andere Typen sind hier nicht erlaubt, auch nicht die Basisklassen `Delegate` und `MulticastDelegate`).
- Dann folgt der Bezeichner für den Event.
- Zuletzt gibt man in geschweiften Klammern die speziellen `add` und `remove` Methoden an. Diese haben ähnlich wie `get` und `set` nur einen sehr verkürzten Methodenkopf (kein Rückgabetyp, Parameterliste, Bezeichner und Modifizierer). Innerhalb dieser Methoden kann man jedoch genauso über das Schlüsselwort `value` auf den übergebenen Delegate zugreifen, den man mit den `+=` bzw. `-=` zu einem in einem Feld referenzierten Delegate hinzufügt bzw. entfernt. Im Beispiel oben wird dazu das Feld `_tick` benutzt. Der Typ von `value` ist dabei natürlich gleich mit dem nach `event` angegebenen Delegatentypen.

Natürlich kann man in den Methoden `add` und `remove` beliebige andere Anweisungen ausführen. Allerdings sollten Sie das nicht tun, da ein Nutzer einer Klasse einen Event nur nutzt, um Delegates zu registrieren (bzw. zu deregistrieren) und dadurch bestimmten Code bei auftretenden Ereignissen auszuführen (daher auch der Name Event).

Als zweite Neuerung sehen wir in der `Start`-Methode die Klasse `Task`. Diese bekommt beim Konstruktorauftruf ebenfalls einen Delegate und sorgt dafür, dass die im Delegate referenzierte Methode auf einem Hintergrundthread ausgeführt wird und nicht auf dem Hauptthread. Somit wird die private Methode `ErzeugeTicks` auch nicht auf demselben Thread aufgerufen, auf dem die Main Methode gestartet wurde. Dies wurde gemacht, damit wir den Thread mit der Methode `Thread.Sleep` schlafen lassen können, bis ein neuer Tick gesendet werden soll – die Abarbeitung des Hauptthreads wird dadurch nicht gestört. Wenn der Thread nicht gerade schläft, wird innerhalb der `while` Schleife der im Feld `_tick` referenzierte Multicast-Delegate ausgelöst (sofern dieser nicht `null` ist).

Die Kommunikation zwischen den beiden Threads geschieht dabei über das Feld `_shouldStop`. Wird dieses vom Hauptthread aus beim Aufruf der Methode `Stop` auf `true` gesetzt, bricht die `while` Schleife im Hintergrundthread ab und die Methode `ErzeugeTicks` wird verlassen – das sorgt dafür, dass auch der Hintergrundthread beendet wird.

Abschließend zu diesem Abschnitt möchte ich anfügen, dass Ihnen klar geworden sein soll, dass Events nichts anderes als ein Methodenpaar `add` und `remove` sind, mit denen man sich auf Multicast-Delegates registrieren bzw. deregistrieren kann. Ihre registrierten Methoden werden dann nach dem Hollywood-Prinzip aufgerufen, wenn ein gewisses Ereignis eintritt. Wie man sich auf Events registrieren kann, sehen wir uns im nächsten Abschnitt an.

5.17.3.6.2 Methoden an Events registrieren

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=04vwTs4a7xA>.

Wenn wir die in Abbildung 160 definierte Klasse `Metronom` und insbesondere ihr `Tick`-Event einsetzen möchten, könnten wir das machen, wie in Abbildung 161 gezeigt. Zunächst werden die Klassen `Metronom` und `Lauscher` instanziiert, wobei letztere nur eine Methode hat, in der auf der Konsole „`Ich hab's gehört`“ ausgegeben wird. Nach diesen beiden Anweisungen werden wie schon bei Delegates auch die Methoden `Console.Beep` und `Lauscher.HörZu` an den Event mit

dem `+=` Operator registriert. Bitte beachten Sie dabei, dass erstere eine statische Methode ist und letztere eine Instanzmethoden. Danach wird `Start` auf dem `metronom`-Objekt aufgerufen, was die im letzten Abschnitt beschriebene Funktionalität mit den Hintergrundthread auslöst. Nachdem dieser Hintergrundthread initialisiert wurde, kommt der `Start`-Aufruf direkt zurück und mit `Console.ReadLine` wird auf eine Eingabe des Nutzers gewartet – nach dieser wird das Metronom mit dem Aufruf von `Stop` abgebrochen.

Solange das Metronom gestartet ist, wird von der Konsole ein Beep und der oben genannte String bei jedem Tick ausgegeben. Dies funktioniert wiederrum nach dem Hollywood-Prinzip: das Metronom ruft über den gekapselten Multicast-Delegate die Methoden auf, die vorher registriert wurden. Auch in diesem Beispiel wird keine der Methoden `Console.Beep` und `Lauscher.HörZu` direkt aufgerufen.

```
public class Lauscher
{
    public void HörZu()
    {
        Console.WriteLine("Ich hab's gehört.");
    }
}

public class Program
{
    private static void Main()
    {
        var metronom = new Metronom();
        var lauscher = new Lauscher();

        metronom.Tick += Console.Beep;           Hier werden zwei
        metronom.Tick += lauscher.HörZu;         Methoden an einen
                                                Event registriert

        metronom.Start();

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

Abbildung 161: Methoden an Events registrieren

Mit der in Abbildung 161 dargestellten Funktionalität kann man sich natürlich fragen, ob Events überhaupt nötig sind, wenn man doch auch einfach einen Delegate über eine Eigenschaft oder gar ein Feld `public` anbieten könnte. Auch dann könnten sich Nutzer über den `+=` Operator registrieren und über den `-=` Operator deregistrieren. Der Unterschied ist aber, dass bei Events keine Zuweisung mit `null` erlaubt ist (Delegates sind immer Referenztypen). Somit würde die folgende Anweisung vom Compiler mit einem Fehler quittiert:

`metronom.Tick = null;`

Dieses Verhalten folgt den Richtlinien der Kapselung, denn ein Nutzer der Klasse kann so zu einem Event registrieren (bzw. deregistrieren), ohne dass er dabei Einfluss auf bereits registrierte Delegates nehmen kann.

5.17.3.6.3 Die Kurzschreibweise für Events

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=-cq9ieD9-KY>.

Genauso wie bei automatischen Eigenschaften gibt es auch für Events eine Kurzschreibweise, die Sie anwenden können, um automatisch vom Compiler ein passendes privates Feld, das einen Delegate referenziert, und die dazugehörigen Methoden `add` und `remove` erzeugen zu lassen. Auf das Feld können Sie dann wie bei automatischen Eigenschaften auch nur über den Eventbezeichner (nicht mehr wie vorher über das Feld) zugreifen. Dies ist die übliche Schreibweise für Events, da man sowieso in der `add` und `remove` Methode vermeiden sollte, etwas anderes zu machen als Events zu registrieren oder zu deregistrieren – deswegen sieht man die lange Schreibweise auch nahezu gar nicht mehr im Programmieralltag. Funktional gesehen bleibt natürlich alles gleich. Abbildung 162 zeigt Ihnen die beiden Varianten nebeneinander in einem Beispiel.

```

public class BeispielKlasse
{
    private Random _random = new Random();

    private Action<int> _event1;
    public event Action<int> Event1
    {
        add { _event1 += value; }
        remove { _event1 -= value; }
    }

    public void LöseEvent1Aus()
    {
        if (_event1 != null)
            _event1(_random.Next());
    }

    public event Action<int> Event2; ----- Kurzschreibweise für Events

    public void LöseEvent2Aus()
    {
        if (Event2 != null)
            Event2(_random.Next());
    }
}

```

Abbildung 162: Kurzschreibweise für Events

5.17.3.6.4 Wann und wie sollte man Events einsetzen?

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=cDdzB5X1wCA>.

Nachdem wir uns in den letzten Abschnitten ausgiebig mit dem Thema Events auseinandergesetzt haben, bleibt jetzt noch die Frage offen, wann wir diese denn sinnvoll einsetzen können. Die Antwort lautet auch hier: so wenig wie möglich, so oft wie nötig. Events drehen den Kontrollfluss um, was den Code grundsätzlich erst einmal schlechter verständlich macht. Einen großen Boost haben Events (bzw. eventähnliche Programmierweisen nach dem Hollywood-Prinzip) Anfang der Neunziger bekommen, als GUI Programmierung auf dem Vormarsch war. Sehr viele GUI-Frameworks haben bspw. einen Event Click auf der Button-Klasse, wobei dieser Event natürlich ausgelöst wird, wenn der Nutzer auf den Button klickt, und dabei wird ebenso natürlich der Code ausgelöst, der auf dem Click-Event registriert war.

Genau nach diesem Schema gibt es unzählige Events bei GUI-Frameworks, bspw. für Scrollen, Mausbewegungen, Tastendrücke usw. Immer wenn der Nutzer diese Events durch bestimmte Aktionen auslöst, kann der Programmierer mit einer gewissen Logik darauf reagieren, indem er sich auf diese Events mit seinen Methoden registriert. Genauso gut könnte man sich aber auch eine

Klasse vorstellen, die einen bestimmten Ordner im Dateisystem überwacht und einen Event auslöst, wenn in diesem Ordner bspw. eine neue Datei erstellt wird. Andere Klassen können sich dann an diesen Event anheften und bestimmte Logik ausführen, um bspw. die neue Datei in irgendeiner Form zu verarbeiten.

Hier wird ersichtlich: Events werden häufig an der Prozessgrenze eingesetzt, um bestimmte Signale an weitere Klassen zu geben, wenn ein bestimmter Zustand eintritt – bspw. wenn der Nutzer mit seiner Maus über einen Button fährt oder wenn ein neuer Wert eines Microcontrollers bereitsteht, der die Anzahl an verarbeiteten Flaschen in einer Abfüllanlage misst. Und genau an diesen Stellen können Events einen sinnvollen Zweck erfüllen. Setzen Sie bitte aber nicht Events um des Events Willen ein.

5.17.3.7 Anonyme Methoden und Lambdas

Delegates referenzieren Methoden und können diese auslösen. Bisher haben wir unsere Methoden immer als Klassenmitglieder direkt definiert, allerdings gibt es auch eine andere Möglichkeit, Methoden zu erstellen: direkt innerhalb einer anderen Methoden als anonyme Funktion. Wie das genau geht, schauen wir uns in den kommenden Abschnitten an.

5.17.3.7.1 Anonyme Methoden mit dem Schlüsselwort delegate

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=20zsXBFEm30>.

Sehen wir uns sofort ein Beispiel an, in dem wir eine anonyme Methode definieren, in einem Delegate referenzieren und sie über ihn ausführen.

```
private static void Main()
{
    List<Person> personen = new List<Person>
    {
        new Person { Vorname = "Walter", Nachname = "White" },
        new Person { Vorname = "Jesse", Nachname = "Pinkman" },
        new Person { Vorname = "Hank", Nachname = "Schrader" }
    }; delegate leitet anonyme Methode ein

    Action<List<Person>> anonymeMethode = delegate(List<Person> liste) - Parameter der anonymen Methode
    {
        foreach (var person in liste)
        {
            Console.WriteLine(person.VollerName);
        }
    };

    anonymeMethode(personen);
}
```

Abbildung 163: Anonyme Methode mit delegate Schlüsselwort

In Abbildung 163 sehen wir, dass in der `Main` Methode zunächst eine Liste mit Personenobjekten erstellt wird. Dann folgt der interessante Teil: innerhalb einer Anweisung mit dem Schlüsselwort `delegate` wird eine anonyme Methode definiert, die einem Delegate vom Typ `Action<List<Person>>` zugewiesen wird (man kann also Generics wiederrum mit einem generischen Typen ersetzen). Nach `delegate` wird die Parameterliste für die Methode angegeben – offensichtlich muss diese natürlich genau einen Parameter haben, dessen Typ `List<Person>` ist. Danach folgt der Scope der anonymen Methode, in dem wir wie gewohnt prozeduralen Code schreiben können. Im obigen Beispiel durchlaufen wir die als Parameter übergebene Liste mit einer `foreach` Schleife und geben den Namen der Personen aus.

Bis zu dieser Anweisung ist die anonyme Methode natürlich nur definiert, aber wurde noch nicht aufgerufen. Dies passiert mit der letzten Anweisung der Methode, bei der der Delegate

anonymeMethode aufgerufen wird. Diesem Aufruf wird die Liste personen übergeben, sodass sie innerhalb der anonymen Methode genutzt werden kann.

Wer jetzt ob dieser Möglichkeit von anonymen Methoden erschreckt ist und nicht weiß, wozu diese sinnvoll eingesetzt werden können, den bitte ich um Geduld. Im nächsten Abschnitt kommt es allerdings noch schlimmer, denn anonyme Methoden bieten im Vergleich zu normalen Methoden sog. Closure.

5.17.3.7.2 Closure bei anonymen Methoden

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=JTzPWAuEysQ>.

Das in Abbildung 163 gesehen Beispiel können wir so abändern, dass für die anonyme Methode kein Parameter notwendig ist:

```
private static void Main()
{
    List<Person> personen = new List<Person>
    {
        new Person { Vorname = "Walter", Nachname = "White" },
        new Person { Vorname = "Jesse", Nachname = "Pinkman" },
        new Person { Vorname = "Hank", Nachname = "Schrader" }
    };

    Action anonymeMethode = delegate
    {
        foreach (var person in personen)
        {
            Console.WriteLine(person.VollerName);
        }
    };
}

anonymeMethode();
```

Anonyme Methoden können auf die Variablen der Methode zugreifen, in der sie definiert sind (Closure)

Abbildung 164: Closure bei anonymen Methoden

Was Sie in Abbildung 164 sehen, bezeichnet man als sog. Closure (dt. Funktionsabschluss – wobei ich diese Übersetzung nicht als gut empfinde und deshalb beim englischen Fachbegriff bleibe). Wir sehen, dass die anonyme Methode auf die Variable personen der Methode Main einfach zugreifen kann, ohne dass diese wie im vorherigen Beispiel per Parameter übergeben werden muss. Beachten Sie, dass für den Delegatetypen nur noch `Action` und nicht mehr `Action<List<Person>>` verwendet wird, weswegen man nach dem Schlüsselwort `delegate` auch keine Parameterliste mehr angeben muss (tatsächlich wurden nicht mal leere runde Klammern nach `delegate` angegeben).

Wenn Closure eingesetzt wird, sorgt der Compiler automatisch dafür, dass die in der anonymen Methode eingesetzten Variablen der äußeren Methode ersterer übergeben werden – und zwar per Call-By-Reference, auch wenn es sich um Wertetypen handelt. Dies sehen wir an folgendem Beispiel:

```

private static void Main()
{
    int meineZahl = 42;
    Action meineMethode = delegate
    {
        meineZahl++;
        Console.WriteLine(meineZahl);
    };

    meineMethode();
    Console.WriteLine(meineZahl); ————— meineZahl hat bei diesen Aufruf den Wert 43 – durch Closure wurde also in der anonymen Methode der Wert verändert
}

```

Abbildung 165: Closure übergibt auch Wertetypen per Call-By-Reference

In Abbildung 165 wird eine `int` Variable mit dem Wert 42 initialisiert. Danach wird nach dem bekannten Muster eine anonyme Methode definiert, die `meineZahl` manipuliert. Wie wir an der letzten Ausgabe mit `Console.WriteLine` sehen, hat `meineZahl` nach dem Aufruf der anonymen Methode den Wert 43 – der anonyme Methode hat also Seiteneffekte auf die Variablen der Main Methode, obwohl es sich bei `int` sogar um einen Wertetypen handelt (der als Parameter immer via Call-By-Value übergeben wird).

5.17.3.7.3 Lambdas – die kürzere Schreibweise für anonyme Methoden

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=LHPWA7ZzS-0>.

Neben der Möglichkeit, anonyme Methoden mit dem Schlüsselwort `delegate` zu definieren, bietet C# auch sog. Lambdas an, um dies zu erledigen. Die Syntax für Lambdas ist noch kürzer als die für `delegate`, weswegen Lambdas auch standardmäßig für anonyme Methoden eingesetzt werden sollten.

Das aus Abbildung 163 bekannte Beispiel kann man mit Lambdas wie folgt schreiben:

```

private static void Main()
{
    List<Person> personen = new List<Person>
    {
        new Person { Vorname = "Walter", Nachname = "White" },
        new Person { Vorname = "Jesse", Nachname = "Pinkman" },
        new Person { Vorname = "Hank", Nachname = "Schrader" }
    };

    Action<List<Person>> anonymeMethode = liste =>
    {
        foreach (var person in liste)
        {
            Console.WriteLine(person.VollerName);
        }
    };
}

anonymeMethode(personen); ————— Anonyme Methoden können auch über Lambdas definiert werden
}

```

Abbildung 166: Anonyme Methode mit Lambda definieren

In Abbildung 166 sehen Sie, dass statt des Schlüsselworts `delegate` direkt der Parameter `liste` angegeben wird, allerdings ohne Typ. Danach folgt der für Lambdas typische Pfeil `=>`, den man als Geht-Über-Nach im Deutschen ausspricht (goes over to im Englischen). Danach folgen die üblichen

geschweiften Klammern für den Scope der Methode, in dem wir beliebig viele Anweisungen unterbringen können. Bitte beachten Sie, dass in diesem Beispiel keine Closure verwendet wird, obwohl diese natürlich genauso möglich ist wie in den vorherigen Beispielen.

Die Syntax für Lambdas sieht jetzt noch nicht so viel kürzer aus als für anonyme Methoden mit `delegate`. Das liegt aber auch daran, dass die anonyme Methode eine `foreach` Schleife und somit mehr als eine Anweisung enthält. Der große Vorteil von Lambdas ist, dass anonyme Methoden mit nur einer Anweisung ohne geschweifte Klammern auskommen. Folgendes Beispiel zeigt, welche Formen bei Lambdas möglich sind:

```
Komplette Lambdaform mit Parametern, explizitem Scope und return
Func<int, int, int> lambda1 = (int wert1, int wert2) => { return wert1 + wert2; };

Bei Lambdas können die Parametertypen weggelassen werden
Func<int, int, int> lambda2 = (wert1, wert2) => { return wert1 - wert2; };

Enthält ein Lambda nur eine Anweisung, können geschweifte Klammern und return weggelassen werden
Func<int, int, int> lambda3 = (wert1, wert2) => wert1 * wert2;

Bei nur einem Parameter können die Parameterklammern ebenfalls weggelassen werden
Func<int, int> lambda4 = wert => wert * wert;

Bei keinen Parametern müssen aber leere Klammern angegeben werden
Action lambda5 = () => Console.WriteLine("Hello World");
```

Abbildung 167: Die verschiedenen Arten zur Angabe von Lambdas

In Abbildung 167 sehen wir in der ersten Zuweisung für `lambda1`, wie die komplette Lambdaform aussieht: zuerst kommen die Parameter (`int wert1, int wert2`), welche wie auch bei normalen Methoden aussehen. Danach folgt der Pfeil `=>` sowie ein Methodenscope, in dem beliebig viele Anweisungen angegeben werden können. Wie wir an den Lambdas zwei bis fünf erkennen können, kann man diese komplette Form stark abkürzen, und zwar nach den folgenden Regeln:

- Parametertypen können grundsätzlich weggelassen werden wie bei `lambda2`. Sie werden dann implizit vom Compiler beim Erstellvorgang gesetzt.
- Wenn im Scope der anonymen Methode nur eine Anweisung steht, können die geschweiften Klammern, das Schlüsselwort `return` sowie das Semikolon für die Anweisung weggelassen werden. Bitte beachten Sie, dass das Semikolon bei `lambda3` nicht zur anonymen Methode gehört, sondern zum Abschluss der Zuweisung.
- Wenn ein Lambda nur einen Parameter besitzt, können die runden Parameterklammern weggelassen werden, wie bei `lambda4` ersichtlich ist. Die Klammern müssen jedoch angegeben werden, wenn ein Lambda keine Parameter entgegen nimmt (siehe `lambda5`).

Wie man sehen kann, können Lambdadefinitionen sehr viel kürzer sein als anonyme Methoden mit `delegate` oder normale Methoden.

5.17.3.7.4 LINQ als Beispiel für den Einsatz von Lambdas

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=Oi2PmptUcAY>.

Nachdem wir uns mit anonymen Methoden auseinandergesetzt haben und verstehen, wie sie funktionieren, haben wir immer noch kein sinnvolles Anwendungsgebiet für sie gesehen. Lambdas sind mit .NET 3.5 eingeführt worden und zwar hauptsächlich aus einem Grund: damit diese in Kombination mit der sog. Language Integrated Query (abgk. LINQ, auf Deutsch etwa

Programmierspracheninterne Abfragesprache) eingesetzt werden können. LINQ nutzt einerseits sog. Extension Methods, die das Interface `IEnumerable<T>` erweitern und Delegates, die bestimmte Funktionalität zur Überprüfung oder Transformation einzelner Elemente in einer Collection referenzieren. Am besten sieht man sich das jedoch an einem Beispiel an.

Nehmen wir an, wir hätten eine Liste, in denen unsortiert mehrere Mitarbeiter mit allen möglichen Mitarbeiterverhältnissen eingetragen sind. Wir möchten diese Liste so transformieren, dass nur Mitarbeiter von Management und Teamleitung enthalten sind, diese nach dem Mitarbeiterverhältnis gruppiert werden und innerhalb dieser Gruppen absteigend nach Gehalt sortiert. Damit wir das in prozeduralem Code schaffen, müssten wir in etwa folgende Methode implementieren:

```
public class MitarbeiterAufschlüsselung
{
    public void SchlüsseleManagementUndTeamleitungAuf(IReadOnlyList<Mitarbeiter> mitarbeiterListe)
    {
        if (mitarbeiterListe == null) throw new ArgumentNullException("mitarbeiterListe");

        var managementUndTeamleitung = new Dictionary<string, List<Mitarbeiter>>();
        foreach (var mitarbeiter in mitarbeiterListe)
        {
            var Mitarbeiterverhältnis = mitarbeiter.Mitarbeiterverhältnis;
            // Diese if Abfrage filtert alle Angestellten hinaus
            if (Mitarbeiterverhältnis == Mitarbeiter.Angestellter)
                continue;

            // Überprüft ob Liste in Dictionary schon erstellt wurde
            if (managementUndTeamleitung.ContainsKey(Mitarbeiterverhältnis) == false)
                managementUndTeamleitung.Add(Mitarbeiterverhältnis, new List<Mitarbeiter>());

            // Füge Mitarbeiter zur entsprechenden Liste in Dictionary ein
            managementUndTeamleitung[Mitarbeiterverhältnis].Add(mitarbeiter);
        }

        // Sortiere Mitarbeiter nach Name in allen Listen
        foreach (var liste in managementUndTeamleitung.Values)
        {
            liste.Sort(VergleicheMitarbeiterGehalt); —————— List<T>.Sort kann einen Delegate entgegennehmen, der für den Vergleich zweier Elemente aufgerufen wird.
        }

        // Gib Ergebnis auf Konsole aus
        foreach (var keyValuePaar in managementUndTeamleitung)
        {
            Console.WriteLine(keyValuePaar.Key);
            Console.WriteLine("-----");

            foreach (var mitarbeiter in keyValuePaar.Value)
            {
                Console.WriteLine(" {0} {1}", mitarbeiter.Name, mitarbeiter.AktuellesGehalt);
            }
            Console.WriteLine();
        }
    }

    private int VergleicheMitarbeiterGehalt(Mitarbeiter ersterMitarbeiter, Mitarbeiter zweiterMitarbeiter)
    {
        if (ersterMitarbeiter == null) throw new ArgumentNullException("ersterMitarbeiter");
        if (zweiterMitarbeiter == null) throw new ArgumentNullException("zweiterMitarbeiter");

        return (int) zweiterMitarbeiter.AktuellesGehalt - (int) ersterMitarbeiter.AktuellesGehalt;
    }
}
```

Bei absteigender Sortierung muss der Rückgabewert negativ sein, wenn das Gehalt des zweiten Mitarbeiters kleiner ist als das des Ersten.

Abbildung 168: Mitarbeiterabfrage nach Mitarbeiterverhältnis und Gehalt

In Abbildung 168 sehe Sie, dass dieser Vorgang in prozeduralem Code relativ komplex ist:

- Zunächst wird ein `Dictionary<string, List<Mitarbeiter>>` gebildet, mit dem die einzelnen Mitarbeiter zu ihren jeweiligen Gruppen (Management oder Teamleiter) zugeordnet werden sollen.
- Innerhalb der ersten `foreach` Schleife werden alle Angestellten gefiltert. Nur andere Mitarbeiter werden zur jeweiligen Liste im Dictionary zugeordnet. Diese Listen werden on-the-fly erstellt, wenn der jeweils erste Mitarbeiter in sie eingefügt werden soll.
- In der nächsten `foreach` Schleife werden alle Listen im Dictionary sortiert. Dazu wird die `List<T>.Sort` Methode eingesetzt, allerdings eine Überladung, die einen Delegate vom Typ `Comparison<T>` entgegennimmt. Dieser Delegate referenziert die Methode `VergleicheMitarbeiterGehalt`, die immer dann von der Sort-Methode aufgerufen wird, wenn beim Sortieren zwei Elemente miteinander verglichen werden müssen. Die absteigende Sortierung wird erreicht, indem in dieser Methode überprüft wird, ob das Gehalt des zweiten Mitarbeiters kleiner ist als das Gehalt des Ersten.
- Am Ende werden die Ergebnisse auf der Konsole ausgegeben.

LINQ kann solche Szenarien sehr vereinfachen. Im folgenden Beispiel sehen wir, wie knapp der Code mit LINQ wird:

```
public class MitarbeiterAufschlüsselung
{
    public void SchlüsseleManagementUndTeamleitungAuf(IReadOnlyList<Mitarbeiter> mitarbeiterListe)
    {
        if (mitarbeiterListe == null) throw new ArgumentNullException("mitarbeiterListe");

        var managementUndTeamleitung =
            mitarbeiterListe.Where(mitarbeiter => mitarbeiter.Mitarbeiterverhältnis != Mitarbeiter.Angestellter)
                .OrderByDescending(mitarbeiter => mitarbeiter.AktuellesGehalt)
                .GroupBy(mitarbeiter => mitarbeiter.Mitarbeiterverhältnis);

        // Gib Ergebnis auf Konsole aus
        foreach (var gruppe in managementUndTeamleitung)
        {
            Console.WriteLine(gruppe.Key);
            Console.WriteLine("-----");
            foreach (var mitarbeiter in gruppe)
            {
                Console.WriteLine(" {0} {1}", mitarbeiter.Name, mitarbeiter.AktuellesGehalt);
            }
            Console.WriteLine();
        }
    }
}
```

Mit den LINQ Methoden Where, OrderByDescending und GroupBy werden die gleichen Verarbeitungen gemacht wie beim prozeduralen Code vorher. Mit Lambdas werden einzelne Elemente gefiltert, eingesortiert und eingruppiert.

Abbildung 169: Mitarbeiterabfrage mit LINQ

In Abbildung 169 wird auf die entgegengenommene List drei LINQ-Methoden aufgerufen:

- Mit `Where` können einzelne Elemente aus Collections gefiltert werden. Dazu gibt man einen Delegaten an, der ein Element der Collection bekommt und mit seinem Rückgabewert vom Typ `bool` entscheiden muss, ob das Element herausgefiltert wird oder nicht.
- Mit `OrderByDescending` werden die Elemente absteigend nach einem bestimmten Schlüssel geordnet. Hier wird im Lambda das aktuelle Gehalt des Mitarbeiters angegeben.
- Mit `GroupBy` werden Elemente anhand eines bestimmten Schlüssels zu einer Gruppe zugeordnet. Im Lambda wurde dafür das `Mitarbeiterverhältnis` angegeben.

LINQ ist deutlich kürzer und ausdrucksstärker als die prozedurale Variante aus Abbildung 168. An diesem Beispiel können Sie erkennen, dass Lambdas vor allem bei Abfragen und Transformationen von Collections sehr gut eingesetzt werden können.

5.17.3.7.5 Wie LINQ intern funktioniert (Level 200)

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=K0frlJBq6DU>.

Das in Abbildung 169 gezeigte Beispiel setzt die LINQ-Methoden Where, OrderByDescending und GroupBy ein. LINQ besteht natürlich nicht nur aus diesen drei Methoden, sondern aus insgesamt über fünfzig, die für unterschiedlichste Verarbeitungsmethoden in Zusammenhang mit Collections eingesetzt werden können (darunter fallen bspw. Sortierung, Abbildungen, Gleichheitsüberprüfung und Aggregieren). Eine Übersicht über die Möglichkeiten von LINQ finden Sie in der MSDN Library unter <http://msdn.microsoft.com/en-us/library/bb397896.aspx>.

Interessant ist für uns jedoch der Fakt, wie wir auf einmal diese LINQ Methoden auf ein Objekt vom Typ `IReadOnlyList<Mitarbeiter>` aufrufen konnten. Immerhin sind die genannten Methoden nicht auf diesem Interface definiert, die Aufrufsyntax erinnert aber sehr an Instanzmethoden. Der Punkt ist, dass LINQ Methoden keine normalen Funktionen sind, sondern sog. Extension Methods.

Extension Methods sind statische Methoden in statischen Klassen, die allerdings auf bestimmten Objekten wie Instanzmethoden aufgerufen werden können. Sie kennzeichnen sich v.a. durch den ersten Parameter aus, der mit dem Schlüsselwort `this` gekennzeichnet ist und dem Objekt entspricht, auf dem die Methode aufgerufen wird.

Damit wir besser verstehen, wie LINQ funktioniert, implementieren wir die `Where` Methode einfach mal nach:

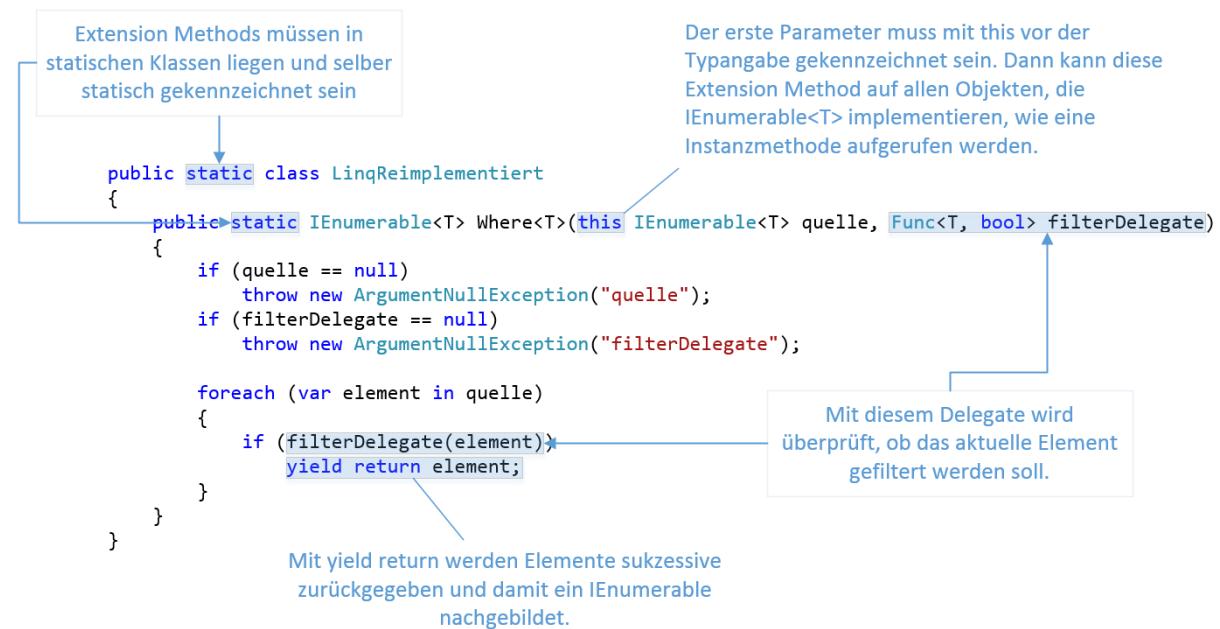


Abbildung 170: Die LINQ Methode Where reimplementiert

In Abbildung 170 sehen wir relativ viele neue Dinge, welche die `Where` Methode ausmachen und denen wir jetzt in den folgenden Einzelpunkten auf den Grund gehen:

- Der markanteste Punkt ist wohl, dass der erste Parameter vor der Typangabe mit dem Schlüsselwort `this` gekennzeichnet ist. Damit kann diese Methode auf allen Objekten, die `IEnumerable<T>` implementieren (und das sind ziemlich viele, man bedenke nur alle Collectionklassen), wie eine Instanzmethode aufgerufen werden. Die Objektreferenz wird in

diesem Fall dem ersten Parameter zugewiesen und muss auch nicht mehr beim Aufruf angegeben werden.

- Der zweite notwendige Punkt ist, das Extension Methods immer statisch sein müssen und immer in statischen Klassen liegen müssen.
- Innerhalb der `Where` Methode wird einfach jedes Element der Collection `quelle` mit einer `foreach` Schleife durchlaufen und in einem `if` Block mit dem Aufruf des Delegates `filterDelegate` überprüft, ob das aktuelle Element in die zurückgegebene Aufzählung (`IEnumerable<T>`) ist oder nicht.
- Der letzte interessante Punkt ist der Zusammenbau des Rückgabewerts: anstatt bspw. eine Liste zu instanziieren, ihr alle gefilterten Elemente hinzuzufügen und diese dann zurückzugeben, wird hier der Ausdruck `yield return element` eingesetzt. Mit `yield` (auf Deutsch in diesem Kontext etwa ‚hergeben, abtreten‘) werden sukzessive die nach `return` angegeben Elemente zurückgegeben. Wenn der `IEnumerable<T>` Rückgabewert mit `foreach` durchlaufen wird, wird beim zweiten `MoveNext` Aufruf die Methode `Where` nach der `yield return` Anweisung fortgesetzt und damit die `foreach` Schleife fortgesetzt. Auch wird die `Where` Methode erst tatsächlich durchlaufen, sobald das zurückgegebene Objekt mit einer `foreach` Schleife tatsächlich durchlaufen wird. Dieses Verhalten bezeichnet man als Deferred Execution (dt. verzögerte Ausführung) von Methoden. Weitere Informationen zu `yield` finden Sie in der MSDN Library unter folgendem Link: <http://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx>.

Obwohl also die `Where` Methode eigentlich nichts anderes macht, als die Elemente, die nicht herausgefiltert werden sollen, in eine neue Aufzählung zu packen, kommt es beim Einsatz unserer nachgebauten `Where` Methode erst zum Aufruf, wenn das zurückgegebene Objekt durchlaufen wird. Das folgende Beispiel illustriert dies:

```
public void GibManagementUndTeamleiterAus(IReadOnlyList<Mitarbeiter> mitarbeiterListe)
{
    var gefilterteMitarbeiter = mitarbeiterListe.Where(m => m.Mitarbeiterverhältnis != Mitarbeiter.Angestellter);

    foreach (var mitarbeiter in gefilterteMitarbeiter)
    {
        Console.WriteLine("{0} {1}", mitarbeiter.Name, mitarbeiter.AktuellesGehalt);
    }
}
```

Erst beim Durchlaufen des Rückgabewerts in der foreach Schleife wird
Where sukzessive aufgerufen, um die gefilterten Elemente
zurückzugeben

Where kann jetzt wie eine
Instanzmethode eingesetzt
werden, wird hier aber
nicht direkt ausgeführt
wegen der yield return
Angabe
(Deferred Execution)

Abbildung 171: Deferred Execution bei der nachgebildeten Methode `Where`

Nicht alle LINQ Methoden setzen Deferred Execution ein, aber ziemlich viele. Umgangssprachlich wird deshalb auch zwischen lazy methods (dt. faule Methoden, also mit Deferred Execution) und greedy methods (dt. gierige Methoden, die direkt ausgeführt werden) unterschieden. Die wichtigsten LINQ Methoden sind in der folgenden Tabelle aufgelistet:

LINQ Extension Method	Bemerkung
All	Überprüft, ob alle Elemente einer Collection eine bestimmte Bedingung erfüllt (greedy).
Any	Überprüft, ob wenigstens ein Element in einer Collection eine bestimmte Bedingung erfüllt (greedy).
Average	Bestimmt den Durchschnittswert eines numerischen Werts, der bei allen Elementen einer Collection vorkommt (greedy).
Contains	Überprüft, ob in einer Collection ein bestimmtes Element enthalten ist (greedy).
Count	Gibt die Anzahl der Elemente in einer Collection zurück (greedy).

<code>Distinct</code>	Filtert alle mehrfach vorkommenden Elemente in einer Collection, sodass eine Menge ohne Duplikate zurückgegeben wird (lazy).
<code>First</code>	Gibt das erste Element einer Collection zurück, dass eine gewisse Bedingung erfüllt (greedy). Kann dies nicht gefunden werden, wird eine Exception ausgelöst.
<code>FirstOrDefault</code>	Wie <code>First</code> , allerdings wird keine Exception ausgelöst, sondern der Standardwert für den entsprechenden Datentypen zurückgegeben.
<code>GroupBy</code>	Gruppert die Elemente einer Collection nach einem spezifizierten Schlüssel auf (lazy).
<code>Last</code>	Wie <code>First</code> , allerdings wird das letzte Element einer Collection, das eine bestimmte Bedingung erfüllt, zurückgegeben (greedy).
<code>LastOrDefault</code>	Wie <code>FirstOrDefault</code> , allerdings wird das letzte Element einer Collection, das eine bestimmte Bedingung erfüllt, zurückgegeben (greedy).
<code>Max</code>	Bestimmt das Maximum eines numerischen Werts, der bei allen Elementen einer Collection vorkommt (greedy).
<code>Min</code>	Bestimmt das Minimum eines numerischen Werts, der bei allen Elementen einer Collection vorkommt (greedy).
<code>OrderBy</code>	Sortiert die Elemente einer Collection nach einer bestimmten Vorgabe (lazy).
<code>OrderByDescending</code>	Wie <code>OrderBy</code> , allerdings in umgedrehter Reihenfolge (lazy).
<code>Reverse</code>	Kehrt die Reihenfolge der Elemente einer Collection um (lazy).
<code>Select</code>	Projiziert die Elemente einer Collection auf einen anderen Datentyp (lazy).
<code>Sum</code>	Bestimmt die Summe eines numerischen Werts, der bei allen Elementen einer Collection vorkommt (greedy).
<code>ToArray</code>	Kopiert die Elemente einer Collection in einen neuen Array und gibt diesen zurück (greedy).
<code>ToList</code>	Kopiert die Elemente einer Collection in ein neues <code>List<T></code> Objekt und gibt dieses zurück.
<code>Where</code>	Filtert alle Elemente einer Collection heraus, die eine gewisse Bedingung nicht erfüllen.

Abbildung 172: Einige wichtige LINQ Methoden

Hier sei nochmals erwähnt, dass sämtliche LINQ Extension Methods keine Seiteneffekte auf das `IEnumerable<T>` Objekt haben, auf dem sie aufgerufen werden, d.h. die Quellcollection wird nicht verändert. Stattdessen wird von vielen Methoden ein neues `IEnumerable<T>` Objekt zurückgegeben, in dem die Änderungen vorgenommen wurden. Dadurch entstehen folgende Vorteile:

- LINQ Methoden können relativ leicht hintereinander aufgerufen werden, wie in Abbildung 169 zu sehen ist. Dadurch kann man sehr knapp und elegant Abfragen und Transformationen für Collections ausdrücken.
- LINQ funktioniert mit allen Klassen, die `IEnumerable<T>` implementieren, also u. a. mit jeder Collection. Dadurch kann man LINQ in vielen Situationen einsetzen.

Wir haben hier nur einen kleinen Ausschnitt von LINQ beleuchtet. Es gibt noch viele weitere Features zu diesem Thema, die Sie bei Interesse gerne selber in der [MSDN Library](#) nachlesen können. Sie werden in zukünftigen Beispielen in diesem Script aber häufiger LINQ Methoden im Einsatz sehen.

5.17.3.8 Zusammenfassung für Delegates

Das Video für diesen Abschnitt finden Sie unter https://www.youtube.com/watch?v=T59uY68G_JE.

Wir haben in den letzten Abschnitten viele Punkte zum Thema Delegates angesehen, die wir uns in der folgenden Liste nochmals zusammenfassen wollen:

- Delegates referenzieren eine oder mehrere Methoden und lösen genau diese aus, wenn sie selbst aufgerufen werden. Delegates sind also multicastfähig.
- Delegates ermöglichen das Hollywood-Prinzip, d.h. wir rufen eigene Methoden nicht mehr direkt selbst auf, sondern diese werden über anderen Objekte (Komponenten) via Delegates aufgerufen. Die aufzurufenden Methoden werden dabei in Delegates gekapselt als Parameter an das Objekt übergeben.
- Klassen können auch Events definieren, welche nichts anderes sind als Multicast-Delegates. Allerdings kann man Events von außen nicht auf `null` setzen (damit wird die Kapselung sichergestellt). Methoden können via Delegates auf Events ausschließlich registriert oder deregistriert werden.
- Delegates haben eine wichtige Bedeutung für anonyme Methoden, da diese ausschließlich über Delegates referenziert werden können. Anonyme Methoden werden häufig dort eingesetzt, wo nur kurze Methoden notwendig sind (bspw. bei LINQ). Im Gegensatz zu normalen Methoden können wir bei ihnen Closure einsetzen.

5.17.4 Strukturen

In den kommenden Abschnitten sehen wir uns als letzten C# Typen Strukturen an. Diese sind relativ ähnlich zu Klassen, haben aber doch einige essentielle Unterschiede, die es zu beachten gilt.

5.17.4.1 Definition von Strukturen

Das Video für diesen Abschnitt finden Sie unter <https://www.youtube.com/watch?v=Svg6Z676oUI>.

Strukturen werden sehr ähnlich zu Klassen definiert und können auch die gleichen Mitglieder enthalten, also Felder, Eigenschaften, Methoden und Events. Sehen wir uns dazu ein Beispiel an:

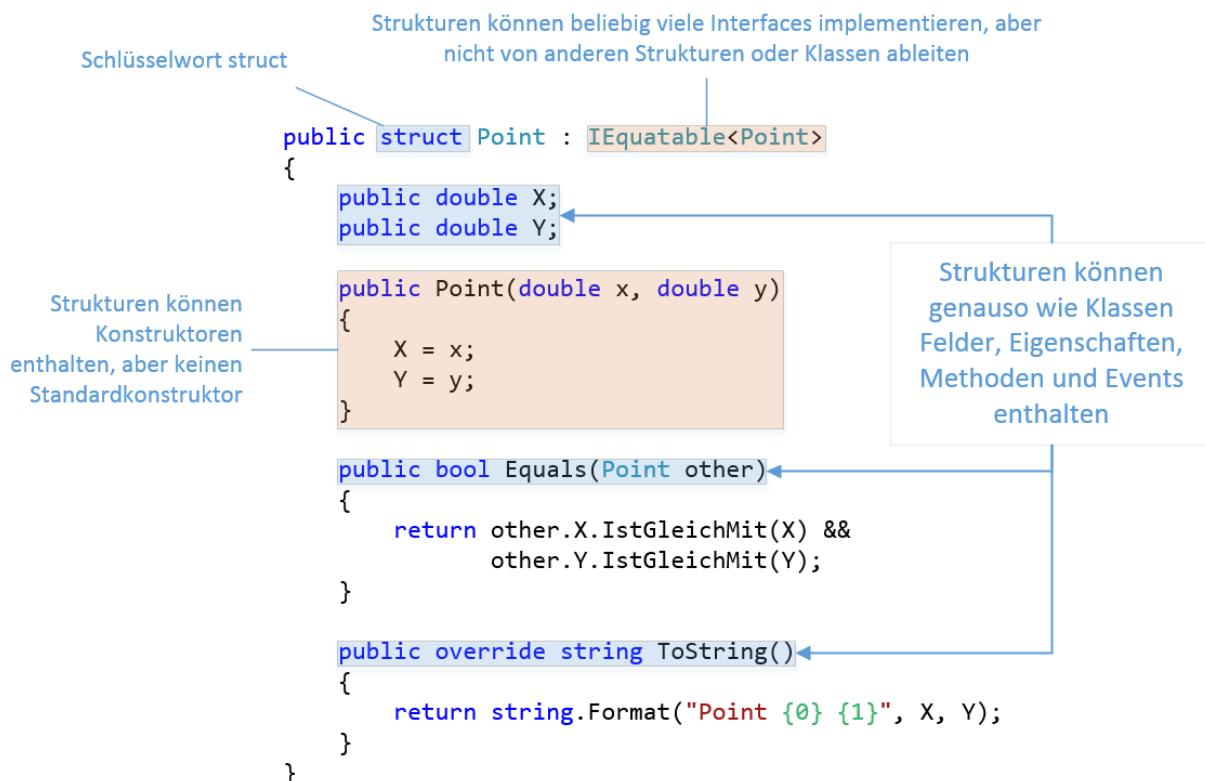


Abbildung 173: Strukturen definieren

In Abbildung 173 sehen Sie die Struktur Point, die zwei Felder X und Y definiert und damit die Koordinaten eines Punkts repräsentiert. Die erste Auffälligkeit ist die Angabe von `struct` statt `class` nach dem Zugriffsmodifizierer. Wie wir sehen, können wir in Strukturen genauso wie in Klassen Methoden, Felder, Eigenschaften und Events festlegen, wobei die letzten beiden im obigen Beispiel nicht zu sehen sind. Unterschiede erkennen wir jedoch schon bei den rot markierten Kästchen: offensichtlich sind Vererbung und Konstruktoren bei Strukturen im Vergleich zu Klassen unterschiedlich gelöst. Welche Unterschiede genau zwischen Klassen und Strukturen bestehen, schauen wir uns im nächsten Abschnitt an.

5.17.4.2 Unterschiede zwischen Strukturen und Klassen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=DplKPFpWFaw>.

In der folgenden Liste sind die wichtigen Unterschiede zwischen Strukturen und Klassen aufgeführt:

- Strukturen unterstützen Vererbung nur sehr eingeschränkt, und zwar in der Hinsicht, dass wie bei Klassen beliebig viele Interfaces implementiert werden können. Ableiten von einer anderen Klasse oder einer anderen Struktur ist jedoch nicht erlaubt. Genauso darf eine Klasse von keiner Struktur erben. Strukturen leiten implizit direkt von `ValueType` ab. Diese abstrakte Klasse haben wir uns bereits in Abbildung 146 angeschaut.
- Strukturen sind immer Wertetypen mit allen Auswirkungen (im Gegensatz zu Klassen, die immer Referenztypen sind; mehr zu diesem Thema besprechen wir in einem späteren Kapitel):
 - Werte von Strukturen liegen direkt in Variablen, Feldern oder Parameter, sie werden bei Zuweisungen per Call-By-Value übergeben. Das steht im Gegensatz zu Objekten von Klassen, die immer auf dem Heap liegen.
 - Deswegen kann Variablen, Parameter und Feldern, die Strukturtypen verwenden, auch nicht `null` zugewiesen werden (`null` bedeutet ja, das eine Referenz nicht auf ein Objekt zeigt).
 - Strukturwerte können zwar als `object` oder über ein Interface, das sie implementieren, angesprochen werden, allerdings tritt dann Boxing / Unboxing auf.
- Strukturen unterstützen keinen Standardkonstruktor. Wenn ein Konstruktor existiert, so muss dieser Parameter besitzen. Es wird auch kein Standardkonstruktor implizit vom Compiler erzeugt. Deshalb ist es auch nicht möglich, Felder bei der Definition inline mit einem Wert zu initialisieren.
- Innerhalb eines Konstruktors müssen alle Felder einer Struktur initialisiert werden.
- Strukturen unterstützen keine Destruktoren.
- Wenn ein Feld vom Typ einer Struktur und mit `readonly` gekennzeichnet ist, dann kann man Eigenschaften oder Felder des Strukturwerts nicht außerhalb des Konstruktors ändern.

5.17.4.3 Strukturen einsetzen

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=nCPMyvkzPck>.

Nicht nur bei der Definition, auch beim Einsatz von Strukturen gibt es einige Dinge zu beachten, wie das folgende Beispiel zeigt:

```

private static void Main()
{
    Point point1; — Strukturen müssen nicht mit new initialisiert werden
    point1.X = 4.2;
    point1.Y = 13.0;

    Console.WriteLine(point1);

    Point point2 = new Point(4.2, 13.0);
    Console.WriteLine(point1.Equals(point2));

    Point point3 = new Point();
    Console.WriteLine(point3);

    Point point4;
    point4.X = 3.5;
    Console.WriteLine(point4);
}

```

Auch wenn hier ein Konstruktoraufruf genutzt wird, landet der resultierende Wert nicht auf dem Heap

Dieser Parameter wird via Call-By-Value übergeben

Hier wird kein Standardkonstruktor aufgerufen, sondern alle Felder der Struktur mit dem jeweiligen Standardwert initialisiert

Hier tritt ein Compilerfehler auf, da erst alle Felder einer Struktur initialisiert werden müssen, bevor sie anderweitig eingesetzt werden kann

Abbildung 174: Die Struktur Point einsetzen

In Abbildung 174 sehen wir einige interessante Sachen, die ebenfalls im Unterschied zu Objekten bzw. Klassen stehen:

- Wie bei point1 und point4 zu sehen ist, müssen Strukturen nicht mit dem new Operator initialisiert werden. Was aber auf jeden Fall passieren muss, ist die Initialisierung aller Felder einer Struktur, bevor diese anderweitig eingesetzt werden kann. Bei point4 wurde das nicht gemacht (Y fehlt), weswegen es bei `Console.WriteLine(point4)` zu einem Fehler kommt.
- Man hat drei Möglichkeiten, um alle Felder einer Struktur zu initialisieren:
 - Man kann sie direkt im Code setzen wie bei point1
 - Man kann einen Konstruktor mit Parametern aufrufen wie bei point2 (auch innerhalb eines Konstruktors müssen alle Felder gesetzt werden)
 - Man kann alle Felder mit ihren jeweiligen Standardwerten initialisieren, wenn man wie im obigen Fall `new Point()` bei point3 aufruft (das ist jedoch kein Standardkonstruktor, sondern eine sog. Standardstrukturinitialisierung)
- Beim Aufruf von `point1.Equals(point2)` wird point2 via Call-By-Value übergeben (d.h. der Wert wird zum Parameter kopiert)

Nachdem wir uns diese Besonderheiten zu Strukturen angesehen haben, bleibt natürlich noch die Frage offen, wann und wie wir Strukturen sinnvoll einsetzen. Das wird im nächsten Abschnitt geklärt.

5.17.4.4 Wann sollte man Strukturen einsetzen?

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=DoWQmrkyYil>.

Da Strukturen zwar Polymorphie über Interfaces ermöglichen, aber dann das Boxing/Unboxing Problem auftritt, sollte man standardmäßig eher zu Klassen statt Strukturen greifen. Strukturen können jedoch sinnvoll eingesetzt werden, wenn...

- ...Sie einen Typ definieren möchten, der standardmäßig Call-By-Value übergeben wird.
- ...Sie einen Typ definieren möchten, der sich verhält wie ein primitiver Datentypen, bspw. `bool` oder `int`.
- ...Sie eine große Anzahl von Werten brauchen, bspw. wenn Sie Grafikprogrammierung machen und dann 3D-Objekte aus mehreren Tausenden Polygonen erstellen (Strukturen können hier einen Performancevorteil gegenüber Klassen bieten).
- ...Sie einen Wert häufig erstellen und dann nur kurz einsetzen (bspw. in einer Schleife). Wertetypen bedeuten für den Garbage Collector keine zusätzliche Arbeit.

Vorsicht: Strukturen können die Performance im Vergleich zu Objekten auch beeinträchtigen, nämlich genau dann, wenn Sie sehr groß werden (was meistens gleichbedeutend ist mit einer großen Anzahl von Feldern). Bedenken Sie, dass Wertetypen standardmäßig per Call-By-Value übergeben werden und bei großen Strukturen dauert der dazugehörige Kopiervorgang eben deutlich länger. Microsoft gibt als Richtwert mit, dass Strukturen nicht größer als 32 Byte (256 Bit) sein sollten.

5.18 Die Common Language Runtime

Wir haben bereits in einigen der vorherigen Abschnitte angesprochen, dass C# Code nicht direkt in Maschinencode übersetzt wird, sondern in eine Zwischensprache, die von der sog. Common Language Runtime (CLR) ausgeführt wird. Mit dieser werden wir uns in den kommenden Abschnitten näher beschäftigen, insbesondere mit dem Speichermanagement (engl. Memory Management), das als Besonderheit den sog. Garbage Collector aufweist, der automatisch nicht mehr genutzte Objekte aus dem Speicher entfernt.

Die CLR bietet dabei folgende Eigenschaften:

- Just-In-Time Compilation (JIT)
- Typsicherheit zur Laufzeit
- Automatisches Speichermanagement
- Erweitertes Exception Handling

Mit dem Thema Exceptions haben wir uns bereits ausführlich im Abschnitt 0 auseinandergesetzt. Zur Typsicherheit lässt sich so viel sagen, dass die CLR auch bei dynamischen Casts zur Laufzeit (nicht nur bei statischen Casts, die der Compiler überprüfen kann) dafür sorgt, dass ein Objekt nicht als ein Typ, mit dem das Objekt nicht in einer Vererbungslinie steht, interpretiert wird. Was wir uns in den kommenden Abschnitten noch im Detail anschauen, sind der Just-In-Time Compiler und die Speicherverwaltung der CLR.

5.18.1 Just-In-Time Compilation der CLR

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=ObIPolRhOoQ>.

Wie bereits erwähnt wird C# Code nicht direkt in Maschinencode kompiliert und gelinkt, sondern in eine Zwischensprache, die sog. Microsoft Intermediate Language (MSIL). Dieser Prozess wird in der nächsten Abbildung nochmals verdeutlicht.

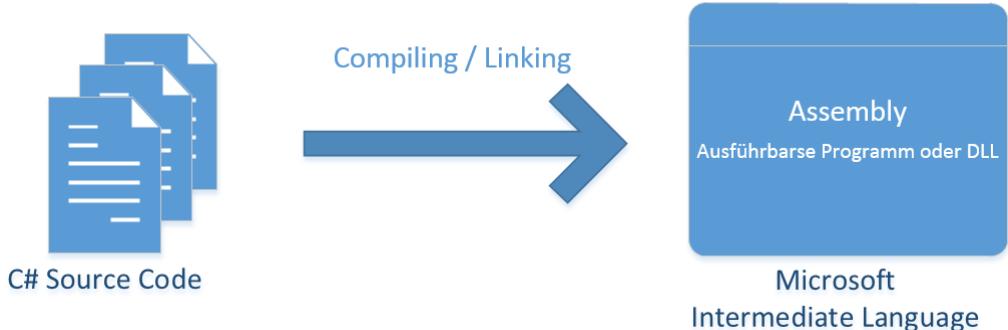


Abbildung 175: C# Code wird zu MSIL kompiliert

Dieser MSIL Code sieht ähnlich aus wie Assembler, allerdings kann man ihn meines Erachtens deutlich besser lesen, da Methodenaufrufe nicht durch Jump-Befehle an bestimmte Adressen dargestellt werden, sondern als tatsächliche Aufrufe ähnlich wie in C#.

Mit ILDasm kann MSIL Code betrachtet werden. Dieses Tool wird automatisch mit Visual Studio installiert und ist standardmäßig im Ordner C:\Program Files (x86)\Microsoft SDKs\Windows\v8.1A\bin\NETFX 4.5.1 Tools zu finden. Eventuell kann bei Ihnen v8.1A anders lauten, wenn Sie über ein anderes Betriebssystem als Windows 8.1 verfügen.

Wir möchten auch noch kurz einen Blick in MSIL werfen. Dazu setzen wir folgendes kurze Stückchen Code ein:

```

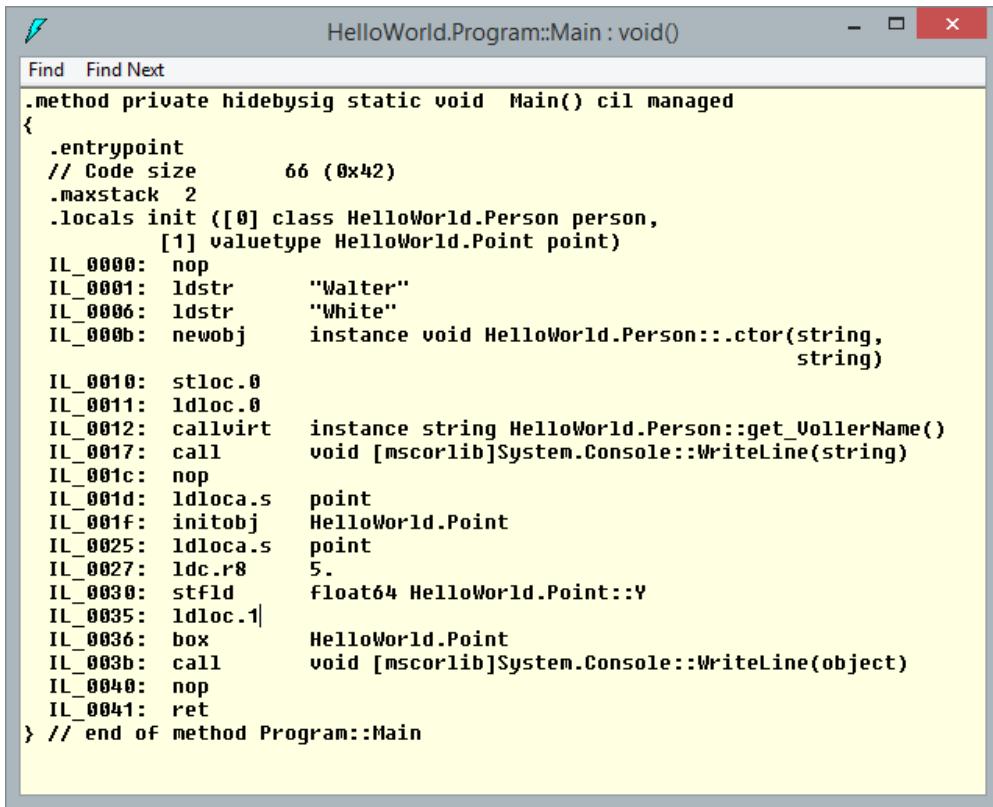
static void Main()
{
    var person = new Person("Walter", "White");
    Console.WriteLine(person.VollerName);

    var point = new Point();
    point.Y = 5.0;
    Console.WriteLine(point);
}

```

Abbildung 176: Beispielcode für MSIL

Wird der Code aus Abbildung 176 übersetzt und mit ILDasm geöffnet, sehen wir folgenden MSIL:



```

HelloWorld.Program::Main : void()
Find Find Next
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size       66 (0x42)
    .maxstack 2
    .locals init ([0] class HelloWorld.Person person,
                 [1] valuetype HelloWorld.Point point)
    IL_0000: nop
    IL_0001: ldstr     "Walter"
    IL_0006: ldstr     "White"
    IL_000b: newobj    instance void HelloWorld.Person::`ctor(string,
                                                               string)
    IL_0010: stloc.0
    IL_0011: ldloc.0
    IL_0012: callvirt   instance string HelloWorld.Person::get_VollerName()
    IL_0017: call      void [mscorlib]System.Console::WriteLine(string)
    IL_001c: nop
    IL_001d: ldloca.s  point
    IL_001f: initobj   HelloWorld.Point
    IL_0025: ldloca.s  point
    IL_0027: ldc.r8    5.
    IL_0030: stfld     float64 HelloWorld.Point::Y
    IL_0035: ldloc.1
    IL_0036: box       HelloWorld.Point
    IL_003b: call      void [mscorlib]System.Console::WriteLine(object)
    IL_0040: nop
    IL_0041: ret
} // end of method Program::Main

```

Abbildung 177: MSIL Code

Wie wir in Abbildung 177 sehen, kann man grob die Struktur des C# Code wiedererkennen. Das liegt daran, dass auch MSIL objektorientiert ist (im Gegensatz zu Assembler). Wie ebenfalls bereits erwähnt, kann man Methodenaufrufe sehr gut nachvollziehen. Insbesondere sehen wir hier, dass der Ausdruck `new Person("Walter", "White")` in einen Konstruktorauftrag mündet, `new Point()` aber nicht, sondern nur mit der Anweisung `initobj` in MSIL quittiert wird. Weiterhin interessant ist das Boxing des Point-Werts, wenn `Console.WriteLine` aufgerufen wird.

Näher möchte ich auch nicht auf MSIL eingehen, da man als .NET Entwickler eher selten auf diese Ebene heruntergeht. Interessant ist aber zu wissen, dass andere .NET Sprachen wie Visual Basic oder F# ebenfalls in MSIL Code übersetzt und letztendlich von derselben Runtime ausgeführt werden.

Wenn ein .NET Programm gestartet wird, wird auch automatisch die CLR hochgefahren. Der JIT-Compiler der CLR nimmt dann Schritt den MSIL-Code und übersetzt diesen in

Maschinencode, der konform zur Plattform ist, auf der die CLR läuft. Dieser Maschinencode wird auch direkt ausgeführt. Das prinzipielle Konzept dahinter ist, dass der Softwareentwickler C# Code nur einmal kompilieren muss und dann auf beliebigen Architekturen und Betriebssystemen (bspw. Windows, Linux, Android oder iOS) laufen lassen kann (Compile Once – Run Everywhere). Wichtig ist ausschließlich, dass die jeweilige Plattform über eine Runtime besitzt. Für Windows ist das eben die besagte CLR, für Linux und Mac OSX gibt es die Mono Runtime, für Android und iOS heißt die Runtime Xamarin. Die folgende Abbildung verdeutlicht dieses Konzept auch nochmals visuell.

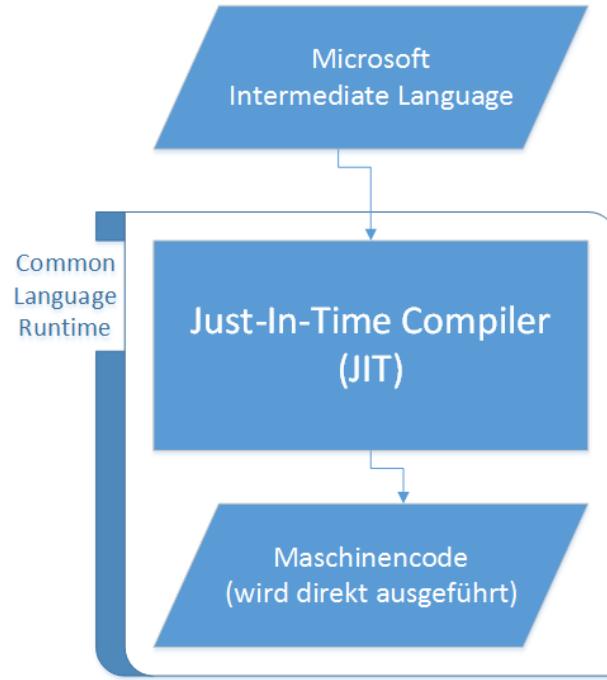


Abbildung 178: JIT-Compilation in der CLR

5.18.2 Speichermanagement in der CLR

Auch zum Speichermanagement haben wir im Verlauf dieses Scripts schon einige Worte verloren. Insbesondere haben wir auf die logische Einteilung des (Arbeits-)Speichers in Stack (dt. Stapel) und Heap (dt. Haufen, teilweise auch Free Store genannt) hingewiesen. Weiterhin haben wir eine Unterscheidung getroffen zwischen den Wertetypen, zu denen Strukturen und Enumerationen gehören, sowie den Referenztypen, zu denen Klassen, Interfaces und Delegates zählen. Wenn letztere mit dem `new` Operator instanziert werden, werden Objekte erstellt und wir wissen ebenfalls, dass Objekte immer auf dem Heap liegen. Bitte beachten Sie, dass Methoden in einem komplett anderen Speicherbereich liegen als Stack und Heap. Der Gesamtspeicher ist also genauso in Code- und Datenteil aufgeteilt wie bei der Ausführung von Maschinencode.

Zu guter Letzt erinnern wir uns auch an den Fakt, dass die CLR über einen Garbage Collector verfügt, der automatisch nicht mehr genutzte Objekte aus dem Heap entfernt. All diese Punkte werden wir uns in den kommenden Abschnitten im Detail ansehen und anhand von Code betrachten, wie sich Stack und Heap in der CLR beim Ausführen von Anweisungen verhalten.

5.18.2.1 Wofür ist der Stack zuständig, wofür der Heap?

Zunächst wollen wir uns genauer anschauen, welche Laufzeitdaten vom Stack verwaltet werden. Wie wir bereits in Abschnitt 5.15.2 anhand des Beispiels `Stack<T>` gesehen haben, ist ein Stack eine LIFO-Datenstruktur (bitte beachten Sie, dass das Thema Datenstrukturen nichts mit dem Thema `struct` zu tun hat), die nur zwei Methoden definiert, eine zum Einfügen eines Elements ans Ende, eine zum Herausnehmen des letzten Elements der Collection.

Nach genau diesem Prinzip verhält sich auch der Stack, der zur Speicherverwaltung in der CLR eingesetzt wird. Auf dem Stack werden folgende Dinge verwaltet:

- Methodeninformationen (insbesondere die Rücksprungadresse zum Aufrufer einer Methode)
- Variablen
- Parameter

Dabei baut sich der Stack sukzessive auf und ab, je nachdem ob eine Methode aufgerufen wird oder endet. Wie das genau funktioniert, sehen wir in einem kommenden Beispiel.

Durch dieses sukzessive Auf- und Abbauen ist der Stack sehr leicht zu implementieren, denn wenn eine Methode endet, kann sämtlicher Speicher, der Informationen, Variablen und Parameter zu dieser Methode enthält, deallokiert werden. Wird diese Methode erneut aufgerufen, werden diese Infos einfach wieder auf den Stack gepusht.

Der Heap verwaltet dagegen ausschließlich dynamisch erzeugte Objekte (Referenztypen). Wenn bspw. `new` auf eine Klasse aufgerufen wird, wird anhand der Klassenstruktur bestimmt, wieviel Speicher auf dem Heap allokiert wird und danach der Konstruktor der Klasse aufgerufen, um (hauptsächlich) die Felder mit bestimmten Werten zu initialisieren und weitere Initialisierungsfunktionalität auszuführen. Der `new` Aufruf entspricht dabei in etwa der Methode `malloc` in C, mit der ebenfalls dynamischer Speicher allokiert werden kann.

Weiterhin muss man sich als Programmierer nicht darum kümmern, diesen dynamischen allokierten Speicher auf dem Heap wieder freizugeben, wie das mit der Methode `free` notwendig ist. Stattdessen stellt die CLR den bereits erwähnten Garbage Collector bereit, über den wir uns genauer in einem späteren Abschnitt beschäftigen.

Zunächst möchten wir uns aber anschauen, wie sich Stack und Heap zur Laufzeit verhalten.

5.18.2.2 Laufzeitverhalten von Stack und Heap visualisieren

Zu diesem Abschnitt gibt es mehrere Videos:

- <https://www.youtube.com/watch?v=eEFBjpsKIZM>
- <https://www.youtube.com/watch?v=WS5MAQfE508>
- <https://www.youtube.com/watch?v=FurWNpGywUY>

Im kommenden Beispiel werden wir uns anschauen, wie sich Stack und Heap verhalten, wenn man Code Schritt für Schritt durchläuft. Das gesamte Beispiel können Sie dabei im Video für diesen Abschnitt nachvollziehen, in dem auch Boxing und Unboxing demonstriert wird. Hier im Script sehen wir uns nur grob an, wie Stack und Heap visuell aufgebaut sind.

5.18.2.2.1 Grundsätzlicher Aufbau der Visualisierung

Alle Speicherabschnitte sind dabei in Rechtecken gezeichnet, die wie folgt aufgebaut sind:

- Methodeninformationen enthalten hauptsächlich die Infos zu Rücksprungadresse und Rückgabewert. Da uns diese nur marginal interessieren, werden sie zu einem Punkt zusammengefasst. Rechtecke für Methodeninformationen sind blau.
- Variablen und Parameter werden über ihren jeweiligen Namen, ihren Typ und den Wert, den sie aktuell halten, angegeben. Dabei wird im Rechteck folgendes Schema angegeben:
variablenName (Typname): variablenWert
Nutzt die Variable einen Referenztyp, wird bei *variablenWert* ref eingetragen und ein Pfeil zum Objekt im Heap gezogen, auf das die Variable oder Parameter verweist. Wenn eine solche Variable auf kein Objekt verweist, wird einfach null eingetragen. Bei Variablen mit

Wertetyp wird der Wert der Variable direkt eingetragen. Rechtecke für Variablen sind violett, die für Parameter orange.

- Bei Objekten wird im Rechteck der Typ angegeben und alle Felder (Methoden liegen, wie bereits erwähnt, nicht direkt im Stack oder Heap). Für Felder gilt dabei dieselbe Syntax wie für Variablen und Parameter. Bitte beachten Sie, dass Objekte einen zusätzlichen Overhead haben (bspw. die vTable für Methodenaufrufe mit später Bindung), die wir in unserem Beispiel nicht betrachten.

Im folgenden Beispiel wird gezeigt, wie Stack und Heap aussehen, wenn man einige Zeilen Code ausführt:

```
private static void Main()
{
    byte rotWert = 255;
    byte andererFarbwert = 0;
    var neueFarbe = ErstelleNeueFarbe(rotWert, andererFarbwert, andererFarbwert);

    var neueAdresse = new Adresse("Universitätsstr. 31",
                                   "93053",
                                   "Regensburg");
    Console.WriteLine(neueFarbe);
    Console.WriteLine(neueAdresse);
}

private static object ErstelleNeueFarbe(byte rot, byte grün, byte blau)
{
    var neueFarbe = new Farbe(rot, grün, blau);
    return neueFarbe;
}
```

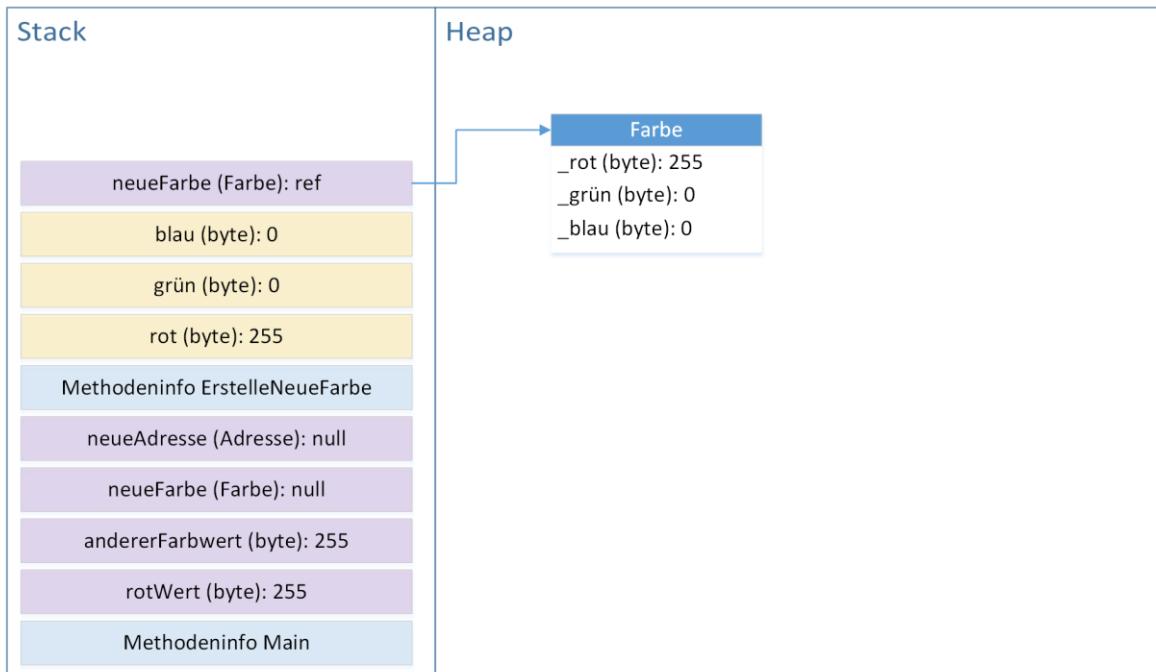



Abbildung 179: Beispiel für Darstellung von Stack und Heap an einer bestimmten Codestelle

5.18.2.2.2 Objektreferenzen innerhalb des Heaps

Wenn die Felder einer Klasse nur Wertetypen nutzen, wie das bei der Klasse **Farbe** bspw. der Fall ist, dann werden die Werte für diese Felder natürlich direkt im Objekt gespeichert. Genau das ist in Abbildung 179 zu sehen.

Felder können aber auch Referenztypen verwenden. Diese Felder sind dann natürlich nichts anderes als Referenzen zu anderen Objekten auf dem Heap. Wenn wir uns bspw. an die Klasse **Lottozahlengenerator** aus Abbildung 52 zurückerinnern, hat diese ein Feld vom Referenztyp **Random** genutzt. Wie das Verhältnis der beiden Objekte zur Laufzeit aussieht, illustriert folgende Abbildung:



Abbildung 180: Objekte können auf andere Objekte im Heap über Felder mit Referenztypen verweisen

Eine besondere Stellung nehmen hier Collections und Arrays ein: da diese alle Referenztypen sind, leben sie alle auf dem Heap. Des Weiteren haben sie ja die Eigenschaft, beliebige viele andere Elemente zu referenzieren. Wenn es sich bei diesen Elementen um Instanzen von Referenztypen handelt, referenziert eine Collection nichts anderes als Objekte auf dem Heap:

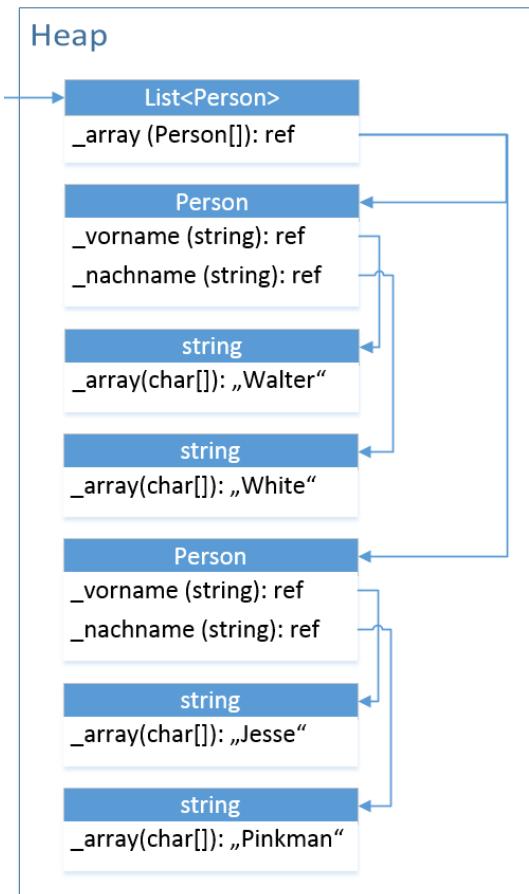


Abbildung 181: Collections können beliebig viele andere Objekte auf dem Heap referenzieren

5.18.2.2.3 Statische Felder und Delegates im Speicher

Statische Felder einer Klasse oder Struktur haben die Besonderheit, dass sie immer auf dem Heap liegen (auch wenn sie innerhalb einer Struktur definiert sind). In unseren Visualisierungen des Heaps werden diese aber weggelassen, da statische Felder nur einmal existieren (sie werden ja nicht pro Instanz erstellt). Weiterhin werden sie auch nicht vom Garbage Collector angetastet – dazu mehr im folgenden Abschnitt.

Wie wir auch bereits in einem vorherigen Abschnitt erwähnt haben, sind Delegates ebenfalls Referenztypen und landen damit auf dem Heap. Sie referenzieren aber nicht andere Objekte, sondern Methoden im Codeteil der Applikation (dies geschieht zwar auch nicht direkt, da sie intern **Type**-Objekte nutzen, um auf die entsprechende Methode zu verweisen, diese internen Details von .NET sollen für uns aber keine Rolle spielen).

5.18.2.3 Garbage Collection in der CLR

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=IdKP5f8D1zE>.

Nachdem wir uns im letzten Videobeispiel ausführlich mit Stack und Heap auseinandergesetzt haben und feststellten, dass der Stackspeicher automatisch auf- und abgebaut wird, sehen wir uns in diesem Abschnitt an, wie nicht mehr verwendete Objekte auf dem Heap entfernt werden. Dazu wird der Garbage Collector (GC) eingesetzt, dessen Vorgehensweise wir uns jetzt genauer betrachten. Eine der wichtigsten Punkte, die man beim GC beachten muss, ist folgender:

Der Garbage Collector wird von der CLR zu nicht-deterministischen Zeitpunkten gestartet. Dabei werden alle anderen Threads angehalten, der GC wird auf einen eigens für ihn reservierten Thread ausgeführt. Da man nie genau weiß, wie lange der GC für die Bereinigung von Objekten auf dem Heap braucht, ist die CLR auch nicht echtzeitfähig.

Der GC geht dabei in drei Schritten vor, um Objekte aus dem Heap zu entfernen:

- Mark-Phase (dt. Markierungsphase): in dieser Phase bestimmt der GC die Objekte, die aus dem Heap entfernt werden sollen. Dazu werden sog. GC Roots (dt. GC Wurzel) gebildet, die als Einstiegspunkt für den GC gelte und über die sämtliche Referenzen, die zwischen den Objekten im Heap vorhanden sind, vom GC abgewandert werden. Jedes Objekt, das der GC so erreichen kann, wird markiert. GC Roots können dabei folgende Dinge sein:
 - Variablen und Parameter auf dem Stack, die Objekte im Heap referenzieren
 - Statische Felder in Klassen und Strukturen, die Objekte auf dem Heap referenzieren
 - Objekte, die an eine native COM+ Library übergeben wurden (nicht wichtig für uns)
 - Objekte, die einen Destruktor (Finalizer) und entsorgt werden können, werden als spezieller GC Root markiert, da diese nicht bei diesem, sondern erst beim nächsten Durchlauf des GC tatsächlich deallokiert werden können.
- Sweep-Phase (dt. Löschphase): in dieser Phase werden vom GC alle Objekte, die in der Mark-Phase nicht markiert wurden, vom Heap entfernt (genauer gesagt: der dazugehörige Speicher deallokiert). Hierzu gibt es jedoch eine Ausnahme: Objekte, welche einen Destruktor besitzen, werden nicht sofort deallokiert, sondern erst im nächsten GC-Durchlauf. Im jetzigen Durchlauf wird lediglich der Destruktor aufgerufen.
- Compact-Phase (dt. Verdichtungsphase): da beim Deallokalieren Speicherlöcher entstanden sein könnten, werden alle noch existierenden Objekte auf dem Heap zusammengeschoben. Danach ist der erste freie Speicherplatz wieder hinter dem letzten Objekt im Heap zu finden. Ausgenommen von dieser Regel sind sog. Pinned Objekts: das sind Objekte, deren Speicheradresse nicht verändert werden darf (bspw. weil diese Objekte für Interop-Szenarien mit nativen Code eingesetzt werden; genauer hierzu ist für uns jedoch irrelevant).

Gerade der letzte Punkt ist extrem wichtig: `new` Operationen für Referenztypen können sehr schnell durchgeführt werden, da die nächste freie Stelle im Heap bereits bekannt. Was tatsächlich die CLR langsamer macht als nativer Maschinencode, sind die Garbage-Collector-Läufe, die je nach Anzahl der Objekte im Heap länger oder kürzer dauern. Und während der GC läuft, steht der Rest der Applikation, wie oben erwähnt.

Lösen Sie niemals in ihren Applikationen den Garbage Collector manuell mit `System.GC.Collect` aus. Der GC ist ein hochoptimiertes System, dass sie durch ihre manuellen Aufrufe in ca. 99 von 100 Fällen langsamer machen würden. Sie können die `Collect` Methode jedoch zu Testzwecken einsetzen oder genau dann, wenn Sie wirklich wissen, was Sie tun.

5.18.2.4 Unterschiede zwischen Beispiel und Wirklichkeit

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ZkbbeaoQfHE>.

In unseren Visualisierungen des Speichers haben wir genau einen Stack und einen Heap betrachtet – das ist allerdings eine vereinfachende Annahme. In Wirklichkeit sieht das ganze wie folgt aus:

- Jeder Thread hat seinen eigenen Stack. Obwohl wir uns noch nicht mit den Details von Threading beschäftigt haben, wissen wir, dass in Applikationen heutzutage gerne Threading

eingesetzt wird, um bestimmte Codeteile parallel auszuführen und damit die Performance und die Responsiveness zu erhöhen. Die Beispiele, die wir in den vorherigen Abschnitten betrachtet haben, sind also alle Single-Threaded.

- Der Heap hingegen existiert nur einmal (es gibt also keine eigene Instanz pro Thread), allerdings wird dieser Heap in der CLR in drei Teile aufgeteilt:
 - Generation 0: Dieser Teil des Heaps enthält nur Objekte, die kurzlebig sind. Jedes Objekt, das neu instanziert wird, wird in Generation 0 gesteckt. Der GC läuft in dieser Generation am häufigsten.
 - Generation 1: Dieser Teil enthält Objekte, die einige GC-Läufe überstanden haben. In diesem Falle werden sie von Generation 0 in Generation 1 erhoben. Der GC läuft auf diese Generation deutlich seltener als auf Generation 0.
 - Generation 2: In dieser Generation sind nur sehr langlebige Objekte enthalten, die viele GC-Läufe überstanden haben. Der GC läuft hier am seltensten.

Damit geht der Garbage Collector folgenden zwei Prinzipien nach:

1. Neue erstellte Objekte tendieren dazu, nur kurzlebig zu sein.
2. Je älter ein Objekt wird, desto wahrscheinlicher ist es, dass es auch weiterhin direkt oder indirekt via einem GC Root referenziert wird.

Wann genau der Garbage Collector läuft und welche Generationen er dabei anvisiert, ist jedoch ein Implementierungsdetail und nicht öffentlich bekannt. Dies hängt auch sehr von der Speichernutzung des Codes, der gerade ausgeführt wird, ab.

5.19 Persistenz mit .NET

Als Persistenz bezeichnet man das Speichern und Laden von Daten, die auf einem nicht-flüchtigem Medium (bspw. einer Festplatte) festgehalten werden. Der Arbeitsspeicher, in dem standardmäßig unsere Daten in Form von Werten und Objekten abgelegt sind, gilt als flüchtiger Speicher, da diese Daten beim Ende eines Programms vom Betriebssystem verworfen und der dazugehörige Speicher für andere Prozesse bereitgestellt wird.

Die üblichsten Datenspeicherformen sind das Dateisystem und Datenbanken. Wir werden uns beide Arten anschauen, wobei Sie in der Vorlesung Datenbanken diese noch viel genauer unter die Lupe nehmen werden.

5.19.1 Dateizugriffe

5.19.1.1 Überblick über Speichern und Laden

Das Video hierzu finden Sie unter https://www.youtube.com/watch?v=UD_nZquZZhg.

Zunächst möchten wir uns grundsätzlich verdeutlichen, wie eine Applikation zur Laufzeit auf eine Datei zugreifen und in sie Daten schreiben bzw. aus ihr Daten lesen kann. Dabei müssen folgende Dinge beachtet werden, die auch in Abbildung 182 gezeigt werden:

- Eine Applikation erhält unter Windows nicht direkt Zugriff auf die Festplatte, sondern bekommt vom Betriebssystem ein sog. Dateihandle zugewiesen. Dieses Dateihandle bekommt man als .NET Entwickler selbst nie zu Gesicht, es wird von der Klasse `FileStream` gekapselt (bei anderen Betriebssystemen ist das Vorgehen ähnlich, allerdings benutzt man dort nicht den Begriff Handle).
- Mit diesem Handle wird intern auf weitere betriebssystemeigene Funktionen zum Lesen oder Schreiben zugegriffen sowie festgelegt, ob bspw. andere Prozesse ebenfalls auf dieselbe Datei schreibend oder lesend zugreifen dürfen.
- Sobald ein Prozess keinen Zugriff mehr auf eine Datei benötigt, sollte er so schnell wie möglich das dazugehörige Handle freigeben, um einerseits Systemressourcen zu schonen und andererseits andere Prozesse, die gegebenenfalls auf diese Datei zugreifen möchten, nicht zu blockieren. Dies wird in .NET über das Interface `IDisposable` gemacht, dass wir schon in einigen der vorherigen Abschnitte zu Gesicht bekommen haben. Die `Dispose` Methode sollte man also so früh wie möglich aufrufen, sobald das Dateihandle nicht mehr benötigt wird (ansonsten würde dies erst beim nächsten GC Lauf gemacht).

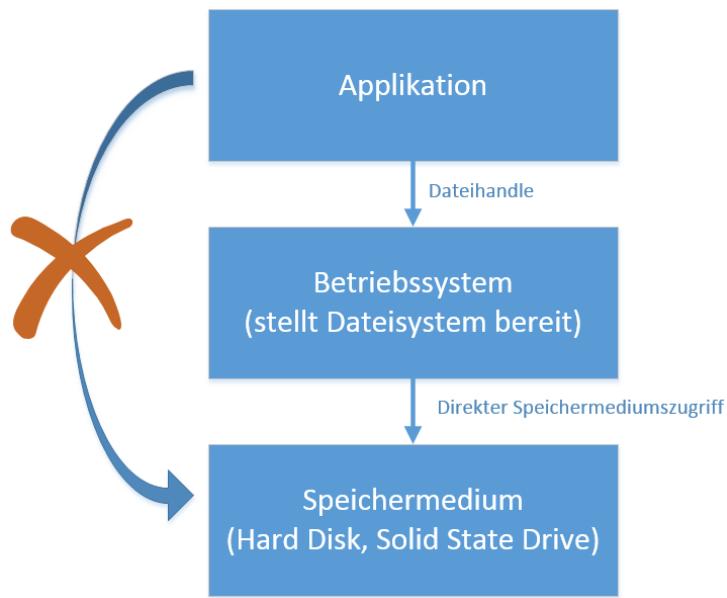


Abbildung 182: Auf Dateien greift man über das Betriebssystem zu

Neben dem Zugriff auf Speichermedien über das Betriebssystem ist es weiterhin auch ausschlaggebend, wie Daten überhaupt in Dateien abgelegt werden können. Grundsätzlich werden in Dateien Bytes gespeichert, d.h. man kann bspw. einen Byte-Array direkt an die entsprechenden Klassen des .NET Frameworks übergeben, damit dieser in die Datei geschrieben wird, wie in der folgenden Abbildung zu sehen ist:

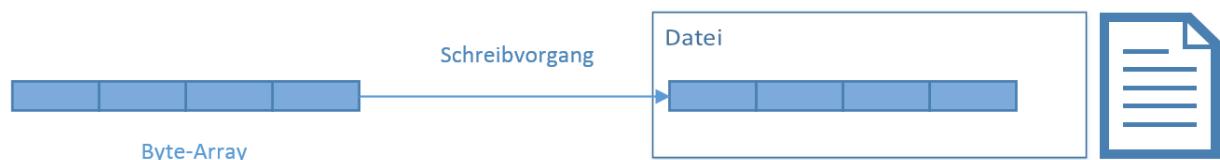


Abbildung 183: Byte Arrays können direkt in Dateien geschrieben werden

Des Weiteren nutzt man häufig die Möglichkeit, Text in Dateien zu speichern. Auch hier werden grundsätzlich erst einmal Bytes in einer Datei abgelegt. Damit Text (also effektiv ein String) zu Bytes umgeformt werden kann, wird ein gewisses Encoding zur Transformation eingesetzt, wie man in folgender Abbildung sehen kann:

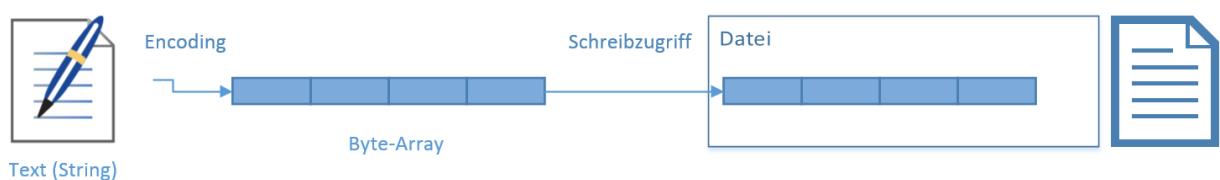


Abbildung 184: Text wird beim Speichern zunächst in ein bestimmtes Byte-Format konvertiert

Auch wenn in einer Textdatei also nichts anderes landet als in einer Binärdatei, wird dieser Unterschied doch häufig zu Rate gezogen, da Textdateien menschenlesbar sind, Binärdateien jedoch eher nicht (es gibt natürlich trotzdem die Möglichkeit, eine Binärdatei bspw. per Hex-Editor zu betrachten). Die gängigsten Encodings sind:

- Unicode Transformation Formats (UTF 16, UTF 8, UTF 32, UTF 7)
- ASCII

Üblicherweise benutzt man die Unicode Formate, wobei man hier natürlich auf die Länge einzelner Zeichen beim jeweiligen Format achten muss. Alle oben genannten UTF Formate unterstützen variable Länge bei der Darstellung von Buchstaben, bspw. kann UTF 16 einen Buchstaben mit einer oder zwei 16 Bit Folgen darstellen, je nachdem, welcher Buchstabe gewählt wurde (dies wurde gemacht, um einerseits eine große Anzahl an Buchstaben zu unterstützen und andererseits den Speicherverbrauch gering zu halten). Um die Implementierungsdetails für die Encodings brauchen wir uns allerdings selbst nicht zu kümmern, denn .NET bringt dafür schon die folgenden Klassen mit:

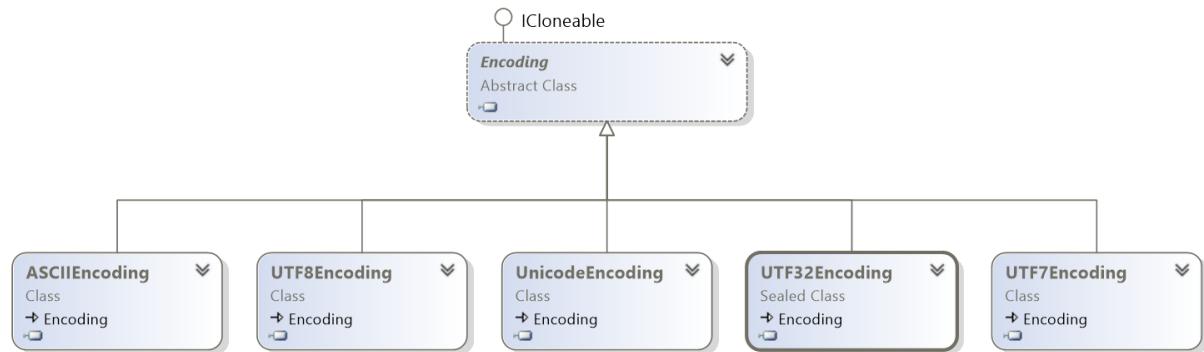


Abbildung 185: Die Encoding-Klassen in .NET

Wie in Abbildung 185 zu sehen ist, leiten alle Klassen von `System.Text.Encoding` ab, welche Sie auch als Abstraktion für Encodings einsetzen können. Das `UnicodeEncoding` steht dabei stellvertretend für UTF 16. Diese Klassen nutzt man hauptsächlich, um Bytes zu Strings oder analog umgekehrt Strings zu Bytes umzuformen. Üblicherweise macht man das allerdings nicht selbst, sondern übergibt ein Encoding an andere Klassen von .NET, die für einen die Arbeit erledigen.

Wir haben uns bis jetzt gedanklich nur mit dem Speichern in Dateien befasst. Beim Laden werden die oben beschriebenen Schritte natürlich umgekehrt vorgenommen: Binärdateien kann man also prinzipiell in einen Byte-Array laden, bei Textdateien muss man die Bytes der Datei erst noch mit einem Encoding interpretieren, um sie wieder in einen String umzuformen (C# bzw. die CLR nutzt übrigens intern UTF 16 für den Typ `string`). Im folgenden Abschnitt schauen wir uns genauer an, welche Klassen wir einsetzen können, um Binär- oder Textdateien mit .NET zu schreiben oder zu lesen.

5.19.1.2 Einfacher Zugriff auf Binär- und Textdateien

5.19.1.2.1 Die Klassen File und FileInfo

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=da8in34l-Fk>.

Für den einfachen Zugriff auf Binär- und Textdateien sind die Klassen `System.IO.File` und `System.IO.FileInfo` gedacht. Beide stellen in etwa dieselbe API bereit, allerdings enthält `File` nur statische Mitglieder, `FileInfo` hingegen muss erst instanziert werden, da diese Klasse hauptsächlich Instanzmethoden bereitstellt.

Hier im Script werden deshalb auch nicht beide APIs komplett aufgelistet, da viele Methoden auf beiden Typen identisch definiert sind. Folgende Tabelle enthält wichtige Methoden der Klasse `File`:

Methode	Beschreibung
Create	Erstellt eine Datei oder überschreibt sie
Delete	Löscht eine Datei
Move	Verschiebt eine angegebene Datei in einen neuen Ordner

Open, OpenRead, OpenWrite	Öffnet eine Datei für Lese- und Schreiboperationen mit einem FileStream
OpenText	Öffnet eine Textdatei für Leseoperationen mit einem StreamReader
ReadAllBytes	Öffnet eine Datei, liest alle Bytes in ein Array und schließt die Datei im Anschluss
ReadAllText	Öffnet eine Textdatei, liest den Inhalt in einen String und schließt die Datei im Anschluss
ReadAllLines	Öffnet eine Textdatei, liest alle Zeilen in ein String-Array und schließt die Datei
AppendAllText	Fügt den übergebenen Text ans Ende einer existierenden Datei an und schließt diese Datei im Anschluss
WriteAllBytes	Erstellt oder überschreibt eine Datei mit den übergegebenen Bytes
WriteAllText	Erstellt oder überschreibt eine Datei mit dem übergebenen String

Abbildung 186: Wichtige Methoden der Klasse **File**

Zusätzlich zu den in Abbildung 186 gezeigten Methoden bietet die Klasse **FileInfo** auch noch folgende Eigenschaften an:

Eigenschaft	Beschreibung
Directory	Liefert das DirectoryInfo -Objekt zum Ordner, in dem die Datei liegt
CreationTime	Liefert oder setzt das Erstellungsdatum der Datei
Extension	Liefert die Dateierweiterung einschließlich des Punktes mit
Name	Liefert den Dateinamen inklusive der Dateierweiterung
FullName	Liefert einen string mit der vollständigen Pfadangabe
Length	Gibt die Länge der Datei in Bytes zurück

Abbildung 187: Wichtige Eigenschaften der Klasse **FileInfo**

Genauere Infos zu diesen beiden Klassen finden Sie in der MSDN Library:

- **FileInfo**: <http://msdn.microsoft.com/en-us/library/system.io.fileinfo.aspx>
- **File**: [http://msdn.microsoft.com/en-us/library/system.io.file\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/system.io.file(v=vs.110).aspx)

5.19.1.2.2 Zugriff auf Ordner im Dateisystem

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=CiQdLsPgyz8>.

Analog zu **FileInfo** und **File** gibt es auch die Klassen **DirectoryInfo** und **Directory**, welche die folgenden Methoden bereitstellen:

Methode	Beschreibung
CreateDirectory	Erzeugt einen (Unter-)Ordner
Delete	Löscht einen Ordner
Exists	Prüft, ob der angegebene Ordner existiert
GetDirectories	Liefert die Unterordner des angegebenen Ordners
GetFiles	Liefert alle Dateien des angegebenen Ordners
GetFileSystemEntries	Liefert alle Unterordner und Dateien im angegebenen Ordner
GetParent	Liefert den übergeordneten Ordner
Move	Verschiebt einen Ordner samt Dateien und Unterordnern zu einem neuen Speicherort

Abbildung 188: Wichtige Methoden der Klasse **Directory**

Wenn wir uns Abbildung 188 genauer ansehen, sehen wir, dass wir mit der Methode **GetFileSystemEntries** alle Informationen zu allen Subordnern und Dateien bekommen. Dies ist möglich, da **FileInfo** und **DirectoryInfo** eine gemeinsame Basisklasse haben, über die sie

angesprochen werden können. Diese Basisklasse `FileSystemInfo` und ihre Hierarchie sehen Sie hier:

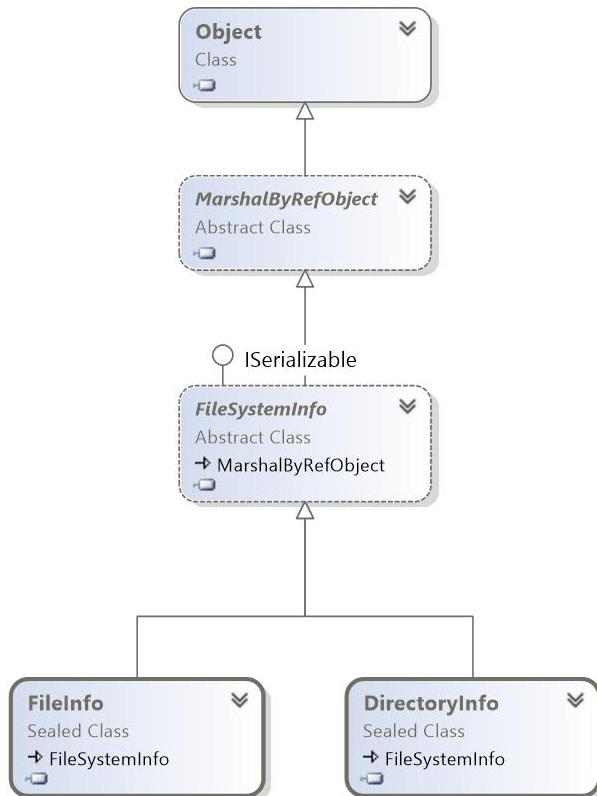


Abbildung 189: Die Hierarchie von `FileSystemInfo`

In Abbildung 189 sehen wir, dass `FileSystemInfo` zusätzlich noch von der Klasse `MarshalByRefObject` ableitet. Diese Klasse können Sie im Wesentlichen ignorieren: sie gehört zu einem alten .NET Standard, der zur Kommunikation zwischen zwei Applikation eingesetzt wurde (für sog. Remote Procedure Calls, kurz RPC). Mittlerweile nutzt man aber neuere Bestandteile des .NET Frameworks, um zwischen zwei Prozessen zu kommunizieren (bspw. mit WebAPI oder mit Windows Communication Foundation, kurz WCF).

Wenn Sie genaueres über die Klassen `DirectoryInfo` und `Directory` wissen möchten, können Sie dies in der MSDN Library nachschlagen:

- `DirectoryInfo`: <http://msdn.microsoft.com/en-us/library/system.io.directoryinfo.aspx>
- `Directory`: <http://msdn.microsoft.com/en-us/library/system.io.directory.aspx>

5.19.1.2.3 Pfade zu Dateien und Ordnern

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=cjDP5U5nmps>.

Ein Pfad beschreibt den Speicherort einer Datei oder eines Ordners. Die Schreibweise des Pfades wird dabei vom Betriebssystem vorgegeben und ist nicht bei allen Plattformen identisch. Üblicherweise beginnt ein absoluter Pfad mit der Angabe der Partition und wird dann fortgesetzt mit den Ordnerangaben, die jeweils durch ein bestimmtes Zeichen (bei Windows der Backslash) getrennt sind. Am Ende einer solchen Kette steht bei Dateien noch der Dateiname, abgeschlossen durch einen möglichen Punkt und der Dateierweiterung (wenn eine Datei diese besitzt).

Typische absolute Pfade sind bspw.:

- „C:\Users\Kenny\Pictures\elephant.png“ für einen absoluten Pfad zu einer Datei

- „D:\Programming\ITPG2\“ für einen absoluten Pfad zu einem Ordner

Neben absoluten Pfaden gibt es auch relative Pfade. Bei ihnen gibt man keine nicht die Partition an, auf der das Ziel liegt und gibt dann über die Ordnerstruktur an, welche Stelle das gesuchte Element liegt, sondern man geht vom einem bestimmten Ordner aus (üblicherweise der, in dem das Programm gerade ausgeführt wird) und zeigt dann mit relativen Pfadangaben auf das Zielelement. Dabei gibt man wie gewohnt Unterordner direkt per Namen, kann durch das Einfügen von zwei Punkten einen Ordner hinaufgehen. Relative Pfade könnten wie folgt aussehen:

- „temp.tmp“ für eine Datei, die im selben Ordner liegt wie das Programm, das gerade ausgeführt wird
- „..\lib\unity.dll“ für eine Datei, die im Parallelordner „lib“ liegt
- „Backup\2002\“ für einen Ordner, der zwei Ebenen unterhalb des aktuellen Verzeichnisses liegt.

Um Pfade in C# programmatisch zu verarbeiten, gibt es die statische Klasse `System.IO.Path`. Mit ihr sind die folgenden Funktionalitäten möglich:

Methode	Beschreibung
<code>ChangeExtension</code>	Ändert die Dateierweiterung für einen Dateipfad
<code>Combine</code>	Kombiniert zwei Pfadangaben (bspw. ein Ordner und ein Dateiname) zu einem Pfad
<code>GetDirectoryName</code>	Liefert aus einer Pfadangabe den untersten Ordnernamen
<code>GetExtension</code>	Liefert aus einer Pfadangabe die Dateierweiterung (einschließlich des Punkts)
<code>GetFileName</code>	Liefert aus einer Pfadangabe den vollständigen Dateinamen
<code>GetFileNameWithoutExtension</code>	Liefert aus einer Pfadangabe den Dateinamen ohne Dateierweiterung
<code>GetFullPath</code>	Liefert den absoluten Pfad zu einer relativen Pfadangabe
<code>GetPathRoot</code>	Liefert das Stammverzeichnis einer Partition für einen Pfad (dieses Stammverzeichnis ist im Dateiexplorer nicht sichtbar)
<code>GetTempPath</code>	Liefert den Pfad zum temporären Ordner für den aktuell angemeldeten Windowsnutzer
<code>HasExtension</code>	Prüft ob ein Pfad eine Dateierweiterung besitzt
<code>IsPathRooted</code>	Überprüft, ob es sich um einen absoluten Pfad handelt

Abbildung 190: Wichtige Methoden der Klasse Path

Von den in Abbildung 190 zu sehenden Methoden der Klasse `Path` benutzt man im Programmieralltag am häufigsten die Methode `Combine`, um aus zwei Teilstücken einen kompletten Pfad zu erstellen. Für einen genaueren Einblick in die API von `Path` können Sie auch hier die MSDN befragen: <http://msdn.microsoft.com/en-us/library/system.io.path.aspx>

5.19.1.2.4 Vereinfachte Pfadangabe im Code durch Verbatim-Stringliterale

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=UJUGbJUXI6g>.

Wie Sie bestimmt wissen, bietet auch C# die Möglichkeit, in Strings bestimmte Sonderzeichen wie `\n` für einen Zeilenumbruch oder `\t` für ein Tabulatorzeichen einzubauen. Beachten Sie dabei, dass jeweils der Backslash zur Einleitung eines solchen Sonderzeichens genutzt wird.

Dies ist für hardcodierte Pfadangaben im Quellcode allerdings nicht hervorragend, denn damit wir in einem Pfad einen Backslash einfügen können, müssen wir ihn mit `\\"` maskieren. Das sorgt dafür, dass unsere Pfadangaben deutlich schlechter lesbar werden:

```
var meinDateiPfad = "C:\\Users\\Kenny\\Pictures\\elephant.png";
```

Glücklicherweise bietet C# auch noch eine andere Art der Stringangabe, nämlich sog. Verbatim-Strings: diese beginnen mit einem @ vor den Anführungszeichen und haben den Vorteil, dass Sonderzeichen in ihnen nicht verarbeitet werden. Das heißt, wir können eben genannten Pfad auch wie folgt hartcodieren:

```
var meinDateiPfad = @"C:\\Users\\Kenny\\Pictures\\elephant.png";
```

Diese Pfadangabe ist identisch mit der Pfadangabe im Dateiexplorer und anderen Programmen, weswegen man dieser Schreibweise üblicherweise den Vorzug gibt.

Strings unterstützen noch viele weitere Möglichkeiten. Wenn Sie mehr darüber erfahren möchten, finden Sie in der MSDN die Referenz dazu: <http://msdn.microsoft.com/en-us/library/ms228362.aspx>

5.19.1.2.5 Die Klasse DriveInfo

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=G63EaHMM3kM>.

Mit der Klasse **DriveInfo** können Sie herausfinden, welche Partitionen und Laufwerke auf einem System verfügbar sind und um welchen Typ es sich jeweils handelt. Dabei bietet diese Klasse die folgenden Mitglieder:

Member	Beschreibung
AvailableFreeSpace	Gibt die Menge an verfügbarem Speicher für den aktuell angemeldeten Nutzer auf einem Laufwerk zurück
DriveFormat	Ruft den Namen des Dateisystems ab
DriveType	Ruft den Laufwerkstyp ab
GetDrives (statisch)	Gibt alle Laufwerke des aktuellen Computers zurück
Name	Liefert den Namen des Laufwerks
RootDirectory	Liefert das Stammverzeichnis des Laufwerks
TotalFreeSpace	Gibt die Menge an verfügbaren Speicher auf einem Laufwerk zurück
TotalSize	Gibt die Größe des Laufwerks zurück
VolumeLabel	Gibt das Label (den vom Nutzer gesetzten Namen) für ein Laufwerk zurück

Abbildung 191: Wichtige Mitglieder der Klasse DriveInfo

Im Gegensatz zur Situation bei Dateien oder Ordnern gibt es für **DriveInfo** keine statische Klasse als Pendant.

5.19.1.2.6 Dateisystemexplorer als Beispiel

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=9Xh72QcXN-0>.

In diesem Beispiel möchten wir die in den vorherigen Abschnitten kennengelernten Klassen in Kombination einsetzen, um alle Ordner und Dateien aller Partitionen bis zur dritten Ebene in der Konsole darzustellen. Dazu schreiben wir eine neue Klasse, die intern Rekursion nutzt, um die Ordner und Dateielemente anzuzeigen.

In Abbildung 192 sehen Sie diese Klasse namens **DateisystemExplorer**. Sie besitzt zwei Felder, wobei _ordnerebene über eine dazugehörige Eigenschaft verfügt, mit der das Feld von außen gesetzt werden kann. Damit kann der Nutzer der Klasse festlegen, wie viele Ebenen des Dateisystems dargestellt werden wollen (der Standardwert ist drei). Das andere Feld _aktuelleOrdnerebene wird nur intern für die Rekursion verwendet und wird für die Abbruchbedingung zu Rate gezogen.

Die interessante Methode in diesem Beispiel ist `ZeigeFestplatteninhalte`. Über Sie wird die Rekursion gestartet, wobei zunächst über `DriveInfo.GetDrives` alle Laufwerkinfos geholt werden und im Anschluss mit der LINQ Methode `Where` auf die gefiltert werden, bei denen es sich um Festplattenlaufwerke handelt. Im Anschluss wird diese gefilterte Collection durchlaufen, jeweils der Rootordner des jeweiligen Laufwerks bestimmt und dann die Methode `ZeigeOrdnerInhalteRekursiv` aufgerufen, welche für die angegebene Anzahl an Ebenen Subordner und Dateien ausgibt.

Diese Methode selbst enthält eigentlich nur die Abbruchbedingung für die Rekursion, die Ausgabe von Subordnern und Dateien eines Ordners wird an die Methode `ZeigeOrdnerInhalte` weiterdelegiert. In dieser werden auf den aktuellen Ordner die Mitglieder `GetDirectories` und `GetFiles` aufgerufen, um die entsprechenden Inhalte eines Ordners zu bekommen. In jeweils einer eigenen `foreach` Schleife werden diese dann auf der Konsole ausgegeben. Beim Aufruf der eben genannten Methoden kann es jedoch zu einer `UnauthorizedAccessException` kommen, da manche Ordner von Windows für Benutzer blockiert werden. Diese Exception wird im entsprechenden `catch` Block abgefangen, damit das Programm nicht abstürzt. Für jeden Unterordner wird jeweils wiederum `ZeigeOrdnerInhalteRekursiv` aufgerufen, damit die Rekursion fortgesetzt wird.

Bitte beachten Sie, dass bei sehr vielen Dateien und Ordnern die Ausgabe länger dauern könnte und nicht sehr übersichtlich ist, da die Konsole für solche Szenarien nicht gemacht wurde.

```

public class DateisystemExplorer
{
    private int _aktuelleOrdnerebene;
    private int _ordnerebene = 3;           Mit Ordnerebene kann der Nutzer festlegen,
                                            wie tief er ins Dateisystem vordringen möchte.
                                            Der Standardwert ist drei.

    public int Ordnerebene
    {
        get { return _ordnerebene; }
        set
        {
            if (value < 1) throw new ArgumentOutOfRangeException("value");
            _ordnerebene = value;
        }
    }                                       Zu Beginn werden alle DriveInfos von
                                            Festplatten gesucht

    public void ZeigeFestplatteninhalte()
    {
        var driveInfos =
            DriveInfo.GetDrives()
                .Where(driveInfo => driveInfo.DriveType == DriveType.Fixed);

        foreach (var driveInfo in driveInfos)
        {
            Console.WriteLine(driveInfo.Name);
            var rootOrdner = driveInfo.RootDirectory;
            ZeigeOrdnerInhalteRekursiv(rootOrdner);           Diese Methode wird rekursiv
            Console.WriteLine();                                aufgerufen zur Ausgabe der
                                                       Ordnerinhalte
        }
    }

    private void ZeigeOrdnerInhalteRekursiv(DirectoryInfo ordnerInfo)
    {
        _aktuelleOrdnerebene++;
        if (_aktuelleOrdnerebene <= _ordnerebene)
            ZeigeOrdnerInhalte(ordnerInfo);
        _aktuelleOrdnerebene--;
    }

    private void ZeigeOrdnerInhalte(DirectoryInfo ordnerInfo)
    {
        var tabs = new string(' ', _aktuelleOrdnerebene * 2);   Innerhalb dieser foreach
                                                               Schleifen werden die
                                                               Ordnerinhalte
                                                               tatsächlich ausgegeben
        try
        {
            foreach (var unterOrdnerInfo in ordnerInfo.GetDirectories())
            {
                Console.WriteLine("{0}{1}", tabs, unterOrdnerInfo.Name);
                ZeigeOrdnerInhalteRekursiv(unterOrdnerInfo);
            }
            foreach (var dateiInfo in ordnerInfo.GetFiles())
            {
                Console.WriteLine("{0}{1}", tabs, dateiInfo.Name);
            }
        }
        catch (UnauthorizedAccessException)           Manche Ordner sind für den Nutzer gesperrt, was
        {                                         mit einer Exception quittiert wird
            Console.WriteLine("{0}Auf diesen Ordner dürfen Sie nicht zugreifen.");
        }
    }
}

```

Abbildung 192: Einsatz von `DriveInfo`, `DirectoryInfo` und `FileInfo` im Verbund

5.19.1.3 Dateistreaming

5.19.1.3.1 Übersicht über Streams in .NET

Das Video hierzu finden Sie unter https://www.youtube.com/watch?v=bBa_Ve18hC8.

Bis jetzt haben wir für den Zugriff auf Dateien einfache Klassen benutzt, die entweder eine komplette Datei schreiben oder laden. Was passiert aber, wenn Dateien so groß sind, dass sie nicht in einem Rutsch in den Arbeitsspeicher geladen werden können? Das Programm stürzt in diesem Fall natürlich ab und kann nicht mehr weiterarbeiten, da der Prozessspeicher aufgebraucht ist. Dies wäre aber eine nicht-tragbare Limitierung in der heutigen Zeit: bspw. könnten wir Blu-Ray Filme nicht anschauen, da diese auf der Disc mehrere GB einnehmen, der Arbeitsspeicher aber deutlich weniger Umfang hat.

Der Punkt ist, dass man Dateien nur in den Fällen komplett in den Speicher einliest, wenn man sich sicher ist, dass diese eine gewisse Größe nicht überschreiten. Ansonsten setzt man auf Streaming. Dabei werden jeweils einige Bytes der Datei in den Speicher gelesen, verarbeitet und können danach wieder deallokiert werden. Es ist dabei nicht notwendig, dass die Datei vollständig im Speicher gehalten werden muss. Das hat z.B. auch den Vorteil, dass der Nutzer bspw. bei einem Video bereits anfangen kann zu schauen, bevor das komplette Video geladen ist.

Zu diesem Zweck gibt es die Basisklasse `System.IO.Stream`. Ein Stream kapselt im Wesentlichen eine Quelle, aus der gelesen oder in die geschrieben werden soll, sowie die aktuelle Position, an der der Stream gerade steht. Über die Methoden `Read` und `Write` kann beginnend an der aktuellen Position eine gewisse Anzahl an Bytes gelesen bzw. geschrieben werden. Einige Streams unterstützen auch das versetzen der aktuellen Position an eine andere Stelle im Stream, was man als Suchen bezeichnet und über die Methode `Seek` ausgeführt wird.

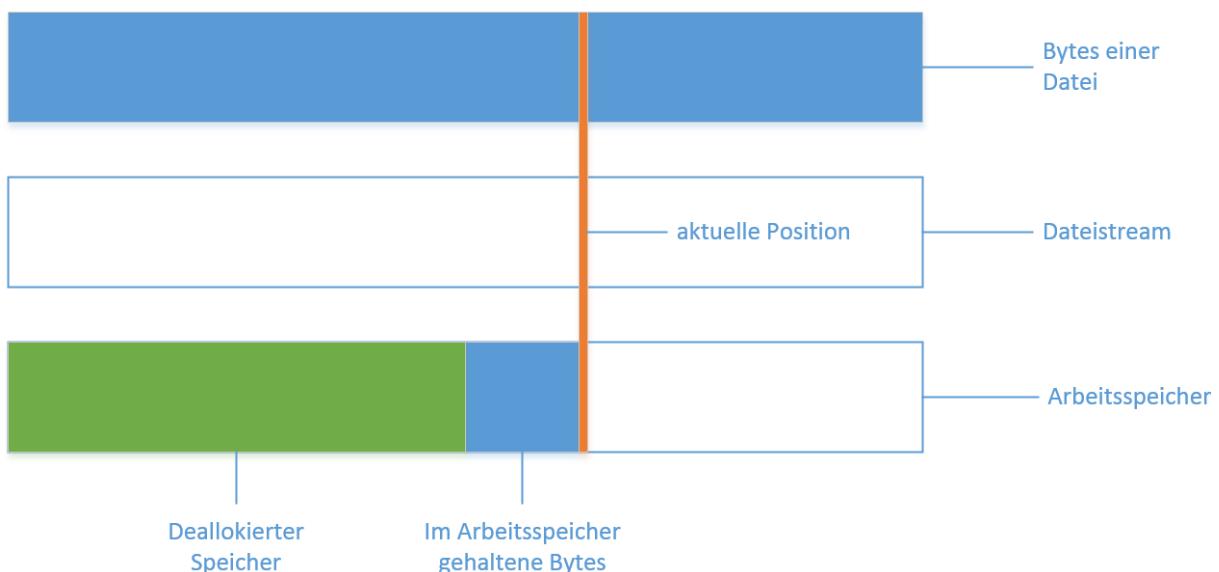


Abbildung 193: Streaming bei lesendem Zugriff auf eine Datei

In Abbildung 193 sehen Sie das eben beschriebene Prinzip am Beispiel eines lesenden Zugriffs auf eine Datei: Der Stream kapselt den Zugriff auf diese und besitzt eine aktuelle Position. Von außen kann der Nutzer den Stream steuern, indem er die Methode `Read` aufruft und damit Bytes erhält, die an der entsprechenden Position in der Datei stecken (beim Aufruf von `Read` wird natürlich auch die aktuelle Position um die entsprechende Anzahl Bytes automatisch verschoben). Bereits verarbeitete Bytes können dann wieder aus dem Speicher entfernt werden, sodass immer nur ein Teil der Datei tatsächlich im Speicher gehalten wird. Dieser deallokierte Speicher ist im Bild grün dargestellt.

Wie weiter oben bereits angedeutet, wird Streaming nicht nur bei Dateien eingesetzt. Einige der wichtigsten Klassen, die von Stream ableiten, sind in folgender Tabelle aufgelistet:

Subklasse von Stream	Beschreibung
FileStream	Repräsentiert einen Stream zu einer Datei im Dateisystem. Eine im Netzwerk verfügbare Datei kann ebenfalls Ziel dieses Streams sein.
MemoryStream	Repräsentiert einen Stream, der im Hauptspeicher gehalten wird (für temporäre Daten oder zu Testzwecken)
NetworkStream	Repräsentiert einen Stream, der Daten via sog. Sockets zur Interprozesskommunikation im Netzwerk sendet. Dieser Stream unterstützt von Haus aus kein Suchen.

Abbildung 194: Wichtige Subklassen von Stream

Dabei bieten alle diese Streams folgende Members, die sie von der Basisklasse erben und jeweils überschreiben:

Mitglied	Beschreibung
Close	Schließt den Stream und gibt mögliche Systemressourcen wieder frei (diese Methode kann auch über <code>IDisposable.Dispose</code> aufgerufen werden, das jeder Stream implementiert)
CurrentPosition	Aktuelle Position innerhalb des Streams
Length	Komplette Größe der zugrunde liegenden Quelle
Read	Liest eine Folge von Bytes von der Quelle und gibt diese als Array zurück
ReadByte	Liest genau ein Byte ein und gibt dieses zurück
Seek	Legt die aktuelle Position innerhalb des Streams fest (sofern das vom jeweiligen Stream unterstützt wird; falls nicht, fliegt eine <code>NotSupportedException</code> zur Laufzeit)
Write	Schreibt eine Folge von Bytes an die aktuelle Position im Stream
WriteByte	Schreibt genau ein Byte an die aktuelle Position im Stream

Abbildung 195: Wichtige Mitglieder der Streamklassen

Passend zum aktuellen Thema sehen wir uns im folgenden Abschnitt die Klasse `FileStream` in einem Beispiel an.

5.19.1.3.2 Die Klasse `FileStream` im Beispiel

Das Video hierzu finden Sie unter https://www.youtube.com/watch?v=lt2ey_kILk0.

Im folgenden Beispiel sehen wir, wie man die Klasse `FileStream` einsetzen kann, um jeweils ein Byte aus einer Textdatei zu lesen, dieses in einen Buchstaben umzuwandeln und dann auf die Konsole zu schreiben:

```

Über den Konstruktor wird ein Dateistrom initialisiert. Über den
FileMode kann man bspw. angeben, ob eine neue Datei erstellt
oder eine bestehende geöffnet werden soll.

private static void Main()
{
    var dateistrom = new FileStream("Sample Text.txt", FileMode.Open);

    while (dateistrom.Position < dateistrom.Length)
    {
        var @byte = dateistrom.ReadByte();  
____ Mit ReadByte wird jeweils genau ein Byte aus der
        Console.WriteLine(Convert.ToChar(@byte));
    }

    dateistrom.Close();  
____ Niemals vergessen: sobald ein Dateistream nicht mehr benötigt
    Console.ReadLine();  
____ wird, sollte man Close oder Dispose aufrufen, damit die
                                dazugehörigen Systemressourcen wieder freigegeben werden
}

```

Abbildung 196: Einen FileStream lesend einsetzen

In Abbildung 196 sehen wir, dass ein Dateistrom direkt über den Konstruktor initialisiert und gestartet wird. Dabei gibt man den Pfad zu einer Datei an (Sample Text.txt ist hier direkt ins Verzeichnis des Programms kopiert worden, siehe Video) und über die `FileMode` Enumeration kann man angeben, wie die Datei behandelt werden soll: in diesem Fall wird sie geöffnet, man kann aber auch spezifizieren, dass eine Datei erstellt / überschrieben oder erweitert werden soll; üblicherweise benutzt man im Programmieralltag `FileMode.Open` oder `FileMode.Create`). Es gibt noch weitere Überladungen des Konstruktors, mit denen man über die Enumerationen `FileAccess` und `FileShare` spezifizieren kann, ob lesend und/oder schreibend zugegriffen wird bzw. ob andere Prozesse diese Datei lesend oder schreibend öffnen dürfen.

Das eigentlich interessante passiert innerhalb der `while` Schleife, die abbricht, sobald das Ende des Dateistroms erreicht ist: mit `dateistrom.ReadByte()` wird jeweils nur ein einzelnes Byte aus der Datei eingelesen und in der Variablen `@byte` festgehalten. Dieses Byte wird im Anschluss konvertiert und auf der Konsole ausgegeben. Im Gegensatz zu den entsprechenden Methoden auf `File` und `FileInfo` wird also nicht die komplette Datei eingelesen – obwohl das natürlich in diesem Beispiel auch ohne Probleme möglich wäre.

Wichtig: vergessen Sie niemals, einen Stream zu schließen, sobald Sie ihn nicht mehr brauchen. Die Systemressourcen für diesen Stream bleiben sonst solange in ihrem Besitz, bis der Garbage Collector das entsprechende Objekt deallokiert (und das kann dauern). Zu diesem Grund implementieren auch alle Klassen, die Systemressourcen kapseln, das Interface `IDisposable` in .NET: rufen sie immer `Dispose` (oder im Fall von Streams auch `Close`; `Close` leitet den Aufruf an `Dispose` intern weiter) auf, wenn das entsprechende Objekt nicht mehr benötigt wird. Sie können für alle Objekte, die `IDisposable` implementieren, auch einen sog. `using` Block einsetzen, an dessen Ende das Objekt automatisch disposed wird. Wie das geht, sehen Sie im Video für diesen Abschnitt.

Auch wenn wir `FileStream` jetzt direkt eingesetzt haben, ist das nicht der übliche Weg, den man im Programmieralltag geht. Üblicherweise setzt man auf andere Klassen, die intern einen `FileStream` verwenden. Warum man das macht, sehen wir uns in den kommenden Abschnitten an.

5.19.1.4 Serialisierung und Deserialisierung

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=NdsgIBI2sQE>.

Binär- und Textdateien sowie Streaming schön und gut, aber was wir eigentlich als objektorientierte Entwickler wollen, sind komplett Objektgraphen speichern oder laden. Wir erinnern uns:

Objektgraphen sind ein Verbund von Objekten, die sich gegenseitig referenzieren und die wir über die Referenz eines dieser Objekte (meist das sog. Rootobjekt) ansprechen. Beispielsweise kann das eine Liste von Kontakten sein, wie im folgenden Beispiel dargestellt:

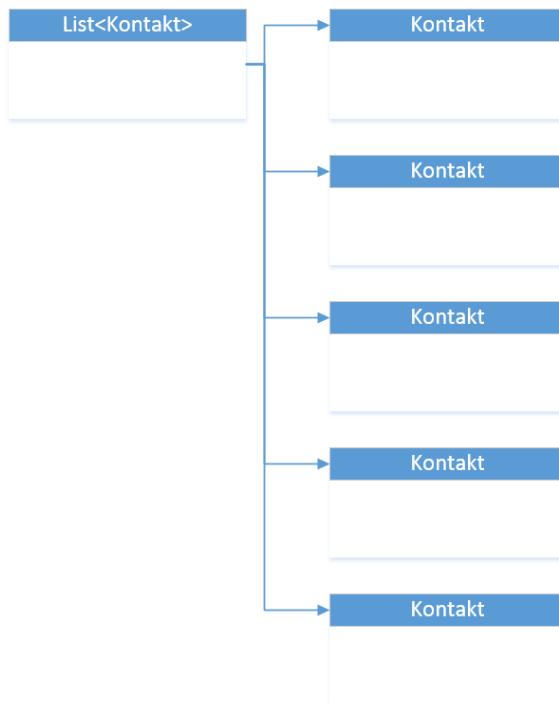


Abbildung 197: Objektgraph mit Liste und fünf weiteren Objekten

In Abbildung 197 sehen Sie einen Objektgraph, der aus einer `List<Kontakt>` besteht, welche wiederum fünf weitere `Kontakt`-Objekte referenziert (der Fakt, dass die Liste intern ein Arrayobjekt nutzt, dass eigentlich die Kontakte referenziert, lassen wir hier zur Vereinfachung des Sachverhalt außen vor). Das Root-Objekt wäre in diesem Fall die Liste, die bspw. per Parameter weitergegeben werden kann und so jedem beliebigen Nutzer Zugriff auf alle Objekte dieses Objektgraphs bietet. Häufig sind Objektgraphen ähnlich wie hier in einer sog. Baumstruktur aufgebaut, das muss aber nicht immer der Fall sein.

Die Frage, die offen ist: wie können wir einen solchen Objektgraph einfach speichern bzw. die Inhalte einer Datei wieder zu einem Objektgraph rekonstruieren? Wie wir wissen, brauchen wir entweder einen String oder einen Byte-Array, um Inhalte in eine Datei zu schreiben. Ein Objektgraph ist vom Typ her mit diesen beiden jedoch nicht kompatibel. Und genau hier kommt die Serialisierung zum Tragen.

Das Überführen von Objektgraphen in ein Text- oder Byteformat, dass bspw. gespeichert oder via Netzwerk übertragen werden kann, bezeichnet man als Serialisieren. Der umgekehrte Prozess, d.h. aus einem Text- oder Byteformat wieder Objekte herzustellen, nennt man Deserialisieren.

Um Objekte zu speichern, brauchen wir also insgesamt einen zweistufigen Prozess:

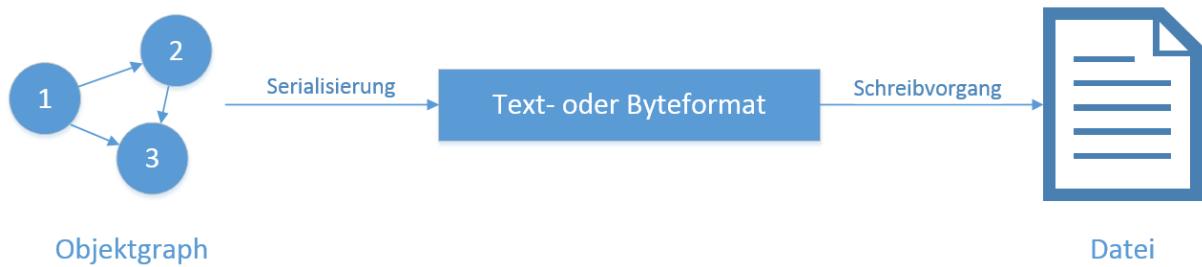


Abbildung 198: Der Prozess des Speicherns von Objektgraphen

Analog umgekehrt sind natürlich auch beim Laden und Umformen von Dateiinhalten zu Objektgraphen zwei Schritte notwendig:

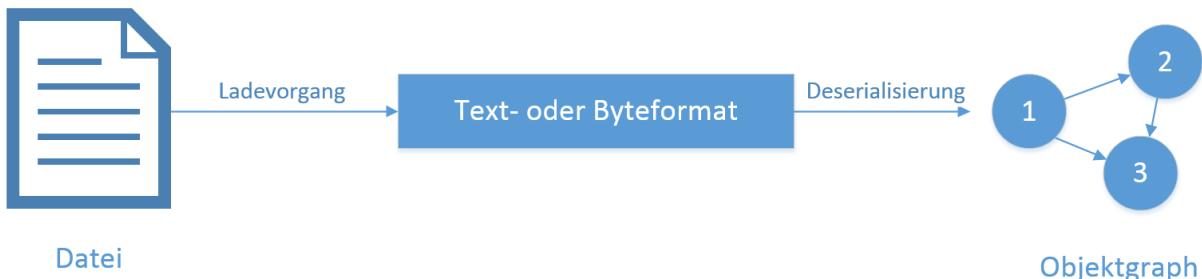


Abbildung 199: Der Prozess des Ladens von Objektgraphen

Glücklicherweise sind diese beiden Prozesse ein bekanntes Problem der Informatik und weitgehend in Frameworks implementiert. Auch das .NET Framework bietet dazu mehrere APIs an, wobei wir uns eine davon anschauen werden. Bevor wir dies machen, schauen wir uns jedoch noch gängige Formate an, in die Objektgraphen serialisiert werden können.

5.19.1.5 Das Serialisierungsformat XML

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=GbwU5K5HRt0>.

In diesem Abschnitt möchten wir genauer darauf eingehen, wie Objektgraphen genau in ein Format überführt werden können, das gespeichert werden kann. Immerhin müssen wir dafür Rechnung tragen, dass beim Serialisieren keine Informationen über ein Objekt verloren gehen. Hierzu brauchen wir eine flexible Darstellungsform, die, egal wie ein Objekt strukturiert ist, dessen Informationen halten kann.

Erfreulicherweise brauchen wir uns auch hierüber keine Gedanken machen, denn es gibt bereits einige standardisierte Serialisierungsformate, wobei XML (Extensible Markup Language) und JSON (JavaScript Object Notation) die zurzeit am häufigsten verwendeten sind. Wir werden uns hier genauer mit XML beschäftigen.

XML kann man wie so vieles in der Programmierung auch am besten an einem Beispiel erklären. Deswegen schauen wir uns auch gleich ein solches an und besprechen dabei, aus welchen Teilen ein XML Dokument eigentlich besteht:

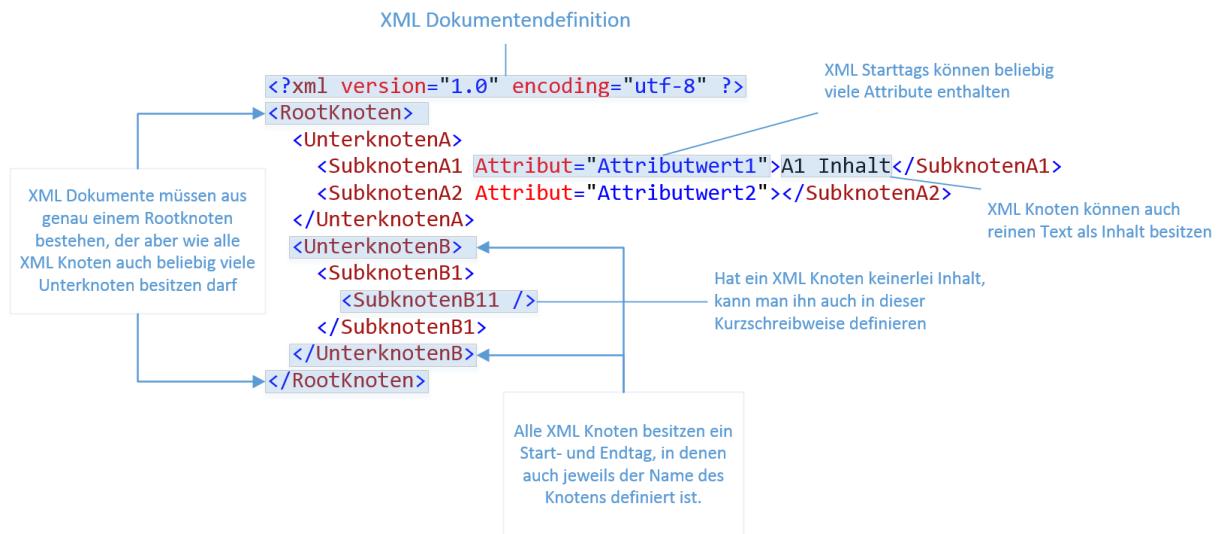


Abbildung 200: Ein XML Beispieldokument

In Abbildung 200 sehen Sie ein XML Dokument, das wie folgt aufgebaut ist:

- Zu Beginn eines XML Dokuments steht immer die XML Dokumentendefinition. Diese ist in nahezu allen Fällen wie oben gekennzeichnet. Üblicherweise nutzt man UTF 8 für XML-Dateien.
`<?xml version="1.0" encoding="utf-8" ?>`
- Danach folgt der sog. Rootknoten. Ein XML Dokument darf auf oberster Ebene nur einen einzigen Knoten besitzen, dieser kann allerdings wie jeder andere XML Knoten auch beliebig viele Unterknoten enthalten.
- Ein XML Knoten besteht aus einem Start- und einem Endtag, in dem jeweils der Bezeichner des Knoten steht. Beim Endtag ist jeweils noch ein Slash vor dem Bezeichner zu finden, der das Ende des Knotens markieren soll.
`<Knotenbezeichner>Inhalt</Knotenbezeichner>`
- Innerhalb dieser Tags stehen die Inhalte des Knotens. Das sind entweder:
 - weitere XML Subknoten
 - Beliebiger Text
- Weiterhin können auf einem Starttag beliebig viele Attribute definiert werden nach dem Schema: **Attributname="Attributwert"**
 Bitte beachten Sie, dass XML Attribute rein gar nichts mit C# / .NET Attributen, die zur Kennzeichnung von bestimmten Codestellen genutzt werden, zu tun haben.
- Wenn ein Knoten keine Inhalte hat (bspw. weil auf ihm bloß Attribute definiert sind), dann kann man ihn auch in einer Kurzschreibweise definieren nach folgendem Schema:
`<Knotenbezeichner />`
 Das Endtag kann dann in diesem Fall weggelassen werden.

Durch die hierarchische Gliederung der Knoten kann man beliebige Informationen auf XML abbilden. Bspw. kann man unsere Kontaktliste aus Abbildung 197 wie folgt darstellen:

```

<?xml version="1.0" encoding="utf-8" ?>
<Kontakte>
    <Kontakt ID="1">
        <Vorname>Walter</Vorname>
        <Nachname>White</Nachname>
        <Alter>52</Alter>
    </Kontakt>
    <Kontakt ID="2">
        <Vorname>Jesse</Vorname>
        <Nachname>Pinkman</Nachname>
        <Alter>27</Alter>
    </Kontakt>
</Kontakte>

```

Abbildung 201: Liste mit Kontakten in XML dargestellt

Wie in Abbildung 201 zu sehen ist, kann eine Klasse also als Knoten mit weiteren Subknoten modelliert werden. Im Beispiel von Kontakt wird eben Vorname, Nachname und Alter jeweils in den Elternknoten eingefügt, wohingegen die ID als Attribut auf dem Starttag des Elternknotens mitaufgenommen wird. Ob Sie primitive Typen als Attribut oder Subknoten modellieren, ist dabei vollkommen egal. Referenziert ihre Klasse aber andere Klassen (also komplexe Datentypen), dann ist man meistens gezwungen, diese Verbindung als Subknoten in XML darzustellen.

Es gibt einen Punkt, den wir bei XML noch nicht angesprochen haben: Knoten können sog. XML Namespaces zugeordnet werden. Durch sog. XML Schemas kann dann der Inhalt eines XML Dokuments verifiziert werden, indem man überprüft, ob die Knoten eines Dokuments aus einem bestimmten Namensraum eingesetzt wurden und diese in einer bestimmten Struktur vorhanden sind. Das Prinzip der XML Namespaces ist für uns jetzt aktuell noch nicht wichtig, dennoch möchte ich kurz zeigen, wie sie funktionieren:

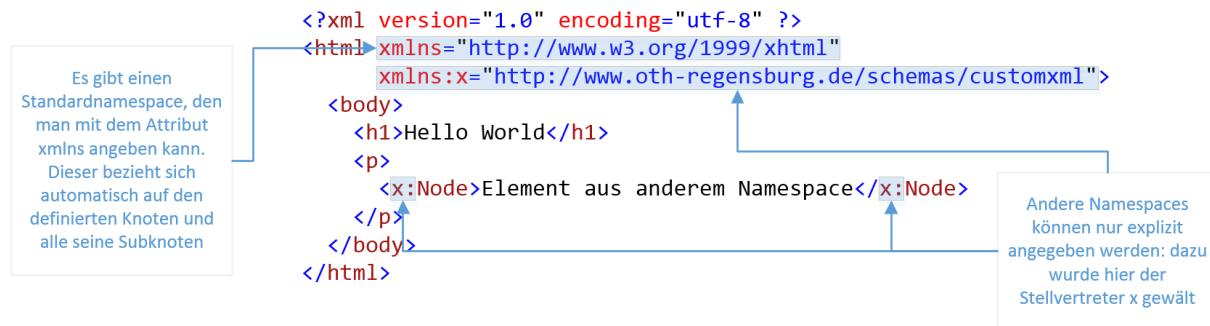


Abbildung 202: XML Namespaces

In Abbildung 204 sehen wir, dass zwei Namespaces definiert worden sind:

- Mit `xmlns="http://www.w3.org/1999/xhtml"` wurde der sog. Standardnamensraum festgelegt, der für den Knoten gilt, auf dem er gesetzt wurde, und auf alle seine Subknoten, welche kein Präfix einsetzen (wie bspw. `x:Node`). D.h. dass die Elemente `html`, `body`, `h1`, und `p` jeweils zu diesem Namespace gehören.
- Andere Namespaces können nur explizit mit einem Stellvertreter angegeben werden, wie das bei `xmlns:x="http://www.oth-regensburg.de/schemas/customxml"` der Fall ist:

der Stellvertreter ist hier x, könnte aber auch jedes andere beliebige zusammenhängende Wort sein (üblicherweise nimmt man aber nur einen Buchstaben). Wenn Knoten zu diesem Namensraum gehören sollen, muss man diesen den entsprechenden Stellvertreter als Präfix mit Doppelpunkt getrennt voranstellen (siehe `x:Node`).

Wir sehen auch, dass XML Namespaces als URLs dargestellt werden: auch das ist üblich, allerdings verbirgt sich dahinter nicht immer ein valider Link. Häufig wird einfach die Domain eines Unternehmens im Namespace mitgenutzt, damit der Namespace eindeutig ist (in der Hoffnung, andere Unternehmen werden den eigenen Domainnamen nicht in ihren XML Namespaces verwenden).

Wie bereits erwähnt: Direkt nützlich für unseren direkten Umgang sind XML Namensräume noch nicht. Bitte nehmen Sie aber mit, dass es sie gibt. U.a. in der Oberflächenprogrammierung werden wir noch auf sie stoßen. Im nächsten Abschnitt möchten wir uns jedoch erstmal mit der Umformung von Objektgraphen zu XML beschäftigen.

5.19.1.6 Objekte automatisch zu XML umformen in .NET

Um einen Objektgraph zu XML mit .NET umzuformen, ist relativ wenig Arbeit nötig. Die Hauptarbeit wird von der Klasse `System.Runtime.Serialization.DataContractSerializer` erledigt. Diese liegt in der Assembly `System.Runtime.Serialization`, die standardmäßig noch nicht von Konsolenprojekten referenziert wird und die Sie deshalb immer zu Ihrem Projekt hinzufügen müssen, wenn Sie diese Klasse einsetzen möchten (im Video zu diesem Abschnitt wird dies auch nochmals erklärt).

`DataContractSerializer` wird sowohl zur Serialisierung als auch zur Deserialisierung von Objekten benutzt. Dazu bietet diese Klasse die Methoden `ReadObject` und `WriteObject` an – letztere übergibt man ein Rootobjekt zur Serialisierung eines Objektgraphs, von ersterer erhält man die Referenz auf das Rootobjekt eines Objektgraphs nach dem Deserialisieren zurück. Dabei benutzt der `DataContractSerializer` jeweils noch eine Instanz von `XmlReader` bzw. `XmlWriter` – diese beiden Klassen können intern einen Stream kapseln und XML aus diesem lesen bzw. auf diesen schreiben. Wie wir diese Klassen einsetzen, sehen wir uns genauer im nächsten Abschnitt an, jetzt möchten wir uns jedoch befassen, nach welchen Kriterien der `DataContractSerializer` die Serialisierung genau vornimmt.

Der `DataContractSerializer` weiß von Haus aus, wie primitive Datentypen und Collections in XML zu überführen sind, allerdings muss man bei komplexen Datentypen wie Klassen als Entwickler selbst festlegen, welche Felder und / oder Eigenschaften vom `DataContractSerializer` beachtet werden sollen. Dies macht man mit den C# / .NET Attributen `DataContractAttribute` und `DataMemberAttribute`, die man auf die entsprechenden Members der Klasse setzt. Sehen wir uns dazu folgendes Beispiel der modifizierten Klasse Kontakt an:

```

[DataContract]
public class Kontakt
{
    [DataMember]
    public int ID { get; set; }

    [DataMember]
    public string Vorname { get; set; }

    [DataMember]
    public string Nachname { get; set; }

    public string VollerName { get { return Vorname + " " + Nachname; } }

    [DataMember]
    public string Telefonnummer { get; set; }

    [DataMember]
    public string Email { get; set; }
}

```

Abbildung 203: Klassen mit Serialisierungsinformationen ausstatten

In Abbildung 203 sehen wir, dass die Klasse Kontakt mit dem **DataContractAttribute** ausgestattet wurde, damit dem **DataContractSerializer** erkenntlich gemacht wird, dass er diese Klasse serialisieren darf. Sobald er dies tut, beschafft er sich via Reflection alle Members, die mit dem **DataMemberAttribute** gekennzeichnet sind und führt diese in XML über. Im obigen Beispiel ist davon die Eigenschaft **VollerName** ausgenommen, da diese nur eine **get** Methode hat, die Vor- und Nachname verbindet und zurückgibt. Diese Information ebenfalls zu speichern wäre redundant, weswegen diese Eigenschaft nicht mit dem **DataMemberAttribute** gekennzeichnet ist.

Wie man alle diese Funktionalität im Verbund einsetzt, sehen wir uns im nächsten Abschnitt an.

5.19.1.7 *DataContractSerializer zum Laden und Speichern einsetzen*

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=wK9J7pyZ9xs>.

Wir möchten nun eine Klasse schreiben, welche beliebige Objekte in einer Datei speichern oder aus einer Datei laden kann. Dazu werden auf der Klasse zwei Methoden **SpeichereObjektgraph** und **LadeObjektgraph** definiert. Innerhalb dieser werden wir jeweils den **DataContractSerializer**, einen **XmlReader** bzw. **XmlWriter** und einen **FileStream** einsetzen, welche die eigentliche Aufgabe im Verbund erledigen. Schauen wir uns dazu direkt das Beispiel an:

```

public class PersistenzService
{
    public void SpeichereObjektgraph<T>(string dateipfad, T rootobjekt)
    {
        if (dateipfad == null) throw new ArgumentNullException("dateipfad");

        XmlWriter xmlWriter = null;
        try
        {
            var fileStream = new FileStream(dateipfad, FileMode.Create);
            xmlWriter = XmlWriter.Create(fileStream, new XmlWriterSettings
            {
                Ein XML Writer kann XML in einen Stream schreiben und
                wird über XmlWriter.Create erzeugt
                CloseOutput = true,
                Indent = true,
                IndentChars = "    "
            });
            var dataContractSerializer = new DataContractSerializer(typeof(T));
            dataContractSerializer.WriteObject(xmlWriter, rootobjekt);
        }
        finally
        {
            if (xmlWriter != null)
                xmlWriter.Close();
        }
    }
}

Wichtig: Systemressourcen freigeben

public T LadeObjektgraph<T>(string dateipfad)
{
    if (dateipfad == null) throw new ArgumentNullException("dateipfad");

    XmlReader xmlReader = null;
    try
    {
        var fileStream = new FileStream(dateipfad, FileMode.Open);
        xmlReader = XmlReader.Create(fileStream, new XmlReaderSettings
        {
            Ein XmlReader kann XML Text in Streams interpretieren
            CloseInput = true
        });
        var dataContractSerializer = new DataContractSerializer(typeof(T));
        return (T) dataContractSerializer.ReadObject(xmlReader);
    }
    finally
    {
        if (xmlReader != null)
            xmlReader.Close();
    }
}

Wichtig: Systemressourcen freigeben
}

```

Mit WriteObject wird der Objektgraph serialisiert und in den Filestream geschrieben

ReadObject deserialisiert einen Objektgraph mithilfe des XmlReader aus einem Stream

Abbildung 204: Die Klasse PersistenzService zum Speichern und Laden von Objektgraphen

In Abbildung 204 wird in der Methode SpeichereObjektgraph<T> folgendes gemacht:

- Zunächst wird ein `FileStream` geöffnet, wobei der Pfad eingesetzt wird, der via Parameter an die Methode übergeben wurde.

- Im Anschluss wird ein `XmlWriter` erstellt. Diese Klasse hat keinen öffentlich zugänglichen Konstruktor, sondern wird über eine sog. statische Factory-Methode `XmlWriter.Create` erstellt. Dieser Methode kann man in der angewandten Überladung auch Einstellungen mit `XmlWriterSettings` mitgegeben – in diesen wurde spezifiziert, dass der `XmlWriter` den Code einrückt (engl. indent) und automatisch den gekapselten `FileStream` schließen soll, wenn er selbst geschlossen wird.
- Zuletzt wird der `DataContractSerializer` instanziert und eingesetzt. Beim Konstruktorauftrag wird ihm der Typ, den er serialisieren soll, übergeben. Mit dem Aufruf von `WriteObject` wird der komplette Objektgraph in einem Rutsch geschrieben – dabei setzt der `DataContractSerializer` den `XmlWriter` ein, um die serialisierten Objekte im XML Format in den Dateistrom zu schreiben.
- In der Methode existiert auch noch ein `try-finally` Block, der eingesetzt wird, um den `XmlWriter` und den darunter liegenden `FileStream` in jedem Fall zu schließen, egal ob eine Exception auftritt oder nicht.

Da diese Methode mit T generisch parametriert ist, können beliebige Rootobjekte übergeben und gespeichert werden. Wichtig ist bloß, dass sämtliche Objekte im Graph entweder primitive Datentypen bzw. Collections oder mit dem `DataContractAttribute` ausgezeichnete Klassen sind, damit der `DataContractSerializer` weiß, wie er die einzelnen Typen in XML überführen soll.

Interessant ist ebenfalls, dass die Lademethode einen sehr ähnlichen Aufbau hat wie die Speichermethode. Natürlich wird hier ein `XmlReader` eingesetzt anstatt eines `XmlWriters` sowie `ReadObject` statt `WriteObject` aufgerufen, ansonsten ist der Code von der Struktur her sehr ähnlich.

Dies ist die einfachste Möglichkeit, Objekte in .NET in ein Standardformat wie XML zu überführen und dann in einer Datei zu speichern (bzw. analog umgekehrt die Daten zu laden), ohne dass man sich dabei einen eigenen Serialisierungsmechanismus überlegen muss. Natürlich gibt es bei der Serialisierung von Objekten noch einiges mehr zu beachten – bspw. können folgende Fragestellungen oder Probleme auftauchen:

- Was passiert, wenn ich in einem Objekt ein anderes Objekt referenziere, dieses aber nur über eine Abstraktion (also Basisklasse oder Interface) anspreche?
- Wie werden die Felder oder Properties, die von der Basisklasse bereitgestellt werden, serialisiert? Was mache ich, wenn eine Basisklasse gar keine Serialisierung unterstützt, weil auf ihr das `DataContractAttribute` nicht definiert ist?
- Was passiert bei zirkulären Referenzen zwischen Objekten (wenn A und B sich jeweils gegenseitig referenzieren) - wie kann der `DataContractSerializer` das auflösen?

Auf diese weitergehenden Fragen werden wir in diesem Kurs nicht eingehen, auch hierfür gibt es aber standardisierte Lösungen in den jeweiligen Frameworks (in unserem Fall muss man beim Erstellen des `DataContractSerializer` einen anderen Konstruktor verwenden, um ihn entsprechend zu konfigurieren). Wenn Sie in Ihrem Programmieralltag vor so einem Problem stehen, dann reicht es meistens, eine Suchmaschine mit den entsprechenden Schlüsselwörtern einzusetzen, um auf Dokumentations- oder Blogseiten zu gelangen, wo relativ leicht die Lösung zu finden ist.

5.19.1.8 Zusammenfassung für Zugriff auf Dateien

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=Gh72SWIlo-Q>.

Wir haben in den letzten Abschnitten gesehen, dass man zwar etwas Hintergrundwissen um Serialisierung, Formate und Streaming benötigt und die entsprechenden Klassen des .NET

Frameworks kennen sollte, dann aber ist Persistenz mit Dateien eine relativ einfache Sache: im Prinzip muss man nur die zu speichernden Klassen mit den entsprechenden Attributen kennzeichnen und dann `DataContractSerializer` und `XmlWriter / XmlReader` in Kombination einsetzen, um die gewünschte Funktionalität des Speicherns oder Ladens zu erreichen.

Dies sind jedoch nicht die einzigen APIs, die dafür eingesetzt werden können: bereits seit .NET 1.0 sind die Klassen `System.Xml.Serialization.XmlSerializer` und `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter` verfügbar, welche genau das gleiche Problem lösen wie `DataContractSerializer`. Der Unterschied ist, dass letzterer XML Dokumente erzeugt, die plattformunabhängig sind, d.h. dass diese auch bspw. von Prozessen, die in Java implementiert sind, leichter konsumiert werden können. `XmlSerializer` erzeugt natürlich auch XML, wohingegen `BinaryFormatter` ein proprietäres Binärformat von Microsoft einsetzt.

Das wichtigste, was Sie jedoch mitnehmen sollten, ist der Punkt, dass man als Entwickler den Anspruch haben sollte, nach der einfachsten API zur Lösung eines Problems zu suchen. Versuchen Sie immer ausgehend von einem Objektgraph hin zum Ziel zu kommen und dabei möglichst viel Funktionalität aus Bibliotheken wiederzuverwenden – Sie sparen dadurch Zeit durch eine Neuimplementierung und höchstwahrscheinlich ist die Frameworklösung deutlich ausgereifter als der Code, den man selbst zu Lösung des Problems erstellen würde.

5.19.2 Ein kurzer Einblick in den Zugriff auf Datenbanken (Level 200)

5.19.2.1 Was sind Datenbanken?

Das Video hierzu finden Sie unter https://www.youtube.com/watch?v=MSBr_Aos9oU.

Datenbanken sind wohl im heutigen Programmieralltag das am häufigsten eingesetzte Mittel zur Persistenz – ursprünglich wurden Sie in den 1960er Jahren entwickelt, um eine zusätzliche Schicht zwischen Betriebssystem und Anwendungen zu haben, damit letztere weniger Code mit der Verwaltung von Dateien enthalten müssen. Dabei haben sich mittlerweile folgende Datenbanktypen etabliert:

- Relationale Datenbanken: diese speichern Daten in verschiedenen Tabellen, wobei innerhalb einer Tabelle ein Dateneintrag eindeutig identifizierbar sein muss durch einen sog. Primärschlüssel. Dieser ist häufig einfach ein Integer, der von der Datenbank für die jeweilige Tabelle automatisch verwaltet werden kann (d.h. neue Einträge in die Tabelle bekommen diesen Schlüssel automatisch zugewiesen). Wenn ein Dateneintrag Einträge aus anderen Tabellen referenzieren möchte, dann wird das über sog. Fremdschlüssel gemacht: dies sind einfach Einträge in einer Tabelle, die auf die Primärschlüssel einer anderen Tabelle verweisen. Bei einer Abfrage an die Datenbank werden üblicherweise mehrere Tabellen miteinbezogen und die jeweiligen Tabellendaten durch sog. Joins zusammengeführt (die Tendenz ist hier: je mehr Joins, desto länger die Abfragedauer).
- NoSQL Datenbanken: diese Datenbanken sind ein Gegenentwurf zu relationalen Datenbanken, die ihre Daten über mehrere Tabellen verteilen. Im Wesentlichen funktionieren Sie häufig wie ein Dictionary, d.h. sie bilden ein Mapping zwischen einem Schlüssel (ID) und dem eigentlichen Objektdaten. Diese sind häufig in einem Serialisierungsformat abgelegt, dass bspw. XML oder JSON entsprechen kann. Der Zugriff erfolgt hier auch meistens einfach über Objekt-IDs, sodass die entsprechenden Objektdaten geladen werden.

Bitte beachten Sie, dass diese Beschreibungen zu den beiden Typen sehr knapp sind und natürlich gibt es auch noch andere Typen von Datenbanken – zu sehr möchten wir in das Thema nicht

einstiegen. Heutzutage werden hauptsächlich relationale Datenbanken genutzt, wobei NoSQL Datenbanken für einige Probleme eine deutlich bessere Performance bieten, u.a. bei der Datenabbildung von sozialen Netzwerken. Wir werden uns dennoch mit dem Zugriff auf relationale Datenbanken beschäftigen.

5.19.2.2 Zugriff auf relationale Datenbanken

Auf relationale Datenbanken wird üblicherweise mit der Sprache SQL (kann auch engl. Sequel ausgesprochen werden, dt. Fortsetzung oder Nachfolge) eingesetzt. Mit ihr kann man sowohl die Struktur von Tabellen definieren, Tabelleneinträge bearbeiten (Einfügen, Verändern, Löschen) sowie Daten abfragen. Syntaxdetails zu SQL möchte ich hier nicht aufführen, da Sie dies in der dafür vorgesehenen Vorlesung Datenbanken machen werden. Allerdings möchte ich hier ein exemplarisches Select-Statement zeigen, mit dem Daten aus zwei Tabellen abgefragt und verknüpft werden:

<code>SELECT Vorlesung.Titel, Professor.Name</code>	Im SELECT Teil der Abfrage werden die Spalten ausgewählt, deren Felder abgefragt werden sollen
<code>FROM Professoren, Vorlesung</code>	Im FROM Teil werden die Tabellen angegeben, welche die obigen Spalten enthalten
<code>WHERE Professor.ID = Vorlesung.ProfID</code>	Im WHERE Teil werden die Informationen aus zwei Tabellen verknüpft, üblicherweise über einen Fremdschlüssen

Abbildung 205: SQL Abfragebeispiel

Der Punkt, auf den ich hinaus möchte, ist folgender: heutzutage ist es durchaus gängig, dass man SQL nicht manuell in selbstgeschriebenen Programmcode erzeugt, sondern einen sog. Objektrelationalen Mapper einsetzt (abgekürzt ORM), der ähnlich wie der Verbund aus `DataContractSerializer`, `XmlWriter` / `XmlReader` und `FileStream` für das automatische Speichern und Laden von Objektgraphen verantwortlich ist – diesmal aber nicht in einer Datei, sondern eben in einer relationalen Datenbank. Der Einsatz eines solchen ORM ist relativ leicht, weswegen wir uns im folgenden Entity Framework anschauen, ein Objektrelationaler Mapper von Microsoft. Eine prominente Alternative zu Entity Framework in .NET ist übrigens NHibernate.

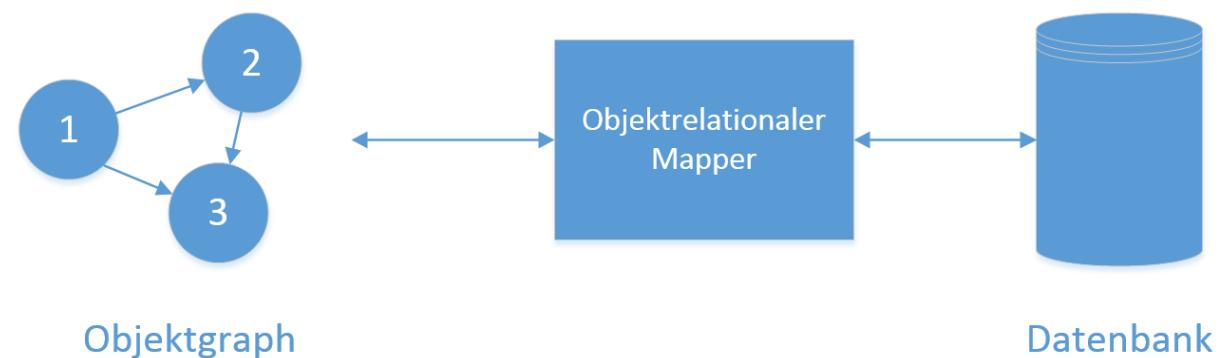


Abbildung 206: ORMs übernehmen den Zugriff auf relationale Datenbanken

5.19.2.3 Der Objektrelationale Mapper Entity Framework im Beispiel

5.19.2.3.1 Wie bekomme ich Entity Framework?

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=wj-cOmxNx5U>.

Entity Framework (abgk. EF) wird standardmäßig nicht von einem Konsolenprogramm referenziert, deshalb muss man genauso wie bei der Assembly `System.Runtime.Serialization` erst zum Projekt hinzufügen. EF war zwar ein Teil von .NET, wurde aber mittlerweile als Open Source Projekt ausgelagert, hauptsächlich um nicht mehr von den Releasezyklen von .NET abhängig zu sein.

(salopp ausgedrückt: eine neue EF Version kann genau dann veröffentlicht werden, wenn sie fertiggestellt wurde). Die noch enthaltene Version von EF in .NET ist 4.0.

Die aktuellste Version 6.1.0 kann man deshalb den Paketmanager NuGet, der seit Version 2012 automatisch in Visual Studio integriert ist, bezogen werden. NuGet ist ein Service, der verschiedene Bibliotheken über einen Web Service bereitstellt und sich auch sehr einfach für die Verwaltung verschiedener Versionen dieser Bibliotheken einsetzen lässt. Um EF über NuGet zu laden, müssen Sie nur einen Rechtklick auf die Projektmappe machen und „Manage NuGet Packages for Solution“ auswählen, wie das im folgenden Screenshot gezeigt wird.

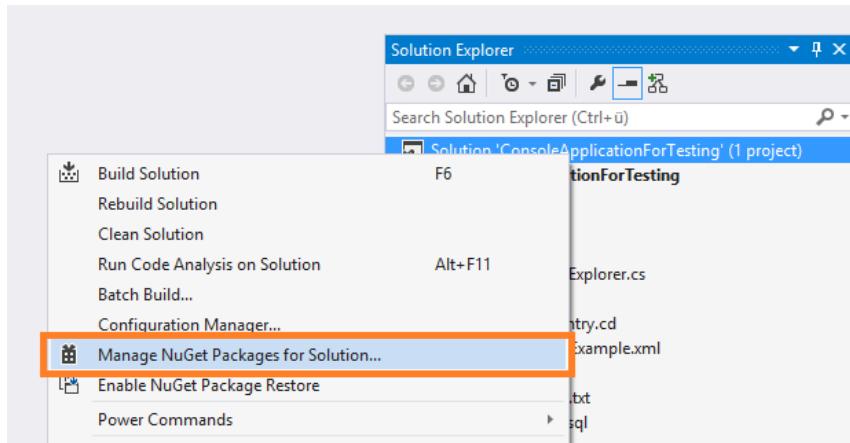


Abbildung 207: NuGet Paketmanager Dialog starten

Im NuGet-Dialogfenster können Sie nun rechts oben in der Suchleiste Entity Framework eingeben, den entsprechenden Eintrag in der Mitte auswählen und dann mit einem Klick auf „Install“ EF zu Ihrem Projekt hinzufügen.

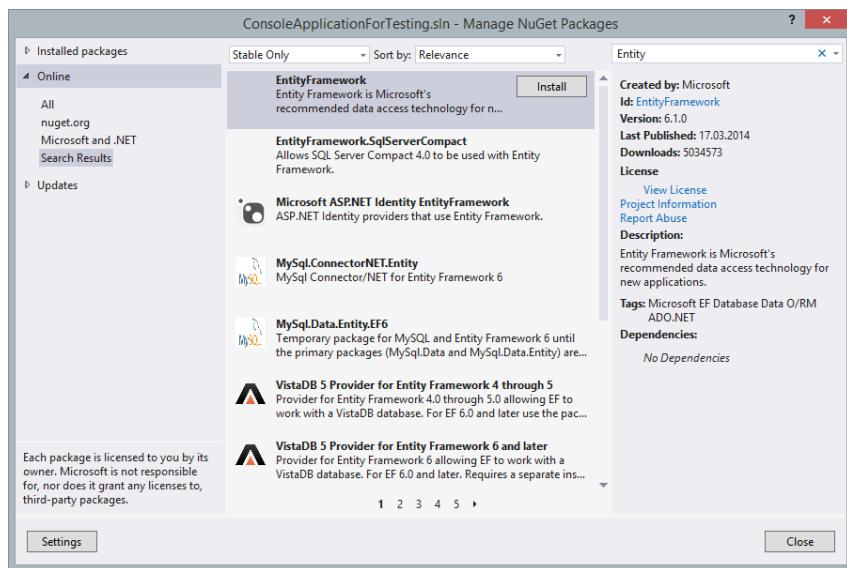


Abbildung 208: Entity Framework über NuGet installieren

NuGet wird dann die aktuellste Version von EF herunterladen und Sie nochmals in zwei Dialogen um die Bestätigung bitten, zu welchem Projekt die Bibliothek hinzugefügt werden soll und ob Sie die Lizenzbedingungen annehmen möchten. Danach müssen die beiden Entity Framework Assemblies, die in der folgenden Abbildung markiert sind, auch in Ihrem Projekt vorhanden sein:

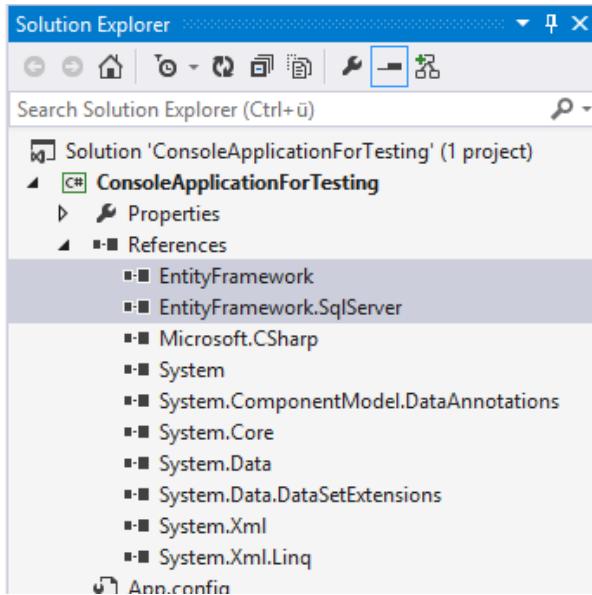


Abbildung 209: Assemblyreferenzen zu Entity Framework

Wenn neue Versionen von EF zur Verfügung gestellt werden, können Sie diese auch genauso leicht via NuGet aktualisieren. Nutzen Sie dazu den Punkt Updates ganz links im NuGet Packages Dialog.

5.19.2.3.2 Das zu speichernde Klassenmodell

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=9kYb1ls1RYY>.

Damit wir Daten in die Datenbank speichern und aus ihr laden können, brauchen wir wieder ein Objektmodell, das wir dazu einsetzen können. In diesem Beispiel möchte ich (vereinfachte) Informationen für Rechnungen darstellen – das heißt es gibt:

- Artikel, die gekauft werden können
- Rechnungen, die verschiedene Posten und einen Käufer haben
- Posten, die je einen Artikel referenzieren und die Anzahl festhalten

```

public class Invoice
{
    public int ID { get; set; }
    public string CustomerName { get; set; }
    public List<InvoiceItem> Items { get; set; }
}

public class InvoiceItem
{
    public int ID { get; set; }
    public Article Article { get; set; }
    public int Quantity { get; set; }
}

public class Article
{
    public int ID { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}

```

Abbildung 210: Eingesetzte Klassen für das EF Beispiel

In Abbildung 210 sehen Sie die vorher beschrieben Klassen. Diese sind in diesem Fall in englischer Sprache gehalten, da EF die Tabellennamen zu diesen Klassen standardmäßig aus den Klassennamen plus Plural-„S“ erstellt – man kann dies zwar anders konfigurieren, damit z.B. der Plural von „Rechnung“ tatsächlich „Rechnungen“ wird, das würde unseren Beispielcode jedoch unnötig erweitern und nicht auf die eigentliche Funktionalität eines ORM hinzielen.

In der folgenden Abbildung finden Sie diese Klassen auch in einem Klassendiagramm:

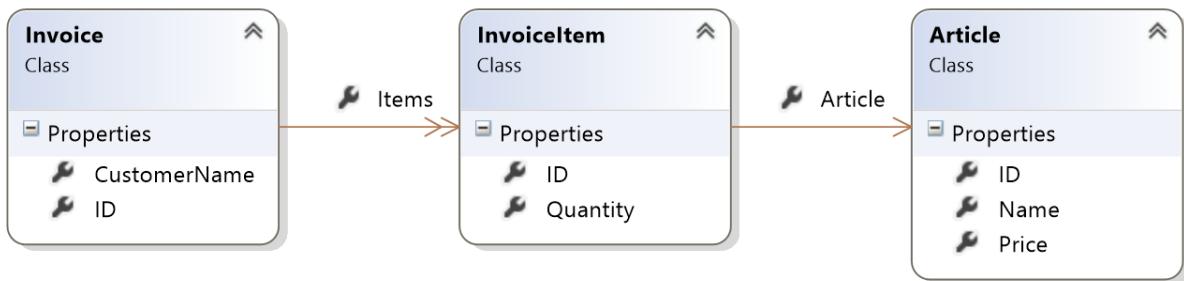


Abbildung 211: Klassendiagramm zu den im Beispiel genutzten Klassen

5.19.2.3.3 Die Klasse DbContext

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=-4Lplde1XiY>.

Damit diese Klassen mit Entity Framework eingesetzt werden können, um Sie in einer Datenbank zu persistieren, muss eine weitere sog. Datenbankkontext-Klasse erstellt werden. Diese muss von `System.Data.Entity.DbContext` ableiten und Eigenschaften vom Typ `DbSet<T>` bereitstellen. Damit wir dies genauer besprechen können, schauen wir uns dazu gleich ein Beispiel an:

Die Datenbankkontext-Klasse
muss von DbContext ableiten

```
public class InvoiceDbContext : DbContext
{
    public DbSet<Invoice> Invoices { get; set; }
    public DbSet<InvoiceItem> InvoiceItems { get; set; }
    public DbSet<Article> Articles { get; set; }
}
```

Alle DbSets, die als publike
Eigenschaften aufgeführt werden,
können direkt abgefragt werden

Abbildung 212: Datenbankkontext in Entity Framework

Über die in Abbildung 212 zu sehende Klasse `InvoiceDbContext` teilen Sie Entity Framework mit, welche Klassen in der Datenbank abgebildet werden sollen: dies machen Sie, indem Sie von `DbContext` ableiten und `DbSet<T>` Eigenschaften für eben genannte Klassen bereitstellen (in unserem Falle sind das `Invoice`, `InvoiceItem` und `Article`). Der Typ `DbSet<T>` verhält sich dabei wie einer Collection, d.h. auch er implementiert `IEnumerable<T>` und besitzt Methoden wie `Add` und `Remove`, um Elemente zu einem `DbSet<T>` hinzuzufügen bzw. zu entfernen.

`DbSet<T>` Eigenschaften haben zwei Aufgaben: einerseits gelten Sie für Entity Framework als Einstiegspunkt um festzustellen, welche Klassen überhaupt in einer Datenbank abgebildet werden sollen (EF nutzt dazu intern Reflection). Andererseits können wir als Programmierer `DbSet<T>` Eigenschaften nutzen, um Datenbankinhalte abzufragen (und ggf. zu verändern).

Die wichtigste Methode, die von `DbContext` geerbt wird, ist `SaveChanges` – mit dieser kann man nämlich die Änderungen, die man an den `DbSet<T>` Objekten eines Datenbankkontexts vorgenommen hat, auf die Datenbank übertragen. Wie diese beiden Sachen im Verbund eingesetzt werden, sehen wir uns im folgenden Beispiel an:

```

private static void Main()
{
    using (var databaseContext = new InvoiceDbContext())
    {
        var soap = new Article { Name = "Soap", Price = 0.33m };
        var wlanCable = new Article { Name = "Wireless LAN Cable", Price = 9.99m };
        var fakeMustache = new Article { Name = "Fake Mustache", Price = 5.99m };

        databaseContext.Articles.AddRange(new[]
        {
            soap,
            wlanCable,
            fakeMustache
        });

        Mit AddRange können zu einem DbSet<T> mehrere
        Objekte auf einmal hinzugefügt werden

        var invoice = new Invoice
        {
            CustomerName = "Kenny Pflug",
            Items = new List<InvoiceItem>
            {
                new InvoiceItem { Article = soap, Quantity = 2 },
                new InvoiceItem { Article = fakeMustache, Quantity = 1 }
            }
        };
        databaseContext.Invoices.Add(invoice); — Mit Add kann ein einzelnes Objekt zu einem DbSet<T>
        databaseContext.SaveChanges(); — hinzugefügt werden
    }
}

```

Mit SaveChanges werden vom Kontext alle Änderungen der verschiedenen DbSets in die Datenbank übertragen

Abbildung 213: Die Klasse `InvoiceDbContext` im Einsatz

In Abbildung 213 sehen Sie, zunächst eine Instanz der Klasse `InvoiceDbContext` innerhalb eines `using` Blocks erstellt wird – dies sorgt dafür, dass am Ende dieses Blocks auf dem Kontextobjekt automatisch `Dispose` aufgerufen wird, um den Kontext und damit die Datenbankverbindung zu schließen (auch bei Datenbankverbindungen handelt es sich um native Ressourcen, die so schnell wie möglich geschlossen werden sollten). Dazu implementiert natürlich `DbContext` auch `IDisposable`.

Innerhalb des `using` Blocks werden die drei Artikel Seife, WLAN Kabel und künstlicher Schnurrbart erstellt und zum `DbSet Articles` hinzugefügt. Danach wird eine neue Rechnung mit zwei Posten erstellt für Seife und den Schnurrbart, wobei nicht nur eine, sondern zwei Seifen gekauft werden. Auch diese Rechnung wird wieder im entsprechenden `DbSet Invoices` festgehalten. Am Ende werden all diese Änderungen über `databaseContext.SaveChanges` in die Datenbank übertragen.

Bitte beachten Sie hierbei:

- Wir mussten die beiden `InvoiceItem` Objekte, die wir zur Rechnung hinzugefügt haben, nicht manuell zum `DbSet InvoiceItems` hinzufügen – dies wird automatisch gemacht, wenn `SaveChanges` aufgerufen wird.
- Da `DbSet<T>` das Interface `IEnumerable<T>` implementiert, können Sie natürlich auch LINQ einsetzen, um bestimmte Objekte aus diesen abzufragen. Ihre LINQ Abfragen werden dann auf SQL Abfragen projiziert – wie das genau funktioniert, müssen Sie allerdings nicht wissen. Genaueres zum Einsatz von LINQ finden Sie im Video zu diesem Abschnitt.

5.19.2.3.4 Moment – in welches Datenbanksystem schreibe ich eigentlich gerade?

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=izFt1N09h2s>.

Wenn wir den Code aus dem vorherigen Abschnitt ausführen, wird tatsächlich eine Datenbankverbindung aufgebaut und die Daten persistiert – aber welchen Datenbankserver sprechen wir eigentlich an? Die Antwort heißt Microsoft LocalDB – dieser Datenbankserver wird automatisch mit Visual Studio 2013 installiert und kann für Entwickler zu Testzwecken eingesetzt werden. Der Vorteil dieses Servers ist, dass sie nur aktiv ist, wenn Datenbankverbindungen zu ihr aufgebaut werden – ansonsten ist sie inaktiv und verbraucht keine Systemressourcen (was natürlich gut ist für Entwicklungsrechner). Andere Datenbankserver, die für den produktiven Einsatz genutzt werden, sind SQL Server, Oracle Database, PostgreSQL oder MySQL.

Als wir Entity Framework via NuGet zum Projekt hinzugefügt haben, wurde ebenfalls die Datei App.config angepasst, welche jetzt wie folgt aussieht:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkId=237468 -->
    <section name="entityFramework"
      type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
            EntityFramework, Version=6.0.0.0,
            Culture=neutral,
            PublicKeyToken=b77a5c561934e089"
      requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory, EntityFramework">
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
    <providers>
      <provider invariantName="System.Data.SqlClient"
        type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
</configuration>
```

Abbildung 214: Die Datei App.config nach der Referenzierung von Entity Framework via NuGet

Wie wir sehen, ist App.config eine XML-Datei, mit der eine Applikation konfiguriert werden kann. Hierzu wurde für EF eingetragen, dass LocalDB als Ziel ausgewählt werden soll, was man im Knoten **defaultConnectionFactory** sehen kann. Sie müssen den Inhalt dieser Datei nicht komplett verstehen, Sie sollen aber wissen, dass verschiedene Frameworkklassen wie **DbContext** nicht nur über Parameter im Code, sondern auch über die Datei App.config konfiguriert werden können.

5.19.2.3.5 Verbindung zum Datenbanksystem in Visual Studio aufnehmen

Sie können über Visual Studio auch eine Verbindung zu LocalDB herstellen und die dort verwalteten Datenbanken betrachten. Dazu müssen Sie das Fenster Server Explorer öffnen, sofern dieses bei Ihnen noch nicht offen ist. Den Server Explorer finden Sie in der Menüleiste und Ansicht -> Server Explorer, wie im nächsten Bild beschrieben:

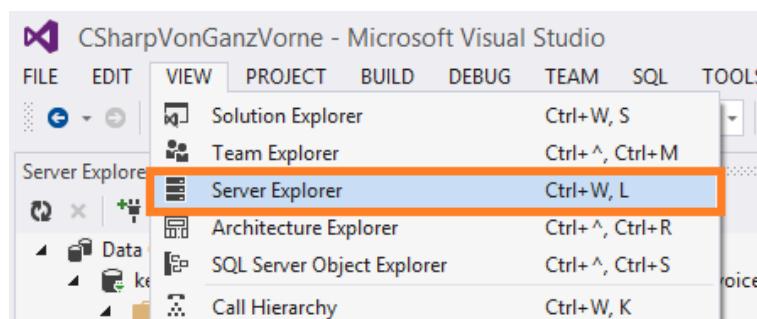


Abbildung 215: Server Explorer über das Menü öffnen

Innerhalb des Server Explorers können Sie Verbindungen zu verschiedenen Servern aufnehmen, u.a. eben auch Datenbankservern. Dazu machen Sie einen Rechtsklick auf Verbindungen und fügen eine neue hinzu, wie im nächsten Bild gezeigt:

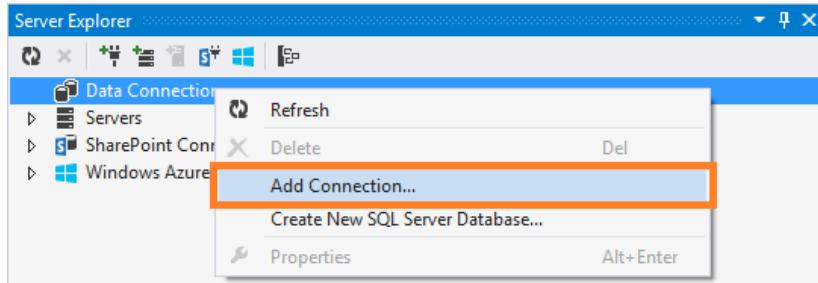


Abbildung 216: Eine Datenbankverbindung im Server Explorer hinzufügen

Im Anschluss sehen Sie folgenden Dialog, bei dem Sie als Datenquelle Microsoft SQL Server (SqlClient) auswählen. Unter Servername geben Sie (localdb)\v11.0 ein (es kann auch v12.0 sein, je nachdem, welche Version von LocalDB bei Ihnen installiert ist). Als Datenbanknamen geben Sie den vollqualifizierten Namen ihrer Klasse an, die von `DbContext` erbt – in meinem Fall ist das `CSharpVonGanzVorne.Konsole.InvoiceDbContext`. Standardmäßig heißt der Datenbankname zu einem Kontext genauso wie diese Klasse, wobei dies natürlich noch konfiguriert werden kann. In Abbildung 217 sehen Sie, wie Sie den eben erwähnten Dialog ausfüllen müssen.

Ein Datenbankserver enthält üblicherweise bereits Datenbanken, bei SQL Server sind das Master, Model, msdb usw. Manipulieren oder Löschen Sie diese auf keinen Fall, da dadurch die Integrität des Datenbankservers sehr wahrscheinlich zerstört wird, was nur durch eine Neuinstallation revidiert werden kann.

Sobald Sie die Verbindung aufgebaut haben, können Sie im Server Explorer die Datenbank betrachten. Schauen Sie dazu innerhalb des Knoten Tables: hier sehen Sie die Tabellen Articles, InvoiceItems und Invoices, die gemäß unseres Klassenmodels von Entity Framework erstellt wurden, um Objekte in einer relationalen Datenbank persistieren zu können. In Abbildung 218 sehen Sie die Inhalte der Tabelle Articles.

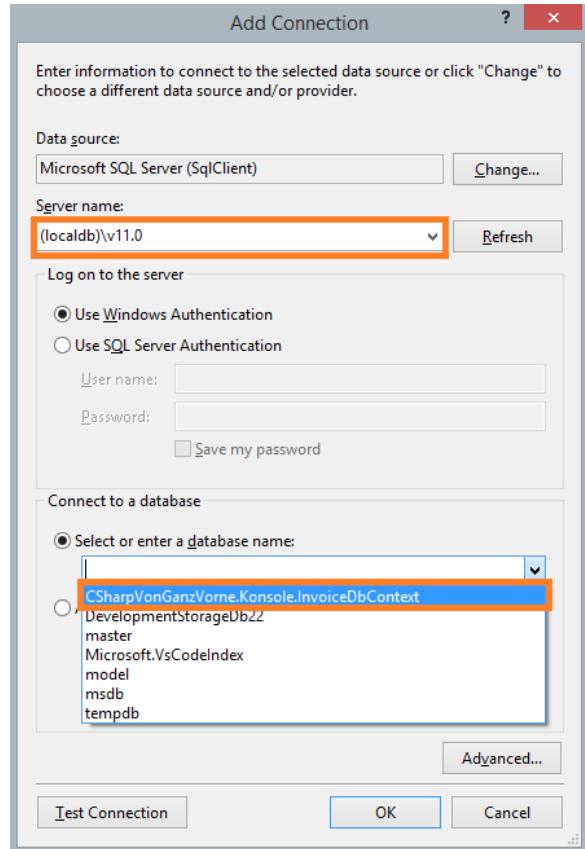


Abbildung 217: Verbindungsdialogeinstellungen für LocalDB

ID	Name	Price
1	Soap	0,33
2	Fake Mustache	5,99
3	Wireless LAN C...	9,99
*	NULL	NULL

Abbildung 218: Tabelle Artikel in der von Entity Framework erstellten Datenbank

Wir sehen, dass der ORM für die ID-Vergabe verantwortlich ist. Ansonsten finden wir aber genau dieselben Daten wieder, die wir auch im Code erstellt haben.

5.19.2.3.6 Zusammenfassung für Objektrelationale Mapper

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=BPq7aYOc7Xg>.

Wir haben in den letzten kurzen Beispielen gesehen, wie man einen ORM einsetzen kann, um Objektgraphen in relationalen Datenbanken zu speichern. Im Prinzip ist das Vorgehen ähnlich wie bei der von uns geschriebenen Klasse **PersistenzService**, auch wenn sich die API etwas unterscheidet: wir übergeben die entsprechenden Objekte an die **DbSet<T>** Collections und rufen zum Abschluss **SaveChanges** auf dem Datenbankkontext auf. Zum Laden nutzt man einfach die **DbSet<T>** Collections, bspw. auch, indem man LINQ Abfragen gegen diese ausführt. Über die

Interna, also bspw. das Öffnen von Datenbankverbindungen oder das Erstellen und Absetzen von SQL Befehlen, müssen wir uns keine Gedanken machen.

Es gibt natürlich noch einige andere Aspekte, die wir beachten müssen, wie bspw. Transaktionen, aber auf diese Themen werden Sie näher in der Vorlesung Datenbanken eingehen. Aus diesem Teil sollten Sie mitnehmen, dass der Zugriff auf relationale Datenbanken relativ schnell machbar ist über ORMs – für Entity Framework haben wir auch gesehen, wie die API dazu konkret aussieht.

Hinter der Klasse DbSet<T> steckt eigentlich das Repository Pattern, hinter DbContext das Unit of Work Pattern. Diese beiden Designmuster werden uns zu einem späteren Zeitpunkt nochmals begegnen – mit ihnen kann man nämlich von der Persistenzschicht in anderen Teilen abstrahieren (salopp ausgedrückt: Klassen in anderen Schichten ist es egal, ob in einer Datei oder Datenbank gespeichert wird – Hauptsache ist, dass die entsprechende Logik über ein gemeinsames Interface aufgerufen werden kann).

5.19.3 Die Unterscheidung zwischen Entities und Value Objects

Das Video zu diesem Abschnitt finden Sie unter https://www.youtube.com/watch?v=2C9_cOh9Avk.

Zum Abschluss für Persistenz werden wir noch zwei Typen von Objekten voneinander unterscheiden, die für das Schreiben von Applikationen, die auf Persistenz setzen, beachten sollten.

Wenn ein Objektgraph gespeichert und zu einem späteren Zeitpunkt wieder geladen wird, dann sitzen die Objekte in einem anderen Speicherbereich und können nicht einfach über ihre Speicheradresse auf Gleichheit überprüft werden. Damit dieses Problem gelöst wird, gibt es die Möglichkeit, seine (Daten-)Klassen als Entities und Value Objects zu modellieren.

Entities und Value Objects werden wie folgt unterschieden:

- Eine Entity ist ein Objekt, das einen eindeutigen Identifier (ID) bekommt, über den es identifiziert werden kann. Alle anderen Eigenschaften dieses Objekts können sich zu ihrer Lebenszeit ändern, der Identifier muss jedoch nach dem Erstellvorgang oder spätestens nach dem ersten Speichervorgang eindeutig bleiben. Als IDs wird üblicherweise der Datentyp **int** oder auch **Guid** (Globally Unique Identifier) gewählt, wobei letzter eine zufällig erstellte 128 Bit Ganzzahl ist, die üblicherweise in 32 Hexadezimalzeichen dargestellt wird. Wenn zwei Entities miteinander verglichen werden, dann gelten sie als gleich, wenn ihre IDs übereinstimmen.
- Ein Value Object hingegen ist ein Objekt, das sich über seine Werte identifiziert – genauso, wie wir es bereits in Abschnitt 5.16.5 angesprochen haben. Value Objects sind unveränderlich (immutable), sodass bspw. immer ein neues Objekt der Klasse **Farbe** erstellt werden muss, wenn eine neue Farbe gebraucht wird. Zwei Value Objects gelten als gleich, wenn alle ihre Felder identische Werte aufweisen.

Wichtig ist dabei, dass nicht nur Entities Value Objects referenzieren können, sondern auch umgekehrt ist diese Beziehung möglich. Weitere Informationen zu diesem Thema werden Sie in der Vorlesung Software Engineering sowie im genialen Buch Domain Driven Design von Eric Evans erhalten.

In der folgenden Abbildung sehen Sie als Beispiel die Klasse **Kunde**, die eine Entity repräsentiert:

```
public class Kunde
{
    public int ID { get; set; } ----- Kunde hat eine ID und ist damit eine Entity

    public string Name { get; set; }

    public int Age { get; set; }

    public Adresse Adresse { get; set; }
}
```

Alle anderen Eigenschaften oder Felder können sich zur Lebenszeit eines Kundenobjekts ändern, wir als Entwickler müssen jedoch sicherstellen, dass die ID unverändert bleibt.

Abbildung 219: Die Klasse Kunde als Beispiel für eine Entity

Beispiele für Value Objects sind bspw. die Klassen **string**, **Adresse** oder **Farbe**, die wir uns alle bereits angesehen haben. Wichtig für Sie ist, dass Sie mitnehmen, wie man zwischen Entities und Value Objects unterscheidet und warum man diese Unterscheidung trifft.

5.20 Oberflächenprogrammierung mit WPF in .NET

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=fh58U4UatsU>.

Als nächsten großen Punkt werden wir uns die Oberflächenprogrammierung in .NET anschauen und dazu die Welt der Konsolenapplikationen verlassen. Zu diesem Zweck werfen wir einen Blick auf die Windows Presentation Foundation (WPF), das aktuellste Framework von Microsoft für den Desktopbereich, das eingesetzt werden kann, um Oberflächen mit bspw. Buttons, Checkboxen, Textboxen und Listboxen zu erstellen. Windows Forms ist der Vorgänger von WPF, hat aber bereits ein deutliches Alter und unterstützt mittlerweile nicht mehr gängige Designmuster, die einem bei WPF viel Code ersparen (v.a. Data Binding und MVVM sind hier bei WPF zu erwähnen). Hinzu kommt: Windows 8 Store Apps (die unter Modern UI laufen) und Windows Phone Apps werden alle mit XAML und ähnlichen Frameworkkomponenten geschrieben. WinRT und Silverlight, die beiden Frameworks für die beiden letztgenannten Plattformen, können als Subsets von .NET angesehen werden – d.h. die APIs sind in weiten Teilen gleich, lediglich weniger der Klassen von .NET stehen zur Verfügung.

Eine Alternative zu Desktopapplikationen oder Apps sind Websites – diese werden heutzutage üblicherweise mit HTML5 / JavaScript geschrieben, wobei üblicherweise auch ein größerer Teil Servercode benötigt wird, der im Microsoftumfeld bspw. mit ASP.NET (MVC) geschrieben werden kann. Weiter werden wir auf diesen Applikationstyp nicht eingehen, sondern uns auf WPF in den kommenden Abschnitten beschränken.

5.20.1 Hello World mit WPF

Bevor wir uns in die Details von WPF begeben, möchten wir ein Hello World Beispiel mit WPF machen, um zu sehen, wie diese Projekte aufgebaut sind und was man beim Erstellen von User Interface Komponenten beachten muss.

5.20.1.1 Ein WPF Projekt erstellen

Zunächst möchten wir uns eine neue Projektmappe mit einem WPF Projekt erstellen. Dazu wählen Sie bitte den Menüleisteneintrag Datei -> Neu -> Projekt... aus, damit der Projektdialog erscheint. Innerhalb dieses Dialogs wählen Sie anstatt eines Konsolenprojekts ein WPF Projekt aus und geben Projektmappe sowie Projekt einen passenden Namen, bspw. so wie es im nächsten Bild gezeigt wird:

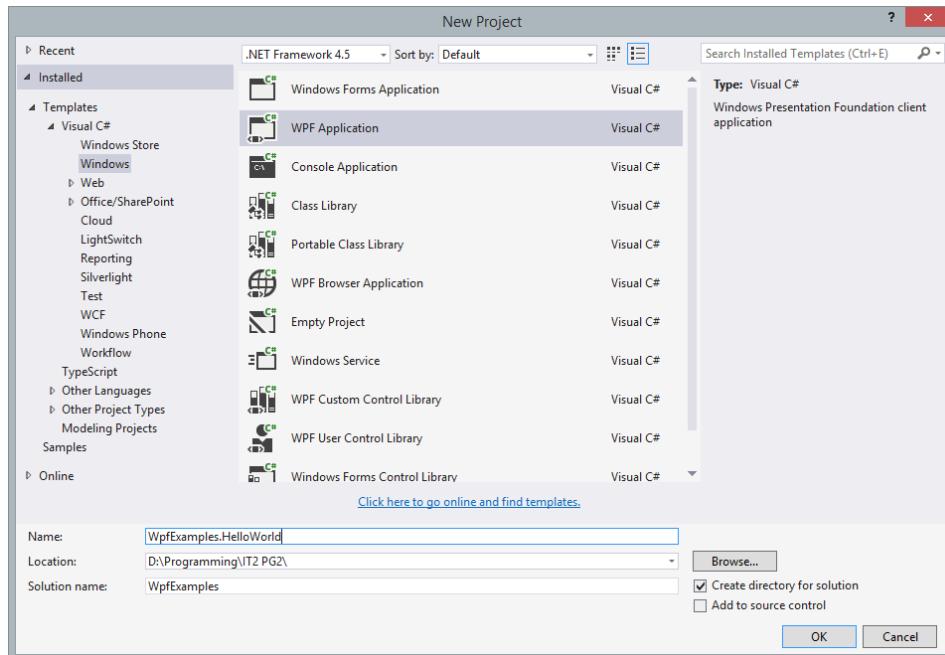


Abbildung 220: Ein WPF Projekt erstellen über den Projektdialog

Wenn das geschehen ist, erhalten wir ein Projekt, in dem einige Dateien mehr enthalten sind als bei einem Konsolenprojekt:

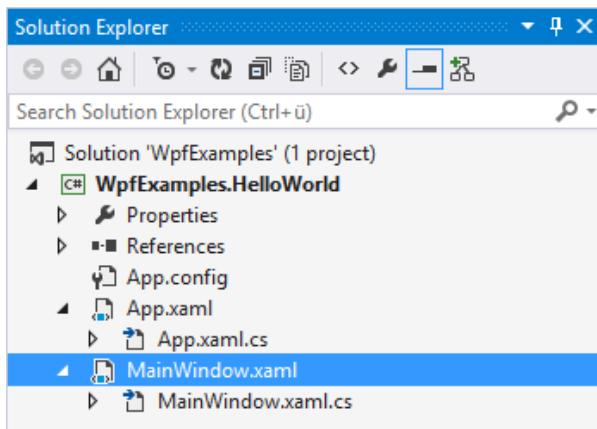


Abbildung 221: Standardstruktur eines WPF Projekts

Gehen wir diese Dateien schrittweise durch:

- App.config ist genauso wie in Konsolenprogrammen vorhanden und kann verwendet werden, um bestimmte Komponenten via XML zu konfigurieren (wie wir es bei Entity Framework gesehen haben).
- MainWindow.xaml und MainWindow.xaml.cs sind zwei Dateien, welche beide die partielle Klasse **MainWindow** beschreiben, welche von der Klasse **System.Windows.Window** ableitet. Diese stellt das Hauptfenster dar, welches angezeigt wird, wenn eine WPF Applikation gestartet wird. Im Moment hat dieses Fenster noch keinen Inhalt.
- App.xaml und App.xaml.cs sind ebenfalls zwei Dateien, die eine partielle Klasse beschreiben, in diesem Fall aber **App**, welche von **System.Windows.Application** ableitet. Sie repräsentiert, wie der Name schon sagt, das Programm und ist für den Start- und Schließvorgang sowie das Bereitstellen von Applikationsweiten Ressourcen hauptsächlich zuständig.

Die Dateien App.xaml und MainWindow.xaml enthalten keinen C# Code, sondern einen XML-Dialekt namens XAML (Extensible Application Markup Language), denn Oberflächen werden in WPF nicht über C# Code aufgebaut, sondern in XAML. Warum XAML als Serialisierungsformat gewählt wurde, werden wir später besprechen.

5.20.1.2 XAML und Code-Behind-Datei modifizieren

Öffnen Sie jetzt die Datei MainWindow.xaml (sofern sie nicht schon geöffnet ist) und modifizieren Sie den XAML Code wie folgt:

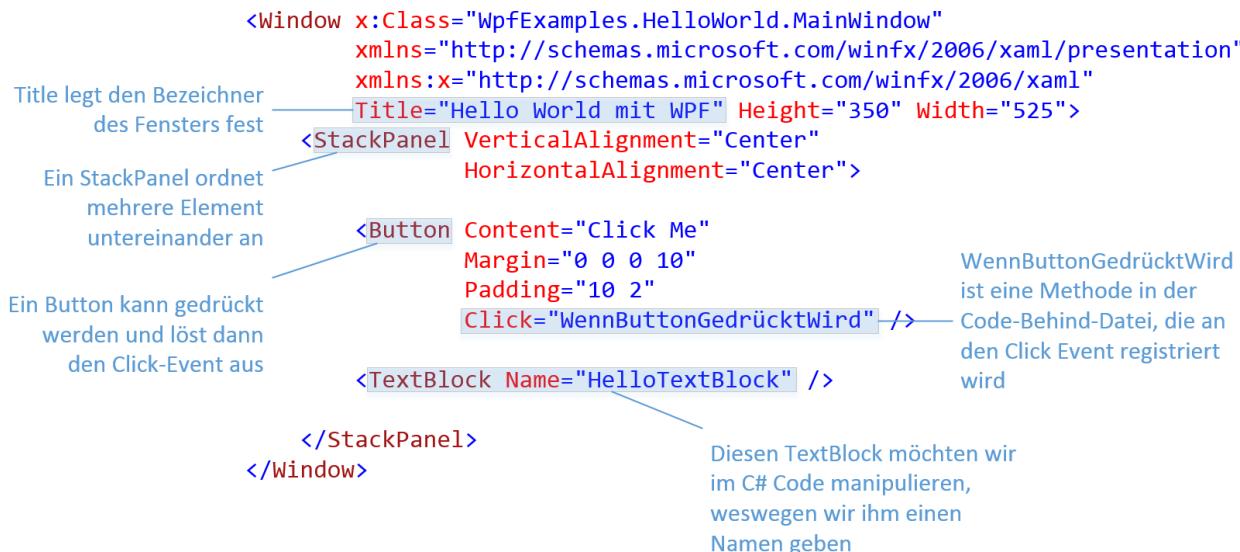


Abbildung 222: MainWindow.xaml im Hello WPF Beispiel

In Abbildung 222 passieren im Wesentlichen folgende Dinge gemacht:

- Beim Knoten **Window** wurde der Wert des Attributs **Title** zu **Hello World mit WPF** geändert
- Der Subknoten **Grid** von **Window** wurde durch ein **StackPanel** ersetzt. Mit den Werten für die Attribute **VerticalAlignment** und **HorizontalAlignment** wurde festgelegt, dass das **StackPanel** in der Mitte des Elternelements **Window** platziert werden soll. Das **StackPanel** selbst ordnet alle seine Kindelemente standardmäßig untereinander an.
- Innerhalb des **StackPanel** Knotens werden ein **Button** und ein **TextBlock** gelegt.
- Beim **Button** legt man mit **Content** fest, was im **Button** stehen soll, **Margin** setzt den Abstand außerhalb des Buttons zu anderen Elementen fest, **Padding** den Abstand innerhalb des Buttonrahmens zum Content.
- Diese Abstände (engl. Thickness) können über mehrere Schreibweisen angegeben werden:
 - **"0 0 0 10"** sagt aus, dass der linke, obere und rechte Abstand (in genau dieser Reihenfolge) jeweils 0 ist, der untere Abstand beträgt 10.
 - **"10 2"** sagt aus, dass der linke und rechte Abstand jeweils 10 sowie der obere und untere Abstand jeweils 2 betragen soll.
 - **"15"** würde aussagen, dass alle vier Abstände (links, oben, rechts und unten) jeweils 15 betragen würden. Das ist im obigen Beispiel allerdings nicht zu sehen.
- Ebenfalls wird der **Click** Event des Buttons mit einem Delegate verknüpft, der auf die Methode **WennButtonGedrücktWird** zeigt. Diese Methode liegt in der Code-Behind-Datei.
- Dem **TextBlock** haben wir ebenfalls einen Namen gegeben über das Attribut **Name** gegeben, damit wir ihn aus der Code-Behind-Datei ansprechen können.

Genau den eben erwähnten **TextBlock** wollen wir jetzt modifizieren, wenn der Button geklickt wird: dazu sehen wir uns die Code-Behind-Datei MainWindow.xaml.cs an und fügen dieser die Methode **WennButtonGedrücktWird** hinzu:

```

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

    private void WennButtonGedrücktWird(object sender, RoutedEventArgs e)
    {
        HelloWorldTextBlock.Text = "Hello Wpf!";
    }
}

Über den vergebenen Namen können
wir in der Code-Behind-Datei auf den
TextBlock zugreifen

```

Abbildung 223: MainWindow.xaml.cs im Hello WPF Beispiel

Wie in Abbildung 223 zu sehen ist, besitzt `MainWindow` einen Konstruktor, in dem die Methode `InitializeComponent` aufgerufen wird, und die eben erwähnte Methode `WennButtonGedrücktWird`, die mit dem `Click` Event des Buttons in der XAML Datei verknüpft wurde. Diese wird folglich jedes Mal vom Button aufgerufen, wenn der Nutzer darauf klickt (Hollywood-Prinzip). In eben genannter Methode setzen wir programmatisch die `Text` Property des `TextBlock` auf `"Hello WPF!"`, wobei wir den `TextBlock` über den Namen, den wir ihm in der XAML Datei gegeben haben, ansprechen können.

Wenn Sie diesen Code ausführen, sehen Sie das Hauptfenster mit einem Button. Wenn dieser geklickt wird, erscheint der Text.

5.20.1.3 Was haben wir gelernt?

Am Hello WPF Beispiel konnten wir folgende Dinge sehen:

- WPF Applikationen sind standardmäßig deutlich komplexer strukturiert als Konsolenapplikationen. Sie umfassen von Beginn an die Klassen `App` und `MainWindow`.
- Das Aussehen der Oberfläche wird nicht in C# Code, sondern mit XAML, ein XML-Dialekt, festgelegt. In XAML können wir Eigenschaften und Events für Elemente setzen – letztere entsprechen normalen C# Klassen.
- XAML ist natürlich ein Serialisierungsformat, das statisch einen Zustand beschreibt. Wenn wir Methoden aus der Code-Behind-Datei ausführen wollen, können wir diese allerdings in XAML an die Events beliebiger Elemente registrieren.
- In der Code-Behind-Datei können wir auf alle Elemente zugreifen, denen in der XAML Datei ein Name gegeben wurde.

5.20.2 Die verschiedenen Elemente in WPF

Nachdem wir jetzt einen groben Eindruck haben, wie WPF Applikationen aufgebaut sind, möchten wir einen Eindruck bekommen, mit welchen WPF Elementen wir eine XAML Datei ausstatten können. Bitte beachten Sie, dass die kommenden Listen nicht vollständig sind. WPF umfasst eine große Anzahl an Elementen, die hier im Script nicht alle aufgezählt werden. Die verschiedenen Elemente sind im Übrigen durch eine große Vererbungshierarchie verbunden, da WPF ein Framework ist, das sehr stark auf Vererbung setzt.

Wichtig: wenn Sie in der MSDN nach weiteren Informationen zu den folgenden Elementen suchen, passen Sie auf, dass Sie die richtige Dokumentationsseite ansteuern. Die nachfolgenden Klassen

existieren häufig mit gleichen Namen in Windows Forms, ASP.NET und WPF – achten Sie also darauf, dass ihre Klasse im Namespace `System.Windows` oder einem Subnamespace davon liegt, bspw. `System.Windows.Controls` (nicht in `System.Windows.Forms` oder `System.Web.UI`)

5.20.2.1 Visuelle Elemente

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=gA2-xW2ReDE>.

Unter visuellen Elementen versteht man einzelne Oberflächenelemente, die zwar sichtbar sind, aber vom Nutzer standardmäßig nicht manipulierbar sind, wie bspw. der `TextBlock`. Sie können natürlich an die Events aller dieser Elemente Methoden registrieren und so eine gewisse Interaktivität selber herstellen (wie wir später sehen werden). Die allgemeine Basisklasse für visuelle Elemente ist `FrameworkElement`. Wichtige visuelle Elemente sind:

Klasse	Bemerkung
<code>Border</code>	Repräsentiert einen Rand / Rahmen um ein anderes Element, das über die Eigenschaft <code>Child</code> gesetzt werden kann. Wichtige weitere Eigenschaften sind <code>BorderBrush</code> , <code>BorderThickness</code> , <code>Background</code> und <code>CornerRadius</code> .
<code>Image</code>	Zeigt ein Bild an, dass in einem üblichen Bildformat (bspw. bmp, jpg, png) vorhanden ist. Dieses Bild kann über die Eigenschaft <code>Source</code> gesetzt werden.
<code>Popup</code>	Repräsentiert ein Popup, das über dem aktuellen Fenster schwebt und beliebigen Inhalt anzeigt, der über die Eigenschaft <code>Child</code> festgelegt werden kann. Über die Eigenschaft <code>IsOpen</code> kann das Popup angezeigt oder geschlossen werden.
<code>TextBlock</code>	Repräsentiert ein Stück Text, das ein- oder mehrzeilig im User Interface sichtbar ist, vom User allerdings nicht manipuliert werden kann. Die wichtigste Eigenschaft ist <code>Text</code> , mit der der angezeigte <code>string</code> gesetzt werden kann.

Abbildung 224: Wichtige visuelle Elemente in WPF

5.20.2.2 Controls

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=6w4O0654rFQ>.

Controls sind visuelle Elemente, die standardmäßig vom Nutzer manipuliert werden können, wie bspw. ein `Button` oder eine `TextBox`. In den meisten Fällen werden diese für Eingabezwecke eingesetzt. Die allgemeine Basisklasse für Controls ist `Control`. Wichtige Controls sind:

Klasse	Bemerkung
<code>Button</code>	Repräsentiert einen Knopf, der gedrückt werden kann. Wichtige Eigenschaften sind <code>Content</code> für den Inhalt des Buttons und der <code>Click</code> Event, der ausgelöst wird, wenn der Nutzer auf den Button klickt.
<code>Calendar</code> oder <code>DatePicker</code>	Repräsentiert einen Kalender, mit dem ein Nutzer ein Datum auswählen kann.
<code>CheckBox</code>	Repräsentiert ein Kästchen, dass der Nutzer entweder an- oder abwählen kann. Wichtigste Eigenschaft ist <code>.IsChecked</code> , mit der festgestellt werden kann, ob die Checkbox ausgewählt wurde oder nicht. Sie wird üblicherweise verwendet, um Optionen an- oder abzuschalten.
<code>Label</code>	Wie ein <code>TextBlock</code> , nur dass ein <code>Label</code> zusätzlich die Möglichkeit bietet, über einen Tastatur-Shortcut ein mit dem Label verlinktes Element in den Fokus zu rücken. Das kann durch die Eigenschaft <code>Target</code> gemacht werden, den Inhalt des Labels setzt man über <code>Content</code> (nicht <code>Text</code> wie bei <code>TextBlock</code>).

PasswordBox	Repräsentiert eine TextBox, in die der Nutzer ein Passwort eingeben kann. Wichtigste Eigenschaft ist <code>SecurePassword</code> , mit der man einen <code>SecureString</code> erhält, der nicht einfach aus dem Speicher ausgelesen werden kann (bspw. über einen Debugger).
ProgressBar	Repräsentiert eine Leiste, in der ein Fortschritt angezeigt, üblicherweise der einer Aktion wie bspw. Kopieren. Wichtige Eigenschaften sind <code>Value</code> , <code>Minimum</code> und <code>Maximum</code> .
RadioButton	Ein RadioButton verhält sich ähnlich wie eine CheckBox , allerdings liegen im Normalfall mehrere Radio Buttons nebeneinander – wenn einer dieser ausgewählt wird, werden alle anderen danebenliegenden automatisch deselektiert. Sie werden üblicherweise dann verwendet, wenn der Nutzer aus mehreren Optionen eine auswählen soll.
ScrollViewer	Repräsentiert einen vertikal und / oder horizontal scrollbaren Bereich, der beliebigen Inhalt anzeigen kann, der über die Eigenschaft <code>Content</code> gesetzt wird. Wird der Inhalt zu groß für den vorgegebenen Bereich, werden automatisch die entsprechenden Scrollbalken eingeblendet, mit denen der Nutzer den Inhalt verschieben kann.
TextBox	Repräsentiert eine Eingabebox, in die der Nutzer Text ein- oder mehrzeilig eingeben kann. Wichtigste Eigenschaft ist <code>Text</code> , mit der der Text ausgelesen oder auch gesetzt werden kann.
ToolTip	Repräsentiert ein Control, dass ein Popup erzeugt und darin Informationen anzeigt, die zu einem bestimmten visuellen Element passen, wenn der Nutzer mit der Maus über letzteres fährt.

Abbildung 225: Wichtige Controls in WPF

5.20.2.3 Panels

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=Czu4uE6ggYs>.

Panels werden eingesetzt, um andere WPF Elemente anzugeordnen. Dabei gibt es auch hier verschiedene Klassen, welche die Anordnung nach unterschiedlichen Prinzipien vornehmen. Häufig können die Kindelemente über sog. Attached Properties (etwas) Einfluss darauf nehmen, wie sie angeordnet werden sollen (dazu später mehr). Die gemeinsame Basisklasse aller Panels ist [Panel](#). Alle Panels verfügen über die Eigenschaft `Children`, eine Collection, die alle Kindelemente enthält. Die am häufigsten eingesetzten Panels sind [StackPanel](#) und [Grid](#).

Klasse	Bemerkung
Canvas	Repräsentiert ein Panel, das als einziges seiner Art seine Kindelemente nicht automatisch anordnet. Jedes Kindelement muss sich über die Attached Properties <code>Canvas.Top</code> und <code>Canvas.Left</code> (wahlweise auch <code>Canvas.Bottom</code> und <code>Canvas.Right</code>) selbst positionieren. Dieses Panel sollte deshalb nur in Ausnahmefällen eingesetzt werden – nutzen Sie stattdessen die anderen Panels, die ihre Kindelemente automatisch anordnen.
DockPanel	Ein DockPanel ordnet seine Kindelemente an, indem jedes an eine bestimmte Ecke angedockt wird. Das letzte Element ist standardmäßig im verbleibenden Raum zu finden. Über die Attached Property <code>DockPanel.Dock</code> können die Kindelemente jeweils festlegen, an welche Seite sie angedockt werden sollen.
Grid	Das Grid ordnet seine Elemente in einem Raster an. Dazu werden üblicherweise Spalten und Zeilen definiert über die Eigenschaften <code>ColumnDefinitions</code> und <code>RowDefinitions</code> , welche die zur Verfügung stehenden Zellen in einem Grid beschreiben. Mit den

	Attached Properties <code>Grid.Column</code> , <code>Grid.ColumnSpan</code> , <code>Grid.Row</code> und <code>Grid.RowSpan</code> kann festgelegt werden, in welcher Zelle ein Kindelement beginnt und über wie viele Zellen es sich erstreckt.
<code>UniformGrid</code>	Repräsentiert ein Raster, dessen Zellen alle die gleiche Größe haben. Wichtig: diese Klasse bietet nicht die gleiche API wie <code>Grid</code> . Mit den Eigenschaften <code>Rows</code> und <code>Columns</code> kann man festlegen, wie viele Zeilen und Spalten im <code>UniformGrid</code> enthalten sein sollen. Man kann aber nicht wie bei <code>Grid</code> mit Attached Properties bestimmen, in welchen Zellen die Elemente genau angeordnet werden sollen. Diese werden einfach nach der gegebenen Reihenfolge im <code>UniformGrid</code> platziert.
<code>StackPanel</code>	Das <code>StackPanel</code> ordnet seine Kindelemente entweder untereinander oder nebeneinander an. Dies kann über die Eigenschaft <code>Orientation</code> gesteuert werden.
<code>VirtualizingStackPanel</code>	Dieses Panel verhält sich wie das eben angesprochene <code>StackPanel</code> , allerdings werden die Elemente, die gerade nicht sichtbar sind, auch nicht gerendert (der Fachbegriff hierzu ist Virtualisierung). Es wird vorrangig im Zusammenhang mit ItemsControls genutzt, die wir im kommenden Abschnitt kennenlernen werden.
<code>WrapPanel</code>	Ein <code>WrapPanel</code> verhält sich ähnlich wie ein <code>StackPanel</code> , allerdings bricht es automatisch um, wenn der zur Verfügung stehende Platz zur Neige geht. Genauso wie bei <code>StackPanel</code> kann man die Orientierung über die Eigenschaft <code>Orientation</code> steuern.

Abbildung 226: Wichtige Panels in WPF

Bitte beachten Sie: Panels können wiederum andere Panels als Kindelemente enthalten, die dann üblicherweise ein Sublayout in einem Elternpanel bilden.

5.20.2.4 Items Controls

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=u5RzXObyglw>.

Items Controls sind die kompliziertesten Elemente in WPF: ähnlich wie Panel ordnen sie mehrere anderer Elemente an – sie können aber auch ganz normale .NET Standardobjekte anzeigen, die sich selber nicht zeichnen können (mehr zum Zeichenprozess später). Dazu nutzen sie intern ein Panel und einen weiteren relativ komplizierten Mechanismus, um nicht-UI Objekte zu UI-Objekten, die sich zeichnen können, zu überführen (diesen Mechanismus müssen Sie aber nicht verstehen). Die Basisklasse für alle Items Controls ist `ItemsControl`. Alle Items Controls verfügen über die beiden Eigenschaften `Items` bzw. `ItemSource`, mit denen man die dargestellten Objekte festlegen kann. Weiterhin gibt es die Eigenschaft `DisplayMemberPath`, mit der man genauer festlegen kann, welche Eigenschaft eines Objekts im Items Control angezeigt werden soll. Zu jedem Items Control gibt es auch eine sog. Item-Klasse, die bei `ComboBox` ein `ComboBoxItem` ist, bei `ListBox` ein `ListBoxItem`, bei `TabControl` ein `TabItem` usw. Diese Item-Klassen kapseln jeweils den Objektinhalt und bringen im Normalfall die Auswahlfähigkeit mit und können entweder von uns als Entwickler explizit angegeben werden oder automatisch vom jeweiligen `ItemsControl` erstellt werden (mehr dazu im Video zu diesem Abschnitt).

Klasse	Bemerkung
<code>ComboBox</code>	Repräsentiert eine Box, mit der der Nutzer einen vorgegebenen Wert auswählen kann. Mit <code>SelectedItem</code> kann man feststellen, welches

	Element ausgewählt ist. Das SelectionChanged Event wird ausgelöst, wenn der Nutzer die Auswahl ändert.
ListBox	Repräsentiert eine Liste, aus der der Nutzer ein oder mehrere Elemente auswählen kann. Wichtige Members sind die Eigenschaften SelectedItem bzw. SelectedItems , welche die zuletzt selektierten Werte enthalten, sowie der SelectionChanged Event, der ausgelöst wird, wenn der Nutzer die Auswahl ändert.
TabControl	Ein TabControl ordnet mehrere Elemente in Reitern an, wovon einer immer aktiv ist und vom Nutzer gesichtet werden kann.
TreeView	Repräsentiert einen Baum, der über beliebig viele Ebenen hinweg ein Kindelemente haben kann (dies ist das einzige hierarchische Items Control). Die TreeViewItems haben die Möglichkeit, aufgeklappt oder eingeklappt zu werden, wie man es bspw. aus dem Dateiexplorer gewohnt ist.

Abbildung 227: Wichtige Items Controls in WPF

5.20.2.5 Fenster und User Controls

Das Video zu diesem Abschnitt finden Sie unter <http://youtu.be/UH9n94xzDjQ>.

Die Klasse **Window** repräsentiert ein Fenster, das Inhalt anzeigt und standardmäßig minimiert, maximiert oder geschlossen werden kann. Dies haben wir auch bereits in **MainWindow** gesehen – diese Klasse leitet nämlich von **Window** ab. **Window** besitzt die wichtigen Methoden **Show** und **ShowDialog**, mit dem es angezeigt werden kann, entweder als normales Fenster oder als Dialogfenster, dass den Zugriff auf das dahinter liegende Hauptfenster verbietet, solange der Dialog offen ist.

Ein Fenster enthält dabei immer eine sog. Ansicht (engl. View), die der Nutzer gerade sieht. Im Hello WPF Beispiel haben wir diese Ansicht direkt im **MainWindow** aufgebaut, was bedingt durch die Einfachheit des Beispiels auch vollkommen OK war. Es ist aber auch üblich, dass man Ansichten in sog. User Controls definiert, die dann im Hauptfenster jeweils angezeigt werden.

Wenn eine Aktion des Nutzers dazu führt, dass (programmatisch) die aktuelle Ansicht ausgeblendet und eine neue Ansicht angezeigt wird, dann spricht man von Navigation, d.h. eine Aktion führt dazu, dass zu einer vorher nicht sichtbaren Ansicht navigiert wird. Diese neue Ansicht muss nicht unbedingt die aktuelle Ansicht ersetzen, sondern kann bspw. auch in einem neuen Dialogfenster gezeigt werden.

Wenn Sie Ihre Ansichten in User Controls definieren möchten, dann fügen Sie Ihrem WPF Projekt über den Hinzufügen-Dialog ein solches hinzu. Gemein ist allen, dass sie von der Klasse **UserControl** ableiten, in eine XAML- und Code-Behind-Datei aufgeteilt sind und wie **MainWindow** einen parameterlosen Konstruktor besitzen müssen, in dem die Methode **InitializeComponent** aufgerufen wird. Bitte entfernen Sie diesen Aufruf nicht – ansonsten kann es vorkommen, dass Ihr Fenster oder User Control nicht mehr richtig funktioniert (der Compiler gibt aber in diesem Fall keine Fehlermeldung aus).

Wenn Sie eigene User Controls in Ihrem Hauptfenster einsetzen wollen, dann müssen Sie zunächst den C# Namespace auf einen XML Namespace in XAML mappen, damit Sie auf Ihre Klasse zugreifen können. Das macht man wie folgt:

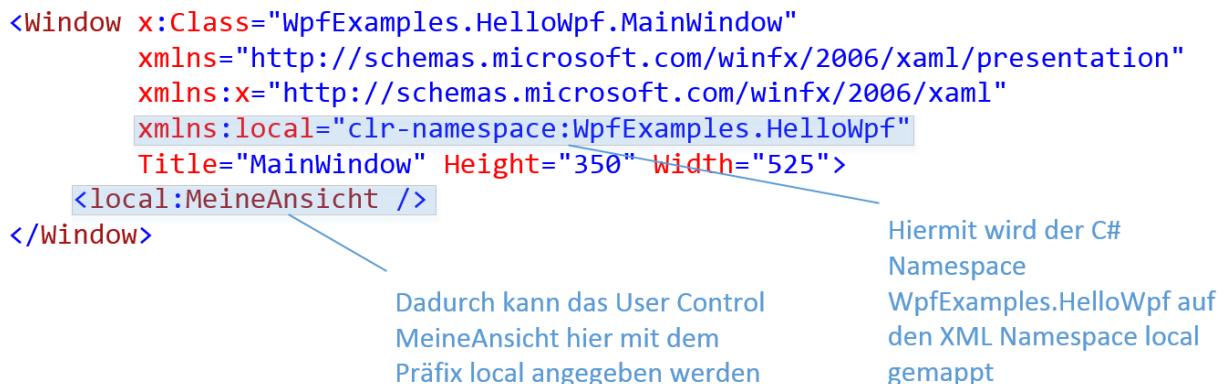


Abbildung 228: XAML Namespace Mapping

Wie Sie in Abbildung 228 sehen, wird der XML Namespace **local** auf den C# Namespace `WpfExamples.HelloWpf` gemappt, indem man ihn nach folgendem Schema angibt:
“clr-namespace:C# Namespace”

Genau dann kann die Klasse **MeinAnsicht** auch in XAML verwendet werden. Alle Elemente, die direkt zum WPF Framework gehören (also bspw. Buttons, Panels, Textblöcke usw.), sind automatisch im Default Namespace `xmlns` enthalten.

5.20.2.6 Wo finde ich weitere Dokumentationen zu WPF Elementen?

Aufgrund der Größe des WPF Frameworks ist es nicht möglich, alle Details zu jeder Klasse hier im Script zu besprechen. Wenn Sie mit einer Klasse arbeiten und genauere Details zu ihr wissen möchten, sollte ihre erste Anlaufstelle die [MSDN Library](#) sein. Weiterhin gibt es auch unzählige Tutorials zu WPF, die leicht über ihre Suchmaschine gefunden werden können. Eine gute Übersicht bietet meines Erachtens Christian Mosers Website <http://www.wptutorial.net/>.

5.20.3 Aufbau üblicher Ansichten

Wir haben bereits im vorletzten Abschnitt den Begriff Ansicht oder View erwähnt – in diesem Abschnitt möchten wir einige Standardansichten, die häufig in User Interfaces eingesetzt werden, besprechen.

5.20.3.1 Master-Detail-Ansichten

Das Video zu diesem Abschnitt finden Sie unter http://youtu.be/_glxIdkjKwU.

Master-Detail-Ansichten werden sehr häufig in Applikationen verwendet, seien es Web Sites, Apps oder Desktopanwendungen. Eine Master-Detail-Ansicht besteht dabei, wie der Name schon sagt, aus zwei Bereichen:

- Im Masterbereich werden mehrere Objekte einer Art angezeigt, aus denen der Nutzer eines auswählen kann. Üblicherweise setzt man hier eine **ListBox** ein, welche die Objekte präsentiert. Von den Objekten wird jeweils nur ein kleiner Bruchteil angezeigt, bspw. der volle Name bei einem Kontakt oder Betreff und Sender einer Email.
- Im Detail-Bereich werden detaillierte Informationen zum ausgewählten Objekt des Masterbereichs angezeigt. Bei einem Kontakt könnte das beispielsweise alle Daten wie Telefonnummer und Emailadresse sein, bei einer Email eben der tatsächliche Text, BC, Sendedatum usw. Hier werden häufig Textblöcke zur Darstellung der Daten verwendet.

Der wichtige Punkt ist, dass der Detail-Bereich immer dann aktualisiert wird, wenn im Masterbereich ein anderes Objekt ausgewählt wird.

Wenn man die Elemente, die im Masterbereich sitzen, manipulieren (also neue Elemente hinzufügen, bestehende löschen oder editieren) kann, dann befinden sich üblicherweise auch Buttons in diesem Bereich, die zu diesem Zweck eingesetzt werden können und dann häufig Dialoge auslösen, in denen der Nutzer die entsprechende Funktionalität ausführen kann.

In der folgenden Skizze sind die besprochenen Elemente aufgezeichnet:

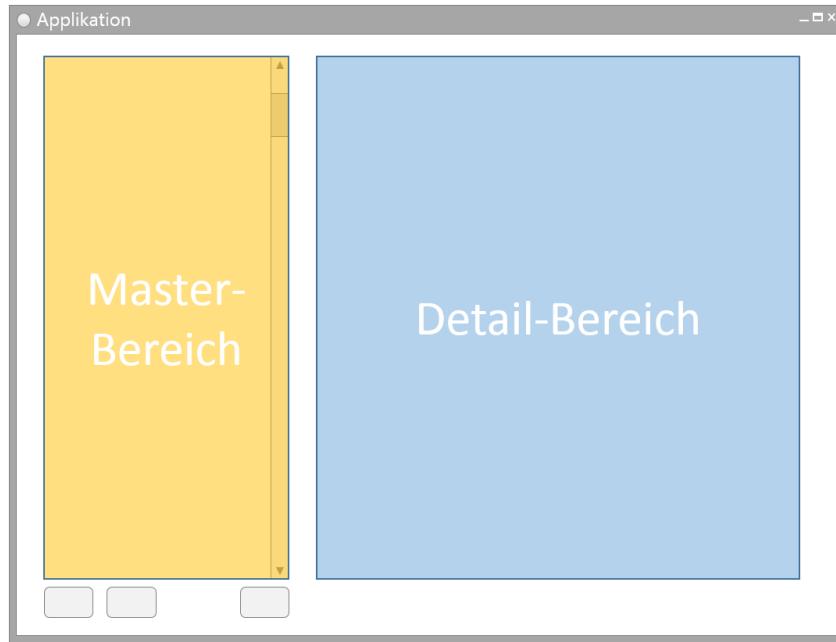


Abbildung 229: Skizze einer einfachen Master-Detail-Ansicht

In XAML könnte man diese Ansicht wie im folgenden Beispiel gezeigt aufbauen:

```

<UserControl x:Class="WpfExamples.HelloWorld.MasterDetailView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="33*" />
            <ColumnDefinition Width="66*" />
        </Grid.ColumnDefinitions>
        Das Grid teilt die komplette
        Ansicht in Master- und
        Detailbereich auf
        <ListBox Name="MasterListBox" Margin="5" SelectionChanged="WennAuswahlGeändertWurde" />
        <StackPanel Grid.Column="1" Margin="5">
            <TextBlock Name="NameTextBlock"
                FontFamily="Segoe UI Light"
                FontSize="36"
                Margin="0 0 0 10" />
            <TextBlock Name="GeburtsdatumTextBlock" Margin="0 0 0 5" />
            <TextBlock Name="TelefonTextBlock" Margin="0 0 0 5" />
            <TextBlock Name="EmailTextBlock" Margin="0 0 0 5" />
        </StackPanel>
        Dieses StackPanel repräsentiert den
        Detailbereich. Alle enthaltenen
        Textblöcke werden aktualisiert, wenn
        der Nutzer die Auswahl ändert
    </Grid>
</UserControl>

```

Die Liste erläutert die XAML-Struktur:

- Das Grid teilt die komplette Ansicht in Master- und Detailbereich auf**: Die Grid-Struktur teilt das Fenster in zwei Spalten von unterschiedlichen Breiten.
- Die ListBox stellt den Masterbereich dar**: Die Liste enthält die Auswahloptionen für den Masterbereich.
- SelectionChanged wird genau dann ausgelöst, wenn der Nutzer ein anderes Element auswählt**: Der Event-Handler 'WennAuswahlGeändertWurde' wird ausgelöst, wenn ein anderes Element in der Liste ausgewählt wird.
- Dieses StackPanel repräsentiert den Detailbereich. Alle enthaltenen Textblöcke werden aktualisiert, wenn der Nutzer die Auswahl ändert**: Das StackPanel enthält vier Textblöcke, die aktualisiert werden, wenn die Auswahl im Masterbereich geändert wird.

Abbildung 230: Master-Detail-Ansicht für Kontakte in XAML

Wie Sie in Abbildung 230 sehen können, wird die Ansicht als **UserControl** wie folgt aufgebaut:

- Ein **Grid** teilt den zur Verfügung stehenden Raum in zwei Spalten auf. Die erste Spalte nutzt dabei 33, die zweite 66 von insgesamt 99 Einheiten. Dies wird durch die Sternchenangabe bei **Width** für **ColumnDefinition** deutlich gemacht. Dadurch skaliert das Verhältnis auch, wenn die Größe des Fensters verändert wird.
- In der ersten Spalte sitzt die **ListBox**, welche alle Elemente bereitstellt, die vom Nutzer ausgewählt werden können. Auf ihren **SelectionChanged** Event ist eine Methode registriert, die den Detailbereich aktualisiert mit dem Element, das aktuell ausgewählt ist. Damit Sie auch im Code-Behind-File angesprochen werden kann, wurde ihr der Name **MasterListBox** gegeben.
- Das darunter liegende **StackPanel** repräsentiert den Detailbereich. In ihm sind mehrere Textblöcke angeordnet, die jeweils die unterschiedlichen Daten für einen Kontakt halten sollen. Sie werden in der Code-Behind-Datei in der Methode **WennAuswahlGeändertWurde** aktualisiert, weswegen wir ihnen allen ebenfalls einen Namen gegeben haben.

Die Code-Behind-Datei sieht dann wie folgt aus:

```

public partial class MasterDetailView : UserControl
{
    public MasterDetailView()
    {
        InitializeComponent();
    }

    public IList<Kontakt> Kontakte
    {
        get { return MasterListBox.ItemsSource as IList<Kontakt>; }
        set
        {
            if (value == null) throw new ArgumentNullException("value");
            MasterListBox.ItemsSource = value;      Mit dieser Eigenschaft kann die Collection mit
                                                    Kontakten von außerhalb gesetzt werden
        }
    }

    private void WennAuswahlGeändertWurde(object sender, SelectionChangedEventArgs e)
    {
        LeereTextblöcke();
        var ausgewählterKontakt = (Kontakt) MasterListBox.SelectedItem;
        if (ausgewählterKontakt == null)
            return;

        SetzeTextblöcke(ausgewählterKontakt);      Diese Methode wird aufgerufen, wenn der
                                                    SelectionChanged Event von der MasterListBox
                                                    ausgelöst wird
    }

    private void SetzeTextblöcke(Kontakt kontakt)
    {
        NameTextBlock.Text = kontakt.VollerName;
        if (kontakt.Geburstdatum.HasValue)
            GeburtsdatumTextBlock.Text = kontakt.Geburstdatum.Value.ToShortDateString();
        EmailTextBlock.Text = kontakt.Email;
        TelefonTextBlock.Text = kontakt.Telefon;

    }

    private void LeereTextblöcke()
    {
        NameTextBlock.Text = GeburtsdatumTextBlock.Text
            = EmailTextBlock.Text = TelefonTextBlock.Text = string.Empty;
    }
}

```

Abbildung 231: Master-Detail-View für Kontakte (Code-Behind)

In Abbildung 231, der Code-Behind Datei für die Master-Detail-Ansicht für Kontakte sehen Sie folgende Dinge:

- Die Methode `WennAuswahlGeändertWurde`, die wir bereits vorher angesprochen haben. In ihr wird das aktuell selektierte Kontaktobjekt geholt und verarbeitet.
- Dazu wird die Methode `SetzeTextblöcke` aufgerufen, welche die Textblöcke im Detailbereich mit den Informationen des aktuellen Kontaktobjekts füllt.
- Über die Eigenschaft `Kontakte` kann die Collection gesetzt werden, deren Objekte innerhalb der `ListBox` angezeigt werden sollen. Dazu wird die eingehende Collection an `ListBox.ItemsSource` weitergegeben.

Bitte beachten Sie, dass das nicht der beste Weg ist, eine Master-Detail-View aufzubauen. Über den Data Binding Mechanismus von WPF könnten wir uns viel Code ersparen – da wir diesen Mechanismus jetzt noch nicht kennen und ich ihn nicht als Grundlagenwissen voraussetze, bleiben wir vorerst bei der eben gezeigten Implementierung.

5.20.3.2 Formularansichten

Das Video zu diesem Abschnitt finden Sie unter <http://youtu.be/-RoNVIDiwf4>.

Formularansichten werden häufig dann eingesetzt, wenn der Nutzer mehrere Daten zu einem Sachverhalt eingeben muss – üblicherweise wird dies in Dialogen gemacht, bei denen ein bestehendes Objekt modifiziert oder ein neues erstellt wird. Dabei werden die Input Controls untereinander angeordnet und ein jeweils nebenan stehendes Label / Textblock beschreibt, zu welchen Punkt man einen Wert eingibt – genauso wie man es bei papierbasierten Formularen gewohnt ist. Eine Skizze einer solchen Formularansicht finden Sie in der kommenden Abbildung:

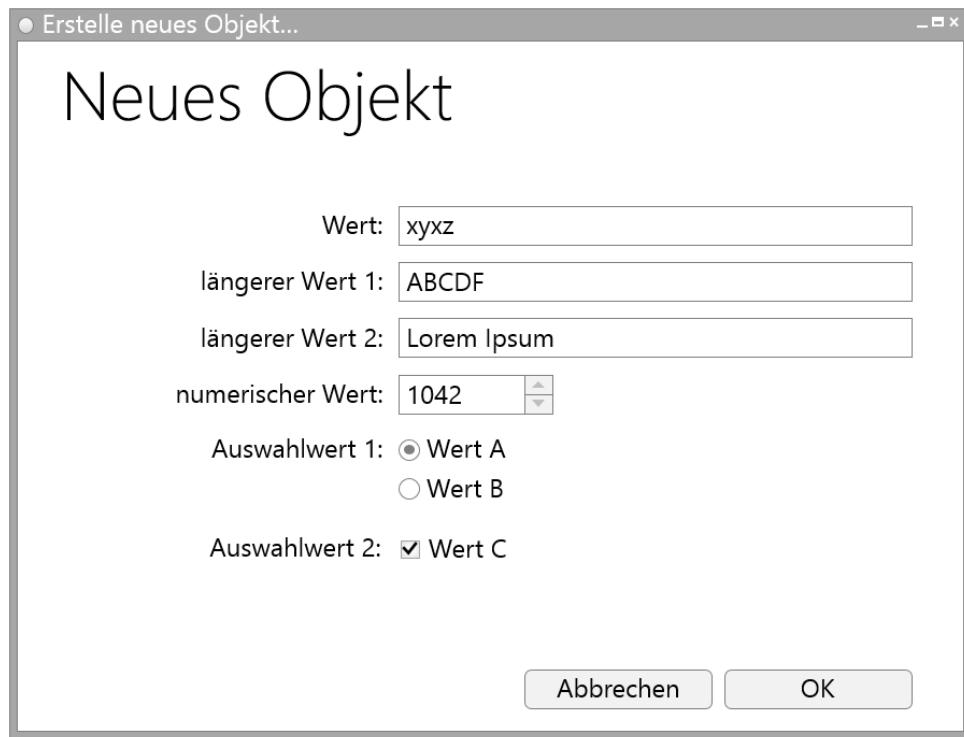


Abbildung 232: Skizze einer Formularansicht

Wie wir in Abbildung 232 zu sehen ist, sollten die Textbeschreibungen rechtsbündig ausgerichtet werden, damit der Nutzer beim Anblick leichter identifizieren kann, zu welchem Input Control diese jeweils gehören.

Wenn wir eine solche Formularansicht für unser Kontaktbeispiel nachbauen möchten, können wir dazu ein neues [Window](#) zum Projekt hinzufügen und die XAML Datei wie folgt gestalten:

```

<Window x:Class="WpfExamples.HelloWorld.EditiereKontaktFormular"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Editiere Kontakt..." Height="250" Width="300">
<Window.Resources>
    <Style x:Key="GridTextBlockStyle" TargetType="TextBlock">
        <Setter Property="Margin" Value="30 0 5 4" />
        <Setter Property="HorizontalAlignment" Value="Right" />
        <Setter Property="VerticalAlignment" Value="Bottom" />
    </Style>
    <Style x:Key="GridInputControlStyle" TargetType="FrameworkElement">
        <Setter Property="Grid.Column" Value="1" />
        <Setter Property="Margin" Value="0 2" />
    </Style>
    <Style x:Key="DialogButtonStyle" TargetType="Button">
        <Setter Property="Width" Value="75" />
        <Setter Property="Margin" Value="0 0 5 0" />
    </Style>
</Window.Resources>
<Grid Margin="5"> _____ Dieses Grid teilt das Fenster in die
    <Grid.ColumnDefinitions> notwendigen Zeilen und Spalten auf
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <TextBlock Text="Editiere Kontakt" FontFamily="Segoe UI Light"
        FontSize="24" Grid.ColumnSpan="2" Margin="0 0 0 10"/> _____ Dieser Textblock repräsentiert die Überschrift

    <TextBlock Grid.Row="1" Style="{StaticResource GridTextBlockStyle}" Text="Vorname:"/>
    <TextBox Name="VornameTextBox" Grid.Row="1" Style="{StaticResource GridInputControlStyle}" />

    <TextBlock Grid.Row="2" Style="{StaticResource GridTextBlockStyle}" Text="Nachname:"/>
    <TextBox Name="NachnameTextBox" Grid.Row="2" Style="{StaticResource GridInputControlStyle}" />

    <TextBlock Grid.Row="3" Style="{StaticResource GridTextBlockStyle}" Text="Geburtsdatum:"/>
    <DatePicker Name="GeburstdatumsAuswahl" Grid.Row="3" Style="{StaticResource GridInputControlStyle}" />

    <TextBlock Grid.Row="4" Style="{StaticResource GridTextBlockStyle}" Text="Telefon:"/>
    <TextBox Name="TelefonTextBox" Grid.Row="4" Style="{StaticResource GridInputControlStyle}" />

    <TextBlock Grid.Row="5" Style="{StaticResource GridTextBlockStyle}" Text="Email:"/>
    <TextBox Name="EmailTextBox" Grid.Row="5" Style="{StaticResource GridInputControlStyle}" />

    <StackPanel Grid.Row="6" VerticalAlignment="Bottom" Grid.ColumnSpan="2" _____ Diese Elemente repräsentieren die Input
        FlowDirection="RightToLeft" Orientation="Horizontal">
        <Button Content="OK" Click="WennOkButtonGedrücktWurde"
            Style="{StaticResource DialogButtonStyle}" />
        <Button Content="Abbrechen"
            Click="WennAbbrechenButtonGedrücktWurde"
            Style="{StaticResource DialogButtonStyle}" />
    </StackPanel>
</Grid>
</Window>

```

Innerhalb dieses StackPanels sind der OK und
Abbrechen Button untergebracht

Abbildung 233: Kontakt-Formularansicht XAML

In Abbildung 233 sehen Sie, dass diese Ansicht wie folgt aufgebaut ist:

- Der zur Verfügung stehende Platz wird durch ein **Grid** in zwei Spalten und sieben Zeilen aufgeteilt. Dabei wird bei den oberen sechs Zeilen die Höhenangabe **Height="Auto"** gemacht, damit sich die Höhe dieser Zeilen an deren Inhalten orientiert. Die letzte Zeile hat keine Höhenangabe, damit sie sich auf den Rest des zur Verfügung stehenden Raums verteilt.

- Der erste **TextBlock** im **Grid** repräsentiert den Titel des Dialogs und erstreckt sich über beide Spalten in Zeile 0 (auch die Indizes in WPF sind nullbasiert).
- Danach folgen mehrere Kombinationen aus **TextBlock** und Input Control (**TextBox** bzw. **DatePicker**), die jeweils in einer Zeile stehen. Der Textblöcke sind in der linken Spalte und rechts ausgerichtet. Die Input Controls stehen in der rechten Spalte und ihnen allen wurde ein Name zugewiesen, damit sie in der Code-Behind-Datei angesprochen werden können.
- Erstmals sehen wir auch, dass Eigenschaften von Attributen nicht direkt als XML Attribute auf Elementen gesetzt werden können, sondern auch über sog. Styles. Drei solcher Styles sind in den Ressourcen des **Window** untergebracht und werden von den jeweiligen Elementen über die Syntax **{StaticResource StyleName}** referenziert. Dies ist die übliche Syntax, um Objekte in Ressourcen zu ansprechen, dazu später mehr.
- Im unteren **StackPanel** sind die beiden Buttons OK und Abbrechen untergebracht. Damit diese horizontal von rechts nach links angeordnet werden, wurde die Eigenschaft **FlowDirection** auf dem **StackPanel** angepasst. Für die Buttons wird ebenfalls je eine Methode auf den **Click** Event registriert, die das Dialogfenster schließt und einen entsprechenden Code an den ShowDialog-Aufrufer zurückgibt.

In Abbildung 234 sehen Sie die Code Behind Datei zu diesem Fenster. In ihr gibt es im Wesentlichen folgende Dinge, die es zu beachten gilt:

- Über die Eigenschaft **EditierterKontakt** kann man das Kontaktobjekt setzen, dessen Werte bearbeitet werden sollen. Wenn es gesetzt wird, werden dessen Eigenschaftswerte in die entsprechenden Input Controls übertragen.
- Wenn der Nutzer die Bearbeitung mit OK bestätigt, wird die Methode **WennOKButtonGedrücktWurde** aufgerufen. In ihr werden die Daten der Input Controls an das in **_editierterKontakt** referenzierte Objekt zurückübertragen und der Dialog erfolgreich abgeschlossen, indem **DialogResult** auf **true** gesetzt und im Anschluss **Window.Close** aufgerufen wird.
- Wenn der Nutzer den Abbrechen Button klickt und dadurch die Methode **WennAbbrechenButtonGedrücktWurde** aufgerufen wird, werden die Daten des ursprünglichen Objekts nicht verändert und der Dialog abgebrochen, indem **DialogResult** auf **false** gesetzt und das Fenster wie eben beschrieben geschlossen wird.

Bitte beachten Sie auch hier, dass wir sehr viel sog. Boiler Plate Code schreiben (hier das Zuweisen der Eigenschaftswerte zu den Input Controls und wieder zurück), den wir uns durch WPF Data Binding sparen könnten. Die hier gezeigte Lösung ist also nicht ideal.

Wie dieses Fenster als Dialog eingesetzt werden kann, können Sie im dazugehörigen Video betrachten, in dem es aus der vorher besprochenen Master-Detail-Ansicht aufgerufen wird. Des Weiteren haben wir in diesem Beispiel auch noch keine Validierung der Eingaben des Nutzers gemacht – auch das müssten wir natürlich in Produktionscode beachten.

```

public partial class EditiereKontaktFormular : Window
{
    private Kontakt _editierterKontakt;

    public Kontakt EditierterKontakt — Über diese Eigenschaft kann der zu editierende
    {                                              Kontakt zugewiesen werden.
        get { return _editierterKontakt; }
        set
        {
            if (value == null) throw new ArgumentNullException("value");

            _editierterKontakt = value;
            VornameTextBox.Text = value.Vorname;
            NachnameTextBox.Text = value.Nachname;
            GeburstdatumsAuswahl.SelectedDate = value.Geburstdatum;
            TelefonTextBox.Text = value.Telefon;
            EmailTextBox.Text = value.Email;
        }
    }

    public EditiereKontaktFormular()
    {
        InitializeComponent();
    }

    private void WennOkButtonGedrücktWurde(object sender, RoutedEventArgs e)
    {
        if (_editierterKontakt != null)
        {
            _editierterKontakt.Vorname = VornameTextBox.Text;
            _editierterKontakt.Nachname = NachnameTextBox.Text;
            _editierterKontakt.Geburstdatum = GeburstdatumsAuswahl.SelectedDate;
            _editierterKontakt.Telefon = TelefonTextBox.Text;
            _editierterKontakt.Email = EmailTextBox.Text;
        }

        DialogResult = true;
        Close();                                            Diese Methode wird durch den Abbrechen
                                                                Button aufgerufen und beendet den Dialog,
                                                                ohne Änderungen am Kontaktobjekt
                                                                vorzunehmen
    }

    private void WennAbbrechenButtonGedrücktWurde(object sender, RoutedEventArgs e)
    {
        DialogResult = false;
        Close();
    }
}

```

Abbildung 234: Kontakt-Formularansicht Code Behind

5.20.4 Die Klasse App

Das Video zu diesem Abschnitt finden Sie unter https://www.youtube.com/watch?v=1_wVTRfQRb4.

Als nächstes möchten wir uns mit der Klasse **App** auseinandersetzen, die neben **MainWindow** automatisch generiert wurde. Diese bildet nämlich den Haupteinstiegspunkt ins Programm: wenn Sie im Solution Explorer den Button „Show All Files“ anklicken wie in Abbildung 235, sehen Sie nicht nur ihre Codedateien, sondern auch die, welche während des Erstellvorgangs automatisch generiert werden.

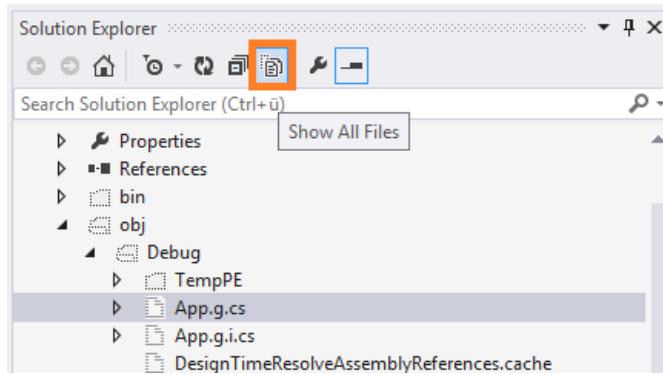


Abbildung 235: Der Show All Files Button im Solution Explorer

Im Unterordner obj\Debug finden Sie dann u.a. die Datei App.g.cs (.g. Dateien zeigen im Allgemeinen an, dass Sie vom Erstellprozess automatisch generiert werden). Diese wird aus der Datei App.xaml erzeugt und enthält den Einstiegspunkt in ihr Programm:

```
public partial class App : System.Windows.Application {

    /// <summary>
    /// InitializeComponent
    /// </summary>
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks", "4.0.0.0")]
    public void InitializeComponent() {

        #line 4 "..\..\App.xaml"
        this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);

        #line default
        #line hidden
    }

    /// <summary>
    /// Application Entry Point.
    /// </summary>
    [System.STAThreadAttribute()]
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks", "4.0.0.0")]
    public static void Main() {
        WpfExamples.HelloWorld.App app = new WpfExamples.HelloWorld.App();
        app.InitializeComponent();
        app.Run();
    }
}
```

Abbildung 236: Die generierte Datei App.g.cs

Hier finden wir die Main Methode, die wir vorher nicht gesehen haben. In ihr wird eine neue Instanz der Klasse App erstellt und im Anschluss auf dieser Instanz die Methode InitializeComponent aufgerufen, in der nur die StartupUri gesetzt wird. Das sorgt dafür, dass beim Aufruf von Application.Run automatisch eine Instanz von MainWindow erzeugt, als Hauptfenster festgelegt und angezeigt wird.

Wenn Sie genauer auf diesen Prozess Einfluss nehmen möchten, können Sie das tun:

- Entfernen Sie das **StartupUri** Attribut aus der Datei App.xaml, damit das **MainWindow** nicht mehr automatisch erstellt wird.

- Überschreiben Sie die Methode `Application.OnStartup`. Rufen Sie in Ihrer eigenen Implementierung auf jeden Fall die Implementierung der Basisklasse auf – danach können Sie allerdings beliebigen Code schreiben, der beim Start der Applikation ausgeführt werden soll. Hier können das `MainWindow` und alle anderen Objekte, welche für die Applikation gebraucht werden, instanziert werden (Composition Root).

In der folgenden Abbildung ist dieses Vorgehen nochmals illustriert:

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);

        var mainWindow = new MainWindow();
        MainWindow = mainWindow;
        mainWindow.Show();
    }
}
```

Die Methode `OnStartup` kann überschrieben werden und stellt damit den gängigsten Einstiegspunkt in WPF Applikationen dar

Die Eigenschaft `Application.MainWindow` sollte immer gesetzt werden, damit sich die Applikation automatisch beim Schließen dieses Fensters beendet.

Abbildung 237: Das `MainWindow` beim Startvorgang manuell erzeugen und anzeigen

Des Weiteren wird die Klasse `App` dazu genutzt, systemweite Ressourcen wie bspw. Styles in seinen Ressourcen bereitzustellen. Diese können dann in allen Ansichten und Fenstern wiederverwendet werden.

5.20.5 Der Renderprozess des User Interfaces und Threading

Nachdem wir in den letzten Abschnitten gesehen haben, wie man verschiedene Ansichten aufbauen und die Klasse `App` einsetzen kann, möchten wir uns jetzt näher mit den Interna von GUI Frameworks im Allgemeinen beschäftigen. Insbesondere werden wir sehen, welcher Mechanismus hinter einem GUI liegt, das sich mehrmals pro Sekunde neu zeichnet und welche Auswirkungen das auf die Programmierung und den Umgang mit Threads hat.

5.20.5.1 Der User Interface Thread

Das Video zu diesem Abschnitt finden Sie unter <https://www.youtube.com/watch?v=LEO1yRMoal4>.

Beginnen möchte ich mit der UI Thread: ein Thread wird zu einem UI Thread erhoben, indem auf ihn eine Methode ausgeführt wird, die ein Schleife enthält, welche nur unter bestimmten Bedingungen abgebrochen und standardmäßig immer wieder ausgeführt wird. Innerhalb dieser Schleife werden periodisch folgende Schritte ausgeführt:

- Die Oberfläche zeichnen
- Betriebssystemnachrichten verarbeiten
- Nutzereingaben verarbeiten

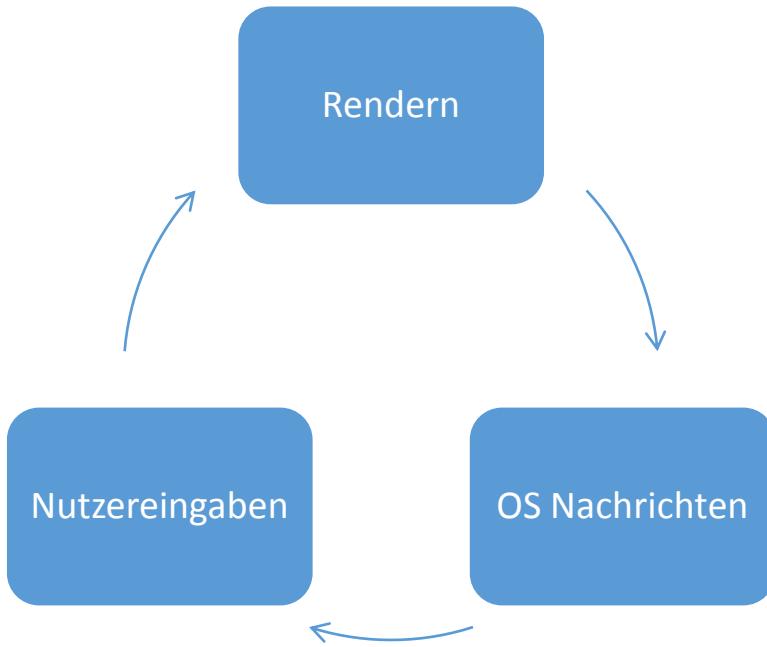


Abbildung 238: Die Aufgabengebiete des UI Threads

Wie in Abbildung 238 zu sehen ist, besteht die Hauptaufgabe des UI Threads im regelmäßigen Neuzeichnen der Applikation und das Verarbeiten und Weiterreichen von Betriebssystemnachrichten oder Nutzereingaben, was ca. 60-mal pro Sekunde passiert. In WPF ist dieses Konzept von der Klasse `System.Windows.Threading.Dispatcher` umgesetzt, welche letztendlich eine priorisierte Schlange darstellt, in der je nach Priorität die oben genannten Funktionalitäten aufgerufen werden. Wenn Sie eine WPF Applikation mit einem Debugger anhalten, werden Sie sehen, dass sich diese immer auf dem UI Thread in der Methode `Dispatcher.Run` oder einer von dieser Methode aufgerufenen Funktion befindet.

Die Schleife des Dispatchers wird standardmäßig erst beendet, wenn alle Fenster einer Applikation geschlossen wurden. Natürlich können wir den Dispatcher auch manuell herunterfahren – üblicherweise tut man das einfach via `App.Shutdown`, denn der Dispatcher wird von der Klasse `Application` gekapselt.

Der viel wichtigere Punkt ist jedoch, dass sämtliche Methoden, die wir auf Events von WPF Elementen registrieren, innerhalb dieses Schleifenmechanismus ausgeführt werden.

Wenn eigene Methoden über das Hollywood-Prinzip auf dem GUI Thread aufgerufen werden, dann dürfen diese nur eine sehr kurze Durchlaufzeit haben. Ansonsten kehrt der Aufruf nicht rechtzeitig zum Dispatcher zurück, damit dieser den Renderprozess wieder anstoßen kann – für den Nutzer wirkt es dann, als wäre die Benutzeroberfläche eingefroren. Diesen Zustand sollten Sie wo möglich vermeiden.

Im Umkehrschluss sollte man als Entwickler dafür sorgen, dass lang andauernde Methoden nicht auf dem GUI Thread, sondern auf einem Hintergrundthread ausgelagert werden sollten. Das heißt nicht, dass man auf Teufel komm raus alle Aktivitäten auf Hintergrundthreads schieben muss – wenn eine Aktion aber länger als 40 – 50 Millisekunden dauert, sollte man das in Erwägung ziehen. Häufig sind dies Aktionen, die mit Netzwerk-, Datei- oder Datenbankzugriffen einhergehen. Aktionen, die nur Berechnungen im Arbeitsspeicher durchführen, sind meistens ausreichend schnell, außer es handelt

sich bspw. um komplexe Algorithmen. Hier sollte man tatsächlich jeweils die Durchlaufzeit messen und dann entsprechend entscheiden.

5.20.5.2 Methoden auf Hintergrundthreads auslagern

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=xNsZNaPhOMM>.

In diesem Abschnitt werden wir uns anschauen, welche APIs existieren, um in .NET eine Methode auf einem neuen Thread auszulagern. Die älteste Variante ist die Klasse `System.Threading.Thread`, mit der man direkt einen neuen Thread erzeugen und starten kann:

```
private static void Main()
{
    var aktuellerThread = Thread.CurrentThread;
    Console.WriteLine("Start auf Thread {0}", aktuellerThread.ManagedThreadId);
    var neuerThread = new Thread(GibInSchleifeAus)
    {
        IsBackground = true,
        Name = "Mein Thread"
    };
    neuerThread.Start();
    Console.ReadLine();
```

Mit `Thread.CurrentThread` kann man auf das Threadobjekt zugreifen, auf dem der aktuelle Code läuft

Der Threadkonstruktor erhält einen Delegate, der ausgeführt wird, wenn der Thread startet

Backgroundthreads werden automatisch beendet, wenn der letzte Hauptthread der Applikation endet

Startet den Thread und kehrt danach sofort zurück

```
private static void GibInSchleifeAus()
{
    for (var i = 0; i < 100; i++)
    {
        Console.WriteLine("Durchlauf {0} auf Thread {1}",
            i,
            Thread.CurrentThread.ManagedThreadId);

        Thread.Sleep(400);
    }
}
```

`Thread.Sleep(400);` Thread.Sleep sorgt dafür, dass ein Thread für die angegebene Zeitdauer unterbrochen wird.

Abbildung 239: Multithreading mit Klasse `Thread`

Wie wir in Abbildung 239 sehen können, kann man die Klasse `Thread` ganz einfach über ihren Konstruktor instanziiieren. Dabei muss man einen Delegate übergeben, der auf die Methode zeigt, die vom neuen Thread aufgerufen wird. Im obigen Fall ist dies die Methode `GibInSchleifeAus`, die hundert Konsolenausgaben macht, wobei nach jeder Ausgabe der Thread mit der statischen Methode `Thread.Sleep` unterbrochen wird. Erst nach (mindestens) 400 Millisekunden wird die Schleife neu begonnen. Was genau hier passiert, werden Sie u.a. in der Vorlesung Betriebssysteme lernen – dort wird noch genauer auf das Threading-Modell eingegangen.

Ein Thread wird dabei über die Methode `Start` gestartet. Diesen Aufruf bezeichnet man als **asynchron**, da die Methode, die auf dem Thread ausgeführt wird (`GibInSchleifeAus`), tatsächlich noch nicht beendet ist, wenn wir aus `Start` in die `Main` Methode zurückkehren.

Ein wichtiger Punkt ist auch, dass der neue Thread als Backgroundthread gekennzeichnet ist mit `IsBackground = true`. Das hat deutliche Auswirkungen:

Hintergrundthreads sind mit Hauptthreads (engl. Foreground Threads) identisch, mit der Ausnahme, dass sie die CLR nicht an der Terminierung hindern. Sobald der letzte Hauptthread einer Applikation endet, beendet die CLR den kompletten Prozess. Dabei werden auch alle noch laufenden Hintergrundthreads gestoppt und beendet.

Genau das ist es, was wir wollen: wenn der Nutzer unsere Applikation beendet, soll nicht noch eine oder mehrere Aktionen, die auf anderen Threads ablaufen, den Prozess am Herunterfahren hindern.

Folglich sollte man als Entwickler grundsätzlich Funktionalität auf einem Hintergrundthread auszulagern. Nur in bestimmten Szenarien macht es Sinn, einen selbsterzeugten Thread als Hauptthread laufen zu lassen – vorher sollte man sich als Entwickler darüber Gedanken gemacht haben und wissen, was man tut.

Zwar kann man mit der Klasse `Thread` neue Hintergrundthreads erstellen und auf ihnen Methoden ausführen, allerdings umgeht dies ein wesentliches Bestandteil der CLR: den sog. Thread Pool. Der Thread Pool ist eine Collection von Background Threads, die von der CLR am Leben erhalten werden und die sofort Funktionalität ausführen können, ohne dass die Erzeugung eines neuen Threads notwendig wäre. Weiterhin hat der Thread Pool den Vorteil, dass weitere Hintergrundthreads automatisch erzeugt und verwaltet werden, wenn diese vom Nutzercode benötigt werden. Dabei wird auch auf die Anzahl an Prozessorkernen geachtet, die das System aufweist, auf dem die Applikation gerade ausgeführt wird. Als Softwareentwickler sollte man genau diesen Thread Pool einsetzen, um Funktionalität auf Hintergrundthreads auszulagern.

Wir werden die Task Parallel Library (TPL), die mit .NET 4.0 veröffentlicht wurde, einsetzen, um den Thread Pool anzusprechen. Es gibt noch zwei ältere APIs (die Klasse `ThreadPool` und asynchron ausgeführte Delegates), die wir hier aber nicht betrachten. Die wichtigste Klasse der TPL haben wir bereits eingesetzt: es ist die Klasse `System.Threading.Tasks.Task` bzw. `System.Threading.Tasks.Task<T>` (für Tasks, die bei einer Berechnung Werte zurückgeben sollen). Bei der Klasse `Metronom` in Abschnitt 5.17.3.6 haben wir Sie bspw. bereits genutzt.

Wenn wir bei unserem vorherigen Beispiel die Klasse `Thread` durch `Task` ersetzen, landen wir bei folgendem Code:

```

private static void Main()
{
    var aktuellerThread = Thread.CurrentThread;
    Console.WriteLine("Start auf Thread {0}", aktuellerThread.ManagedThreadId);
    var neuerTask = new Task(GibInSchleifeAus);
    neuerTask.Start();
}

Console.ReadLine(); Ein Task wird
                     ebenfalls mit Start
                     asynchron ausgeführt
}

private static void GibInSchleifeAus()
{
    for (var i = 0; i < 100; i++)
    {
        Console.WriteLine("Durchlauf {0} auf Thread {1}",
                           i,
                           Thread.CurrentThread.ManagedThreadId);

        Thread.Sleep(400);
    }
}

```

Statt der Klasse Thread sollten wir die Klasse Task nutzen, um Methoden auf den Thread Pool ausführen zu lassen

Abbildung 240: Mit Task Funktionalität asynchron auf dem Thread Pool ausführen

Grundsätzlich ist die API von **Task** nicht sehr verschieden von der API der Klasse **Thread**. Auch hier übergeben wir einen Delegate an den Konstruktor von **Task** und führen im Anschluss über den Aufruf von **Start** die Funktionalität asynchron auf dem Thread Pool aus. Bei **Task<T>** ist das Vorgehen identisch, nur muss hier die aufrufende Methode eben einen Rückgabewert haben.

Die **Task** Klassen bietet insgesamt eine deutlich bessere API als **Thread**. Darunter fallen bspw.:

- Automatisches Handling bei der Anzahl an Threads, die eingesetzt werden.
- Besserer Unterstützung zum Abbrechen laufender Tasks über die Klasse **CancellationToken**.
- Die Möglichkeit, mehrere Tasks direkt hintereinander zu reihen mit der Methode **ContinueWith**.
- Parent-Child Tasks: ein Task kann mehrere Subtasks referenzieren, die ihm untergeordnet sind. Der Parent Task gilt erst dann als beendet, wenn sowohl er als auch sämtliche Child Tasks terminieren.

Bitte beachten Sie: wenn mehrere Threads auf dieselben Objekte und Werte schreibend und lesend zugreifen, kann es zu Inkonsistenzen bei diesen kommen (sog. Race Conditions). In diesem Fall ist es sinnvoll, den Zugriff auf kritische Sektionen über die Klasse Monitor oder das Schlüsselwort lock zu regeln, um eben genannte Inkonsistenzen zu vermeiden.

Genauer werden wir auf diese spezielle Funktionalitäten nicht eingehen, da dies noch in der Vorlesung Betriebssysteme genauer beschrieben wird. Wenn Sie sich in Ihrer Programmierkarriere mit Threading bei der TPL auseinandersetzen müssen, können Sie weitere Infos in der MSDN Library unter diesem Link finden:

[http://msdn.microsoft.com/en-us/library/dd460717\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx)

Wenn wir Funktionalität bei Applikationen mit User Interface auf einen Hintergrundthread auslagern, dann müssen wir allerdings einen noch einen wichtigen Punkt beachten, den wir im nächsten Abschnitt besprechen.

5.20.5.3 Threadaffinität von UI-Elementen

Das Video hierzu finden Sie unter https://www.youtube.com/watch?v=_sI3v2DoupE.

Sehen Sie sich folgenden Code an:

```
public partial class MainWindow : Window
{
    private void WennButtonGedrücktWird(object sender, RoutedEventArgs e)
    {
        Task.Factory.StartNew(BerechneKleinsteVielfacheZahl);
    }

    private void BerechneKleinsteVielfacheZahl() — Diese Methode wird asynchron auf dem Thread
    { — Pool ausgeführt
        ProgressBar.Indeterminate = true; ←
        var untererDividend = Convert.ToInt32(UntereGrenzeTextBox.Text); ←
        var obererDividend = Convert.ToInt32(ObererGrenzeTextBox.Text); ←

        var dividenden = Enumerable.Range(untererDividend, obererDividend - untererDividend)
            .ToList();

        for (var i = 1L;; i++)
        {
            if (ÜberprüfeZahlMitDividenden(i, dividenden))
            {
                ErgebnisTextBlock.Text = i.ToString("N"); ←
                break;
            }
        }

        ProgressBar.Indeterminate = false; ←
    }

    private static bool ÜberprüfeZahlMitDividenden(long i, IEnumerable<int> dividenden)
    {
        return dividenden.All(dividend => i % dividend == 0);
    }
}
```

Bei diesen Zugriffen kommt es jeweils zu einer Exception, da UI Elemente threadaffin sind

Abbildung 241: Zugriff auf UI Elemente von Hintergrundthreads löst Exceptions aus

In Abbildung 241 sehen Sie eine Code-Behind-Datei, welche die kleinste vielfache Zahl aus einer gegebenen Menge an Dividenden berechnet. Die Dividenden können in XAML über zwei Textboxen spezifiziert werden, so wie wir es auch bereits aus Übungsblatt 2 kennen (mit oberer und unterer Grenze).

Der Punkt ist, dass die Methode BerechneKleinsteVielfacheZahl auf dem Thread Pool ausgeführt wird. Innerhalb dieser Methode wird auf User Interface Elemente wie den Fortschrittsbalken, die beiden Textboxen und den Textblock zugegriffen. Das ist jedoch nicht erlaubt, es kommt wie angedeutet zu Exceptions, denn:

UI-Elemente sind threadaffin. Das bedeutet, dass Sie auf die Mitglieder dieser Objekte nur über den UI-Thread, auf dem sie erstellt wurden, zugreifen dürfen. Wenn Sie bspw. auf einem Hintergrundthread auf Eigenschaften von User Interface Elementen zugreifen, dann erhalten Sie eine InvalidOperationException.

Weiterhin ist noch anzumerken, dass die Task Klassen Exceptions automatisch fangen – d.h. Sie bekommen diese Exceptions in Visual Studio nicht in der üblichen Weise zu Gesicht. Warum das ein essentielles Feature von Tasks ist, sehen wir uns im nächsten Abschnitt genauer an.

Die Frage, die offen bleibt, ist folgende: wie kann ich meinen Code sinnvoll gestalten, sodass UI Elemente nur auf dem UI Thread angesprochen werden, das User Interface aber gleichzeitig reaktionsfähig (engl. responsive) bleibt, weil langandauernde Methoden auf den Thread Pool ausgelagert werden?

5.20.5.4 Die Schlüsselwörter `async` und `await`

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=qIqlBTaCci0>.

Die einfachste Möglichkeit, um eben genanntes Problem zu lösen, ist der Einsatz von `async` und `await`. Mit der Syntax `await Task` kann man einen Task auf einen Hintergrundthread ausführen lassen, während die aktuelle Methode zurückkehrt zum Dispatcher, sodass dieser weiter periodisch rendern kann. Sobald der Task abgeschlossen ist, wird die Methode wieder aufgerufen und nach dem `await` Statement fortgeführt, was dazu führt, dass asynchrone Aufrufe durch den Einsatz von `async` und `await` aussehen, als würden sie synchron verarbeitet werden.

Jede Methode, die wenigstens ein `await` enthält, muss dabei mit dem Modifizierer `async` gekennzeichnet sein. Sehen wir uns dazu das folgende Beispiel an, das eine modifizierte Variante des vorherigen ist:

```

public partial class MainWindow : Window
{
    private IEnumerable<int> _dividenden;

    public MainWindow()
    {
        InitializeComponent(); Methoden, die wenigstens ein await enthalten, müssen mit
        async gekennzeichnet werden
    }

    private async void WennButtonGedrücktWird(object sender, RoutedEventArgs e)
    {
        ProgressBar.Indeterminate = true;
        ErgebnisTextBlock.Text = string.Empty;
        var untererDividend = Convert.ToInt32(UntereGrenzeTextBox.Text);
        var obererDividend = Convert.ToInt32(ObereGrenzeTextBox.Text);
        _dividenden = Enumerable.Range(untererDividend, obererDividend - untererDividend)
            .ToList(); Mit await wird auf das Ende eines asynchronen Tasks
            gewartet, ohne den UI Thread zu blockieren – sobald der Task
            beendet ist, wird bei der nächsten Anweisung weitergemacht
        try
        {
            long number = await Task<long>.Factory.StartNew(BerechneKleinsteVielfacheZahl());
            ErgebnisTextBlock.Text = number.ToString("N0");
        }
        catch (ArgumentException exception)
        {
            ErgebnisTextBlock.Text = exception.Message;
        }
        ProgressBar.Indeterminate = false;
    }

    private long BerechneKleinsteVielfacheZahl()
    {
        for (var i = 1L; i < long.MaxValue; i++)
        {
            if (ÜberprüfeZahlMitDividenden(i))
                return i;
        }

        throw new ArgumentException("Zahl kann nicht gefunden werden.");
    }

    private bool ÜberprüfeZahlMitDividenden(long zahl)
    {
        return _dividenden.All(dividend => zahl % dividend == 0);
    }
}

```

Abbildung 242: *async und await im Einsatz*

In Abbildung 242 sehen Sie, dass alle Zugriffe auf die UI Controls direkt in *WennButtonGedrücktWird* durchgeführt werden. Die langandauernde Berechnung der kleinsten Vielfachen Zahl wird jedoch auf den Thread Pool mit dem Aufruf **Task**<**long**>.Factory.StartNew ausgelagert. Der wichtige Punkt ist, dass wir vor dessen Rückgabewert **await** stehen haben: dadurch wird die aktuelle Methode sofort abgebrochen, wodurch der Dispatcher auch wieder zur Ausführung kommt. Sobald der Task auf dem Hintergrundthread abgeschlossen ist (das kann übrigens auch bedeuten, dass er durch eine Exception zum Absturz gebracht wurde), wird die Methode *WennButtonGedrücktWird* erneut angesteuert und nach dem **await** Ausdruck fortgesetzt – damit hat man als Programmierer die Illusion, es handle sich hier um synchron ausgeführten Code.

Hier macht es jetzt tatsächlich Sinn, dass die Klasse **Task** Exceptions auf Hintergrundthreads automatisch fängt: diese werden nämlich erneut auf dem UI Thread durch **await** ausgelöst und können dort in **try**-**catch** Blöcken gefangen und verarbeitet werden.

Der eigentliche Grund für das Fortführen der Methode WennButtonGedrücktWird auf dem UI Thread, wenn der oben gezeigt Task endet, ist der Fakt, dass auf dem UI Thread ein sog. System.Threading.SynchronizationContext registriert ist, der eingehende Delegates an den Dispatcher weiterleitet. Würde dieser WPF SynchronizationContext nicht existieren, dann könnten in die Schleife des UI Thread keine Delegates, die von Hintergrundthreads kommen, eingereiht werden.

Das Thema [SynchronizationContext](#) ist sehr komplex und wird nicht Teil der Prüfung sein. Wenn Sie sich genauer mit dem Thema befassen wollen, empfehlen sich folgende Onlineartikel:

- <http://www.codeproject.com/Articles/31971/Understanding-SynchronizationContext-Part-I>
- <http://www.codeproject.com/Articles/32113/Understanding-SynchronizationContext-Part-II>
- <http://www.codeproject.com/Articles/32119/Understanding-SynchronizationContext-Part-III>

Als Alternative zu [async await](#) kann man die Methode [Dispatcher.BeginInvoke](#) verwenden, die einen Delegate auf dem UI Thread zur Ausführung einreihrt. Allerdings haben wir dann natürlich nicht mehr die Syntaxvorteile von ersterem und um den Exceptiontransport zum UI Thread müssen wir uns in diesem Szenario auch selbstständig kümmern. Deswegen kann ich Ihnen nur raten, [async await](#) den Vorzug zu geben.

5.20.5.5 Zusammenfassung für GUI und Threading

In den letzten Abschnitten haben wir uns ausgiebig mit dem UI Thread und Threading im Allgemeinen auseinandergesetzt. Zusammenfassend können wir folgende Aussagen treffen:

- Der UI Thread ist für das periodische Rendern der Oberfläche zuständig – langlebige Operationen sollten deshalb nicht auf ihm, sondern auf einem Hintergrundthread ausgeführt werden.
- UI Elemente sind threadaffin, d.h. die Mitglieder eines solchen Objekts können nur über den UI Thread aufgerufen werden.
- Wenn UI Elemente mit den (Zwischen-)Ergebnissen einer langlebigen Operation auf einem Hintergrundthread aktualisiert werden sollen, dann muss diese Aktualisierung wieder auf dem UI Thread ausgeführt werden. Am einfachsten geht das mit [async await](#) und der [Task](#) Klasse, [Dispatcher.BeginInvoke](#) ist allerdings auch eine Option.
- Wenn man bei mehreren Threads auf dieselben Objekte / Werte zugreift, dann kann es zu sog. Race Conditions kommen. Versuchen Sie deshalb, die Daten lokal pro Thread zu halten und nicht mit anderen Threads gleichzeitig zu teilen.

5.20.6 Wichtige Klassen in der WPF Vererbungshierarchie (Level 200)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=eIWxiss7eKg>.

Bereits in vorherigen Abschnitten habe ich erwähnt, dass WPF ein Framework ist, das sehr stark auf Vererbung setzt. Wir werden uns in diesem Abschnitt genauer mit den wichtigsten Basisklassen von WPF beschäftigen und sehen, wie die Vererbungslinie dieser Klassen die grundsätzliche Funktionalität von WPF Elemente implementiert.

In der folgenden Klassenhierarchie sehen sie die Basisklassen von WPF:

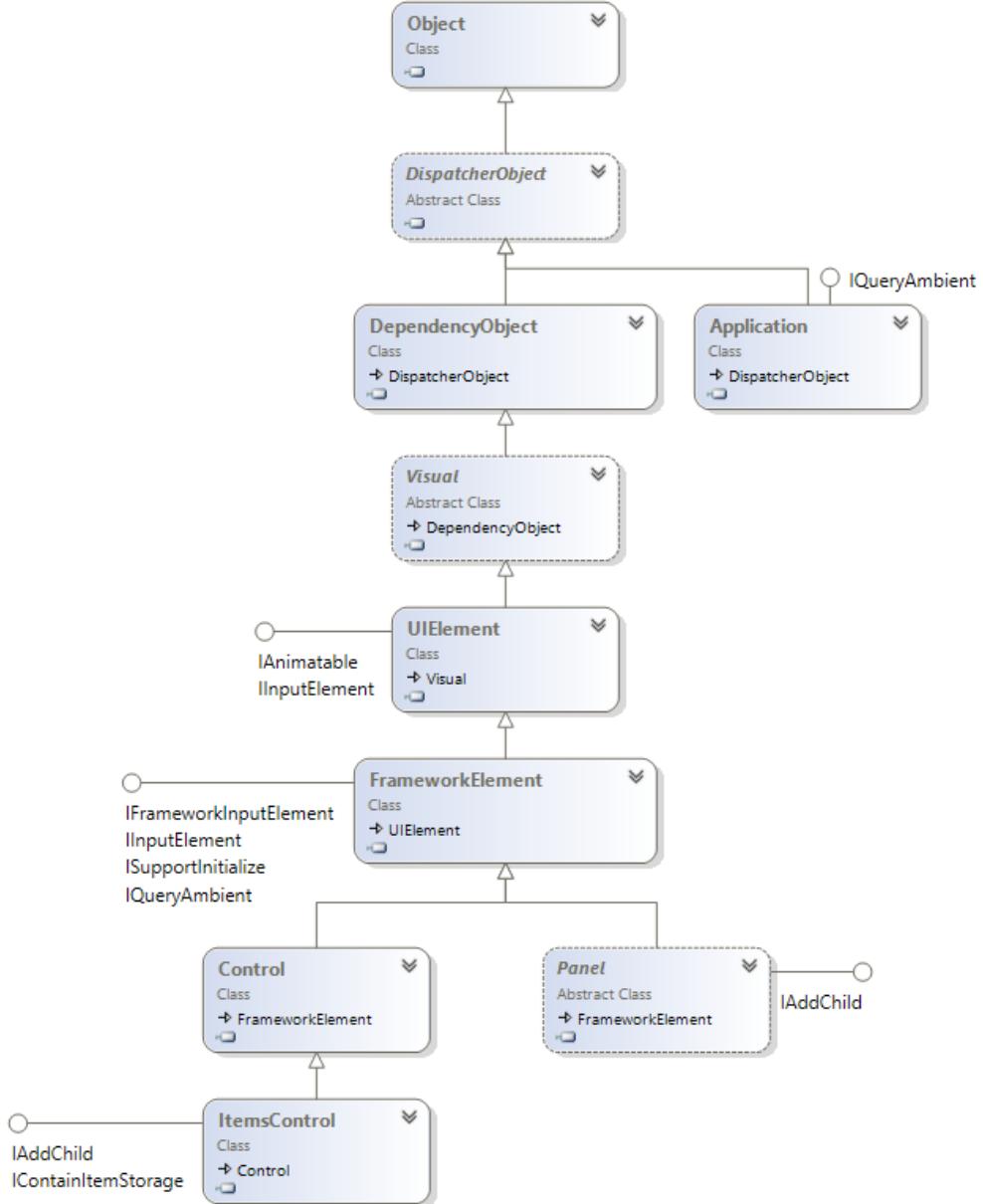


Abbildung 243: Wichtige Basisklassen in WPF

Wie sie Abbildung 243 sehen, sind es einige Klassen, welche die Basis für fast alle anderen Klassen in WPF bilden. Gehen wir diese Schritt für Schritt durch:

- **DispatcherObject** implementiert die Threadaffinität. Wenn eine Instanz davon erstellt wird, wird sie dem **Dispatcher** für den aktuellen Thread verbunden. Dazu bietet diese Klasse auch die Eigenschaft **Dispatcher** an, mit dem man auf die entsprechende Instanz des verbundenen Dispatchers zugreifen kann.
- **Application** leitet direkt von **DispatcherObject** ab und repräsentiert die WPF Applikation, die wir bereits im Abschnitt 5.20.4 besprochen haben (**App** leitet immer von **Application** ab).
- **DependencyObject** ist die Basisklasse für alle Objekte, die mit Dependency Properties ausgestattet werden können. Was Dependency Properties genau sind, sehen wir uns im nächsten Abschnitt an.
- Die Klasse **Visual** implementiert die Fähigkeit, sich selbst zu rendern. Alle Klassen, die von **Visual** ableiten, können also am WPF Renderprozess teilnehmen.

- **UIElement** leitet wiederum von Visual ab und implementiert mehrere Fähigkeiten wie die Verschachtelung von Elementen, Grundlegende Logik zur Größe und Positionierung von Elementen, Fokus und Nutzereingaben (via Maus, Tastatur, Stylus oder Touch).
- **FrameworkElement** ist die wichtigste dieser Klassen, denn sie ist die direkte oder indirekte Basisklasse für viele andere Klassen im WPF Framework. Sie implementiert u.a. Support für den WPF Layoutprozess, die Auflösung von Data Bindings und Resources in XAML, Styling und Animationen.
- **Control** ist die Basisklasse für alle Controls, die vom Nutzer zur Eingabe von Werten oder zum Auslösen bestimmter Aktionen genutzt werden können (siehe Abschnitt 5.20.2.2).
- **Panel** ist die Basisklasse für alle WPF Elemente, die beliebig viele andere WPF Elemente anordnen (siehe Abschnitt 5.20.2.3).
- **ItemsControl** ist die Basisklasse für alle Elemente, die mehrere Objekte anzeigen (diese müssen nicht **Visuals** sein) und Selektionsmechanismen für diese anbieten (siehe Abschnitt 5.20.2.4).

Warum erzähle ich Ihnen so viel über die Vererbungshierarchie in WPF? Dadurch, dass sehr viele Klassen, die in WPF häufig zum Einsatz kommen wie **Button**, **CheckBox**, **TextBox**, **Grid**, **StackPanel** und andere, die wir bereits besprochen haben, von eben genannten Klassen ableiten, können Sie natürlich auch darauf bauen, dass die Eigenschaften, Events und Methoden der Basisklassen auf all diesen WPF Elemente zur Verfügung stehen. Beispiele wären u.a.:

- Width, Height, MaxHeight, MinHeight, MaxWidth, MinWidth, Margin, Padding, VerticalAlignment und HorizontalAlignment für Layout
- Mouse Events wie MouseEnter, MouseLeave, MouseDown, MouseLeftButtonDown, MouseLeftButtonUp, MouseRightButtonDown, MouseRightButtonUp oder MouseWheel (analog dazu gibt es auch Events für Tastatur, Stylus und Toucheingaben)
- Visibility, Opacity und OpacityMask zum Ein- und Ausblenden bzw. dem durchsichtigen Darstellen (Alphakanal) von WPF Elementen
- Background, BorderBrush, BorderThickness, Foreground, FontFamily, FontStyle und FontWeight für alle Controls

Diese Liste ist nur ein kleiner Auszug der zu Verfügung stehenden Mitglieder – der Umfang oben genannter Basisklassen ist riesig.

Nehmen Sie sich an diesem Frameworkaufbau durch Vererbung dennoch kein Beispiel: WPF ist sehr schwer zu erweitern, da man sich zunächst mit all den Funktionalitäten der verschiedenen Basisklassen vertraut machen muss. Wie auch schon in Abschnitt 5.12.4 angemerkt: versuchen Sie, ihre Vererbungshierarchien grundsätzlich so flach wie möglich zu halten und setzen Sie auf Komposition statt Vererbung, um die Funktionalität von Klassen zu erweitern (nach dem Prinzip Favor Composition over Inheritance). In den meisten Fällen wird das zu flexibleren, leichter testbaren und leichter wartbaren Code führen.

5.20.7 Data Binding und Dependency Properties in WPF (Level 200)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=3EF1UKx0Guc>.

Im vorletzten Abschnitt zum Thema Oberflächenprogrammierung werden wir uns mit Data Binding und Dependency Properties beschäftigen, welche zwei fortgeschrittene Themen in WPF sind. Data Binding ist dabei der Mechanismus, mit dem man eine Eigenschaft eines Zielobjekts an die Eigenschaft eines Quellobjekts binden kann, sodass erstere aktualisiert wird, wenn sich der Wert von letzterer ändert.



Abbildung 244: Überblick über den Data Binding Mechanismus in WPF

Wie in Abbildung 244 zu sehen ist, muss bei einem WPF Data Binding die Eigenschaft des Zielobjekts zwingend eine Dependency Property sein – was das genau bedeutet, darauf werden wir gleich eingehen.

Zunächst möchte ich allerdings ein einfaches Beispiel zeigen, in dem wir sehen können, was Data Binding konkret bedeutet. Sehen Sie sich folgende Ansicht an:

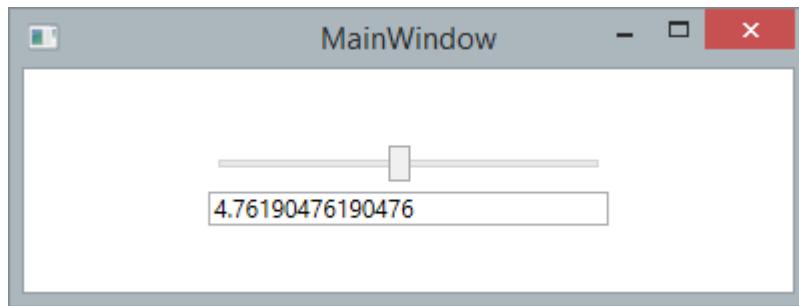


Abbildung 245: Simples Data Binding Beispiel mit TextBox und Slider

```

<UserControl x:Class="WpfExamples.HelloWorld.DataBindExample"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="550">
    <UserControl.Resources>
        <Style x:Key="ControlStyle" TargetType="Control">
            <Setter Property="Width" Value="200" />
            <Setter Property="Margin" Value="0 5" />
        </Style>
    </UserControl.Resources>
    <StackPanel VerticalAlignment="Center" HorizontalAlignment="Center">
        <Slider x:Name="MeinSlider" Style="{StaticResource ControlStyle}" />
        <TextBox Text="{Binding ElementName=MeinSlider, Path=Value}" />
    </StackPanel>
</UserControl>

```

Dieses Data Binding sorgt dafür, dass die Textbox
den aktuellen Wert des Sliders enthält.

Abbildung 246: XAML für die Ansicht aus Abbildung 245

Wie wir in Abbildung 246 sehen können, wird die **Text** Property der **TextBox** nicht einfach mit einem Wert gesetzt, sondern mit einem Data Binding versehen. Dieses sorgt dafür, dass der dargestellte Text immer dann aktualisiert wird, wenn sich der Wert des Sliders ändert. Dies ist auch in die andere Richtung möglich: wenn Sie einen Wert in die **TextBox** eingeben und den Fokus mit Tab wechseln, dann wird der Wert des Sliders aktualisiert. Der Hintergrund dafür ist, dass ein Data Binding für die **Text** Property der **TextBox** grundsätzlich in beide Richtungen funktioniert: von der Quelle zum Ziel als auch vom Ziel zur Quelle.

Im oberen Beispiel haben wir ein sog. Element Binding genutzt, da wir im Binding selbst ein anderes XAML Element über seinen Namen in **ElementName** referenzieren. Sie können bei einem Binding aber auch die Eigenschaft **Source** verwenden, um auf ein beliebiges anderes Objekt zu verweisen. Standardmäßig zeigt Source auf den **DataContext**, der über alle WPF Elemente im Darstellungsbaum weitergegeben wird.

Data Binding ist dadurch auch der wichtigste Mechanismus für das Model-View-View Model Pattern (MVVM), das sehr verbreitet im .NET und WinRT Umfeld ist. In ihm wird die Logik und die Darstellung einer Ansicht entkoppelt nach folgendem Prinzip: die sog. View enthält nur das Aussehen (im Wesentlichen XAML) und erhält in den DataContext das passende View Model zur Ansicht. Über Data Binding werden die Eigenschaften und Methoden des View Models in der View referenziert. Die eigentliche Logik wird im View Model untergebracht, eine unabhängige Klasse, die kein WPF Element repräsentiert – dadurch ist sie auch sehr leicht testbar. Das MVVM Pattern ist dabei eine Abwandlung des Model-View-Controller (MVC) bzw. Model-View-Presenter (MVP) Pattern.

Genauere Beispiele zu Data Binding und MVVM finden Sie in den Videos zu diesem Abschnitt. Wenn Sie sich darüber hinaus über diese Themen informieren möchten, kann ich Ihnen folgende Artikel empfehlen:

- Data Binding Übersicht: [http://msdn.microsoft.com/en-us/library/ms752347\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms752347(v=vs.110).aspx)
- MVVM Explained:
<http://www.codeproject.com/Articles/100175/Model-View-ViewModel-MVVM-Explained>

Für den Rest dieses Abschnitts möchte ich noch kurz auf Dependency Properties eingehen – diese sind nämlich ein wichtiger Bestandteil von WPF. Dabei können diese, wie der Name schon sagt, tatsächlich in Verbindung mit normalen .NET Properties gebracht werden, denn diese sollen durch Dependency Properties ersetzt werden. Sie werden dabei durch die Klasse **System.Windows.DependencyProperty** repräsentiert und können in Klassen, die von **System.Windows.DependencyObject** ableiten, eingesetzt werden.

Der wichtigste Vorteil, den Dependency Properties bieten, ist der, dass Werte für diese auf jedem Dependency Object gesetzt werden können, egal ob dieses Objekt die Eigenschaft dazu definiert oder nicht. Das ist bei normalen Eigenschaften nicht möglich, sorgt aber beispielsweise dafür, dass wir einen **TextBlock** mit einem Wert via Grid.Row mit der Information ausstatten können, in welcher Zeile im Grid dieser liegen soll.

Dependency Properties werden auch etwas anders definiert als normale Eigenschaften in .NET. Sehen wir und dazu als Beispiel die **TextBlock.Text** Eigenschaft an, die wir bereits häufig in unseren Beispielen eingesetzt haben:

```

namespace System.Windows.Controls
{
    public class TextBlock : FrameworkElement
    {
        public static readonly DependencyProperty TextProperty =
            DependencyProperty.Register("Text",
                typeof(string),
                typeof(TextBlock),
                /* ... weitere Parameter entfernt */);

        public string Text
        {
            get { return (string) GetValue(TextProperty); }
            set { SetValue(TextProperty, value); }
        }

        // Weitere Mitglieder zur Vereinfachung entfernt
    }
}

```

Dependency Properties werden standardmäßig als public static readonly Feld definiert

Um eine Dependency Property existiert ein normale .NET Property, welche den Zugriff auf den Wert eines entsprechenden Objekts regelt.

Abbildung 247: Definition der Dependency Property Text in der Klasse TextBlock

Wie wir in Abbildung 247 sehen, werden Dependency Properties standardmäßig als Feld definiert, dass `public static readonly` ist. Dieses Feld wird dann als Parameter eingesetzt zum Setzen bzw. Auslesen von Werten bei den Methoden `DependencyObject.SetValue` und `DependencyObject.GetValue` (siehe Wrappereigenschaft `Text`). Diese Methoden sind leider nicht generisch, sodass wir bei `GetValue` zum entsprechenden Typ casten müssen.

Der Punkt ist, dass fast alle Eigenschaften der WPF Elemente als Dependency Properties implementiert sind. Sie bieten dadurch folgende Vorteile:

- Automatische Benachrichtigung bei Wertänderungen: Dependency Properties teilen mit, wenn sich der entsprechende Wert auf einem Objekt geändert hat.
- Data Binding Support: Dependency Property können als Ziel für WPF Data Bindings verwendet werden.
- Styling Support: Dependency Property Werte können über WPF Styles gesetzt werden.
- Resource Support: Dependency Properties können ihre Werte aus den Ansichts- oder Applikationsressourcen beziehen.
- Attached Properties: Werte können auch auf `DependencyObjects` gesetzt werden, welche die entsprechende Dependency Property gar nicht definieren.
- Animationen: Dependency Properties können mit dem WPF Animationssystem ihren Wert über einen gewissen Zeitraum ändern.

5.20.8 Zusammenfassung für WPF und Oberflächenprogrammierung

Das Video hierzu finden Sie unter https://www.youtube.com/watch?v=RsmRqS_6B3Q.

In den letzten Abschnitten haben wir einen kleinen Einblick in WPF bekommen und gesehen, dass sich Oberflächenprogrammierung fundamental von der Programmierung in anderen Bereichen

unterscheidet. Allerdings haben wir auch viele fortgeschrittene WPF Themen wie bspw. Data Binding, Dependency Properties, Routed Events, Styles und Resources, Control- und Data Templates, Multithreading sowie Animationen nur im kleinen Umfang oder gar nicht behandelt. Wenn Sie mehr über diese Dinge lernen möchten, kann ich Ihnen die folgenden Bücher empfehlen:

- Pro WPF in C# von Matthew MacDonald
(Apress Verlag, englisch):
<http://www.apress.com/9781430243656>
- Windows Presentation Foundation von Thomas Claudius Huber
(Galileo Computing Verlag, deutsch)
https://www.galileo-press.de/windows-presentation-foundation-45_3179/

Wahlweise können Sie natürlich auch in die WinRT oder Windows Phone Entwicklung einsteigen, bei denen ebenfalls der Einsatz von XAML / C# möglich ist und deren Projekte sehr ähnlich zu WPF Projekten aufgebaut ist.

Zusammenfassend können wir jedoch folgende Dinge feststellen:

- Ein Fenster bzw. eine Ansicht besteht in WPF aus einer XAML und einer Code-Behind-Datei, die zusammen eine Klasse formen. In ersterer wird das Aussehen der Oberfläche beschrieben, in letzterer kann Code stehen, der hauptsächlich dynamisch zur Laufzeit nach dem Hollywood-Prinzip von Events der WPF Elemente aufgerufen wird.
- Fenster bzw. Ansichten sind aus mehreren WPF Elementen aufgebaut, wobei Panels das Layout weiterer Elemente bestimmen. Grundsätzlich sollten wir Panels einsetzen, die ihre Kindelemente automatisch anordnen, anstatt letztere absolut zu positionieren. Alle Elemente einer Ansicht bilden eine Baumhierarchie, die wir in WPF in XAML ausdrücken.
- Eine Applikation mit User Interface besitzt einen UI Thread, auf dem ein Schleifenkonstrukt installiert wird. Innerhalb dieser Schleife wird immer wieder der Renderprozess angestoßen sowie Betriebssystemnachrichten und Nutzereingaben verarbeitet bzw. weitergeleitet.
- Wenn wir langlebige Operationen auf dem UI Thread ausführen, dann sorgen erstere dafür, dass letzterer blockiert wird und damit die Oberfläche nicht mehr zeichnen kann. Dieses „Einfrieren“ des UI Interfaces sollte unter allen Umständen vermieden werden, bspw. indem man solche Operationen auf einen Hintergrundthread auslagert.
- Vorsicht: Zugriff auf WPF Elemente ist nur auf dem UI Thread gestattet, was man auch als Threadaffinität bezeichnet. Wenn Sie bei Hintergrundthread-Operationen die Werte von WPF Elemente aktualisieren möchten, müssen Sie diese Anweisungen erst wieder in die Schleife des UI Threads einreihen. In WPF geht das mit `Dispatcher.BeginInvoke` bzw. `async await`.
- WPF setzt intern stark auf Vererbung und bietet mit Dependency Properties Support für wichtige UI Techniken wie Data Binding, Styling, Resources und Animationen. Diese Funktionalität ist mit normalen .NET Eigenschaften nicht ohne weiteres möglich.

6 Objektorientiertes Design (OOD)

In diesem Abschnitt sehen wir uns die grundlegenden Säulen der objektorientierten Programmierung (OOP) sowie allgemeine Muster und Prinzipien an, die Sie auf alle objektorientierten Sprachen anwenden können. Einerseits sind diese Themen notwendig, um objektorientierte von nicht-objektorientierten Sprachen abzugrenzen. Andererseits ist der wichtigere Punkt aber, dass Sie sich an ihnen orientieren können, um mittelgroße bis große Softwarelösungen zu erstellen, die flexibel, leicht wartbar und leicht testbar sind.

6.1 Die Abgrenzung zwischen objektorientiert und nicht-objektorientiert

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=87rPw8F5MqM>.

Zunächst möchte ich darauf eingehen, wo die wesentlichen, allgemein anerkannten Unterschiede zwischen objektorientierten und nicht-objektorientierten Sprachen liegen. Diese sind:

- Kapselung
- Vererbung
- Polymorphie

Diese drei Begriffe bilden die Grundsäulen für Objektorientiertes Programmieren.

Zur Wiederholung der ersten beiden Begriffe: Kapselung bedeutet im Wesentlichen, dass ein Teilproblem komplett durch eine Klasse erfasst und gelöst wird. Dazu stellt die Klasse eine einfache, leicht verständliche API bereit, die ein Nutzer der Klasse einsetzen kann, um das Problem zu lösen, ohne wirklich zu verstehen, was genau bei der Problemlösung passiert. Eine solche Klasse ist aus Daten, die sie zur Lösung des Problems braucht, und Methoden, die auf diesen Daten operieren, aufgebaut und sollte Höhe über die Daten haben – das bedeutet nichts anderes, als dass Felder, denen entsprechende Werte zugewiesen wurden (dies können auch Referenzen auf andere Objekte sein), grundsätzlich **private** sind. Außerhalb des Klassenscopes ist kein Zugriff auf diese Felder möglich – außer die Klasse erlaubt dies durch bestimmte Methoden (in C# üblicherweise Eigenschaften). Innerhalb der Klasse sind Felder aber so etwas wie globale Variablen: sie können von allen Mitgliedern (sofern man den statischen und nicht-statischen Kontext beachtet) jederzeit angesprochen werden und stellen in vielen Fällen auch den Status eines Objekts dar. Genauere Infos zur Kapselung finden Sie in Abschnitt 5.7.

Vererbung bedeutet grundsätzlich erstmal, dass eine Klasse von einer anderen Klasse ableiten kann, um dadurch sämtliche Members der Basisklasse, die nicht **private** sind, ebenfalls nutzen zu können. Dadurch kann die API der Basisklasse erweitert werden – das ist jedoch nicht der Haupteinsatzgrund für Vererbung. Wenn die Basisklasse nämlich virtuelle oder abstrakte Methoden bereitstellt, dann können diese in einer Subklasse außer Kraft gesetzt und durch eine neue Implementierung ausgetauscht werden (in C# mit dem Schlüsselwort **override**). Zusammen mit der Möglichkeit, Subklassen über die Ist-Eine-Beziehung auch als Referenz ihrer Basisklassen zu interpretieren, haben wir so die Möglichkeit, Funktionalität durch den Austausch von Objekten zu ändern, ohne dass wir auf der Nutzerseite eine einzige Zeile Code anfassen müssen.

Polymorphie (dt. Mehrgestaltigkeit) bedeutet nichts anderes, als die Funktionalität eines Programms zu ändern oder zu erweitern, indem man Objekte austauscht, ohne dass die Nutzerseite diesen Austausch bemerkt. Dabei ist hier die Ist-Eine-Beziehung der wichtige Part – Polymorphie ist aber auch über andere Mechanismen wie bspw. Funktionszeiger oder Generics möglich.

Polymorphie drückt sich eindeutig in der Verwendung von puren abstrakten Basisklassen oder Interfaces in C# aus: beide sind nämlich Abstraktionen. Abstraktionen zeichnen sich dadurch aus, dass Sie selbst nicht als Objekt instanziert werden können (konkret bedeutet das, dass abstrakte Klassen wie auch Interfaces nicht mit `new` instanziert werden können), sie beschreiben aber, wie die API einer Klasse / Struktur auszusehen hat, welche sich an die Abstraktion hält.

Ein Beispiel für Polymorphie sehen wir uns gleich in einem der nächsten Abschnitte an. Die abschließende Frage, den ich in diesem Abschnitt stellen möchte: ist die Programmiersprache C eine objektorientierte Sprache? Sind dort Kapselung, Vererbung und Polymorphie möglich? Die Antwort ist grundsätzlich Nein, da C die entsprechenden Funktionalitäten wie Zugriffsmodifizierer oder das Ableitungszeichen : nicht direkt zur Verfügung stellt. Das heißt aber nicht, dass objektorientiertes Programmieren in C gänzlich unmöglich ist – allerdings sind Umwege nötig, die häufig im Einsatz von Makros enden.

Wenn Sie eine neue Programmiersprache lernen und feststellen möchten, ob diese objektorientiert ist oder nicht, dann können Sie folgende Fragen dazu verwenden:

- Gibt es Konstrukte, mit denen ich Funktionalität kapseln kann (inkl. Datenhoheit)?
- Gibt es einen Vererbungsmechanismus, sodass ein Konstrukt / Objekt die Fähigkeiten eines anderen erhält / erbt?
- Wie kann ich Abstraktionen erstellen und diese für polymorphe Zwecke einsetzen?

6.2 SOLID Principles of Object-Oriented Design

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ipVj7NQsy8M>.

Nachdem wir uns im letzten Abschnitt mit der Abgrenzung von objektorientierten zu nicht-objektorientierten Sprachen Gedanken gemacht haben, sprechen wir in diesem über SOLID, einen fundamentalen Satz an Designprinzipien, die man als objekt-orientierter Programmierer befolgen kann und sollte, um mittelgroße bis große Softwarelösungen so umzusetzen, dass der dazugehörige Source Code „gut“ ist.

Es folgt sofort die Frage: was ist „guter“ Code? Natürlich kann man hier anfügen, dass gut ein relativer (womöglich auch noch moralisch behafteter) Ausdruck ist, dennoch herrscht heutzutage einigermaßen Einigkeit in der objekt-orientierten Entwicklergemeinde darüber.

„Guter“ Source Code zeichnet sich wie folgt aus:

- Er ist leicht verständlich für andere Programmierer.
- Er ist leicht erweiterbar / änderbar (flexibel) – auf neue Anforderungen oder die Entdeckung von Fehlern kann also schnell reagiert werden.
- Er ist leicht testbar – ein einzelnes Objekt sollte unabhängig von anderen Objekten, die es referenziert, in speziellen Testszenarien überprüft werden können.

Bitte beachten Sie auch, dass in dieser Aufzählung „Guter Code ist performant“ explizit nicht erwähnt ist. Als Softwareentwickler sollte man sich (außer in Spezialgebieten wie bspw. dem High Computing Bereich) zunächst keine Gedanken über Performance machen, da optimierter Source Code häufig nicht leicht verständlich ist.

Doch zurück zum eigentlichen Thema. SOLID ist dabei eine Gedächtnisstütze für die folgenden Prinzipien:

- Single Responsibility Principle
- Open / Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Die Anwendung dieser Prinzipien kann zu leicht verständlichen, flexiblen und testbaren Code führen – das muss aber nicht immer der Fall sein. Genauso wie Designprinzipien, die wir in einem späteren Kapitel ansprechen werden, sind die SOLID Prinzipien nur Guidelines und stellen keine vorgeschriebenen Gesetze dar. Diese Prinzipien wurden von Robert C. Martin, einem sehr bekannter amerikanischen Softwareentwickler ca. im Jahr 2000 aufgestellt als die sog. „First Five Principles“ für objektorientierte Programmierung – das Akronym SOLID wurde später von Michael Feathers verliehen. Wir werden uns diese fünf Prinzipien in den folgenden Abschnitten im Detail anschauen, allerdings nicht in der oben präsentierten Reihenfolge.

Vorsicht: die SOLID Prinzipien sind genau wie viele Design Patterns alles andere als intuitiv und viele Programmierer kennen sie nicht. Das macht sie zu „gefährlichem“ Wissen in der Hinsicht, dass bspw. Ihre zukünftigen Arbeitskollegen ihren Code nicht verstehen, wenn sie diese Prinzipien anwenden.

6.2.1 Dependency Inversion Principle (DIP)

Als erstes der SOLID Prinzipien werden wir uns das Dependency Inversion Principle anschauen. Um dies verständlicher zu gestalten, werden wir anhand eines Beispiels das DIP herleiten.

6.2.1.1 Kopierprogramm als Ausgangsbeispiel für das DIP

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=sHs3IAx7hB0>.

Nehmen wir an, wir möchten eine Komponente schreiben, die Eingaben vom Nutzer entgegennimmt und diese auf der Konsole direkt wieder ausgibt. Diese Komponente nennen wir Copy, die grundlegende Funktionalität zum Einlesen ist uns über `Console.ReadKey` und `Console.Write` bereits bekannt. Der Code dazu könnte wie folgt aussehen:

```
public class Program
{
    public static void Main()
    {
        Copy();
    }

    public static void Copy()
    {
        var sollEnden = false;
        while (sollEnden == false)
        {
            var tastenInfo = Console.ReadKey(true);
            if (tastenInfo.Key == ConsoleKey.Escape)
            {
                sollEnden = true;
                continue;
            }
            Console.Write(tastenInfo.KeyChar);
        }
    }
}
```

Abbildung 248: Das Copy Programm

Die Struktur des Programms kann man dabei wie in folgender Abbildung visualisieren:

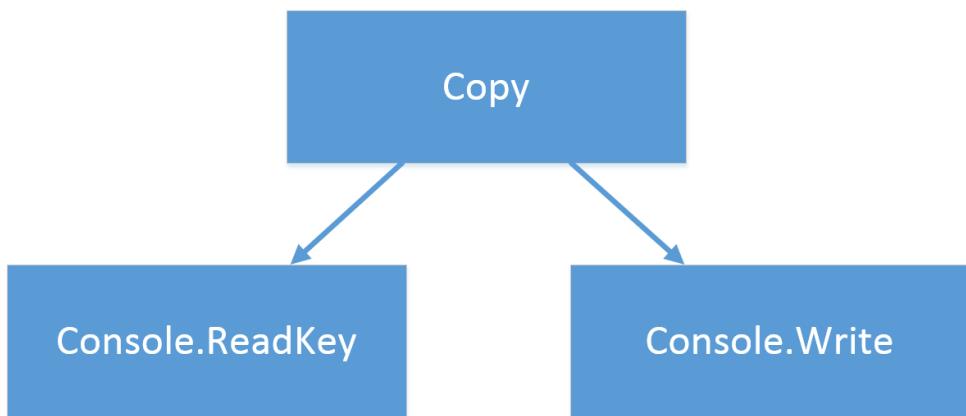


Abbildung 249: Die Methode Copy referenziert sowohl Console.ReadKey als auch Console.Write

In Abbildung 249 sind die drei Module (oder Komponenten) Copy, `Console.ReadKey` und `Console.Write` zu sehen. Copy wird dabei als High Level Module bzw. High Level Component bezeichnet, da sie die beiden Low Level Modules `Console.ReadKey` und `Console.Write` referenziert und in sich aufruft. Genau deshalb wurden auch die Pfeile vom High Level Module zu den beiden Low Level Modules gezogen.

Lassen Sie sich vom neuen Begriff Modul (bzw. Komponente) nicht abschrecken. Ein Modul umfasst im Kontext der Programmierung ein oder mehrere Methoden / Klassen, die zusammen eine Funktionalität bereitstellen. Wenn ein Modul andere Module nutzt, dann spricht man von einem „Higher Level Module“, das mehrere andere Funktionalitäten orchestriert. Modul und Komponente werden in diesem Script als identische Begriffe verwendet.

Bitte beachten Sie: dieses Beispiel ist bewusst klein gehalten, damit es leichter verständlich ist. Mit dem tatsächlichen Programmieralltag hat es nichts zu tun.

6.2.1.2 Das Problem der direkten Abhängigkeit

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ZN3x-TLTuG4>.

Unser Beispiel aus dem vorherigen Abschnitt ist zwar lauffähig, allerdings haben wir das Problem, dass `Copy` nicht wiederverwendbar ist, da diese Methode direkte Abhängigkeiten zu `Console.ReadKey` und `Console.Write` besitzt. Nehmen wir an, wir möchten die Ausgabe optional auch in eine Datei schreiben – in diesem Fall können wir die `Copy` Komponente nicht wiederverwenden, ohne ihren Funktionskörper zu ändern und neu zu kompilieren. Das mag vielleicht in unserem Beispiel jetzt einfach sein – aber Kompilieren und Deployment kann ein sehr schwieriger und langandauernder Prozess sein bei großen Softwareprojekten.

Um eine optionale Ausgabe in eine Datei zu machen, könnte man das `Copy` Modul wie folgt anpassen:

```

public static void Copy(Ausgabeziel ausgabeziel)
{
    var sollEnden = false;
    while (sollEnden == false)
    {
        var tastenInfo = Console.ReadKey(true);
        if (tastenInfo.Key == ConsoleKey.Escape)
        {
            sollEnden = true;
            continue;
        }
        if (ausgabeziel == Ausgabeziel.Konsole)
            Console.Write(tastenInfo.KeyChar);
        else
            Datei.Schreibe(tastenInfo.KeyChar);
    }
}

```

Abbildung 250: Modifizierte Variante von Copy, in der Dateiausgabe ebenfalls möglich ist

In Abbildung 250 sehen wir, dass am Ende der Methode Copy ein **if else** Block hinzugefügt wurde, in dem je nach Parameterangabe zur Ausgabe die Konsole oder eine Datei angesteuert wird. Folgende Probleme können dabei aber auftauchen:

- Wenn wir andere Ausgabeziele hinzufügen möchten, müssen wir in der Copy-Methode den **if else** Block anpassen und dort die entsprechende Funktionalität aufrufen.
- Wenn wir andere Eingabemethoden (bspw. das Auslesen eines Netzwerkstreams) ermöglichen möchten, müssen wir einen weiteren **if else** Block für das Einlesen erstellen und die jeweilige Funktionalität aufrufen.
- Im Endeffekt landet man in großen **if else** Blöcken (oder wahlweise **switch** Statements), welche die jeweiligen Aufrufe je nach Kontext machen. Diese Blöcke zerstören die Lesbarkeit und Einfachheit des Copy Moduls – dadurch ist es schlechter wartbar.

6.2.1.3 Programmieren gegen Abstraktionen als Lösung des Problems

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=iF9ktxl7leA>.

Um dieses Problem zu lösen, kann man das Dependency Inversion Principle einsetzen, das folgendes besagt:

Dependency Inversion Principle (DIP)

1. High Level Modules sollten nicht abhängig sein von Low Level Modules. Beide sollten von Abstraktionen abhängen.
2. Abstraktionen sollten keine Abhängigkeiten zu Details haben, sondern Details zu Abstraktionen.

Das bereits früher im Script erwähnte Prinzip der Programmierung gegen Abstraktionen ist also nichts anderes als die Anwendung des Dependency Inversion Principles. Konkret heißt das in diesem Beispiel, dass Copy nicht **Console**.ReadKey und **Console**.Write direkt aufrufen darf, sondern diese Aufrufe über Abstraktionen durchführt.

Abstraktionen sind in C#, wie bereits öfters erwähnt, entweder abstrakte Klassen oder Interfaces. Für diesen Fall werde ich letztere verwenden, die wie folgt aussehen könnten:

```

public interface IReader
{
    ReadResult ReadElement();
}

public interface IWriter
{
    void Write(char character);
}

public class ReadResult
{
    public char Character { get; set; }
    public bool IsLastCharacter { get; set; }
}

```

IReader stellt die Abstraktion dar für Console.ReadKey
IWriter ist die Abstraktion für Console.Write
ReadResult repräsentiert eine Datenklasse, die einheitlich von allen IReader-Implementierungen genutzt werden soll

Abbildung 251: Die Abstraktionen zum Copy Beispiel

Diese in Abbildung 251 zu sehenden Interfaces müssen natürlich in zwei Klassen implementiert werden:

```

public class ConsoleReader : IReader
{
    public ReadResult ReadElement()
    {
        var tastenInfo = Console.ReadKey(true);
        return new ReadResult
        {
            Character = tastenInfo.KeyChar,
            IsLastCharacter = tastenInfo.Key == ConsoleKey.Escape
        };
    }
}

public class ConsoleWriter : IWriter
{
    public void Write(char character)
    {
        Console.Write(character);
    }
}

```

Die Funktionalität von ReadElement liest eine Tasteninfo ein und erstellt ein ReadResult-Objekt aus den Infos
Write delegiert den Aufruf einfach an Console.Write weiter

Abbildung 252: Implementierungen zu den Interfaces IReader und IWriter

Damit wir diese beiden neuen Abstraktionen im Copy–Modul nutzen können, müssen wir sie wie folgt umgestalten:

Die Instanzen zu reader und writer
werden als Parameter übergeben

```

public static void Copy(IReader reader, IWriter writer)
{
    var sollEnden = false;
    while (sollEnden == false)
    {
        var readResult = reader.ReadElement();
        if (readResult.IsLastCharacter)
        {
            sollEnden = true;
            continue;
        }
        writer.Write(readResult.Character);
    }
}

```

Abbildung 253: Umformung von Copy nach dem DIP

In Abbildung 253 sehen wir, dass zur Methode Copy zwei Parameter hinzugefügt wurden, welche die Abhängigkeiten referenzieren. Diese werden natürlich über die Interfaces `IReader` und `IWriter` angesprochen. Genau das ist auch der Grund, warum sie übergeben werden müssen, denn:

Wenn Sie innerhalb einer Klasse oder Methode gegen Abstraktionen programmieren, dann müssen Sie sich die entsprechenden Objekte dazu von außen übergeben lassen, denn Abstraktionen (also abstrakte Klassen oder Interfaces) können nicht direkt instanziert werden. Dieses Vorgehen des Übergebens von Abhängigkeiten bezeichnet man als Dependency Injection.

Im obigen Beispiel sehen wir die konkrete Form der Method Injection, da alle Abhängigkeiten von Copy als Parameter in die Methode übergeben werden. Das ist an sich vollkommen ok, allerdings ist Copy selbst noch eine statische Methode, die in der Klasse `Program` liegt. Das heißt, dass zu Copy selbst keine Abstraktion erstellt werden kann – statische Methoden sind als Implementierung von Interfaces bzw. beim Überschreiben von virtuellen oder abstrakten Methoden nicht erlaubt.

Damit wir Copy tatsächlich als vollwertiges Modul ansprechen können, werden wir es in eine eigene Klasse überführen:

```

public class CopyProcess
{
    private readonly IReader _reader;
    private readonly IWriter _writer;

    public CopyProcess(IReader reader, IWriter writer)
    {
        _reader = reader;
        _writer = writer;
    }

    public void Copy()
    {
        var sollEnden = false;
        while (sollEnden == false)
        {
            var readResult = _reader.ReadElement();
            if (readResult.IsLastCharacter)
            {
                sollEnden = true;
                continue;
            }
            _writer.Write(readResult.Character);
        }
    }
}

```

Method Injection wurde durch
Constructor Injection ersetzt

Copy ist jetzt in einer eigens
dafür geschaffenen Klasse und
nicht mehr statisch

Abbildung 254: Das Copy Modul ist jetzt in einer eigenen Klasse angesiedelt

Wie wir in Abbildung 254 sehen können, ist die Methode Copy jetzt Teil der Klasse `CopyProcess` und nicht mehr statisch. Dadurch ist es auch möglich, die Method Injection aus der vorherigen Abbildung durch sog. Constructor Injection auszutauschen – bevorzugen Sie diese wenn möglich, denn so können Nutzer ihrer Klassen sofort beim Instanziieren ansehen, welche Abhängigkeiten dieses Objekt hat. Die Abhängigkeiten werden den entsprechenden Feldern zugewiesen und können dann innerhalb der Copy Methode, die jetzt wieder ohne Parameter auskommt, eingesetzt werden.

Die Struktur hat sich durch unsere Anpassungen wie folgt geändert:

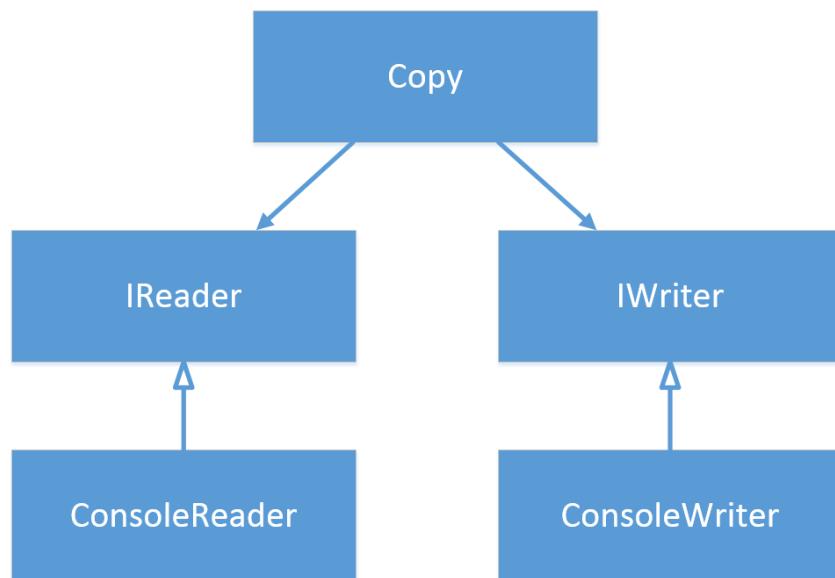


Abbildung 255: Struktur von Copy nach DIP

Wie Sie in Abbildung 255 sehen können, ist Copy nicht mehr direkt von den Low-Level-Komponenten `Console.ReadKey` und `Console.WriteLine` abhängig, sondern von Abstraktionen. Die Abstraktionen müssen jeweils in wenigstens einer konkreten Klasse implementiert sein, allerdings haben wir dadurch den Vorteil der Wiederverwendbarkeit für Copy, denn wir können `IReader` und `IWriter` in beliebigen anderen Klassen implementieren, welche Werte nach einer anderen Vorgehensweise bereitstellen bzw. verarbeiten und sie dann dem `CopyProcess`-Objekt bei der Instanzierung über Dependency Injection bereitstellen. Genau das ist Polymorphie – wir können Objekte beliebig austauschen, ohne dass Copy den Unterschied bemerkt.

Insgesamt kann man sagen, dass die sog. **Kopplung** zwischen Copy und anderen Komponenten, die Copy einsetzt, abgenommen hat durch die eingeführten Interfaces. In diesem Zusammenhang spricht man auch von loser Kopplung (engl. loose Coupling).

6.2.1.4 Zusammenfassung für DIP

Sie können das Dependency Inversion Principle einsetzen, um Higher Level Components unabhängig von den von Ihnen eingesetzten Objekten zu halten und damit die Wiederverwendbarkeit und Flexibilität dieser Komponenten zu erhöhen. Setzen Sie das DIP aber nicht blind ein – bei kleinen Softwarelösungen ist es nicht immer vorteilhaft, da durch die zusätzlichen Abstraktionen die Programmstruktur zunächst komplizierter wird. Gerade wenn ihre Lösungen aber größer werden oder Sie Frameworks oder Bibliotheken entwickeln, sollten Sie das DIP häufig einsetzen.

Achten Sie dabei auf folgende Punkte:

- Programmieren Sie in Ihren Klassen gegen Abstraktionen, nicht gegen konkrete Typen. Das bedeutet den Einsatz von Interfaces oder abstrakten Klassen. Erstellen Sie keine Objekte direkt mit `new` und rufen Sie keine statischen Methoden auf.
- Lassen Sie sich konkrete Objekte zu den geforderten Abstraktionen über Dependency Injection übergeben – am besten via Constructor Injection, denn so sieht ein Nutzer Ihrer Klasse sofort bei der Instanzierung, welche Abhängigkeiten er liefern muss, damit das Objekt problemfrei funktioniert.
- Constructor Injection kann durch andere Dependency Injection Mechanismen wie Method Injection oder Property Injection ersetzt werden – in diesem Fall sollten Sie allerdings wissen, was Sie tun und warum Sie es tun. Wenn Sie keine Ahnung haben, benutzen Sie Constructor Injection.
- Üblicherweise werden die Abhängigkeiten zwischen verschiedenen Objekten an genau einer Stelle aufgelöst: dem Composition Root, den man im Normalfall an den Beginn eines Programms setzt, bevor die eigentliche Funktionalität ausgeführt wird. Hier werden alle Objekte erstellt und via Dependency Injection miteinander verknüpft, die für den (korrekten) Ablauf des Programms notwendig sind. In großen Projekten kann man das nicht nur händisch tun, sondern einen Dependency Injection Container (DI-Container) einsetzen, der diese Aufgabe via Reflection löst.

Das DIP und Dependency Injection ist meines Erachtens das wichtigste der SOLID Prinzipien. Wenn Sie mehr zum Thema erfahren möchten, kann ich Ihnen das sehr gute Buch [Dependency Injection in .NET](#) von Mark Seemann empfehlen. Weiterhin können Sie auch hin und wieder auf seinem Blog [blog.ploeh.dk](#) vorbeischauen, da er aktiv zu diesem Thema postet.

6.2.2 Single Responsibility Principle (SRP)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=DFosy5C56r8>.

Das Single Responsibility Principle ist wie folgt definiert:

Single Responsibility Principle

Jedes Objekt sollte genau eine eindeutige Aufgabe haben und diese Aufgabe sollte von der entsprechenden Klasse komplett gekapselt sein.

Es sollte niemals mehr als einen Grund geben, wegen dem sich die Implementierung einer Klasse ändern kann.

Wir werden uns im nächsten Abschnitt ein Beispiel zum SRP anschauen, aber ich werde hier bereits einige Richtlinien zur Erfüllung dieses Prinzips festhalten:

- Das SRP sorgt hauptsächlich dafür, dass Klassen nicht zu groß werden, da man vermeidet, mehrere nicht-zusammenhängende Probleme in einer Klasse zu lösen
- Durch diese feinere Granularität (im Idealfall löst jede Klasse genau ein Problem) steigt zwar die Gesamtzahl der Klassen, dafür kann eine einzelne Klasse leichter wiederverwendet werden. Viele kleine Klassen mit eindeutigen Aufgaben tendieren zu einem flexibleren Design als wenige große Klassen.
- Eine Klasse, die nach dem SRP implementiert ist, hat wahrscheinlich weniger Abhängigkeiten zu anderen Objekten als große Klassen, die mehrere Aufgaben abdecken – damit ist die Kopplung zwischen Typen geringer.
- Ebenfalls interessant ist in diesem Kontext der Begriff **Kohäsion** (engl. Cohesion), der aussagt, wie oft ein Feld innerhalb der Methoden einer Klasse angewandt wird. Wird bspw. ein Feld bei großen Klassen in nur einer von zehn Mitgliedsmethoden genutzt, dann spricht man von niedriger Kohäsion. Klassen, die nach dem SRP implementiert sind, tendieren zu einer hohen Kohäsion (was wünschenswert ist).

6.2.3 Open / Closed Principle (OCP)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=D7QAFXnbAWs>.

Das Open / Closed Principle ist wie folgt definiert:

Open / Closed Principle

Softwarekomponenten (Klassen, Funktionen, etc.) sollten offen sein für Erweiterungen, aber geschlossen für Veränderungen.

Neue Funktionalität bzw. neues Verhalten kann zu späteren Zeitpunkten hinzugefügt werden, ohne dabei den Source Code von bestehenden Klassen verändern zu müssen (ausgenommen ist dabei der Composition Root).

Der Schlüssel zur Erfüllung des OCP ist auch hier der Einsatz von Abstraktionen. Am besten lässt sich das an einem Beispiel zeigen.

Sehen Sie sich den Ausgangs-Code zu Übung 4 (Shop zur Vergoldeten Rose) an: in der Methode `AktualisiereArtikelqualität` sind sehr viele ineinander verschachtelte `if else` Blöcke enthalten, die je nach Artikelzugehörigkeit den Qualitätswert eines Artikels anpasst. Ist diese Methode nach dem OCP implementiert? Nein, natürlich nicht – denn Anpassungen können im jetzigen Fall nur gemacht werden, indem man den Source Code von dieser Methode direkt ändert. Diese Methode gilt auch als Beispiel für die Verletzung des SRP: da diese Methode weiß, wie die Qualität alle gerade bestehenden Artikelgruppen geändert wird, gibt es mehrere Gründe, weshalb sie sich ändern kann.

Die Lösung dieses Problems sind wir so angegangen, dass eine Abstraktion `Artikel` erstellt wurde, mit der der Aufruf von `AktualisiereArtikelqualität` in mehrere Klassen ausgelagert wurde. Die eigentliche Methode hat dann nur noch die Liste von Artikeln durchlaufen und die entsprechende Methode auf jedem Artikel aufgerufen. Dadurch kam in etwa folgende Struktur zustande:

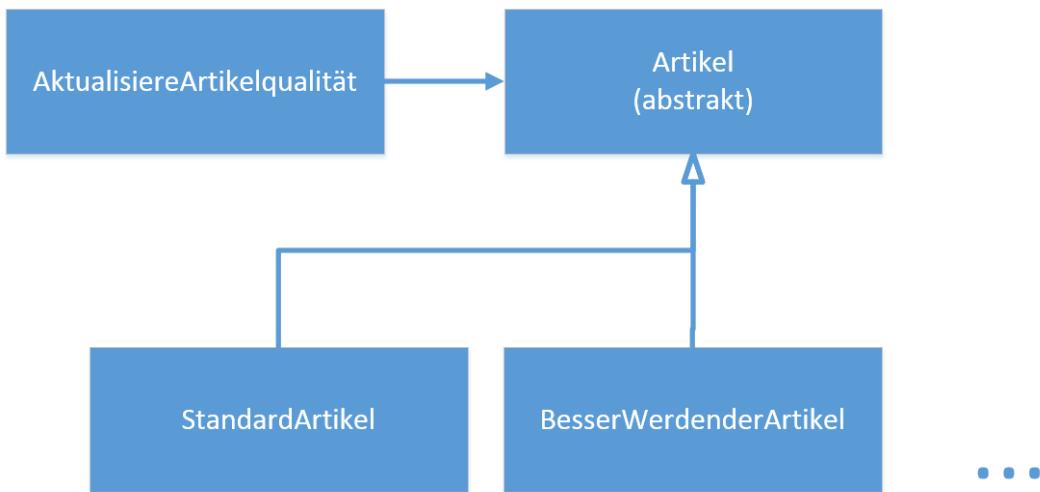


Abbildung 256: Struktur zur Erfüllung des OCP bei Shop zur Vergoldeten Rose

Die drei Punkte in Abbildung 256 deuten dabei an, dass noch mehrere Klassen von der Abstraktion `Artikel` ableiten. Durch die Einführung dieser Abstraktion können wir die `AktualisiereArtikelqualität` Funktionalität auch zu einem späteren Zeitpunkt erweitern, indem wir von `Artikel` eine neue Klasse ableiten und für die entsprechenden Members nach einem anderen Prinzip implementieren. Dabei müssen wir keine der bestehenden Komponenten, die wir oben im Bild sehen, anpassen.

Das Open / Closed Principle findet im Programmieralltag am häufigsten da Anwendung, wo große if else Blöcke (bzw. große switch Statements) zu finden sind. Diese sind häufig ein Anzeichen für die Verletzung des OCP (und meist auch des SRP). Ersetzen Sie diese Strukturen durch polymorphe Aufrufe auf eine Abstraktion – dadurch verschwinden diese if else Blöcke komplett.

6.2.4 Liskov Substitution Principle (LSP)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=fU1AM2LJ1fo>.

Dieses von Barbara Liskov postulierte Prinzip ist wie folgt definiert:

Liskov Substitution Principle

Wenn Objekte vom Typ T durch Objekte von Typ S ersetzt werden (wobei S eine Subklasse von T ist), dann dürfen die (semantischen) Einschränkungen der Basisklasse T von der Subklasse S nicht verletzt werden.

Dies wird im Englischen auch als Strong Behavioral Subtyping bezeichnet.

Die Definition dieses Prinzips ist meines Erachtens nach relativ kompliziert – wenn man sich aber ein Beispiel dazu ansieht, wird es klar, was damit gemeint ist:

Nehmen wir an, wir hätten eine Klasse namens `Rechteck` geschrieben, die wie folgt strukturiert ist:

```

public class Rechteck
{
    private int _höhe = 1;
    private int _breite = 1;

    public virtual int Höhe
    {
        get { return _höhe; }
        set
        {
            if (value < 1) throw new ArgumentOutOfRangeException();
            _höhe = value;
        }
    }

    public virtual int Breite
    {
        get { return _breite; }
        set
        {
            if (value < 1) throw new ArgumentOutOfRangeException();
            _breite = value;
        }
    }

    public virtual int BerechneFläche()
    {
        return Höhe * Breite;
    }
}

```

Abbildung 257: Die Klasse Rechteck

Die in Abbildung 257 zu sehende Klasse ist vor allem eines wichtig: alle Methoden sind **virtual** und können somit in Subklassen überschrieben werden. Nehmen wir an, dass wir genau das machen möchten mit der Klasse **Quadrat**, die wie folgt aussieht:

```

public class Quadrat : Rechteck
{
    public override int Breite
    {
        get { return base.Breite; }
        set { SetzeHöheUndBreite(value); }
    }

    public override int Höhe
    {
        get { return base.Höhe; }
        set { SetzeHöheUndBreite(value); }
    }

    private void SetzeHöheUndBreite(int wert)
    {
        base.Höhe = base.Breite = wert;
    }
}

```

Abbildung 258: Die Klasse Quadrat leitet von Rechteck ab

Zunächst würde man denken, dass die in Abbildung 258 gezeigte Ableitung vollkommen OK ist, aber das ist nicht der Fall: es kann zu subtilen Fehlern kommen, wenn der Nutzer ein **Rechteck** erwartet,

dabei aber über die Ist-Eine-Beziehung eine Instanz von **Quadrat** bekommt. Sehen Sie sich dazu folgenden Unit Test an (was Unit Tests genau sind, klären wir noch in einem späteren Kapitel):

```
[TestMethod]
✖ | 0 references
public void QuadratErfülltInvarianteVonRechteck()
{
    Rechteck rechteck = new Quadrat();
    rechteck.Breite = 5;
    rechteck.Höhe = 4;

    Assert.AreEqual(20, rechteck.BerechneFläche());
}
```

Abbildung 259: Die Klasse **Quadrat** erfüllt nicht alle Invarianten der Basisklasse **Rechteck**

In Abbildung 259 sehen Sie, dass ein **Quadrat**-Objekt erstellt, aber als **Rechteck** interpretiert wird über die Ist-Eine-Beziehung. Im Anschluss wird Breite und Höhe auf die Werte 5 bzw. 4 gesetzt – laut den Eigenschaften eines Rechtecks müsste also die Fläche damit 20 Einheiten betragen. Genau das wird auch im **Assert.AreEqual** Aufruf überprüft – der Test schlägt aber fehl, weil bei der zweiten Zuweisung **rechteck.Höhe = 4;** natürlich auch die Breite des Quadrats geändert wird. **Quadrat** stellt also laut LSP eine Verletzung der Invarianten der Klasse **Rechteck** dar.

Sie können dies vermeiden, indem Sie nicht Klassen implementieren und andere Klassen davon ableiten lassen, sondern tatsächlich Abstraktionen bilden. Abstraktionen haben erfahrungsgemäß weniger strenge Invarianten (Einschränkungen), welche Subklassen erfüllen müssen, sondern eher die Aufgabe, die API zu beschreiben, die bei einem Objekt aufgerufen werden kann.

6.2.5 Interface Segregation Principle (ISP)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=i7zkzfS1TIY>.

Das ISP ist wie folgt definiert:

Interface Segregation Principle

Nutzer von Abstraktionen sollten in ihnen keine Methoden vorfinden, die sie selber nicht aufrufen.

Dieses Prinzip sagt also nichts anderes aus, als dass Interfaces und abstrakte Klassen auf den Nutzer zugeschnitten sein sollen und nicht auf die Klasse, welche das Interface implementiert oder von der abstrakten Klasse ableitet. Daraus folgt für den Programmieralltag:

- Abstraktionen sollten klein und fokussiert sein, damit Sie auf den Nutzer zugeschnitten sind.
- Das sorgt ähnlich wie beim SRP dafür, dass wir eine große Anzahl kleiner Abstraktionen haben anstatt wenige mit vielen Members.

Im Wesentlichen kann man das ISP als das SRP für Abstraktionen auffassen. Durch die geringe Anzahl an Members bei einer Abstraktion kann man z.B. im Programmieralltag viel schneller eine neue

Klasse zu einem bestehenden Interface implementieren und so die bestehende Funktionalität erweitern bzw. austauschen.

6.2.6 Zusammenfassung für die SOLID-Prinzipien

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ok1KA6PRQAg>.

Wenn Sie die SOLID Prinzipien anwenden, sorgt das in Ihrer Entwicklungsumgebung für folgende Änderungen:

- Die Anzahl an Klassen und Abstraktionen steigt.
- Ihre Klassen werden deutlich kürzer (ich selbst habe selten mehr als 100 Zeilen Code pro Klasse)
- Sie verwenden einen Composition Root in Anwendungen, um Abhängigkeiten zwischen Objekten an einer Stelle aufzulösen.
- Sie können durch Einsatz von Polymorphie die Funktionalität eines Programms erweitern oder ändern, ohne dabei bestehende Klassen zu modifizieren.
- Sie können Ihre Klasse wahrscheinlich öfter wiederverwenden.
- Ihr Code wird leichter testbar.

Bitte beachten Sie: in sehr kleinen Projekten kann die Programmierung nach SOLID auch nachteilig sein – da im Normalfall Projekte aber eher wachsen als schrumpfen und Flexibilität im Programmieralltag immer wichtiger wird, kann ich Ihnen nur empfehlen, die SOLID-Prinzipien zu Ihrem festen Handwerkszeug zu machen, wenn Sie objektorientiert programmieren.

6.3 Weitere bekannte Prinzipien

6.3.1 Don't repeat yourself (DRY)

Das DRY-Prinzip haben wir bereits in Abschnitt 5.16.3 bei Value Objects kennengelernt – es sagt aus, dass eine Information in einem System (bei uns der Source Code) an genau einer Stelle unmissverständlich definiert sein soll. Für uns heißt das, dass wir Codeduplizierungen vermeiden sollen und dazu möchte ich Ihnen folgende Tipps an die Hand geben:

- Wenn Sie Codeteile kopieren oder immer wieder sehr ähnlich Codeteile schreiben, dann ist das ein Hinweis darauf, dass Sie gegen das DRY Prinzip verstößen. Können Sie diese Funktionalität in eine neue Klasse / Methode zusammenfassen und diese parametrieren, um die Codeduplizierungen durch entsprechende Aufrufe zu beseitigen (bspw. auch über das Template Method Pattern)?
- Private statische Hilfsmethoden in Klassen sind meistens ein Hinweis auf einen Verstoß gegen das DRY und SRP Prinzip: eine solche Methode ist unabhängig von den Instanzmitgliedern einer Klasse, aber trotzdem nicht aus anderen Scopes heraus ansprechbar aufgrund des **private** Modifizierer. Kann diese Funktionalität nicht in eine eigene Klasse ausgelagert werden?
- Manchmal ist es auch nicht möglich (bspw. durch die Struktur der Programmiersprache oder bei Performanceproblemen), Code nach dem DRY-Prinzip zu schreiben (hochoptimierter Code hat meistens die Tendenz, nicht leicht verständlich zu sein und gegen mehrere Programmierprinzipien zu verstößen). Scheuen Sie in diesem Fall nicht, die entsprechenden Prinzipien zu missachten. Prinzipiell sollten Sie aber darauf abzielen, ihren Code nach dem DRY Prinzip zu strukturieren.

Das DRY Prinzip ist auch unter der Abkürzung SPOT (Single Point of Truth) bekannt.

6.3.2 Favor Composition over Inheritance

Dieses Prinzip sagt aus, dass man die bestehende Funktionalität einer Source Code Basis erweitern sollte, in dem man neue Klassen schreibt, welche bestehende Klassen kapseln (Komposition). Dies sorgt im Normalfall für ein flexibleres Klassenmodell, v.a. wenn man zusätzlich SOLID nutzt und über Dependency Injection je nach Fall die passenden Objekte zusammenbringt. Im Gegensatz dazu steht die Erweiterung eines bestehenden Klassenmodells durch Vererbung – das ist im Regelfall aber deutlich unflexibler und schwieriger zu verstehen (siehe WPF Vererbungshierarchie).

6.3.3 Hollywood Prinzip (Inversion of Control)

Auch das Hollywood Prinzip (engl. Inversion of Control) haben wir bereits kennengelernt, als wir uns mit Delegates und Events in Abschnitt 5.17.3.3 beschäftigt haben. Es sagt aus, dass Methoden nicht in unserem Code direkt von uns, sondern von anderen Komponenten, die üblicherweise Teil eines Frameworks sind, aufgerufen werden. Dazu werden diese Methoden an die entsprechenden Komponenten registriert – in C# passiert das üblicherweise über Events bzw. Delegates. In anderen Sprachen wie C++ oder Java werden dazu Abstraktionen genutzt.

6.3.4 Tell Don't Ask (Level 200)

Dieses Prinzip wird auf Abstraktionen angewendet, wenn Sie nach den SOLID-Prinzipien programmieren: es sagt aus, dass Abstraktionen möglichst keine Eigenschaften bzw. Methoden haben sollten, die als Rückgabewert einen `bool` haben, der zur Abfrage genutzt wird, ob bestimmte Operationen möglich sind. Dies sorgt im Normalfall dafür, dass der Nutzercode, der gegen die Abstraktion programmiert, (größere) `if else` Konstrukte braucht, um je nach Rückgabewert zu bestimmten anderen Aufrufen zu verzweigen. Letztendlich ist das eine Verkomplizierung der Nutzerlogik – vermeiden Sie diese durch ordentlich polymorphe Aufrufe.

Als Negativbeispiel dafür können Sie die Abstraktion `Type` nehmen: in der Methode `GibTypArtAus` in Abbildung 131 muss erst über mehrere Schritte hinweg mit `IsClass` und `IsInterface` abgefragt werden, um welche Art von Typ es sich handelt. Es wäre viel einfacher, wenn es einfach eine Eigenschaft gäbe, die die tatsächliche Typart zurückgibt, sei es als `string`, Enumerationswert oder als Objekt. Vermeiden Sie solche Dinge in Ihrem Code.

6.3.5 Keep it simple and sweet (KISS)

Das KISS-Prinzip (das teilweise auch als “Keep It Simple, Stupid” in der Langform angegeben wird) sagt im Wesentlichen nur aus, dass der Source Code so einfach wie möglich gehalten werden sollte und keine unnötigen Verkomplizierungen enthält (Eigentlich schlimm genug, dass man das als allgemein bekanntes Prinzip formulieren muss, da es anscheinend nicht von jedem Programmierer implizit eingehalten wird).

6.3.6 Zusammenfassung für Designprinzipien

Wir haben uns in den letzten Abschnitten mit verschiedenen Designprinzipien beschäftigt. Ich möchte Ihnen dazu noch folgende Punkte mit auf den Weg geben:

- Designprinzipien sind nicht in Stein gemeißelte Gesetze, sondern Vorgehensvorschläge, die Sie beim Schreiben Ihres Codes einsetzen können.
- Üblicherweise sorgen Sie für Code, der flexibler, leichter verständlich und leichter wart- und testbar ist. Setzen Sie Ihre Erwartungen aber nicht zu hoch an: wenn eine Codebasis 100.000 Zeilen Code hat, dann wird diese nicht wesentlich kürzer, bloß weil man auf etablierte OOD-Prinzipien umstellt. Wahrscheinlich ist die Funktionalität aber sinnvoller über mehrere Typen hinweg verteilt.
- Teilweise können Sie von den Prinzipien abweichen oder Sie für ihre Belange modifizieren – in beiden Fällen sollten Sie allerdings genau wissen, warum Sie dies machen.

- Ich gehe nicht davon aus, dass Sie nach dem Lesen dieser Abschnitte ein OOD-Prinzipienmeister sind – wichtig ist aber, dass Sie die Kenntnis dieser Prinzipien mit auf ihren Programmierweg nehmen. Versuchen Sie nach und nach über die kommenden Jahre, sich diese Prinzipien anzueignen, damit Sie neuen Code automatisch danach gestalten können.

6.4 Objektorientierte Analyse (OOA)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=xF5fEfordAw>.

Die objektorientierte Analyse ist ein Verfahren, das eingesetzt werden kann, um aus Textbeschreibungen ein objektorientiertes Klassenmodell abzuleiten. Häufig ist es im Programmieralltag so, dass man mit Kollegen aus einem bestimmten Fachbereich kooperieren muss, um bestehende (alltägliche) Prozesse in Software abzubilden und automatisieren zu können. Idealerweise kommen diese Anforderungen in einem Anforderungsdokument (bspw. als Last- oder Pflichtenheft), allerdings können diese Textbeschreibungen auch in einfacher Prosa vorliegen, bspw. indem man den Kollegen aus einem Fachbereich interviewt.

6.4.1 Die Schritte der objektorientierten Analyse

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=rAJsfgS5aoU>.

Die objektorientierte Analyse besteht aus fünf Einzelschritten:

1. Alle Substantive erfassen: die Substantive eines Texts sind potentielle Kandidaten für Klassen.
2. Die Daten für jedes Substantiv erfassen: in diesem Schritt beschreibt man, wie die Substantive untereinander gegliedert sind und erhält damit Kandidaten für Felder in Klassen
3. Alle Verben erfassen: die Verben eines Texts sind potenzielle Kandidaten für die Methoden bei Klassen.
4. Adjektive und Adverbiale erfassen: diese beschreibend häufig Randbedingungen, die von Klassenmodell eingehalten werden müssen.
5. Modellverfeinerung: das entstandene Klassenmodell ist in den seltensten Fällen abgeschlossen – es müssen in der Regel weitere Klassen und Methoden hinzugefügt werden.

Damit dieses Vorgehen etwas leichter zu verstehen ist, sehen wir uns in den folgenden Abschnitten dazu ein Beispiel an.

6.4.2 Ein Beispiel zur objektorientierten Analyse

Als Beispiel für die OOA werden wir den folgenden Text zum Spiel Vier Gewinnt nehmen, der aus der deutschen Wikipedia stammt:

Das Spiel wird auf einem senkrecht stehenden hohlen Spielbrett gespielt, in das die Spieler abwechselnd ihre Spielsteine fallen lassen. Das Spielbrett besteht aus sieben Spalten (senkrecht) und sechs Reihen (waagerecht). Jeder Spieler besitzt 21 gleichfarbige Spielsteine. Wenn ein Spieler einen Spielstein in eine Spalte fallen lässt, besetzt dieser den untersten freien Platz der Spalte. Gewinner ist der Spieler, der es als erster schafft, vier oder mehr seiner Spielsteine waagerecht, senkrecht oder diagonal in eine Linie zu bringen. Das Spiel endet unentschieden, wenn das Spielbrett komplett gefüllt ist, ohne dass ein Spieler eine Viererlinie gebildet hat.

In den kommenden Abschnitten werden wir die einzelnen Schritte der OOA durchführen, um diesen Prosatext zu analysieren und aus ihm ein Klassenmodell abzuleiten.

6.4.2.1 Schritt 1: Substantive erfassen

Zunächst müssen die Substantive des Texts erfasst werden, denn diese stellen Kandidaten für mögliche Klassen dar. Im Folgenden sehen Sie den obigen Wikipediatext, in dem allerdings alle Substantive fett markiert sind:

*Das **Spiel** wird auf einem senkrecht stehenden hohlen **Spielbrett** gespielt, in das die **Spieler** abwechselnd ihre **Spielsteine** fallen lassen. Das **Spielbrett** besteht aus sieben **Spalten** (senkrecht) und sechs **Reihen** (waagerecht). Jeder **Spieler** besitzt 21 gleichfarbige **Spielsteine**. Wenn ein **Spieler** einen **Spielstein** in eine **Spalte** fallen lässt, besetzt dieser den untersten freien **Platz** der **Spalte**. **Gewinner** ist der **Spieler**, der es als erster schafft, vier oder mehr seiner **Spielsteine** waagerecht, senkrecht oder diagonal in eine **Linie** zu bringen. Das **Spiel** endet unentschieden, wenn das **Spielbrett** komplett gefüllt ist, ohne dass ein **Spieler** eine **Viererlinie** gebildet hat.*

Wenn man mehrfach vorkommende Substantive entfernt, dann kommt man auf folgende Klassenkandidaten:

- Spiel
- Spielbrett
- Spieler
- Spielstein
- Spalte
- Reihe
- Gewinner
- (Vierer-)Linie

6.4.2.2 Schritt 2: Relationen erfassen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ILmbmCfyQ6w>.

Im jetzigen Schritt wird festgelegt, welche Daten zu einem Substantiv gehören. Dabei werden potenzielle Kandidaten für Felder in Klassen gefunden, wobei es natürlich häufig vorkommt, dass ein Substantiv zum Feld in einem anderen Substantiv wird. Dadurch entsteht eine Hierarchie zwischen den Klassenkandidaten.

Beim obigen Text kann man die Substantive in folgende Struktur bringen:

- Spiel
 - Spielbrett
 - Spieler
 - Gewinner
- Spielbrett
 - Reihen
 - Spalten
 - Plätze
 - Spielsteine
- Spieler
 - Spielsteine

6.4.2.3 Schritt 3: Verben erfassen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=2okEFFOYSxE>.

Im diesem Schritt werden alle Verben des Texts erfasst, denn diese bilden Kandidaten für Methoden in den einzelnen Klassen. Im Folgenden sind alle Verben im Text fett markiert:

Das Spiel wird auf einem senkrecht stehenden hohlen Spielbrett gespielt, in das die Spieler abwechselnd ihre Spielsteine fallen lassen. Das Spielbrett besteht aus sieben Spalten (senkrecht) und sechs Reihen (waagerecht). Jeder Spieler besitzt 21 gleichfarbige Spielsteine. Wenn ein Spieler einen Spielstein in eine Spalte fallen lässt, besetzt dieser den untersten freien Platz der Spalte. Gewinner ist der Spieler, der es als erster schafft, vier oder mehr seiner Spielsteine waagerecht, senkrecht oder diagonal in eine Linie zu bringen. Das Spiel endet unentschieden, wenn das Spielbrett komplett gefüllt ist, ohne dass ein Spieler eine Viererlinie gebildet hat.

Die Verben sollten jeweils in die Standardform gebracht werden. Dann bleiben folgende Verben übrig:

- spielen
- fallen lassen
- bestehen
- besitzen
- besetzen
- seien
- schaffen
- bringen
- enden
- bilden

Bitte beachten Sie, dass Verben wie bestehen und besitzen meistens keine Methodenkandidaten darstellen, sondern stattdessen eher auf Feldreferenzen (Schritt 2) hinweisen. Alle anderen Verben können aber zu den entsprechenden Klassen zugeordnet werden.

6.4.2.4 Schritt 4: Adjektive und Adverbiale erfassen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=2Zc7kwnfV0k>.

Als vorletzter Schritt werden die Adjektive und Adverbiale des Texts erfasst, im Folgenden wie üblich fett markiert:

Das Spiel wird auf einem senkrecht stehenden hohlen Spielbrett gespielt, in das die Spieler abwechselnd ihre Spielsteine fallen lassen. Das Spielbrett besteht aus sieben Spalten (senkrecht) und sechs Reihen (waagerecht). Jeder Spieler besitzt 21 gleichfarbige Spielsteine. Wenn ein Spieler einen Spielstein in eine Spalte fallen lässt, besetzt dieser den untersten freien Platz der Spalte. Gewinner ist der Spieler, der es als erster schafft, vier oder mehr seiner Spielsteine waagerecht, senkrecht oder diagonal in eine Linie zu bringen. Das Spiel endet unentschieden, wenn das Spielbrett komplett gefüllt ist, ohne dass ein Spieler eine Viererlinie gebildet hat.

Daraus lassen sich folgende Randbedingungen ableiten, die man auf die jeweiligen Klassenkandidaten beziehen sollte:

- Spieler sind **abwechselnd** an der Reihe
- Ein Spielbrett besteht aus sieben Spalten und sechs Reihen
- Wird ein Spielstein in eine Spalte gesetzt, landet er im **untersten freien** Platz
- Gewinner wird der erste mit vier Spielsteinen **waagerecht, senkrecht oder diagonal**
- Das Spiel endet unentschieden, wenn das Spielbrett **komplett gefüllt** ist ohne Viererreihe

6.4.2.5 Schritt 5: Klassenmodell aufbauen und verfeinern

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=idUHDjx8is>.

Aus den Informationen, die man aus den vorherigen Schritten gewonnen hat, kann man nun ein Klassenmodell aus den jeweiligen Kandidaten bestimmen. Aus dem obigen Beispiel kann man bspw. folgendes ableiten:

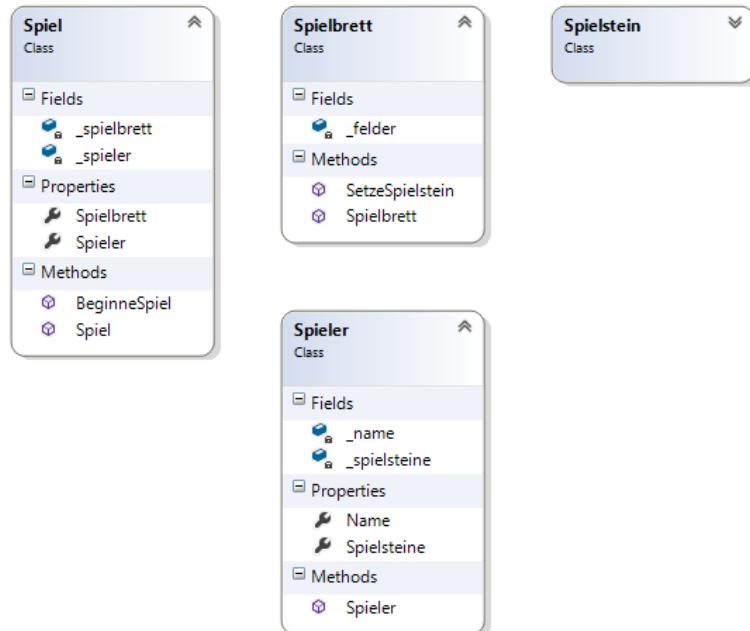


Abbildung 260: Aus OOA abgeleitetes Klassenmodell

Bitte beachten Sie, dass das aus der OOA gebildete Modell in nahezu allen Fällen unvollständig ist. Im Normalfall muss man dieses Modell mit weiteren Klassen und Methoden erweitern, um tatsächlich die Software zu erhalten, die alle (intuitiven) Anforderungen der Kunden / Nutzer abdeckt.

Idealerweise können Sie dann iterativ vorgehen: wenn Sie nach einem OOA-Durchlauf merken, dass Ihr Klassenmodell noch unvollständig ist, dann rufen Sie die entsprechenden Gesprächspartner wieder an den Tisch und besprechen mit ihnen die Details, die Ihrer Meinung noch fehlen. Bei dieser Gelegenheit können Sie sich auch ihr bestehendes Modell absegnen lassen – holen Sie sich regelmäßig Feedback von Ihren Kunden.

6.4.3 Zusammenfassung für die objektorientierte Analyse

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=cQo99PEFVtM>.

Mit der OOA ist es möglich, aus einem Textdokument, sei es in Prosa oder bereits in Anforderungsform, ein Klassenmodell zu entwickeln, das man üblicherweise als Einstieg in die Entwicklung nimmt und danach schrittweise verfeinert. Verstehen Sie dabei die OOA als ein Teil eines kompletten Entwicklungsprozesses – über weitere Teile werden Sie in der Vorlesung Software Engineering erfahren.

Im Embedded-, Automotive- bzw. Low-Level-Umfeld geht man heutzutage beim Entwicklungsprozess häufig nach dem sog. V-Modell vor (inklusive der Erstellung von Lasten- und Pflichtenheften), im Business-To-Business (B2B) oder Business-To-Consumer (B2C) Bereich setzt man eher auf agile Entwicklungsmethoden, wobei vorrangig Scrum im aktuellen Geschehen eingesetzt wird. In beiden Entwicklungsprozessen kann die OOA als Teilschritt eingesetzt werden.

6.5 Objektorientierte Design Patterns

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=nAVbge6NjN8>.

Nachdem wir uns bereits mit objektorientierten Designprinzipien und der objektorientierten Analyse beschäftigt haben, werden wir uns in den kommenden Abschnitten objektorientierte Design Patterns anschauen. Diese stellen abstrakte und wiederverwendbare Muster zu allgemein bekannten OOD-Problemen dar, die für deren Lösung eingesetzt werden können. Betrachten Sie dabei ein Design Pattern aber nicht als definitive Lösung, sondern als Vorlage, die Sie ggfs. selbst modifizieren können.

Design Patterns können sich auf mehrere Ebenen des Applikations- und Systemdesigns beziehen und beschreiben im Normalfall das Verhältnis zwischen mehreren Klassen und Abstraktionen, um ein gewisses Problem zu lösen. Das nützt Ihnen als Softwareentwickler in der Hinsicht, dass sie bereits gelöste Probleme nicht erneut lösen müssen – im Idealfall können Sie ein Design Pattern anwenden und für ihre Belange anpassen.

6.5.1 Historie zu Design Patterns (Level 200)

Ursprünglich kommt der Begriff Design Pattern aus der Architektur: Christopher Alexander hat in seinen Veröffentlichungen von wiederkehrenden Mustern im Stadt- und Gebäudebild gesprochen und diese unterschiedlichen Muster spezifiziert. Diesen Gedanken haben Kent Beck und Ward Cunningham für die Softwareentwicklung aufgegriffen und 1987 erstmals der Öffentlichkeit präsentiert. Sie zeigten, dass auch bestimmte Klassen und andere Typen in einem gewissen Verhältnis zueinander stehen können und damit jeweils ein OOD Pattern formen.

1994 kam dann das erste umfangreiche Kompendium „Design Patterns: Elements of Reusable Object-Oriented Software“, das von der sog. Gang of Four (Erich Gamma, Richard Helen, Ralph Johnson und John Vlissides) in Buchform publiziert wurde. Obwohl dieses Buch meines Erachtens relativ schwer zu lesen und nicht für Anfänger geeignet ist, hat es doch einen gewissen Kultstatus errungen.

Seitdem sind Design Patterns allgemein anerkannt (auch wenn meines Erachtens nach deutlich mehr als 50% der Softwareentwickler die grundlegenden Patterns nicht effektiv einsetzen können). Die Anzahl der Patterns bleibt dabei nicht stehen, da im Normalfall zu jeder neuen Mechanismus, der in Programmiersprachen eingebaut wird, auch dazugehörige Patterns entstehen und verallgemeinert werden.

6.5.2 Organisation von Design Patterns

Wie wir bereits festgestellt haben, spricht ein Design Pattern im Normalfall genau ein Problem an – es kann aber durchaus auch sein, dass es mehrere Patterns gibt, die zu einer Kategorie an Problemen passt. Im Folgenden zeige ich Ihnen bspw. die Namen aller mir bekannten Design Patterns, die mit der Erstellung von Objekten zu tun haben, sog. Creational Patterns:

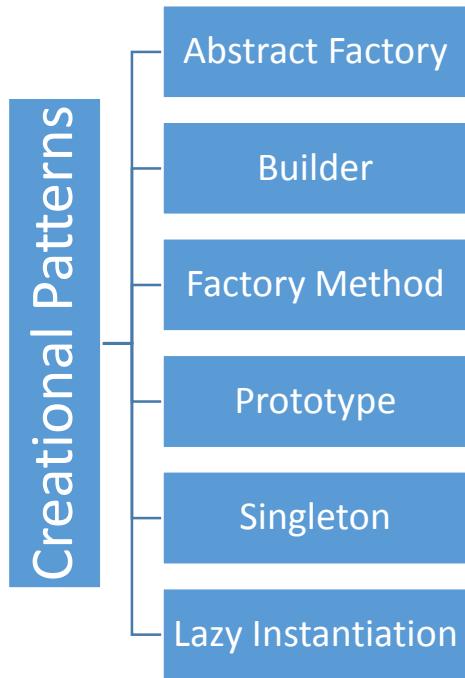


Abbildung 261: Bekannte Creational Design Patterns

Wie in Abbildung 261 zu sehen ist, können mehrere Patterns in eine Kategorie zusammengefasst werden. Von diesen Kategorien gibt es sehr viele:

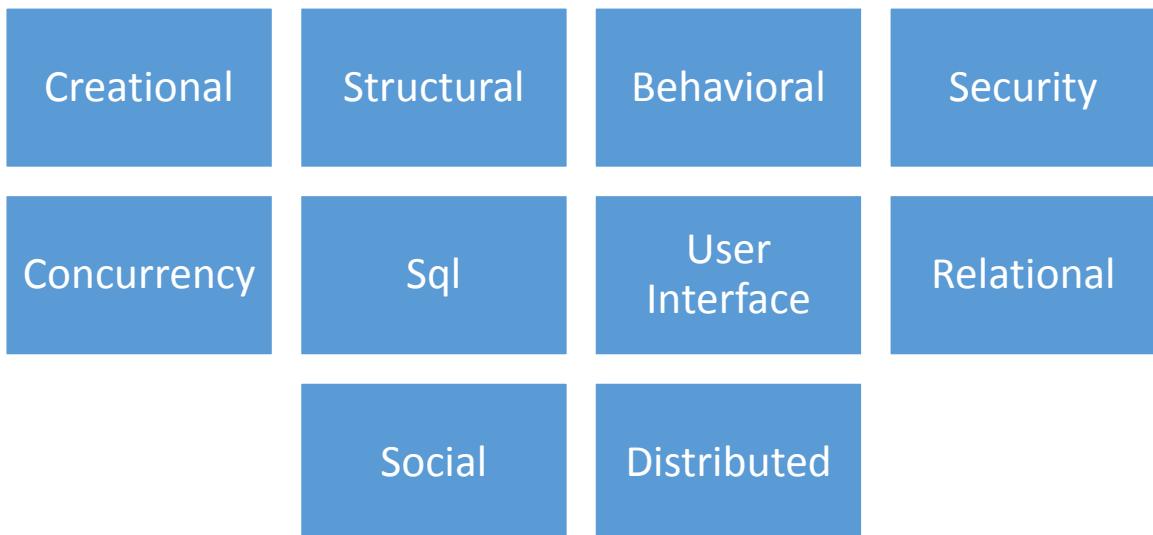


Abbildung 262: Verschiedene Kategorien für Design Patterns

Bitte beachten Sie, dass diese Liste an Kategorien nicht vollständig ist – genauso wie im Laufe der Zeit sich neue Design Patterns bilden können, genauso kann natürlich auch die Liste dieser Kategorien wachsen. Es ist ratsam, sich ab und an mit Design Patterns und deren neuesten Entwicklungen zu beschäftigen.

6.5.3 Warum Design Patterns?

Design Patterns haben folgende Vorteile:

- Sie verhindern, dass man als Softwareentwickler das „Rad ständig neu erfindet“. Wenn Sie während der Programmierung auf ein allgemein bekanntes Problem stoßen, welches durch ein Design Pattern gelöst ist, dann wenden Sie es doch einfach auf ihren Spezialfall an.
- Durch die allgemeine Bekanntheit von Design Patterns können Entwickler sie im täglichen Sprachgebrauch einsetzen, um miteinander effektiver zu kommunizieren. U.a. kann man das am Beispiel Zimmermann erklären: auch dort gibt es wiederkehrende Muster, bspw. eine sog. Zapfenverbindung zwischen zwei Holzteilen. Diese Zapfenverbindung kann für das jeweilige Stück unterschiedlich aussehen, aber das Muster dahinter ist jeweils dasselbe, und jeder Zimmermann kann via dem Musternamen darüber leicht mit einem anderen Zimmermann reden.
- Dadurch erhöht sich die Produktivität, da kognitive Last entfällt und Kommunikation zwischen Teammitgliedern effektiver wird.
- Grundsätzlich sind Software Design Patterns so ausgelegt, dass sie die Anwendung objektorientierter Prinzipien fördern, d.h. sie verbessern im Großen und Ganzen das System- und Applikationsdesign.



*Abbildung 263: Zwei unterschiedliche Zapfenverbindungen (Holztechnik), die aber nach demselben Muster funktionieren
(Analogie für Design Patterns in der Softwareentwicklung)*

6.5.4 Factory

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=6se3AfVVdg>.

Das Factory Design Pattern ist eines der am häufigsten genutzten Designmuster und ist wie folgt definiert:

Factory Design Pattern (Creational Pattern)

Eine Factory erstellt für einen Client einer Objektinstanz, üblicherweise zu einer Abstraktion. Dadurch kann der Client gegen eine Abstraktion programmieren und dennoch innerhalb seines Codes Instanzen zu dieser Abstraktion erzeugen.

Es ist durchaus auch möglich, dass eine Factory konkrete Objekte nicht über deren Abstraktion zurückgibt. In diesem Fall wird sie häufig eingesetzt, um die Erstellung von komplexen Objektgraphen abzukapseln.

Die Factory gibt es in drei Ausführungen, wobei ich persönlich nur die dritte Variante empfehlen kann, da die beiden vorherigen gegen die SOLID Prinzipien verstößen:

1. Factory Method: dies ist einfach eine statische Methode in einer beliebigen Klasse, die als Rückgabetyp den Abstraktionstyp hat und vom Client aufgerufen wird. Da diese Methode

statisch ist, ist natürlich keinerlei Polymorphie möglich und der Client ist hart an die Factory Method gekoppelt, was aber in Ordnung ist, wenn der Client der Composition Root der Applikation ist.

2. Simple Factory: die Factory ist hier eine konkrete Klasse, die der Client kennt. Die Klasse hat eine Methode Create (Erstelle), die als Rückgabetyp den Abstraktionstyp hat (siehe Abbildung 264). Auch hier ist das Problem, das zwischen Client und Factory keine Polymorphie möglich ist. Im Composition Root kann man sie aber genauso wie die Factory Method problemlos einsetzen.
3. Abstract Factory: bei dieser Variante programmiert der Client nicht gegen eine konkrete Factory, sondern gegen eine Abstraktion der Factory. Dadurch ist Polymorphie möglich und es kann verschiedene konkrete Factories geben, die jeweils Instanzen von anderen Implementierungen der Abstraktion zurückgeben (siehe Abbildung 265). Diese Variante sollte man meines Erachtens standardmäßig verwenden, v.a. wenn man Core-Logik schreibt.

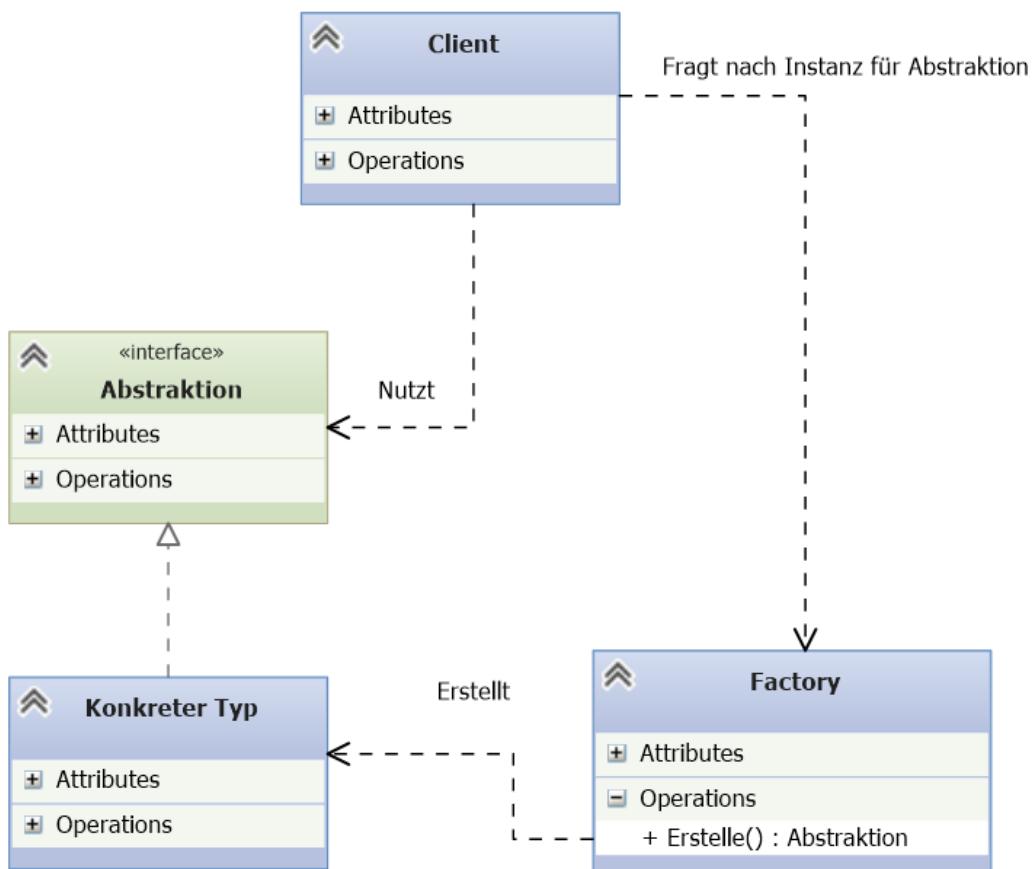


Abbildung 264: UML Diagramm für Simple Factory Pattern

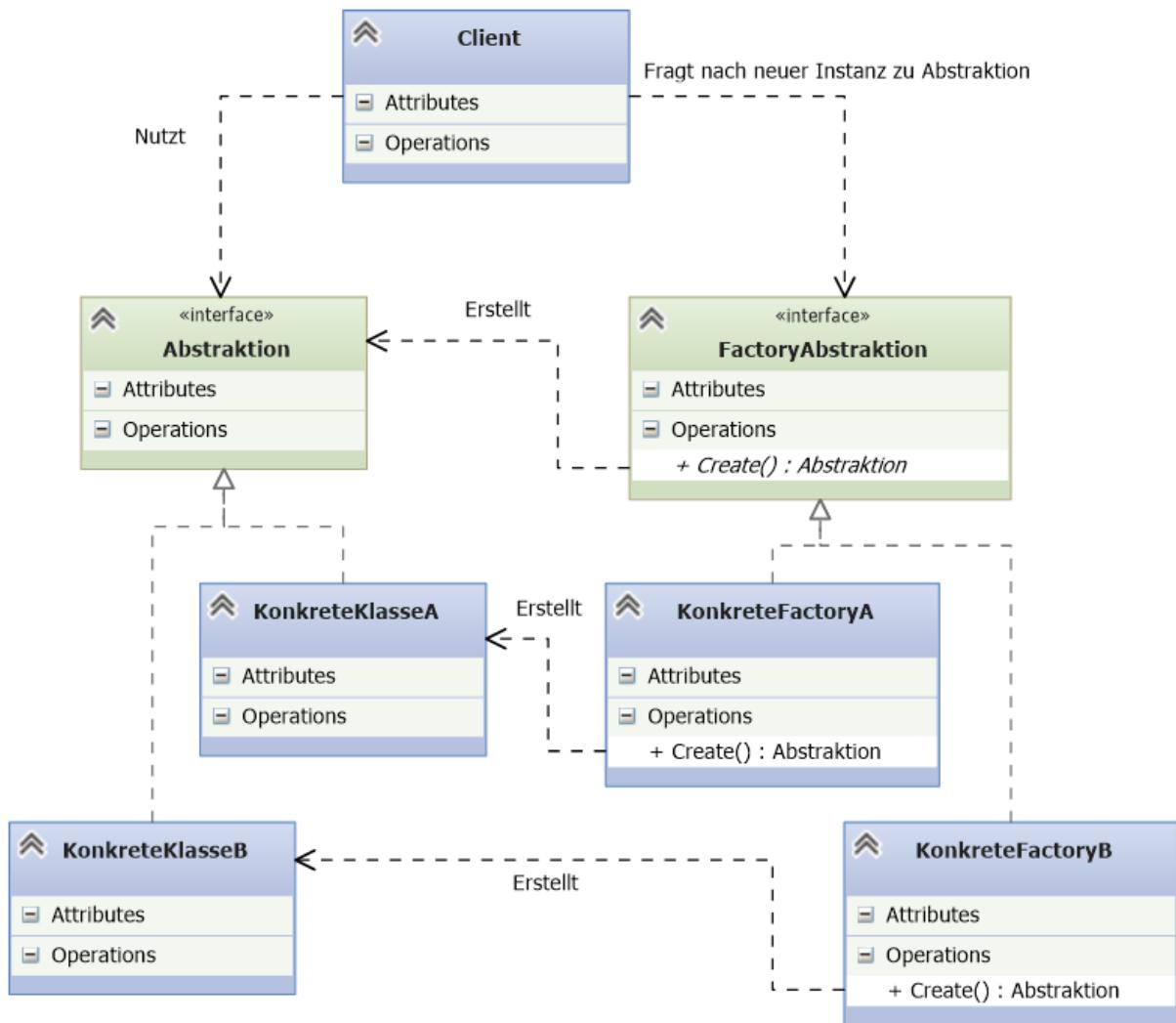


Abbildung 265: UML Diagramm des Abstract Factory Patterns

Bitte beachten Sie auch, dass eine Factory-Klasse (bzw. Factory-Abstraktion) nicht nur eine Methode zum Instanziieren von Objekten bereitstellen kann, sondern auch mehrere. Auch kann eine Factorymethode natürlich Parameter entgegennehmen, die in den Objekterstellvorgang miteinfließen.

6.5.5 Command

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=aS5IC107WKg>.

Das Command Design Pattern ist genauso wie die Factory ein sehr häufig eingesetztes Designmuster und ist wie folgt definiert:

Command Design Pattern (Behavioral Pattern)

Ein Command kapselt eine Funktion / Aktion als Objekt. Dabei besitzt es eine Methode namens Ausführen (engl. Execute), mit dem der Client die Aktion ausführen kann. Der Client ist durch das Command von der Ausführungslogik und ihren Abhängigkeiten komplett entkoppelt.

Am einfachsten kann man sich ein Command als Objekt vorstellen, das eine bestimmte Methode aufruft und die dazu notwendigen Parameter (Abhängigkeiten) bereitstellt. Für den Client hat das

den Nutzen, dass er keine Parameter spezifizieren muss, um eine gewisse Funktionalität aufzurufen (dadurch ist er entkoppelt). Weiterhin ermöglichen Kommandos bspw. auch verzögerte Ausführung, da man sie z.B. in einer Liste festhalten und zu einem gewissen, späteren Zeitpunkt auslösen kann. Man kann sich sogar vorstellen, dass Kommandos persistent gemacht werden können, um sie bei Prozessende zu speichern und bei Neustart wieder zu laden, sodass sie als ausführbare Aktionen dann wieder zur Verfügung stehen.

Das führt zu folgenden Konsequenzen:

- Ein Command sollte alle Informationen zum Ausführen der Aktion in sich vereinen (Clients sollten keine Argumente übergeben müssen bei Ausführung).
- Es sollte einfach sein, neue Kommandos zu bestehenden hinzuzufügen (Open/Closed Principle).

Die Struktur ist dabei, dass der Client gegen eine Kommandoabstraktion programmiert, von der wiederrum beliebig viele konkrete Implementierungen existieren können. Nach diesen wird dann bestimmte Funktionalität beim Aufruf von Execute durchgeführt.

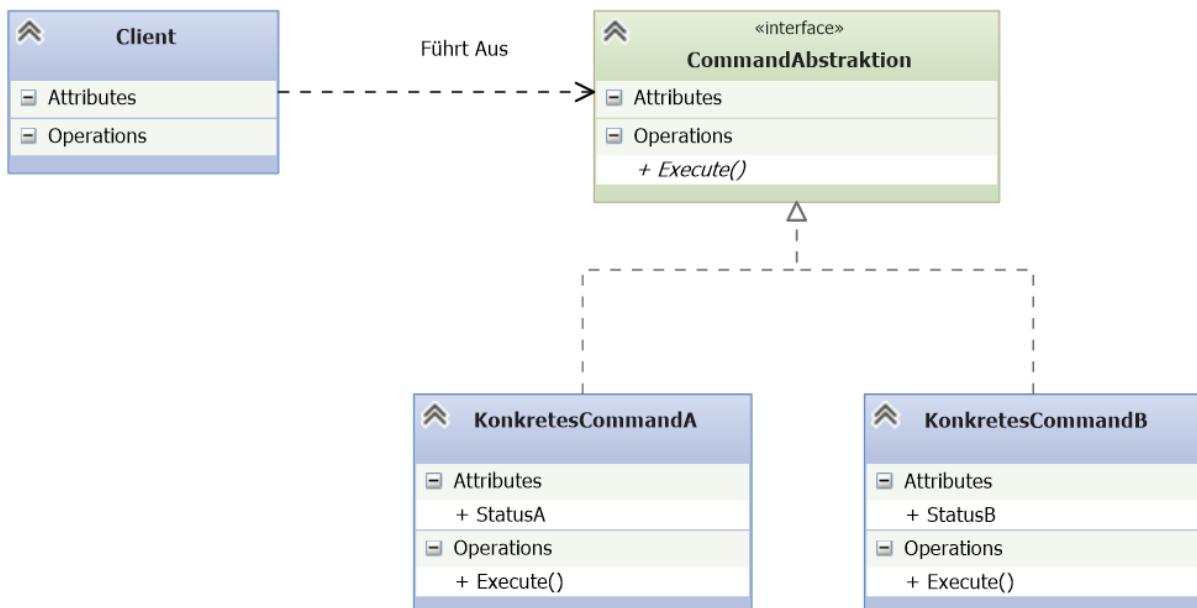


Abbildung 266: UML Diagramm des Command Pattern

Bitte beachten Sie noch folgende Dinge:

- Wenn Kommandos dynamisch zur Laufzeit erstellt werden sollen (nicht im Composition Root), dann wird dies meistens über Factories gemacht.
- Commands müssen nicht nur Execute enthalten, sondern können auch weitere Members aufführen. Häufig wird bspw. auch eine Undo-Funktionalität auf Kommandos implementiert (bspw. indem intern das Memento Pattern verwendet wird).

6.5.6 Decorator

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=f-ma3bUqnFY>.

Das Decorator Pattern ist meines Erachtens eines der wichtigsten Design Patterns, denn damit kann man Klassen, die (wenigstens) ein Interface implementieren, flexibel erweitern:

Decorator Design Pattern (Structural Pattern)

Ein Decorator fügt Funktionalität zu einem bestehenden Objekt hinzu, ohne dass dabei die Implementierung von letzterem geändert werden muss. Der Decorator kapselt hierzu das eigentliche Objekt, implementiert dieselbe API und leitet Aufrufe an das Objekt weiter. Vor und nach diesen Aufrufen kann zusätzliche Funktionalität platziert werden.

Der Decorator wird deshalb häufig genutzt, um sog. Cross Cutting Concerns zu bestehenden Klassen hinzuzufügen. Solche Cross Cutting Concerns sind beispielsweise:

- Logging
- Authentifizierung und Autorisierung
- Caching
- Thread-Synchronisation
- Benachrichtigungen

Wenn ein Decorator eine bestehende Klasse erweitern soll, dann leitet dieser von derselben Abstraktion wie die Klasse ab und erwartet (üblicherweise im Konstruktor) auch eine Instanz dieser Abstraktion. In den einzelnen implementierten Methoden der Abstraktion werden dann die Aufrufe an das gekapselte Objekt weitergegeben – vor und nach diesen Aufrufen kann aber beliebige weitere Funktionalität ausgeführt werden.

Die Struktur sieht dabei wie folgt aus:

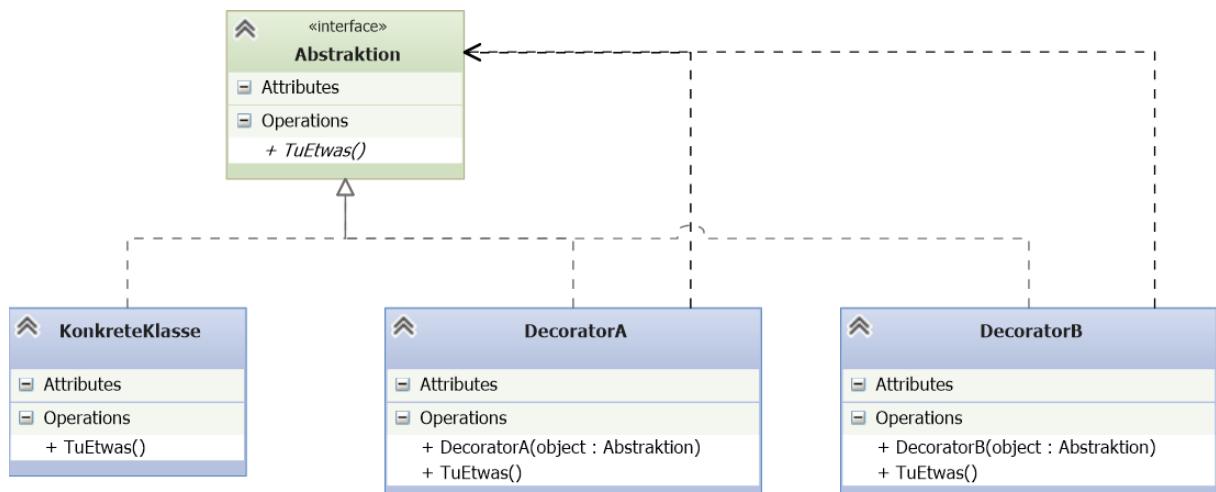


Abbildung 267: UML Diagramm für das Decorator Pattern

Bitte beachten Sie auch, dass in Abbildung 267 nicht nur ein Decorator, sondern zwei implementiert wurden (DecoratorA und DecoratorB). Bei jedem Aufruf von `TuEtwas` können diese jetzt bestimmte Funktionalität ausführen. Da alle Decorators im Konstruktor ein Objekt vom Typ der Abstraktion erwarten, können diese beliebig ineinander verschachtelt werden. Dadurch unterstützt dieses Pattern auch das Open / Closed Principle.

6.5.7 Adapter

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=2H-5vTD3bKk>.

Das Adapter Pattern ist wie folgt definiert:

Adapter Design Pattern (Structural Pattern)

Ein Client soll eine konkrete Klasse über eine Abstraktion ansprechen, wobei diese die Abstraktion nicht implementiert. Der Adapter kapselt diese konkrete Klasse, implementiert die Abstraktion und leitet die Aufrufe an die gekapselte Klasse weiter.

Die Adapterklasse stellt also einen Wrapper für die eigentliche Klasse dar, der sie „fit macht für die Abstraktion“. Dies muss man häufig dann machen, wenn man Bibliotheks- oder Frameworkklassen wiederverwenden möchte, die allerdings noch nicht die Interfaces implementieren, die man in seinem eigenen Projekt vorgegeben hat.

Die Struktur des Adapter Design Patterns sieht dabei wie folgt aus:

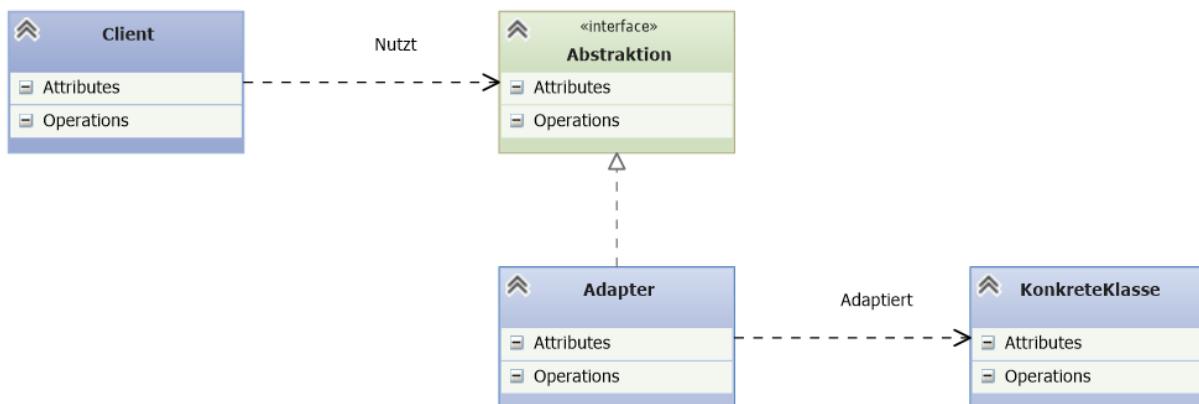


Abbildung 268: UML Diagramm zu Adapter Design Pattern

6.5.8 Strategy

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=JgKfTWtadtU>.

Das Strategy Pattern ist wie folgt definiert:

Strategy Design Pattern (Behavioral Pattern)

Es gibt Situationen, in denen sich Objekte derselben Klasse unterschiedlich verhalten sollen. Diese unterschiedlichen Verhalten werden nicht direkt in die Klassen implementiert und über Verzweigungslogik ausgeführt, sondern in separaten Strategy-Klassen, die über eine gemeinsame Abstraktion angesprochen werden können. Diese Strategies werden über Komposition zur eigentlichen Klasse hinzugefügt.

Durch diesen Mechanismus kann man verschiedenen Verhaltensweisen in die jeweiligen Objekte dynamisch zur Laufzeit injizieren. Im Client selbst werden dann einfach Aufrufe bestimmter Methoden an die Strategy weitergeleitet. Die Struktur sieht dabei wie folgt aus:

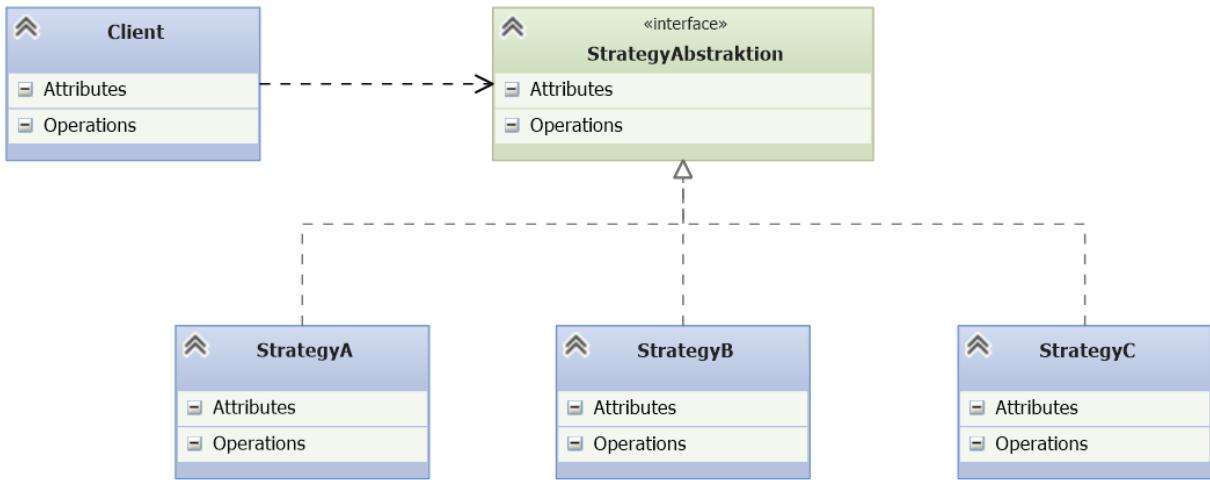


Abbildung 269: UML Diagramm für Strategy Pattern

6.5.9 Memento (Level 200)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=vsBVentNJ8I>.

Dieses Pattern ist wie folgt definiert:

Memento Design Pattern (Behavioral Pattern)

Ein Memento ist ein Objekt, das den Status eines anderen Objekts festhält, ohne dabei gegen die Kapselung zu verstößen. Dieser Status kann zu einem späteren Zeitpunkt über das Memento wiederhergestellt werden.

Das Memento Pattern wird hauptsächlich für die Implementierung von Undo / Redo Funktionalität eingesetzt. Diese sollte nicht in bestehenden Klassen untergebracht werden, da dies mit sehr hoher Wahrscheinlichkeit ein Verstoß gegen das Single Responsibility Principle (SRP) wäre. Ein Memento ist also ein Objekt, dessen einzige Aufgabe es ist, diesen Status für spätere Aktionen festzuhalten.

Üblicherweise werden Mementos in Kombination mit zwei anderen Objekten genutzt:

- Der sog. Originator kann Mementos erstellen bzw. einen Status durch ein Memento wiederherstellen.
- Der sog. Caretaker verwaltet die Mementoobjekte, ohne dabei auf deren Inhalte zurückzugreifen.

Wenn wir uns einen Undo / Redo Stack vorstellen, ist es die Aufgabe des Caretakers, diesen Stack von Mementos zu verwalten: Jedes Mal, wenn eine Änderung durchgeführt wird, wird ein Memento erzeugt und auf den Stack gepusht. Wenn eine Undo-Operation durchgeführt wird, wird das oberste Memento des Stack genutzt und dem Originator übergeben, damit dieser seinen ursprünglichen Status wiederherstellen kann. Sofern ein Redo Stack existiert, kann dieses Memento nun dort hinzugefügt werden. Dieser muss natürlich geleert werden, wenn eine komplett neue Aktion durchgeführt wird.

Mementos sollten als Value Objects implementiert sein (sie halten also ausschließlich den Status, stellen sonst aber keine Funktionalität zur Verfügung). Nur der Originator sollte den Zustand des Mementos setzen oder auslesen können. Die Struktur sieht dabei wie folgt aus:

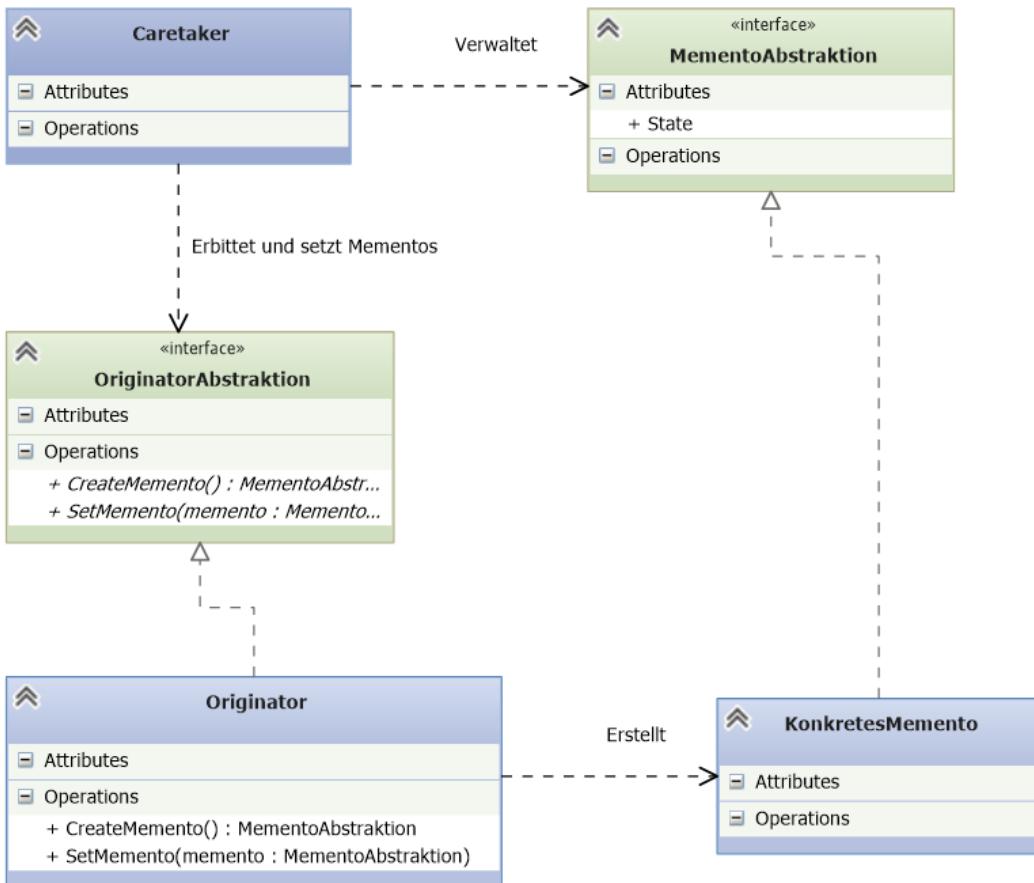


Abbildung 270: UML Diagramm zu Memento Pattern

6.5.10 Repository und Unit of Work (Level 200)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=G738cPs8Ack>.

Ein Repository ist wie folgt definiert:

Repository Design Pattern

Ein Repository kapselt den Zugriff auf ein (persistentes) Medium, um daraus Objekte zu laden und dem Nutzer bereitzustellen. Für den Nutzer sieht die API eines Repositories dabei ähnlich aus wie die einer Collection: man kann Elemente hinzufügen bzw. entfernen sowie nach bestimmten Elementen suchen.

Medien, die von Repositories angesprochen werden, sind dabei üblicherweise Datenbanken, das Dateisystem oder Web Services. Für den Nutzer des Repositories ist es dabei egal, welche konkrete Form dahintersteckt, da er natürlich über eine Abstraktion zugreift. Die Struktur sieht dabei wie folgt aus:

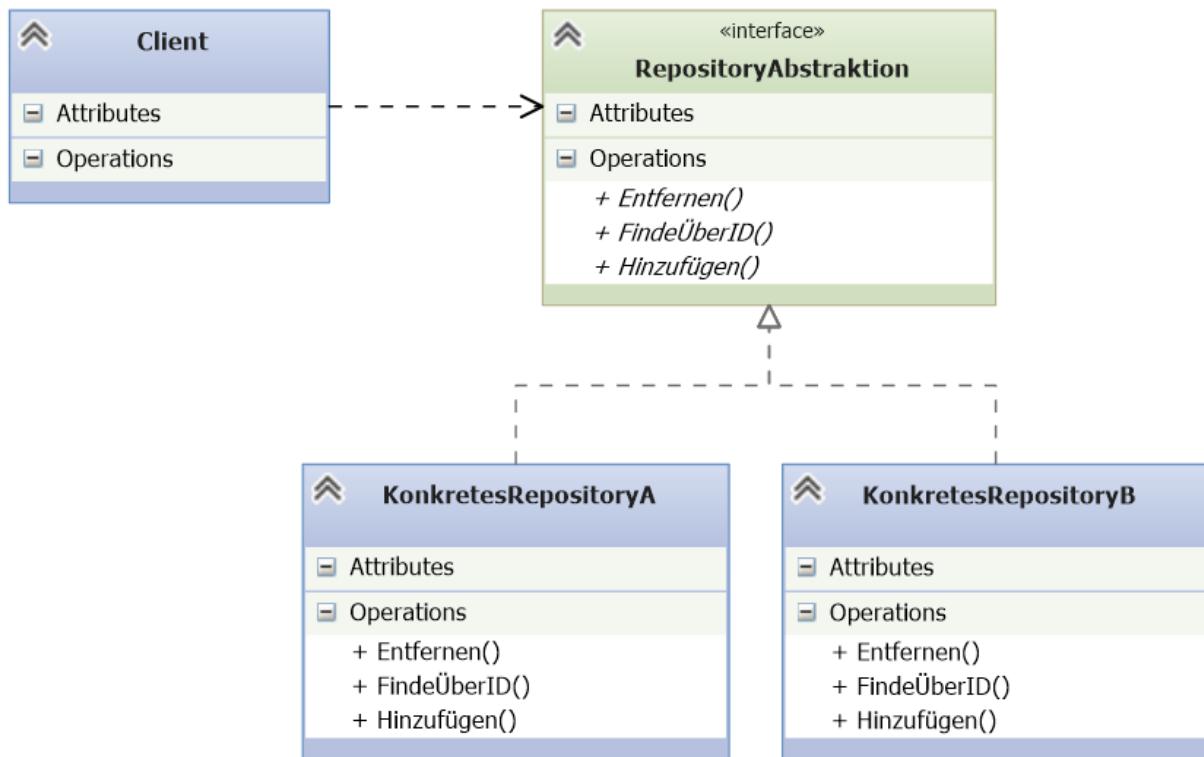


Abbildung 271: UML Diagramm zu Repository Pattern

Zusätzlich zu den in Abbildung 271 gezeigten Methoden kann ein Repository auch eine Methode **Speichern** besitzen – das macht aber nur dann Sinn, wenn man in seiner Applikation ein einziges Repository einsetzt (z.B. wenn man nur einen Typ Objekte speichern möchte). Wenn das nicht der Fall ist, sollte man die Speichern-Methode auf eine sog. Unit of Work auslagern, die mehrere Repositories kapselt und auf der eine Methode **SaveChanges** definiert ist, welche dafür sorgt, dass die Änderungen aller Repositories in das jeweilige persistente Medium übertragen werden – dies beschreibt man dann als eine Transaktion. Die Struktur für eine Unit of Work sieht dann wie folgt aus:

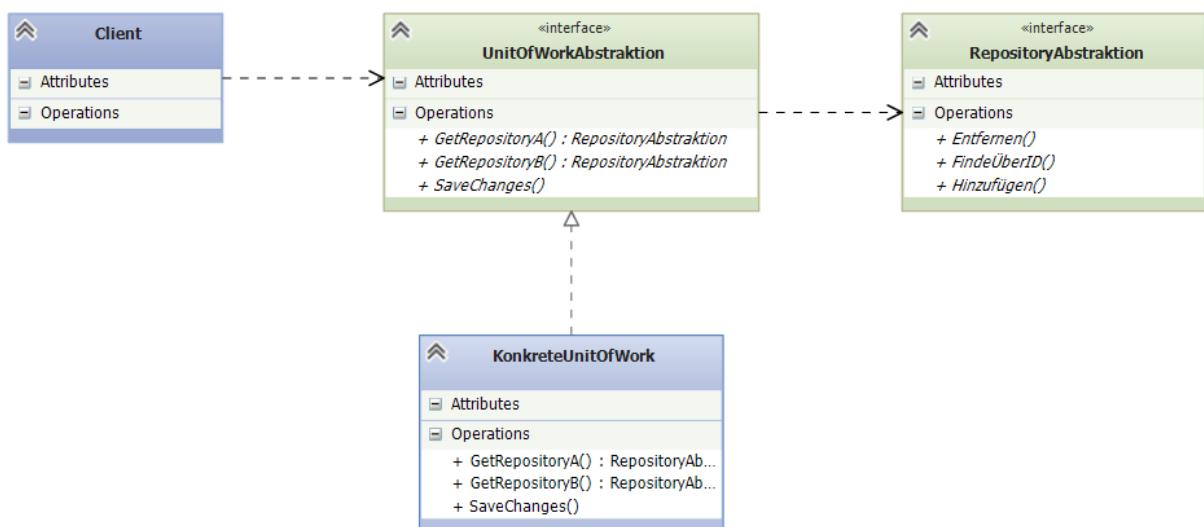


Abbildung 272: UML Diagramm zu Unit Of Work Pattern

Die Klasse **DbContext**, die wir von Entity Framework kennengelernt haben, ist genau nach dem Muster von Abbildung 272 aufgebaut: sie selbst stellt eine Unit of Work dar, die jeweiligen

Eigenschaften vom Typ `DbSet<T>` sind Repositories, welche den Zugriff auf persistierte Objekte bereitstellen.

6.5.11 Observer (Level 200)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=IF-OfbU5g8k>.

Das Observer Design Pattern ist wie folgt definiert:

Observer Design Pattern (Behavioral Pattern)

Ein Observer ist ein Objekt, das sich bei einem anderen Objekt registriert, um dieses zu überwachen. Das andere Objekt benachrichtigt den Observer bei einer Statusänderung, sodass der Observer dadurch eine bestimmte Funktionalität auslösen kann.

Das Observer Pattern ist letztendlich eine Anwendung des Hollywood-Prinzips, denn das observierte Objekt sorgt durch eine Statusänderung dafür, dass der Observer benachrichtigt wird und so weitere Funktionalität auslösen kann.

Dieses Pattern hat in C# eine besondere Bedeutung, denn im Gegensatz zu anderen objektorientierten Programmiersprachen wie C++ oder Java gibt hier Events, die eine Implementierung des Observer Patterns darstellen.

Noch anzumerken ist, dass sich beliebig viele Observer an ein Objekt registrieren können. Die Struktur für dieses Pattern sieht dabei wie folgt aus:

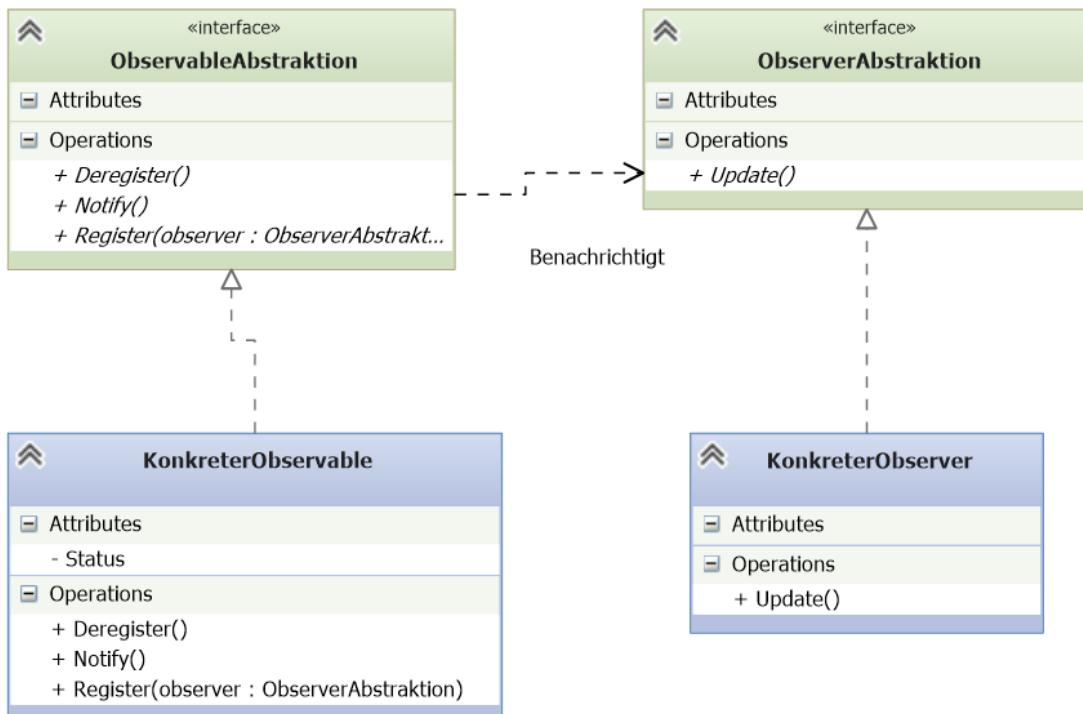


Abbildung 273: UML Diagramm für Observer Pattern

6.5.12 Model-View-View Model (MVVM) (Level 200)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=vIWYYCAwqcE>.

Das Model-View-View Model Pattern ist ein XAML bzw. HTML 5 / JavaScript spezifisches Pattern, da es sich ausgiebig der umfassenden Data Binding Mechanismen bedient, die in diesen Frameworks möglich sind.

Model-View-View Model Design Pattern (User Interface Pattern)

MVVM separiert das Aussehen und die Logik einer Ansicht in zwei verschiedene Klassen, sodass diese unabhängig voneinander entwickelt werden können. Die View (dt. Ansicht) besteht dabei im Idealfall ausschließlich aus XAML Elementen und keinem Code Behind, das View Model ist eine UI-unabhängige Klasse, die das Observer Pattern implementiert und das Model (die Daten, die in einer Ansicht angezeigt werden sollen) für das User Interface aufbereitet. View und View Model werden über Data Binding miteinander verbunden.

Dieses Pattern wurde vor allem deswegen eingeführt, damit die UI Logik mit Unit Tests getestet werden kann. Da das View Model nicht von **UIElement** ableitet und dadurch auch nicht am Zeichenprozess teilnehmen muss, kann man sich in Unit Tests die Mühe sparen, einen UI Thread für jeden Testfall hochzufahren.

Ein weiterer Vorteil dieses Patterns ist der Fakt, dass es den Designer / Developer Workflow ermöglicht. Dabei kann ein UI Designer unabhängig von einem Softwareentwickler die View erstellen, während letzterer sich um das View Model kümmert.

Wichtig ist, dass das View Model in .NET das Interface **INotifyPropertyChanged** implementiert, was letztendlich eine Umsetzung des Observer Patterns ist. Dadurch können sich die UI Elemente, die an die entsprechenden Eigenschaften des View Models via Data Binding gebunden sind, automatisch bei Wertänderung aktualisieren. Die Struktur sieht dabei wie folgt aus:

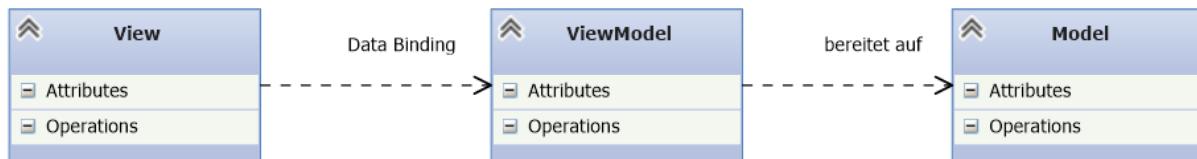


Abbildung 274: UML Diagramm zu MVVM

Optional kann zwischen View und View Model bzw. zwischen View Model und View auch noch eine Abstraktion platziert werden. Prinzipiell ist das aber nicht notwendig, da Data Binding via Reflection funktioniert (also sowieso schon typagnostisch ist) bzw. das View Model das Model hart koppeln darf. Häufig wird auch das Command Pattern im View Model eingesetzt, in denen ausführbare Methoden des View Model gekapselt sind und wie Data Binding üblicherweise mit Buttons verbunden werden.

6.5.13 Zusammenfassung für Design Patterns

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=GcPzedUj7dk>.

Wir haben noch längst nicht alle Design Patterns, die zurzeit erfasst und katalogisiert sind, in diesem Subkapitel angesprochen. Ich hoffe aber, dass Sie einen Eindruck davon bekommen haben, dass es für viele Programmiersituationen, die auch Ihnen mit der Zeit begegnen werden, bereits Vorlagen gibt, die beschreiben, wie Sie das jeweilige Problem angehen können.

Ich gehe nicht davon aus, dass Sie nach dieser Vorlesung bereits perfekt im Umgang mit den hier vorgestellten Patterns sind – allerdings würde ich Ihnen raten, dass Sie sich im Laufe ihrer Karriere als

Softwareentwickler hin und wieder einen Überblick über aktuelle Design Patterns verschaffen und ab und zu ein Ihnen unbekanntes Pattern verinnerlichen. Mit einem Fundus von zwanzig bis dreißig Design Patterns kann man den meisten Situationen im Programmieralltag gut begegnen und bereits sehr effektiv Softwaredesignkonzepte erstellen.

Wenn Sie mehr über Design Patterns erfahren möchten, dann kann ich Ihnen folgende Materialien empfehlen:

- Agile Principles, Patterns and Practices in C# von Robert. C. Martin (Uncle Bob)
Dieses Buch ist mittlerweile auch kostenfrei als PDF-Version erhältlich:
<http://bit.ly/SKmGom>
- Die Website www.oodesign.com hat Beschreibungen zu häufig genutzten Design Patterns
- Das Online-Videotraining-Portal www.pluralsight.com hat einen fünfzehnstündigen Kurs zu häufig eingesetzten Design Patterns unter „Software Practices -> Design Patterns Library“.

6.6 Eine Einführung in automatisiertes Testen (Level 200)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=HoTdJkc7KbQ>.

Vielleicht haben Sie sich auch schon mal über folgende Situation geärgert: Sie haben gerade ein neues Feature fertigprogrammiert und möchten dieses ausprobieren – damit das geht, starten Sie ihre Applikation über Visual Studio im Debug-Modus. Um zur richtigen Stelle zu kommen, müssen Sie erst einige andere Schritte durchführen, um die neue Funktionalität einzusetzen. Während des Tests fällt Ihnen auf, dass etwas nicht stimmt – sie beenden deshalb die Applikation, ändern Ihren Code, und starten erneut, um nochmals zu testen. Dabei führen Sie erneut sämtliche Schritte aus...

Gerade dieses Szenario kann automatisiertes Testen beheben, denn anstatt manuell über das User Interface der Applikation die entsprechende Funktionalität anzusteuern, werden einfach in speziellen Methoden programmatisch alle notwendigen Objekte erstellt und die entsprechende Funktionalität ausgeführt, um einen bestimmten Vorgang zu testen. Und genau das werden wir uns in den kommenden Abschnitten etwas genauer anschauen.

6.6.1 Was genau bedeutet automatisiertes Testen?

Wenn man von automatisierten Tests spricht, dann sollten folgende Sachverhalte erfüllt sein:

- Die Tests können vollautomatisch auf Knopfdruck ausgeführt werden und benötigen keine Interaktion mit einem Nutzer.
- Dadurch bedingt können Tests wiederholt ausgeführt werden, um die Korrektheit der durch die Tests abgedeckten Aspekte zu überprüfen – dies nennt man Regression Testing im Fachausdruck. Durch Regression Testing kann bei Änderungen am Source Code sichergestellt werden, dass alle getesteten Funktionalitäten korrekt implementiert sind.

Durch automatisiertes Testen haben wir also nicht nur die Möglichkeit, manuelle Tests auf ein minimales Maß zu reduzieren, sondern wir können diese Tests auch immer wieder regelmäßig ausführen und so sicherstellen, dass unser Source Code zu jedem Zeitpunkt richtig funktioniert. Vor allem kann man in automatisierten Tests aber auch Bedingungen leichter herstellen, die man über das normale User Interface nur schlecht erreichen kann – bspw. wenn Zufall oder Zeit mit involviert sind.

Automatisierte Tests werden dabei grob in zwei Kategorien eingeteilt:

- Unit Tests testen die Funktionalität / das Verhalten einer einzigen Klasse. Alle Abhängigkeiten, die eine solche Klasse hat, werden durch sog. Test Doubles ersetzt. Test Doubles sind spezielle Objekte, die nur für den Testfall vorgesehen sind.
- Integrationstests testen im Gegensatz zu Unit Tests mehrere Objekte im Verbund. Dabei können zwar auch Test Doubles eingesetzt werden, grundsätzlich möchte man in diesen Tests aber feststellen, ob die entsprechenden Objekte korrekt miteinander interagieren.

Bitte beachten Sie, dass Sie in Unit Tests niemals externe System wie bspw. eine Datenbank oder einen Web Service miteinbeziehen sollten, denn Unit Tests überprüfen eine einzelne Klasse in Isolation. Das macht sie in Ihrer Ausführung auch besonders schnell – ein Unit Test sollte im Normalfall eine Durchlaufzeit von nur wenigen Millisekunden haben, damit der Entwickler auch große Test Suits mit hunderten oder gar tausenden Tests immer wieder ohne gefühlten Zeitverlust ausführen kann.

Bei Integrationstests ist es hingegen erlaubt, externe Systeme miteinzubinden (was meistens die Durchlaufzeit in die Höhe treibt).

6.6.2 Automatisierte Tests in Visual Studio

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=6ZS0SytcoWA>.

Visual Studio bringt direkt Unterstützung für automatisiertes Testen mit. Tests werden über den sog. Test Explorer ausgeführt, bei dem auch gleich ersichtlich ist, welche Tests fehlgeschlagen sind und welche nicht. Den Test Explorer können Sie in der Menüleiste unter Test -> Windows -> Test Explorer einblenden, sofern er bei Ihnen noch nicht sichtbar ist.

Um automatisierte Tests zu schreiben und auszuführen nutzt man üblicherweise Frameworks. Die populärsten in .NET sind:

- NUnit
- xunit.net
- MSTest (direkt mit Visual Studio mitgeliefert)

Die ersten beiden kann man sich über den Paketmanager NuGet besorgen, letzteres kommt direkt mit Visual Studio. Genau deswegen werden wir es auch für unsere Zwecke einsetzen.

Bitte mischen Sie Testcode und Produktionscode nicht. Erstellen Sie je ein eigenes Projekt für Unit Tests sowie Integrationstests und referenzieren Sie die entsprechenden Projekte, welche Produktivcode, den Sie testen möchten, enthalten.

Der Grund hierfür ist, dass Tests üblicherweise nicht mit an den Kunden ausgeliefert werden sollen. Wie man ein Testprojekt erstellt, zeige ich Ihnen hier:

1. Machen Sie einen Rechtsklick auf die Solution und wählen Sie Add -> New Project aus.
2. Gehen Sie nun im linken Masterbereich auf Visual C# -> Test und wählen Sie den Eintrag Unit Test Project

In Abbildung 275 sehen Sie auch nochmals, wie der Dialog zur Erstellung des Projekts aussehen sollte. Vergessen Sie nicht, die entsprechenden Projekte in Ihrem neuen Projekt zu referenzieren.

Innerhalb des neuen Projekts finden Sie genau eine Datei, die auch schon die Struktur einer Testklasse beziehungsweise eines Tests anzeigt. Um als Test erkannt zu werden, muss eine Methode...

- ...mit dem **TestMethodAttribute** gekennzeichnet sein
- ...**void** als Rückgabetyp haben
- ...keine Parameter entgegennehmen
- ...nicht statisch sein
- ...in einer Klasse liegen, die mit dem **TestClassAttribute** gekennzeichnet ist

Alle Methoden, die diese Struktur aufweisen, können dann über den Test Explorer automatisch ausgeführt werden wie in Abbildung 276 gezeigt.

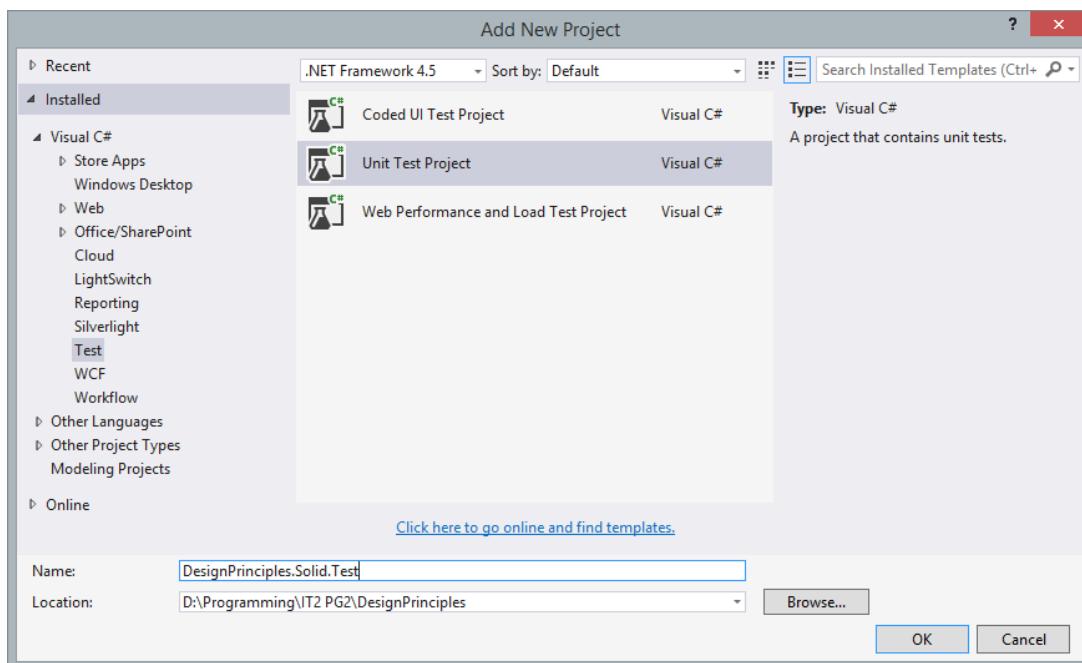


Abbildung 275: Ein Unit Test Projekt über den Projektdialog erstellen

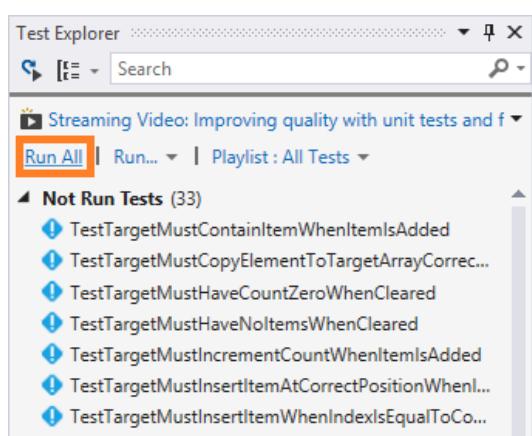


Abbildung 276: Alle Unit Tests in der Solution über den Test Explorer ausführen

Auch bei den anderen Test Frameworks NUnit und xunit.net werden Testmethoden bzw. Testklassen auf ähnlich Art und Weise definiert – wenn Sie diese einsetzen wollen, schauen Sie doch einfach auf der entsprechenden Website vorbei:

- <http://www.nunit.org/>

- <https://github.com/xunit/xunit>

6.6.3 Die Struktur von automatisierten Tests

Nachdem wir im letzten Abschnitt gelernt haben, wie die Infrastruktur zur Ausführung von automatisierten Tests aussieht, werden wir uns in diesem Abschnitt genauer damit beschäftigen, welchen Code wir in unsere Tests schreiben. Prinzipiell kann man Testmethoden beliebig lang machen und beliebigen Code ausführen – das würde ich Ihnen allerdings nicht empfehlen.

Prinzipiell sollten Sie Ihre Testmethoden nach dem Arrange-Act-Assert Prinzip (Abk. AAA) aufbauen:

1. In der Arrange-Phase werden alle Objekte erzeugt und verknüpft, die für den Test notwendig sind.
2. In der Act-Phase wird die Funktionalität ausgeführt, die getestet werden soll.
3. In der Assert-Phase wird überprüft, ob die ausgeführte Funktionalität das richtige Ergebnis geliefert hat (to assert heißt auf Deutsch „annehmen / folgern“).

Dies kann man sich natürlich am besten an einem Beispiel verdeutlichen: im Folgenden zeige ich Ihnen einen Unit Test, den ich für Aufgabe 5 (List<T> nachimplementieren) geschrieben habe. Den kompletten Entstehungsprozess können sie sich auch im Video nochmals anschauen können:

<https://www.youtube.com/watch?v=T3hcyK1xYjQ>

Auch finden Sie auf der GRIPS-Plattform den Source Code zur Lösung von Übung 5, in dem alle Unit Tests enthalten sind.

*Aus dem Namen sollte man herauslesen
können, was genau im Test passiert (nicht vor
langen Namen zurückschrecken)*

```
[TestMethod]
public void TestTargetMustIncrementCountWhenItemIsAdded()
{
    // Arrange
    var testTarget = new List<int>(); ━━━━━━ In der Arrange-Phase werden alle nötigen  
                                            Objekte für den Test erzeugt (hier nur die zu  
                                            testende Liste)
    // Act
    testTarget.Add(42); ━━━━━━ In der Act-Phase wird die zu testende  
                               Funktionalität ausgeführt
    // Assert
    Assert.AreEqual(1, testTarget.Count); ━━━━━━ Mit der statischen Klasse Assert können  
                                              Überprüfungen gemacht werden
}
```

Abbildung 277: Ein Unit Test nach dem AAA-Prinzip

In Abbildung 277 sehen Sie, dass der Code der Testmethode in die eben genannten Phasen aufgeteilt ist:

1. In der Arrange-Phase wird hier nur die zu testende Liste erzeugt, da keine anderen Objekte notwendig sind. Falls unsere zu testende Klasse aber Abhängigkeiten zu anderen Typen hätte (idealerweise Abstraktionen), würden diese hier ebenfalls erzeugt werden. Im Fall von Unit Tests wären das Test Doubles, nicht die tatsächlichen Instanzen von konkreten Klassen, die im Produktionscode verwendet werden.
2. In der Act-Phase wird die Funktionalität, die getestet werden soll, ausgeführt. In diesem Fall ist es das Ausführen der Add Methode.

3. In Phase 3 wird mit der **Assert** Klasse überprüft, ob sich durch das Hinzufügen zur Liste die Count Property geändert hat. In diesem Fall erwarten wir, dass sie eins ist.

Ein Unit Test gilt als fehlgeschlagen, wenn in ihm eine Exception auftritt. Und genau das macht das jeweilige **Assert** Statement: wenn die Bedingung, die es betrachtet, nicht erfüllt ist, wird eine Exception ausgelöst.

Gutes (Unit) Testing von Klassen ist dennoch nicht einfach, denn folgende Dinge sollten beachtet werden:

Achten Sie auf folgende Dinge, wenn Sie Ihre Tests schreiben:

- Nicht mehr als ein Assert Statement in einem Test: der Grund hierfür ist, dass jedes Assert Statement bei Nichterfüllung eine Exception auslöst. Was passiert aber mit möglichen Statements, die nach der Assertion folgen? Diese werden nicht ausgeführt, könnten aber auf weitere Fehler hindeuten.
- Wenn Sie das obige Prinzip einhalten, erhalten Sie eine gute Defect Localization. Wenn Sie ihre Tests gut benennen, jeweils nur ein Assert Statement ausführen und Sie auf einen fehlgeschlagenen Test treffen, dann sollten Sie genau wissen, an welcher Stelle Sie Ihren Source Code ändern müssen, um den Fehler zu beheben.
- Das bringt aber auch einige Nachteile mit sich: Sie müssen in Unit Tests eventuell mehrmals dieselbe Situation erstellen, aber auf unterschiedliche Sachen testen. Nutzen Sie in diesem Fall Design Patterns wie das Builder Pattern, das besonders auf diesen Fall zugeschnitten ist.
- Achten Sie darauf, dass Ihre Tests unabhängig voneinander sind. Z.B. sollte es vollkommen egal sein, in welcher Reihenfolge Ihre Tests ablaufen.

Aus diesen Hinweisen folgen mehrere Konsequenzen:

- Sie werden viele Tests schreiben, die aber in sich atomar sind (d.h. sie behandeln genau einen Aspekt der Funktionalität der Klasse, die gerade getestet wird – beachten Sie dabei das Single Responsibility Principle).
- Dadurch wird die Anzahl Ihrer Tests sehr groß – es ist nicht ungewöhnlich, ca. fünf bis zwanzig Tests pro Klasse zu haben.

Der große Vorteil ist: wenn Sie eine gute Testabdeckung haben und Ihre Tests (relativ) strikt nach dem AAA-Muster entwickeln, dann werden Sie (in der Regel) eine so gute Testabdeckung haben, dass Ihre Zeit, in der Sie Ihren Code debuggen, gegen null gehen sollte. Genauso sollte Ihre Fehlerrate gegen null gehen – auch wenn sie sehr wahrscheinlich nicht ganz null sein wird.

Der große Nachteil von automatisierten Testen ist, dass es in etwa 50% Ihrer Entwicklungszeit für Code in Anspruch nimmt – d.h. dass Sie ca. doppelt so lange brauchen, um denselben Code mit den entsprechenden automatisierten Tests zu entwickeln (dabei beziehe ich mich auf meine eigenen Erfahrungen). Der Vorteil ist, dass ihre Debugging-Zeit gegen null geht – und das kann einen sehr großen Einfluss auf Ihren Entwicklungszyklus haben, wesentlich größer als die 50% mehr für die eben besagte Entwicklungszeit.

Letzen Endes sollten Sie meines Erachtens diese Punkte der Entwicklung ohne Tests vorziehen, außer Sie sind wirklich in argen Zeitnöten, sodass die Erstellung von Tests ihr Vorhaben gefährdet.

6.6.4 Was sollte ich testen?

Im Prinzip können Sie jede Klasse ihrer Solution auf Korrektheit überprüfen – dazu möchte ich Ihnen aber noch folgende Tipps an die Hand geben:

- Datenklassen, die ausschließlich aus Feldern und Eigenschaften bestehen, brauchen Sie nicht zu testen. Beim Testen geht es um die Sicherstellung von Funktionalität.
- Testen Sie nicht Implementierungsdetails, sondern das Verhalten ihrer Klasse. Das sorgt dafür, dass Ihre Tests entkoppelter von der Klasse sind, die Sie testen. Unit Tests sollen bloß festlegen, dass eine Klasse nach einem vorgegebenen Muster funktioniert – wie die Klasse das umsetzt, ist jedoch egal.
- Dazu gehört auch, dass für Tests ausschließlich die **public** API einer Klasse genutzt werden soll. Rufen Sie niemals **private** Methoden in Ihren Tests auf.

6.6.5 Testgetriebene Entwicklung

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=3aDwEZ1mN6Q>.

Mittlerweile werden automatisierte Tests nicht nur genutzt, um bestehende Funktionalität zu testen, sondern auch, um noch gar nicht existierenden Code zu beschreiben. Das Paradigma dahinter ist Test-Driven-Development (TDD):

- Bei TDD wird für neue Funktionalität (in einer Klasse) zunächst ein Test geschrieben, der fehlschlägt – Grund dafür ist natürlich, dass die entsprechende Klasse diese Funktionalität noch gar nicht implementiert (Red).
- Als zweiter Schritt wird die entsprechende Funktionalität in der Klasse nachgezogen, damit der entsprechende Test ohne Probleme durchgeführt wird (Green).
- Als letzter Schritt kann der Code in seiner jetzigen Form refactored werden. Da wir Tests haben, welche die geforderte Funktionalität abdecken, können wir diese immer wieder ausführen, um sicherzugehen, dass unser Refactoring keine unerwünschten Nebeneffekte hat (Refactoring).

Als Refactoring bezeichnet man das Umgestalten von Code, so dass er eher den anerkannten Design Prinzipien und Design Patterns entspricht, von denen wir bereits einige kennengelernt haben. Der Refactoring-Urgott in der Programmierszene ist Martin Fowler, der zu diesem Thema ein ganzes Buch geschrieben hat:

<http://amzn.to/1jj0rwr>

Der TDD-Zyklus sieht dabei wie folgt aus:

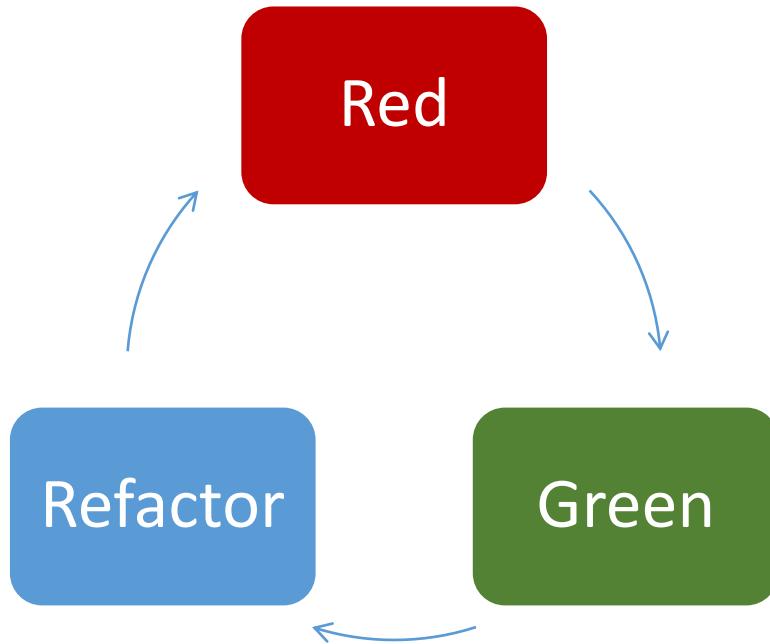


Abbildung 278: Der TDD-Zyklus

Wenn Sie mit einem Test nach dem TDD-Zyklus fertig sind, schreiben Sie einen neuen Test, der das Verhalten Ihrer aktuellen Klasse weiter beschreibt.

6.6.6 Zusammenfassung für automatisiertes Testing

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=y4GxptFzCak>.

Automatisierte Tests sind mittlerweile in der Softwareentwicklung ein fester Bestandteil, allerdings sind sie genauso wie Produktionscode Teile der Software, die gepflegt werden müssen. Ich bin in dieser Vorlesung nicht auf die Einzelheiten zu „guten Tests“ eingegangen, um den Rahmen nicht zu sprengen, aber betrachten Sie bitte ihren Testcode mit der gleichen Sorgfalt wie Ihren Produktionscode.

Natürlich ist automatisiertes Testen ein Bereich der Informatik, der mittlerweile gut erschlossen ist, weswegen ich Ihnen folgende Ressourcen zur weiteren Fortbildung empfehle:

- www.plurasight.com hat mehrere Kurse, die (Unit) Testing und TDD betreffen, v.a. auch im Umgang mit den verschiedenen Frameworks. Diese würde ich als Einstieg empfehlen.
- Wenn Sie mehr über Design Patterns bei automatisierten Tests erfahren möchten, dann kann ich Ihnen das Buch [xUnit Design Patterns](#) empfehlen – es ist zwar keine leichte Lektüre, aber in ihm finden sie alle Entwicklungsmuster, die Sie für das effektive Schreiben von Tests brauchen können.
- Das Buch [Growing Object-Oriented Software, Guided By Tests](#) ist eine sehr praktische Anleitung zum Thema automatisiertes Testen, v.a. da ein konkretes, umfangreiches Beispiel in über 10 Kapiteln durchgeführt wird. Der Source Code ist zwar in Java geschrieben, lässt sich aber gut auf C# übertragen.

Ich halte es für essentiell wichtig für Ihre Programmierkarriere, dass Sie sich mit diesem Bereich auch nach der Vorlesung aktiv auseinandersetzen.

7 Objektorientierte Programmierung in C++

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=78CQltVRinw>.

In den kommenden Abschnitten werden wir uns zum Abschluss dieses Scripts mit der objektorientierten Sprache C++ beschäftigen. Im Gegensatz zu C# wird C++ üblicherweise als native Programmiersprache eingesetzt, d.h. der Source Code wird direkt in ausführbare Maschinenanweisungen übersetzt und nicht wie C# in eine Intermediate Language, die von einer Laufzeit ausgeführt wird.

Weiterhin haben Sie in C++ einen viel höheren Freiheitsgrad bei der Nutzung des Prozessspeichers – damit sind aber auch einige Fallstricke verbunden, die es zu beachten gilt und die sonst zu sehr subtilen Bugs führen können (Bjarne Stroustrup, der Erfinder und Entwickler von C++, sagt einst folgenden Satz: C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off). Doch dazu später mehr.

7.1 Hello World in C++

Zunächst möchten wir uns ansehen, wie wir in Visual Studio ein C++ Konsolenprojekt erstellen und über dieses Hello World ausgeben können.

Starten Sie dazu wie gewohnt den New Project Dialog und wählen Sie im Anschluss im linken Masterbereich Other Languages -> Visual C++ aus. Im Detailbereich finden Sie dann eine Win32 Console Application (bitte wählen Sie **nicht** CLR Console Application, da dies letztendlich nur eine .NET Applikation ist, die in C++ geschrieben ist, aber genauso in MSIL übersetzt wird wie C#). Vergeben Sie wie sonst auch einen passenden Solution- und Projektnamen.

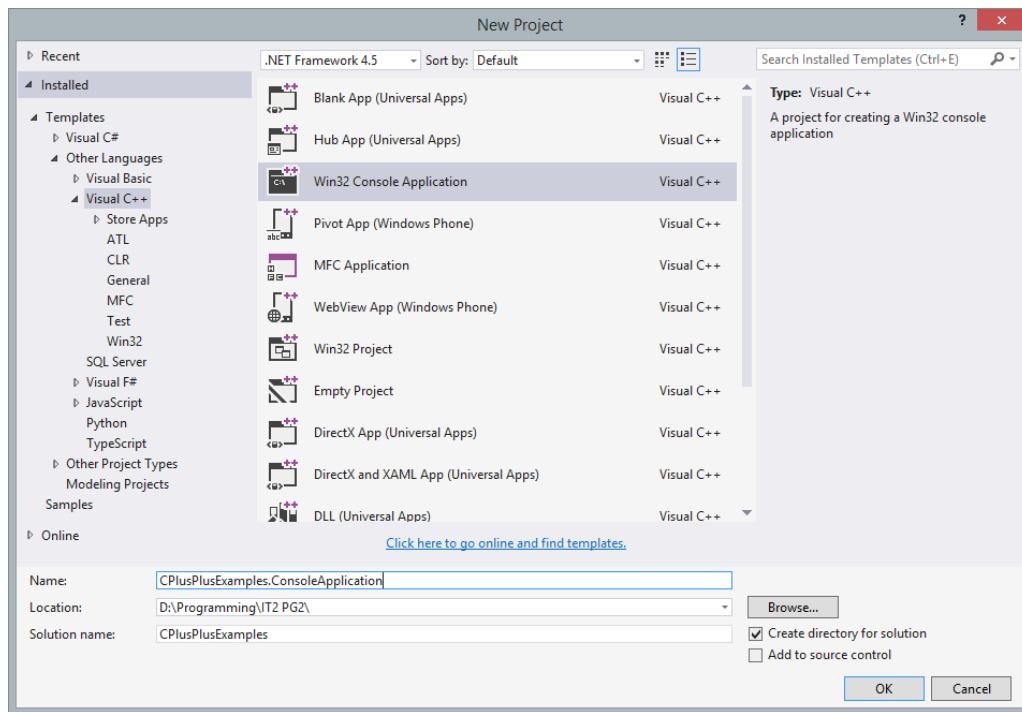


Abbildung 279: Eine natives C++ Konsolenprojekt in Visual Studio erstellen

Anders als bei C# erscheint jetzt ein Wizard, indem Sie definieren können, wie genau Ihr C++ Projekt erstellt werden soll. In der ersten Ansicht wird Ihnen nur eine Übersicht angeboten, die Sie einfach mit einem Klick auf Next überspringen können.

In der folgenden Ansicht, den Application Settings, können Sie einige Details festlegen. Da wir hier mit einem komplett leeren Projekt beginnen wollen, selektieren Sie einfach unter Additional options die Checkbox Empty project.

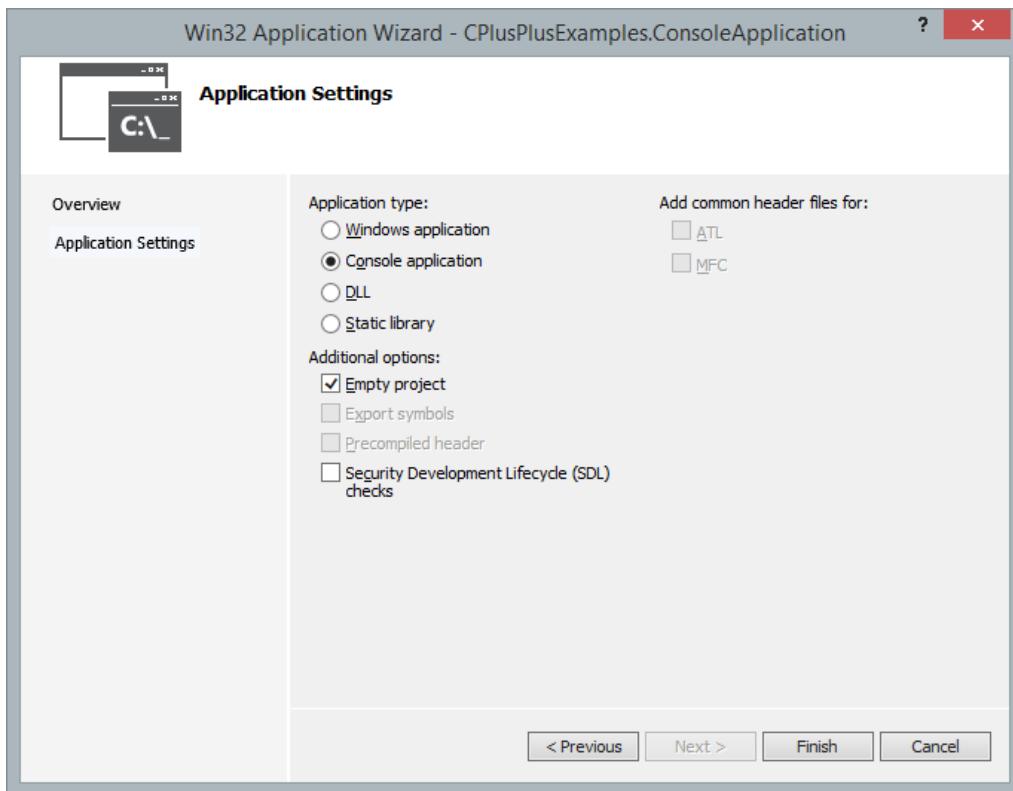


Abbildung 280: Die C++ Applikation als leeres Projekt konfigurieren

Auch wenn die Optionen noch deaktiviert sind, möchte ich kurz auf Sie eingehen:

- Precompiled Headers: Diese Option sorgt dafür, dass (bestimmte) Header-Dateien in eine Zwischenform übersetzt werden, die es dem Compiler ermöglicht, das jeweilige Projekt schneller zu erstellen. Das ist insbesondere dann wichtig, wenn bestimmte Bibliotheken eingebunden werden, die im Wesentlichen nur aus eingebundenen Header-Dateien bestehenden und keine DLL bereitstellen (bspw. die Boost C++ Library). In diesem Fall würde bei jedem Kompiliervorgang auch die komplette Bibliothek übersetzt werden, was durch Precompiled Headers verhindert wird – dadurch braucht der Erstellvorgang deutlich weniger Zeit.
- Export Symbols gibt an, dass die Debugging Symbols neben der *.exe Datei mitabgelegt werden. Dadurch kann man die Applikation leichter debuggen, auch wenn auf dem Zielrechner keine Entwicklungsumgebung zur Verfügung steht. Für uns im Kontext der Vorlesung ist diese Funktionalität nicht relevant.
- Security Development Lifecycle (SDL) checks können auch ignoriert werden – Sie lassen das Programm zur Laufzeit abstürzen, wenn die C Stringverarbeitung über `char*` nicht richtig im Code angewendet wird. Da wir diese aber nicht nutzen, können wir auch diese Option ignorieren.

Klicken Sie im Anschluss auf Finish, um das Projekt zu erstellen.

Sie sehen jetzt ein leeres Projekt, das ähnlich wie ihr C Projekte aus PG1 aufgebaut sein sollte, denn auch in C++ unterscheidet man zwischen Header und Source Files. Zunächst möchten wir eine Datei

erstellen, in die wir unsere `main` Methode setzen können. Dazu machen wir einen Rechtsklick auf Source Files -> Add -> New Item...

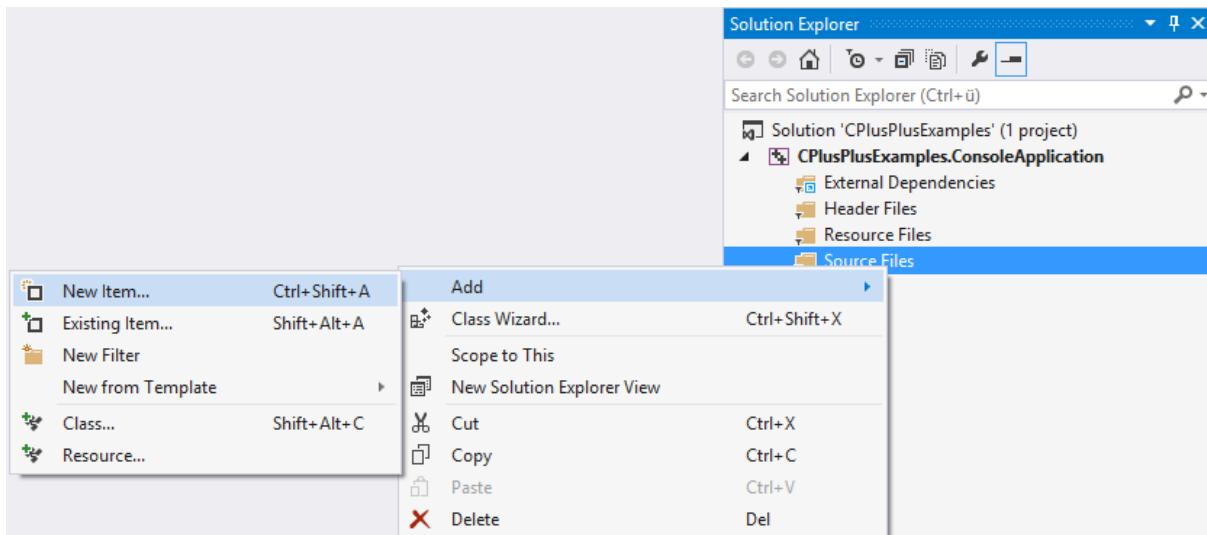


Abbildung 281: Den New Item Dialog öffnen

Im folgenden Dialog können Sie zwischen einer `cpp` (Source File) und einer `h` (Header File) unterscheiden – wir nutzen natürlich dein `cpp` Datei für `main`.

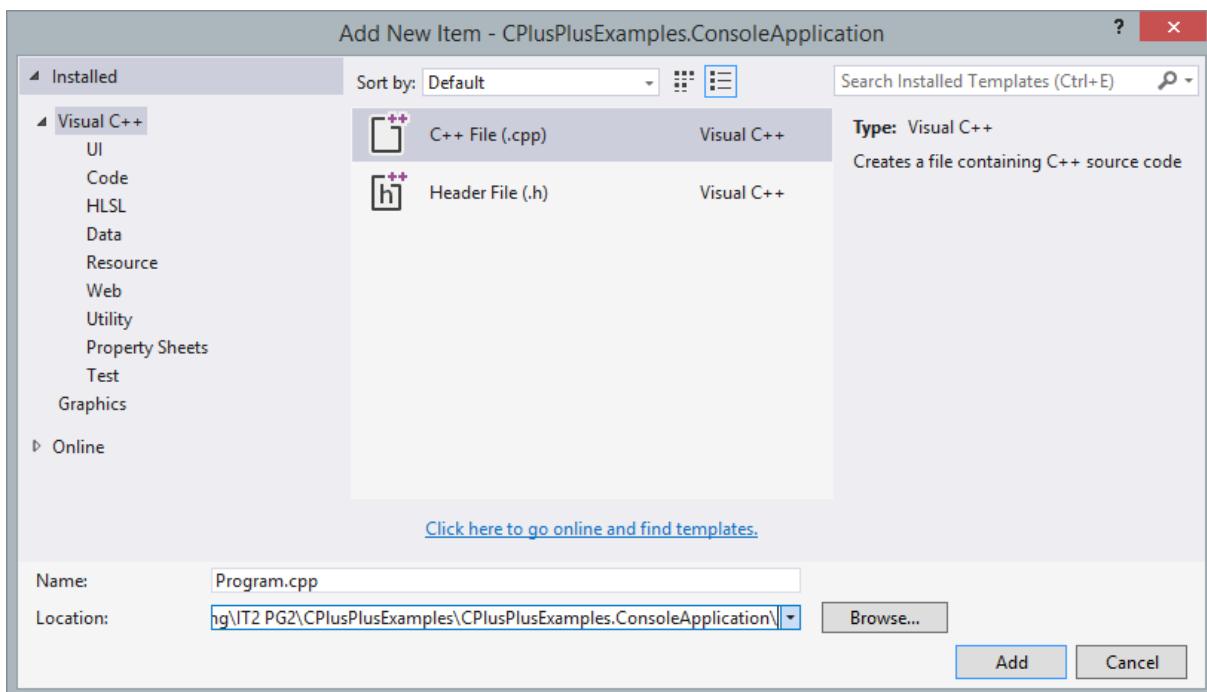


Abbildung 282: Eine neue cpp Datei zum Projekt hinzufügen

Der Unterschied zwischen `.h` und `.cpp` Dateien ist der, dass Header Dateien im Wesentlichen Deklarationen enthalten, Source Files jedoch Definitionen. Anders als in C# muss in C++ eine Methode oder Typ zumindest bekannt (deklariert) sein, damit sie oder er genutzt werden kann (dazu werden die entsprechenden Header eingebunden). Die tatsächliche Definition erfolgt dann in Source Files.

In unserer neuen `cpp` Datei können wir nun folgenden Source Code eingeben:

```

#include <iostream>
#include <limits>

```

Mit der Präprozessor-Direktive #include werden Header-Dateien eingebunden

```

using std::cout;
using std::endl;
using std::cin;
using std::numeric_limits;

```

Über using Statements können Typen und Funktionen im späteren Source Code direkt angesprochen werden

```

void RequireUserToPressEnter();

```

```

void main()
{
    cout << "Hello World!" << endl;
    RequireUserToPressEnter();
}

```

Main Funktion

```

void RequireUserToPressEnter()
{
    cin.ignore(numeric_limits<int>::max(), '\n');
}

```

Methoden müssen zumindest deklariert sein, bevor sie aufgerufen werden können. Die Definition kann später erfolgen

Abbildung 283: Hello World in C++

Wie wir in Abbildung 283 sehen, muss die main Funktion in C++ klein geschrieben werden, ansonsten gelten für sie die gleichen Bestimmungen wie auch in C#. Innerhalb der Funktion wird zunächst auf der Konsole „Hello World!“ ausgegeben. Dabei wird der << Operator genutzt, der formatierten Text auf in den Konsolen-Output-Stream cout schreibt. endl ist ein Objekt, das einen Zeilenumbruch beschreibt.

In der Methode RequireUserToPressEnter wird das cin Objekt genutzt, das die Methode ignore bereitstellt. Diese ignoriert alle Eingaben bis auf Enter, das durch ‘\n’ repräsentiert ist. Die Angabe numeric_limits<int>::max() gibt das Maximum des Typs int zurück, was lediglich aussagt, dass beliebig viele Eingaben des Nutzers ignoriert werden sollen, bis ‘\n’ gefunden wird. Erst nachdem der Nutzer dies gedrückt hat, kehrt diese Funktion zurück.

Bitte beachten Sie auch, dass RequireUserToPressEnter über der main Methode deklariert wurde, damit sie in main aufgerufen werden kann. Dies ist notwendig, da die Definition unterhalb von main liegt und deshalb zu diesem Zeitpunkt noch nicht bekannt ist.

Folgende Dinge möchte ich Ihnen noch mit an die Hand geben, die Sie aus diesem Beispiel mitnehmen sollten:

- In C++ sind anders als in C# freie Funktionen erlaubt. Freie Funktionen sind Funktionen, die nicht innerhalb einer Klasse liegen.
- Genauso wie in C# können Sie mit using bestimmte Typen oder Funktionen aus Namespaces einbinden, sodass wir nicht den Vollqualifizierten Bezeichner angeben müssen. Statt diese einzeln aufzulisten, können Sie auch einfach einen ganzen Namensraum einbinden mit der Anweisung using namespace std;

- In C# haben wir (hauptsächlich) gegen das .NET Framework programmiert, in C++ werden wir die sog. Standard Library ansprechen. Diese ist ein Library, die mit jedem bekannten C++ Compiler ausgeliefert wird. Über `#include` Angaben mit spitzen Klammern können die Header der Standard Library eingebunden werden.

7.2 Die wichtigsten Unterschiede zwischen C# und C++ auf einen Blick

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=Lweca-0D-Dw>.

In der folgenden Liste finden Sie die wichtigsten Unterschiede zwischen C# und C++:

- C++ hat keinen Reflectionmechanismus. Neben den programmatischen Möglichkeiten, die Reflection bietet, haben Sie auch den Nachteil, dass die Auto vervollständigung IntelliSense nur begrenzt funktioniert und die Refactoring-Tools fehlen (bspw. lassen sich Bezeichner nicht einfach ändern oder Methoden extrahieren in Visual Studio).
- In C++ dürfen freie Funktionen definiert werden (Funktionen, die nicht innerhalb eines Klassenscopes liegen). Das hat v.a. darin seine Gründe, dass C komplett in C++ enthalten ist (sämtlicher C Code kann also mit einem C++ Compiler übersetzt werden) und C natürlich keine Klassen kennt.
- Strukturen und Klassen sind in C++ dasselbe. Der einzige Unterschied ist, dass der Standardaccessor bei Strukturen `public` ist, bei Klassen jedoch `private`. Wer die Accessors jedoch immer explizit angibt, findet in beiden Typen keinen Unterschied.
- In C++ gibt es nur die Accessors `public`, `protected` und `private`.
- Aus C++ wird, wie bereits erwähnt, üblicherweise direkter Maschinencode erzeugt, der nicht in einer Runtime (CLR) läuft. Dadurch stehen natürlich auch keine Runtime-spezifischen Elemente wie der Garbage Collector zur Verfügung – wir müssen uns also um das Aufräumen des Heaps selbst kümmern.
- In C++ gibt es keine Interfaces. Abstraktionen können ausschließlich über abstrakte Klassen erstellt werden (diese werden in C++ auch pure virtual classes genannt).
- In C++ ist Mehrfachvererbung bei Klassen möglich.
- Es gibt in C++ keine allgemeine Basisklasse wie `object` für alle anderen Klassen.
- Objekte können in C++ sowohl auf dem Heap als auch auf dem Stack allokiert werden. In C# ist dies nur auf dem Heap möglich.
- C++ unterstützt Pointer und die dazugehörige Zeigerarithmetik aus C. Neben diesen gibt es auch noch sog. Referenzen, die es nur in C++ gibt.
- Es gibt in C++ zwar Funktionszeiger als Äquivalent für Delegates in C#, diese sind aber lange nicht so mächtig wie letztere und können auch nicht einfach auf Memberfunktionen zeigen. Delegates sollten am besten über Abstraktionen in C++ nachgebaut werden.
- Genauso wenig gibt es in C++ Eigenschaften oder Events. In C++ verwendet man explizit Getter und Setter in Klassendeklarationen. Events sollten, wie eben erwähnt, über Abstraktionen nachgebaut werden.
- C++ verwendet eine Aufteilung in Header und Source Files, um Deklaration und Definition zu trennen. In C# ist eine solche Aufteilung nicht notwendig. Auch müssen in C# keine Header importiert werden, um die entsprechenden Typen und Members einzusetzen.
- Statt `var` können Sie in C++ das Schlüsselwort `auto` verwenden, um den Typ einer Variablen vom Compiler bestimmen zu lassen.
- C++ kennt keine Attribute.
- In C++ können Makros eingesetzt werden.

Gerade Zeiger, Referenzen und die Lebenszeit dynamisch erzeugter Objekte sind in C++ wichtige Punkte, mit denen sich Entwickler auseinandersetzen sollten, um subtile Bugs oder Speicherlöcher zu vermeiden.

7.3 Modern C++

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ZkF44IMX3WM>.

Wir werden uns in den kommenden Abschnitten speziell mit „Modern C++“ auseinandersetzen. Im Grunde kann man es mit traditionellem C++ vergleichen, mit folgenden zwei Unterschieden:

1. Manuelles Memory Management ist für uns als Entwickler gefährlich und muss gut durchdacht sein, um Speicherlöcher oder fehlerhafte Zugriffe auf bereits deallokierten Speicher zu vermeiden. In Modern C++ werden diese Probleme durch Smart Pointers, die mit der Standard Library kommen, komplett umgangen – wir müssen uns nicht mehr um die Freigabe dynamisch allokiertener Objekte kümmern.
2. Arrays bieten auch in C++ keine gute API (in der Tat sogar noch eine schlechtere, denn im Gegensatz zu C# wissen sie nicht, wie groß sie sind). Stattdessen werden wir auch hier die Collections einsetzen, die mit der Standard Library kommen.

7.4 Primitive Datentypen und Flow Control in C++

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=Y4X4sKNFObk>.

Im Prinzip können Sie davon ausgehen, dass die primitiven Datentypen wie `int`, `bool`, `double` oder `long` auch in C++ zur Verfügung stehen. Sie müssen lediglich folgende Unterschiede betrachten:

- Numerische Werte können implizit zu `bool` konvertiert werden. Dabei wird aus 0 der Wert `false`, alle anderen Werte werden zu `true` konvertiert.
- Es gibt die numerischen Datentypen `long long` und `long double`, die jeweils 64 bzw. 128 Bit umfassen.
- Bei Buchstaben wird ein Unterschied gemacht zwischen ASCII (`char`) und Unicode (`wchar_t`). Für unsere Zwecke wird `char` reichen, in Produktionscode sollte man sich bei C++ aber überlegen, ob man Unicode oder ASCII verwenden möchte und dies dann über das ganze Projekt durchgängig einsetzen.
- `string` ist kein primitiver Datentyp in C++, wird aber von der Standard Library im Header `<string>` bereitgestellt. Diese Variante nutzt ASCII Buchstaben, Sie können aber auch `wstring` für Unicode-Strings verwenden.
- Statt `null` schreibt man in C++ `nullptr` für Pointer, die auf kein Objekt zeigen sollen.

Auch bei der Kontrollflussteuerung ist vieles identisch wie in C#: Konstrukte wie `if`, `else`, `while`, `do while` und `for` existieren in der gleichen Weise. Im Wesentlichen gibt es zwei Unterschiede:

- Es gibt keine `foreach` Schleife in C++. Man kann aber eine Abwandlung der `for` Schleife nutzen, um eine Collection zu durchlaufen.
- Bei `switch` muss nicht in jedem `case` ein `break` sein. Dadurch kann der Code von einem `case` in einen anderen übergehen.

7.5 Zeiger

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=EWMkseQRWZM>.

In C++ haben Sie die Möglichkeit, Zeiger einzusetzen, um bestimmte Speicherabschnitte des Heaps oder des Stacks zu referenzieren. Weitestgehend werden Zeiger genauso wie in C verwendet – schauen wir uns dazu gleich ein Beispiel an:

```

void main()
{
    Für einen Zeiger gibt
    man Typname* an
    double value1 = 42.0;
    double* pointerToValue1 = &value1;
    cout << *pointerToValue1 << endl;

    Hier wird direkt die
    Adresse, auf die der
    Zeiger zeigt, ausgegeben
    double value2(43.0);
    double* pointerToValue2 = &value2;
    cout << pointerToValue2 << endl;

    double* pointerToValue3 = new double(44.0);— Dieser double wird auf dem
    *pointerToValue3 += 1;— Heap erstellt
    cout << *pointerToValue3 << endl;
    Alle auf dem Heap allokierten
    delete pointerToValue3;— Objekte müssen manuell
    pointerToValue3 = nullptr;— gelöscht werden
}

```

Abbildung 284: Beispiel für Zeiger in C++

In Abbildung 284 sehen wir folgende Dinge:

- Im ersten Statement wird ein **double** auf dem Stack in der Variable **value1** erstellt. In der zweiten Anweisung wird die Adresse dieser Variable über den **&** Operator geholt und wiederum der Variablen **pointerToValue1** zugewiesen. Beachten Sie, dass der Typ von **pointerToValue1** vom Typ **double*** ist, also ein Zeiger auf einen **double**.
- Der Wert von **value1** wird dann wiederum über den Zeiger ausgegeben. Dazu wird der Zeiger dereferenziert, indem man vor den Variablenamen ein Sternchen ***** schreibt (***pointerToValue1**).
- Dasselbe wird in den nächsten drei Statements gemacht, mit dem Unterschied, dass **pointerToValue2** nicht dereferenziert wird und deshalb die Adresse zurückgibt, an der die Variable **value2** liegt.
- In Statements sieben bis neun wird der **double** Wert nicht auf dem Stack, sondern auf dem Heap mit **new double(44.0)** erzeugt. Der **new** Operator gibt die Adresse für den allokierten Heapspeicher zurück, welche wiederrum in der Variablen **pointerToValue3** gespeichert wird.
- Wichtig: sämtlicher Speicher, den Sie mit dem **new** Operator auf dem Heap allokiert haben, müssen Sie auch wieder vom Heap manuell deallozieren. Nutzen Sie dazu in C++ den **delete** Operator, hinter den Sie den Zeiger angeben. Der Speicherbereich, auf den der Zeiger zeigt, wird dann wieder freigegeben. **delete** ist damit das Pendant zur Funktion **free** in C.

Vor der Deallokation von **pointerToValue3** sieht deshalb das Speicherbild wie folgt aus:

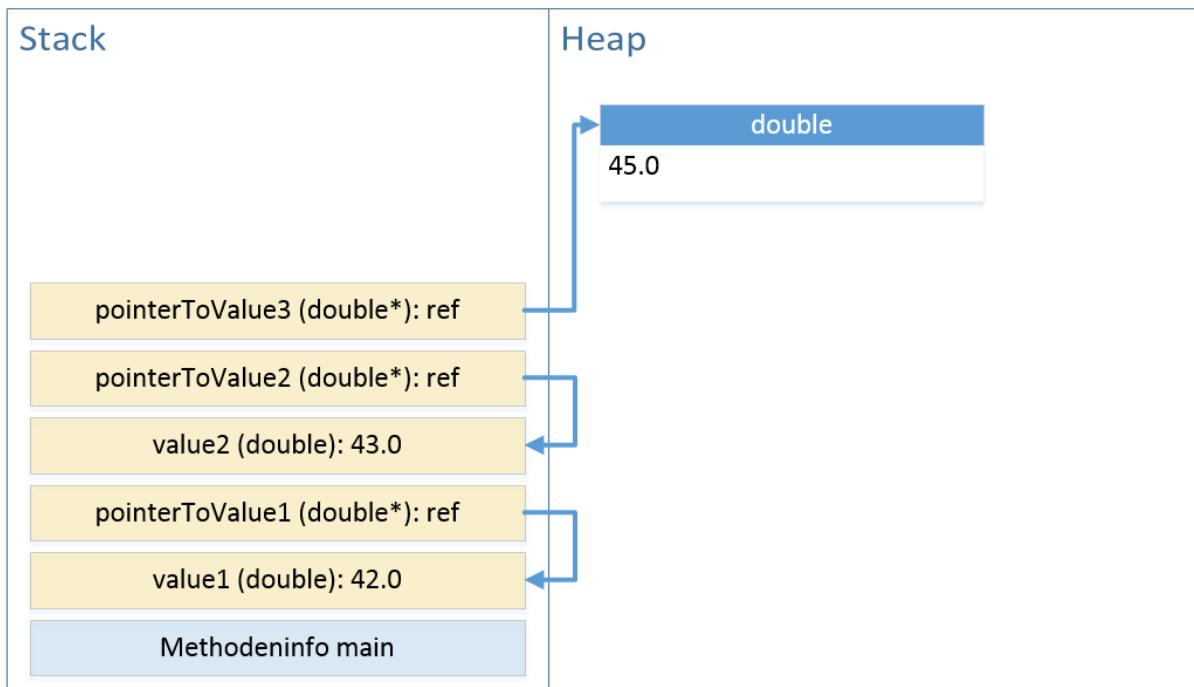


Abbildung 285: Speicherbild vor dem Deallozieren von `pointerToValue3`

Beachten Sie bitte folgende drei Dinge bei Zeigern in C++:

- Wenn Sie `new` in C++ einsetzen, wird Heapspeicher allokiert und Sie bekommen einen Pointer vom entsprechenden Typ zurück. Vergessen Sie nicht, diesen Speicher manuell zu deallozieren (oder die Smart Pointers der Standard Library einzusetzen).
- Zeiger können nicht nur auf Heapspeicher, sondern auch auf Speicherstellen im Stack zeigen. Nutzen Sie niemals `delete` in Verbindung mit einem Pointer, der auf den Stack zeigt, da sie sonst eine Exception zur Laufzeit erhalten und Ihr Programm abstürzt. Der Stackspeicher wird automatisch dealloziert, wenn eine Methode endet.
- Setzen Sie Zeigervariablen nach einer `delete` Operation auf `nullptr`. Dies zeigt an, dass ein Zeiger auf keinen Speicherbereich schaut. Wenn erneut auf diesen Zeiger `delete` aufgerufen würde, löst das keine Exception aus.

7.6 Konstante Werte und Zeiger

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=nJ692TUqb38>.

Genauso wie in C# haben Sie in C++ auch die Möglichkeit, Variablen als konstant festzulegen mit dem Schlüsselwort `const`:

```
void main()
{
    const int value1 = 35;
    value1 = 42;
}
```

Abbildung 286: Konstanten definieren in C++

Wie wir in Abbildung 286 sehen können, ist es nicht erlaubt, eine solche Konstante im weiteren Verlauf der Methode zu ändern.

In C++ können Sie `const` aber auch auf Zeiger anwenden, wobei es dort drei Möglichkeiten gibt:

```
void main()
{
    const int value1 = 35;

    Ein const vor dem Pointertyp
    bedeutet, dass man den
    referenzierten Speicherbereich nicht
    über diesen Zeiger manipulieren kann

    int value2 = 87;
    int value3 = 90;
    const int* const pointerToValue1 = &value1;
    *pointerToValue1 = 42;

    Ein const nach dem Pointertyp
    bedeutet, dass man den Zeiger nach
    Initialisierung nicht auf eine andere
    Speicherstelle zeigen lassen kann

    int value4 = 2;
    const int* pointerToValue2 = &value2;
    pointerToValue2 = &value3;

    Dieser Pointer ist eine Kombination
    aus den beiden vorherigen

    const int* const pointerToValue4 = &value4;
    *pointerToValue4 = 3;
    pointerToValue4 = &value3;
}
```

Abbildung 287: Die drei möglichen Varianten von konstanten Zeigern in C++

In Abbildung 287 sehen wir folgende Dinge:

- Bei `const Zeigertyp*` ist es einem Zeiger verboten, den referenzierten Speicher zu manipulieren.
- Bei `Zeigertyp* const` kann einem Zeiger nicht im späteren Verlauf eine andere Speicheradresse zugewiesen werden (was umgangssprachlich als „Zeiger umbiegen“ bezeichnet wird).
- Die Kombination aus beiden ist auch möglich: mit `const Zeigertyp* const` ist es weder erlaubt, die Adresse zu ändern noch den Inhalt des Speichers zu manipulieren (diese Zeigerdefinitionen sieht man aber eher selten im Programmieralltag).

Das Schlüsselwort `const` kann aber nicht nur innerhalb von Funktionen eingesetzt werden, sondern auch auf Klassenmethoden, wie wir uns später noch ansehen werden.

7.7 Referenzen in C++

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=YUiXVyu-7aY>.

7.7.1 lvalue Referenzen

Neben Zeigern, die Ihnen schon aus C bekannt sind, gibt es in C++ auch Referenzen (bitte verwechseln Sie diesen Term nicht mit C# Referenztypen).

Eine Referenz wird wie folgt definiert:

```

void main()
{
    int value = 42;
    int& referenceToValue = value;
    referenceToValue += 5;
    cout << value << endl;
}

```

Als Wert für eine Referenz kann man
eine Variable angeben

Alle Änderungen an der
Referenz ändern auch den
ursprünglichen Wert

Eine Referenz wird definiert, wenn man nach dem Typ ein kaufmännisches Und (&) schreibt

Abbildung 288: Eine Referenz in C++ definieren

In Abbildung 288 sehen wir folgende Dinge:

- Eine Referenz wird in C++ definiert, indem man nach dem Typ ein kaufmännisches Und (&) schreibt.
- Einer Referenz kann man wie im obigen Beispiel den Wert einer Variablen zuweisen. Dann schaut die Referenz auf denselben Speicherbereich (wie ein Pointer).
- Als Wert für Referenzen können allerdings auch bspw. dereferenziert Pointer dienen (auch wenn das selten gemacht wird). Letztlich kann alles, was einen dedizierten Speicherbereich darstellt, als Wert für eine Referenz genutzt werden.
- Alle Änderungen an der Referenz werden auf den referenzierten Speicherbereich angewandt.

Genau genommen haben wir es im obigen Beispiel mit sog. lvalue References zu tun. Diese sind im Wesentlichen Zeiger mit folgenden Einschränkungen:

- Einer lvalue Reference kann nach der Definition nicht eine andere Speicheradresse zugewiesen werden (kein „umbiegen“ möglich).
- Um den Wert des Speicherbereichs, auf den die lvalue Reference zeigt, zu ändern, muss die Referenz nicht dereferenziert werden.

lvalue Reference (lvalue steht übrigens für left value – das ist ein Wert, der links bei einer Zuweisung steht) spielen ihre Vorteile v.a. in Collection-Basierten-Schleifen und Parametern von Methoden aus. Dort können Sie deutliche Performancevorteile bringen, was wir im nächsten Abschnitt sehen werden.

7.7.2 lvalue Referenzen und die Range-Based-For-Loop

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ISiB5Yvf118>.

Nehmen wir einmal an, wir hätten drei Temperaturwerte in einem Array gespeichert und möchten die Durchschnittstemperatur berechnen. Der Code dazu könnte wie folgt aussehen:

```

Bei Arrays in C++ sind die eckigen Klammern nach
void main()           dem Variablenbezeichner
{
    int temperaturwerte [] = { 30, 32, 29 };

    double durchschnittstemperatur = 0.0;
    for (int wert : temperaturwerte)  Die Range-Based-For-Loop ist in C++
    {                                     der Ersatz für die foreach Schleife
        durchschnittstemperatur += (double) wert;
    }
    durchschnittstemperatur /= 3;

    cout << "Durchschnittstemperatur: " << durchschnittstemperatur << endl;
}

```

Abbildung 289: Range-Based-For-Loop in C++

In Abbildung 289 sehen wir, dass eine Range-Based-For-Loop genutzt wird, um den Array temperaturwerte zu durchlaufen. Dabei wird pro Schleifendurchlauf ein Wert des Arrays der Variablen wert zugewiesen, und zwar per Call-By-Value. Wenn wir uns also das Speicherbild beim zweiten Schleifendurchlauf ansehen, würde es wie folgt aussehen:

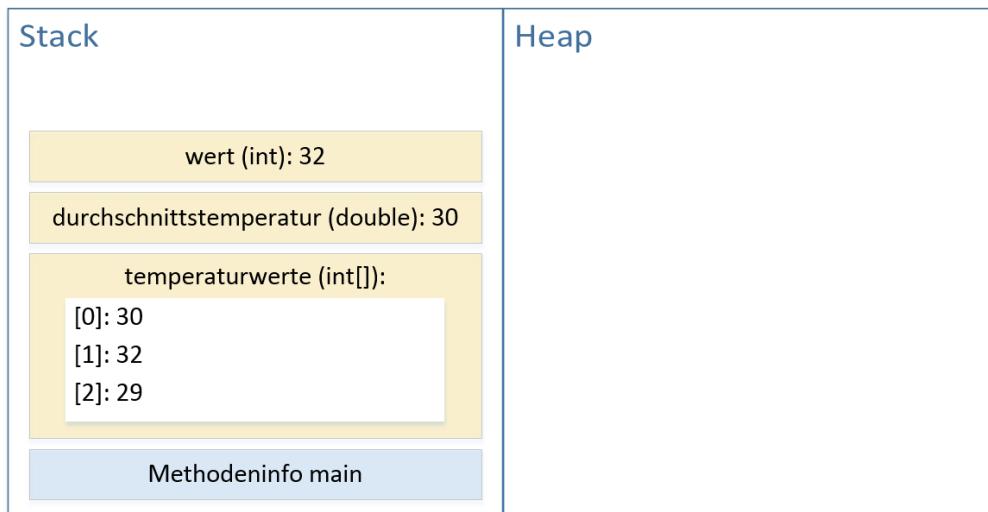


Abbildung 290: Speicherabbild beim 2. for Schleifendurchlauf des vorherigen Beispiels

In diesem Fall ist es kein Problem, dass die Werte aus dem Array via Call-By-Value an die Laufvariable der **for** Schleife übergeben werden, aber nehmen wir an, dass wir alle Temperaturwerte im Array von Celsius nach Fahrenheit innerhalb der Schleife umrechnen und auf derselben Position im Array speichern möchten – dies ist mit den oben gezeigten Mitteln nicht möglich.

Zu Hilfe kommt hier die lvalue Referenz, die wir als Laufvariable in der **for** Schleife angeben können:

```

void main()
{
    int temperaturwerte [] = { 30, 32, 29 };

    double CelsiusToFahrenheit = 9.0 / 5.0;
    for (int& wert : temperaturwerte)
    {
        wert = (wert * CelsiusToFahrenheit) + 32;
    }
}

```

Durch die Verwendung einer lvalue Referenz
können Arraywerte auch zurückgeschrieben
werden

Abbildung 291: Durch lvalue Referenzen in for Schleifen können Collectionelemente direkt manipuliert werden

Dadurch, dass in Abbildung 291 beim Typen für die Laufvariable der `for` Schleife eine lvalue Referenz verwendet wird, zeigt diese direkt auf das jeweilige Element im Array. Wenn man dieser Variablen also einen neuen Wert zuweist, ändert sich auch das entsprechende Element im Array. Das Speicherbild sieht dann beim zweiten Durchlauf wie folgt aus:

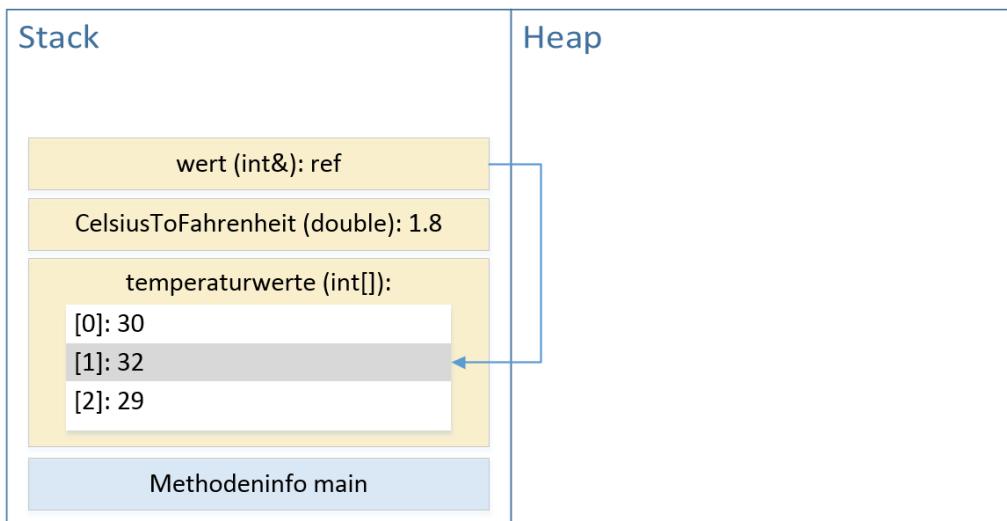


Abbildung 292: Speicherbild bei Einsatz von lvalue Referenzen in Range-Based-For-Loops

Dieses Verhalten ist mit Pointern zwar prinzipiell auch möglich, bedarf aber deutlich mehr Code und ist dadurch nicht so gut lesbar.

7.7.3 Konstante lvalue Referenzen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=5I17RaIGjCI>.

Wir haben bereits in Abschnitt 7.6 über konstante Werte und konstante Zeiger gesprochen. In C++ sind natürlich auch konstante Referenzen möglich.

Konstante lvalue Referenzen können den Speicherbereich, auf den sie verweisen, nicht verändern. Damit sind sie äquivalent mit konstante Zeiger nach dem Schema `const Pointer* const`.

Dazu wollen wir uns natürlich auch ein Beispiel anschauen:

```

void main()
{
    int value = 5;
    int& intReference1 = value;
    intReference1 += 10;

    int& intReference2 = 5;      Nicht konstanten Referenzen
    intReference2 += 10;         können keine Literale zugewiesen
                                werden

    const int& intReference3 = 5;
    intReference3 += 10;         Konstante Referenzen können den
                                Speicherinhalt nicht verändern
}

```

Abbildung 293: Konstante lvalue Referenzen

In Abbildung 293 sehen Sie folgende Dinge:

- In den ersten drei Statements passiert nichts Unbekanntes: es wird die `int` Variable `value` auf dem Stack erstellt mit dem Wert 5, danach eine Referenz auf diese Variable erstellt und der Wert um zehn erhöht.
- In den Statements vier und fünf sehen wir allerdings, dass wir normalen lvalue Referenzen keine Literale zuweisen können (im obigen Fall 5). Der Grund hierfür ist aber auch klar: Literale können ihren Wert nicht ändern – das wäre aber über die Referenz möglich.
- Die Lösung ist der Einsatz einer konstanten lvalue Referenz wie in den Statements sechs und sieben: diese dürfen laut Vorschrift nicht den Speicherbereich, auf den sie verweisen, manipulieren, weswegen hier auch Literale zugewiesen werden können.

7.7.4 rvalue Referenzen (Level 200)

In diesem Abschnitt werden wir nur kurz darauf eingehen, was eine sog. rvalue Reference ist. Diese werden häufig genutzt, um Code zu optimieren, indem Zwischenergebnisse von Ausdrücken festgehalten werden. Sehen Sie sich dazu folgendes Beispiel an:

```

void main()
{
    int x(5);
    int y = (2*x + 3) + (x * x);

Eine rvalue Referenz
wird mit zwei
kaufmännischen Und
(&&) gekennzeichnet
    int&& rValueReference1 = 2 * x + 3;      Eine rvalue Referenz kann nur
                                                ein Ausdruck zugewiesen
    int&& rValueReference2 = x;               werden

    cout << rValueReference1 << endl;
}

Ivalues können rvalue
Referenzen nicht zugewiesen
werden

```

Abbildung 294: rvalue Referenzen in C++

In Abbildung 294 sehen Sie, dass rvalue Referenzen mit zwei kaufmännischen Und (`&&`) gekennzeichnet werden. Der wichtige Punkt ist, dass sie nur Ausdrücke enthalten können, die evaluiert werden können. Die Zuweisung von lvalues, die auf der linken Seite einer Zuweisung stehen können (wie im obigen Beispiel `x`) ist nicht möglich.

Damit zeigen rvalue Referenzen nicht auf konkrete Speicherbereiche im Stack oder Heap, sondern auf temporäre Zwischenergebnisse, die beim Ausführen von Ausdrücken entstehen. Die beiden häufigsten Gründe, warum sie eingesetzt werden, sind die Implementierung von...

- ...Move Semantics
- ...Perfect Forwarding

Beide Dinge sind fortgeschrittene Themen in der C++ Programmierung, weswegen ich hier nicht weiter darauf eingehen werden. Es gibt aber genügend Tutorials und Erklärungen zu diesem Thema im Internet, u.a. unter http://thbecker.net/articles/rvalue_references/section_01.html. Das wichtigste, was Sie aus diesem Abschnitt mitnehmen sollten, ist, dass Sie rvalue References an den zwei kaufmännischen Und erkennen und wissen, dass Sie in diesem Fall keine lvalues einsetzen können (rvalues kommen bspw. in der Standard Library an einigen Stellen vor).

7.8 Wann setze ich Pointer und Referenzen ein?

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=r9A1TVL9o1w>.

Wir haben uns in den letzten Abschnitten ausführlich mit Zeigern und Referenzen auseinandergesetzt und uns dabei hauptsächlich prozeduralen Code angesehen, der zwar beispielhaft ist, aber keine tatsächliche Aussagekraft hat.

Zeiger und Referenzen haben ihre große Bedeutung in Parametern, denn über diese können Werte via Call-By-Reference statt Call-By-Value übergeben werden. In C++ sind wir selbst für diese Details zuständig, wohingegen in C# uns diese Aufgabe abgenommen wurde durch das automatische Verhalten von Value Types und Reference Types.

Des Weiteren sollten Sie folgende Unterschiede zwischen Zeigern und lvalue Referenzen beachten:

- Ein Pointer kann eine neue Adresse zugewiesen werden, einer Referenz jedoch nicht. Letztere verhalten sich also immer wie konstante Pointer.
- Zeigern kann direkt `nullptr` zugewiesen werden, Referenzen jedoch nicht. Dennoch ist es möglich, dass Referenzen auf `nullptr` zeigen, wie in Abbildung 295 zu sehen ist.
- Es gibt keine Referenzarithmetik, so wie es Zeigerarithmetik gibt. Referenzen können also bspw. nicht inkrementiert werden, um auf die nächste Speicherstelle zu schauen.

```
void main()
{
    int* intPointer = nullptr;
    int& intReference = *intPointer;
}
```

Dieser Referenz wird `nullptr`
zugewiesen

Abbildung 295: Auch Referenzen können über dereferenzierte Zeiger `nullptr` zugewiesen werden

Intern wird eine lvalue Referenz vom Compiler als Zeiger umgesetzt. Deswegen können Sie sich letztendlich aussuchen, welche Variante bei Funktionen Ihnen besser passt. Im folgenden Beispiel sehen Sie die Methode Square einmal mit Zeiger, einmal mit Referenz:

```

int Square(const int const* number)
{
    if (number == nullptr)
        throw new invalid_argument("number is nullptr");
    return (*number) * (*number);
}

int Square(const int& number)
{
    if (&number == nullptr)
        throw new invalid_argument("number is nullptr");
    return number * number;
}

```

Überprüfen Sie sowohl bei Zeigern als auch bei Referenzen auf nullptr

Abbildung 296: Referenzen und Zeiger als Mittel zur Übergabe von Werten via Call-By-Reference

In Abbildung 296 sehen Sie, dass der Code mit Referenz etwas einfacher ist als der mit Pointer, das Prinzip dahinter ist aber gleich: der Parameter `number` wird beim Methodenaufruf nicht kopiert, sondern als Verweis auf einen bestehenden Speicherbereich (sei es auf dem Stack oder auf dem Heap) übergeben.

Nutzen Sie auch in C++ in Ihren Funktionen Guard Clauses, um eingehende Parameter zu evaluieren und ggfs. eine Exception zu werfen. Überprüfen Sie bei Zeigern und Referenzen jeweils, ob diese `nullptr` sind.

7.9 Klassen in C++

7.9.1 Klassendeklaration und -Definition

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=oAfNHEVDQcc>.

Wie wir schon vorher erwähnt haben, werden Klassen in C++ üblicherweise nicht in eine Datei geschrieben, sondern in eine Header Datei und eine Source Datei aufgeteilt. Das kann man natürlich händisch machen, indem man zwei Mal eine Datei zum Projekt hinzfügt, jedoch bietet Visual Studio genau dafür einen Wizard an, den Sie über einen Rechtsklick auf Projekt -> Add -> Class... erreichen:

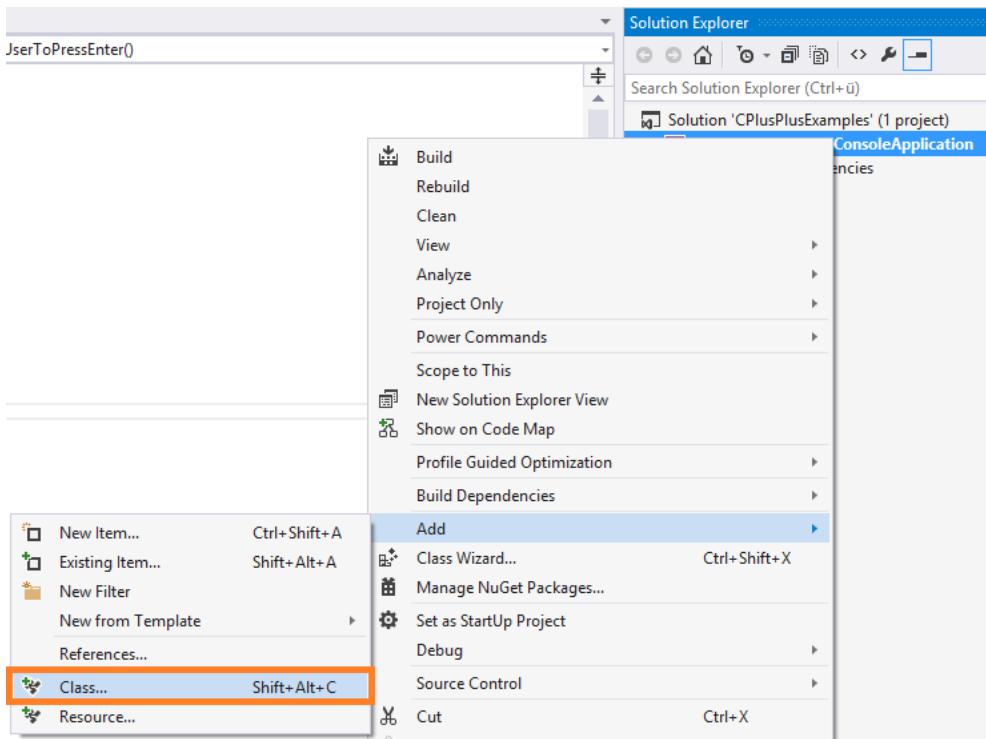


Abbildung 297: C++ Klassendialog in Visual Studio öffnen

Im Anschluss müssen Sie nur mit Add die Auswahl der C++ Klasse bestätigen und Sie erhalten folgenden Dialog, indem Sie nur noch den Klassennamen eingeben müssen:

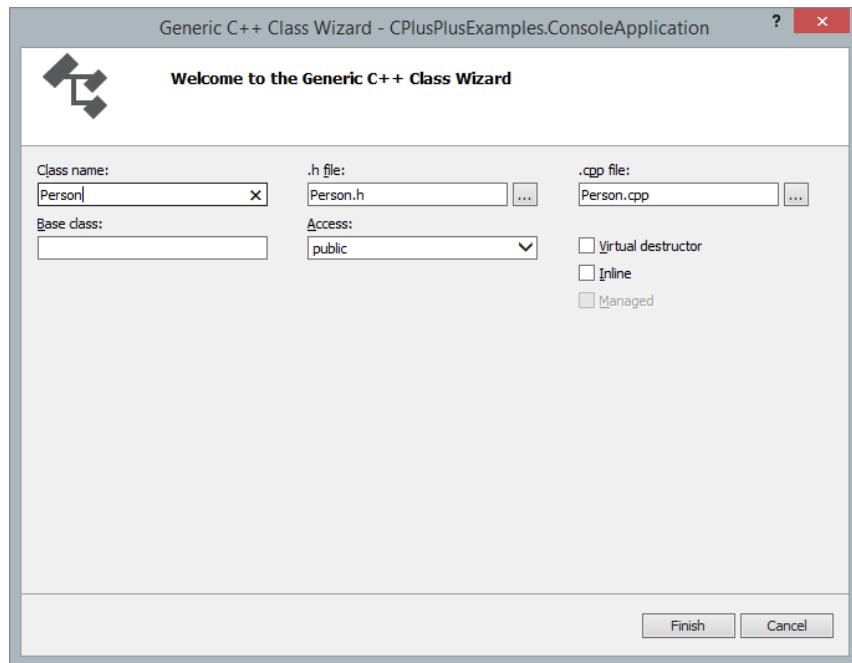


Abbildung 298: Der C++ Klassendialog in Visual Studio

Mit einem Klick auf Finish werden dann die dazugehörigen Header und Source Files erzeugt.

Öffnen Sie dann die Header-Datei und deklarieren Sie dort Ihre Klasse. Im Wesentlichen trägt man hier die Felder und Methodendeklarationen einer Klasse ein. Diese könnte für die Klasse Person wie folgt aussehen:

```

#pragma once

#include <string>

class Person
{
private:
    std::string _vorname;
    std::string _nachname;
    int _alter;

public:
    Person(std::string vorname, std::string nachname, int alter);
    std::string GetName();
    int GetAlter();
};


```

Klassen haben in C++ keine Modifizierer

Accessors für Members bilden Sektionen und werden nicht auf jedes Member angewandt

Die Trennung zwischen Namespace und Typname geschieht mit ::

Abbildung 299: Die Header Datei der Klasse Person

In Abbildung 299 finden Sie die Inhalte der Headerdatei der Klasse Person. Im Folgenden werden wir aufzählen, was dabei zu beachten ist:

- Nach der Klassendeklaration muss ein Semikolon stehen (das ist bei C# nicht der Fall).
- Die Klassendeklaration hat in C++ keine Modifizierer. Man gibt ausschließlich `class Klassenname` an. Danach folgen wie gewohnt die geschweiften Klammern für den Klassenscope.
- Anders als in C# werden die Zugriffsmodifizierer (engl. Accessors) nicht auf jedes Mitglied gesetzt, sondern in Sektionen angegeben. Dazu gibt man bspw. einmal `private` gefolgt von einem Doppelpunkt an – alle darauffolgenden Mitglieder sind dann automatisch private, bis der nächste Accessor gesetzt wird. Im obigen Beispiel werden die drei Felder `private` beschrieben, der Konstruktor und die beiden Get-Methoden sind `public`.
- In dieser Headerdatei wurden keine `using` Statements zur automatischen Einbindung von Namespaces benutzt, weswegen der vollqualifizierte Name für den Typen `string` angegeben werden muss. Dabei trennt man zwischen Namespace und Typname mit :: (zwei Doppelpunkten).

Verwenden Sie in Ihren Headerdateien keine using Statements, die Namenräume einbinden. Der Grund ist, dass sämtliche Dateien, die ihren Header einbinden, ebenfalls diese using Statements besitzen, was teilweise nicht gewünscht ist und ggfs. auch subtile Bugs auslösen kann, wenn es zu Namenskonflikten kommt.

Nachdem wir die Klasse in der Headerdatei definiert haben, können wir jetzt die fehlenden Methoden in der cpp Datei definieren:

```

#include "Person.h"          Die Headerdatei wird eingebunden
using std::string;          In Source Files ist die Einbindung von
                            Namensräumen vollkommen ok

Person::Person(string vorname, string nachname, int alter)
    : _vorname(vorname), _nachname(nachname), _alter(alter)
{
    Feldwerte können direkt über diese
    Syntax zugewiesen werden
}

int Person::GetAlter()
{
    return _alter;
}

string Person::GetName()
{
    return _vorname + " " + _nachname;
}

```

In der cpp Datei wird kein Klassenscope aufgemacht, sondern die Methoden der Klasse wie freie Methoden implementiert, wobei vor dem Bezeichner der Klassenname mit Doppelpunkt stehen muss.

Abbildung 300: Die Source Datei der Klasse Person

Gehen wir auch hier die einzelnen Punkte durch, die wir im Source File in Abbildung 300 finden:

- Im Source File referenziert man üblicherweise die Headerdatei, deren Deklarationen definiert werden. Bitte beachten Sie, dass bei diesem `#include` der Headerdatei in doppelten Anführungszeichen gesetzt werden und nicht in spitzen Klammern wie bei den Headers der Standard Library (der Grund dafür ist, dass der Compiler beim Einbinden an anderen Orten sucht).
- In Source Files können Sie ohne Probleme Namespaces einbinden, wie es oben mit `using std::string;` getan wird.
- In der Source Datei wird der Klassenscope nicht erneut geöffnet, sondern die Methoden der deklarierten Klasse als frei liegende Funktion implementiert. Der Unterschied zu richtigen freien Funktionen ist, dass vor allen Funktionsbezeichnern der Klassenname gefolgt von `::` (zwei Doppelpunkten) stehen muss. Dies ordnet die Definition der entsprechenden Deklaration in der Klasse zu.
- In Konstruktoren können Sie Feldwerte nicht nur innerhalb des Konstruktorscopes setzen, sondern auch mit der oben gezeigten Syntax: dazu schreiben Sie einfach einen Doppelpunkt nach der Parameterliste des Konstruktors und führen dann kommagetrennt die Zuweisungen nach dem Schema `_fieldname(parametername)` auf. Diese Art der Feldzuweisung ist sogar noch schneller als die Zuweisung innerhalb des Scopes, denn bei letzterem wird jedem Feld zunächst der Standardwert des jeweiligen Typs zugewiesen.

Damit ist unsere erste C++ Klasse deklariert und definiert. Wir können uns jetzt mit der Instanziierung von Klassen beschäftigen.

7.9.2 Klassen auf Stack oder Heap instanzieren

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=5EzPbt2L6Hc>.

Wie bereits erwähnt können wir in C++ Objekte sowohl auf dem Heap als auch auf dem Stack erstellen. Im folgenden Beispiel wird beides gemacht mit der eben geschriebenen Klasse `Person`:

```

Diese Instanz wird mit dem new Operator auf dem Heap
erstellt. Die Variable ist ein Zeiger auf dieses Objekt.

void main()
{
    Person* person1 = new Person("Walter", "White", 52);           | Diese Instanz liegt auf dem Stack
                                                                direkt in Variable person2

    Person person2("Jesse", "Pinkman", 27);                         | Bei Zeigern nutzt man den
                                                                Pfeiloperator -> zum Zugriff

    cout << person1->GetName() << endl;
    cout << person2.GetName() << endl;                                | Bei Objekten auf dem Stack nutzt man
                                                                den Punktoperator . zum Zugriff

    RequestEnterFromUser();
    delete person1;                                                 | Objekte auf dem Heap müssen
                                                                manuell deallokiert werden, sonst
                                                                entsteht ein Speicherloch
}

```

Abbildung 301: Objekte auf Heap und Stack erzeugen in C++

In Abbildung 301 sehen wir folgende Dinge:

- In der ersten Anweisung wird ein Objekt der Klasse **Person** auf dem Heap erzeugt mit dem **new** Operator. Genauso wie in C# sorgt **new** dafür, dass der entsprechende Speicher im Heap allokiert wird und im Anschluss der Konstruktor aufgerufen wird, um die einzelnen Felder zu initialisieren. Zuletzt wird die Adresse dieses neuen Objekts in der Variablen **person1** festgehalten – diese muss zwangsweise ein Pointer sein, weswegen nach dem Typnamen noch der * (Stern) steht.
- In der zweiten Anweisung wird ebenfalls ein Objekt erstellt, allerdings liegt dies jetzt auf dem Stack allokiert. Dazu definiert man in C++ wie gewohnt eine Variable und setzt dahinter geschweifte Klammern, die den Konstruktorauftruf kennzeichnen (hier gibt man die Parameter an).
- In den nächsten beiden Statements werden die Namen der beiden Personenobjekte ausgegeben. Bei Pointern wird zum Dereferenzieren und Zugriff der Pfeiloperator (->) genutzt, der eigentlich nur eine abkürzende Schreibweise für (*person1).GetName() ist. Da **person2** direkt auf dem Stack liegt, muss hier nur der Punktoperator zum Zugriff genutzt werden.
- Bitte beachten Sie: in nativem C++ gibt es keinen Garbage Collector – d.h. wir müssen das Objekt, das im Heap liegt und von der Variable **person1** referenziert wird, manuell deallokiieren. Das funktioniert mit dem **delete** Operator, hinter den man einen Pointer angibt. Der Speicherbereich, auf den der Pointer zeigt, wird dann gelöscht.
- Alle Objekte, die auf dem Stack allokiert wurden, werden automatisch deallokiert, wenn die entsprechende Methode endet. Deswegen müssen wir auf **person2** nicht **delete** aufrufen.

Das Speicherbild sieht bei diesem Beispiel also wie folgt aus:

Code

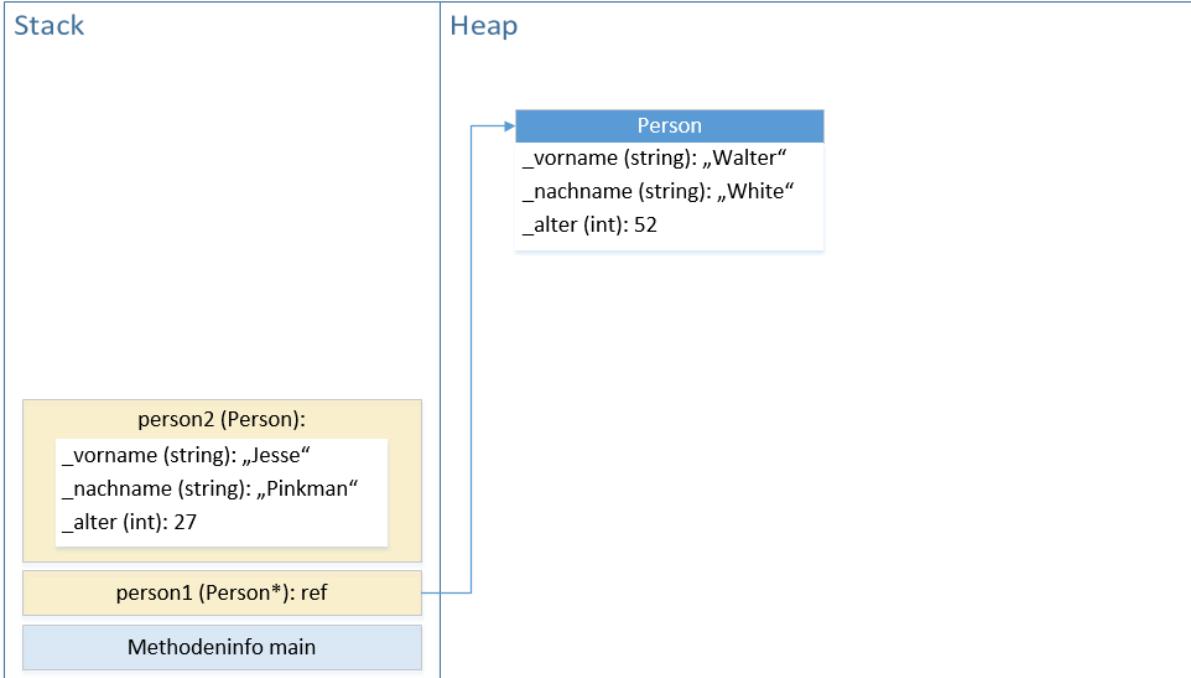


Abbildung 302: Speicherabbild nach dem Erstellen der beiden Objekte auf Stack und Heap

Bitte beachten Sie auch, dass wir in diesem Fall nicht nur eines der Personenobjekte auf dem Stack erstellt, sondern auch keine expliziten Stringobjekte im Heap haben, wie es bei C# der Fall wäre. Sie sind jeweils direkt Teil der jeweiligen Objekte, da in der Klasse Person keine Pointer auf Strings definiert wurden, sondern direkt Strings.

Im Gegensatz zu C# haben Sie also die Möglichkeit, bei der Instanziierung von Klassen selbst zu entscheiden, ob sich das Resultat wie ein Value Type oder Reference Type verhält (um im CLR Jargon zu bleiben). Das bringt natürlich große Freiheiten und Möglichkeiten zur Optimierung mit sich, allerdings sind auch viele Fallstricke zu beachten. Welche Prinzipien hier eine gute Lösung darstellen, sehen wir uns in einem späteren Abschnitt an.

7.9.3 Konstruktoren in C++

7.9.3.1 Explizite Konstruktoren

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=jd0F11C3gOA>.

Konstruktoren funktionieren genauso wie in C#, es gibt aber einen wesentlichen Unterschied: explizite Konstruktoren. Sehen Sie sich dazu folgendes Beispiel an:

```

class Rechteck
{
private:
    int _breite;
    int _höhe;

public:
    Rechteck(int seitenlänge)
    {
        if (seitenlänge < 1)
            throw new std::invalid_argument("...");

        _breite = seitenlänge;
        _höhe = seitenlänge;
    }

    Rechteck(int breite, int Höhe)
    {
        if (breite < 1)
            throw new std::invalid_argument("...");
        if (Höhe < 1)
            throw new std::invalid_argument("...");

        _breite = breite;
        _höhe = Höhe;
    }

    int BerechneFläche();
};

```

Die Klasse Rechteck besitzt zwei Konstruktoren

Abbildung 303: Die Klasse Rechteck besitzt zwei Konstruktoren

In Abbildung 303 sehen Sie, dass die Klasse **Rechteck** zwei Konstruktoren besitzt – bei einem kann man die Parameter **breite** und **höhe** explizit angeben, beim Konstruktor mit einem Parameter legt man nur die Seitenlänge an, die dann sowohl auf **breite** als auch **höhe** gesetzt wird.

Der Punkt ist, dass Konstruktoren mit einem Parameter automatisch zur impliziten Konvertierung von Parametertyp zu Klassentyp genutzt werden (in unserem Fall also von **int** zu **Rechteck**). Sehen Sie sich dazu folgende **main** Methode an:

```

void main()
{
    Rechteck rechteck = 4;

    cout << rechteck.BerechneFläche() << endl;
}

```

Hier wird automatisch der Konstruktor mit einem Parameter aufgerufen

Abbildung 304: Implizite Aufrufe für Konstruktoren mit einem Parameter

In Abbildung 304 sehen Sie, dass eine **Rechteck**-Instanz auf dem Stack erstellt wird, indem der Variable 4 zugewiesen wird. Dies funktioniert, weil in der Klasse **Rechteck** ein Konstruktor existiert, der einen Parameter vom Typ **int** hat.

Wenn Sie dies verhindern möchten, müssen Sie den Konstruktor mit einem Parameter mit dem Modifizierer `explicit` kennzeichnen:

`explicit Rechteck(int seitenlänge).`

Bitte beachten Sie, dass Sie auch Konstruktoren mit mehreren oder keinen Parametern als `explicit` kennzeichnen können – Sinn ergibt das aber nicht und wird auch vom Compiler beim Erstellvorgang verworfen. Weiterhin ist es nicht möglich, über diese Syntax Objekte auf dem Heap zu erstellen – implizite Konstruktoraufrufe können nur für Variablen bzw. Parameter auf dem Stack genutzt werden.

7.9.3.2 Copy-Konstruktoren

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=QY9eKMrm704>.

Genauso wie in C# gibt es in C++ einen Standardkonstruktor, der automatisch vom Compiler erstellt wird, wenn wir als Programmierer keine Konstruktoren in einer Klasse angeben haben. Unabhängig davon gibt es in C++ ebenfalls einen sog. Copy-Konstruktor, der ebenfalls implizit erstellt wird, sofern wir keinen angeben. Sehen Sie sich folgendes Beispiel an:

```
void main()
{
    Rechteck rechteck1(4, 6);
    Rechteck rechteck2 = rechteck1;
    Rechteck rechteck3(rechteck1);

    Rechteck* rechteck4 = new Rechteck(rechteck1);
}
```

rechteck2 und rechteck3 werden über den Copy-Konstruktor auf dem Stack erstellt

rechteck4 wird auf dem Heap über den Copy-Konstruktor erstellt

Abbildung 305: Instanzen über Copy-Konstruktoren erstellen

In Abbildung 305 sehen Sie folgende Dinge:

- `rechteck2` und `rechteck3` werden jeweils auf dem Stack mit dem Copy-Konstruktor von `Rechteck` initialisiert und enthalten damit dieselben Werte für `_breite` und `_höhe` wie `rechteck1`.
- Analog dazu kann man den Copy-Konstruktor auch für Instanzen auf dem Heap anwenden, indem man ihn bei einem `new` Aufruf verwendet. Dies geschieht bei `rechteck4`.

Der Standard-Copy-Konstruktor, der für jede Klasse erzeugt wird, kopiert dabei einfach jedes Bit vom Ursprungs- in den Zielspeicherbereich. Insbesondere wenn Klassen allerdings auf andere Elemente via Zeigern verweisen, kann es notwendig sein, einen eigenen Copy-Konstruktor für eine Klasse zu definieren. Für die Klasse `Rechteck` würde dieser wie folgt aussehen:

```

Rechteck(const Rechteck& rechteck)
{
    // Spezieller Code für Copy-Konstruktor
}

```

Abbildung 306: Copy-Konstruktor für Klasse Rechteck

In Abbildung 306 sehen Sie, dass Sie einen Copy-Konstruktor definieren können, indem Sie als Parametertyp eine konstante Referenz auf den Klassentypen einsetzen. Innerhalb eines solchen Konstruktors können Sie dann selber festlegen, wie die Feldwerte des anderen Objekts jeweils übernommen werden. Ich möchte Ihnen aber noch folgenden Hinweis geben:

Wenn Sie in C++ Smart Pointers einsetzen und nach den SOLID-Prinzipien programmieren, dann sollten Sie nur in seltenen Ausnahmefällen den Copy-Konstruktor in Ihren Klassen definieren müssen. Das hängt vor allem damit zusammen, dass ein einzelnes Objekt nur in seltenen Fällen auch für die Lebenszeit von anderen Objekten zuständig ist.

7.9.4 Destruktoren in C++

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=PYdIUdQPJ1A>.

In C++ hat der Destruktor einen ganz anderen Stellenwert als in C#, wo er nur implementiert werden sollte, um native Ressourcen wie bspw. Dateihandles, Datenbank- oder Netzwerkverbindungen zu schließen.

In C++ hat jede Klasse einen Standarddestruktor, welcher die letzte Methode repräsentiert, die beim Deallocieren eines Objekts aufgerufen wird. Dieser Standarddestruktor ist üblicherweise leer, d.h. es wird standardmäßig einfach der Speicherbereich, den das Objekt belegt. Das kann aber zu Speicherlöchern führen, wenn eine Klasse bspw. im Konstruktor eine Abhängigkeit auf dem Heap erstellt:

```

class Dependency { /* ... */ };

class DependentClass
{
private:
    Dependency* _dependency;           DependentClass erstellt sich im
                                         Konstruktor ein Objekt auf dem
                                         Heap, welche es in seinen Methoden
                                         verwendet

public:
    DependentClass()
    {
        _dependency = new Dependency();   ← Hier wird ein neues Objekt auf dem Heap
                                         erstellt und in _dependency gespeichert

        // Hier könnten andere Methoden untergebracht sein,
        // welche _dependency nutzen
    }

    ~DependentClass()
    {
        delete _dependency;            ← Würde das im Konstruktor erstellte
                                         Objekt hier im Destruktor nicht
                                         gelöscht, entstünde ein Speicherleck
    }
};

```

Abbildung 307: Destruktor in C++

In Abbildung 307 sehen Sie, dass die Klasse `DependentClass` sich eine Abhängigkeit im Konstruktor selbst erstellt mit dem Aufruf `new Dependency()`. Würde diese Klasse über keinen expliziten Destruktor verfügen, in dem der Speicherbereich für die Abhängigkeit wieder freigegeben wird, dann würde es zu einem Speicherleck kommen, wenn ein Objekt von `DependentClass` deallokiert wird.

Dieses Programmiermuster sollten Sie in C++ genauso wenig verfolgen wie in C#. Lassen Sie sich die Abhängigkeiten für Klassen von außen übergeben (Dependency Injection) und nutzen Sie Smart Pointers, die Speicher auf dem Heap automatisch deallokiieren, wenn sie Out-Of-Scope gehen. Das sorgt dafür, dass Sie nur in seltenen Fällen einen Destruktor implementieren müssen.

7.9.5 Statische Klassenmethoden

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=T9NYZ3PFiOA>.

Genauso wie in C# können Sie auch in C++ Methoden in Klassen als statisch kennzeichnen – Sie brauchen dann kein Objekt mehr, um diese Methoden aufzurufen.

```
class Mathe {  
public:  
    static int Potenziere(int zahl, int exponent);  
};  
  
→ int Mathe::Potenziere(int wert, int exponent)  
{  
    if (exponent < 0)  
        throw new invalid_argument("...");  
  
    int rückgabewert = 1;  
  
    for (int i = 0; i < exponent; i++)  
    {  
        rückgabewert *= wert;  
    }  
  
    return rückgabewert;  
}
```

Mit dem Modifizierer `static` können Methoden als statisch deklariert werden

`static` darf bei der Definition nicht angegeben werden

Abbildung 308: Statische Methoden in Klassen deklarieren und definieren

In Abbildung 310 sehen Sie, dass die Methode `Potenziere` der Klasse `Mathe` als statisch gekennzeichnet wurde. Den Modifizierer `static` dürfen Sie dabei nur bei der Deklaration innerhalb des Klassenscopes anwenden, bei der Definition in der cpp Datei darf er nicht mitangegeben werden.

Die Methode kann dann von Nutzern wie folgt aufgerufen werden:

```

    Statische Methoden
    werden in C++ über
    Klassenname:: aufgerufen
void main()
{
    auto ergebnis = Mathe::Potenziere(2, 4);

    cout << "2 hoch 4 ist " << ergebnis << endl;
}

```

Abbildung 309: Statische Methoden in C++ aufrufen

In Abbildung 309 sehen Sie, dass statische Methoden über das Schema **Klassenname**::**Methodename**(**Parameterliste**) aufgerufen werden. Zwischen Klassenname und Methodenname wird also kein Punkt zur Trennung eingesetzt, sondern zwei Doppelpunkte.

Genauso wie in C# bieten statische Methoden (bzw. freie Funktionen) in C++ keine Möglichkeit für polymorphe Aufrufe. Bedenken Sie also, dass Ihr Code unflexibler wird, je mehr Aufrufe dieser Art in ihm vorkommen.

7.9.6 Konstante Klassenmethoden

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=G4eAPU5ZhdA>.

Neben der Möglichkeit, Variablen, Zeiger und Referenzen als konstant zu deklarieren, gibt es in C++ auch die Möglichkeit, Methoden in Klassen als konstant zu kennzeichnen. Das bedeutet nichts anderes, als dass der Aufruf dieser Methode nichts am Status des Objekts oder von statischen Feldern ändert. Sehen wir uns dazu ein Beispiel an:

```

class A
{
private:
    int _zähler = 0;
public:

    void ErhöheZähler();
    int GetZähler() const;
};

void A::ErhöheZähler()
{
    _zähler++;
}

int A::GetZähler() const
{
    return _zähler;
}

```

Mit const nach der Parameterliste kann man eine Methode als konstant deklarieren bzw. definieren

Abbildung 310: Membermethoden als konstant definieren

In Abbildung 310 sehen Sie, dass der Getter **GetZähler** mit dem Modifizierer **const** nach der Parameterliste gekennzeichnet ist, um auszusagen, dass innerhalb dieser Methode keine Feldwerte

geändert werden. Natürlich darf man in solchen Methoden auch keine Änderungen an Feldern vornehmen, also bspw. `_zähler` erhöhen. Dies würde der Compiler mit einem Fehler quittieren.

Achten Sie beim Schreiben Ihrer Klassen auf die sog. **Const Correctness**:

- Kennzeichnen Sie Methoden als konstant, wenn Sie keine Feldwerte verändern.
- Wenn Sie Parameterwerte als Zeiger oder Referenzen übernehmen, machen Sie diese konstant, sofern Sie den Zeiger nicht umbiegen und / oder den Speicherbereich, auf den Sie verweisen, nicht verändern. Kennzeichnen Sie auch Felder mit Pointer- / Referenztypen nach demselben Muster.

Klassen nachträglich mit Const Correctness auszustatten ist meines Erachtens eine sehr langwierige Arbeit. Ich rate ihnen deshalb, Ihre Methoden, Parameter und Felder direkt bei der Erstellung mit `const` zu kennzeichnen (bspw. können Sie jeden Getter als konstant kennzeichnen, da diese Methoden üblicherweise nur einen (Feld-)Wert zurückgeben).

7.9.7 Klassenvererbung in C++

7.9.7.1 Von einer Basisklasse ableiten

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=QD9x8yvFn7o>.

Prinzipiell funktioniert Vererbung in C++ genauso wie in C#, allerdings gibt es auch hier einige feine Unterschiede. Sehen wir uns dazu gleich ein Beispiel an:

```

class A
{
private:
    int _zähler = 0;
public:

    void ErhöheZähler();
    int GetZähler() const;
};

void A::ErhöheZähler()
{
    _zähler++;
}

int A::GetZähler() const
{
    return _zähler;           Beim Ableiten muss vor der Basisklasse ein
}                                         Modifizierer angegeben werden

class B : public A
{
public:
    void ErhöheZählerUm3();
};

void B::ErhöheZählerUm3()
{
    for (int i = 0; i < 3; i++)
    {
        A::ErhöheZähler();      Auf Mitglieder der Basisklasse wird über
    }                         Klassenname:: zugegriffen (diese Angabe ist
}                                         optional, wenn es keinen Namenskonflikt gibt)
}

```

Abbildung 311: Beispiel für Vererbung in C++

In Abbildung 311 sehen Sie, dass die uns bereits bekannte Klasse **A** als Basisklasse für **B** fungiert. Beim Ableiten muss in C++ allerdings vor der Basisklasse ein Zugriffsmodifizierer angegeben werden, was folgende Auswirkungen hat:

- Wenn sie vor der Basisklasse **public** angeben (wie im obigen Fall), dann werden alle **public** und **protected** Members der Basisklasse wie gewohnt übernommen. **private** Members der Basisklasse können nicht in der Subklasse angesprochen werden.
- Bei der Angabe von **protected** werden **public** Members der Basisklasse zu **protected** Members der Subklasse.
- Bei der Angabe von **private** werden **public** und **protected** Members der Basisklasse zu **private** Members der Subklasse.

Mit dieser Angabe kann man also die Sichtbarkeit der Basisklassenmitglieder für die Subklasse verändern, was durchaus auch zu Zwecken der Kapselung eingesetzt werden kann.

Leiten Sie von Basisklassen standardmäßig public ab, ansonsten können Sie die Ist-Eine-Beziehung der Vererbung nicht anwenden (das ist insbesondere beim Einsatz von Basisklassen als Abstraktionen wichtig).

Leiten Sie von Basisklassen nur dann protected oder private ab, wenn Sie wissen was Sie tun. Üblicherweise nutzt man das, um Kapselung in C++ über Vererbung herzustellen.

Weiterhin sehen wir im obigen Beispiel auch, dass es das C# Schlüsselwort **base** in C++ nicht gibt. Wenn man auf Mitglieder der Basisklasse zugreifen möchte, gibt man einfach den Klassennamen gefolgt von zwei Doppelpunkten und dem entsprechenden Mitgliedsbezeichner an (im obigen Beispiel `A::ErhöheZähler`). Diese Angabe ist allerdings optional und wird üblicherweise vom Compiler automatisch gesetzt, wenn wir sie nicht schreiben.

7.9.7.2 Die Ist-Eine-Beziehung zwischen Basis- und Subklasse in C++

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=JbD85f0kpig>.

Die Ist-Eine-Beziehung haben wir in C# als den Mechanismus kennengelernt, mit dem wir Objekte über Referenzen zu einer ihrer Basisklassen ansprechen können. Diese Funktionalität ist elementar wichtig für Polymorphie, denn hinter Basisklassenreferenzen, die als Abstraktionen fungieren, können beliebe Objekte gesteckt werden, welche diese Abstraktionen implementieren.

Genau dasselbe ist natürlich in C++ auch möglich: wir können Zeiger oder Referenzen auf Objekte (im Heap oder Stack) nicht nur vom eigentlichen Typ, sondern auch von Basisklassentypen erstellen.

Betrachten Sie dazu folgendes Beispiel:

```
void main()
{
    B b;
    b.ErhöheZählerUm3();

    A* pointer = &b;
    pointer->ErhöheZähler();

    cout << pointer->GetZähler() << endl;
}
```

Eine Instanz von B kann mit einem Pointer auf A angesprochen werden

Analog geht das auch mit Referenzen

Abbildung 312: Die Ist-Eine-Beziehung mit Zeigern und Referenzen in C++

In Abbildung 312 sehen Sie, dass eine Instanz der von vorigen Beispielen bekannten Klasse B auf dem Stack erstellt wird und dann jeweils ein Pointer und eine Referenz vom Basisklassentyp A auf dieses Objekt erstellt wird. Über diese wird dann jeweils `ErhöheZähler` aufgerufen.

Solange Pointer oder Referenzen für diese Zwecke eingesetzt werden, ist alles wie in C#, aber sobald Sie Call-By-Value Semantik einsetzen, ändert sich das drastisch – betrachten Sie dazu folgendes Beispiel:

```

class X
{
private:
    int _feldInX;
public:
    X(int x) : _feldInX(x) { }
    int GetX() const { return _feldInX; }

};

class Y : public X
{
private:
    double _feldInY;
public:
    Y(int x, int y) : X(x), _feldInY(y) { }
    int GetY() const { return _feldInY; }
};

void main()
{
    Y yObject(42, 20);
    X xObject = yObject;
}

```

Bei dieser Zuweisung entsteht
 Slicing, da durch den Call-By-Value
 Aufruf alle Informationen der
 Klasse Y abgeschnitten werden

Abbildung 313: Slicing bei der Ist-Eine-Beziehung via Call-By-Value

In Abbildung 313 sehen Sie, dass eine Instanz der Klasse **Y** auf dem Stack erstellt wird und dann via Call-By-Value der Variable **xObject** zugewiesen wird, welche vom Basisklassentyp **X** ist (hier wird übrigens implizit der Copy-Konstruktor ausgeführt). Wichtig ist, dass hier kein Pointer bzw. keine Referenz im Spiel ist, weswegen letztendlich folgendes Memory-Abbild entsteht:

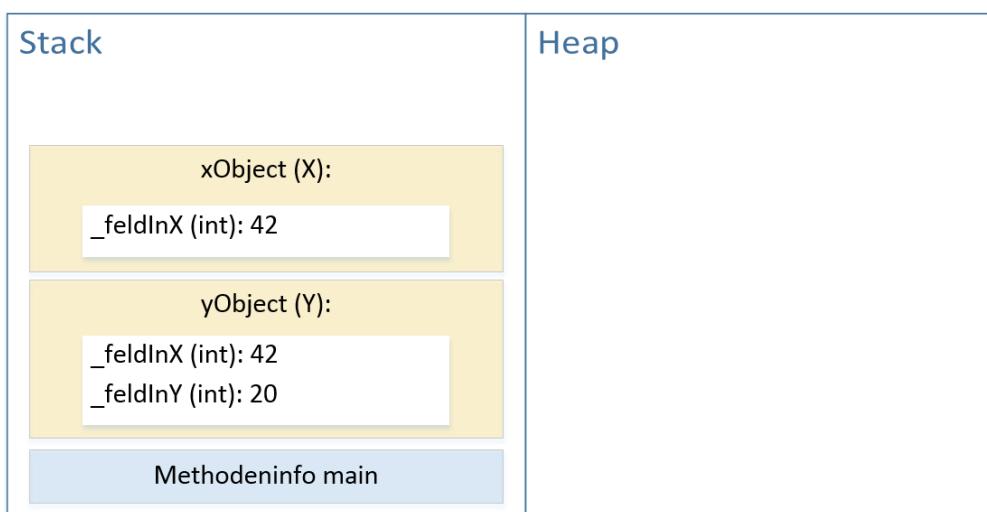


Abbildung 314: Memory-Abbild nach Slicing

Wenn Sie die Ist-Eine-Beziehung über Call-By-Value einsetzen, wird das entsprechende Objekt beim Kopiervorgang zerstückelt – d.h. alle Informationen, die zur Subklasse gehören, werden abgeschnitten, da tatsächlich eine neue Instanz der Basisklasse mithilfe der Informationen eines Objekts der Subklasse gebildet wird.

Nutzen Sie deshalb für die Ist-Eine-Beziehung immer Referenzen oder Zeiger, außer Sie möchten Slicing als expliziten Mechanismus (ich kann mir aber kein Szenario vorstellen, in dem Sie das wirklich möchten).

Vermeiden Sie Slicing – nutzen Sie Pointer und Referenzen in C++, wenn Sie Objekte über Ihre Abstraktion ansprechen möchten.

Als letzten Punkt in diesem Abschnitt möchte ich noch darauf hinweisen, **dass die Ist-Eine-Beziehung nicht möglich ist, wenn Sie von einer Basisklasse mit den Accessors protected oder private ableiten**. Der Compiler verhindert in diesen Fällen, dass Sie Objekte einer Subklasse über eine Basisklasse ansprechen – sei es über Call-By-Value oder Call-By-Reference.

7.9.7.3 Abstraktionen in C++

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=1nErNEPzcew>.

Wie bereits erwähnt, gibt es in C++ keine Interfaces und wir haben ausschließlich über abstrakte Basisklassen die Möglichkeit, Abstraktionen zu bilden. Die Syntax dazu für virtuelle Methoden ist identisch im Vergleich zu C#, für abstrakte Methoden unterscheidet sie sich aber deutlich, wie wir im Beispiel in Abbildung 315 sehen können.

Dort sehen wir die Klasse Artikel, die uns bereits aus Übung 4 in C# bekannt ist. Diese Klasse besitzt zwei virtuelle Methoden GetQualität und SetQualität, die in Subklassen überschrieben werden können. Beim Aufruf von virtuellen Methoden tritt in C++ dabei genauso dynamische Bindung auf wie in C#. Bitte beachten Sie in diesem Zusammenhang auch, dass bei der Definition von virtuellen Methoden der Modifizierer **virtual** nicht mitangegeben werden muss.

Die Methode AktualisiereQualität hingegen ist eine abstrakte Methode und diese werden in C++ nach dem Schema **virtual Rückgabetypr Methodename(Parameterliste) = 0;** gebildet, d.h. nach dem Funktionskopf einer virtuellen Methode wird noch = 0 gesetzt. Abstrakte Methoden werden in C++ deshalb auch als Pure Virtual Functions bezeichnet. Bitte beachten Sie hier, dass eine Klasse, die wenigstens eine abstrakte Methode enthält, automatisch zu einer abstrakten Klasse wird, ohne dass der Klassenkopf noch speziell gekennzeichnet werden müsste. In C# war hier auch bei der Klasse die Angabe von **abstract** notwendig.

Es ist in C++ extrem wichtig, dass Basisklassen einen virtuellen Destruktor besitzen. Wenn Instanzen von Subklassen über eine Referenz der Basisklasse angesprochen werden und beim Deallokalieren der Destruktor aufgerufen wird, kann sonst nicht gewährleistet werden, dass der Destruktor der Subklasse läuft. Wenn in diesem Objekte auf dem Heap freigegeben werden, entstünde dann ein Speicherleck.

Es hat zwar mit dem Thema dieses Abschnitts nicht direkt etwas zu tun, aber ich möchte es trotzdem nochmals erwähnen: die Basisklasse **Artikel** beachtet die Const Correctness. Zu sehen ist das an den **const** Angaben für Methoden, welche die Feldwerte nicht verändern, sowie an den Parametern, die ebenfalls mit **const** gekennzeichnet sind.

```

class Artikel
{
private:
    std::string _name;
    int _qualität;
    int _haltbarkeit;

public:
    std::string GetName() const;
    void SetName(const std::string name);

Virtuelle Methoden
werden bei der Deklaration mit virtual gekennzeichnet
    virtual int GetQualität() const;
    virtual void SetQualität(const int qualitàt);

    int GetHaltbarkeit() const;
    void SetHaltbarkeit(const int haltbarkeit);

    virtual void AktualisiereQualität() = 0; — Für abstrakte Methoden
                                                — hängt man hinter einer virtuellen Methode = 0

    virtual ~Artikel() { } — Abstrakte Klassen sollten immer einen
                           — virtuellen Destruktor enthalten

};

// Weitere Members hier der Einfachheit halber weggelassen

int Artikel::GetQualität() const
{
    return _qualität;
}

void Artikel::SetQualität(const int qualitàt)
{
    if (qualität < 0)
    {
        _qualität = 0;
        return;
    }

    if (qualität > 50)
    {
        _qualität = 50;
        return;
    }

    _qualität = qualitàt;
}

```

Abbildung 315: Beispiel für Abstraktionen in C++

7.9.7.4 Abstraktionen implementieren

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ABOwfWy5uK0>.

Wenn wir von der im vorigen Abschnitt definierte Klasse Artikel ableiten möchten, um AktualisiereQualität zu überschreiben, dann funktioniert auch das in C++ ähnlich wie in C#, allerdings müssen auch hier einige Bedingungen beachtet werden:

Überschriebene Methoden sollte man mit override kennzeichnen

```
class Standardartikel : public Artikel
{
public:
    virtual void AktualisiereQualität() override;
};

void Standardartikel::AktualisiereQualität()
{
    SetHaltbarkeit(GetHaltbarkeit() - 1);

    if (GetHaltbarkeit() > 0)
        SetQualität(GetQualität() - 1);
    else
        SetQualität(GetQualität() - 2);
}
```

Abbildung 316: Abstraktionen implementieren in C++

In Abbildung 316 sehen wir, dass wie schon bei anderen Mechanismen auch die entsprechenden Schlüsselwörter `virtual` und `override` nur bei der Deklaration, aber nicht bei der Definition angegeben werden müssen. Dabei muss `override` nach der Parameterliste stehen, `virtual` steht weiterhin vor dem Rückgabetyp.

Die Angabe von `virtual` und `override` beim Überschreiben von Basisklassenmethoden ist jedoch nicht notwendig – Sie können diese Schlüsselworte auch weglassen. Ich würde Ihnen aber auf jeden Fall empfehlen, `override` zu setzen, da dadurch der Compiler überprüft, ob Sie eine Methode tatsächlich überschreiben: ist bspw. im Bezeichner ein Fehler enthalten, würde eine neue Methode gebildet, anstatt die bestehende zu überschreiben.

7.9.7.5 Mehrfachvererbung mit Klassen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=9fSeDLiUQT4>.

Wie wir bereits erwähnt haben, gibt es in C++ Mehrfachvererbung bei Klassen, die so in C# nicht möglich ist. Grundsätzlich muss man einfach die entsprechenden Klassen kommagetrennt hintereinander angeben, von denen man ableiten möchte. Wenn es dabei zu keinen Konflikten mit Mitgliederbezeichnern kommt, funktioniert das Ganze auch problemlos.

Es kann aber sein, das bspw. zwei Methoden aus verschiedenen Basisklassen denselben Namen besitzen, wie im diesem konstruierten Beispiel:

```

class A { public: void Foo() { } };

class B { public: void Foo() { } };

class C : public A, public B { };
Klasse C leitet von
A und B ab

void main()
{
    C instanceOfC;
    instanceOfC.Foo();          Diese Methode kann nicht
                                aufgerufen werden, da der
                                entsprechende Bezeichner
                                sowohl in A als auch B vorkommt

    instanceOfC.A::Foo();
    instanceOfC.B::Foo();
}

Dieses Problem kann man
umgehen, indem man die
entsprechende Klasse mit
dem :: Operator angibt

```

Abbildung 317: Auflösung von Namensüberschneidungen bei Mehrfachvererbung in C++

Wie wir in Abbildung 317 sehen können, leitet Klasse **C** sowohl von Klasse **A** als auch Klasse **B** ab, welche beide die Methode **Foo** implementieren. Wenn wir versuchen, auf eine Instanz von **C** die Methode **Foo** aufzurufen, dann kann dieser nicht vom Compiler aufgelöst werden. Dies kann man umgehen, indem man mit dem **::** Operator (engl. Scope Resolution Operator) die entsprechende Klasse angibt, von der Sie die Funktion aufrufen möchten.

7.9.7.6 Das Diamond Problem und virtuelle Vererbung in C++ (Level 200)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ZjrTTvkizgk>.

Ein weiteres Problem mit Mehrfachvererbung ist das sog. Diamond Problem. Dieses entsteht, wenn die Basisklassen einer Subklasse wiederum dieselbe Basisklasse besitzen, wie in Abbildung 318 zu sehen ist.

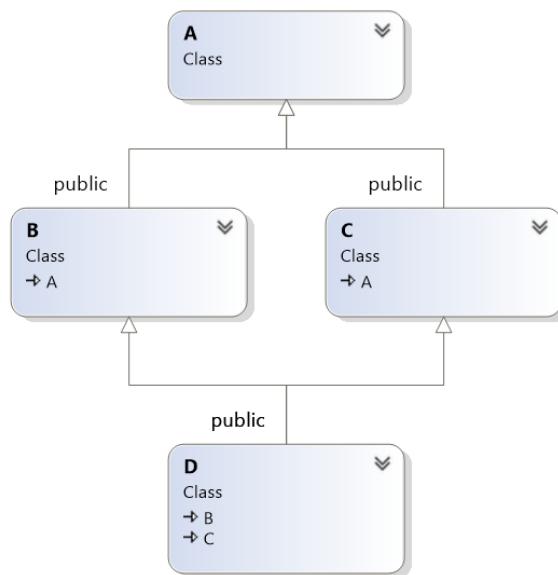


Abbildung 318: Struktur des Diamond Problems in C++

Folgendes Problem kann bei dieser Konstellation auftreten:

```

class A { public: void Foo() { } };

class B : public A { };

class C : public A { };

class D : public B, public C { };

void main()
{
    D instanceOfD;

    instanceOfD.Foo();          ← Diese Aufrufe können nicht
                                aufgelöst werden, da für eine
                                Instanz von D zweimal die
                                Members von A im Codespeicher
                                abgelegt werden

    instanceOfD.A::Foo();       ← Für B bzw. C kann dieser Aufruf
                                problemlos aufgelöst werden

    instanceOfD.B::Foo();
    instanceOfD.C::Foo();
}

```

Abbildung 319: Ein simples Codebeispiel für das Diamond Problem in C++

In Abbildung 319 sehen können wir folgende Punkte feststellen:

- In Klasse A wird die Funktion Foo definiert.
- Klasse B und C leiten jeweils von A ab, definieren selber aber keine Members.
- Klasse D leitet wiederum sowohl von B als auch C ab.
- Wird nun (wie in der main Funktion) eine Instanz von D gebildet, dann kann man auf diesem Objekt nicht einfach die Methode Foo aufrufen, nicht mal über die Angabe A::Foo().
- Erst wenn man über den Scope Resolution Operator :: B oder C angibt, kann Foo aufgerufen werden.

Der Grund dafür ist im Codeteil des Speichers zu suchen: wir müssen uns genauer ansehen, wie die Klasse D dort in diesem Fall strukturiert wird:

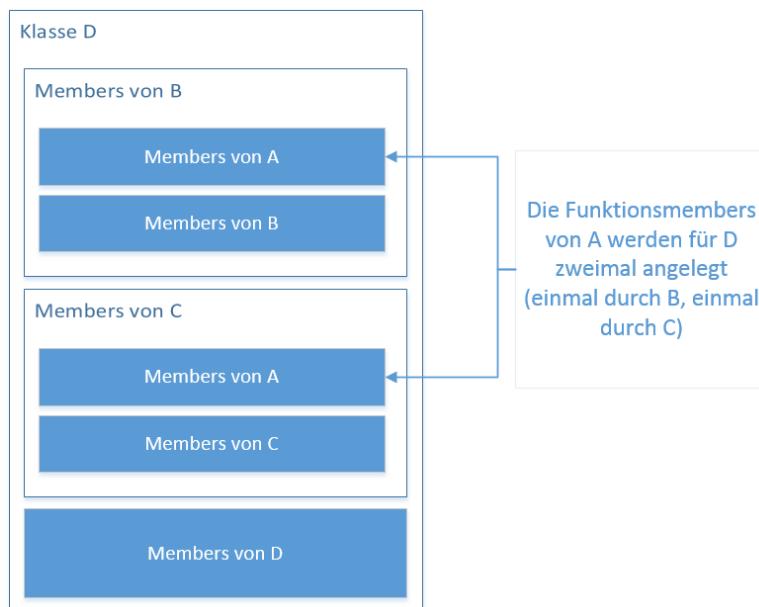


Abbildung 320: Struktur der Klasse D im Codeteil des Speichers beim Diamond Problem

Wenn die Struktur einer Subklasse im Codeteil erstellt wird, dann werden zunächst die Funktionsmitglieder aller Basisklassen abgelegt und danach die jeweiligen Mitglieder der Subklasse. Bei Mehrfachvererbung werden jeweils die Members der direkten Basisklassen hintereinandergelegt – in unserem Beispiel wird für **D** also zunächst die von **B** und im Anschluss die von **C**. In Abbildung 320 können wir dabei auch das Problem erkennen: **B** wie auch **C** selbst haben intern die Struktur, dass jeweils Members von **A** am Anfang stehen – so kann es beim Diamond Problem vorkommen, dass bei der Strukturierung der Klasse **D** die Funktionsmitglieder der Klasse **A** zweimal vorkommen.

Neben der oben gezeigten Möglichkeit zur Auflösung über den Scope Resolution Operator gibt es in C++ einen zweiten Mechanismus, mit dem man das Diamond Problem umgehen kann: virtuelle Vererbung. Durch die Angabe des Modifizierer **virtual** beim Ableiten kann sichergestellt werden, dass bei Diamond Konstellationen die Funktionsmembers von **A** nur einmal im Speicher abgelegt werden:

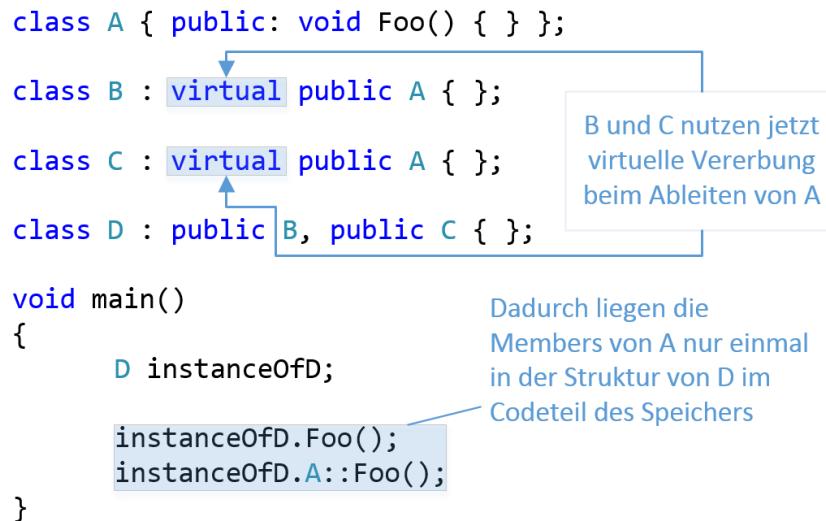


Abbildung 321: Virtuelle Vererbung zur Lösung des Diamond Problems in C++

In Abbildung 321 können wir sehen, dass die Klassen **B** und **C** jetzt virtuell von Klasse **A** ableiten, da bei der Angabe der Basisklasse jeweils noch der Modifizierer **virtual** mit angegeben ist. Das sorgt dafür, dass die Struktur für **D** im Codeteil des Speichers sich wie folgt ändert:

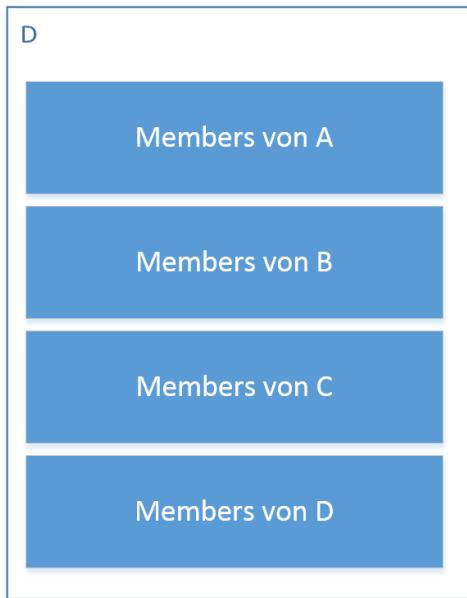


Abbildung 322: Speicherstruktur von D nach Einsatz von virtueller Vererbung

Wie wir in Abbildung 322 erkennen können, hat jetzt **B** und **D** nicht jeweils einen eigenen Satz Members von **A**, sondern müssen sich diese teilen.

Vorsicht bei virtueller Vererbung: Wenn eine Instanz von D erstellt wird, werden die Konstruktoren von B und C vor A aufgerufen. Umgekehrt analog werden diese Klassen auch als letztes bei Destruktoraufrufen angesprochen.

Meine Empfehlung in diesem Fall ist: vermeiden Sie bei Ihrem Klassendesign Diamond Konstrukte und setzen Sie virtuelle Vererbung nur im Notfall ein, um die entsprechenden Funktionsmitglieder von Basisklassen leichter zugänglich für Nutzer zu machen. Ein wesentlich detailliertere Beschreibung zum Diamond Problem finden Sie unter:

http://www.cprogramming.com/tutorial/virtual_inheritance.html

7.9.7.7 Vererbung verhindern

Das Video hierzu finden Sie unter https://www.youtube.com/watch?v=GxJaxEMd_jo.

Genauso wie in C# können Sie in C++ auch das Ableiten von einer Klasse verhindern, indem Sie bei deren Deklaration das Schlüsselwort **final** hinter dem Klassenbezeichner angeben:

```

final nach dem Klassenbezeichner sorgt dafür,
dass von VersiegelteKlasse nicht abgeleitet
werden kann

class VersiegelteKlasse final : public Basisklasse
{
    // Members der Klasse
public:
    void TuEtwas() final override;
};

Auch virtuelle oder abstrakte Methoden können
mit final vor dem weiteren Überschreiben in
Subklassen geschützt werden

```

Abbildung 323: Das Schlüsselwort final in C++

Wie wir in Abbildung 323 sehen können, kann final nicht nur auf Klassen, sondern auch auf virtuelle oder abstrakte Methoden angewandt werden – dadurch können diese in weiteren Subklassen nicht überschrieben werden.

7.10 Templates

Der Template-Mechanismus in C# ist im Wesentlichen gleich mit dem Generic-Mechanismus in C#: sie können damit Vorlagen für Klassen und Funktionen erstellen, die dann für mehrere Typen angewandt werden können, wenn man das Template nach einem entsprechenden Typ auflöst. U.a. werden Templates häufig bei Collections eingesetzt, damit diese für alle möglichen Typen angewandt werden können, wie wir in Abschnitt 5.15 bereits festgestellt haben.

Templates unterscheiden sich allerdings in C++ in einem Punkt wesentlich von Generics in C#: Sie werden komplett zur Compilezeit aufgelöst und haben deshalb auch keinen Performanceeinfluss zur Laufzeit, wenn bspw. die erste `vector<Person*>` erstellt wird. Der Compiler überprüft beim Erstellvorgang, welche Typen für ein Template eingesetzt werden und erstellt jeweils eine eigene konkrete Klasse, die dann zur Laufzeit eingesetzt wird. Die Vorteile von Templates und Generics sind aber die gleichen: typsichere Klassen, die für mehrere Zwecke eingesetzt werden können und dadurch dem DRY-Prinzip folgen.

7.10.1 Template-Funktionen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=-OPmc22SKzo>.

Sehen wir zu Beginn ein Beispiel für Template-Funktionen in C++ an:

Mit dieser Syntax definiert man ein Template, dass in der darauffolgenden Methode eingesetzt werden kann

```
template<class T>
T max(const T& first, const T& second)
{
    return first < second ? second : first;
}

void main()
{
    cout << max(4, 30) << endl;
    cout << max(42.0, 27.7) << endl;
    cout << max(true, false) << endl;

    Rechteck rechteck1(4, 6);
    Rechteck rechteck2(2, 3);
    auto größeresRechteck = max(rechteck1, rechteck2);
}
```

Die Methode max kann jetzt für int, double und bool verwendet werden

Dieser Aufruf gibt einen Compilerfehler, da die Klasse Rechteck den < Operator nicht überschreibt

Abbildung 324: Beispiel für eine Template-Funktion in C++

In Abbildung 324 sehen wir, dass die freie Funktion `max` als Template-Funktion definiert ist: über ihr ist mit dem Ausdruck `template<class T>` der Bezeichner `T` als Template definiert worden und kann nun bei Ihr eingesetzt werden. `T` gilt dabei nur für die jeweils nächste Klasse bzw. Funktion, d.h. wollte man eine weitere Template-Funktion schreiben, müsste man für diese erneut ein Template definieren.

In `main` wird diese Methode dann insgesamt vier Mal aufgerufen. Die ersten drei Aufrufe für `int`, `double` und `bool` funktionieren problemlos, bei der von uns erstellten Klasse `Rechteck` erhalten wir allerdings einen Compilerfehler: Grund dafür ist, dass dieser beim Erstellvorgang versucht, eine Überladung von `max` für `Rechteck` zu erstellen, dabei aber auf das Problem stößt, dass `Rechteck` den `<` Operator nicht überlädt (einem Thema, mit dem wir uns später noch beschäftigen werden).

Hier sehen wir den Compile-Time-Check von Templates: in C# war es nötig, dass wir mit where Klauseln Einschränkungen auf einen Generic machen mussten, damit wir bei diesem eine bestimmte API einsetzen konnten. In C++ ist das nicht notwendig: Sie können für Templates beliebige Funktionsaufrufe machen bzw. wie im obigen Beispiel beliebige Operatoren einsetzen – der Compiler überprüft beim Erstellen, ob der substituierte Typ die entsprechende API zur Verfügung stellt.

7.10.2 Template-Klassen

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=ZSkxEc2Ve98>.

Neben Methoden kann man Templates natürlich wie aus C# auch gewohnt auf Klassen anwenden. Auch hier ist die Syntax etwas anders als in C#, die zugrunde liegenden Prinzipien sind aber gleich. Sehen wir uns dazu gleich ein Beispiel an:

```

template <class T>
class IQueue
{
public:
    virtual void Enqueue(const T item) = 0;
    virtual T Dequeue() = 0;
    int GetCount() const = 0;
    virtual ~IQueue();
};

Hier wird ein Template namens T für die Abstraktion IQueue gebildet und kann dann beliebig auf den Members der Klasse eingesetzt werden

```

Keine Angabe von T nach Klassenbezeichner

Abbildung 325: Beispiel Template-Klasse in C++

Wie auch schon aus dem vorigen Beispiel bekannt wird für die in Abbildung 325 zu sehende Abstraktion `IQueue` ein Template namens `T` erstellt. Das bedeutet wie in C#, dass `T` für beliebige Members der Klasse verwendet werden, z.B. wie im obigen Fall als ParameterTyp für die Methode `Enqueue` bzw. als Rückgabetyp für die Methode `Dequeue`. Wichtig: nach dem Bezeichner für die Klasse, die wir gerade definieren, folgt keine Angabe des Templatebezeichners (wie wir sie aus C# kennen). Deswegen schreiben wir tatsächlich nur `IQueue`, als Nutzer spricht man aber durchaus von `IQueue<T>`.

Interessanter wird es, wenn Sie von Template-Klassen ableiten: hier haben Sie wie in C# die Möglichkeit, dass Sie das Template der Basisklasse auflösen oder weiterführen, wie in folgendem Beispiel angedeutet ist:

```

Das Template der Basisklasse kann in Subklassen fortgeführt werden

template <class T>
class Queue : public IQueue < T >
{
    // Members von Queue der Einfachheit halber weggelassen
};

class IntQueue : public IQueue < int > Es ist aber auch möglich, das Template für Subklassen aufzulösen
{
    // Members von IntQueue der Einfachheit halber weggelassen
};

```

Abbildung 326: Von Template-Klassen ableiten in C++

Wie man in Abbildung 326 sehen kann, wird für die Basisklasse dann das jeweilige Template bzw. der konkrete Typ zur Auflösung angegeben.

Templates haben in C++ einen großen Nachteil: Bei ihnen ist keine Aufteilung in Header und Source Datei möglich. Das liegt daran, dass Templates immer inline sind (sie werden vom Compiler beim Erstellvorgang in einzelne Klassen aufgelöst) – folglich müssen Template-Klassen bei der Deklaration auch gleich definiert werden.

Das ist übrigens auch der Grund, warum Precompiled Headers einen so großen Stellenwert haben: populäre C++ Libraries wie die Standard Library oder Boost C++ enthalten sehr viele Template-Klassen, die in Header Files definiert sind. Bei jedem Erstellvorgang müssten diese ebenfalls erneut kompiliert werden – Precompiled Headers können dieses Problem beheben.

7.11 Operatorenüberladung in C++

Genau wie in C# können wir in C++ Operatoren für unsere Klassen überladen, haben dabei aber viel mehr Freiheiten: wir können alle Operatoren bis auf die folgenden überladen:

- `::` (Scope Resolution Operator)
- `?:` (Conditional Operator)
- `.` (Direct Member Selection Operator)
- `sizeof` (Size-of Operator)
- `.*` (De-Reference Pointer to Class Member Operator)

Ansonsten gibt es keine Einschränkungen – bspw. können auch Operatoren wie der `=` Operator oder der `()` Operator Ziel unserer Überladungen sein.

7.11.1 Operatorenüberladungen deklarieren und definieren

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=N4w4fYyJwHo>.

In Abbildung 324 hatten wir das Problem, dass `Rechteck` nicht mit der Template Funktion `max` genutzt werden konnte, da diese Klassen den `<` Operator nicht überlädt. Genau das ändern wir im folgenden Beispiel:

```
class Rechteck
{
private:
    int _breite;
    int _höhe;

public:
    explicit Rechteck(int seitenlänge);
    Rechteck(int breite, int Höhe);
    int BerechneFläche() const;
    bool operator <(const Rechteck& rechteck) const;
};

bool Rechteck::operator <(const Rechteck& rechteck) const
{
    return BerechneFläche() < rechteck.BerechneFläche();
}
```

Operatorenüberladungen
werden prinzipiell
deklariert / definiert wie in
C#, achten Sie aber auch
Const Correctness

Abbildung 327: Operatorenüberladung in C++

In Abbildung 327 sehen wir, dass Operatorenüberladungen prinzipiell genauso definiert werden wie in C#. Allerdings sollte man bei den Vergleichsoperatoren und Logikoperatoren beachten, dass man die entsprechenden Überladungsmethoden mit `const` kennzeichnet, da sie das zugrunde liegende Objekt nicht verändern. Achten Sie im Allgemeinen bei Methoden auf diese Const Correctness.

Bitte beachten Sie auch, dass C++ Ihnen keine Einschränkungen macht, wie Sie Operatoren zu implementieren haben, so wie es in C# der Fall ist (Abschnitt 5.13.4). Bspw. können Sie durchaus nur

den < Operator überladen, ohne dabei den > Operator zu beachten. Bedenken Sie dabei aber den Kontext des Nutzers Ihrer Klasse, der eventuell nicht nur den < Operator nutzen möchte.

7.11.2 Functors (Level 200)

Das Video hierzu finden Sie unter <https://www.youtube.com/watch?v=-yHZT-45O30>.

Wie wir bereits festgestellt haben, können wir in C++ deutlich mehr Operatoren überladen als in C# - einer dieser Operatoren ist der Funktionsaufrufoperator () . Damit lassen sich sog. Functors implementieren: dies sind Objekte, auf die der Funktionsaufrufoperator angewendet werden kann, was letztendlich aussieht, als würde man eine Methode aufrufen.

Sehen wir uns dazu gleich ein Beispiel an:

```
class AddierFunctor
{
private:
    int _zahl;
public:
    explicit AddierFunctor(int zahl = 0);

    int GetZahl() const;

    void operator () (const int zahl); // Hier wird der () Operator überschrieben, sodass man diesen direkt auf Instanzen dieser Klasse anwenden kann.

};

AddierFunctor::AddierFunctor(int zahl = 0)
    : _zahl(zahl)
{
}

int AddierFunctor::GetZahl() const
{
    return _zahl;
}

void AddierFunctor::operator () (const int zahl) // Hier wird der () Operator überschrieben, sodass man diesen direkt auf Instanzen dieser Klasse anwenden kann.
{
    _zahl += zahl;
}
```

Abbildung 328: Beispiel für eine Functor-Implementierung

In Abbildung 328 sehen wir die Klasse AddierFunctor, welche den Operator () überschreibt. In dieser wird ausschließlich die übergebene Zahl zum Feld _zahl hinzugezählt. Beachten Sie dabei, dass im Funktionskopf das erste paar Klammern den Operator identifiziert, das zweite Paar Klammern stellt die tatsächliche Parameterliste dar.

Objekte dieser Klasse können dann wie folgt genutzt werden:

```

void main()
{
    AddierFunctor functor;

    functor(2);
    functor(5);
    functor(10); Dieser Aufruf sieht aus wie ein  
Methodenaufruf, ist aber  
tatsächlich nur der Aufruf des  
() auf dem Objekt

    cout << functor.GetZahl() << endl;
}

```

Abbildung 329: Functors einsetzen in C++

In Abbildung 329 sehen wir einen Instanz der Klasse `AddierFunctor`, auf die direkt der () Operator angewendet wird. Dies sieht aus wie ein Funktionsaufruf, endet aber letztendlich in der Methode, die in der Klasse `AddierFunctor` den () Operator überlädt.

Sie können Functors bspw. für das Strategy-Pattern einsetzen. Anstatt gegen eine Abstrakte Strategy auf Clientseite zu programmieren können Sie sich über ein Template einen Functor übergeben lassen, den Sie entsprechend aufrufen. Dadurch können Sie verschiedene Functors implementieren, von denen Sie je nach Situation einen an den Client übergeben – ohne dass diese Functors eine gemeinsame Abstraktion haben müssen.

Dies ist ein Hinweis darauf, dass in C++ polymorphes Verhalten auch über Templates erzielt werden kann.

7.12 Wichtige Klassen der Standard Library

Wir haben bereits einige Klassen der Standard Library genutzt wie `string` oder `vector<T>`. In den kommenden Abschnitten möchte ich Ihnen noch kurz einige weitere interessante Header vorstellen und auf Smart Pointers eingehen.

7.12.1 Wichtige Headerdateien der Standard Library

Die folgenden Headers werden im Programmieralltag häufig gebraucht:

- Smart Pointers: `<memory>`
- Datum und Zeitrechnung: `<chrono>` und `<ctime>`
- Mathematik, komplexe Zahlen und Matrizenberechnungen: `<cmath>`, `<complex>` und `<numeric>`
- Zufallszahlen: `<random>`
- Regular Expressions: `<regex>`

Smart Pointers werden wir uns im kommenden Abschnitt noch genauer ansehen.

7.12.2 Smart Pointers

Im Header `<memory>` findet man u.a. zwei Klassen, die zu Smart Pointers zählen:

- `shared_ptr<T>` stellt einen Pointer auf ein Objekt dar, der weitergereicht werden kann via Call-By-Value. Dabei zählt er die Anzahl an Referenzen, die gerade auf den gekapselten Speicherbereich blicken. Sobald einer dieser Smart Pointer Kopien Out-Of-Scope gehen, wird die Referenzanzahl verringert und wenn diese schließlich 0 erreicht, wird der dazugehörige Speicherbereich automatisch deallokiert.

- `unique_ptr<T>` repräsentiert einen einzigartigen Pointer auf ein Objekt, d.h. dieser Pointer kann nicht via Call-By-Value weitergegeben werden wie der `shared_ptr<T>`. Sobald dieser Pointer Out-Of-Scope geht, wird auch der dazugehörige Speicherbereich freigegeben.

Da man `unique_ptr<T>` Instanzen nicht weitergeben kann (via Zuweisung), sollten Sie in den meisten Situationen `shared_ptr<T>` bevorzugen, insbesondere wenn Sie Dependency Injection in Ihrem C++ Code nutzen möchten.

Sehen wir uns zu Shared Pointers ein Beispiel an:

```

class Dependency
{
public:
    void DoMore() { }
};

class DependentClass
{
private:
    shared_ptr<Dependency> _dependency;           DependentClass nutzt
                                                    einen shared_ptr auf
                                                    die Abhängigkeit

public:
    DependentClass(shared_ptr<Dependency> dependency)   /\
        : _dependency(dependency)
    {

    }

    void DoSomething()
    {
        // Here a calculation with dependency is performed
        _dependency->DoMore();                         shared_ptr können wie normale Pointer
                                                        dereferenziert werden
    }
};

void TuEtwas()  Hier werden zwei Objekte mit new erstellt und via Shared Pointers verwaltet.
{
    shared_ptr<Dependency> dependency(new Dependency());
    shared_ptr<DependentClass> dependentClass(new DependentClass(dependency));
}

dependentClass->DoSomething();
}                                Wenn Shared Pointer Out-Of-Scope gehen, wird ihr Referenzzähler automatisch dekrementiert
void main()
{
    TuEtwas();
}

```

Abbildung 330: Shared Pointers einsetzen

In Abbildung 330 sehen Sie folgende Dinge:

- In `main` wird ausschließlich die Methode `TuEtwas` aufgerufen.
- In letzterer werden zwei Objekte auf dem Heap instanziert, einmal von der Klasse `Dependency`, einmal von der Klasse `DependentClass`.
- Die Adressen dieser Objekte werden nicht in Pointern, sondern in `shared_ptr<T>` festgehalten, die aber auch auf dem Heap liegen.

- Bei der Instanziierung des `DependentClass` Objekts wird der Shared Pointer `dependency` an den Konstruktor übergeben. Bei dieser Übergabe wird der Pointer kopiert (bitte nicht verwechseln mit dem `Dependency`-Objekt, dieses wird nicht kopiert) und dabei wird intern der sog. Referenzzähler für den Shared Pointer um ein erhöht auf den Wert 2.
- Danach wird `DependentClass` : : `DoSomething` aufgerufen, was nur illustrieren soll, das innerhalb dieser Methode der Aufruf an `Dependency` : : `DoMore` über die Abhängigkeit weitergereicht werden kann.
- Danach ist die `TuEtwas` Funktion beendet und wird damit vom Stack deallokiert. Das hat folgende Auswirkung:
 - Die Variablen `dependency` und `dependentClass` werden automatisch deallokiert.
 - Dies sorgt dafür, dass der Referenzzähler für diese Shared Pointer verringert wird. Für `Dependency` ist der Referenzzähler damit bei 1, für `DependentClass` bei 0.
 - Dadurch, dass der Referenzzähler für letzteres Objekt bei 0 liegt, wird das Objekt automatisch vom Heap deallokiert.
 - Innerhalb dieses Objekts liegt der letzte Smart Pointer auf das `Dependency`-Objekt, sodass auch dieser jetzt den Wert 0 erreicht und automatisch deallokiert wird.

Wie wir sehen können, zieht sich das automatische Deallokalieren eines Objekts dazu führen, dass auch ein anderes Objekt deallokiert wird, wenn es über Shared Pointers referenziert wurde. Und genau das ist der große Vorteil von Ihnen.

7.13 Große C++ Softwarelösungen aufbauen

Nachdem wir uns in den letzten Abschnitten mit C++ spezifischen Dingen beschäftigt haben, möchte ich noch kurz darauf eingehen, wie Sie größere Solutions in C++ aufbauen können.

Der wichtige Punkt ist, dass hier kein großartiger Unterschied zu C# besteht: sie können die in Kapitel 6 angesprochenen Themen genauso auf ihren C++ Code anwenden.

- Nutzen Sie verschiedene Klassen, um verschiedene Teilprobleme in Ihrem Softwareprojekt zu lösen. Gehen Sie dabei nach dem Single Responsibility Principle vor, um Ihre Klassen feingranular zu strukturieren und dadurch die Wiederverwendbarkeitschance zu erhöhen.
- Achten Sie auf Erweiterbarkeit in Ihren Klassen: programmieren Sie gegen Abstraktionen und ersetzen Sie bspw. große `if else` Konstrukte durch polymorphe Aufrufe. Dadurch können Sie Ihre Software flexibel erweitern und leichter testen.
- Nutzen Sie etablierte Design Patterns für bekannte Probleme in der Programmierung, bspw. Commands, Factories, Repositories oder Mementos.
- Über einen Composition Root können Sie alle notwendigen Objekte für Ihren Programmablauf instanziiieren und via Dependency Injection die Abhängigkeiten unter diesen auflösen. Achten Sie darauf, dass Sie in C++ selbst dafür zuständig sind, ob Werte via Call-By-Value oder Call-By-Reference übergeben werden.
- Events werden zwar nicht direkt von C++ unterstützt, können aber über Abstraktionen oder Funktoren nachgebaut werden. Weiterhin können Sie das Observer Pattern einsetzen.
- Über Smart Pointers können Sie manuelles Speichermanagement umgehen. Setzen Sie `shared_ptr<T>` ein, wenn Sie einen Zeiger auf ein Objekt weitergeben möchten, oder `unique_ptr<T>`, wenn nur eine einzige Referenz auf ein Objekt existieren darf. Wenn Sie nach dem DIP programmieren, werden Sie aber hauptsächlich Shared Pointers nutzen.
- Nutzen Sie automatisierte Tests, um schnell Feedback darüber zu erhalten, ob Ihr Code das macht, was er soll.