

# **.NET Technologien**

HS2021

**Marco Agostini**



Computer Science  
University of Applied Sciences of Eastern Switzerland  
September 2021

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	.NET Framework	5
1.2	CLR: Common Language Runtime	5
1.3	CTS: Common Type System	5
1.3.1	Reference- und Value Typen	5
1.3.2	Boxing und Unboxing	6
1.4	CLS: Common Language Specification	6
1.5	Microsoft Intermediate Language (MSIL)	6
1.6	Assemblies	7
1.6.1	Modules & Metadata	7
<b>2</b>	<b>Projekte und Referenzen in Visual Studio</b>	<b>8</b>
2.1	Projektdateien	8
2.2	Referenzen	8
2.3	Packages & NuGet	8
<b>3</b>	<b>C# Grundlagen</b>	<b>9</b>
3.1	Naming Guidelines	9
3.2	Sichtbarkeitsattribute	9
3.3	Primitivtypen	10
3.4	Statements	10
3.5	Namespaces	11
3.6	Main Methode	11
3.6.1	Top-Level Statements	11
3.7	Enumerationstypen	11
3.8	Object	11
3.9	Arrays	12
3.10	Strings	12
3.11	Symbole	12
<b>4</b>	<b>Klassen und Structs</b>	<b>13</b>
4.1	Klassen	13
4.1.1	Partielle Klassen	13
4.1.2	Abstrakte Klassen	13
4.2	Structs	14
4.3	Properties	14
4.4	Auto-Implemented Properties	14
4.5	Initialisierte Properties	14
4.6	Indexer	15
4.7	Konstruktoren	15
4.7.1	Statische Konstruktoren	16
4.7.2	Dekonstruktoren	16
4.7.3	Initialisierungs-Reihenfolge	16
4.8	Operatoren - Überladung	16
<b>5</b>	<b>Methoden</b>	<b>17</b>
5.1	Statische Methoden	17
5.2	Value Parameter	17
5.3	Reference Parameter	17
5.4	Out-Parameter	17
5.5	In Parameter	17
5.6	Params-Array	17
5.7	Optionale Parameter (Default Values)	17
5.8	Named Parameter	18
5.9	Überladung	18
5.10	Lambda Expressions	18
5.10.1	Type Inference	18

<b>6</b>	<b>Vererbung</b>	<b>19</b>
6.1	Typprüfungen . . . . .	19
6.2	Type Casts . . . . .	19
6.3	Methoden . . . . .	19
6.4	Interfaces . . . . .	20
6.4.1	Interfaces Implementieren . . . . .	20
<b>7</b>	<b>Delegates &amp; Events</b>	<b>21</b>
7.1	Delegates . . . . .	21
7.2	Multicast Delegates . . . . .	21
7.3	Events . . . . .	21
7.4	Anonyme Methoden . . . . .	21
<b>8</b>	<b>Generics</b>	<b>22</b>
8.1	Type Constraints . . . . .	22
8.2	Vererbung . . . . .	22
8.3	Nullable Types . . . . .	22
<b>9</b>	<b>Iteratoren</b>	<b>23</b>
9.1	Interfaces . . . . .	23
9.2	Zugriff . . . . .	23
9.3	Iterator-Methoden . . . . .	23
9.3.1	yield return . . . . .	23
9.4	Spezifische Iteratoren . . . . .	23
9.5	Extension Methode . . . . .	23
9.6	Deferred Evaluation . . . . .	24
<b>10</b>	<b>Exceptions</b>	<b>25</b>
10.1	Klasse System.Exception . . . . .	25
10.2	Rethrowing . . . . .	25
10.3	Exception-Klassen . . . . .	26
10.4	Catch-Klausel . . . . .	26
10.5	Exception Filters . . . . .	26
<b>11</b>	<b>Language Integrated Query (LINQ)</b>	<b>27</b>
11.1	LINX Extension Methods . . . . .	27
11.2	Expression-Bodies Members . . . . .	28
11.3	Query Expressions Syntax . . . . .	28
11.4	Gruppierung . . . . .	28
11.5	Inner Joins . . . . .	29
11.6	Group Joins . . . . .	29
11.7	Left Outer Joins . . . . .	29
11.8	Select Many . . . . .	30
<b>12</b>	<b>Direct Initialization</b>	<b>31</b>
12.1	Object Initializers . . . . .	31
12.2	Collection Initializers . . . . .	31
12.3	Anonymous Types (Let) . . . . .	31
<b>13</b>	<b>Tasks</b>	<b>32</b>
<b>14</b>	<b>Entity Framework Core</b>	<b>33</b>
14.1	OR-Mapping . . . . .	33
14.1.1	Mapping-Ansätze . . . . .	33
14.1.2	Model . . . . .	34
14.1.3	Relationale DB (SQL Server) . . . . .	35
14.2	DB Context . . . . .	37
14.2.1	Change Tracking . . . . .	37
14.2.2	Ladestrategien (Lazy, Explizit, Eager Loading) . . . . .	38
14.3	LINQ to Entities . . . . .	39
14.3.1	CUD Operationen (Create, Update, Delete) . . . . .	39
14.3.2	CUD von Assoziationen . . . . .	40

14.4	Optimistic Concurrency . . . . .	41
14.5	Vererbung . . . . .	42
14.6	Database Migrations . . . . .	43
<b>15</b>	<b>Google Remote Procedure Call (gRPC)</b>	<b>44</b>
15.1	Architektur . . . . .	44
15.2	Protocol Buffers . . . . .	44
15.3	gRPC # API . . . . .	46
15.4	Beispiel Customer Service . . . . .	46
15.5	Streams . . . . .	47
15.6	Exception Handling . . . . .	48
<b>16</b>	<b>Reflection</b>	<b>49</b>
16.1	Type-Discovery . . . . .	49
16.2	Member auslesen . . . . .	49
16.3	Field Information . . . . .	50
16.4	Property Info . . . . .	50
16.5	Method Info . . . . .	50
16.6	Constructor Info . . . . .	50
<b>17</b>	<b>Attributes</b>	<b>50</b>
17.1	Anwendungsfälle . . . . .	50
17.2	Custom Attributes . . . . .	50

# 1 Introduction

## 1.1 .NET Framework

- Aktuelle werden über 30 Sprachen unterstützt
- Der Source Code wird in die Intermediate Language IL (ähnlich wie Assembler, vergleichbar mit Java Bytecode kopiert)
- Alle Sprachen nutzen das selbe Objektmodell und Bibliotheken
  - gemeinsamer IL-Zwischencode
  - gemeinsames Typensystem (CTS)
  - gemeinsame Runtime (CLR)
  - gemeinsame Klassenbibliotheken
  - Das CLS definiert Einschränkungen an interoperablen Schnittstellen
- Der Debugger unterstützt alle Sprachen (auch Cross-Language Debugging möglich)

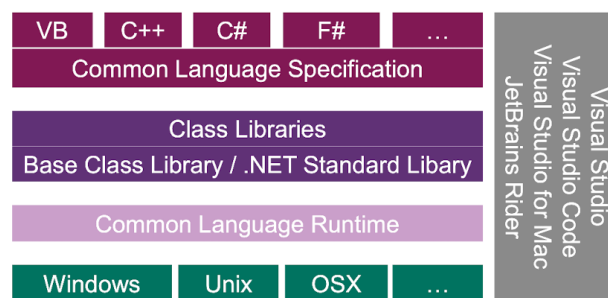


Figure 1: .NET Framework Architektur

## 1.2 CLR: Common Language Runtime

Die CLR ist die Laufzeitumgebung für .NET-Code und umfasst Funktionen wie: Just in Time Compilation etc. Man versteht unter dem CLR ein sprachunabhängiges, abstrahiertes Betriebssystem. Verantwortlichkeiten: Memory Management, Class Loading, Garbage Collection, Exceptions, Type Checking, Code Verification des IL-Codes, Debugging und Threading. Die CLR ist mit der Java VM vergleichbar.

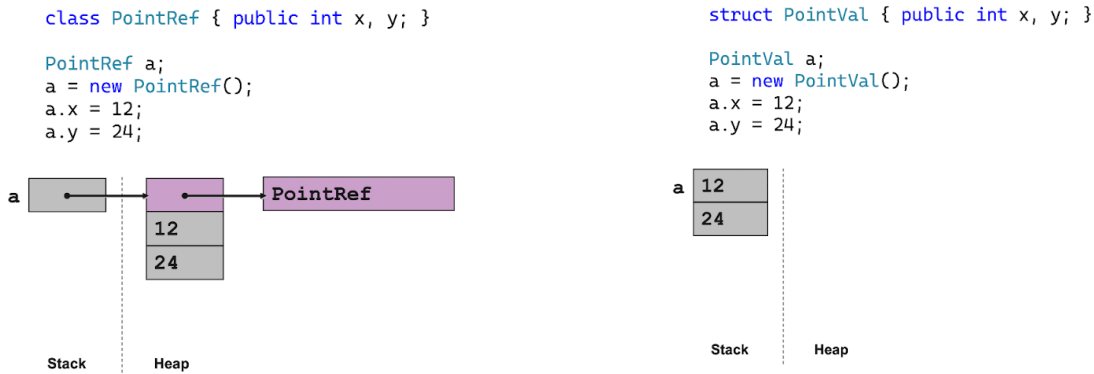
## 1.3 CTS: Common Type System

Das allgemeine Typensystem legt fest, wie Typen in der Common Language Runtime deklariert, verwendet und verwaltet werden. Außerdem ist das System ein wichtiger Bestandteil der Laufzeitunterstützung für die sprachübergreifende Integration. Alle Typen in .NET sind entweder Werttypen oder Verweistypen. Alle Typen sind von System.Object abgeleitet. CTS ist teil der CLR.

### 1.3.1 Reference- und Value Typen

In .NET unterscheidet man zwischen Referenz- (Klassen) und Value Typen (Structs, Enum und primitive Datentypen).

- Reference Types
  - Werden auf dem Heap gespeichert
  - Variable enthält Referenz
  - Automatisch Garbage Collection
  - Konstruktor erzeugt und initialisiert Objekt
  - Objekt hat eine Referenz auf seine Typenbeschreibung
- Value Types
  - Zur Speicherung Rohrer Werte auf dem Stack
  - Konstruktor macht nur eine Initialisierung
  - Boxing: automatische Umwandlung von Reference Type
  - sealed



### 1.3.2 Boxing und Unboxing

Boxing ermöglicht, dass Value- und Reference Types polymorph behandelt werden können. **Boxing:** Kopiert Value Type in einen Reference Type. Value Type wird implizit Konvertiert (UpCast). **Unboxing:** Kopiert Reference Type in eine Value Type. Explizite Konversion nötig!

## 1.4 CLS: Common Language Specification

Um vollständige Interoperabilitätsszenarien zu aktivieren, müssen alle Objekte, die im Code erstellt werden, sich auf eine gewisse Gemeinsamkeit in den Sprachen verlassen, von denen sie verwendet werden (die ihre Aufrufer sind). Da es zahlreiche verschiedene Sprachen gibt, legt .NET diese Gemeinsamkeiten in der sogenannten Common Language Specification (CLS) fest. Die CLS definiert einen Satz von Funktionen, die viele gängige Anwendungen benötigen. Darüber hinaus bietet sie für jede Sprache, die in .NET implementiert wird, Anweisungen, was diese unterstützen muss. Die CLS ist eine Teilmenge des CTS.

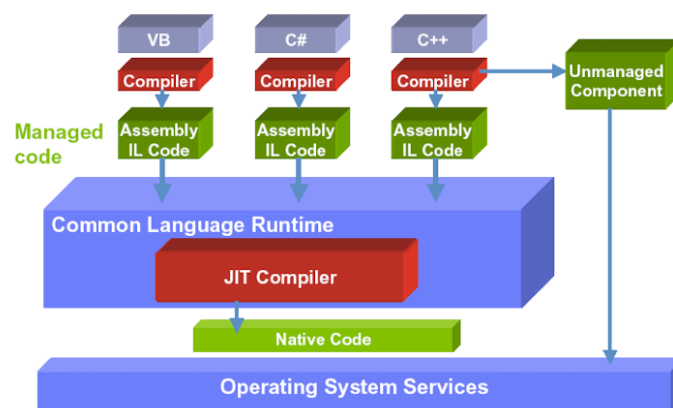


Figure 2: CLR: Common Language Runtime Architecture

## 1.5 Microsoft Intermediate Language (MSIL)

Die MSIL ist eine vorkompilierte Zwischensprache, welche: *Prozessor-unabhängig, Assembler-ähnlich und Sprach-unabhängig ist.*

1. Sprachspezifischer Kompilier kompiliert nach MSIL

2. Just In Time Compiler (JIT) Compiler aus dem CLR kompiliert in nativen plattformabhängigen Code.
- Vorteile
    - Portabilität (Nicht-Intel-Prozessoren, Unix etc.)
    - Typensicherheit (Beim laden des Code können Typen- Sicherheit und weiter Security-Checks durchgeführt werden.
  - Nachteile
    - Laufzeiteffizient (Kann durch den JIT-Compiler wettgemacht werden)

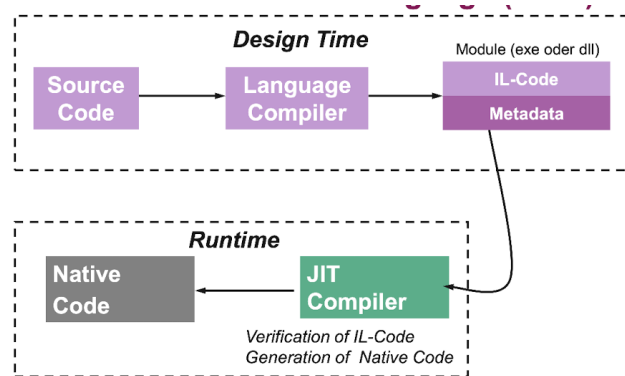


Figure 3: MSIL Kompilierung

## 1.6 Assemblies

Die Kompilation erzeugt Assemblies, welche mit einem JAR-File verglichen werden können. Assemblys sind ausführbare Dateien (.exe) oder Dynamic Link Library-Dateien (.dll) und bilden die Bausteine von .NET-Anwendungen. Sie stellen der Common Language Runtime die Informationen zur Verfügung, die sie zum Erkennen der Typenimplementierungen benötigt.

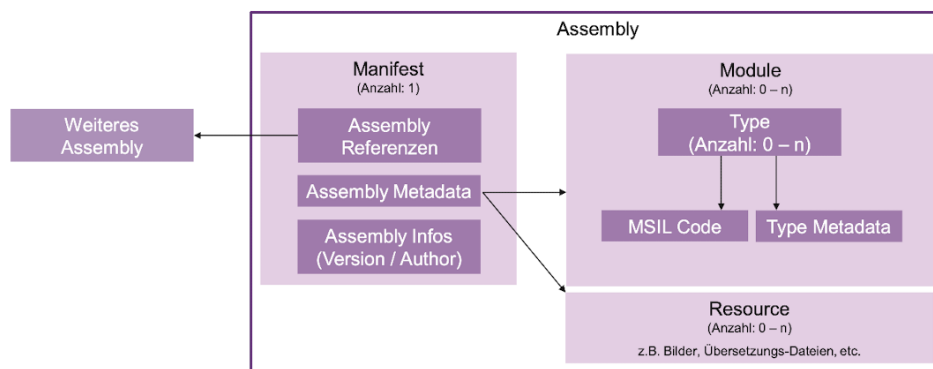


Figure 4: Assembly Überblick

### 1.6.1 Modules & Metadata

Eine Kompilation erzeugt ein Modul mit Code/MSIL und den Metadaten. Die Metadaten beschreiben alles Aspekte ausser die Programmlogik des Codes.

## 2 Projekte und Referenzen in Visual Studio

Projekt-Dateien werden als XML-Datei verwaltet. Neu ist das \*.csproj seit dem Jahre 2017. Die Struktur ist identisch, wird aber anders interpretiert. Es gibt zwei Build Engines: Microsoft Build Engine (MSBuild) und .NET CORE CLI (dotnet build) - ist indirekt auch wieder MSBuild.

### 2.1 Projektdateien

Die neuen Projektdateien (\*.csproj) sind viel schlanker als die alten. Enthält neue Definition der vom Compiler unterstützten XML-Elemente und Attribute.

### 2.2 Referenzen

Im Projektfile wird zwischen verschiedenen Referenzen unterschieden.

- Vorkompiliertes Assembly
  - Im File System, Debugging nicht verfügbar, Navigation auf Metadaten-Ebene
- NuGet Package
  - Externe Dependency, Debugging nicht verfügbar, Navigation auf Metadaten-Ebene
- Visual Studio Projekt
  - In gleicher Solution vorhanden, Debugging und Navigation verfügbar
- .NET Core oder .NET Standard DSK
  - Zwingend, Normalerweise "Microsoft.NETCore.App", Bei .NET Standard "NETStandard.Library"

### 2.3 Packages & NuGet

.NET wird neu in kleineren NuGet Packages ausgeliefert und ist kein monolithisches Framework mehr. Vorteile: unterschiedliche Release-Zyklen, Erhöhte Kompatibilität, kleinere Deploymenteinheiten.



## 3 C# Grundlagen

Ähnlichkeiten zu Java: Objektorientierung, Interfaces, Exceptions, Threads, Namespaces (wie Packages), Strenge Typenprüfung, Garbage Collection, Reflection und dynamisches Laden von Code. Neues: Referenzparameter, Objekte am Stack, Blockmatrizen, Enumerationstypen, Uniformes Typensystem, goto, Systemnahes Programmieren, Versionierung.

### 3.1 Naming Guidelines

Element	Casing	Beispiel
Namespace Klasse / Struct Interface Enum Delegates	PascalCase, Substantive	System.Collections.Generic BackColor IComparable Color Action / Func
Methoden	PascalCase Aktiv-Verben / Substantive	GetDataRow UpdateOrder
Felder Lokale Variablen Parameter	CamelCase	name orderId
Properties Events	PascalCase	OrderId MouseClicked

Figure 5: Naming Guidelines

### 3.2 Sichtbarkeitsattribute

Attribut	Beschreibung
public	Überall sichtbar
private	Innerhalb des jeweiligen Typen sichtbar
protected	Innerhalb des jeweiligen Typen oder abgeleiteter Klasse sichtbar
internal	Innerhalb des jeweiligen Assemblies sichtbar
protected internal	Innerhalb des jeweiligen Typen oder abgeleiteter Klasse sichtbar oder Innerhalb des jeweiligen Assemblies sichtbar
private protected* (seit C# 7.2)	Innerhalb des jeweiligen Typen oder abgeleiteter Klasse sichtbar, wenn diese im gleichen Assembly ist

Figure 6: Sichtbarkeitsattribute

Typ	Standard	Zulässig (Top-Level*)	Standard für Members	Zulässig für Members
class	internal	public / internal	private	public protected internal private protected internal private protected
struct	internal	public / internal	private	public internal private
enum	internal	public / internal	public	—
interface	internal	public / internal	public	—
delegate	internal	public / internal	—	—

\* Gilt nicht für «nested types»

Figure 7: Standard Sichtbarkeiten von Typen

### 3.3 Primitivtypen

#### Ganzzahlen

Numerisch Werte können für eine bessere Lesbarkeit mit dem "\_" geschrieben werden. Bestimmung des Typen: Ohne Suffix=kleinster Type aus int, uint, long, ulong. Suffix u | U =kleinster Type aus uint, ulong. Suffix l | L = kleinster Type aus long, ulong.

#### Fliesskommazahlen

Bestimmung des Typen: Ohne Suffix=double. Suffix f | F=float. Suffix d | D=double. Suffix m | M=decimal.

#### Typenkompatibilität

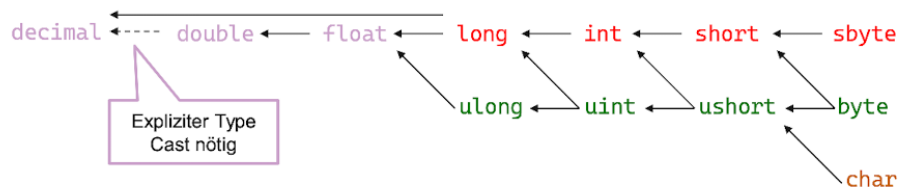


Figure 8: Typenkompatibilität

### 3.4 Statements

#### If Statements

```
1 int value = 5;
2 if (0 >= value && value < 9) {
3     /* ... */
4 } else if (value > 10) {
5     /* ... */
6 } else {
7     Console.WriteLine("Invalid: " +
8         value);
9 }
```

#### Switch Statement

```
1 string ctry = "Germany"; string
   language;
2 switch (ctry) {
3     case "England":
4     case "USA":
5         language = "English"; break;
6     case "Germany":
7     case "Austria":
8     case "Switzerland":
9         language = "German"; break;
10    case null:
11        Console.WriteLine("Country null");
12        break;
13    default:
14        Console.WriteLine("Unknown" + ctry
15            ); break;
16 }
```

#### Loops

```
1 // Kopfgesteuert
2 int i = 1;
3 int sum = 0;
4 while (i <= 3)
5 {
6     sum += i;
7     i++;
8 }
9 // Fussgesteuert
10 int i = 1;
11 int sum = 0;
12 do
13 {
14     sum += i;
15     i++;
16 } while (i <= 3)
17 // Foreach
18 int[] a = { 3, 17, 4, 8, 2, 29 };
19 foreach (int x in a)
20 {
21     sum += x;
22 }
```

#### Jumps

```
1 for (int i = 0; i < 10; i++) {
2     if (i == 1) { continue; }
3     if (i == 3) { goto myLabel; }
4     if (i == 5) { break; }
5     Console.WriteLine(i);
6 myLabel: ;
7 }
```

### 3.5 Namespaces

Entspricht in Java dem Package. Strukturiert den Quellcode und ist hierarchisch aufgebaut. Beinhaltet: Andere Namespaces, Klassen, Interfaces, Structs, Enums, Delegates.

### 3.6 Main Methode

Einstiegspunkt (entry point) eines Programmes. Zwingend nötig für Executables (Console Application, Windows Application etc.) Klassischerweise genau 1x mal erlaubt. Wenn mehrere Main-Methoden vorhanden sind, muss dies in der Projektdatei angegeben werden.

#### 3.6.1 Top-Level Statements

Erlaubt das Weglassen der Main-Methode als entry point. Vereinfacht z.B. Beispiel-Applikationen.

**Regeln:**

- Nur 1x pro Assembly erlaubt
- Argumente heißen fix args
- Exit Codes erlaubt
- VOR dem top-level statements können usings definiert werden.
- NACH dem top-level statements können Typen definiert werden.

```

1 using System;
2 for (int i = 0; i < args.Length; i++) {
3     ConsoleWriter.Write(args, i);
4 }
5 Class ConsoleWriter {
6     public static void Write(string[] args, in t) // Top Level Statement
7     {
8         Console.WriteLine("Arg {i} 0 {args[i]}");
9     }
10 }

```

### 3.7 Enumerationstypen

Ist eine Liste vordefinierter Konstanten inklusive Wert (Default-Typ Int32). Der erste Wert ist per default eine 0 und die andere folgen n+1.

```

1 enum Days { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
2 Days today = Days.Monday;
3 if (today == Days.Monday) { /* ... */ }
4 enum Days { Monday = 10, Tuesday, Wednesday, Thursday, Friday, Saturday };
5 enum Days:byte { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
6     Saturday };
7 // Parsing
8 bool success3 = Enum.TryParse("Monday", out Days day3);

```

### 3.8 Object

System.Object ist die Basisklasse für alle Typen und Objekte. Das keyword object is ein Alias für System.Object.

```

1 o1 = "Test"; o1 = 123; o1 = new Rectangle();
2 void Push(object x) { /* ... */ }
3 Push("Test"); Push(123);
4 Push(new Rectangle());
5 Push(new int[3]);

```

### 3.9 Arrays

Einfachste Datenstruktur für Listen, Ausprägungen: Eindimensional, Mehrdimensional (Rechteckig), Mehrdimensional Jagged (Ausgefranst). Länge aller Dimensionen müssen bei der Instanziierung bekannt sein. Alle Werte sind nach der Instanziierung initialisiert (false, 0, null etc.). Arrays sind eine Klasse und leben immer auf dem Heap!

```

1 // Eindimensionale Arrays
2 int[] array1 = new int[5];           // Deklaration (Value Type)
3 int[] array2 = new int[] { 1, 3, 5, 7, 9 }; // Deklaration & Wertdefinition
4 int[] array3 = int[] { 1, 2, 3, 4, 5, 6 }; // Vereinfachte Syntax ohne new
5 int[] array4 = { 1, 2, 3, 4, 5, 6 }; // Vereinfachte Syntax ohne new Typ
6 object[] array5 = new object[5];     // Deklaration (Reference Type)
7 // Mehrdimensionale Arrays
8 int[,] multiDim1 = new int[2, 3];    // Deklaration
9 int[,] multiDim2 = { { 1, 2, 3 }, { 4, 5, 6 } }; // Deklaration &
    Wertdefinition
10 // Mehrdimensionale Arrays Jagged
11 int[][] jaggedArray = new int[6][]; // Deklaration
12 jaggedArray[0] = new int[] { 1, 2, 3, 4 }; // Wertdefinition

```

### 3.10 Strings

Datenstruktur für Zeichenketten. Eigenschaften: Reference Type, string ist Alias für "System.String", nicht modifizierbar, Verkettung möglich, Wertevergleich mit == oder Equals Methode möglich. Nicht Null/0 terminiert, Indexierung möglich, Länge wird durch das Property "Length" ermittelt.

#### Interning

Strings werden intern wiederverwendet und ein gleicher String nicht unbedingt kopiert. Erst "string.Copy(...)" erzeugt eine echte Kopie.

```

1 string s1 = "Hello ";
2 string s2 = "World";
3 string s3 = s1 + s2;
4 // Unveraenderbarkeit wird durch den Compiler sichergestellt
5 string s1 = "Hello ";
6 string s2 = "World";
7 string s3 = s1 + s2; // string s3 = System.String.Concat(s1, s2);
8 s3 += "!"; // s3 = System.String.Concat(s3, "!");
9 // String Interpolation
10 string s3 = $"{DateTime.Now}: {"Hello"}";
11 string s4 = $"{DateTime.Now}: {(DateTime.Now.Hour < 18 ? "Hello" : "Good
    Evening")}";

```

### 3.11 Symbole

Sind Unicode codiert und Case-Sensitive. @ für Verwendung von Schlüsselwörtern als Identifier.

## 4 Klassen und Structs

### 4.1 Klassen

Sind Referenz Typen und werden auf dem Heap angelegt.

#### Vererbung

Ableiten von Basisklasse möglich, Verwenden als Basisklasse möglich, Implementieren von Interfaces möglich.

#### Felder

Eine Konstante einer Klasse muss zur Compilezeit berechenbar sein. Ein "readonly" Feld darf zur Laufzeit nicht mehr geändert werden.

#### Nested Types

Sind für spezifische Hilfsklassen gedacht. Die äussere Klasse hat Zugriff auf innere Klasse (Nur Public). Die innere Klasse hat Zugriff auf äussere Klasse (auch private!).

#### Statische Klassen

Regeln: Nur statische Members erlaubt, kann nicht instanziiert werden, sind "sealed".

#### Statische Usings

Verkürzen den Quellcode bei Verwendung von statischen Klassen. Regeln: Nur statische Klassen sowie Enums erlaubt. Importiert alle: statischen Members, Statischen Nested Members. Namenskonflikte: Normale Namensauflösungsregeln wie bei überladenen Methoden, bei identischen Signaturen muss Klassenname vorangestellt werden..

```
1 class Stack {
2     const long = 12; // Konstante muss zur Compilezeit berechenbar sein
3     int[] values;
4     int top = 0;
5     public Stack(int size) { /* ... */ }
6     public void Push(int x) { /* ... */ }
7     public int Pop() { /* ... */ }
8 }
9 Stack s = new Stack(10);
```

#### 4.1.1 Partielle Klassen

Das partial Keyword erlaubt die Definition in mehreren Files. Es sind auch partielle Methoden möglich. Regeln: "partial" muss bei allen Teilen angemerkt sein, Alle Teile müssen das gleiche Sichtbarkeitsattribut haben. Es gibt auch partielle Methoden.

```
1 // File1.cs
2 partial class MyClass {
3     public void Test1() { } }
4 // File2.cs
5 partial class MyClass {
6     public void Test2() { } }
7 // Verwendung
8 MyClass mc = new MyClass();
9 mc.Test1();
10 mc.Test2();
```

#### 4.1.2 Abstrakte Klassen

Eine Mischung aus Klasse und Interface. Deklaration mit dem Schlüsselwort "abstract". Regeln: Kann nicht direkt instanziiert werden, Kann beliebig viele Interfaces implementieren, Alle Interface-Members müssen deklariert werden, Abgeleitete (nicht abstrakte) Klassen müssen alle abstrakten Members implementieren, Abstrakte Members können nur innerhalb abstrakter Klassen deklariert werden, Dürfen nicht «sealed» (versiegelt) sein (siehe später).

## 4.2 Structs

Sind Value types und werden auf dem Stack oder "in-line" in einem Objekt auf dem Heap abgelegt.

### Vererbung

Ableiten von Basisklasse nicht möglich, Verwenden als Basisklasse auch nicht möglich, Implementieren von Interfaces ist möglich.

### Felder

Felder dürfen im Struct nicht initialisiert werden!

```

1 struct Point {
2     int x;
3     int y;
4     public Point(int x, int y)
5     {
6         this.x = x; this.y = y;
7     }
8     public void MoveX(int x) { /* ... */ }
9     public void MoveY(int y) { /* ... */ }
10 }
11 Point p = new Point(2, 3);

```

### Verwendung

Ein Struct sollte nur unter folgenden Umständen verwendet werden. In allen anderen Fällen sollte eine Klasse verwendet werden.

- Repräsentiert einen einzelnen kleinen Wert
- Instanzgrösse ist kleiner als 16 Byte
- Ist immutable
- Wird nicht häufig geboxt
- Ist entweder: kurzlebig, eingebettet in anderes Objekt

## 4.3 Properties

Properties sind eine Kurzform für Get-/Set-Methoden. Diese können für Structs und Klassen eingesetzt werden. Sind ein reines Compiler-Konstrukt. Nutzen: Benutzersicht/Implementation können unterschiedlich sein, Validierung beim Zugriff, Ersatz für Public Fields auf den Interfaces, Über Reflection als Property identifizierbar. Private properties können nur von der Klasse selbst verwendet werden.

```

1 class MyClass {
2     private int length; // Backing field
3     // Property
4     public int Length {
5         get { return length; }
6         set { length = value; }
7     }
8     // Compiler Output
9     public int get_Length() { return length; }
10    public void set_Length(int value) { length = value; }

```

## 4.4 Auto-Implemented Properties

Das Backing field sowie die Setter und Getter werden automatisch generiert. Für jede abweichende Get-/Set-Logik muss Property explizit implementiert werden.

```

1 // Auto Property - backing field gets generated automatically
2 public int LengthAuto { get; set; }
3 public int LengthInitializes {get; /* set; */ };

```

## 4.5 Initialisierte Properties

Properties können bei der Objekt Erstellung direkt initialisiert werden. Das ist ein reines Compiler-Feature.

```

1 MyClass mc = new MyClass()
2 {
3     Length = 1,
4     Width = 2
5 };
6 // Compiler Output
7 MyClass mc = new MyClass();
8 mc.Length = 1;
9 mc.Width = 2;

```

## 4.6 Indexer

Indexer ermöglichen, dass Instanzen einer Klasse oder Struktur wie Arrays indiziert werden. Der indizierte Wert kann festgelegt oder ohne explizite Angabe eines Typs oder Instanzmembers abgerufen werden. Indexer ähneln Eigenschaften. Der Unterschied besteht jedoch darin, dass ihre Zugriffsmethoden Parameter verwenden. Das Schlüsselwort `this` definiert den Indexer. Das Schlüsselwort `value` für Zugriff auf Wert in Setter.

```

1 class SampleCollection<T> {
2     private T[] arr = new T[100];
3     // Define the indexer to allow client code to use [] notation.
4     public T this[int i] {
5         get { return arr[i]; }
6         set { arr[i] = value; }
7     }
8 }
9 class Program
10 {
11     static void Main() {
12         var stringCollection = new SampleCollection<string>();
13         stringCollection[0] = "Hello, World";
14         Console.WriteLine(stringCollection[0]);
15     }
16 }

```

## 4.7 Konstruktoren

Bei jedem Erzeugen einer Klasse / eines Structs verwendet (Aufruf von «new»).

### Default-Konstruktor Klasse

- Automatisch generiert, wenn nicht vorhanden
- Wenn anderer Kstr. vorhanden nicht mehr
- Kann manuell implementiert werden
- Initialisiert Felder mit `default(T)`
- Kann beliebig viele Felder initialisieren

### Default-Konstruktor Klasse

- Wird immer automatisch generiert!
- Kann nicht manuell definiert werden
- Initialisiert Felder mit `default(T)`
- Muss alle Felder initialisieren!

Typ	Default	Typ	Default
class	null	int	0
struct	Struct Alle Members sind default(T)	long	0L
bool	false	sbyte	0
byte	0	short	0
char	'\0'	uint	0
decimal	0.0M	ulong	0
double	0.0D	ushort	0
float	0.0F	enum	Resultat aus (E)0 E = Enumerations-Typ

Figure 9: Standard-Werte

#### 4.7.1 Statische Konstruktoren

Für statische Initialisierungsarbeiten verwendet und sind identisch für die Klasse und Struct. Regeln: zwingend Parameterlos, Sichtbarkeit darf nicht angegeben werden, Wird genau einmal ausgeführt, Kann nicht explizit aufgerufen werden.

#### 4.7.2 Dekonstruktoren

Ermöglichen Abschlussarbeiten beim Abbau eines Objekts (wie Java-Finalizer) Regeln: Nur bei Klassen erlaubt, Zwingend Parameterlos/Sichtbarkeitslos, Maximal einer erlaubt, Wird vom Garbage Collector aufgerufen und kann nicht explizit aufgerufen werden.

```
1 class MyClass {  
2     ~MyClass() {  
3         // Freigabe von File-Handles etc.  
4     }  
5 }
```

#### 4.7.3 Initialisierungs-Reihenfolge

### 4.8 Operatoren - Überladung



## 5 Methoden

### 5.1 Statische Methoden

Es gibt zwei Ausprägungen von statischen Methoden: Prozedur (Aufgabe ohne Rückgabewert), Funktion.

### 5.2 Value Parameter

Kopie des Inhalts wird übergeben (Wert oder Heap-Referenz).

```
1 void IncVal(int x) { x = x + 1; }
2 void TestIncVal() {
3     int value = 3;
4 }
5 IncVal(value); // value == 3
```

### 5.3 Reference Parameter

Adresse der Variable wird übergeben. Variable muss initialisiert sein. Es muss eine Variable übergeben werden. Hierfür wird das ref keyword verwendet.

```
1 void IncRef(ref int x) { x = x + 1; }
2 void TestIncRef() {
3     int value = 3;
4 IncRef(ref value); // value == 4
```

### 5.4 Out-Parameter

Werden wir ref-Parameter verwendet, aber sind für die Initialisierung gedacht. Das bedeutet, dass die Variable vorher nicht initialisiert werden muss. Das out keyword muss beim Aufrufer und in der Methode deklariert werden.

```
1 static void Init(out int val) {
2     val = 100;
3 }
4 int value;
5 Init(out value);
```

### 5.5 In Parameter

### 5.6 Params-Array

Erlaubt beliebig viele Parameter. Muss am Schluss der Deklaration stehen. Nur ein params-Array ist erlaubt. Darf nicht mit ref oder out kombiniert werden.

```
1 void Sum(out int sum, params int[] values) {
2     sum = 0;
3     foreach (int i in values) sum += i;
4 }
```

### 5.7 Optionale Parameter (Default Values)

Optionale Parameter ermöglichen die Zuweisung eines Default-Values. Deklaration muss hinter der erforderlichen Parameter erfolgen! Muss zur Compilezeit berechenbar sein.

```
1 private void Sort(int[] array, int from=0, bool ascending=true) { /* . */ }
```

## 5.8 Named Parameter

Identifikation der optionalen Parameter anhand des Namens (anstatt anhand der Position).

```
1 Sort(a, ignoreCase: true, from: 3);
```

## 5.9 Überladung

Mehrere Methoden mit dem gleichen Namen möglich. Voraussetzung: Unterschiedliche Anzahl Parameter oder Parametertypen oder Parameterarten (ref/out). *Der Rückgabetype spielt dabei keine Rolle!*

```
1 void Test(int x) { /* ... */ }
2 void Test(char x) { /* ... */ }
3 void Test(int x, long y) { /* ... */ }
4 void Test(long x, int y) { /* ... */ }
5 void Test(ref int x) { /* ... */ }
```

## 5.10 Lamda Expressions

- Eine Lambda Expression ist eine anonyme Methode
  - Keine Implementation einer benannten Methode nötig
  - Kein "delegate" Schlüsselwort nötig
  - Angabe von Parametertypen ist optional
- Ist die Basis für das Erzeugen von Delegates und Expression Trees.
- Zwei Ausprägungen: Expression Lambdas, Statement Lambdas.
- Werden in Delegates konvertiert

```
1 // Expression Lambda
2 Func<int, bool> fe = i => i % 2 == 0;
3 // Statement Lambda
4 Func<int, bool> fs = i => {
5     int rest = i%2;
6     bool isRestZero = rest == 0;
7     return isRestZero;
8 };
```

### 5.10.1 Type Inference

Parametertypen sind meistens redundant definiert. Somit ist die Angabe der Parameterliste obsolet.

```
1 Func<int, int, int, bool> p04;
2 p04 = (a, b, c) => true;
```

## 6 Vererbung

- Nur eine Basisklasse erlaubt
- Beliebig viele Interfaces erlaubt
- Structs können nicht erweitert werden
- Structs können nicht erben
- Structs können Interfaces implementieren
- Klassen sind direkt / indirekt über System.Object abgeleitet
- Structs sind über Boxing mit System.Object kompatibel

### 6.1 Typprüfungen

Prüft ob ein Objekt mit einem Typen kompatibel ist. Liefert true, wenn: Typ von obj identisch wie "T" ist, Typ von obj eine Sub-Klasse von "T" ist.

```

1 class Base { }
2 class Sub : Base { }
3 class SubSub : Sub { }
4 public static void Test() {
5     SubSub a = new SubSub();
6     if (a is SubSub) { /* ... */ } // True
7     if (a is Sub) { /* ... */ } // True
8     if (a is Base) { /* ... */ } // True
9     a = null;
10    if (a is SubSub) { /* ... */ } // False / NULL
11 }
```

### 6.2 Type Casts

Explizite Typumwandlung als Hinweis für den Compiler. Regeln: Null kann auch gecasted werden, Compilerfehler wenn Type Cast nicht zulässig ist oder wenn NULL in eine Value Type gecasted wird. Value Types können nie null sein!

```

1 class Base { }
2 class Sub : Base { }
3 class SubSub : Sub { }
4 public static void Test() {
5     Base b = new SubSub();
6     Sub s = b as Sub; // as keyword for the compiler
7 }
```

### 6.3 Methoden

Die Subklasse kann Members der Basisklasse überschreiben: Methoden, Properties, Indexer. Schlüsselwort "virtual" um Basis-Methode überschreibbar zu machen. Schlüsselwort "override" um die Basis-Methode zu überschreiben. Members sind per default NICHT virtual und override! Regeln: Signatur muss identisch sein, gleiche Rückgabewert, gleiche Sichtbarkeit.

#### Achtung!

Virtual kann nicht mit: static, abstract, private, override verwendet werden!

```

1 class Base {
2     public virtual void G() { /* ... */ }
3 class Sub : Base {
4     public override void G() { /* ... */ }
5 class SubSub : Sub {
6     public override void G() { /* ... */ }
7 }
```

## 6.4 Interfaces

Interface ähnelt einer rein abstrakten Klasse. Regeln: Kann nicht direkt instanziiert werden, Interface kann andere Interfaces erweitern, Sichtbarkeit auf Members darf nicht angegeben, Members sind implizit «abstract virtual», Members dürfen nicht «static» sein oder ausprogrammiert werden, Name beginnt mit einem grossen «I».

```
1 interface ISequence {
2     void Add(object x);           // Method
3     string Name { get; }         // Property
4     object this[int i] { get; set; } // Indexer
5     event EventHandler OnAdd;     // Event
6 }
```

### 6.4.1 Interfaces Implementieren

Regeln: Eine Klasse kann beliebig viele Interfaces implementieren, Alle Interface-Members müssen auf der Klasse vorhanden sein, «override» ist nicht nötig ausser allfällige Basisklasse definiert gleichen Member, Kombination mit «virtual» und «abstract» ist erlaubt, Implementierte Interface-Members müssen «public» und dürfen nicht «static» sein.

```
1 interface ISequence {
2     void Add(object x);           // Method
3     string Name { get; }         // Property
4     object this[int i] { get; set; } // Indexer
5     event EventHandler OnAdd;     // Event }
6 class List : ISequence {
7     public void Add(object x) { /* ... */ }
8     public string Name { get { /* ... */ } }
9     public object this[int i] { get { /* ... */ } set { /* ... */ } }
10    public event EventHandler OnAdd;
11 }
```

## 7 Delegates & Events

### 7.1 Delegates

Ein Delegat ist ein Typ, der ähnlich einem Funktionszeiger in C und C++ eine Methode sicher kapselt. Im Gegensatz zu C-Funktionszeigern sind Delegate objektorientiert, typensicher und sicher. Der Typ eines Delegaten wird durch den Namen des Delegaten definiert. Verwendung: Methoden können als Parameter übergeben werden, Definition von Callback-Methoden. Eine Zuweisung von null ist erlaubt.

```

1 public delegate void Notifier(string sender);
2 class Examples {
3     public static void Test() {
4         // Deklaration Delegate-Variable
5         Notifier greetings;
6         // Zuweisung einer Methode
7         greetings = new Notifier(SayHi);
8         // Kurzform
9         greetings = SayHi;
10        // Aufruf einer Delegate-Variable
11        greetings("John");
12    }
13    private static void SayHi(string sender) {
14        Console.WriteLine("Hello {0}", sender);
15    }

```

### 7.2 Multicast Delegates

Jeder Delegate-Typ ist ein Multicast Delegate. Delegate-Variable kann beliebig viele Methoden-Referenzen enthalten. Zuweisungen können mit: =, +=, -= gemacht werden.

### 7.3 Events

Events sind Instanzen von Delegates, wobei das Delegate implizit private ist, damit es das Event nur von intern getriggert werden kann. (Compiler Feature) Ein Event ist normalerweise void. Events werden benötigt um zwischen Objekten zu kommunizieren. Ändert etwas in einem Objekt werden die andere benachrichtigt (Observer). Jeder Event verfügt über kompilergenerierte, öffentliche Add(+=) und Remove(-=) Methoden für das Subscriben von Methoden, Lamdas, etc.

```

1 public delegate void AnyHandler(object sender, EventArgs e);

```

### 7.4 Anonyme Methoden

Anonyme Methoden sind immer in-place.

```

1 class AnonymousMethods {
2     int sum = 0;
3     void SumUp(int i) { sum += i; }
4     void Print(int i) { Console.WriteLine(i); }
5     void Foo() {
6         List<int> list = new List<int>();
7         list.ForEach(SumUp);
8         list.ForEach(Print);
9     }
10 }

```

## 8 Generics

Generics können in Klassen, Structs und Delegates verwendet werden. Generics sind für Value Types schneller, bei Reference Type jedoch nicht (Verglichen mit object). Die hat den Grund, dass kein Boxing und Unboxing verwendet wird. Vorteile: Hohe Wiederverwendbarkeit, Typensicherheit, Performance. Hauptanwendungsfall sind die Collections.

### 8.1 Type Constraints

Mit dem Keyword where kann eine Regel definiert werden, die der dynamische Typ erfüllen muss.

Constraint	Beschreibung
where T : struct	T muss ein Value Type sein.
where T : class	T muss ein Reference Type sein. Darunter fallen auch Klassen, Interfaces, Delegates
where T : new()	T muss einen parameterlosen «public» Konstruktor haben. Dieser Constraint muss – wenn mit anderen kombiniert – immer zuletzt aufgeführt werden
where T : «ClassName»	T muss von Klasse «ClassName» ableiten.
where T : «InterfaceName»	T muss Interface «InterfaceName» implementieren.
where T : TOther	T muss identisch sein mit TOther. oder T muss von TOther ableiten.

Figure 10: Type Constraints

### 8.2 Vererbung

Generische Klassen können von anderen generischen Klassen erben.

### 8.3 Nullable Types

- Der "?" Operator erlaubt es NULL werte einem Wertetype zuzuweisen. Der Typ ist dann *Nullable < T >*
- Arithmetische Ausdrücke mit Null sind immer false, ausser "null == null"
- Der "??" Operator erlaubt es einen Default Wert anzugeben, falls die Variable leer ist.

## 9 Iteratoren

Iteratoren werden für das Iterieren über die Collections verwendet. Kriterien: Muss IEnumerable / IEnumerable<T> implementieren, Muss eine Implementation von ähneln. → Methode GetEnumerator() mit Rückgabewert e, e hat eine Methode bool MoveNext(), e hat ein Property Current.

### 9.1 Interfaces

Es gibt zwei Interfaces, weil die ersten Varianten des .NET Frameworks keine Generics unterstützt haben. Sobald ein Generisches Interfaces implementiert wird, muss auch ein nicht generische implementiert werden.

```

1 // Nicht generische Variante
2 public interface IEnumerable
3 {
4     IEnumerator GetEnumerator();
5 }
6 public interface IEnumerator
7 {
8     object Current { get; }
9     bool MoveNext();
10    void Reset();
11 }
12 // Generische Variante
13 public interface IEnumerable<out T> : IEnumerable {
14     IEnumerator<T> GetEnumerator();
15 }
16 public interface IEnumerator<out T> : IDisposable, IEnumerator {
17     T Current { get; }
18     // Weitere Members werden vererbt
19 }

```

### 9.2 Zugriff

Mehrere aktive Iteratoren zur gleichen Zeit sind erlaubt. Enumerator-Objekt muss Zustand vollständig kapseln. So können unerwünschte Seiteneffekte unterbunden werden. Collection darf während der Iteration aber nicht verändert werden.

### 9.3 Iterator-Methoden

#### 9.3.1 yield return

Gibt für den nächsten Wert für die nächste Iteration eines "foreach" Loops zurück.

```

1 yield return "expression";

```

### 9.4 Spezifische Iteratoren

### 9.5 Extension Methode

Erlaubt das Erweitern bestehender Klassen um Methoden. Die Signatur der Klasse wird nicht verändert. Der Aufruf sieht jedoch so aus, als wäre es eine Methode der Klasse. *Deklaration:* Muss in einer statischen Klasse deklariert sein, muss "static" sein, Erster Parameter mit "this" Schlüsselwort voranstehend.

```

1 public static class ExtensionMethods {
2     static string ToStringSafe(this object obj) {
3         return obj == null
4             ? string.Empty : obj.ToString();
5     }
6 public static void Test() {

```

```
7  int myInt = 0;
8  object myObj = new object(); // Objects not null
9  myInt.ToString();
10 myInt.ToStringSafe();
11 myObj.ToString();
12 myObj.ToStringSafe(); // Object is null
13 myObj = null; myObj.ToString(); // Error
14 myObj.ToStringSafe(); // Works
15 }
16 }
```

## 9.6 Deferred Evaluation

Werte werden erst dann berechnet, wenn sie abgefragt werden.



## 10 Exceptions

Behandelt unerwartete Programmezustände oder Ausnahmeverhalten zur Laufzeit. Exceptions sollten so selten wie möglich und so oft wie nötig verwendet werden. Möglichst konkrete Exception-Klassen verwenden. NIE über eine Web-Schnittstelle übermitteln!

Regeln: catch-Block wird sequenziell gesucht, Exception-Typ muss von System.Exception abgeleitet werden, finally-Block wird immer ausgeführt.

Exceptions in C# sind unchecked. Das bedeutet Aufrufer einer Methode müssen die Exception nicht behandeln, wenn sie dies nicht wollen. Grosser Unterschied zu Java.

```
1 FileStream s = null;
2 try {
3     s = new FileStream(@"C:\Temp\Test.txt", FileMode.Open);
4 } catch (FileNotFoundException e) {
5     Console.WriteLine("{0} not found", e.FileName);
6 } catch (IOException) {
7     Console.WriteLine("IO exception occurred");
8 } catch {
9     Console.WriteLine("Unknown error occurred");
10 } finally {
11     if (s != null) s.Close();
12 }
```

### 10.1 Klasse System.Exception

Ist die Basisklasse für alle Exceptions. Alle Exceptions müssen von der Klasse System.Exception ableiten. Besteht aus: Konstruktor und Properties.

#### Properties

- InnerException: Verschachtelte Exception
- Message: Fehlermeldung als String
- Source: Name der Applikation, Objekts, Frameworks, welches den Fehler verursacht hat
- StackTrace: Methodenaufruflkette als String
- TargetSite: Ausgeführter Code-Teil, der den Fehler verursacht

```
1 public class Exception : ISerializable, _Exception {
2     public Exception();
3     public Exception(string message);
4     public Exception(string message, Exception innerException);
5     public Exception InnerException { get; }
6     public virtual string Message { get; }
7     public virtual string Source { get; set; }
8     public virtual string StackTrace { get; }
9     public MethodBase TargetSite { get; }
10    public override string ToString();
11    /* ... */
12 }
```

### 10.2 Rethrowing

Der Stack Trace bleibt erhalten, wenn das "throw" erneut ausgeführt wird.

```
1 try {
2     throw new Exception("Failure");
3 } catch (Exception e) {
4     throw;
5 }
```

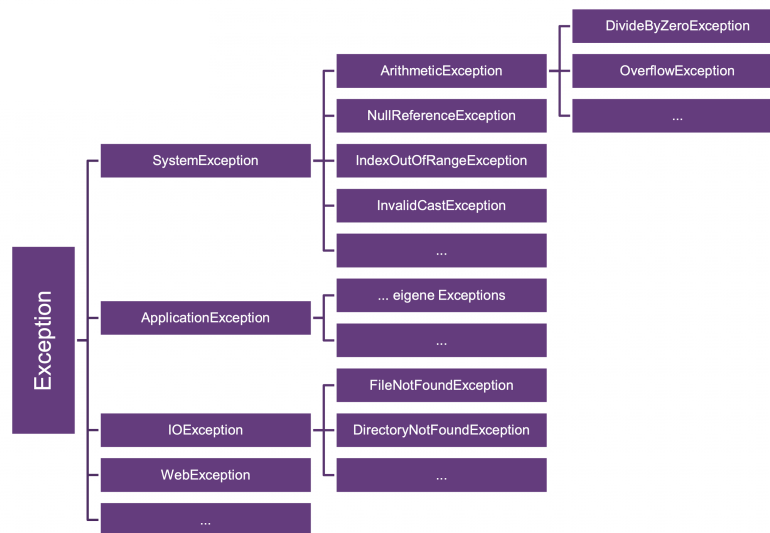


Figure 11: Exception-Typen

### 10.3 Exception-Klassen

### 10.4 Catch-Klausel

Der Call Stack wird rückwärts nach passender "catch" Klausel durchsucht. Programmabbruch mit Fehlermeldung und Stack Trace, wenn keine "catch" Klausel gefunden wird.

### 10.5 Exception Filters

Catch-Block wird nur unter definierter Bedingung ausgeführt. Diese erwartet eine "bool" Expression.

```

1 try {
2 } catch (Exception e) when (DateTime.Now.Hour < 18) {
3 } catch (Exception e) when (DateTime.Now.Hour >= 18) {
4 }

```

## 11 Language Integrated Query (LINQ)

LINQ erlaubt eine Query Syntax um Abfragen an beliebigen Datenstrukturen zu machen. Man unterscheidet den Extension- und Query Expression Syntax (Erinnert an SQL), wobei beide die gleichen Dinge erlauben. Auch LINQ ist reines Compiler Feature.

LINQ kann beliebige Collections durchsuchen, die `IEnumerable<T>` implementieren. LINQ kennt zwei verschiedene Syntaxen: LINQ Query syntax und die LINQ Method syntax. Der Compiler wandelt Query Expressions in Lambda Expressions (Aufruf der Extension Methods) um.

- Reine Compiler-Technologie
- Query-Syntax (ähnlich SQL)
- Beliebige Datenstrukturen als Basis (Objekte, XML etc.)
- Typensicherheit
- Erlaubt funktionale Programmierung mit Lambda
- Erlaubt deklarativen Programmierstil mittels Anonymous Types und Object "initializers"
- Verbesserung Type Inference

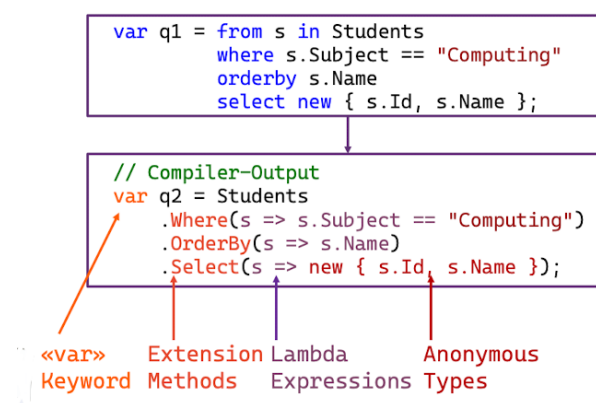


Figure 12: Compilerumwandlung zu Extension Methods

```

1 // Sequenz           Elemente
2 string[] name = { "Tom", "Dick", "Harry" };

1 // Query Expression syntax
2 var queryWashingtonSorted =
3     from e in employees
4     where e.State == "WA"
5     orderby e.Name descending
6     select new
7     { e.Name, e.Address };
8 // Query Extension Method / Lambda syntax
9 queryWashingtonSorted = employees
10     .Where(e => e.State == "WA")
11     .OrderByDescending(e => e.Name)
12     .Select(e => new
13     { e.Name, e.Address });

```

### 11.1 LINX Extension Methods

LINQ definiert in der Klasse `Enumerable` `using System.Linq`; eine Vielzahl von Query Operatoren. Die Methoden in dieser Klasse stellen eine Implementierung von Operatoren zum Abfragen von Datenquellen bereit, die `IEnumerable<T>` implementieren.

```

1 int[] numbers = { 1, 4, 2, 9, 13, 8, 9, 0, -6, 12 };
2 IEnumerable<int> res = numbers
3     .Where(i => i >= 5)

```

```

4  .OrderBy(i => i);
5  IEnumerable<int> sqr = numbers
6  .Skip(2)
7  .Take(4)
8  .Select(k=> k * k);

```

## 11.2 Expression-Bodies Members

Ermöglichen uns in einer kurz gehaltenen Syntax Methoden und Properties zu definieren. Dies erlaubt es uns den return sowie den body wegzulassen.

```

1 public class Examples {
2     private int value;
3     // Constructors / Destructors (C# 7.0)
4     public Examples(int v) => this.value = v;
5     ~Examples() => this.value = 0;
6     // Methods (C# 6.0)
7     public int Sum(int x, int y) => x + y;
8
9     public int GetZero() => 0;
10    public void Print() => Console.WriteLine("...");
11    // Properties (C# 6.0)
12    public int Zero => 0;
13    public int Bla => Sum(Zero, 2);
14    // Getters/Setters (C# 7.0)
15    public int Value {
16        get => this.value;
17        set => this.value = value;
18    }
19 }

```

## 11.3 Query Expressions Syntax

```

1 // 1. Datenquelle waehlen
2 int[] numbers={0,1,2,3,4,5,6};
3
4 // 2. Query erstellen
5 var numQuery = from num in numbers
6     where (num % 2) == 0
7     select num;
8
9 // 3. Query ausfuehren
10 foreach (int num in numQuery) { Console.Write("{0,1} ", num); }

```

- from: Datenquelle
- where: filter
- orderby: Sortierung
- select: Projektion
- group: Gruppierung ein eine Sequenz von Gruppen Elementen
- join: Verknüpfung zweier Datenquellen
- let: definition von Hilfsvariablen

## 11.4 Gruppierung

```

1 // q: IEnumerable<IGrouping<string, string>>
2 var q = from s in Students
3     group s.Name by s.Subject;
4 }
5

```

Standard	Positional	Set Operations
Select	First[OrDefault] Erstes passendes Element für Prädikat	Distinct Distinkte Liste der Elemente
Where	Single[OrDefault] Erstes passendes Element für Prädikat	Union Distinkte Elemente zweier Mengen
OrderBy[Descending]	ElementAt Element an numerischer Position	Intersection Überschneidende Elemente zweier Mengen
ThenBy[Descending]	Take / Skip Alle Elemente vor/nach einer numerischen Position	Except Elemente aus Menge A die in Menge B fehlen
GroupBy	TakeWhile / SkipWhile Alle Elemente vor/nach passendem Prädikat	Repeat N-fache Kopie der Liste
Join / GroupJoin	Reverse Alle Elemente in umgekehrter Reihenfolge	
Count / Sum / Min / Max / Average		

Figure 13: Query Operatoren / Extension Methoden

```

6 // Gruppierung mit direkter Wiederverwendung
7 var q = from s in Students
8   group s.Name by s.Subject into g
9   select new {
10     Field = g.Key,
11     N = g.Count()
12   };
13
14 // Anz. Bestellungen pro Datum
15 from best in Bestellungen
16 group best by best.Datum into datumGroup
17 orderby datumGroup.Key
18 select new {
19   Datum = datumGroup.Key,
20   Anzahl = datumGroup.Count()
21 };

```

## 11.5 Inner Joins

Ein Inner Join nimmt nur jene Ergebnisse, die nicht null sind.

```

1 var q = from s in Students
2   join m in Markings on s.Id equals m.StudentId
3   select s.Name + ", " + m.Course + ", " + m.Mark

```

## 11.6 Group Joins

Ein Group Join verwendet die into Expression.

```

1 var q =
2   from s in Students
3   join m in Markings on s.Id equals m.StudentId
4   into list
5   select new
6     Name = s.Name,
7     Marks = list
8   };

```

## 11.7 Left Outer Joins

```

1 var q = from s in Students
2   join m in Markings on s.Id equals m.StudentId into match

```

```
3 from sm in match.DefaultIfEmpty()
4 select s.Name + ", " + (sm == null
5     ? "?"
6     : sm.Course + ", " + sm.Mark);
```

## 11.8 Select Many

Erleichtert das Zusammenfassen verschachtelter Listen.

```
1 // Liste der Projekte mit den zugeordneten Mitarbeitern
2 projList = projects
3     .SelectMany(p => p.Employees
4         .Select(e => new
5             {
6                 Project = p.Name,
7                 Employee = e.Name
8             })
9     )
10     .OrderBy(p => p.Project)
11     .ThenBy(p => p.Employee);
```

## 12 Direct Initialization

### 12.1 Object Initializers

Object Initialisierer erlaubt das Instanzieren und Initialisieren einer Klasse in einem einzigen Statement. Die Objekte lassen sich auch erzeugen, wenn kein passender Konstruktor zur Verfügung steht.

```
1 Student s1 = new Student("John") {  
2     Id = 2009001, // Set public field  
3     Subject = "Computing" // Set property  
4 };  
5 Student s2 = new Student {  
6     Name = "Ann",  
7     Id = 2009002,  
8     Subject = "Mathematics"  
9 };
```

### 12.2 Collection Initializers

Ist das selbe wie Objekte Initializers, jedoch mit Listen.

```
1 List<int> l1 = new List<int> { 1, 2, 3, 4 };  
2 Dictionary<int, string> d1 = new Dictionary<int, string>  
3     {1, "a"},  
4     {2, "b"},  
5     {3, "c"}  
6 };  
7 d1 = new Dictionary<int, string> {  
8     [1] = "a",  
9     [2] = "b",  
10    [3] = "c"  
11 };  
12  
13 object s = new Dictionary<int, Student> {  
14     { 2009001, new Student("John") {  
15         Id = 2009001,  
16         Subject = "Computing" } },  
17     { 2009002, new Student {  
18         Name = "Ann", Id = 2009002,  
19         Subject = "Mathematics" } }  
20 };
```

### 12.3 Anonymous Types (Let)

- Mit dem Schlüsselwort let wird der Typ vom Compiler herausgefunden
- let kann nur für lokale Variablen verwendet werden. Der Einsatz bei Parametern, Klassenvariablen und Properties ist nicht erlaubt.
- Der Typ wird aus der Zuweisung abgeleitet, wobei die Variable zu 100% typensicher bleibt.

## 13 Tasks

Leichtgewichtige Variante eines Threads und repräsentiert eine asynchrone Operation.

### Task

- Hat einen Rückgabewert
- Unterstützt "Cancellation" via Token
- Mehrere parallele Operationen in einem Task
- Vereinfachter Programmfluss (async/wait)
- Verwendet einen Thread Pool
- Eher ein High level Konstrukt

### Thread

- Kein Rückgabewert (Alternativen aber möglich)
- Keine "Cancellation"
- Nur eine Operation in einem Thread
- Keine Unterstützung for async/wait.
- Thread Pool muss explizit (manuell) verwendet werden.
- Low Level Konstrukt



## 14 Entity Framework Core

Unter .NET kommt ADO.NET als Entity Framework zum Einsatz. Man unterscheidet zwischen zwei Varianten wie die Entitätsklassen/Datenbanken erstellt werden können. Das Entity Framework muss mit NuGet installiert werden. Es werden verschiedene Provider unterstützt: MS SQL, MySQL, PostgreSQL, SQLite, SQL Compact, in-memory.

**DB First** Man erstellt zuerst ein Domain Model und generiert daraus die Klassen. Man kann das Model auch von einer existierenden Datenbank ableiten und dann wieder die Klassen daraus generieren

**Code First** Man erstellt die Model Klassen und lässt die Datenbank automatisch generieren.

**Model First** Man erstellt zuerst das EDM (Entity Data Model) und generiert daraus die Database, sowie die Klassen

### 14.1 OR-Mapping

OR-Mapping ist eine Technik der Softwareentwicklung, mit der ein in einer objektorientierten Programmiersprache geschriebenes Anwendungsprogramm seine Objekte in einer relationalen Datenbank ablegen kann. EF Core verwendet für das verschiedene Provider, welche eine Vielzahl von SQL- und NoSQL-Datenbanken unterstützen.

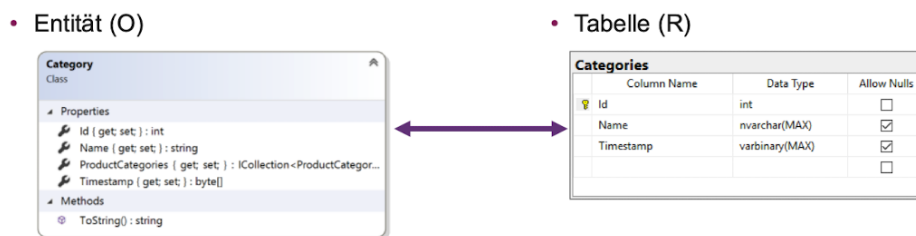


Figure 14: Mapping von Objekten auf Relationen

#### 14.1.1 Mapping-Ansätze

Das Mapping von Klassen auf das darunterliegende Speichermodell (Datenbank) kann auf drei verschiedene Ansätze realisiert werden. **Mapping: By Convention, By Attributes, By Fluent API.**

**Providers** Diverse relationale SQL Providers.

**Entity** Ein Objekt mit einem Key (z.B ID). Mehrere dieser Objekte werden zu einem Entity-Set zusammengefasst.

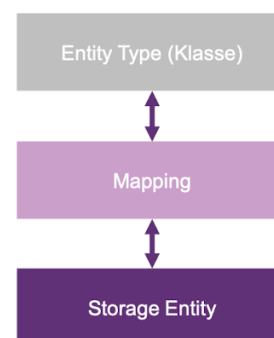
**Mapping** Mapping der Klassen auf das darunter liegende Speichermodell.

**Mappingansätze** Zuordnung von Entity Type zu Storage Entity, Property zu Column, Entity Key zu Primary Key, Foreign Key zu Relationship.

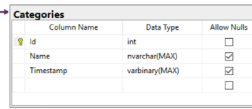
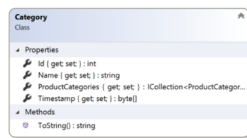
**Storage Entity** Relationales Modell/Graph/Collection, abhängig vom gewählten Provider. Ausprägungen: Table, View, Stored Procedures, etc. Inhalte: Columns, Primary KEys, Unique Key Constraints, Foreign Keys.

**Association** Definiert eine Assoziation zwischen Entitäten (z.B Navigation Properties, Foreign Keys).

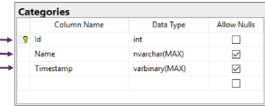
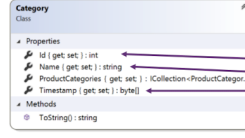
**Mapping-Arten:** Das Mapping kann über das Entity-Level sowie das Attribut-Level realisiert werden.



## Entity-Level



## Attribut-Level



## 14.1.2 Model

**Convention** Automatisches Mapping ohne explizite Konfiguration.

**Fluent API** Extensions Method Syntax, Überschriebene Methode von "OnModelCreating" im DbContext **protected override void OnModelCreating(ModelBuilder modelBuilder).**

**Data Annotations|Attributes** Deklaratives Mapping, Attribute direkt auf Model-Klassen

## Include/Exclude Entities

```

1 public class ShopContext : DbContext {
2     // Convention - DbSet-Property im Context
3     public DbSet<Category> Categories { get; set; }
4
5     // Fluent API - Entry im Builder oder Ignore im Model Builder
6     protected override void OnModelCreating( ModelBuilder modelBuilder) {
7         modelBuilder.Entity<AuditEntry>();
8         modelBuilder.Ignore<Metadata>();
9     }
10 }
11 public class Category {
12     public int Id { get; set; }
13     public string Name { get; set; }
14     public ICollection<Product> Products { get; set; } // Convention -
15     // Indirekt via Navigation Property
16     public ICollection<Metadata> Metadata { get; set; }
17 }
18 public class Product { /* ... */ }
19 public class AuditEntry { /* ... */ }
20 [NotMapped] // Data Annotations
21 public class Metadata { /* ... */ }

```

## Include/Exclude Properties

```

1 public class ShopContext : DbContext {
2     public DbSet<Category> Categories { get; set; }
3     protected override void OnModelCreating(ModelBuilder modelBuilder) {
4         modelBuilder.Entity<Category>()
5             .Property(b => b.Name);
6         modelBuilder.Entity<Category>()
7             .Ignore(b => b.LoadedFromDatabase); // Fluent API - Ignore im
8         // Builder
9     }
10 }
11 public class Category {
12     public int Id { get; set; }
13     public string Name { get; set; }
14     [NotMapped] // Data
15     // Annotations
16     public DateTime LoadedFromDatabase { get; set; }
17 }

```

## Keys

```

1 public class ShopContext : DbContext {
2     public DbSet<Category> Categories { get; set; }
3     protected override void OnModelCreating(ModelBuilder modelBuilder) {

```

```

4      modelBuilder.Entity<Category>()
5          .HasKey(e => e.Id)                                // Fluent API -
Einzigste Möglichkeit fuer Composite Keys
6          .IsRequired();
7      }
8  }
9  public class Category {
10      [Key]                                                  // Data Annotations
11      public int Id { get; set; }
12      public string Name { get; set; }
13  }
14  public class Tanslation {
15      public string Language { get; set; }
16      public int CategoryId { get; set; }
17  }

```

### Required/Optional

```

1  public class ShopContext : DbContext {
2      public DbSet<Category> Categories { get; set; }
3      protected override void OnModelCreating(ModelBuilder modelBuilder) {
4          modelBuilder.Entity<Category>()
5              .Property(e => e.Name)
6              .IsRequired();                                // Fluent API
7      }
8  }
9  public class Category {
10      public int Id { get; set; }
11      [Required]                                            // Data Annotations
12      public string Name { get; set; }
13      public bool? IsActive { get; set; }
14  }

```

**Maximum Length Convention:** Keine Restriktion / z.b. NVARCHAR(MAX), 450 Zeichen bei Keys

```

1  .Property(e => e.Name).HasMaxLength(500)                // Fluent API
2  [MaxLength(500)]                                         // Data Annotations

```

**Indexes Convention:** Werden bei Foreign Keys automatisch erstellt

```

1  modelBuilder.Entity<Category>()                          // Fluent API - Non-unique
    Index
2      .HasIndex(b => b.Name);
3  modelBuilder.Entity<Category>()                          // Fluent API - Unique Index
4      .HasIndex(b => b.Name)
5      .IsUnique();
6  modelBuilder.Entity<Category>()                          // Fluent API - Multi-column
    Index
7      .HasIndex(b => new { b.Name, b.IsActive });
8  // Data Annotations - nicht unterstuetzt

```

### 14.1.3 Relationale DB (SQL Server)

**Tabellen** Convention: Tabellename = Klassenname (Pluralized) (z.b. dbo.Categories)

```

1  // Fluent API - Name zwingend, Schema optional
2  modelBuilder.Entity<Category>()
3      .ToTable("Category", schema: "dbo");
4  // Data Annotations - Name zwingend, Schema optional
5  [Table("Category", Schema = "dbo")]
6  public class Category {...}

```

**Spalten** Convention: Spaltenname = Property-Name

```

1 // Fluent API
2 modelBuilder.Entity<Category>()
3     .Property(e => e.Name)
4     .HasColumnName("CategoryName", order: 1);
5 // Data Annotations
6 [Column("CategoryName", Order = 1)]
7 public string Name { get; set; }

```

**Datentypen / Default Values Convention:** Keine Default Values

```

1 // Fluent API
2 modelBuilder.Entity<Category>()
3     .Property(e => e.Name)
4     .HasColumnName("CategoryName")
5     .HasColumnType("NVARCHAR(500)") // Datentyp-Name des
        Zielsystems
6     .HasDefaultValue("----"); // Default (Wert/
        Gueltige SQL Expression)
7 //Data Annotation - Datentyp-Name des Zielsystems, Default Values nicht
        unterstuetzt.
8 [Column("CategoryName", TypeName = "NVARCHAR(500)")]
9 public string Name { get; set; }

```

**Relationship / Association – One-to-Many / Fully Defined Relationships** Convention: Collection Navigation Property (1-Ende), Reference Navigation Property (N-Ende), Foreign Key Property

```

1 // Fluent API - HasOne/WithMany oder HasMany/WithOne
2 modelBuilder.Entity<Product>()
3     .HasOne(p => p.Category)
4     .WithMany(b => b.Products)
5     .HasForeignKey(p => p.CategoryId)
6     .HasConstraintName("FK_Product_CategoryId");
7 public class Product {
8     public int Id { get; set; }
9     public int CategoryId { get; set; }
10    //Data Annotations - Auf Navigation Property wird Foreign Key Property
        definiert
11    [ForeignKey(nameof(CategoryId))]
12    public Category Category { get; set; }
13 }
14 public class Category {
15     public int Id { get; set; }
16     public ICollection<Product> Products { get; set; }
17 }

```

**Relationship / Association – One-to-Many / No Foreign Key Property** Convention: Collection Navigation Property (1-Ende), Reference Navigation Property (N-Ende)

```

1 // Fluent API - .HasForeignKey weglassen
2 modelBuilder.Entity<Product>()
3     .HasOne(p => p.Category)
4     .WithMany(b => b.Products)
5     .HasConstraintName("FK_Product_CategoryId");
6 public class Product {
7     public int Id { get; set; }
8     //Data-Annotation - Foreign Key weglassen
9     public Category Category { get; set; }
10 }
11 public class Category {
12     public int Id { get; set; }
13     public ICollection<Product> Products { get; set; }
14 }

```

**Relationship / Association – One-to-Many / Single Navigation Property** Convention: Collection Navigation Property (1-Ende)

```

1 //Fluent API - .HasOne ist anders
2 modelBuilder.Entity<Product>()
3     .HasOne<Category>()
4     .WithMany(b => b.Products)
5     .HasConstraintName("FK_Product_CategoryId");
6 public class Product {
7     //Data Annotations - Foreign Key + Navigation Property weglassen
8     public int Id { get; set; }
9 }
10 public class Category {
11     public int Id { get; set; }
12     public ICollection<Product> Products { get; set; }
13 }

```

**Relationship / Association - One-to-one / Many-to-many** One-to-one + Many-to-many

**One-to-one:** Nur Reference Navigation Property, keine Collection Navigation Property, `.HasOne(...).WithOne(...)`

**Many-to-many:** aktuell nicht unterstützt, work-around (Assoziations-Klasse, zwei One-to-many Relationships)

## 14.2 DB Context

Der DbContext ist das Herzstück des Entity Frameworks. Er ist die Verbindung zwischen unseren Entitätsklassen und der Datenbank. Der DbContext ist verantwortlich für die Datenbankinteraktionen wie das Abfragen der Datenbank und das Laden der Daten in den Speicher als Entität. Er verfolgt auch die an der Entität vorgenommenen Änderungen und speichert die Änderungen in der Datenbank. Kombiniert die Patterns: Repository, Unit of Work.

- **Design Time:** Model definieren (OR-Mapping), Konfiguration, Database Migrations.
- **Run-Time:** Connections verwalten, CRUD Operationen, Change Tracking, Caching, Transaction Management.

### DbContext Lifecycle

- **Sollte nicht zu lange leben:** Limitierte Anzahl Connections im Client Connection Pool, Change-Tracking wird über die Zeit ineffizient.
- **Sollte nicht geshared werden:** Ist nicht thread-safe, Exception kann Instanz unbrauchbar machen.

```

1 using (ShopContext context = new ShopContext()) {
2     /* Context / Database Operations */
3 }

```

### 14.2.1 Change Tracking

Registriert alle Änderungen an getrackten Entities. Aktualisiert den Entity State. Agiert ohne die Datenbank und schreibt nur die Änderungen (Keine Live Checks).

- `Add()` -> Added
- `Remove()` -> Deleted
- `Update()` -> Modified
- `Unchanged()` -> Unchanged
- `Not tracked` -> Detached

```

1 // New Record
2 Category cat = new Category { Name = "Laptops" }; // EntityState.Detached
3 context.Add(cat); // EntityState.Added
4 context.SaveChanges(); // EntityState.Unchanged

```

```

5 cat.Name = "Notebooks"; // EntityState.Modified
6 context.SaveChanges(); // EntityState.Unchanged
7 context.Remove(cat); // EntityState.Deleted
8 context.SaveChanges(); // EntityState.Unchanged

```

### 14.2.2 Ladestrategien (Lazy, Explizit, Eager Loading)

Es wird standardmässig Lazy Loading verwendet.

**Lazy Loading** Assoziationen werden per se nicht geladen werden aber bei Zugriff auf Property automatisch nachgeladen. Collections werden komplett geladen. Passiert in separater Abfrage. Daten werden erst geladen, wenn sie referenziert werden. z.B erst wenn effektiv auf die Membervariable (Liste aus mehreren Items) zugegriffen wird. Für das Lazy Loading müssen die Methoden **virtual** definiert werden.

**Eager Loading** Assoziationen werden per se nicht geladen. Include() Statement für einzelne Assoziationen. Passiert in der gleichen Abfrage per JOIN.

**Explicit Loading** Assoziationen werden per se nicht geladen und werden explizit nachgeladen. Collections werden komplett geladen. Passiert in separater Abfrage. Even with lazy loading disabled it is still possible to lazily load related entities, but it must be done with an explicit call. To do so you use the Load method on the related entity's entry

```

1 Order order = context.Orders.First();
2 var customer = order.Customer; //customer is ''null''
3 var items = order.Items; //items is ''null''

1 // lazy loading - zusätzliche Ladelogik ausführen bei Zugriff auf Property
2 //Variante 1: Manuell - Auf Auto-Properties verzichten & Logik manuell
  implementieren
3 //Variante 2: Proxies
4 public class Order {
5     public int Id { get; set; }
6     public virtual Customer Customer { get; set; } // virtual wichtig
7 }
8 public class OrderProxy : Order {
9     public virtual Customer Customer { /* ??? */ } // virtual wichtig
10 }
11 Order order = context
12     .Orders
13     .First();

1 // eager loading - mit .Include wird definiert was alles zusätzlich ins RAM
  geladen werden soll
2 Order order = context
3     .Orders
4     .Include(o => o.Customer) // Eager Loading
5     .Include(o => o.Items)
6     .ThenInclude(oi => oi.Product) // Cascaded eager
  loading
7     .First();

1 // explicit loading - .Load() führt dazu dass die Daten nachgeladen und in
  den Parent geladen werden D.h. es wird eine neue Query an die DB gemacht.
2 Order order = context
3     .Orders
4     .First();
5 context
6     .Entry(order)
7     .Reference(o => o.Customer) // Parents
8     .Load();
9 context
10    .Entry(order)

```

```

11     .Collection(o => o.Items) // Collections
12     .Query()
13     .Where(oi => oi.QuantityOrdered > 1)
14     .Load();

```

### 14.3 LINQ to Entities

Das EF führt keine LINQ queries aus! Das EF Core generiert Queries und die Datenbank führt diese dann aus.  
**Einfaches Beispiel**

```

1 // DbContext instanzieren, DB Verbindung oeffnen, Cache/Change Tracker
  initialisieren.
2 using (ShopContext context = new ShopContext()) {
3     Category category = context // Abfrage mit LINQ (
  direkt)
4     .Categories
5     .Single(c => c.Id == 1);
6
7     category.Name = $"{category.Name} / Changed"; // Daten aendern /
  speichern
8     context.SaveChanges();
9
10    var categories = context.Categories;
11    foreach (Category c in categories) { Console.WriteLine(c.Name); }
  // Abfrage mit LINQ (deferred)
12 } // Context schliessen - Cache invalidieren / Datenbank-Verbindung
  zurueck in Connection Pool

```

#### 14.3.1 CUD Operationen (Create, Update, Delete)

DbContext agiert nach dem Unit of Work (UoW) pattern. Objekt wird beim Laden aus der Datenbank automatisch der UoW registriert. Änderungen werden aufgezeichnet. Beim Speichern werden alle Änderungen in einer einzigen Transaktion geschrieben.

##### Insert

```

1 using (ShopContext context = new ShopContext()) {
2     Category cat = new Category { Name = "Notebooks" };
3     // Add to Context (3 alternatives)
4     // - Use .Add(...) to apply to whole graph
5     // - Use .State when only for this entity
6     context.Add(cat);
7     context.Categories.Add(cat);
8     context.Entry(cat).State = EntityState.Added;
9     // Save SQL is executed here
10    context.SaveChanges();
11    // Check Primary Key
12    int id = cat.Id; // Category.Id is populated
13 }

```

##### Update

```

1 using (ShopContext context = new ShopContext());
2 Category cat = await context
3     .Categories
4     .FirstAsync();
5 // Change
6 cat.Name = "Changed";
7
8 // Save SQL is executed here
9 await context.SaveChangesAsync();

```

**Delete**

```

1 using (ShopContext context = new ShopContext()) {
2     Category cat = context.Categories.First(c => c.Name == "Notebooks");
3     // Remove (3 alternatives)
4     // - Use .Remove(...) to apply to whole graph
5     // - Use .State when only for this entity
6     context.Remove(cat);
7     context.Categories.Remove(cat);
8     context.Entry(cat).State = EntityState.Deleted;
9     // Save SQL is executed here
10    context.SaveChanges();
11 }

```

**14.3.2 CUD von Assoziationen**

Assoziationen können durch drei Operationen angepasst werden: Anpassung Navigation Properties, Hinzufügen/Entfernen von Elementen in Collection Navigation Properties, Setzen des Foreign keys.

**Durch Anpassung von Navigation Properties**

order.Customer = customer

**Hinzufügen / Entfernen von Elementen in Collection Navigation Properties**

```

1 customer.Orders.Add(order);
2 customer.Orders.Remove(order);

```

**Setzen des Foreign Keys**

order.CustomerId = 1;

-> einzige Variante, welche keine weiteren Datenbankzugriffe benötigt

**Insert Object Graph**

```

1 using (ShopContext context = new ShopContext()) {
2     Customer cust = new Customer {
3         Name = "Anna"
4         Orders = new List<Order> {
5             new Order { /* ... */ },
6             new Order { /* ... */ }
7         }
8     };
9     // Add to Context
10    context.Add(cust);
11    // Save - SQL is executed here
12    context.SaveChanges();
13 }

```

**Insert Related Entity**

```

1 using (ShopContext context = new ShopContext()) {
2     Customer cust = context
3         .Customers
4         .Include(c => c.Orders)
5         .First();
6     cust.Orders.Add(new Order());
7
8     // Save - SQL is executed here
9     context.SaveChanges();
10 }

```

**Change Relationship 1**



```

1 using(ShopContext context = new ShopContext()) {
2     Order order = context
3         .Orders
4         .First();
5     //Change - via Nav.Property
6     order.Customer = context
7         .Customers
8         .First(c => c.Name == "Angela");
9
10    // Save - SQL is executed here
11    context.SaveChanges();
12 }

```

### Change Relationship 2

```

1 using(ShopContext context = new ShopContext()) {
2     Order order = context
3         .Orders
4         .First();
5     //Change - via Foreign Key
6     order.CustomerId = 2;
7
8     // Save - SQL is executed here
9     context.SaveChanges();
10 }

```

## 14.4 Optimistic Concurrency

**Optimistic:** Transaktionen laufen unbehindert an. Beim Abschliessen wird in einer Validierungsphase überprüft, ob Konflikte aufgetreten sind und gegebenenfalls die Transaktion zurückgesetzt.

Wenn ein Objekt im gleichen Context geladen wird, gibt der Context das gecachte Objekt zurück. Die Objekte werden anhand ihrem Entity Key gecached.

### Erkennung von Konflikten:

**Timestamp** Pro Record Timestamp / Row Version wird ein Timestamp erstellt. Beim Laden wird die Versionsnummer als Sessionzustand vermerkt. Validierung: Beim Zurückschreiben der Daten, wird die Session-Versionsnummer mit der Versionsnummer der DB verglichen.

**Concurrency Token** Für jedes geladene Datenfeld wird der ursprünglich gelesene Wert in der Applikation gespeichert. Änderungen werden auf einer Kopie ausgeführt. Validierung: Beim Zurückschreiben wird der ursprünglich gelesene Wert mit dem aktuellen Wert in der DB verglichen.

### Timestamp

```

1 public class ShopContext : DbContext {
2     public DbSet<Category> Categories { get; set; }
3     protected override void OnModelCreating(ModelBuilder modelBuilder) {
4         modelBuilder.Entity<Category>()
5             .Property(p => p.Timestamp)
6             .IsRowVersion();    // Fluent API
7     }
8 }
9 // --- ODER ---
10 [Timestamp]                // Data Annotations
11 public byte[] Timestamp { get; set; }

```

### Concurrency Token

```

1 public class ShopContext : DbContext {
2     public DbSet<Product> Products { get; set; }
3     protected override void OnModelCreating(ModelBuilder modelBuilder) {
4         modelBuilder.Entity<Product>()

```

```

5         .Property(p => p.Name)
6         .IsConcurrencyToken();           // Fluent API
7     modelBuilder.Entity<Product>()
8         .Property(p => p.Price)
9         .IsConcurrencyToken();           // Fluent API
10    }
11 }
12 // --- ODER ---
13 [ConcurrencyCheck]                       // Data Annotations
14 public string Name { get; set; }

```

### Konfliktbehandlung

DbUpdateConcurrencyException beinhaltet fehlerhafte "Entries" (Aktuelle Werte, Originale Werte, Datenbank Werte).

- 1. Ignorieren - Der Fehler wird ignoriert und die Daten überschrieben
- 2. Benutzer Fragen - Der User entscheide, was passieren soll.
- 3. Autokorrektur - das Programm kann den Error verarbeiten

## 14.5 Vererbung

Wird aufgrund von: Speicherplatzverbrauchs, Zu vielen Tabellen, Inkonsistenzen nicht verwendet. Es gibt drei verschiedene **Ansätze**: Table per Hierarchy, Table per Type, Table per Concrete Type.

### Table per Hierarchy

Eine Tabelle pro Vererbungs-Hierarchie (EF Core Standard). Nur über den DbContext definiertbar. Abgeleitete Klassen als DbSet Properties definieren oder mit dem modelBuilder.

```

1 modelBuilder.Entity<Product>()
2     .HasDiscriminator<int>("ProductType")
3     .HasValue<Product>(0)
4     .HasValue<MobilePhone>(1)
5     .HasValue<Tablet>(2);

```

	Id	Name	Description	Price	ProductType	OperatingSystem	SupportsUmts	HasKeyboard
1	1	Apple iPad	NULL	600.00	2	NULL	NULL	0
2	2	Samsung Galaxy Tab	NULL	400.00	2	NULL	NULL	0
3	3	Apple iPhone	NULL	19000.00	1	OSX	0	NULL
4	4	Samsung Galaxy Note	NULL	1200.00	1	Android	0	NULL

Figure 15: Table per Hierarchy

### Table per Type

Tabelle für jeden konkreten Typ mit allen benötigten Feldern. Keine Fremdschlüsselbeziehung (Parent-Child). Nachteil: Joins

- **Tabelle "Products"**: Id, Name, Description, Price
- **Tabelle "MobilePhones"**: Id, OperatingSystem, SupportsUmts
- **Tabelle "Tablets"**: Id, HasKeyboard

### Table per Concrete Type

Tabelle für Parent (gemeinsame Felder) und Tabellen für Childs (eigene Felder) mit Verweis auf Parent. Nachteil: Duplicate columns

- **Tabelle "Products"**: Id, Name, Description, Price
- **Tabelle "MobilePhones"**: Id, Name, Description, Price, OperatingSystem, SupportsUmts
- **Tabelle "Tablets"**: Id, Name, Description, Price, HasKeyboard

## 14.6 Database Migrations

Während Entwicklung: Modell anpassen, Migration erstellen, Review der Migration, eventuelle Korrekturen anbringen

Deployment: Änderungen gemäss Migration-Reihenfolge auf Datenbank deployen, Rollback auf älteren Stand via Down-Migration möglich

**Migration erstellen** dotnet ef migrations add InitialCreate -> DB wurde noch nicht erstellt

**Datenbank Deployment** dotnet ef database update -> DB wurde erstellt

```
1 using(var context = new AngProjContext()) {
2     var database = context.Database;
3     //"Dev" Ansatz (loeschen / neu erstellen)
4     database.EnsureDeleted(); //Loescht DB
5     database.EnsureCreated(); //Erstellt DB falls nicht vorhanden
6
7     //Automatische Migration auf neuesten Stand
8     database.Migrate(); //Migration DB zur neusten Version
9
10    //Migrations abfragen
11    IEnumerable<string> migrations;
12    migrations = database.GetMigrations(); //Abfrage von Migration-Names im
    DbContext
13    migrations = database.GetPendingMigrations();
14    migrations = database.GetAppliedMigrations();
15    var m = context.GetService<IMigrator>();
16    m.Migrate("<MigrationName>"); //Explizite Migration auf spezifische
    Version
17 }
```

## 15 Google Remote Procedure Call (gRPC)

Der neue Standard-Technologie für die Backend-Kommunikation in .NET. Primär Server-to-Server Kommunikation im Fokus (Microservices). Hohe Performance von zentraler Bedeutung. Nicht als Frontend-API gedacht. Ist der Ersatz für Windows Communication Foundation. . Kommunikation über HTTP/2 (Multiplexing (mehrere gRPC Calls pro TCP/IP Connection), Bidirectionales Streaming, Parallele Requests und Responses in einer einzigen TCP Verbindung, Kommunikation wegen HTTPS verschlüsselt).

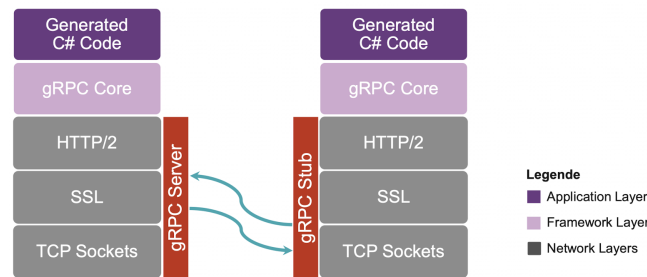


Figure 16: Kommunikationsstack

**Kommunikationsprotokoll** HTTP/2

**Interface Definition Language** Google Protocol Buffers

### 15.1 Architektur

gRPC ist ein SDK und kann in verschiedene IDEs integriert werden.

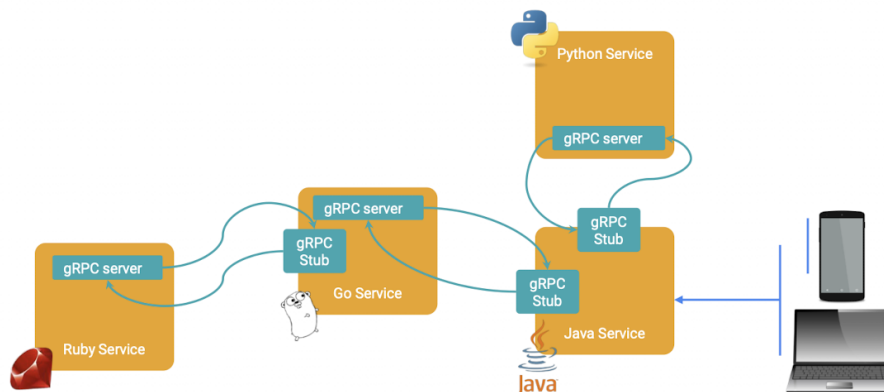


Figure 17: Architekturbeispiel

**HTTP/2 Features**

**Multiplexing** Mehrere gRPC Calls pro TCP/IP Session

**Bidirectional Streaming** Asynchronis, nicht-blockierendes Senden und Empfangen von Streams

**HTTPS** HTTP/2 und gRPC basieren voll auf HTTPS → Kommunikation ist immer verschlüsselt

### 15.2 Protocol Buffers

- Interface Definition Language (IDL)
  - Eine Subform einer Domain Specific Language (DSL)
  - Beschreibt ein Service Interface plattform- und sprachneutral.
- Data Model
  - Beschreibt Messages resp. Request- und Response-Objekte
- Wire Format
  - Beschreibt das Binärformat zu Übertragung

- Serialisierung- und Deserialisierungsmechanismen
- Service-Versionierung

### Proto Files

Datei-Endung "\*.proto". Service-Methoden haben immer einen Parameter und einen Rückgabewert. 1 oder mehr Services und 1 oder mehr Service-Methoden pro Service. 1 oder mehr Message Type Fields. Service Methoden haben immer 1 Parameter und 1 Rückgabewert.

### Messages

Angabe der Feldtypen: Skalarer Typ, Anderer Message Type, Enumeration. Unique Field Name und Unique Field Number.

.proto	C#	Default	Beschreibung
double	double	0	
float	float	0	
int32	int	0	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.
int64	long	0	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.
bool	bool	false	
string	string	<empty>	A string must always contain UTF-8 encoded or 7-bit ASCII text.
bytes	ByteString	<empty>	May contain any arbitrary sequence of bytes.
[...]	Weitere Ganzzahltypen wie [u s][int long][32 64] oder fixed[32 64] <a href="https://developers.google.com/protocol-buffers/docs/proto3#scalar">https://developers.google.com/protocol-buffers/docs/proto3#scalar</a>		

Figure 18: Felder Datentypen

### Proto Files

```

1 syntax = "proto3";
2 option csharp_namespace = "_01_BasicExample";
3 package Greet;
4
5 // The greeting service definition.
6 service Greeter {
7   rpc SayHello (HelloRequest) returns (HelloReply);
8 }
9
10 // The request message containing the user's name.
11 message HelloRequest {
12   string name = 1;
13 }
14 // The response message containing the greetings.
15 message HelloReply {
16   string message = 1;
17 }
```

### Fields/Repeated Fields

- Angabe des Feldtypen Skalarer Werttyp, andere Message Type, Enumeration
- Unique Field Name Wird für Generatoren verwendet, Lower Snake Case (underscores)
- Unique Field Number Identifikator für das Binärformat.

```

1 message SearchRequest {
2   string query = 1;
3   int32 page_number = 2;
4   int32 result_per_page = 3;
5 }
6 message SearchResponse {
7   repeated string results = 1; // Repeated = Liste
8 }
```

### Enumerations

Von der Idee her analog zu Enumerationstypen (enum) in .NET. Enum-Member mit dem Wert 0 muss zwingend existieren.

```
1 message SearchRequest {
2   Color searchColor = 1;
3   Size searchSize = 2;
4   enum Color {
5     RED = 0; // 0 must exist
6     GREEN = 1;
7   }
8 }
9 enum Size {
10  S = 0; // 0 must exist
11  M = 1;
12  L = 2;
13 }
```

### Reserved Fields

Für Versionierung gedacht. Wiederverwendung wird vom Protocol Buffer Compiler verhindert.

```
1 message SearchRequest {
2   reserved 1, 3, 20 to 30;
3   reserved "page_number", "result_per_page";
4   string query = 1; // Compilerfehler
5   int32 page_number = 2; // Compilerfehler
6   int32 result_per_page = 3; // Compilerfehler
7 }
```

## 15.3 gRPC # API

### Protocol Buffer Compiler

- protoc.exe mit Plugin für C# Code Generierung
- Automatische in Build Pipeline eingebunden
- NuGet Package: grpc.Tools
- Proto-Compiler Output in ".obj" order

## 15.4 Beispiel Customer Service

Zwei Services Customer und Order Service.

### Customer Service

```
1 service CustomerService {
2   rpc GetCustomers (google.protobuf.Empty)
3     returns (GetCustomersResponse);
4   rpc GetCustomer (GetCustomerRequest)
5     returns (GetCustomerResponse);
6 }
```

### Order Service

```
1 service OrderService {
2   rpc GetOrders (GetOrdersRequest)
3     returns (GetOrdersResponse);
4 }
```

### Service Implementation

```
1 //Customer Service
```

```

2 public class MyCustomerService : CustomerService.CustomerServiceBase {
3     public override async Task<GetCustomersResponse> GetCustomers(Empty
      request, ServerCallContext context) { /* ... */ }
4     public override async Task<GetCustomerResponse> GetCustomer(
      GetCustomerRequest request, ServerCallContext context) { /* ... */ }
5 }
6
7 //Order Service
8 public class MyOrderService : OrderService.OrderServiceBase {
9     public override async Task<GetOrdersResponse> GetOrders(GetOrdersRequest
      request, ServerCallContext context) { /* ... */ }
10 }

```

### Client-Implementation (Customer)

```

1 // The port number (5001) must match the port of the gRPC server.
2 GrpcChannel channel = GrpcChannel.ForAddress("https://localhost:5001");
3
4 // Customer service calls
5 var customerClient = new CustomerService.CustomerServiceClient(channel);
6
7 var request1 = new Empty();
8 GetCustomersResponse response1 = await customerClient.GetCustomersAsync(
      request1);
9 Console.WriteLine(response1);
10
11 var request2 = new GetCustomerRequest { IdFilter = 1 };
12 GetCustomerResponse response2 = await customerClient.GetCustomerAsync(
      request2);
13 Console.WriteLine(response2);
14
15 request2.IncludeOrders = false;
16 response2 = await customerClient.GetCustomerAsync(request2);
17 Console.WriteLine(response2);

```

### Client-Implementation (Order)

```

1 // The port number (5001) must match the port of the gRPC server.
2 GrpcChannel channel = GrpcChannel.ForAddress("https://localhost:5001");
3
4 // Order service calls
5 var orderClient = new OrderService.OrderServiceClient(channel);
6
7 var request3 = new GetOrdersRequest { CustomerIdFilter = 1 };
8 GetOrdersResponse response3 = await orderClient.GetOrdersAsync(request3);
9 Console.WriteLine(response3);

```

## 15.5 Streams

Es werden **drei Modi** unterstützt: Server Streaming (Server - Client), Client Streaming (Client - Server), Bidirectional / Duplex Streaming Call. Es werden zwei verschiedene Modi von Lesen unterstützt: Synchrones und asynchrones Lesen. **Reliability** End-to-End Reliability (garantiertes Ausliefern der Nachrichten gewährleistet), Ordered Delivery (Reihenfolge gewährleistet).

```

1 service FileStreamingService {
2     rpc ReadFiles (google.protobuf.Empty)
3         returns (stream FileDto);
4     rpc SendFiles (stream FileDto)
5         returns (google.protobuf.Empty);
6     rpc RoundtripFiles (stream FileDto)
7         returns (stream FileDto);
8 }

```

```

9 message FileDto {
10     string file_name = 1;
11     int32 line = 2;
12     string content = 3;
13 }

```

## 15.6 Exception Handling

Grundsätzlich immer via "RpcException" (basierend auf StatusCodes).

```

1 public class RpcException : Exception {
2     public RpcException(Status status);
3     public RpcException(Status status, string message);
4     public RpcException(Status status, Metadata trailers);
5     public RpcException(Status status, Metadata trailers, string message);
6     public Status Status { get; }
7     public StatusCode StatusCode { get; }
8     public Metadata Trailers { get; }
9 }

```

Status Code	Beschreibung
OK	Kein Fehler, alles okay.
Cancelled	Operation wurde abgebrochen (meist durch Client).
Unknown	Fehler kann nicht ermittelt werden / ist unbekannt. Default wenn Exception nicht behandelt wird.
InvalidArgument	Ungültige Argumente beim Aufruf angegeben.
DeadlineExceeded	Deadline überschritten bevor Anfrage erfolgreich beendet wurde.
NotFound	Angefragte Ressource wurde nicht gefunden.
AlreadyExists	Zu erstellende Ressource existiert bereits.
PermissionDenied	Aufrufer hat keine Berechtigung um die Operation auszuführen.
Unauthenticated	Aufrufer ist nicht authentifiziert (angemeldet).
ResourceExhausted	Eine Ressource ist aufgebraucht (Per-User-Quota / Speicherplatz / etc.)

Status Code	Beschreibung
FailedPrecondition	Vorbedingungen sind falsch (ungültiger Systemstatus, keine Datenbankverbindung, Bestellung abschliessen obwohl bereits abgeschlossen).
Aborted	Anfrage wurde abgebrochen (Concurrency Issues, Transaktionsabbruch, etc.).
OutOfRange	Ungültiger Range bei Anfrage (Geburtsdatum in der Zukunft, etc.).
Unimplemented	Methode wurde (noch) nicht implementiert.
Internal	Allgemeiner interner Serverfehler
Unavailable	Service ist aktuell nicht verfügbar.
DataLoss	Datenverlust oder korrupte Daten.

### Unbehandelte Exception

Exception wird auf Server nicht gefangen (Server Runtime fängt Exception, Wirft RpcException)

```

1 public override async Task<Empty> Unhandled(Empty request, ServerCallContext
    context) {
2     throw new Exception("Unhandled Exception");
3 }

```

### Behandelte Exception mit Trailers

Exception wird auf Server gefangen und korrekt verpackt. Metadata-Klassen verwenden, Key-Value-Pair-Liste.

```

1 public override async Task<Empty> Trailers(Empty request, ServerCallContext
    context) {
2     throw new RpcException(new Status(StatusCode.NotFound, "Something was
    not found."),
3         new Metadata {
4             { "error-details", "Here are some more details..." },
5             { "error-obj" + Metadata.BinaryHeaderSuffix, Encoding.UTF8.
    GetBytes("...payload...") }
6         }
7     );
8 }

```



## 16 Reflection

Unter Reflection versteht man die Analyse von Metadaten eines Objekts zur Laufzeit. Mit Reflection lassen sich Typen suchen und instanzieren. Die abstrakte Basisklasse `System.Type` repräsentiert einen Typen. `System.RuntimeType` erbt von `System.Type`. Mit Reflection können auch `private` Felder gelesen und geschrieben werden.

### 16.1 Type-Discovery

Suche aller Typen in einem Assembly.

```

1 Assembly a01 = Assembly.Load("mscorlib");
2
3 Type[] t01 = a01.GetTypes();
4 foreach (Type type in t01)
5 {
6     Console.WriteLine(type);
7     MemberInfo[] mInfos = type.GetMembers();
8     foreach (var mi in mInfos)
9     {
10         Console.WriteLine(
11             "\t{0}\t{1}",
12             mi.MemberType,
13             mi);
14     }
15 }
```

### 16.2 Member auslesen

```

1 Type type = typeof(Counter);
2 MemberInfo[] miAll = type.GetMembers();
3 foreach (MemberInfo mi in miAll)
4 {
5     Console.WriteLine(
6         "{0} is a {1}",
7         mi, mi.MemberType);
8 }
9 Console.WriteLine("-----");
10 PropertyInfo[] piAll = type.GetProperties();
11 foreach (PropertyInfo pi in piAll)
12 {
13     Console.WriteLine(
14         "{0} is a {1}",
15         pi, pi.PropertyType);
16 }
17
18 // Filter members according to BindingFlags or FilterName
19 Type type = typeof(Assembly); BindingFlags bf =
20     BindingFlags.Public |
21     BindingFlags.Static |
22     BindingFlags.NonPublic |
23     BindingFlags.Instance |
24     BindingFlags.DeclaredOnly;
25
26 System.Reflection.MemberInfo[] miFound = type.FindMembers(
27     MemberTypes.Method, bf, Type.FilterName, "Get*"
28 );
```

### 16.3 Field Information

Die Field Info beschreibt ein Feld einer Klasse (Name, Typ, Sichtbarkeit). Die Felder können mit `object GetValue(object obj)` und `void SetValue(object obj, object value)` auch gelesen und geschrieben werden.

### 16.4 Property Info

Die Property Info beschreibt eine Property einer Klasse (Name, Typ, Sichtbarkeit, Informationen zu Get/Set). Auch Properties lassen sich lesen und schreiben.

### 16.5 Method Info

Die Method Info beschreibt eine Methode einer Klasse (Name, Parameter, Rückgabewert, Sichtbarkeit). Sie leitet von Klasse `MethodInfo` ab. Die Methode wird mit `Invoke()` aufgerufen.

### 16.6 Constructor Info

Die Constructor Info beschreibt ein Konstruktor einer Klasse (Name, Parameter, Sichtbarkeit). Wie `MethodInfo` leitet er wegen seinen ähnlichen Eigenschaften von `MethodInfo` ab und wird mit `Invoke()` aufgerufen.

## 17 Attributes

Attribute bieten die Möglichkeit, Informationen in deklarativer Form mit Code zu verknüpfen. Attribute können außerdem als wiederverwendbare Elemente genutzt werden, die auf eine Vielzahl von Zielen angewendet werden können.

### 17.1 Anwendungsfälle

- Object-relationales Mapping
- Serialisierung (WCF, XML)
- Security und Zugriffssteuerung
- Dokumentation

### 17.2 Custom Attributes