.NET Technologien | MsTe

Zusammenfassung

INHALTSVERZEICHNIS

1. ÜBERBLICK & ARCHITEKTUR

1.1. VERGLEICH .NET, .NET CORE UND .NET FRAMEWORK

.NET Framework (2002-2019)	*.NET Core (2016-2019)*	*.NET (ab 2020)*
Für Windows entwickelt & eng mit OS verzahnt, letzte Version 4.8 erhält nur noch Security-Updates *(kein End-of-Life)*	Cross-Platform-Implementation welche neben .NET FW entwickelt wurde, limitierte Anzahl Features im Vergleich zu .NET FW, keine Updates mehr	Vereint .NET Framework & .NET Core-Features, jedes Jahr eine neue Version *(18 Monate Support)*, jedes zweite Jahr eine LTS-Version *(3 Jahre Support)*

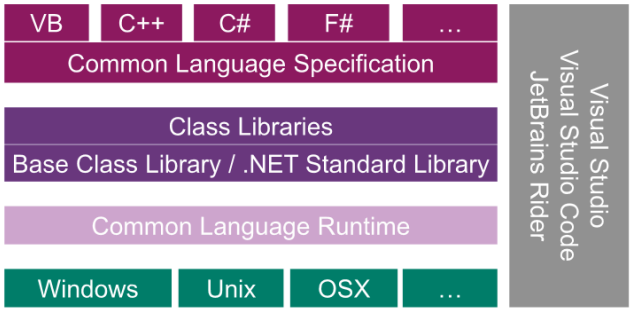
1.2. .NET PLATTFORM GRUNDSTRUKTUR

- ***Common Language Runtime (CLR):*** Mächtige Laufzeitumgebung für verschiedene Sprachen, ähnlich Java Virtual Machine.
  - **Common Type System (CTS):** Gemeinsames Typensystem für alle .NET-Sprachen.
  - **Common Language Specification (CLS):** Gemeinsame Sprach-Eigenschaften für alle .NET-Sprachen.
- ***.NET Base Class Library (BCL):*** Basis-Klassen für alle .NET-Sprachen
  - **ADO.NET / Entity Framework Core:** Klassen für DB-Zugriff
  - **ASP.NET Core:** Web-Programmierung
  - Umfangreiche Klassen für XML, JSON, Zugriff auf Dateisystem
  - **Windows Presentation Foundation (WPF) / Windows Forms:** Klassen für Windows-GUIs

1.3. COMMON LANGUAGE RUNTIME (CLR)

Laufzeitumgebung für .NET-Code *(«managed code»)*. Umfasst:

- JIT-Compiler *(Intermediate Language Code zu Maschinencode)*
- Class Loader *(für das Laden von Klassen-Code zur Laufzeit)*
- Speicherverwaltung / Garbage Collection
- Sprachübergreifendes Debugging
- Exception Handling
- Type Checking
- IL Code Verification
- Thread Management
- Base Class Library



1.3.1. Common Intermediate Language (CIL)

Ist eine ***vorkompilierte Zwischensprache***. Prozessor-unabhängig, Assembler-ähnlich, Sprach-unabhängig. CLS-kompatible Bibliotheken können von allen .NET-Sprachen verwendet werden.

Umfang: Virtuelle Stack-Maschine ohne Register, Vererbung / Polymorphie und Common Type System (CTS) für komplexe Datentypen / Objekte und Boxing / Unboxing. Früher Microsoft Intermediate Language (MSIL) genannt.

Vorteile

- ***Portabilität*** *(auf andere OS und Prozessorarchitekturen)*
- ***Typsicherheit*** *(Beim Laden des Codes können Security-Checks durchgeführt werden)*

Nachteile

- ***Laufzeiteffizienzverlust*** *(Kann durch JIT-Compiler oder direkte Übersetzung auf eine Plattform wettgemacht werden*

Build-Prozess:

Hauptsprache-Sourcecode *(z.B. C#)* → Hauptsprache-Compiler → IL-Code → JIT-Compiler → Native Code
Erst ab dem ***Programm-Start***, das heisst zwischen dem IL-Code und dem JIT-Compiler, wird das Programm ***Plattform-abhängig***.

Build produziert ein ***Assembly*** *(*.dll oder *.exe)* und ein ***Symbol-File*** *(*.pdb - Programm Database für Debugging-Zwecke)* pro Projekt.

Cross-Language Development

Objekt-Modell und Bibliotheken sind in Plattform integriert. Die ***Sprachwahl ist sekundär***, ***Konzepte sind allgemeingültig***. Die ***Common Language Specification (CLS)*** bietet allgemeine ***Regeln*** für Cross-Language Development. Debugging wird von allen Sprachen unterstützt, auch Cross-Language Debugging möglich. Aktuell umfasst .NET ca. 30 Sprachen *(C#, F#, VB.NET, C++, J#, IronPython, IronRuby, …)*.

1.3.2. Kompilierung

Der ***Source Code*** wird während der Design Time *(Build-Prozess)* mit dem ***Language Compiler*** in den ***IL-Code*** umgewandet. Dieser wird dann mit dem ***JIT-Compiler*** während der Runtime in ***Native Code*** *(Assembler-Code der Platform)* übersetzt. Alternativ kann der Source Code auch ***direkt via Native AOT*** *(Ahead-of-time compilation)* in Native Code übersetzt werden.

Just-in-Time (JIT) Kompilierung

Nach der Kompilierung *(ohne Native AOT)* liegen die Methoden im Assembly als IL-Code vor. Beim Aufruf einer Methode wird erkannt, dass dieser Code noch nicht ersetzt wurde und der JIT-Compiler ersetzt diese Methode an dieser physischen Stelle im RAM den IL- durch Assembler-Code. Der nächste Aufruf erfolgt direkt auf den *(gecachten)* Assembler-Code.

1.3.3. Assemblies

Die ***Kompilation*** erzeugt Assemblies. Diese entsprechen ungefähr einem JAR-File in Java.
(Assembly = selbstbeschreibende Komponente mit definierter Schnittstelle.)

- Deployment- und Ausführungs-Einheit
- Executable oder Library *(*.exe bzw. *.dll)*
- Dynamisch ladbar
- Selbstbeschreibende Software-Komponente *(enthält Metadaten)*
- Definiert Typ-Scope *(Sichtbarkeit)*
- Kleinste versionierte Einheit
- Einheit für Security-Überprüfung *(Code Access Security / Role-Based Security)*

Ein ***Assembly*** besteht aus dem ***Manifest*** *(Header-Informationen wie Referenzen auf alle Dateien des Assemblies, Referenzen auf andere Assemblies, Metadaten wie Name & Versionsnummer etc.)*, $0 - n$ ***Modulen*** und $0 - n$ ***Ressourcen*** *(Bilder, Übersetzungsdateien etc.)*. Ein ***Modul*** enthält $0 - n$ Typen *(Klassen, Interfaces etc.)*, die wiederum aus ***CIL-Code und ihren Metadaten*** bestehen *(Informationen der Signatur wie z.B. Sichtbarkeit, Abstrakt, Statisch etc.)*

Modules & Metadata

Kompilation erzeugt ein Modul mit ***CIL-Code und Metadaten***. Die ***Metadaten*** beschreiben verschiedene ***Attribute*** des Codes *(Sichtbarkeit, Typ, Name, Funktionen, Parameter etc.)*. Die ***Programmlogik*** steckt im ***CIL-Code*** *(Klassen-Definitionen, Methoden-Definitionen, Feld-Definitionen usw)*. Die Metadaten lassen sich über ***.NET Reflection*** abfragen (siehe «Reflection» (Seite 57)).

1.4. COMMON TYPE SYSTEM (CTS)

CLR hat ein integriertes, einheitliches Typen-System. Diese Typen sind im ***Laufzeitsystem definiert***, nicht in Programmiersprache. Das Typensystem ist ***single-rooted***, d.h. alle Typen sind von `System.Object` abgeleitet. Es gibt 2 Kategorien, ***Reference- und Value-Typen***. Auch wird ***Boxing/Unboxing*** unterstützt.

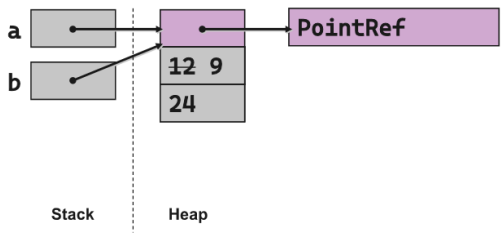
Reflection: Programmatisches Abfragen des Typensystems. Ist für alle Typen verfügbar *(ausser Security-Einschränkungen vorhanden)*, erweiterbar über «Custom Attributes» (siehe «Custom Attribute» (Seite 59)).

1.4.1. Reference- & Value Types

	Reference (`class`, Objekte)	***Value (`struct`, Primitive Typen)***
Speicherort	Heap	Stack
Variable enthält	Objekt-Referenz	Wert
Nullwerte	Möglich	Nie
Default value	`null`	`0 | false | '\0'`
Zuweisung / Methodenaufruf	Kopiert Referenz	Kopiert Wert
Ableitung möglich *(Vererbbarkeit)*	Ja	Nein *(sealed)*
Garbage Collected	Ja	Nicht benötigt

Reference Types

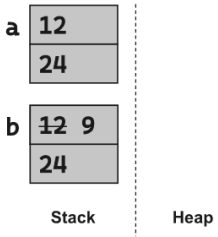
Zuweisung: Objekt-Referenz wird kopiert




```
class PointRef { public int X, Y; }
PointRef a = new PointRef();
a.X = 12; a.Y = 24;
PointRef b = a;
b.X = 9;
Console.WriteLine(a.X); // Prints 9
Console.WriteLine(b.X); // Prints 9
```

Value Types

Zuweisung: Wert wird kopiert



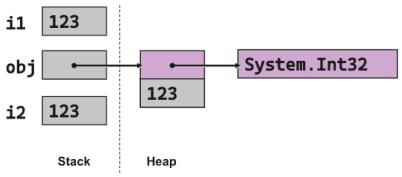

```
struct PointVal { public int X, Y; }
PointVal a = new PointVal();
a.X = 12; a.Y = 24;
PointVal b = a;
b.X = 9;
Console.WriteLine(a.X); // Prints 12
Console.WriteLine(b.X); // Prints 9
```

1.4.2. Boxing / Unboxing

Polymorphe Behandlung von Value- und Reference-Types. Üblicherweise mit `object` als «Box» durchgeführt.

- ***Boxing:*** Kopiert Value Type in einen Reference Type. Der dazugehörige Value Type wird mit in den Reference Type gespeichert *(implizite Konvertierung, upcast)*.
- ***Unboxing:*** Kopiert Reference Type in einen Value Type *(explizite Konversion, downcast)*.

```
System.Int32 i1 = 123;
System.Object obj = i1; // Boxing
System.Int32 i2 = (System.Int32)obj; // Unboxing
```



1.5. COMMAND LINE INTERFACE CLI

Komplexe Command Line Tool-Chain «`dotnet.exe`». Ist Teil des .NET (Core) SDK, Basis für high-level Tools.

Command-Struktur: `dotnet[.exe] <Verb> <argument> --<option> <param>`

- ***`<verb>`:*** Auszuführende Aktion *(build, run, etc.)*
- ***`<arg>`:*** Argument für vorangehendes Verb
- ***`<option>`:*** Option / Switch Parameter. Mehrere Optionen möglich
- ***`<param>`:*** Parameter zur Option. Nicht zwingend.

Beispiele: `dotnet publish my_app.csproj` oder `dotnet build --output /build_output`

1.6. PROJEKTE & REFERENZEN

1.6.1. Projekt-Dateien

Im XML Format mit `.csproj` Endung. ***Build-Engines:*** Microsoft Build Engine «MSBuild» oder .NET Core CLI (`dotnet build`). Einfache, dynamische Grobstruktur *(Property*: Projekteinstellungen, Item*: Zu kompilierende Items, Target*: Sequenz auszuführender Schritte)*

1.6.2. Referenzen

- ***Vorkompiliertes Assembly*** *(Im File System, Debugging nicht verfügbar, Navigation nur auf Metadaten-Ebene)*
- ***SDK-Referenz*** *(z.B. verwendete .NET-Version, Zwingend)*
- ***NuGet Package*** *(Externe Dependency, Debugging nicht verfügbar, Navigation nur auf Metadaten-Ebene)*
- ***Visual Studio Projekt:*** *(In gleicher Solution vorhanden, Debugging und Navigation verfügbar)*

1.7. PACKAGES & NUGET

NuGet ist der neue Standard für Packaging. .NET wird neu in ***kleineren Packages*** geliefert. ***Vorteile:*** Erlaubt Release-Zyklen unabhängig von .NET/Sprachreleases, Erhöht Kompatibilität durch Kapselung, kleinere Deployment-Einheiten.

- ***Entwicklungsprozess:*** Create Project, Create Manifest, Compile Project, Create Package, Publish Package, Consume Package
- ***Deployment:*** Lokales NuGet Repository auf Entwicklungsrechner, Self-hosted NuGet Repository oder Hosted NuGet Repository.

2. C# GRUNDLAGEN

Naming Guidelines

Element	*Casing*	*Beispiel*

Namespace
Klasse / Struct
Interface
Enum
Delegates
	PascalCase *(erster Buchstabe gross)*
	Substantive
		`System.Collections.Generic`
		`BackColor`
		`IComparable`
		`Color`
		`Action / Func`

Methoden	PascalCase, Aktiv-Verben / Substantive	`GetDataRow`, `UpdateOrder`

Felder *(mit Underscore Prefix)*
Lokale Variablen
Parameter
	CamelCase *(erster Buchstabe klein)*	`_name`
		`orderId`
		`orderId`

Properties Events	PascalCase	`OrderId` `MouseClick`

2.1. 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, der abgeleiteter Klasse oder des jeweiligen Assemblies sichtbar
`private protected`	Innerhalb des jeweiligen Typen oder abgeleiteter Klasse sichtbar, wenn diese im gleichen Assembly ist

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`	-	-

2.2. NAMESPACES

Entspricht in Java dem «Package». Adressiert via «Classpath». ***Strukturiert*** den Quellcode. Ist ***hierarchisch*** aufgebaut und ***nicht*** an physikalische Strukturen gebunden *(wie z.B. Ordnerstruktur)*.
Beinhaltet andere Namespaces, Klassen, Interfaces, Structs, Enums und Delegates.

Es sind ***mehrere*** Namespaces in einem File möglich und ein Namespace kann in ***mehreren*** Files definiert sein. Namespace und Ordnerstruktur können sich ***unterscheiden***.

```
namespace A { class C {} }
namespace B {}
```

Namespaces werden in andere Namespaces ***importiert*** mit `using System`.
Es sind auch Alias-Namen möglich: `using F = System.Windows.Forms ... F.Button b;`

2.2.1. File-Scoped Namespaces

Erlaubt das Entfernen von {} nach Deklaration des Namespaces. Reduziert ***Einrückung*** des Codes. Dann ist aber nur ***ein Namespace pro File*** erlaubt.

```
// File1.cs "Klassisch"
namespace OstDemo
{
  class X {}
}
```

```
// File1.cs "File-Scoped Namespace"
namespace OstDemo;
class X {}
namespace OstDemo2 {} // Compiler-Fehler
```

2.2.2. Global `using` directives

Erlaubt «globale» Deklaration von usings. Gelten für das ***ganze*** Projekt, verkleinert Boilerplate Code im Header. Das `using`-Statement in den einzelnen Files kann dann weggelassen werden.

Deklarationsmöglichkeiten:
– C# Direktive, meist in Datei `GlobalUsings.cs`, z.B. `global using Azure.Core;`
– MSBuild `/*.csproj` Datei z.B. `<ItemGroup><Using Include="Azure.Core" /></ItemGroup>`

Implicit global using directives

Vordefinierte Liste globaler «usings», bei Projektgenerierung vom Compiler erstellt. Muss im `.csproj` mit `<ImplicitUsings>enable</ImplicitUsings>` aktiviert werden. Welche Usings verwendet werden, ist abhängig von gewählten SDK.

2.3. MAIN-METHODE

Ist der ***Einstiegspunkt*** eines Programms. Ist zwingend für Executables und klassischerweise genau 1x erlaubt. Das Programm ***beginnt*** mit der ersten Anweisung in der Main-Methode und ***endet*** mit der letzten Anweisung in der Main-Methode. Befindet sich meist in der Datei `Program.cs`.

2.3.1. Anforderungen

- ***Sichtbarkeit*** der Methode und beinhaltender Klasse ***nicht relevant***
- Die Main-Methode muss ***`static`*** sein, beinhaltende Klasse nicht
- ***Gültige Rückgabetypen:*** `void`, `int`, `Task`, `Task<int>`
- ***Gültige Parametertypen:*** Keine Parameter oder `string[]`

```
// Examples (some missing for brevity)
static void Main() { }
static void Main(string[] args) { }
static int Main(string[] args) { }
static async Task<int> Main() { }
```

2.3.2. Argumente

- Diverse Möglichkeiten für Zugriff
  - Über einen `string[]`-Parameter
  - Ohne `string[]`-Parameter über statische Methode: `System.Environment.GetCommandLineArgs();`
- Können beim Aufruf mit Space getrennt angegeben werden
- Parsen mit NuGet Package `System.CommandLine` empfohlen

```
class ProgramArgs {
  static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
      Console.WriteLine(
        $"Arg {i} = {args[i]}"); }
```

```
> MyApp.exe alpha beta gamma
    Arg 0 = alpha
    Arg 1 = beta
    Arg 2 = gamma
```

2.3.3. Top-level Statements

- Erlaubt ***Weglassen der Main-Methode*** als Entry Point.
- ***Regeln:*** Nur 1x pro Assembly erlaubt, Argumente heissen fix `args`, Exit Codes sind erlaubt, ***vor*** den top-level Statements

können `usings` definiert werden, ***nach*** den top-level Statements können Typen/Klassen definiert werden.

```
using System; // main() code start
for (int i = 0; i < args.Length, i++) {
  ConsoleWriter.Write(args, i);
} // main() code end
class ConsoleWriter {
  public static void Write(string[] args, int i) {
    console.WriteLine($"Arg {i} = {args[i]}");
  }
}
```

2.4. ENUMERATIONSTYPES: ENUM

Liste ***vordefinierter Konstanten*** inklusive Wert. Ist standardmässig `int`, kann aber mit jedem Integertyp ersetzt werden, um Wertebereich/Speichernutzung anzupassen *(byte, sbyte, short, ushort, uint, long, ulong)*. Index beginnt bei 0.

Deklaration

```
enum Days { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };

enum Days { Sunday = 10, Monday, Tuesday, Wednesday, Thursday, Friday = 9, Saturday };
/* Indexe: 10, 11, 12, 13, 14, 9, 10 */
```

Duplikate in den Werten sind erlaubt. Bei Cast in ein Enum wird aber immer die erste Definition verwendet. Im Beispiel oben ist also `(Days)10` immer `Sunday`, nie `Saturday`.

Verwendung

```
Days today = Days.Monday; if (today == Days.Monday) { /* ... */ }
```

Werte auslesen

```
int sundayValue = (int)Days.Sunday; Console.WriteLine("{0} / #{1}", Days.Sunday, sundayValue);
// Output: Sunday / #10
foreach (string name in Enum.GetNames(typeof(Days))) { Console.WriteLine(name); }
// Output: Monday\nTuesday\nWednesday ...
```

String zu Enum parsen

```
Days day1 = (Days)Enum.Parse(typeof(Days), "Monday") // Non-Generic, Exception on failure

Days day2;
bool success2 = Enum.TryParse("Monday", out day2); // Generic, variable already defined
bool success3 = Enum.TryParse("Monday", out Days day3) // Generic, initializes new variable
```

2.5. OBJECT

Basisklasse aller Typen. Einer `object`-Variable kann jeder Typ zugewiesen werden, siehe «Boxing / Unboxing» (Seite 4). Methoden mit `object`-Parameter können ebenfalls alle Typen annehmen, erlaubt dynamische Methoden.

```
public class Object {
  public Object() { };
  // Compares references of the objects, can be overridden to compare values
  public virtual bool Equals(object obj);
  public static bool Equals(object objA, object objB);
  public virtual int GetHashCode();
  public Type GetType();
  // Creates a shallow copy of the current Object, references in the object stay the same
  protected object MemberwiseClone();
  // Compares references of the objects, can't be overridden
  public static bool ReferenceEquals(object objA, object objB);
  public virtual string ToString();
}
```

2.6. STRINGS

```
string s1 = "Test";
```

Ein `string` ist ein ***reference type*** und ***nicht modifizierbar*** *(Modifizierung wird in einen Aufruf von `System.String.Concat(...)` umgewandelt, ein neuer `string` wird kreiert)*. Eine ***Verkettung*** ist mit dem + Operator möglich und ein ***Wertevergleich*** mit == oder `equals()`. Ein C# `string` ist ***nicht*** mit `\0` terminiert. ***Indexierung*** ist möglich, Die Länge wird mit dem `.Length`-Property ermittelt.

2.6.1. String Interpolation

Mit einem $ vor dem String können Werte und Expressions innerhalb von `{...}` direkt in einen String evaluiert und eingefügt werden.

```
string s2 = $"{DateTime.Now}: {(DateTime.Now.Hour < 18 ? "Hello" : "Good Evening")}";
```

2.6.2. Raw String Literals

Mehrzeilige Strings ohne spezielle Behandlung des Inhalts *(keine Escape-Sequences, alle Characters sind «normaler Text»)*

- Deklariert mit ***mindestens 3 Double Quotes*** *(erlaubt mehrere aufeinanderfolgende Double Quotes im Raw String, muss immer mit 1 Double Quote mehr initialisiert werden, als im String beinhaltet)*
- ***Einrückung:*** Die Position der «closing quote» definiert das 0-te Level *(linker Rand)* des Raw Strings.
- ***Verwendung:*** Einbetten von strukturierten Text-Daten *(JSON, XML)*, Text bei welchem Whitespace-Formatierung relevant ist.

```
string s3 =
"""
{
  "Name": "Nina",
  "Profession": "WordPress Web Dev"
}
"" \ \\ ' ''
""";
```

2.6.3. Verbatim String Literals

Simplere Variante des Raw Strings.

- Deklariert mit ***`@"[string]"`***
- Double Quotes müssen mit einem zweiten Double Quote escaped werden
- ***Verwendung:*** Simple (Multiline-)Strings mit Escape-Chars, z.B. Windows-Dateipfade

```
string str;
// file "C:\sample.txt"
str = "file \"C:\\sample.txt\"";
str = "file \x0022C:\u005csample.txt\x0022";
str = @"file
""C:\sample.txt""";
// newline inside verbatim string is ignored
```

2.6.4. Vergleiche

Bei C# wird mit ==, != und `Equals()` der ***Inhalt*** der Strings verglichen.

```
string s1 = "Test"; string s2 = "Test";
bool result1 = s1.Equals(s2);                  // True
bool result2 = string.Equals(s1, s2);          // True
bool result3 = s1 == s2                        // True
```

Strings werden intern ***wiederverwendet***. Erst `string.Copy(...)` erzeugt eine echte Kopie.

```
bool result4 = string.ReferenceEquals(s1, s2); // True
string s3 = string.Copy(s1);
bool result5 = string.Equals(s1, s3);          // True
bool result6 = string.ReferenceEquals(s1, s3); // False
```

2.7. ARRAYS

Einfachste Datenstruktur für ***Listen***. Können ***eindimensional***, ***mehrdimensional*** und ***rechteckig*** oder ***ausgefranst/jagged*** sein. Die Länge ***aller Dimensionen*** ist bei der Instanzierung bekannt. ***Alle Werte*** sind nach der Instanzierung initialisiert *(`false`, `0`, `null`, etc.)*. Arrays sind ***zero-based*** und immer auf dem ***Heap***.

2.7.1. Eindimensionale Arrays

```
int[] array1 = new int[5];                  // Deklaration (Value Type) mit Länge 5
int[] array2 = new int[] { 1, 3, 5, 7, 9 }; // Deklaration & Wertedefinition
int[] array3 = int[] { 1, 2, 3, 4, 5, 6 };  // Vereinfachter Syntax ohne new
int[] array4 = { 1, 2, 3, 4, 5, 6 };        // Vereinfachter Syntax ohne new / Typ
object[] array5 = new object[5];            // Deklaration (Reference Type)
```

– ***Value Types:*** `int[] a = { 1, 3, 5 } // speichert Wert`
– ***Reference Types:*** `object[] a = new object[3]; a[1] = new object(); a[2] = 5; // speichert Referenz`

Length & Indexzugriff

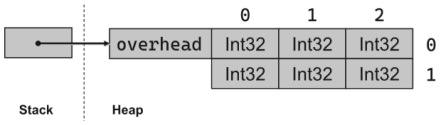
```
int length = array2.Length; // returns 5
int value1 = array2[4]; // returns 9
int value2 = array2[5]; // System.IndexOutOfRangeException
```

Vereinfachter Syntax

```
int[] array5 = { 1, 2, 3, 4, 5, 6 }     // Deklaration ohne new
array5 = new int[] { 1, 2, 3, 4, 5, 6 } // Zuweisung mit new type[] OK
array5 = new [] { 1, 2, 3, 4, 5, 6 }    // Zuweisung mit new[] OK
array5 = { 1, 2, 3, 4, 5, 6 };          // Compilerfehler, Zuweisung ohne new unzulässig
```

2.7.2. Mehrdimensionale Arrays (rechteckig)

```
int[,] a = new int[2,3];           // Deklaration
a[0, 1] = 9;                       // Schreiben
int x = a[0, 1];                   // Lesen - returns 9
int[,] b = { { 1, 2 }, { 4, 5 } }; // Deklaration & Wertedefinition
```

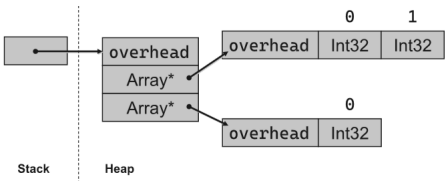


Length

```
int length = a.Length;        // returns 6
int length0 = a.GetLength(0); // returns 2 - Länge der 0. Dimension
int length1 = a.GetLength(1); // returns 3 - Länge der 1. Dimension
```

2.7.3. Mehrdimensionale Arrays (jagged)

```
int[][] a = new int[2][];  // Deklaration - "Liste" von Arrays
a[0] = new int[2];         // Wertedefinition
a[1] = new int[1];         // Wertedefinition
a[0][1] = 9;               // Schreiben
int x = a[0][1];           // Lesen - returns 9
```



Length

```
int length = a.Length;     // returns 2 - Länge der 0. Dimension
int length0 = a[0].Length; // returns 2 - Länge des ersten Array
int length1 = a[1].Length; // returns 1 - Länge des zweiten Arrays
```

2.7.4. Vorteile von Blockmatrizen

– ***Speicherplatz-Effizienz:*** Verbraucht weniger Speicher, da weniger Referenzen verwaltet werden müssen
– ***Schnelleres Allozieren:*** Speicher kann als gesamter Block alloziert werden, bei jagged Arrays sind Dimension $2 - n$ manuell zu allozieren
– ***Schnellere Garbage Collection:*** Weniger Verwaltungsaufwand weil nur 1 Array statt $n + 1$

Der ***Zugriff*** ist jedoch ***nicht schneller***, weil der ***Boundary-Check*** nur bei 1-dimensionalen Arrays optimiert wird.

2.8. SYMBOLE

2.8.1. Identifiers

Sind ***Case-sensitive***, Unicode kann verwendet werden. Wenn ein ***Schlüsselwort*** als Identifier verwendet werden soll, muss ein `@` vor das Schlüsselwort gestellt werden.

Syntax: `(letter | '_' | '@' ){ letter | digit | '_' }`

```
string someName; int sum_of3; int _10percent; int @while; double 🍆; double \u03c0; int f\u0061ck;
```

2.8.2. Schlüsselwörter

Im Vergleich zu anderen Sprachen hat C# relativ viele Schlüsselwörter. Viele sind aber kontextabhängig und werden beim Schreiben von «normalem» Code wenig verwendet.

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

2.8.3. Kommentare

- ***Single-Line:*** `// ...`
- ***Multi-Line:*** `/* ... */`
- ***Dokumentation von Methoden, Feldern, Properties:*** `/// <summary>...</summary> <returns>...</returns>`

2.9. PRIMITIVTYPEN

2.9.1. Ganzzahlen

- ***Ohne Suffix:*** Kleinster Typ aus `int | uint | long | ulong`
- ***Suffix `u | U`:*** Kleinster Typ aus `uint | ulong`
- ***Suffix `l | L`:*** Kleinster Typ aus `long | ulong`

Syntax

- ***Regulär:*** `digit{digit}{Suffix}`
- ***Hexadezimal:*** `"0x" hexDigit{hexDigit}{Suffix}`
- ***Binär:*** `"0b" [0|1]{[0|1]}{Suffix}`

```
object number;
number = 17;          // int
number = 9876543210; // long
number = 17L;         // long
number = 17u;         // uint
number = 0x3e;        // int
number = 0x3eL;       // long
number = 0b11101011; // int
```

2.9.2. Fliesskommazahlen

- ***Ohne Suffix oder `d | D`:*** `double`
- ***Suffix `f | F`:*** `float`
- ***Suffix `m | M`:*** `decimal`

Syntax

```
[Digits] ["." [Digits]] [Exp] [Suffix]
Digits: digit {digit}
Exp: ("e" | "E") ["+" | "-"] [Digits]
Suffix: "f" | "F" | "d" | "D" | "m" | "M"
```

```
object number;
number = 3.14; // double
number = 1E-2; // double
number = .1;   // double
number = 10f;  // float
```

Lesbarkeit von numerischen Werten

"_" für ***bessere optische Strukturierung***. Hat keine eigentliche Funktion, funktioniert mit allen numerischen Werten. Ist überall erlaubt, ausser am Anfang oder Ende einer Zahl. Wird vom Compiler einfach entfernt.

```
object number;
number = 9_876_543_210; // 9876543219
number = 0x1000_0000;   // 0x10000000
number = 0b1110_1011;   // 0b11101011
number = 1_23_456789_0; // 1234567890
number = 10__0E-2_2     // 100E-22
```

Escape-Sequenzen

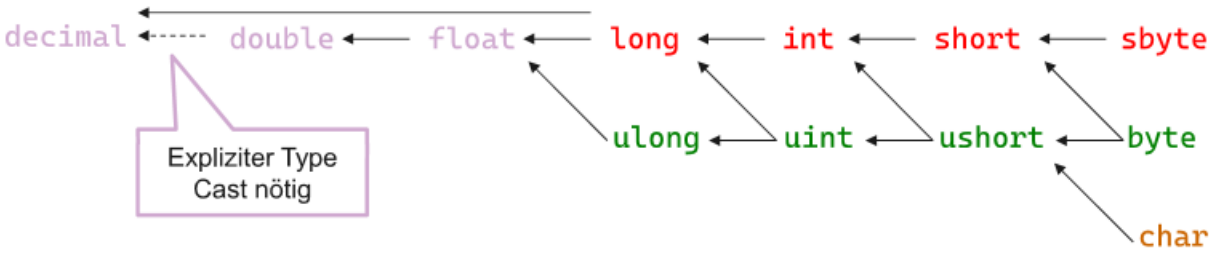
- `\'`, `\"`, `\\`: für den jeweiligen Char
- `\n`: newline
- `\t`, `\v`: horizontal/vertical tab

2.9.3. Zeichen / Zeichenketten

- ***String:*** `"{char}"` *(Nicht erlaubt: ", |end-of-line| und \)*
- ***Char:*** `'char'` *(Nicht erlaubt: ', |end-of-line| und \)*

```
string str;
// file "C:\sample.txt"
str = "file \"C:\\sample.txt\"";
```

2.9.4. Typkompatibilität



Typen mit validem Pfad werden implizit konvertiert, bei anderen ist ein expliziter Type Cast nötig

Übersicht einiger Beispiel-Casts. Links: Typ der Zielvariable, Rechts: Typ des Quellvariable

Erlaubt		***Nicht erlaubt ohne Cast***
`int = short;`	`decimal = (decimal)double;`	`short = int;`
`float = char;`	`byte = (byte)long;`	`char = int;`
`decimal = long;`	`uint = (uint)int;`	`char = float;`
`double = byte;`	`decimal = char; (!)`	`decimal = float;`

2.10. STATEMENTS

2.10.1. Switch case

Möglich mit Ganzzahlen *(`[s]byte, [u]short, [u]int [u]long`)*, `char`, `string` und `enum`. Die Cases sind ***fall-through***, ist ein Case also nicht mit `break`, `throw`, `return` oder `goto` abgeschlossen, wird der Code im nächsten Case ausgeführt, ohne dass die Case-Kondition nocheinmal geprüft wird.

Der `default`-Case ist optional und wird ausgeführt, wenn kein anderer Case zutrifft. `null`-case ist erlaubt.

```
string country = "Germany"; string lang;
switch (country) {
  case "Germany":
  case "Switzerland":
    lang = "German"; break;
  case null:
    Console.WriteLine("Country null"); break;
  default: Console.WriteLine("Error"); break;
}
```

2.10.2. Jumps

- ***`break`:*** Aktuellen Loop beenden
- ***`continue`:*** Zur nächsten Loop-Iteration *(z.B. nächstes Item in `foreach`)*
- ***`goto case`:*** Sprung zu Case innerhalb eines `switch`
- ***`goto label`:*** Sprung zum Label *(Keine Sprünge in Methoden hinein oder aus `finally`-Block heraus)*

3. KLASSEN & STRUCTS

3.1. STRUCTS

Structs sind ***Value Types***, d.h. sie sind auf dem ***Stack*** angelegt *(oder «in-line» in einem Objekt auf dem Heap → alle Struct-Objekte werden innerhalb vom enthaltenen Objekt angelegt und gespeichert)*

```
struct Point {
  int _x = 0;
  int _y = 0;

  public Point(int x, int y) { /* ... */ }
  public void MoveX(int x) { /* ... */ }
  public void MoveY(int y) { /* ... */ }
}
```

3.1.1. Vererbung

Ableiten von der Basisklasse ist ***nicht*** möglich, Verwendung als Basisklasse ebenfalls ***nicht***. Structs können aber Interfaces implementieren.

3.1.2. Verwendung von Structs

Ein Struct sollte nur verwendet werden, wenn:

- Repräsentiert ***einzelnen Wert*** *(Keine Instanzen)*
- Instanzgrösse ist ***kleiner*** als 16 Byte
- Ist ***immutable*** *(Kann nicht verändert werden)*
- Wird ***nicht*** häufig ***geboxt***
- Ist entweder ***kurzlebig*** oder wird in andere Objekte ***eingebettet***.

In allen anderen Fällen sollte eine Klasse verwendet werden.

3.1.3. Instanzierung von Klassen/Structs

Erzeugt aus Klasse/Struct ein Object. Dabei wird:

1. Speicherplatz im RAM alloziert
2. Speicher initialisiert *(`default` oder mitgegebene Werte)*

```
Point p1 = new Point(0, 1); // Klassisch
Point p2 = new(1, 1);       // Target typed new
```

3.2. KLASSEN

Klassen sind ***reference Types***, d.h. sie werden auf dem ***Heap*** angelegt.

```
class Stack {
  int[] _values; // reference type
  int _top = 0;
  public Stack(int size) { /* ... */ }
  public void Push(int x) { /* ... */ }
  public int Pop() { /* ... */ }
}
Stack s = new Stack(10); // Instanzierung
```

3.2.1. Vererbung

Ableiten von der Basisklasse ist möglich, Verwendung als Basisklasse ebenfalls. Klassen können auch Interfaces implementieren.

3.2.2. Felder

Ein Feld ist eine Variable eines beliebigen Typs, die in einer Klasse oder einem Struct deklariert wird.

- ***Feld:*** Initialisierung in Deklaration optional, Initialisierung darf nicht auf (andere) Felder und Methoden zugreifen.
- ***`readonly` Feld:*** In Deklaration oder Konstruktor initialisiert, muss ***nicht*** zur Compilezeit berechenbar sein. Wert darf danach nicht mehr geändert werden.
- ***Konstante `const`:*** Muss einen Initialisierungswert haben, dieser muss zur Compilezeit berechenbar sein *(keine Objektinitialisierung mit new!)*.

```
class MyClass {
  int _value = 9; // standardmässig private
  const long Size = int.MaxValue / 3 + 1234;
  readonly DateTime _date1 = DateTime.Now;
  readonly DateTime _date2;
  public MyClass() {
    _date2 = DateTime.Now();
  }
  public void DoSomething() {
    _value = 10;
    _date2 = DateTime.Now; // Compilerfehler
  }
}
```

3.2.3. Nested Types

Werden für ***spezifische Hilfsklassen*** gebraucht.

Regeln:

- Die ***äussere Klasse*** hat Zugriff auf die innere Klasse *(Nur auf «Public Members»)*
- Die ***innere Klasse*** hat Zugriff auf die äussere Klasse *(Auch auf «Private Members»)*
- ***Fremde Klassen*** erhalten Zugriff auf innere Klasse, wenn diese ***«public»*** ist
- ***Erlaubte Nested Types sind:*** Klassen, Interfaces, Structs, Enums, Delegates

```
public class OuterClass {
  private int _outerValue;
  InnerC _innerInstance = new();
  public void OuterMethod() {
    _innerInstance.InnerMethod(this);
  }
  public class InnerClass {
    public void InnerMethod(OuterClass outerClass) {
      outerClass._outerValue = 123;
} } }
```

Anwendung

```
OuterClass outer = new(); outer.OuterMethod();
OuterClass.InnerClass i = new(); i.InnerMethod();
```

3.2.4. Statische Klassen

Regeln: Nur ***statische Member*** erlaubt, kann ***nicht instanziert*** werden, sind «sealed».
Zweck: ***Sammlung*** von Standard-Werten oder Funktionalitäten, Definition von Erweiterungsmethoden.

```
static class MyMath {
  public const double Pi = 3.14159;
  public static double Sin(double x) { ... }
  public static double Cos(double x) { ... }
}
```

3.2.5. Statische Usings

Verkürzt Quellcodes bei Verwendung von statischen Klassen.

Regeln:

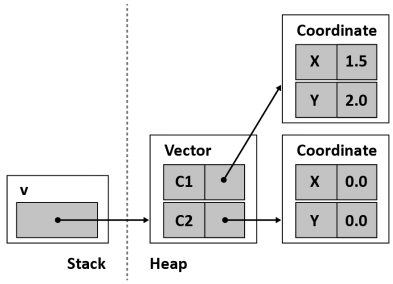
- Nur ***statische Klassen*** sowie Enums erlaubt
- Importiert ***alle statischen Members*** und ***statischen Nested Types***
- Bei ***Namenskonflikten*** gelten die Auflösungsregeln wie bei überladenen Methoden.
- Bei ***identischen Signaturen*** muss der Klassenname vorangestellt werden.

```
using static System.Console;
using static System.Math;
using static System.DayOfWeek;

class ExamplesStaticUsing {
  static void Test() {
    WriteLine(Sqrt(3 * 3 + 4 * 4));
    WriteLine(Friday - Monday);
  }
}
```

3.3. MEMORY MODELL

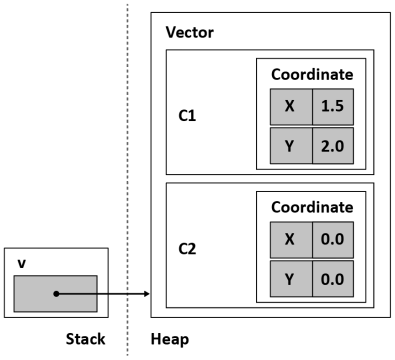
3.3.1. Beispiel Klasse (Call by Reference)




```
class Coordinate {
  public double X { get; set; }
  public double Y { get; set; }
}
class Vector {
  public Coordinate C1 { get; set; }
  public Coordinate C2 { get; set; }
}

Vector v = new();
v.C1.X = 1.5;
v.C1.Y = 2.0;
```

3.3.2. Beispiel Struct (Call by Value)




```
struct Coordinate {
  public double X { get; set; }
  public double Y { get; set; }
}
class Vector {
  public Coordinate C1 { get; set; }
  public Coordinate C2 { get; set; }
}

Vector v = new();
v.C1.X = 1.5;
v.C1.Y = 2.0;
```

3.4. METHODEN

In anderen Sprachen ***«Funktionen»*** genannt.

3.4.1. Statische Methoden

Es gibt Methoden ***mit Rückgabewert*** *(Funktionen)* und ***ohne Rückgabewert*** *(Prozedur / Aktion)*.

Verwendung innerhalb `MyClass`:

```
Print();
int value1 = GetValue();
PrintStatic();
int value2 = GetValueStatic();
```

Verwendung ausserhalb `MyClass`:

```
MyClass mc = new();
mc.Print();
int value1 = mc.GetValue();
MyClass.PrintStatic();
int value2 = MyClass.GetValueStatic();
```

```
class MyClass {
  // Prozedur / Aktion
  public void Print() { }

  // Funktion
  public int GetValue() { return 0; }

  // Statische Prozedur / Aktion
  public static void PrintStatic() { }

  // Statische Funktion
  public static int GetValueStatic()
    { return 0; }
}
```

3.4.2. Parameter

- ***Value-Parameter:*** Kopie des Stack-Inhaltes wird übergeben *(Call by Value)*. Modifiziert nur diese Kopie, Original-Variabel im Aufrufer bleibt unberührt.
- ***Ref-Parameter:*** Adresse der Variable wird übergeben, Variable muss initialisiert sein *(Call by reference)*. Modifiziert Original-Variable im Aufrufer. In der Signatur und beim Methodenaufruf mit `ref` gekennzeichnet.
- ***out-Parameter:*** Wie Ref-Parameter, aber zur Initialisierung von Werten gedacht. Ist zuerst leer, muss in Methode ***zwingend initialisiert*** werden.
  Anwendung: Variablen können direkt im Methodenaufruf deklariert werden, nicht benötigte out-Parameter können mit Underscore `out _` ignoriert werden.

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

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

void Init(out int a, out int b) { a = 1; b = 2; }
void TestInit() {
  Init(out int a1, out int b1); // a1 == 1, b1 == 2
  Init(out int a3, out _); // a3 == 1, b weggeworfen
}
```

Parameter Array

Erlaubt ***beliebig viele Parameter***, muss am Schluss der Deklaration stehen. Nur eines pro Methode erlaubt, keine Kombination mit `out` oder `ref`. Ab C# 13 auch mit `IEnumerable` und Subklassen möglich *(z.B. `List<>`)*.

```
void Sum(out int sum, params int[] values) { ... }
Sum(out int sum1, 1, 2, 3);
void Sum(params List<int> values) { ... }
Sum(new List<int> { 1, 2, 3 });
```

Optionale Parameter

Erlaubt ***Zuweisung eines Default-Values***, Deklaration muss ***hinter*** erforderlichen Parametern erfolgen. Weglassen ist nur am Ende erlaubt. Muss zur Compiletime berechenbar sein *(Konstanter Ausdruck oder Struct/Enum-Initialisierung)*. `default` erlaubt.

```
void Sort(int[] arr, int from = 0, int to = -1){...}
Sort(a); Sort(a, 0); // beides erlaubt
Sort(a, , 3);        // Weglassen nicht erlaubt
void Bad(object x = new()) { ... } // nicht erlaubt
void Ok(int x = default) { ... }   // erlaubt
```

Named Parameter

Erlaubt Identifikation ***anhand Namen*** anstelle anhand der Position. Weglassen beliebiger optionaler Parameter erlaubt. Positionsparameter *(im Beispiel `a`)* immer noch davor. Können aber auch mit Namen angegeben werden, dann ist die Reihenfolge egal.

```
void Sort(int[] arr, int from = 0, int to = -1){...}
Sort(a);
Sort(a, from: 0); // a muss zuerst stehen
Sort(a, to: 10);  // a muss zuerst stehen
Sort(from: 2, to: 5, arr: a); // a darf hinten sein
```

3.4.3. Überladung / Overloading

Mehrere Methoden mit ***gleichem Namen*** möglich.

Voraussetzung:

- Unterschiedliche Anzahl Parameter **oder**
- Unterschiedliche Parametertypen **oder**
- Unterschiedliche Parameterarten *(`ref`/`out`)*.

Der ***Rückgabetyp*** spielt bei der Unterscheidung keine Rolle.

```
void Test(int x, long y) { ... }
void Test(long x, int y) { ... }
int i; long l; short s;
Test(i, l); Test(l, i); // klappen wie erwartet
Test(i, s); Test(i, i); // Fehler: Mehrdeutig
```

Konflikte

- ***Rückgabetyp*** ist ***kein*** Unterscheidungsmerkmal
- ***params-Array*** wird als ***normales Array*** interpretiert
- ***Default-Parameter: kein*** Unterscheidungsmerkmal
- Methoden ***ohne optionale Parameter*** haben Vorrang

```
// Compilerfehler: Identisch trotz return type
void Test() { ... } bool Test() { ... }
// Compilerfehler: Identisch trotz params array
int A(int[] x){...} int A(params int[] x){...}
// A(3); nimmt Overload ohne optionale Params
int A(int x){...} int A(int x, int y = 0){...}
```

3.5. PROPERTIES

Ersatz für Java-style ***Getter-/Setter-Methoden***. Reines Compiler-Konstrukt, verhält sich wie «Public Field».

Nutzen:

- Benutzersicht/Implementation können unterschiedlich sein
- Validierung beim Zugriff
- Ersatz für «Public Fields» auf Interfaces
- Über Reflection als Property identifizierbar

Regeln:

- Read- und/oder Write-Only möglich *(nur `get` oder `set`)*
- Schlüsselwort `value` für Zugriff auf Wert in Setter
- Sichtbarkeiten können verändert werden
  (z.B. `private set { _length = value }`)

```
class MyClass {
  // Backing-Field, nur bei eigenem get/set nötig
  private int _length;
  // Property
  public int Length {
    // Standard-Implementation bei {get; set;}
    get { return _length; }
    set { _length = value; }
  }
}
// Verwendung
MyClass mc = new();
mc.Length = 12;
int length = mc.Length;
```

3.5.1. Auto-implemented Properties

Syntaktische Vereinfachung, Backing Field & Getter/Setter werden automatisch generiert. Bei Initialisierung direkt hinter Property darf der Setter weggelassen werden.

```
public int LengthAuto { get; set; }
// nur zuweisbar hinter Property und im Konstruktor
public int LengthInit { get; } = 5;
```

3.5.2. Objekt-Initialisierung

Properties können direkt initialisiert werden.

```
MyClass mc = new() { Length = 1; Width = 2; }
```

3.5.3. Init-only setters

Property muss während Initialisierung gesetzt werden, danach readonly. Ebenfalls custom `init`-Logik möglich.

```
public int LengthInitOnly { get; init; }
MyClass mc = new() { var LengthInitOnly = 1 };
mc.LengthInitOnly = 2; // Compilerfehler
```

3.6. EXPRESSION-BODIED MEMBERS

Methoden *(und Operatoren, Konstruktoren, Properties)* mit genau einem Statement. Wird mit `=>` hinter Methode gekennzeichnet *(Klammern & `return` entfallen dadurch)*.
Get/Set und reine Set-Properties brauchen einen Statement-Block *({...})*.

```
public MyClass(int v) => _value = v;
int Sum(int x, int y) => x + y;
void Print() => Console.WriteLine("Hoi!");
int Four => Sum(2, 2);
int x { get => _value; set => _value = value; }
```

3.7. KONSTRUKTOREN

Bei jedem Erzeugen einer Klasse / eines Structs verwendet *(Aufruf von «new()»)*. Default-Konstruktor *(ohne Parameter)* wird generiert, falls keiner definiert. ***Private Konstruktoren*** können nur intern verwendet werden. Es wird ***kein*** Default-Konstruktor erzeugt, wenn ein privater Konstruktor existiert.

```
class MyClass {
  private int _x, _y;
  public MyClass() : this(0, 0) { } // Default Ctor
  public MyClass(int x) : this(x, 0) { }
  public MyClass(int x, int y){_y = y; _x = x;}
}
```

Regeln

Konstruktoren können ***überladen*** werden. Aufruf anderer Konstruktoren mittels `this` *(damit Verkettung möglich, siehe Beispiel oben)*, Aufruf auf Basis-Klassen-Konstruktor mittels `base`.

3.7.1. Primary Constructors

Vereinfachung / Verkürzung der Konstruktorlogik. Die Werte für die Konstruktoren werden der Klasse selbst als Parameter mitgegeben, der reguläre Konstruktor fällt weg.

Regeln:

- Klassenparameter *(im Beispiel `x`, `y`)* sind innerhalb der ganzen Klasse verfügbar
- Können zur Initialisierung verwendet werden
- Können verändert werden, ist aber bad practice

```
class Point(int x, int y) {
  // Property Initialisierung
  public int X { get; } = x;
  public int Y { get; } = y;
  public void Print() {
    // Auch in Funktionen verwendbar
    Console.WriteLine($"Point: {x}/{y}");
  }
}
Point p = new(1, 2); // Anwendung
```

3.7.2. Default Constructor

Ein Default Constructor hat keine Parameter. Er hat in Klassen und Structs andere Eigenschaften.

Klasse	***Struct***
– Parameterloser Konstruktor	– Parameterloser Konstruktor
– ***Nicht*** zwingend vorhanden	– ***Immer*** vorhanden
– Automatisch generiert, wenn nicht vorhanden	– Automatisch generiert, wenn nicht vorhanden
– ***Nicht*** automatisch generiert, wenn anderer Konstruktor vorhanden	– Automatisch generiert, wenn anderer Konstruktor vorhanden
– Initialisiert Felder mit «`default`» Wert	– Initialisiert Felder mit «default» Wert
– Konstruktor kann beliebig viele Felder initialisieren	– Konstruktor muss ***alle*** Felder initialisieren
	– «`default`» Literal verwendet diesen Default-Konstruktor

3.7.3. Statische Konstruktoren

Werden für statische ***Initialisierungsarbeiten*** verwendet. Identisch bei Klasse & Struct.

Regeln

Zwingend Parameterlos, Sichtbarkeit darf nicht angegeben werden. Es ist nur ***ein*** statischer Konstruktor erlaubt. Wird genau ***einmal*** ausgeführt: Entweder bei erster Instanzierung des Typen oder bei erstem Zugriff auf statisches Member des Typen. Kann ***nicht*** explizit aufgerufen werden.

```
class MyClass {
  static MyClass() {
    /* ... */
  }
}
struct MyStruct {
  static MyStruct() {
    /* ... */
  }
}
```

3.7.4. Initialisierungsreihenfolge ohne Vererbung

Statische Klassenvariablen → Statische Konstruktoren → normale Klassenvariablen → normaler Konstruktor.

Beim ***zweiten Aufruf*** der Klasse wird der ***statische Teil weggelassen***.

```
class Base { // Zahl = Aufruf-Reihenfolge
  private static int _baseStaticValue = 0; // 1.
  private int _baseValue = 0;              // 3.
  static Base() {}                         // 2.
  public Base() {}                         // 4.
}
```

3.7.5. Initalisierungsreihenfolge mit Vererbung

Zuerst in Subklasse: Statische Klassenvariablen → Statische Konstruktoren → normale Klassenvariablen, dann Reihenfolge in ***Basisklasse*** wie ohne Vererbung, zuletzt ***normaler Konstruktor*** von ***Subklasse***.

Beim ***zweiten Aufruf*** der Subklasse wird der ***statische Teil*** in Sub und Base ***weggelassen***.

```
class Sub : Base { // Sub-Aufruf / Base-Aufruf
  private static int _subStaticValue = 0; // 1. / 4.
  private int _subValue = 0;              // 3. / 6.
  static Sub() {}                         // 2. / 5.
  public Sub() {}                         // 8. / 7.
}
```

3.7.6. Konstruktoren in Base- und Subklasse

Impliziter Aufruf des Basisklassenkonstruktors			***Expliziter Aufruf***

`class Base {`
`// Default Constructor`
`}`
`class Sub : Base {`
`  public Sub(int x) {}`
`}`
	`class Base {`
	`  public Base() {}`
	`}`
	`class Sub : Base {`
	`  public Sub(int x) {}`
	`}`
		`class Base {`
		`  public Base(int x) {}`
		`}`
		`class Sub : Base {`
		`  public Sub(int x) {}`
		`}`
			`class Base {`
			`  public Base(int x) {}`
			`}`
			`class Sub : Base {`
			`  public Sub(int x)`
			`  : base(x) {}`
			`}`

`Sub s = new Sub(1);`			
Konstruktoraufrufe OK – `Base()` – `Sub(int x)`	Konstruktoraufrufe OK – `Base()` – `Sub(int x)`	***Compilerfehler!*** Default-Konstruktor für `Base` nicht mehr automatisch erzeugt	Konstruktoraufrufe OK – `Base(int x)` – `Sub(int x)`

3.7.7. Destruktoren / Finalizer

Finalizer ermöglichten ***Abschlussarbeiten*** beim Abbau eines Objekts. Nur bei ***Klassen***. Zwingend ***parameterlos*** und ohne Visibility. Nur ***1 Finalizer*** pro Klasse erlaubt.

Wird nicht-deterministisch vom Garbage Collector aufgerufen, ***kein expliziter Aufruf*** möglich. Danach Aufruf von Finalizer der Basisklasse. Vergleiche Kapitel «Deterministic Finalization: Dispose()» (Seite 20).

```
class MyClass {
  ~MyClass() {
    /* Freigabe von File-Handles,
    Netzwerk-Streams etc. */
  } }
// ~ Operator (Finalizer) wird vom Compiler
// in diesen Code umgewandelt
override void Finalize() { try {/*Code in
Finalizer*/} finally { base.Finalize(); } }
```

3.8. OPERATOREN

Bausteine für ***Verknüpfung von Werten*** *(Addition, Zuweisung, Methodenaufrufe, Type Casts, …)*

Speziell:

- Logische Operatoren *(||, &&, !)* sind ***short-circuit***, d.h. wenn die linke Kondition für das Endresultat genügt, wird die rechte nicht mehr evaluiert.
- Bit-Operatoren *(&, |, ~)* verarbeiten Ganzzahlen, Chars, Enums & Bools auf Bitebene.
- Expliziter ***Overflow-Check*** mit `checked(x * x)`.
- `typeof(int)` liefert Typ von Klasse/Element für Vergleiche
- `sizeof(int)` liefert Datengrösse von Value Types in Bytes.

Unäre: `+x, -x, ~x, !x, x++, x--, ++x, --x, true, false`
Binäre: `x + y, x - y, x * y, x / y, x % y, x & y, x | y, x ^ y, x << y, x >> y, x == y, x != y, x < y, x > y, x <= y, x >= y`
Conditional: `x && y, x || y`
Index: `a[i], a?[i]`
Type Cast: `(T)x`
Kombinierte Zuweisung: `+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`
Diverse: `^x, x = y, x.y, x?.y, c ? t : f, x ?? y, x ??= y, x..y, =>, f(x), as, await, checked, unchecked, default, delegate, is, name, nameof, new, sizeof, stackalloc, switch, typeof`

3.8.1. Operator Overloading

Die ***Funktionsweise von einem Operator*** kann für eine Klasse ***definiert/geändert*** werden, indem er overloaded wird. Somit können Funktionen wie das Vergleichen oder das «Addieren» von zwei Objekten einer Klasse realisiert werden.

Ein Operator kann «alles» machen, man sollte sie aber nur dann implementieren, wenn Resultat eines Operators für den Anwender «natürlich» wirkt.

«When in doubt, do as the `int`s do»

```
class Point {
  private int _x, _y;
  public Point(int y, int y)
    { _x = x; _y = y; }
  // returns negated points
  public static Point operator ~(Point a)
    => new(a._x * -1, a._y * -1);
  // adds a points with another point
  public static Point operator +(Point a, Point b)
    => new(a._x + b._x, a._y + b._y); }
```

Regeln

Methode muss `static` sein, Schlüsselwort `operator` gefolgt von z.B. `+`, Unäre Operatoren haben 1 Parameter, Binäre 2 Parameter, Rückgabetyp ist frei wählbar, mindestens 1 Parameter muss vom Typ der enthaltenden Klasse sein, müssen teilweise als Paar überladen werden *(`==` `!=`, `<` `>`, `<=` `>=`, `true` `false`)*. Es können nur Unäre und Binäre Operatoren überladen werden, Index- und Type Cast-Operator verfügen aber über Alternativen *(Indexer, user-defined conversion operators)*.

3.9. INDEXER

Für eine Klasse können ***Aufrufe über einen Index*** definiert werden. Da der Index-Operator «`[]`» nicht überladen werden kann, gibt es Indexer. Die Klasse verhält sich dann wie ein Array.

Der Typ des Indexers kann frei bestimmt werden. Häufige Beispiele sind für `int` *(Array-like)*, `string` *(Dictionary-like)* oder 2-Dimensionale Indexer *(Mehrdimensionale Arrays)*

Regeln: `get` und `set` möglich, `this` definiert Indexer, `value` für Wert im Setter, Overloading möglich.

```
class SimpleDictionary {
  private string[] _arr = new string[10];
  public int this[string search] {
    get {
      for (int i = 0; i < _arr.Length; i++) {
        if (_arr[i] == search) return i;
      }
      return -1;
    } }
  // 2-Dimensionaler Indexer, hier so wenig sinnvoll
  public string this[int i1, int i2] { ... } }
string location = words["Hello There"];
```

3.10. PARTIELLE KLASSEN & METHODEN

3.10.1. Partielle Klassen

Definition eines Typen über mehrere Dateien. Member können ganz normal aufgerufen werden. Funktioniert mit Klassen, Structs und Interfaces.

Verwendung: Aufsplitten grosser Dateien, Trennen von (teilweise) Generator-erzeugtem Code

```
// File1.cs
partial class MyClass
  { public void Test1() {} }
// File2.cs
partial class MyClass
  { public void Test2() {} }
```

Regeln

`partial` muss bei allen Definitionen der Klasse angemerkt sein. Alle Definitionen müssen dieselbe Visibility haben, Class Modifier müssen überall angegeben werden *(Basisklasse, implementierte Interfaces, `sealed` etc.)*, alle Teile müssen zusammen die Voraussetzungen der implementierten Interfaces erfüllen.

3.10.2. Partielle Methoden

Trennung von Deklaration und Implementation einer Methode. Ermöglicht ***«Hooks»*** in Code *(vgl. Event-Registrierung in JavaScript)*. Funktioniert in `partial` Klassen & Structs. ***Kann, muss aber nicht implementiert sein.*** Wenn implementiert, wird Methode normal aufgerufen. Wenn nicht, wird der Methodenaufruf einfach ignoriert.

Regeln: Immer `void` und `private`, kann `static` sein.

```
partial class NiceGUI { // File1.cs
  public void MainWindow() {
    OnClick(); // ausgeführt, wenn implement.
  }
  partial void OnClick();
  partial void OnHover();
}
partial class NiceGUI { // File2.cs
  partial void OnClick() { ... }
}
```

4. VERERBUNG & DETERMINISTIC FINALIZATION

4.1. VERERBUNG UND TYPE CHECKS

Konstruktoren werden nicht vererbt. Nur eine Basisklasse, aber beliebig viele Interfaces erlaubt. Structs haben nur Vererbung via Interfaces. Klassen sind (in)direkt von `System.Object` abgeleitet und können mit dieser Klasse Boxing/Unboxing betreiben.

4.1.1. Zuweisungen

- ***Statischer Typ:*** Typ der Variable
- ***Dynamischer Typ:*** Referenziertes Heap-Objekt

Regeln:

- Eine Zuweisung ist ***immer möglich, wenn*** Statischer Typ = Dynamischer Typ oder Statischer Typ eine Basisklasse von Dynamischer Typ ist.
- Zuweisung ***verboten, wenn*** Statischer Typ Subklasse von Dynamischer Typ oder nicht in derselben Vererbungshierarchie wie Dynamischer Typ ist.

```
class Base {}
class Sub : Base {}
class SubSub : Sub {}

public static void Test() {
  Base b = new Base(); // DynType == StaticType
  b = new Sub();       // Dyn Type: Sub
  b = new SubSub();    // Dyn Type: SubSub
  Sub s = new Base();  // Compilerfehler
}
```

4.1.2. Type Check: `obj is T`

Prüft, ob Objekt mit einem Typen ***kompatibel*** ist.
`true`, wenn Objekt identisch wie `T` oder Subklasse davon.
`false`, wenn Objekt Basisklasse von `T`, nicht in derselben Vererbungshierarchie wie `T` oder `null` ist.

```
SubSub a = new SubSub();
if (a is SubSub) {...} // true
if (a is Sub) {...}    // true
if (a is Base) {...}   // true
a = null; if (a is SubSub) {...} // false
```

4.1.3. Type Casts: `(T)obj`

Explizite Typumwandlung. `null` kann auch gecastet werden. Compilerfehler, wenn Type Cast nicht zulässig oder `null` in einen Value Type gecastet wird. `InvalidCastException` bei Laufzeit-Fehler.

```
Base b = new SubSub();  Sub s = (Sub)b;
SubSub ss1 = (SubSub)b; SubSub ss2 = (SubSub)null;
int i = (int)null; // Compilerfehler
string str = (string)s // Compilerfehler
object o = "Hi"; b = (Base)o; //InvalidCastException
```

4.1.4. Type Casts mit `as` Operator

Syntactic Sugar für Casting, macht `null`-Check vor Cast:
`obj is T ? (T)obj : (T)null`

Anstatt Laufzeitfehler hat man bei ungültigen Casts `null`.

```
Base b = new SubSub(); Sub s = b as Sub;
int i = null as int; // Compilerfehler
string str = s as string // Compilerfehler
object o = "Hi"; b = o as Base; // o = null
```

4.1.5. Type Check mit Type Cast: `obj is T result`

Syntactic Sugar für Type Check, returnt der `is`-Type Check `true`, wird das Objekt in die `result`-Variable gecasted. Arbeitet nicht mit nullable Types, im Gegensatz zu `as`.
Da das ganze Konstrukt `bool` returnt, kann es z.B. in `if`-Statements verwendet werden.

```
Base a = new SubSub()
if (a is SubSub casted) {
  Console.WriteLine(casted);
}
```

4.2. VERERBUNG IN METHODEN

4.2.1. Methoden überschreiben

Subklasse kann Member *(Methoden, Properties, Indexer)* der Basisklasse überschreiben.
- ***`virtual`:*** In Basisklasse, um Methode überschreibbar zu machen
- ***`override`:*** In Subklasse um Basismethode zu überschreiben

```
class Base {
  public virtual void G() { ... } }
class Sub : Base {
  public override void G() { ... } }
class SubSub : Sub {
  public override void G() { ... } }
```

Regeln
- Members sind ***per Default weder `virtual` noch `override`***, auch dann nicht, wenn Basis-Member `virtual` ist.
- ***Identische Signatur*** *(Gleiche Anzahl Parameter, gleiche Parametertypen und -arten (`ref`/`out`), gleiche Sichtbarkeit und gleicher Return Type)*
- `virtual` verboten mit: `static`, `abstract` *(impliziert `virtual`)*, `private` *(unsichtbar für Subklasse)*, `override` *(implizit `virtual`)*

4.3. DYNAMIC BINDING

Methode des ***dynamischen Typs*** wird aufgerufen.
Falls dynamischer Typ *(`SubSub`)* konkreter als statischer Typ *(`Base`)* und Funktion *(`G()`)* als `virtual` markiert, wird von oben nach unten in der Hierarchie nach konkretester Methode `G()` mit `override` gesucht.

```
Base b = new SubSub();
b.G(); // SubSub.G()
Sub s = new SubSub();
s.G(); // SubSub.G();
```

4.3.1. Methoden überdecken

Unterbricht Dynamic Binding! Durch Weglassen von `override` kann eine Subklasse Member der Basisklasse überdecken. Kann durch bewusstes Übersteuern des Basismembers oder durch versehentliche Wahl derselben Signatur passieren.

Compilerwarnung bei Aufruf von `SubSub.G()`:

```
'SubSub.H()' hides inherited member 'Sub.H()'.
To make the current member override that
implementation, add the override keyword.
Otherwise add the new keyword.
```

```
class Base {
  public virtual void G() { ... }
}
class Sub : Base {
  public override void G() { ... }
}
class SubSub : Sub {
  public void G() { ... } // kein Override!
}
```

4.3.2. Methoden überdecken mit `new`

Obige Compilerwarnung kann mit `new` vermieden werden. Sagt Compiler, dass Member bewusst überdeckt wurde. Kann nicht zusammen mit `override` verwendet werden, mit `virtual` aber schon.
Compilerwarnung bei `SubSub.G2()`: `new` soll nicht verwendet werden, wenn es keinen Member überdeckt.

```
class Base {
  public virtual void G() { ... } }
class Sub : Base {
  public override void G() { ... } }
class SubSub : Sub {
  public new void G() {...} // bewusst überdeckt
  public new void G2() { ... } } // Compilerwarnung
```

4.3.3. Komplexes Beispiel

`J()`

```
Base b1 = new SubSub();     b1.J();  // Base.J()
Sub s1 = new SubSub();      s1.J();  // Sub.J()
SubSub ss1 = new SubSub(); ss1.J(); // SubSub.J()
Base b2 = new Sub();        b2.J();  // Base.J()
Sub s = new SubSub();       s.J();   // Sub.J()
```

`K()`

```
Base b1 = new SubSub();     b1.K();  // Sub.K()
Sub s1 = new SubSub();      s1.K();  // Sub.K()
SubSub ss1 = new SubSub(); ss1.K(); // SubSub.K()
Base b2 = new Sub();        b2.K();  // Sub.K()
Sub s = new SubSub();       s.K();   // Sub.K()
```

`L()`

```
Base b1 = new SubSub();     b1.L();  // Base.L()
Sub s1 = new SubSub();      s1.L();  // SubSub.L()
SubSub ss1 = new SubSub(); ss1.L(); // SubSub.L()
Base b2 = new Sub();        b2.L();  // Base.L()
Sub s = new SubSub();       s.L();   // SubSub.L()
```

```
class Base {
  public virtual void J() { ... }
  public virtual void K() { ... }
  public virtual void L() { ... }
}
class Sub : Base {
  public new void J() { ... }
  public override void K() { ... }
  public new virtual void L() { ... }
}
class SubSub : Sub {
  public new void J() { ... }
  public new void K() { ... }
  public override void L() { ... }
}
```

Eselsbrücke:
`virtual`: Kette starten, ***`override`:*** Kette fortführen,
`new`: Kette beenden, ***`new virtual`:*** Neue Kette starten

4.4. ABSTRAKTE KLASSEN

Mischung aus Klasse und Interface, muss überschrieben werden. Im Gegensatz zu `virtual` ist aber keine Implementation in der Basisklasse möglich. Mit `abstract` deklariert. Für alle Klassenmember-Arten möglich

Regeln: Keine direkte Instanzierung, beliebig viele Interfaces implementierbar, von abstrakten Klassen abgeleitete Klassen ***müssen*** alle abstrakten Member implementieren, abstrakte Member nur innerhalb abstrakter Klassen, Klasse darf nicht `sealed` sein, Methoden nicht `static` oder `virtual` *(Methoden sind selbst implizit `virtual`)*.

```
abstract class Sequence {
  public abstract void Add(int i);
  public abstract string Name { get; }
  public abstract object this[int i] { get; set; }
  public override string ToString() { return name; }
}

class List : Sequence {
  public override void Add(int x) { ... }
  public override string Name { get { ... } }
  public override object this[int i] { ... }
/* ToString muss nicht implementiert werden */ }
```

4.5. INTERFACES

Ähnlich wie abstrakte Klasse. Darf keine Sichtbarkeit vorgeben, kann andere Interfaces erweitern. Keine Instanzierung. Member sind implizit `abstract virtual` und dürfen nicht `static` sein. Name beginnt mit `I`.

Implementation: Klassen dürfen beliebig viele Interfaces implementieren *(`class A : I1, I2 { ... }`)*. Alle Interface-Member *(direkt oder von Basisklasse geerbt)* müssen in Klasse vorhanden und `public` sein, `static` Member verboten. `override` nicht nötig, ausser Basisklasse definiert gleichen Member. Kombination mit `abstract` & `virtual` erlaubt.

```
interface ISequence {
  void Add(object x);
  string Name { get; }
  object this[int i] { get; set; }
  event EventHandler OnAdd;
}
class List : ISequence {
  public void Add(object x) { ... }
  public string Name { get { ... } }
  public object this[int i] { get {...} set {...} }
  public event EventHandler OnAdd;
}
```

4.5.1. Verwendung

Klassen mit implementiertem Interface können wie bei normaler Basisklasse zugewiesen, umgewandelt und auf Vererbung geprüft werden.

```
List list1 = new List(); list1.Add("Nina");
ISequence list2 = new List(); list2.Add("Jannis");
list1 = (List)list2;
list1 = list2 as List;
list2 = list1;
if (list1 is ISequence) { ... }
```

4.5.2. Naming Clashes in Interfaces

Mehrere Interfaces können gleiche Member definieren, was zu Kollisionen führt.

Szenarien:

1. Beide `Add()` haben ***dieselbe Funktionalität***. Logik für beide lässt sich in einer Methode abbilden
   → ***Methode regulär implementieren***
2. Beide `Add()` haben eine ***andere Funktionalität***. Logik ist für beide unterschiedlich
   → ***Methoden explizit implementieren***
3. Wie 2., es gibt aber ein ***«Default-Verhalten»***. Identische Signatur, aber unterschiedliche Logik. Eine Methode soll als Standard aufgerufen werden.
   → ***Methoden regulär & explizit implementieren***
4. Beide `Add()` haben einen ***anderen Rückgabewert*** und es gibt ein «Default-Verhalten».
   → ***Methoden regulär & explizit implementieren***

```
interface ISequence { void Add(object x); }
interface IShoppingCart {
  void Add(object x);
  int Add(object x);
}

class ShoppingCart : ISequence, IShoppingCart {
  // Welches Add() wird aufgerufen?

  // Szenario 1: Gleiche Funktionalität
  public void Add(object x) { ... }

  // Szenario 2: Andere Funktionalität
  void ISequence.Add(object x) { ... }
  void IShoppingCart.Add(object x) { ... }

  // Szenario 3: wie 2. mit Default-Verhalten
  public void Add(object x) { ... } // Eigene Methode
  // void ISequence.Add() existiert nicht
  void IShoppingCart.Add() { ... } // Interface-Methode

  // Szenario 4: Andere Rückgabetypen
  public void Add(object x) {...}
  void ISequence.Add(object x) { ... }
  int IShoppingCart.Add(object x) { ... }
}
```

Anwendung

```
ShoppingCart sc = new ShoppingCart();
ISequence isq = new ShoppingCart();
IShoppingCart isc = new ShoppingCart();

// Szenario 1
sc.Add("Hello");  // ShoppingCart.Add
isq.Add("Hello"); // ShoppingCart.Add
isc.Add("Hello"); // ShoppingCart.Add

// Szenario 2
sc.Add("Hello");  // Compilerfehler
isq.Add("Hello"); // ISequence.Add
isc.Add("Hello"); // IShoppingCart.Add

// Szenario 3
sc.Add("Hello");  // ShoppingCart.Add
isq.Add("Hello"); // ShoppingCart.Add
isc.Add("Hello"); // IShoppingCart.Add
((IShoppingCart)sc).Add("Hello");
//IShoppingCart.Add

// Szenario 4
sc.Add("Hello");  // ShoppingCart.Add
isq.Add("Hello"); // ISequence.Add
isc.Add("Hello"); // IShoppingCart.Add
```

4.5.3. Default Interface Methods

Formulieren von Logik auf Interfaces. Führen «quasi-Mehrfachvererbung» ein. Zusätzlich zu normalen Interface-Membern auch für Operatoren möglich.

Motivation

Interface wird mehrfach implementiert, damit muss Logik basierend auf Interface ausgelagert/kopiert werden.
Im Beispiel müsste ohne Default Interface Methods `FullName` und `Print()` in `Person1` und `Person2` implementiert werden
→ ***Duplizierter Code***

Regeln

Hat Zugriff auf alle (vererbten) Member, `virtual` / `static` möglich, Anwender muss Variable in Interface casten.

```
interface IPerson {
  string FirstName { get; }
  string LastName { get; }
  string FullName => $"{FirstName} {LastName}";
  void Print() => Console.WriteLine(FullName);
}
public class Person1 : IPerson {
  public string FirstName { get; }
  public string LastName { get; }
  // Implementation von FullName und Print entfällt
}
public class Person2 : IPerson { ... }
// Anwendung
Person p1 = new(); p1.Print(); // Compilerfehler
IPerson p2 = p1;   p2.Print(); // Funktioniert
```

4.6. SEALED KLASSEN

Verhindert das Ableiten von Klassen, wie `final` in Java.
Einsatzzweck: Verhindert versehentliches Erweitern, Methoden können statisch gebunden werden → Performancegewinn.

```
sealed class Sequence {
  // ...
}
class List : Sequence { ... } // Compilerfehler
```

4.6.1. Sealed Member

Verhindert das ***weitere Überschreiben*** eines bestimmten überschreibbaren Members. Muss in Verbindung mit `override` angewendet werden. Kombination auf einem `sealed` Member mit `new` / `virtual` ist nicht erlaubt.

Erbt eine Klasse einen `sealed` Member, ist jedoch Überdecken mit `new` als auch `virtual` wieder erlaubt.

```
abstract class Sequence {
  public virtual void Add(object x) {}
  public sealed void X() } // Compilerfehler
class List : Sequence {
  public override sealed void Add(object x){} }
class MyList : Sequence {
  public override void Add(object x) {} // Error
  public new virtual void Add(object x) {} // OK
}
```

4.7. DETERMINISTIC FINALIZATION: DISPOSE()

Beim Entfernen eines Objekt vom Heap sollen deren ***unmananged Ressourcen*** *(nicht von .NET Runtime verwaltet, z.B. File Handles, DB-Verbindungen etc.)* ***manuell freigegeben*** werden, es muss also nicht wie beim Finalizer das ganze Objekt gelöscht werden. Dafür kann das ***Interface `IDisposable` implementiert*** werden. In der `Dispose()`-Methode können diese spezifiziert werden *(z.B. offene File Handles/Verbindungen schliessen)*. In der Standard-Variante muss aber `Dispose()` jedes Mal manuell aufgerufen werden und es gibt kein Exception Handling.
→ Fehleranfällig!

```
public class DataAccess : IDisposable {
  private DBConnection _con;
  public DBAccess()
    { _conn = new SqlConnection(); }
  // Finalizer, aufräumen beim Löschen durch GC
  ~DataAccess() ⇒ _con?.Dispose();
  public void Dispose() { // Manuelles Löschen
    _con?.Dispose();
    // GC muss Finalizer nicht erneut aufrufen
    GC.SuppressFinalize(this);
  } }
// Verwendung, bad practice, besser mit `using`
var da = new DataAccess(); ...; da.Dispose();
```

4.7.1. Was sollte ein Dispose erledigen?

- Aufräumen von Unmanaged Ressourcen
- Fehlerfreie Ausführung *(keine Exceptions)*
- Sicherstellung einmalige Ausführung
- Sicherstellung vollständiger Cleanup
- Aufräumen von Managed Ressourcen
- Vererbung, Ressourcen in Basisklassen
- Synchrone/Asynchrone Ausführung
- Unterscheidung Finalizer vs. Deterministic Finalization

4.7.2. `using`-Statement

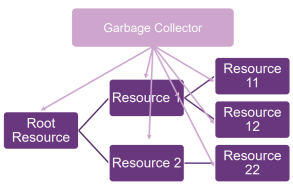
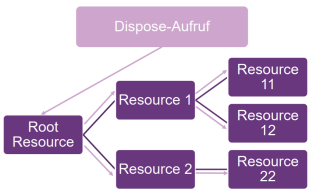
Syntactic Sugar für Finalization, stellt ***Aufruf von `Dispose()` sicher***. Kann mehrere Ressourcen gleichzeitig disposen. Kann als Scope-Block oder als einzelne Ressource definiert werden, welche bis Methodenende gültig ist. Auch als `async`-Variante möglich.

```
using DataAccess da1 = new(); // einzelne Resource
using (DataAccess da2 = new()) { // Scope-Block
  da1.Fetch(); da2.Fetch();
} // `da2` hier disposed
da1.Fetch();// Methodenende, `da1` danach disposed
```

4.7.3. Dispose Pattern

Aus Performancegründen sollte unterschieden werden, ob ein `Dispose()` aus einem Finalizer *(also vom Garbage Collector)* oder von einem manuellen `Dispose()`-Aufruf stammt.
Der Code rechts erweitert die oben definierte Klasse `DataAccess` um ein einfaches Beispiel eines Dispose-Patterns: Wird manuell disposed, macht der GC dies nicht nocheinmal. Mehrmaliges manuelles Disposen wird ebenfalls verhindert. Die neue `Dispose(bool)`-Methode kann von Subklassen überschrieben werden.

```
private bool _disposed; // Dispose() bereits gecallt?
~DataAccess() ⇒ Dispose(false); // Finalizer-Aufruf
public void Dispose() { // Manueller Dispose-Aufruf
  Dispose(true);
  GC.SuppressFinalize(this); //Finalizer nicht mehr nötig
}
protected virtual void Dispose(bool disposing) {
  if (_disposed) { return; } // bereits Disposed
  // GC disposed `_con` selber, nur bei manuellem nötig
  if (disposing) { _con?.Dispose(); }
  _disposed = true;
}
```

Destructor & Finalizer	***Deterministic Finalization***
Kann nur bei Entfernen eines Objekts vom Garbage Collector aufgerufen werden. Nur für unmanaged Ressourcen. GC entfernt Ressourcen in zufälliger Reihenfolge, Abhängigkeiten werden nicht berücksichtigt. 	Kann manuell auch vor Entfernen eines Objekts aufgerufen werden. Auch geeignet für Managed Ressourcen. Jede Methode räumt ihre Ressourcen in festgelegter Reihenfolge auf. 

5. DELEGATES & EVENTS

5.1. DELEGATES

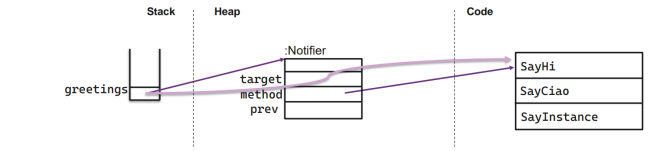
Ein Delegate ist ein Typ, welcher ***Referenzen auf 0 bis n Methoden*** enthält. Die Typsicherheit wird vom Compiler garantiert. ***Vereinfacht gesagt ist ein Delegate ein Interface mit einer Methode, ohne das Interface aussen herum.***
Delagates werden auf Namespace-Ebene definiert *(ausserhalb von Klassen/Interfaces)*. Sie verwenden intern eine Linked List, um die Referenzen zu speichern.
Verwendung: ***Methoden als Parameter übergeben***, Definition von Callback-Methoden.

Einer ***Delegate-Variable*** kann ***jede Methode mit passender Signatur*** zugewiesen werden. Zuweisung von `null` ist erlaubt. Der Aufruf der Funktion passiert direkt über die Delegate-Variable. Exception, wenn diese `null` ist.

```cs
// Definition des Delegates
public delegate void Notifier(string sender);

class DelegateExample {
  public static void Main() {
    Notifier greeter; //Deklaration Delegate-Variab.
    // Zuweisen einer Methode
    greeter = new Notifier(SayHi);
    greeter = SayHi; // Kurzform
    greeter("Nina"); // Aufruf von SayHi()
  }
  private static void SayHi(string sender){
    Console.WriteLine($"Hi {sender}");
  } }
// Der Notifier als Interface sähe so aus:
public interface INotifier
  { void Notifier(string sender); }
List<INotifier> greeter; // Liste der Funktionen
```

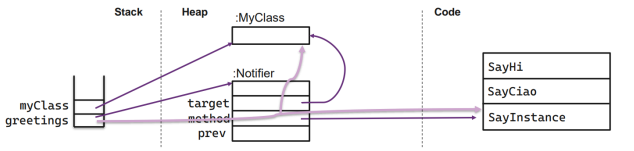
Delegate mit statischer Funktion




```cs
Notifier greetings;
greetings = new Notifier(SayHi);
greetings("John");
```

```cs
// Deklaration eines Delegate-Typs
public delegate void Notifier(string sender);
class Examples
{
    public static void SayHi(string sender)
    {
        Console.WriteLine("Hello {0}", sender);
    }
}
```

Delegate mit Instanzfunktion




```cs
Notifier greetings;
MyClass myClass = new();
greetings = new Notifier(myClass.SayInstance);
greetings("John");
```

```cs
// Deklaration eines Delegate-Typs
public delegate void Notifier(string sender);
class MyClass
{
    public void SayInstance(string sender)
    {
        Console.WriteLine("Inst. {0}", sender);
    }
}
```

Delegate-Zuweisung: `DelegateType delegateVar = obj.Method`
Speichert Methode *(`Method`)* und Empfängerobjekt *(`obj`, oft implizit durch `this`)*.

Eine `Method`, welche einem Delegate zugwiesen wird…
– muss von der Signatur mit `DelegateType` übereinstimmen
– kann `static` sein *(dann mit Klassenname anstatt `obj`)*,
– darf weder `abstract`, `virtual`, `override` noch `new` sein

Delegate-Aufruf: `object result = delegateVar[.Invoke](params)`
Kann entweder direkt oder über `.Invoke()` ausgeführt werden. Parameter können wie gewohnt übergeben werden, genau wie die Handhabung des Rückgabewerts. `delegateVar` sollte vor dem Aufruf auf `null` geprüft werden: `delegateVar?.Invoke(params);`

5.2. MULTICAST DELEGATES

Multicast Delegates können ***mehrere Methoden*** gleichzeitig referenzieren. Jeder Delegate ist bereits ein Multicast Delegate. Mit `+=` können einer Delegate-Variable Methoden hinzugefügt, mit `-=` wieder entfernt werden.

Wird eine ***Delegate-Variable aufgerufen***, werden die Methoden in ***Hinzufügungsreihenfolge*** ausgeführt. Beim ***Entfernen*** wird die zuletzt hinzugefügte Instanz dieser Methode entfernt. Der Rückgabewert eines Multicast Delegates entspricht dem letzten Funktionsaufruf *(Gilt auch für `out`/`ref` Parameter)*.

Delegates können mit dem + Operator auch ***kombiniert*** werden.

```cs
public static void Main() {
  Notifier greeter;
  greeter = SayHi;
  greeter += SayCiao; greeter += SayHi;
  greeter -= SayHi; // entfernt hinterstes SayHi()
  greeter("Nina"); // SayHi(), dann SayCiao()
  Notifier a = SayHi; Notifier b = SayCiao;
  Notifier ab = a + b;
}
private static void SayHi(string sender) {
  Console.WriteLine($"Hi {sender}");
}
private static void SayCiao(string sender) {
  Console.WriteLine($"Bye {sender}");
}
```

5.2.1. Beispiel: Funktionsparameter

Methode für das Ausführen einer beliebigen Funktion auf jedem Element eines Arrays. Es wird eine Funktion `ForAll()` mit einem `int`-Array und einem Delegate `Action` als Parameter erstellt. In `MyClass` befinden sich Funktionen, welche als Delegate `ForAll()` übergeben werden können.

Test-Code

```
int[] array = { 1, 5, 8, 14, 22 };
Action v1 = MyClass.PrintValues;
ForAll(array, v1);
MyClass c = new();
Action v2 = c.SumValues;
ForAll(array, v2); Console.WriteLine(c.Sum);
```

```
public delegate void Action(int i);
public class MyClass {
  public static void PrintValues(int i)
   { Console.WriteLine($"Value {i}"); }
  public void SumValues(int i) { Sum += i; }
  public int Sum { get; private set; }
}
public class FunctionParameter {
  static void ForAll(int[] arr, Action action) {
    Console.WriteLine("ForAll called");
    if (action == null) { return; }
    foreach (int t in arr) {
      action(t);
} } }
```

5.2.2. Beispiel: Callback mit Delegates

Tickende Uhr mit ***dynamisch wählbarem Intervall***. Bei ***jedem Tick*** wird eine Benachrichtigung ausgegeben und ***Instanzen*** *(hier `ClockObserver`)* können sich ***registrieren***, um eine Funktion jeden Tick laufen zu lassen. Diese ***Subscriber*** werden im `EventTickHandler` registriert und bei jedem Tick werden alle Funktionen in diesem ***Delegate*** aufgeführt *(Code in `Tick()`)*.
Im Beispiel unten registrieren wir die Funktion `OnTickEvent` des `ClockObserver`, welche alle 1'000 bzw. 300ms aufgerufen wird.

Aufruf der Uhr

```
public static void Test() {
  Clock c1 = new Clock(1000);
  Clock c2 = new Clock(300);
  ClockObserver t1 = new("Observer 1");
  ClockObserver t2 = new("Observer 2");
  c1.add_OnTickEvent(t1.OnTickEvent);
  c2.add_OnTickEvent(t2.OnTickEvent);
  // Do Stuff here...
  c1.remove_OnTickEvent(t1.OnTickEvent);
  c2.remove_OnTickEvent(t2.OnTickEvent);
}
```

```
public delegate void TickEventHandler
  (int ticks, int interval);

public class Clock(int interval) {
  private int _interval = interval; // Anzahl Aufrufe
  private int _ticks; // Anzahl vergangene Ticks
  private TickEventHandler OnTickEvent;
  public void add_OnTickEvent(TickEventHandler h)
   { OnTickEvent += h; }
  public void remove_OnTickEvent(TickEventHandler h)
   { OnTickEvent -= h; }
  private void Tick(object sender, EventArgs e) {
    _ticks++;
    onTickEvent?.Invoke(ticks, interval);
  }
}

public class ClockObserver(string name) {
  private string _name = name;
  public void OnTickEvent(int ticks, int i) {
    Console.WriteLine($"Observer {_name}: Clock mit
Interval {ticks} hat zum {i}. Mal getickt.");
  }
}
```

5.3. EVENTS

Events sind ***Syntactic Sugar für Delegates***. Das Event generiert innerhalb der Klasse einen `private` Delegate und `private` Subscribe/Unsubscribe-Logik. Auf diese kann von aussen über +=/-= zugegriffen werden.
Angewendet auf das zweite Delegate-Beispiel sieht der Code so aus:

Angepasster Aufruf der Uhr

```
  c1.OnTickEvent += t1.OnTickEvent;
  c2.OnTickEvent += t2.OnTickEvent;
  // Do Stuff here...
  c1.OnTickEvent -= t1.OnTickEvent;
  c2.OnTickEvent -= t2.OnTickEvent;
```

```
public delegate void TickEventHandler
  (int ticks, int interval);

public class Clock(int interval) {
  private int _interval = interval; // Anz. Aufrufe
  private int _ticks; // Anzahl vergangene Ticks
  private event TickEventHandler OnTickEvent;
  private void Tick(object sender, EventArgs e) {
    _ticks++;
    onTickEvent?.Invoke(ticks, interval);
  }
}
// Code für ClockObserver bleibt unverändert
```

5.3.1. Event Syntax

```
public delegate void AnyHandler(object sender, AnyEventArgs e);
```

Events haben immer den ***Rückgabetyp `void`***. Der ***erste Parameter*** ist der Sender des Events, der Absender übergibt beim Aufruf des Events immer implizit `this`. Der ***zweite Parameter*** ist eine beliebige Subklasse von `EventArgs`. Diese ***enthält Informationen zum Event*** *(z.B. linke/rechte Maustaste bei Klick)* und kann um eigene Parameter für das Event ergänzt werden. Dies hat den ***Vorteil***, dass bei Änderungen an den Parametern der Aufruf des Clients auf das Event nicht angepasst werden muss.

5.3.2. Event Handling

Einem Event können ***mehrere Parameter*** mitgegeben werden. Dazu erstellt man zusätzlich zum Event eine weitere Klasse, welche von `EventArgs` abgeleitet ist und alle Properties für das Event enthält. Möchte man nun ein weiteres Property `MousePosition` hinzufügen, kann dies einfach zu den `ClickEventArgs` hinzugefügt werden. Der Aufruf von `OnClick()` ändert sich nicht.

Im Normalfall hat die `EventArgs`-Klasse einen ***Konstruktor***, welche alle Properties entgegen nimmt. Bei manuellen Aufrufen durch `.Invoke()` können so mit `new ClickEventArgs(...)` alle Parameter übergeben werden.

```
public delegate void ClickEventHandler
  (object sender, ClickEventArgs e);

public class ClickEventArgs : EventArgs {
  public string MouseButton { get; set; }
  public string MousePosition { get; set; }
}

public class Button {
  public event ClickEventHandler OnClick;
}
// Anwendung
public class Usage {
  public void Test() {
    Button b = new();
    b.OnClick += OnClick();
  }
  private void OnClick
    (object sender, ClickEventArgs clickEventArgs) {
      /* Das passiert beim Buttonklick */
  }
}
```

5.4. LAMBDA EXPRESSIONS

Methoden, welche als Parameter übergeben wurden, mussten bisher ***einer Variable zugewiesen*** und benannt werden, auch wenn sie nur einmal verwendet werden. Ein weiterer Nachteil ist, dass diese Funktionen nur auf ***globale Variablen*** zugreifen können.
Abhilfe schaffen ***Lambda-Expressions***: Der Code wird in-place angegeben, es ist keine Deklaration einer Methode nötig und Lambda Expressions können ebenfalls auf lokale Variablen zugreifen.

```
class LambdaExample {
  // Auskommentierter Code nicht mehr nötig
  //int _sum = 0;
  //void SumUp(int i) { _sum += i; }
  //void Print(int i) { Console.WriteLine(i); }
  void Test() {
    List<int> list = new();
    list.ForEach(/* Print */
      i => Console.WriteLine(i));
    list.ForEach(/* SumUp */ i => sum += 1); } }
```

5.4.1. Syntax

- ***Expression Lambda:*** *`(parameters) => expression`*
  Ohne Klammern, genau ein Statement erlaubt
  (auch Funktionen & Actions mit Rückgabetypen)
- ***Statement Lambda:*** *`(parameters) => {statements;}`*
  Block mit geschweiften Klammern, beliebig viele Statements erlaubt, mit/ohne Rückgabewert

Der ***Lambda Operator =>*** ist kein «grösser gleich», sondern ein Trennzeichen. Ausgesprochen als «goes to» oder «geht über in».

```
// Expression Lambda
Func<int, bool> fe = i => i % 2 == 0;
Func<int, int, int> e = (a, b) => Multiply(a, b);
// void type Lambdas sind Actions
Action<int> e2 = i => Console.WriteLine(i);

// Statement Lambda
Func<int, bool> fs = i => {
  int rest = % 2;
  return rest == 0;
}
```

5.4.2. Parameter & Type Inference

Lambdas können 0 bis n Parameter haben. Sie können als ***`Func`*** oder ***`Action`*** definiert sein *(mit/ohne Rückgabetyp)*

Regeln: Klammern um Parameter *(ausser bei genau einem Parameter oder wenn Parametertyp aus Delegate abgeleitet werden kann)*, `ref` / `out` sind erlaubt. Jeder Parameter muss implizit in jeweiligen Delegate-Parameter konvertierbar sein.

Typen sind meist redundant definiert: In der Deklaration werden hier die Typen schon angegeben und müssen in der Parameterliste nicht wiederholt werden.

```
Func<bool> p1; // 0 Parameter, returnt bool
Func<int, bool> p2; // 1 int Parameter, returnt bool
Func<int, int, bool> p3;
p1 = () => true;
p2 = a => true; // Ausnahme mit Klammern
p2 = (a) => true;
p3 = (a, b) => true;
// mit Angabe des Parametertyps
p2 = int a => true; // Compilerfehler
p2 = (int a) => true;
p3 = (int a, int b) => true;
```

5.4.3. Closures

Verwendet eine Methode eines Delegates ***lokale Variablen***, würde dies normalerweise zu Problemen führen, da die Runtime diese ***nach einem Aufruf der Funktion vernichten würde***. Um dies zu verhindern, gibt es Closures.
In einem Closure können Funktionen ***von aussen*** auf ***lokale Variablen innerhalb*** einer Delegate-Funktion zugreifen. Dazu wird die Variable vom Compiler in ein Hilfsobjekt ausgelagert. *(eigene Klasse, welche die Logik und die Variablen des Delegates speichert)*.
Die ***Lebenszeit*** des Hilfsobjekt ist gleich lang wie die des Delegate-Objekts.

Im ***Beispiel*** wird in der Funktion `CreateAdder()` ein Delegate returnt, in welchem die lokale Variable `x` inkrementiert wird *(`++x`)*. Adder ist also ein Closure. Dann wird zwei Mal `add()` aufgerufen. Es wird nur beim ersten Mal `x = 0` initialisiert.

```
public delegate int Adder();

class ClosureExample {
  static void Test() {
    Adder add = CreateAdder();
    Console.WriteLine(add()); // Output: "1"
    Console.WriteLine(add()); // Output: "2"
  }

  static Adder CreateAdder() {
    int x = 0; // nur bei 1. Aufruf initialisiert
    return () => ++x; // wird jedes Mal inkrement.
  }
}
```

5.5. ZUSAMMENFASSUNG DELEGATES & LAMBDA

```
public delegate void MyDel(string sender);
public class Summary {
  void Print(string sender)
    => Console.WriteLine(sender);
  static void PrintStatic(string sender)
    => Console.WriteLine(sender);
  void Test(){
    MyDel x1;
    /* Aufrufe siehe rechts */
  }
}
```

```
// Standard (Instanzmethode)
x1 = new MyDel(this.Print);
x1 = this.Print;
// Standard (Statisch)
x1 = new MyDel(Summary.PrintStatic);
x1 = Examples.PrintStatic;
// Expression Lambda
x1 = sender => Console.WriteLine(sender);
// Statement Lambda
x1 = sender => { Console.WriteLine(sender); }
```

6. GENERICS, NULLABILITY & RECORDS

6.1. GENERICS

Generics erlauben ***typsichere Datenstrukturen*** ohne Festlegung auf einen bestimmten Typen. Erlaubt die Implementation von Strukturen ohne die Verwendung von `object`. Stattdessen wird ***`T` (generischer Typparameter)*** verwendet. `T` wird dabei beim Aufruf einer Klasse/Funktion festgelegt. Innerhalb einer generischen Klasse ist ***`T` immer derselbe deklarierte Typ*** *(z.B. `string`)*.
Vorteile: Hohe Wiederverwendbarkeit, Typsicherheit *(bei `object` können sämtliche Typen übergeben werden)*, Performance *(Kein Boxing/Casting nötig)*. Hauptanwendungsfall sind Collections.

```
public class Buffer<T>
{
  T[] _items;

  public void Put(T item) { ... }
  public T Get() { ... }
}
// Verwendung
Buffer<int> buffer = new();
buffer.Put(1);
int i = buffer.Get();
```

6.1.1. Regeln für Generics

`T` kann intern wie ***normaler Typ*** verwendet werden.
Generisch sein können: Klasse, Struct, Interface, Delegates, Events, einzelne Methoden *(auch wenn Klasse nicht generisch)*.
Deklaration immer nach Name des Elements in spitzen Klammern. `Class` und `Class<T>` sind voneinander ***unabhängig*** und darum nicht direkt ineinander castbar.

Beliebige Anzahl generische Typparameter erlaubt:
`Buffer<T1, T2, TElement>` *(Bei mehreren wird empfohlen, aussagekräftige Namen, welche mit T beginnen, zu verwenden)*

```
public class Buffer<TElement, TPriority>
{
  TElement[] _items;
  TPriority[] _priorities;
  public void Put(TElement item, TPriority prio)
  { ... }
}
// Verwendung
Buffer<string, float> b = new();
b.Put("Hello", 0.3f);
b.Put("World", 0.2f);
```

6.1.2. Generic Type Inference

Redundante Typparameter können bei Methoden ***weggelassen*** werden, der Compiler fügt sie automatisch ein. Dies funktioniert aber nur, wenn `T` ein ***Parameter*** ist. Wenn `T` nur Return Type oder nicht in Signatur vorhanden ist, muss der Typ zwingend angegeben werden.

```
public void Print<T>(T t) { ... }
public T Get<T>() { ... }
public void TypeInference() {
  Print<int>(12); Print(12); // beides OK
  int i1 = Get<int>(); // OK
  int i2 = Get(); /* Compilerfehler */ }
```

6.1.3. Type Constraints

Bei Generics können alle Member von `object` als `T` übergeben werden. Möchte man dies einschränken, kann ein ***Type Constraint*** verwendet werden. Dies wird bei der Anwendung vom Compiler überprüft.
Beispielsweise kann mit
`where TPriority : IComparable`
spezifiziert werden, dass `TPriority` das `IComparable` Interface implementieren muss. Dadurch hat `TPriority` im Generic Zugriff auf die Methoden von `IComparable` *(`Equals()`, `CompareTo()`, …)*

```
class OrderedBuffer<TElement, TPriority>
  where TPriority : IComparable {
  TElement[] _data;
  TPriority[] _prio;
  int _lastElem;
  public void Put(TElement x, TPriority p) {
    int i = _lastELem;
    while (i ⩾ 0 && p.CompareTo(_prio[i]) > 0) {
      _data[i + 1] = _data[i];
      _prio[i + 1] = _prio[i]; i--;
    }
    data[i + 1] = x; _prio[i + 1] = p;
} }
```

Constraint	***Beschreibung***
`where T : struct`	T muss ein ***Value Type*** sein *(T muss auf Stack oder inline in Objekt liegen, T kann nie `null` sein, `null`-Check nicht erlaubt)*
`where T : class`	T muss ein ***Reference Type*** *(Klasse, Interface, Delegate)* sein *(T liegt auf Heap, T kann `null` sein)*
`where T : new()`	T muss ***parameterlosen `public` Konstruktor*** haben. *(T muss instanzierbar sein)* *(Dieser Constraint muss zuletzt aufgeführt werden)*
`where T : Name`	T muss von Klasse `Name` ableiten oder Interface `Name` implementieren *(T hat Zugriff auf alle Member. Es sollte meistens auf ein Interface anstatt auf eine Klasse eingeschränkt werden, da es dem Caller mehr Freiheiten bietet)*
`where T : TOther`	T muss mit `TOther` identisch sein bzw. davon ableiten *(auch **Naked Type Constraint** genannt, nur sehr spezifische Anwendungen, siehe Kapitel «Extension Methods» (Seite 32))*
`where T : class?`	T muss ein Nullable Reference Type sein *(Klasse, Interface, Delegate)* *(siehe «Nullable Reference Types (Klassen)» (Seite 27))*
`where T : not null`	T muss ein Non-Nullable Value/Reference Type sein

Kombination von Type Constraints

Sollen ***mehrere Type Constraints*** verwendet werden, muss für jeden Parameter eine eigene `where`-Klausel geschrieben werden. Mehrere `where` für einen Typparameter sind nicht erlaubt. Allfälliger `new`-Constraint muss am Ende der `where`-Klausel stehen.

```
class MultiConstraints<T1, T2>
  where T1 : struct
  where T2 : Buffer, IEnumerable<T1>, new()
  {
    /* ... */
  }
```

6.1.4. Generics Behind the Scenes

Konkretisierung mit Value Types	***Konkretisierung mit Reference Types***

`Buffer<int> a = new();`
1. CLR erzeugt zur Laufzeit `Buffer<int>`
2. `T` wird durch `int` ersetzt
 	
`Buffer<int> b = new();`
3. Wiederverwendung von `Buffer<int>`;
 	
`Buffer<float> c = new();`
4. CLR erzeugt zur Laufzeit `Buffer<float>`
5. `T` wird durch `float` ersetzt
	`Buffer<string> a = new();`
	1. CLR erzeugt zur Laufzeit `Buffer<object>`
	2. Wird für alle Reference Types wiederverwendet, da nur Referenzen gespeichert werden
	
	`Buffer<string> b = new();`
	3. Wiederverwendung von `Buffer<object>`
	
	`Buffer<Node> c = new();`
	4. Wiederverwendung von `Buffer<object>`

6.2. VERERBUNG BEI GENERICS

Generische Klassen könne von anderen generischen Klassen erben. Es gibt folgenden Möglichkeiten für Basistypen:

- ***Normale Klassen:*** `class MyList<T> : List { }`
  Die Zuweisung an eine Variable eines nicht-generischen Basistyp ist immer möglich.
- ***Weitergabe des Typparameters an generische Basisklasse:*** `class MyList<T> : List<T> { }`
  Die Zuweisung an eine Variable ist nur möglich, wenn Typparameter kompatibel sind.
- ***Konkretisierte generische Basisklasse:*** `class MyIntList : List<int> {}`
- ***Mischform:*** `MyIntKeyDict<T> : Dictionary<int, T> { }`

Typparameter werden nicht vererbt: `class MyList : List<T> { } // Compilerfehler`
Hier ist `T` dem Compiler nicht bekannt. Entweder muss `T` durch einen konkreten Typen ersetzt werden, oder die Basisklasse generisch gemacht und der Parameter `T` an sie weitergegeben werden.

Type Checks und Casts funktionieren gleich wie bei normalen Typen. Reflection unterstützt Abfrage von konkretisierten und nicht konkretisierten Typparametern.

6.2.1. Methoden überschreiben

Generische Basisklasse
T wird allen Overrides übergeben

```
class Buffer<T>
{
  public virtual void Put(T x)
    { ... }
}
```

Konkretisierte Basisklasse
T wird durch `int` ersetzt

```
class MyIntBuffer : Buffer<int>
{
  public override void Put(int x)
    { ... }
}
```

Generische Vererbung
T bleibt generisch

```
class MyBuffer<T> : Buffer<T>
{
  public override void Put(T x)
    { ... }
}
```

6.3. GENERISCHE DELEGATES

Der Code aus «Beispiel: Funktionsparameter» (Seite 22) kann generisch gemacht werden. Das Delegate, die `ForAll()`-Methode und die `PrintValues`-Action können nun mit sämtlichen Typen umgehen:

Test-Code

```
int[] a1 = { 1, 5, 8, 14, 22 };
string[] a2 = { "a", "b", "c" };
ForAll(a1, MyClass.PrintValues);
ForAll(a2, MyClass.PrintValues);
```

```
public delegate void Action<T>(T i);
public class MyClass {
  public static void PrintValues<T>(T i)
   { Console.WriteLine($"Value {i}"); }
}
public class FunctionParameter {
  static void ForAll<T>(T[] arr, Action<T> action) {
    Console.WriteLine("ForAll called");
    if (action == null) { return; }
    foreach (T t in arr) {
      action(t);
} } }
```

6.3.1. Delegate-Typen

Delegate	*Parameter*	*Return Type*	*Signatur*

Aktion	0 bis n	`void`	`public delegate void Action();`
			`public delegate void Action<in T1, ... , in T16>`
			`(T1 obj1, ..., T16 obj16);`

Funktion	0 bis n	non-`void`	`public delegate TResult Func<out TResult>();`
			`public delegate TResult Func<in T1, ..., in T16, out TResult>`
			`(T1 obj1, ..., T16 obj16);`

Prädikat	1	`bool`	`public delegate bool Predicate<in T>(T obj);`

EventHandler	`object sender, TEventArgs e`	`void`	`public delegate void EventHandler<TEventArgs>`
			`(object sender, TEventArgs e);`
			`public delegate void EventHandler(object sender, EventArgs e);`

6.4. GENERISCHE COLLECTIONS

Collections sind Typen, welche ***mehrere Elemente eines Typs*** enthalten. Alle Collections leiten von ***`ICollection<T>`*** ab, welches Methoden zur Collection-Manipulation zur Verfügung stellt *(`Add(T)`, `Remove(T)`, `Contains(T)` etc.)*.
`ICollection<T>` implementiert wiederum `IEnumerable<T>`, was Iteration per `foreach`-Loop ermöglicht, siehe Kapitel «Iteratoren» (Seite 31)
Die meisten Klassen und Interfaces aus `System.Collections` haben eine ***generische*** und eine ***nicht-generische Variante***. Häufig verwendet man auch nur die generischen Varianten.
Häufig verwendet: `List<T>`, `SortedList<T>`, `Dictionary<TKey, TValue>` *(Hashtable)*, `SortedDictionary<T>`, `LinkedList<T>`, `Stack<T>`, `Queue<T>`, `IEnumerable<T>`, `ICollection`

6.5. NULLABILITY

Es gibt spezielle Typen, welche `null` als eine Art ***«unbekannter»*** bzw. «nicht existenter» ***Wert*** zulassen. Diese Typen werden ***Nullable Types*** gennant.

6.5.1. `default` Operator

Der Default Operator liefert den Default-Wert für den angegebenen Typen *(Referenz-Typen: `null`, Zahlentypen: `0`, Bool: `false`, Enum: `0` zu Enum gecastet)*. Dies ist vor allem ***für Generics nützlich***, da so für jeden Typ automatisch der entsprechende Default-Typ gefunden werden kann.

```
public void NullExamples<T>() {
  T x1 = null;        // 2x Compilerfehler, er weiss
  T x2 = 0;           // nicht, ob Typ kompatibel ist
  T x3 = default(T); // OK
  T x4 = default;     // OK
}
```

6.6. NULLABLE VALUE TYPES (STRUCTS)

Value Types kann dank Generics ***`null` zugewiesen werden***. Dazu kann der Typ in die Wrapper-Klasse `System.Nullable<T>` gepackt werden. Diese besitzt ein ***Property `value`***, welches den tatsächlichen Wert enthält und ***`HasNull`***, welches angibt, ob der Wert `null` ist. Ist der Wert `null`, ist der Wert in `value` undefiniert.
Wird im Status `null` auf `Value` zugegriffen, erhält man eine `InvalidOperationException`.

```
public struct Nullable<T>
  where T : struct
{
  public Nullable(T value);

  public bool HasValue { get; }
  public T Value { get; }
}
```

6.6.1. `T?` Syntax

Syntactic Sugar, um die Lesbarkeit von Nullable Types zu erhöhen. Direkte Zuweisung von `null` möglich.

```
int? x = 123;  // new System.Nullable<int>(123);
int? y = null; // new System.Nullable<int>();
```

6.6.2. Sicheres Lesen & Type Casts von Value Types

Lesen

```
int? x = null;
// Klassisch
int x1 = x.HasValue ? x.Value : default;
// Via Methode
int x2 = x.GetValueOrDefault();
// Via Methode & eigenem Default
int x3 = x.GetValueOrDefault(-1);
```

Type Casts

```
int i = 123;
int? x = i;
int j = (int)x;
// Compiler-Output der obigen Statements:
int i = 123;
int? x = i;       // Kein Problem
int j = x.Value; // Schlecht, wenn x null ist
```

6.6.3. Nullability und Operatoren

```
int x = 1;
int? y = 2;
int? z = null;
```

Ist `int?` eine Zahl, verhält sich der Typ wie `int`.
">" und "<" funktionieren nur bei Typen, die diese Operatoren implementiert haben, also z.B. bei `int`.

Expression	***Ausdruckstyp***	***Resultat***
`x + z`	`int?`	`null`
`x + null`	`int?`	`null`
`x + null < y`	`bool`	`false`
`x + null == z`	`bool`	`true`
`x + null ⩾ z`	`bool`	`false`

6.7. NULLABLE REFERENCE TYPES (KLASSEN)

Die ***«null-state analysis»*** ist ein reines Compilerfeature, welches auf ***«static flow analysis»*** basiert *(Kann Objekt `null` sein?)*. Die Implementation ***unterscheidet sich*** massiv von den ***Nullable Value Types***. Generiert Compilerwarnungen, wenn in der `*.csproj` oder dem Präprozessor mit `#nullable enable` aktiviert. Sanfter Ansatz, da nicht alle .NET Libraries direkt umgestellt werden können.

Die Analyse nimmt an, dass ***jede Referenz `null` sein könnte***, ausser sie stammt aus einem Assembly, welches bereits mit null-state analysis geprüft wurde oder die Referenz wurde in der eigenen Codebase ***schon auf `null` geprüft***.

Der ***Typ einer Reference Variable*** kann wie bei den Value Types mit einem ***Fragezeichen*** versehen werden: ***`string?`***. Ebenfalls möglich ist der ***Null-forgiving Operator `!`*** hinter einem Wert oder einer Variable, welche explizit erlaubt, eine non-nullable Variable auf `null` zu setzen. ***Extrem gefährlich***, sollte nur mit Bedacht eingesetzt werden, da damit Compiletime und Runtime inkonsistent werden.

```
string? nameNull = null;
string name = nameNull; // Warning

if (nameNull is null) { // oder `nameNull == null`
  name = nameNull;      // Warning
  name = nameNull!;     // OK, but wrong
} else {
  name = nameNull;      // OK
}
```

6.8. NULLABLE SYNTACTIC SUGAR

Um den Umgang mit Nullable Types zu erleichtern, gibt es ***eigene Operatoren*** für die `null`-Handhabung.

6.8.1. `is null` / `is not null`

Mit den `is null` / `is not null` Operatoren kann auf `null` überprüft werden. ***Bei Value Types*** wird `HasValue` abgefragt, ***bei Reference Types*** *(auch ohne `?`)* mit `ReferenceEquals()`, ob es sich um eine Null-Referenz handelt. ***`obj is null`*** ist ***`obj == null`*** vorzuziehen, da == manuell überschrieben worden sein könnte und dadurch evtl. abweichende Logik besitzt.

```
int? x = null; string s = null;
bool b1a = x is null; bool b1b = x is not null;
bool b2a = s is null; bool b2b = s is not null;
// Compiler Output:
bool b1a = x.HasValue; bool b1a = !x.HasValue;
bool b2a = object.ReferenceEquals(x, null);
bool b2b = !object.ReferenceEquals(x, null);
```

6.8.2. Null-coalescing operator: `??`

Binärer Operator, welcher den ***linken Teil zurückgibt***, wenn dieser ***nicht `null`*** ist. ***Ansonsten*** wird der ***rechte Teil*** zurückgegeben. Kann z.B. dazu verwendet werden, bei `null` Exceptions zu werfen oder einen Default-Wert zu spezifizieren.

```
int? x = null;
int i = x ?? -1; // x wenn null, sonst -1
// Compiler Output:
int i = x is not null
  ? x.GetValueOrDefault()
  : -1;
```

6.8.3. Null-coalescing assignment operator: `??=`

Unärer Operator, welcher der ***Variable links*** den ***Wert rechts zuweist***, wenn die Variable `null` ist. Hat dasselbe Verhalten wie die anderen Assignment-Operatoren.

```
int? i = null; i ??= -1;
// Compiler Output:
if (i is null) { i = -1; }
```

6.8.4. Null-conditional operator: `?.`

Führt den ***rechten Teil*** aus, sofern der ***linke Teil nicht `null`*** ist. Ansonsten wird ein Default-Wert generiert. Funktioniert auch mit Delegates via `.Invoke()`. Nützlich, um auf Member eines Objekts zuzugreifen, welches `null` sein könnte.

```
object o = null; Action a = null;
string s = o?.ToString();
a?.Invoke();
// Compiler Output:
string s = o is not null ? o.ToString() : default;
if (a is not null) { a(); }
```

6.9. RECORD TYPES

```
public record [class|struct] Person(int Id, string Name);
```

Ein Record ist eine reine ***Datenrepräsentationsklasse***, welche nur initialisierbar ist *(immutable)*. Vereinfacht Arbeit mit nullable Reference Types. ***Compiler generiert eigene Klasse mit diversen Member*** *(Konstruktor, read-only Properties, `Equals` und ==, ≠ Operatoren, `ToString()`)*. Ein Record kann nur initialisiert werden, wenn alle Properties angegeben werden. Vererbungen werden berücksichtigt.

Verwendungszweck: Klassen ohne Methoden, z. B. für Werte aus DBs, oder Werte, die nur im Programm hin und hergeschoben werden.

6.9.1. Manuelle Deklaration

Kann wie ***normale Klasse*** deklariert werden. Der Nutzen ist so aber beschränkt, es werden so nur die Value Equality und `ToString()` generiert. Dafür ist es möglich, eigene Konstruktoren zu erstellen. Grundsätzlich aber besser den Positional Syntax verwenden.

Anwendung

```
Person p1 = new();
Person p2 = new(1, "Nina");
```

```
public record Person {
  // Eigener Default Konstruktor
  public Person() : this(0, "") { ... }
  public Person(int id, string name) {
    Id = id;
    Name = name;
  }
  public int Id { get; init; }
  public string Name { get; init; }
}
```

6.9.2. Gemischte Deklaration mit Positional Syntax

Die vom Record generierten Properties werden als ***init-only-setter*** erstellt. Positionale Deklaration kann zusätzlich zu benötigten Properties ergänzt werden, hier `Name` und `DoSomething()`.

Zu beachten gilt, dass `Name` ***non-nullable*** ist und eine Compilerwarnung generiert, wenn nicht gesetzt.

```
public record Person(int Id) {
  // Id implizit durch Constructor deklariert
  public string Name { get; init; }
  public void DoSomething() { }
}
Person p1 = new(0);  // OK, aber Compilerwarnung
p1.Id; p1.Name = ""; // 2x Compilerfehler
Person p2 = new(0) { Name = "" }; // OK
```

6.9.3. Deklaration mit Vererbung

Records unterstützen Vererbung, dazu muss die Basisklasse ebenfalls ein Record sein. Der zu aufrufende Konstruktor der Basisklasse wird hinter dem Record-Header angegeben. Subklassen können Basisklassen zugewiesen werden.

```
public abstract record Person(int Id);
public record SpecialPerson
  (int Id, string Name) : Person(Id);
SpecialPerson p1 = new(1, "Nina");
Person p2 = p1;
```

6.9.4. Value Equality

Code zur Value Equality wird vom ***Record automatisch generiert***. Vergleicht sämtliche Werte der Properties, also keine Reference Equality. Auch Properties allfälliger Basisklassen werden verglichen. Soll unterschieden werden, ob es zwei unterschiedliche Instanzen sind, kann ***`ReferenceEquals()`*** verwendet werden.

```
public record Person(int Id, string Name);
Person p1 = new(0, "Nina");
Person p2 = new(0, "Nina");

bool eq1 = p1 == p2; // true, false bei !=
bool eq2 = p1.Equals(p2); // true
bool eq3 = ReferenceEquals(p2); // false
```

6.10. VERÄNDERN VON RECORDS

Records können zwar ***nicht verändert werden***, aber beim Zuweisen an eine neue Variable kann ***nondestructive mutation*** durchgeführt werden. Mit dem Keyword ***`with`*** können einzelne Properties angepasst werden, was das Erzeugen leicht veränderter Records erleichtert.

```
public record Person(int Id, string Name);
Person p1 = new(0, "Nina");
person p2 = p1 with { Id = 1 }; // neue Id
bool eq1 = p1 == p2; // false
Person p3 = p1 with { }; // exakte Kopie
bool eq3 = p1 == p3; // true
```

7. EXCEPTIONS & ITERATOREN

7.1. EXCEPTIONS

Exceptions behandeln ***unerwartete Programmzustände*** oder ***Ausnahmeverhalten*** zur Laufzeit. Im Gegensatz zu Java muss der Aufrufer einer Methode Exceptions dieser nicht unbedingt behandeln (sogenannte ***«Unchecked Exceptions»***). Auch muss nicht an der Signatur angegeben werden, welche Art von Exceptions eine Methode wirft. Dies ist zwar kürzer und bequemer, aber auch weniger sicher und robust.

Best Practices:

- Exceptions sollten ***Ausnahmen*** sein und nicht für den «normalen» Programmfluss verwendet werden
- Wenn möglich sollten ***Vorbedingungen geprüft*** werden, um Exceptions zu ***vermeiden***
- Exceptions sind ***«Fehlercodes» vorzuziehen*** *(z.B. bei Fehler -1 returnen)*
- ***Konkrete Fehlerbeschreibung*** *(Möglichst konkrete Exception-Klasse verwenden, detaillierte Exception Message)*
- ***Nie*** Fehler über ***Web-Schnittstelle*** übermitteln *(offenbart Internas & erhöht Verletzbarkeit des Systems)*
- ***Aufräumen*** bei Exceptions *(Sockets, File Handles, Transaktionen schliessen/beenden)*

Exception Handling basiert auf ***folgenden Keywords:***

- ***`try:`*** Anweisungsblock, welcher Exception verursachen kann
- ***`catch:`*** Anweisungsblock, welcher eine bestimmte Exception behandelt
- ***`finally:`*** Anweisungsblock, welcher nach `try` und nach `catch` garantiert einmalig ausgeführt wird
- ***`throw:`*** Statement löst Exception aus *(im Gegensatz zu Java nicht zusätzlich bei Methodensignatur nötig)*

`try-catch`-Blöcke sind relativ ***performanceintensiv***.

```
FileStream s = null;
try {
  s = new(@"C:\urMom.mp4", FileMode.Open); ...;
} catch (FileNotFoundException e) {
  Console.WriteLine($"{e.FileName} not found");
} catch (IOException) {
  Console.WriteLine("I/O exception occurred");
} catch {
  Console.WriteLine("Unknown Exception occurred");
} finally {
  if (s != null) { s.Close(); }
}
```

Regeln

- Es wird ***sequenziell*** nach einem passenden `catch`-Block gesucht *(von oben nach unten)*
- Der ***Name*** des Exception-Objekts darf ***weggelassen*** werden, wenn nicht verwendet
- ***Eigene Exceptions*** müssen von `System.Exception` abgeleitet werden
- `finally`-Block wird ***immer*** ausgeführt, wenn vorhanden

7.1.1. Klasse `System.Exception`

Basisklasse für alle Exceptions. Hat Konstruktoren für Fehlerbeschreibung und allfällige Inner Exceptions.

Properties & Methoden:

- ***`InnerException`:*** Verschachtelte Exceptions
- ***`Message`:*** Menschenlesbare Fehlerbeschreibung
- ***`Source`:*** Name des Objekts, Frameworks etc., welches den Fehler verursachte
- ***`StackTrace`:*** Aktueller Call Stack als String
- ***`TargetSite`:*** Ausgeführter Code-Teil, der Fehler verursacht
- ***`ToString()`:*** Fehlermeldung & Stack Trace als String

```
public class Exception : ISerializable, ... {
  public Exception() { ... }
  public Exception(string message) { ... }
  public Exception(string message,
    Exception innerException){ ... }
  public Exception InnerException { get; }
  public virtual string Message { get; }
  public virtual string Source { get; set; }
  public virtual string StackTrace {get; set; }
  public MethodBase TargetSite { get; }
  public override string ToString();
}
```

7.1.2. `throw` Keyword

Kann ***implizit*** aufgerufen werden, wenn ungültige Operation durchgeführt wird oder Methode aufgerufen wird, welche eine ***unbehandelte Exception*** wirft.

Explizite Aufrufe können in ***eigenen Methoden*** implementiert werden.

```
// Division durch 0
int i = 0; int x1 = 12 / i;
// ungültiger Indexzugriff
int[] arr = new int[5]; int x2 = arr[12];
// null-Zugriff
object o = null; string x3 = o.ToString();
throw new Exception("An Error occured");
```

7.1.3. Exception Filters

`catch`-Block kann mit dem ***`when`-Keyword*** nur unter definierter Bedingung ausgeführt werden. Die Bedingung muss ***`bool`*** zurückgeben. Innerhalb der Bedingung ist der Zugriff auf das Exception-Objekt möglich.

```
try { ... }
catch (Exception e)
  when (DateTime.Now.Hour < 18) { ... }
catch (Exception e)
  when (DateTime.Now.Hour ⩾ 18) { ... }
```

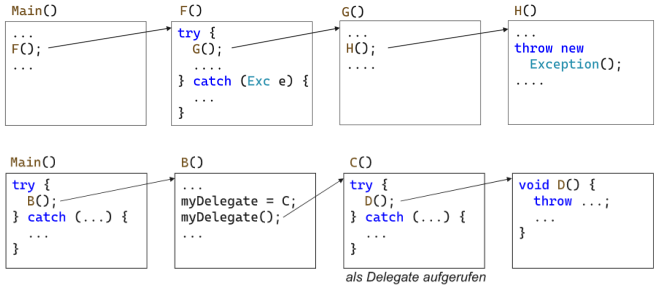
7.1.4. `catch-throw`-Block

Klassisch mit `throw e`	***Rethrowing mit `throw`***
Neuer Stack Trace wird begonnen	Stack Trace bleibt erhalten
`try { throw new Exception("Blöd gloffe"); } catch (Exception e) { throw e; // mit Weitergabe von e }`	`try { throw new Exception("Blöd gloffe"); } catch (Exception e) { throw; // ohne Weitergabe von e }`

7.2. SUCHE NACH CATCH-KLAUSEL

Call Stack wird ***rückwärts*** nach passender `catch`-Klausel durchsucht. Programm wird mit Fehlermeldung und Stack Trace ***abgebrochen***, falls keine gefunden.

Das selbe Prinzip gilt auch für ***Delegates:*** Sie werden wie normale Methoden behandelt. Etwas komplizierter wird das Finden bei Multicast Delegates.



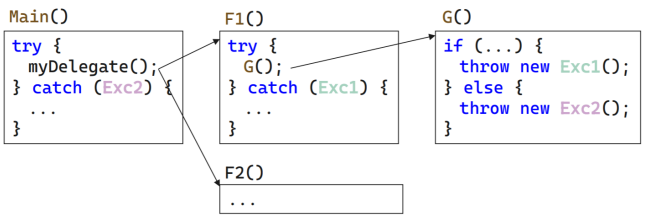
7.2.1. `catch` mit Multicast Delegates

Szenario 1:

- `Exc1` wird in `G()` ausgelöst
- `catch` für `Exc1` in `F1()` behandelt Ausnahme
- `F2()` wird aufgerufen *(nächste Delegate-Methode)*

Szenario 2:

- `Exc2` wird in `G()` ausgelöst
- Kein `catch` für `Exc2` gefunden
- `catch` für `Exc2` in `Main()` behandelt *(Delegate-Ausführung beendet)*



7.2.2. Beispiel: Argumente prüfen

Zwei verschiedene Exception-Typen
- ***ArgumentNullException***
  bei `null`-Werten
- ***ArgumentOutOfRangeException***
  bei ungültigen Wertebereichen

`nameof` wird zum Auslesen des Parameternamens verwendet und ist ***Refactoring-stabil.***
(Namensänderungen werden «übernommen»)

```
string Replicate(string s, int nTimes) {
  if (s is null) {
    throw new ArgumentNullException(nameof(s));
  } if (s.Length == 0) {
    throw new ArgumentOutOfRangeException(nameof(s));
  } if (nTimes ⩽ 1) {
    throw new ArgumentOutOfRangeException(nameof(nTimes));
  }
  return new StringBuilder().Insert(0, s, nTimes).ToString();
}
```

7.3. ITERATOREN

7.3.1. `foreach` Loop

Wird für das ***Iterieren über Collections*** verwendet. Mit `continue` wird zur nächsten Iteration fortgefahren, mit `break` wird der gesamte Loop beendet.

Syntax:

```
foreach (ElementType elem in collection) {
 /* statements */
}
```

Als Collection gilt, wenn ein Typ `IEnumerable` / `IEnumerable<T>` implementiert oder einer Implementation davon ähnelt *(Typ hat Methode `GetEnumerator()` mit Return Type e, e hat eine Methode `MoveNext()` mit Return Type `bool`, e hat ein Property `current`).*
Ein `foreach`-Loop ist nur ***Syntactic Sugar*** für einen Iterator.

```
List<int> list = new() { 1, 2, 3, 4, 5, 6 };
foreach (int i in list) {
  if (i == 3) continue;
  if (i == 5) break;
  Console.WriteLine(i);
} // Compiler Output:
IEnumerator enumerator = list.GetEnumerator():
try {
  while (enumerator.MoveNext()) {
    /* Statements aus foreach-Loop */
  }
}
finally {
  IDisposable disp = enumerator as IDisposable;
  if (disp ≠ null) disp.Dispose();
}
```

7.3.2. Iteratoren-Interface

Non-generic **(eher nicht verwenden)**

```
public interface IEnumerable {
  IEnumerator GetEnumerator();
}

public interface IEnumerator {
  object Current { get; } // Aktuelles Objekt
  bool MoveNext();        // Hole nächstes Element
  void Reset();           // zum Ausgangsstand zurück
}
```

Generic **(Best Practice)**

```
public interface IEnumerable<out T> : IEnumerable {
  IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<out T>
  : IDisposable, IEnumerator {
    T current { get; }
    // Member von IEnumerator geerbt
  }
```

7.3.3. Iteratoren-Zugriff

Es sind mehrere aktive Iteratoren auf denselben Objekt erlaubt. Das Enumerator-Objekt muss den Zustand der zu iterierenden Collection vollständig kapseln, ansonsten passieren unerwünschte Seiteneffekte. ***Implikation: Collection darf während Iteration nicht verändert werden.*** Soll das geschehen, muss eine zweite Collection angelegt werden.

7.4. YIELD

Da eine «rohe» Implementation sowohl eines generischen als auch nicht-generischen Iterators ***sehr aufwendig*** ist, gibt es das `yield` Keyword, um dessen Implementation zu vereinfachen, da es die entsprechenden `Enumerators` für uns generiert.

- ***`yield return`:*** Gibt den nächsten Wert für die nächste Iteration zurück. Beim nächsten Aufruf des Iterators wird von dieser Stelle an fortgefahren.
- ***`yield break`:*** Terminiert die Iteration

```
class MyIntList {
  public IEnumerator<int> GetEnumerator() {
    yield return 1;
    yield return 3;
    yield break;
    yield return 6;
  } // sehr simples Demonstrationsbeispiel
}   // mit hardcodierten Werten
// Returnt bei mehrmaligem Aufruf: 1 3
// (ohne 6, wegen `yield break`)
```

Nun muss nur noch die `GetEnumerator()`-Methode *(generisch oder nicht-generisch)* implementiert werden, welche mindestens ein `yield return` beinhaltet. Die Implementation von `IEnumerable` ist optional, aber empfohlen.

7.5. SPEZIFISCHE ITERATOREN

Standard Iterator

Hier wird `yield return` erstmals in einem Loop verwendet und gibt jedes Element der Liste einmal zurück. Damit können Instanzen von `MyIntList` als Collection in einem `foreach`-Loop verwendet werden. `MyIntList l = new(); foreach(int e in l) { ... }`

Spezifische Iterator-Methode

Mit der ***Range-Methode*** kann über einen Teil der Collection iteriert werden. Der Return Type ändert sich von `IEnumerator<T>` zu `IEnumerable<T>`. Um diesen Iterator im `foreach` zu erhalten, muss die spezifische Methode aufgerufen werden.
`foreach (int e in l.Range(2, 7)) { ... }`

Spezifisches Iterator-Property

Funktioniert ähnlich wie Methode, Property muss ebenfalls spezifisch aufgerufen werden. Eher selten verwendet.
`foreach (int e in l.Reverse) { ... }`

```
class MyIntList {
  private int[] _x = new int[10];
  // Standard Iterator
  public IEnumerator<int> GetEnumerator() {
    for (int i = 0; i < _x.Length, i++) {
      yield return _x[i];
    } // yield break ist implizit
  }
  // Spezifische Iterator-Methode
  public IEnumerable<int> Range(int from, int to) {
    for (int i = from, i < to, i++) {
      yield return _x[i];
    }
  }
  // Spezifisches Iterator-Property
  public IEnumerable<int> Reverse {
    get {
      for (int i = _x.Length - 1; i ⩾ 0; i--)
      {
        yield return _x[i];
      } } } }
```

7.5.1. Anleitung: Normale Methode zu Iterator Methode umschreiben

1. Verwendeter Collection-Typ in Signatur zu ***IEnumerable*** umändern
2. Variable der Liste und ***return-Statements*** entfernen, stattdessen überall, wo `Add()` o.ä. verwendet wird, durch ein ***yield return*** ersetzen.

7.6. EXTENSION METHODS

Extension Methods erlauben das ***Erweitern bestehender Klassen***. Die Signatur der Klasse wird dadurch nicht verändert. Der Aufruf sieht jedoch so aus, als wäre es eine Methode der Klasse, der Compiler wandelt ihn in einen normalen statischen Methodenaufruf um. Typsicherheit ist gewährleistet. ***Importieren der Klasse in Verwendungsort mit `using <classname>`.***

Deklaration:

- Muss in ***statischer Klasse*** deklariert sein
- Die Methode muss selbst ***`static`*** sein
- Der ***erste Parameter*** muss angeben, auf welcher Klasse die Methode verwendbar ist und das ***`this`-Keyword*** muss davorstehen.

```
public static class ExtensionMethods {
  static string ToStringSafe(this object obj) {
    return obj == null
      ? string.Empty : obj.ToString();
  }

  public static void TestExtensions() {
    int i = 0;
    i.ToString(); i.ToStringSafe(); // Beide OK
    object nullObj = null;
    nullObj.ToString();      // Runtime Error
    nullObj.ToStringSafe(); // OK
  }
}
```

Regeln: ***Kein Zugriff auf interne Member*** aus Extension Method heraus, Extension Method ist ***nur sichtbar***, wenn der ***Namespace importiert*** wird, in welcher die Methode definiert wurde *(hier `ExtensionMethods`)*. Bei ***Namenskonflikten*** gewinnt immer die Non-Extension-Methode. Erlaubt auf Klassen, Structs, Interfaces, Delegates, Enumeratoren und Arrays.

7.7. DEFERRED EVALUATION

Ein Aufruf von `GetEnumerator()` oder `Range()` führt die Iterator-Methode noch nicht aus. Erst der Aufruf von `IEnumerator<T>.MoveNext()` tut dies. Im `foreach`-Loop wird sie implizit aufgerufen.

```
MyIntList list = new();
// Keine Ausführung
IEnumerator<int> er1 = list.GetEnumerator();
IEnumerable<int> range = list.Range(4, 8);
IEnumerator<int> er2 = range.GetEnumerator();

er1.MoveNext(); // Ausführung
int i1 = er1.Current;
er2.MoveNext(); // Ausführung
int i2 = range.GetEnumerator().Current;
foreach (int i in range) { ... } // Ausführung
```

```
class MyIntList {
  public IEnumerator<int> GetEnumerator() {
    for (int i = 0; i < _x.Length, i++) {
      yield return i;
    }
  }

  public IEnumerable<int> Range(int from, int to) {
    for (int i = from, i < to, i++) {
      yield return i;
  } }
```

7.7.1. Extension Methods und Iteratoren

Der ganze bisherige Stoff dieses Kapitels zielt darauf ab, für ***Collections neue Query-Operatoren*** schaffen zu können. Diese werden als Extension Method auf `IEnumerable` erstellt. Damit lassen sich alle Collections, die davon ableiten, um beispielweise ***Filter- oder Gruppierungsmethoden*** erweitern. ***Query-Methoden*** sollten immer einen `IEnumerable` zurückgeben, damit mehrere davon mit dem «.»-Operator aneinandergehängt werden können. Sie werden ebenfalls «deferred» evaluiert.

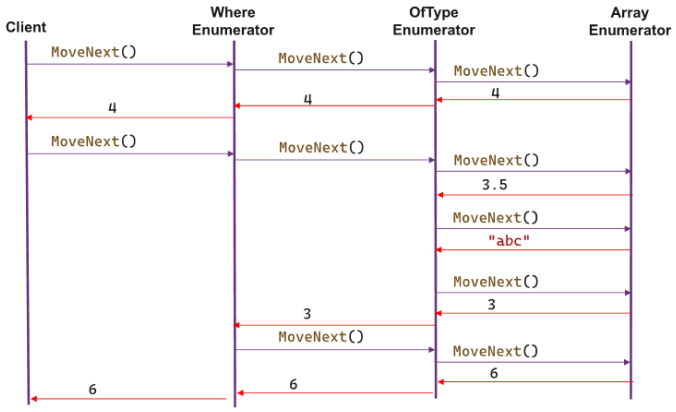
```
public static void Test() {
  object[] list { 4, 3.5, "abc", 3, 6 };

  // Hier noch keine Ausführung auf dem Array
  IEnumerable<int> res = list
    .QueryOfType<int>()
    .QueryWhere(k ⇒ k % 2 == 0);

  // Erst hier wird tatsächlich iteriert
  foreach (int i in res) { ... }

  // Schreibweise ohne Erweiterungssyntax
  // Verschachtelt, schlecht lesbar
  res = QueryWhere(
    QueryOfType<int>(list), k ⇒ k % 2 == 0));
  )
}
```

```
static class QueryExtensions {
  // Filtert Collection nach mitgegebener Funktion
  public static IEnumerable<T> QueryWhere<T>(
    this IEnumerable<T> source,
    Predicate<T> pred)
  {
    foreach (T item in source) {
      if (predicate(item)) { yield return item; }
    }
  }
  // Filtert Collection nach mitgegebenem Typ
  public static IEnumerable<T> QueryOfType<T>(
    this IEnumerable source)
  {
    foreach (object in source) {
      if (item is T) { yield return (T)item; }
} } }
```



8. LINQ (LANGUAGE INTEGRATED QUERY)

Mit LINQ können von beliebigen Datenstrukturen mit SQL-ähnlichen Queries Daten ***typsicher bearbeitet*** werden. LINQ erlaubt es ***Funktional*** *(durch Lambda Expressions)* und ***Deklarativ*** *(durch Anonymous Types und Object Initializers)* zu arbeiten und verbessert ***Type Inference*** *(Weglassen von redundanten Typ-Informationen)*.

LINQ ist die ***Schnittstelle*** zwischen den ***.NET-Sprachen*** und ***Datenquellen***. Die Architektur basiert auf dem Provider-Modell, das heisst, für jeden Datentyp gibt es einen eigenen LINQ-Provider *(LINQ to Objects, to Entities, to SQL, to XML etc.)*.

Beispiel Provider: LINQ to Objects

Im LINQ to Objects-Provider werden ***Abfragen auf die Objektstruktur*** durchgeführt. Es können mehrere Abfragen aneinandergekettet werden, es muss aber am Ende eine Immediate-Evaluation-Query stehen, um die Daten zu erhalten *(siehe Kapitel «Immediate Evaluation» (Seite 34))*.

Query 1 ist die simpelst mögliche LINQ-Abfrage: Es werden alle Elemente im Objekt ausgewählt und wieder zurückgegeben.

In ***Query 2*** wird zuerst nach Namen gefiltert, welche mit «B» beginnen, diese dann sortiert und zurückgegeben.

```
string[] cities = { "Bern", "Basel", "Zürich",
"Rapperswil", "Genf" };
// q = query expression, l = extension method
// Query 1: Reine Selektion
IEnumerable<string> q1 =
  from c in cities select c;
IEnumerable<string> l1 = cities.Select(c ⇒ c);

// Query 2: Alle Namen mit B sortieren
IEnumerable<string> q2 =
  from c in cities
  where c.StartsWith("b")
  orderby c
  select c;
IEnumerable<string> l2 = cities
  .Where(c ⇒ c.StartsWith("B"))
  .OrderBy(c ⇒ c).Select(c ⇒ c);
```

8.1. LINQ EXTENSION METHODS

LINQ ***definiert*** eine ***Vielzahl an Query Operatoren***, welche wie in Kapitel «Extension Methods und Iteratoren» (Seite 33) `IEnumerable<T>` implementieren und sind vom Aufbau her diesen sehr ähnlich.

Query Operatoren sind ebenfalls mit ***`yield return`*** implementiert, unterstützen also deferred evaluation. Queries können ***beliebig oft*** ausgeführt werden und die Resultate können sich aufgrund veränderter Daten auch ***unterscheiden***.

```
string[] cities = { "Bern", "Basel", "Zürich",
"Rapperswil", "Genf" };
IEnumerable<string> citiesB = cities
  .Where(c ⇒ c.StartsWith("B"));

// Run 1: iteriert durch 2 Städte (Bern, Basel)
foreach (string c in citiesB) { ... }

cities[0] = "Luzern";
// Run 2: iteriert nur durch 1 Stadt (Basel)
foreach (string c in citiesB) { ... }
```

8.1.1. Immediate Evaluation

Einige Methoden führen die gesamte Kette an Queries direkt aus. Queries mit ***Immediate Evaluation*** haben einen Return Type, welcher nicht `IEnumerable` ist.

Beispiele: `ToList()`, `ToArray()`, `Count()`, `First()`, `Last()`, `Sum()`, `Average()`

Die ***Resultate*** dieser Queries werden direkt in die ***Variable gespeichert***.

```
string[] cities = { "Bern", "Basel", "Zürich",
"Rapperswil", "Genf" };

List<string> citiesB = cities
  .Where(c ⇒ c.StartsWith("B"))
  .ToList(); // Ausführung

int citiesEndLCount = cities
  .Where(c ⇒ c.EndsWith("l"))
  .Count(); // Ausführung
```

8.1.2. Wichtigste Query Operatoren

Standard	*Positionelle Operatoren*	*Set Operationen*
`Select()`	`First()`, `FirstOrDefault()` *Erstes passendes Element für Prädikat*	`Distinct()` *Nur einzigartige Elemente*
`Where()`	`Single()`, `SingleOrDefault()` *Wie `First`, throwt wenn nicht genau 1 Element*	`Union()` *Einzigartige Elemente zweier Mengen*
`OrderBy()`, `OrderByDescending()`	`ElementAt()` *Element an angegebenem Index*	`Intersection()` *Überschneidende Elemente zweier Mengen*
`ThenBy()`, `ThenByDescending()`	`Skip()`, `Take()`, *Die nächsten n Elemente skippen/returnen*	`Except()` *Elemente aus Menge A, welche in B fehlen*
`GroupBy()`	`SkipWhile()`, `TakeWhile()` *Elemente skippen/returnen bei passendem Prädikat*	`Repeat()` *n-fache Kopie der Liste*
`Join()`, `GroupJoin()`	`Reverse()` *Alle Elemente in umgekehrter Reihenfolge*	
`Count()`, `Sum()`, `Min()`, `Max()`, `Average()`		

8.2. OBJECT INITIALIZERS

Hat man ***verschachtelte Objekte***, ist es sehr mühsam, diese zu initialisieren und die gewünschten Werte zu setzen. Um dies zu vereinfachen, gibt es die ***Object Initializers***. Sie erlauben das Instanzieren und Initialisieren einer Klasse in einem Statement, auch ohne passenden Konstruktor.

Syntax

– ***Default Constructor***
  `new class { member = value, ... }`
– ***Spezieller Konstruktor***
  `new class(args) { member = value, ... }`

Die Initializer werden vom Compiler in simple Member Assignments umgewandelt.

Object Initializers werden vor allem in ***Lambda Expressions*** häufig verwendet, meistens in Kombination mit der Extension-Methode `Select()`.

```
class Student {
  public string Name;
  public int Id;
  public string Subject { get; set; }
  public Student() { ... }
  public Student(string name) { Name = name }
}
Student s1 = new("Nina") {
  Id = 69420,
  Subject = "Multimedia Production"
}
Student s2 = new() {
  Id = 30035
  Name = "Jannis"
  Subject = "Master of Desaster"
}
// Generiert aus IDs 'Student' objects
int[] ids = { 404, 666, 2048 };
IEnumerable<Student> students = ids
  .Select(n ⇒ new Student { Id = n });
```

8.3. COLLECTION INITIALIZERS

Mit einer ähnlichen Syntax ist es auch möglich, ***Collections*** und ***Dictionaries*** zu ***initialisieren***. Dictionaries haben zwei verschiedene Syntaxen: regulär und via Indexer. Die resultierenden Dictionaries sind aber gleich.

Syntax

Collections

```
new collection { elem1, ..., elemN }
```

Dictionaries

```
new dictionary {
  { key1, val1 }, ..., { keyN, valN } };
new dictionary {
  { [key1], val1 }, ..., { [keyN], valN } };
```

```
List<int> l1 = new() { 1, 2, 3, 4 }
double[2, 2] a = {
  [0, 0] = 1.2, [0, 1] = 5.9,
  [1, 0] = 7.5, [1, 1] = 0.9
};

Dictionary<int, string> d = new() { // Regulär
  { 1, "a" },
  { 2, "b" },
  { 3, "c" }
};
Dictionary<int, string> d2 = new() { // via Index
  [1] = "a",
  [2] = "b",
  [3] = "c"
};
```

8.4. TYPE INFERENCE

Mit Type Inference kann der ***statische Typ*** einer Variable ***vom Compiler*** selbst bestimmt werden. Anstelle des Typen wird das ***Keyword*** `var` verwendet. Dazu müssen aber Deklaration und Initialisierung im gleichen Statement sein, da der ***Typ*** aus dieser ***initialen Zuweisung abgeleitet*** wird. Der Compiler ersetzt dann `var` durch den eigentlichen Typen. `var` kann nur für lokale Variablen, nicht aber für Parameter, Return Types, Klassenvariablen, Properties etc. verwendet werden.

```
// Entspricht string v
var v = "Hello";
// 2x Compilerfehler
v = 42;
var w;
```

Vorteile: Bessere Lesbarkeit des Codes *(lange Typen wie `Dictionary<string, IComparable<object>>` können bei der Deklaration wegfallen)*, erlaubt anonyme Typen.

8.5. ANONYMOUS TYPES

Zwischenresultate in Abfragen müssen irgendwie ***gespeichert*** werden, gelten aber meistens nur für diese Abfrage. Es macht keinen Sinn, dafür eine eigene Klasse bzw. Typ zu erstellen, aber es muss dennoch ein konkreter Typ dafür existieren.

Hier kommen Anonymous Types ins Spiel: Damit können strukturierte Werte ***erzeugt*** werden, ohne dafür einen eigenen Typ definieren zu müssen. Die ***Propertynamen*** können dabei ***Explizit*** *(Variable `a`)* oder ***implizit*** *(Variable `b`)* sein.

Die anonymen Typen von `a` und `b` sind ***identisch***, können also wiederverwendet werden.

```
var a = new { Id = 1, Name = "Nina" };
var b = new { a.Id, a.Name };
var query = studentList
  .GroupBy(s => s.Subject)
  .Select(grp => new {  // Neuer Anonymer Typ
    Subject = grp.Key,  // Subject-Name
    Count = grp.Count() // Anzahl
  });
```

Regeln: Die generierten Properties *(hier `Id, Name`)* sind ***read-only***, Implizite/Explizite Propertynamen können gemischt werden, Anonymer Typ direkt von `System.Object` abgeleitet *(hat deswegen `Equals(), GetHashCode(), ToString()`)*, kann nur einer Variable vom Typ `var` zugewiesen werden.

8.6. QUERY EXPRESSIONS

Neben den Extension Method gibt es noch eine zweite Syntax, um LINQ Queries zu schreiben: Die von SQL inspirierten Query Expressions. Es gibt folgende Query-Keywords:

- ***`from:`*** Definiert Range-Variable & Datenquelle
- ***`select:`*** Rückgabe durch Projektion auf Elementtypen
- ***`where:`*** Filter nach Kriterium
- ***`orderby:`*** Sortieren
- ***`group ... by ... [into]:`***
  Zusammenfassen von Elementen in Gruppen
- ***`join ... on:`*** Verknüpfung zweier Datenquellen
- ***`let:`*** Definition von Hilfsvariablen

```
int[] numbers = { 0, 1, 2, 3 };
var numQuery = from num in numbers
               where (num % 2) == 0
               select num;

var q1 = from s in students
         where s.Subject = "Informatik"
         orderby s.Name
         select new { s.Id, s.Name };
// wird vom Compiler umgewandelt in
var q1 = students
    .Where(s => s.Subject == "Informatik")
    .OrderBy(s => s.Name)
    .Select(s => new { s.Id, s.Name });
```

Eine Query ***beginnt*** immer mit ***`from`*** und ***endet*** mit ***`select`*** oder ***`group`***. Die Query Expressions werden vom Compiler wieder in die Extension Method Syntax umgewandelt.

Return Type bei `new{}`: `IEnumerable<AnonymousType{ /* return elements */ }>`

8.6.1. Range Variablen

Bei `from`, `join` und `into` werden ***Resultate*** in ***Range Variablen*** gespeichert. Diese sind ***read-only***. Sie dürfen nicht denselben Namen wie äussere lokale Variablen haben. Range Variablen sind innerhalb der Expression bis zur nächsten `into`-Klausel sichtbar.

```
// Range Variablen in Query: s, m, g
var q = from s in students
join m in Markings on s.Id equals m.StudentId
group s by s.Subject into g
select g;
```

8.6.2. Gruppierung

Besteht aus `group`, `by` und optional `into`-Keywords.
Die Werte werden zu ***Key/Value Pairs*** in ein ***`IGrouping<TKey, TElement>`*** *(Subklasse von `IEnumerable`)* zusammengefasst. Der Key ist der Wert nach dem `by`, die Values nach dem `group`. Im Beispiel wäre der Key `Subject` und die Values `Name`.

```
var q1 = from s in students
         group s.Name by s.Subject;
foreach (var group in q1) {
  Console.WriteLine(group.Key);
  foreach (var name in group) {
    Console.WriteLine($"- {name}"); } }
```

`into` speichert das Resultat in eine Variable `g`. ***`s`*** ist dann ***nicht mehr sichtbar***, `g` kann als Gruppe weiterverwendet werden. Es ist also nicht möglich, im anonymen Typen von `select` Werte hinzuzufügen, welche nicht im `group by`-Statement und damit in `g` gespeichert sind *(Keine Werte aus `s`)*.

```
var q2 = from s in students
         group s.Name by s.Subject into g
         select new
         { Field = g.Key, N = g.Count() };
foreach (var in q2)
  { Console.WriteLine($"{x.Field}: {x.N}") }
```

8.6.3. Explizite Inner Joins

Verknüpft zwei Mengen über einen Key mit den ***`join`*** und ***`on`***-Keywords. Verknüpfung muss zwingend über `equals` und nicht `==` erfolgen.

```
var q = from s in students
        join m in Markings
          on s.Id equals m.StudentId
        select s.Name + ", " + m.Course
                      + ", " + m.Mark;
```

8.6.4. Implizite Inner Joins

Verknüpft zwei Mengen über einen Key mit den ***`from`*** und ***`where`***-Keywords. Es kann sowohl `equals` als auch `==` verwendet werden. Weniger effizient, da Kreuzprodukt gebildet wird.

```
var q = from s in students
        from m in Markings
          where s.Id equals m.StudentId
        select s.Name + ", " + m.Course
                      + ", " + m.Mark;
```

8.6.5. Group Joins

Pro Student wird eine ***Liste aller Markings*** erstellt. Die Range Variable `s` bleibt sichtbar, im Gegensatz zum `group-by-into`-Ansatz.

```
var q = from s in students
        join m in Markings
          on s.Id equals m.StudentId
          into list
        select new { Name = s.Name, Marks = list };
```

8.6.6. Left Outer Joins

Verknüpft zwei Mengen über einen Key. Wenn kein rechtes Element gefunden wurde, verbleibt linkes Element trotzdem in der Liste.

Der Join basiert auf dem Group Join mit anschliessender `from`-Klausel. Hier wird mit `DefaultIfEmpty()` ein möglicher `null` Zugriff verhindert.

```
var q = from s in students
        join m in markings
          on s.Id equals m.StudentId
          into list
        from sm in list.DefaultIfEmpty()
        select s.Name + ", " + (sm == null
          ? "?"
          : sm.Course + ", " + sm.Mark);
```

8.6.7. `let`-Klausel

Erlaubt das ***Definieren von Hilfsvariablen***. Nützlich, um Zwischenresultate abzuspeichern. Kein direktes Equivalent in der Expression Syntax.

```
var q = from s in students
        let year = s.Id / 1000
        where year == 1996
        select s.Name + " " + year.ToString();
```

8.6.8. Select Many

Erleichtert das Zusammenfassen verschachtelter Listen.
Führt für jedes Listenelement auf oberster Stufe den Selector aus *(im Beispiel pro Zeile)*. Diese liefert selbst eine (Teil-)Liste. Teillisten werden untereinander gehängt.

Anwendung:

```
List<List<string>> list = new() {
  new() { "a", "b", "c" },
  new() { "1", "2", "3" },
  new() { "ä", "ö", "ü" }
};
var q1 = list.SelectMany(s ⇒ s);
// Equivalent zu:
var q2 = from segment in list
         from token in segment
         select token;
```

```
foreach (string line in q1)
  { Console.Write($"{line}.") }
// Output: a.b.c.1.2.3.ä.ö.ü.
```

9. TASKS & ASYNC/AWAIT

9.1. TASKS

Ein Task ist ein ***Platzhalter*** für ein Ergebnis, das noch nicht bekannt ist. Repräsentiert eine ***asynchrone*** Operation. Ist ein ***first-class citizen***.

Anwendungsfälle: Waiting, Returning Results, Cancellation, Pause / Resume, Composition of tasks, Processing errors, Debugging, …

```
Task task = Task
  .Run(() ⇒ { /* some workload */ } )
  .ContinueWith(t ⇒ 1234);

task.Wait();
```

Task	*Thread*
– Hat Rückgabewert – Unterstützt «Cancellation» via Token – Mehrere parallele Operationen in einem Task – Vereinfachter Programmfluss – eher ein «high level» Konstrukt	– Hat keinen Rückgabewert – Keine «Cancellation» – Nur eine Operation in einem Thread – Keine Unterstützung für async / await – eher ein «low level» Konstrukt

9.1.1. Task API

Starten eines Tasks: Via Factory *(t1, bietet weitere Optionen)* oder direkt via Task mit Default-Values *(t2)*.

Resultat abwarten: Busy Wait, Expliziter `Wait()` *(Unterstützt auch Timeouts)* oder via Awaiter *(Optimiertes Exception Handling)*

Achtung: Synchrone Waits sind gefährlich, weil sie den aktuellen Thread blockieren *(z.B. den UI Thread in GUI Applikationen)*.

Blockierende Task-APIs: `Task.Result`, `Task.Wait()`, `Task.WaitAll()`, etc.

```
Task<int> t1 = Task.Factory.StartNew(
  () ⇒ { Thread.Sleep(2000); return 1; }
);
Task<int> t2 = Task.Run(
  () ⇒ { Thread.Sleep(2000); return 1; }
);
// Busy wait for result (bad idea!)
while (!t1.IsCompleted) { /* Do other stuff */ }
int result1 = t1.Result;
// Explicit wait
t1.Wait(); int result2 = t1.Result;
// Using awaiter
int result3 = t1.GetAwaiter().GetResult();
```

9.1.2. Synchrone waits vs. Continuations

Synchrone Waits (blockierend)

```
Task<int> t1 = GetSomeCustomerIdAsync();
// Blockiert aktuellen Thread
int id = t1.Result;

Task<string> t2 = GetOrdersAsync(id);
// Blockiert aktuellen Thread
Console.WriteLine(t2.Result);
```

Continuations (nicht blockierend)

```
Task<int> t1 = GetSomeCustomerIdAsync();
t1.ContinueWith(id ⇒ {
  // Resultat schon vorhanden, nicht blockierend
  Task<string> t2 = GetOrdersAsync(id.Result);
  t2.ContinueWith(order ⇒
    // Resultat schon vorhanden, nicht blockierend
    Console.WriteLine(order.Result)
  );
});
```

9.2. ASYNC / AWAIT

Synchrone Operation	*Asynchrone Operation*
Durchläuft die gesamte Logik und returnt anschliessend aus der Methode. Blockiert den Aufrufer, bis diese fertig gelaufen ist.	Ruft eine Methode auf, ohne auf das Resultat zu warten. Möglichkeit zur Benachrichtigung bei Fertigstellung oder Rückgabe eines «Task»-Objekts, auf welchem Status abgefragt werden kann.

9.2.1. Async

Markiert die Methode als «asynchron». Hat ***eingeschränkte Rückgabewerte:*** `Task` *(ohne Rückgabewert)*, `Task<T>` *(Rückgabewert T)* und `void` *(Fire and Forget, sollte vermieden werden, da Erfolg des Tasks nicht überprüft werden kann)*

9.2.2. Await

Alles nach `"await"` wird vom Compiler zu einer «Continuation» umgewandelt. ***Nur in «async» Methoden erlaubt.***

9.2.3. Beispiel: Auslesen von Files

Liest zwei Dateien parallel aus, wartet nicht blockierend, verwendet Resultate.

```
Task<string> t1 = File.ReadAllTextAsync(@"C:\DotNetPrüfungLösungen.txt");
Task<string> t2 = File.ReadAllTextAsync(@"C:\BreathOfTheWildStrats.txt");
// do other stuff...
string[] allResults = await Task.WhenAll(t1, t2); // Blockiert aktuellen Thread nicht
// Resultate auslesen
Console.WriteLine(t1.Result); Console.WriteLine(t2.Result); // Zugriff wäre auch via `allResults` möglich
```

9.2.4. async / await vs. Continuation

async / await (nicht blockierend)

```
int id = await GetSomeCustomerIdAsync();
// Resultate direkt im Zugriff
string t2 = await GetOrdersAsync(id);
Console.WriteLine(t2);
```

Der Compiler generiert aus `async/await` im Hintergrund Continuations.

Continuations (nicht blockierend)

```
Task<int> t1 = GetSomeCustomerIdAsync();
t1.ContinueWith(id ⇒ {
  // Resultat schon vorhanden, nicht blockierend
  Task<string> t2 = GetOrdersAsync(id.Result);
  t2.ContinueWith(order ⇒
    // Resultat schon vorhanden, nicht blockierend
    Console.WriteLine(order.Result)
  ); });
```

9.2.5. Beurteilung

Vorteile	*Nachteile*
– Code sieht immer noch synchron aus – Keine Continuations nötig – Ersetzt Multithreading für asynchrone Ausführungen	– Overhead ist relativ gross – Lohnt sich daher erst bei «längeren» Operationen – `await` nicht erlaubt in lock-Statements und in älteren Versionen von `catch`- und `finally`-Blöcken

9.3. CANCELLATION SUPPORT

Ein ***Cancellation Token*** ist ein integriertes Programmiermodell für das Abbrechen von Programmlogik. Verwendet die Klasse ***`CancellationToken`***. Muss durch die gesamte Aufrufkette durchgereicht werden, letzter Parameter jeder asynchronen Methode.

Manueller Abbruch

Überprüfung, ob ein Abbruch angefordert wurde, geht via ***`IsCancellationRequested` Property*** oder via ***`ThrowIfCancellationRequested()` Methode***, welche eine `OperationCanceledException` wirft.

```
static async Task PauseAsync(CancellationToken ct) {
    // kann durch `ct` abgebrochen werden
    await Task.Delay(10_000, ct);
}
static void ManualCancellation(CancellationToken ct)
{
  for (int i = 0; i < 100; i++) { /* ... */
    // Property-Variante
    if (ct.IsCancellationRequested) { /* ... */ }
    // Methoden-Variante
    ct.ThrowIfCancellationRequested(); // Exception
  } }
```

9.3.1. Cancellation Token Source

Klasse zur Erstellung und Steuerung von Cancellation Tokens. Kann beliebig viele Tokens emittieren, diese sind dann an diese Source gebunden. Beim Auslösen des Abbruchs an einer Source wird für alle Tokens dieser Source ein Abbruch angefordert. Es sollte ein Token pro «Unit of Work» generiert werden *(alles was zusammenhängt)*

```
CancellationTokenSource cts = new();    // Emittierende Source
CancellationToken ct = cts.Token;       // Neuer Token generieren
Task t1 = LongRunning(1_000, ct);       // 1s tied to `cts`
Task t2 = LongRunning(3_000, ct);       // 3s tied to `cts`
Task t3 = LongRunning(3_001, default); // 3s independent / not cancellable

await Task.Delay(2_000, ct);

Console.WriteLine("Cancelling");        // Abbruch von `t1` & `t2`
cts.Cancel();                           // aber `t1` ist schon fertig,
Console.WriteLine("Canceled");          // darum wird nur `t2` abgebrochen

async Task LongRunning(int ms, CancellationToken ct) {
  Console.WriteLine($"{ms}ms Task started.");
  await Task.Delay(ms, ct);
  Console.WriteLine($"{ms}ms Task completed.");
};
```

```
Console
---------------------------
1000ms Task started.
3000ms Task started.
3001ms Task started.
1000ms Task completed.
Cancelling
Canceled
3001ms Task completed.
```

10. ENTITY FRAMEWORK

Das Entity Framework *(bzw. EF Core)* ist ein ***O/R Mapping Framework***. Es verbindet ***Objektorientiertes*** *(Domain Model)* mit ***Rationalem*** *(Relational Model)*. Das EF hat diverse ***Basis-Funktionalitäten:*** Mapping von Entitäten, CRUD Operationen *(Create, Read, Update, Delete)*, Key-Generierung, Caching, Change Tracking, Optimistic Concurrency, Transactions und eine eigene CLI.

Wie LINQ ist das Entity Framework ***providerbasiert.*** Es sind für die allermeisten relationalen SQL-DBs Provider auf NuGet verfügbar.

10.1. OR-MAPPING

Das OR-Mapping mappt .NET-Objekte auf relationale Tabellen in DBs. Dabei stellt eine .NET-Klasse einen ***Entity Type*** und die Tabelle *(oder anderes relationales Modell)* eine ***Storage Entity*** dar. Das Mapping sorgt für die Konvertierung.

- ***Entität:*** Strukturierter Datensatz mit einem Key, Instanz eines «Entity Type», gruppiert in «Entity Sets», kann von anderen Entitäten erben.
- ***Relationship/Association:*** Definiert Beziehung zwischen zwei Entity Types, zwei Enden mit Kardinalitäten. Abgebildet durch Navigation Properties *(Association Sets)* und Foreign Keys.

	Entity Type	***Storage Entity***
Ausprägungen	Klasse	Table, View, Stored Procedures, Graph, Collection
Inhalte	Properties, Entity Keys, Alternate Keys, Relationships/Associations, Foreign Keys	Columns, Primary Key, Unique Key Constraints, Foreign Keys

Das Entity Framework bietet drei Ansätze, um mit dem Mapping zu starten:
- ***Database First:*** Der Datensatz besteht schon, aus dieser Struktur wird passender Code generiert
- ***Model First:*** Es wird zuerst ein Model erstellt, aus welchem DB und Code generiert werden. Veraltet.
- ***Code First:*** Aus bestehenden Codestrukturen wird die Datenbankstruktur generiert

10.2. OR-MODEL

Beim ***Kreieren des Models*** für das Mapping gibt es ***keine Vorgaben*** bezüglich Basisklassen, zu implementierenden Interfaces, Konstruktoren etc. Auf die Werte wird via Properties zugegriffen.

Es gibt drei verschiedene Ansätze für das Mapping. Es muss jeweils nur einer verwendet worden, in den Beispielen werden jedoch alle drei aufgezeigt.
- ***Mapping By Convention*** *(automatisches Mapping ohne explizite Konfiguration)*
- ***Mapping By Attributes/Data Annotations*** *(Durch Attribute direkt an Klassen/Properties)*,
- ***Mapping By Fluent API*** *(Model Builder mit LINQ-Extension-Method ähnlichen Queries und überschreiben von `OnModelCreating()`)*

Folgende Elemente werden gemapped:
- Entity Type ⇔ Storage Type
- Property ⇔ Column
- Entity Key ⇔ Primary Key
- Relationship ⇔ Foreign Key

10.2.1. Include/Exclude von Entities

(1) Convention

Alle Klassen werden gemapped, wenn ein `DbSet`-Property im Context vorhanden ist. Indirekt werden Klassen auch via Navigation Properties gemapped *(Relationships, hier `Products` und `Metadata`)*.

(2) Fluent API

In `OnModelCreating()` durch `Entity<T>()` bzw. `Ignore<T>()`.

(3) Data Annotations

Es werden alle Entities gemapped, ausschliessen durch `[NotMapped]` Annotation an Klasse.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; } // (1)
  protected override void OnModelCreating(
     ModelBuilder modelBuilder) {
    modelBuilder.Entity<AuditEntry>(); // (2)
    modelBuilder.Ignore<Metadata>(); // (2)
  }
}
public class Category {
  public int Id { get; set; }
  public string Name { get; set; }
  public ICollection<Product> Products { get; set; } //(1)
  public ICollection<Metadata> Metadata { get; set; } //(1)
}
public class Product { ... }
public class AuditEntry { ... }
[NotMapped] // (3)
public class Metadata { ... }
```

10.2.2. Include/Exclude von Properties

(1) Convention
Alle `public` Properties mit Getter und Setter werden gemapped.

(2) Fluent API
In `OnModelCreating()` durch `Entity<T>().Property()` bzw. `Entity<T>().Ignore()`.

(3) Data Annotations
Es werden alle Properties gemapped, durch `[NotMapped]` Annotation an Property können bestimmte Properties ausgeschlossen werden.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Category>()
     .Property(b ⇒ b.Name); // (2)
    modelBuilder.Ignore<Metadata>()
     .Ignore(b ⇒ b.LoadedFromDatabase); // (2)
  }
}
public class Category {
  public int Id { get; set; } // (1)
  public string Name { get; set; } // (1)
  [NotMapped] //(3)
  public DateTime LoadedFromDatabase { get; set; }
}
```

10.2.3. Keys

Für Primary Keys siehe «Primary Keys, Default Schema & Computed Columns» (Seite 43)

(1) Convention
Property mit Namen «[Entity]Id»
(z.B. `Category.Id` oder `Category.CategoryId`)

(2) Fluent API
In `OnModelCreating()` durch `Entity<T>().HasKey()`. Einzige Möglichkeit für Composite Primary Keys.

(3) Data Annotations
`[Key]` Annotation an Property.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Category>()
     .HasKey(e ⇒ e.Id); // (2)
} }
public class Category {
  [Key] // (3)
  public int Id { get; set; } // (1)
  public string Name { get; set; }
  [NotMapped] // (3)
  public DateTime LoadedFromDatabase { get; set; }
}
```

10.2.4. Required, Optional, Nullability

(1) Convention
Nullable Types sind `NOT NULL`, non-nullable `NULL`. Ausnahme: Wenn nullable Reference Types deaktiviert sind, sind sämtliche Reference Types `NULL`.

(2) Fluent API
In `OnModelCreating()` durch `Entity<T>().Property().IsRequired([true|false])`.

(3) Data Annotations
`[Required]` oder `[Required(false)]` Annotation an Property.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Category>()
     .Property(e ⇒ e.Name)
     .IsRequired(); // (2)
    modelBuilder.Entity<Category>()
     .Property(e ⇒ e.Description)
     .IsRequired(false); // (2)
} }
public class Category {
  public int Id { get; set; }
  [Required] // (3)
  public string Name { get; set; }
  [Required(false)] // (3)
  public string? Description { get; set; } }
```

10.2.5. Maximum Length

(1) Convention
Keine Restriktion, bzw. nur durch DB selbst: `NVARCHAR(MAX)`. Bei Keys Limit von 450 Zeichen.

(2) Fluent API
In `OnModelCreating()` durch `Entity<T>().Property()` `.HasMaxLength(int)`.

(3) Data Annotations
`[MaxLength(int)]` Annotation an Property.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Category>()
     .Property(e ⇒ e.Name)
     .HasMaxLength(500); // (2)
  } }
public class Category {
  public int Id { get; set; }
  [MaxLength(500)] // (3)
  public string Name { get; set; }
}
```

10.2.6. Unicode

(1) Convention
Strings sind immer Unicode durch `NVARCHAR`.

(2) Fluent API
In `OnModelCreating()`
durch `Entity<T>().Property()`
`.IsUnicode([true|false])`.

(3) Data Annotations
`[Unicode]` oder `[Unicode(false)]` Annotation an Property.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Category>()
     .Property(e ⇒ e.Name)
     .IsUnicode(false); //(2)
  }
}
public class Category {
  public int Id { get; set; }
  [Unicode(false)] //(3)
  public string Name { get; set; }
}
```

10.2.7. Precision

(1) Convention
Pro Datentyp im Provider festgelegt.
(Precision: Anzahl Digits total, Scale: Anzahl Nachkommastellen)

(2) Fluent API
In `OnModelCreating()` durch
`Entity<T>().Property()`
`.HasPrecision(precision, scale)`
oder `.HasPrecision(precision)`.

(3) Data Annotations
`[Precision(precision, scale)]` oder
`[Precision(precision)]` Annotation an Property.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Category>()
     .Property(e ⇒ e.Price)
     .HasPrecision(10, 2); // HasPrecision(10); (2)
  }
}
public class Category {
  public int Id { get; set; }
  [Precision(10, 2)] // [Precision(10)] (3)
  public decimal Price { get; set; }
}
```

10.2.8. Indexes

Es gibt drei verschiedene Indextypen:
- ***Non-unique Index*** *(z.B. Kategorien)*
- ***Unique Index*** *(z.B. Keys)*
- ***Multi-Column-Index*** *(besteht aus mehreren Spalten)*

(1) Convention
Werden bei Foreign Keys automatisch erstellt.

(2) Fluent API
In `OnModelCreating()` durch
`Entity<T>().HasIndex()` *(Non-unique)*
`.HasIndex().IsUnique()` *(Unique)*
`.HasIndex(x ⇒ new { ... })` *(Multi-Column)*

(3) Data Annotations
`[Precision(name)]` *(Non-Unique)*
`[Precision(name, IsUnique = true)]` *(Unique)*
`[Precision(name1, name2,...)]` *(Multi-Column)*

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Category>()
     .HasIndex(b ⇒ b.Name); // (2)
    modelBuilder.Entity<Category>()
     .HasIndex(b ⇒ b.Name)
     .IsUnique(); // (2)
    modelBuilder.Entity<Category>()
     .HasIndex(b ⇒ new { b.Name, b.IsActive }); // (2)
  }
}
[Index(nameof(Name))] // (3)
[Index(nameof(Name), IsUnique = true)] // (3)
[Index(nameof(Name), nameof(IsActive))] // (3)
public class Category {
  public int Id { get; set; }
  public string Name { get; set; }
  public bool? IsActive { get; set; }
}
```

10.2.9. Entity Type Configuration

Die Nachteile der Fluent API sind, dass sie viel Text beinhaltet und unstrukturiert ist. Mit der Entity Type Configuration kann die Konfiguration im `OnModelCreating()` in eigene Klassen und damit Dateien ausgelagert werden.

Dazu muss eine Klasse von `IEntityTypeConfiguration<T>` ableiten und `Configure()` überschreiben. Die Syntax ist mit `OnModelCreating()` identisch.

```
internal class CategoryTypeConfig
  : IEntityTypeConfiguration<Category> {

  public void Configure(
    EntityTypeBuilder<Category> builder)
  {
    builder.Property(p ⇒ p.Timestamp)
      .IsRowVersion();
} }
```

Anschliessend muss die Konfiguration in der entsprechenden `OnModelCreating()` registriert werden. Alternativ auch mit `[EntityTypeConfiguration]` Annotation.

```
public class ShopContext : DbContext {
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.ApplyConfiguration(
      new CategoryTypeConfig()
    ); } }
```

10.3. RELATIONALE DATENBANKEN

Bisher waren alle Mappings ***unabhängig vom Provider***. Die nachfolgenden Beispiele beziehen sich auf ***Microsoft SQL Server***. Im Model Builder bzw. der Fluent API gibt es zusätzliche Extension Methods nur für relationale Provider.

10.3.1. Tabellen

(1) Convention
Tabellenname = `dbo.DbSet-Name`
(z.B. `dbo.Categories`)

(2) Fluent API
In `OnModelCreating()` durch `Entity<T>()`
`.ToTable(table, schema: schema)`
Tabellename zwingend, Schema optional.

(3) Data Annotations
`[Table(name, Schema = schema)]` Annotation an Klasse.
Tabellenname zwingend, Schema optional.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; } // (1)
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Category>()
     .ToTable("Category", schema: "dbo"); // (2)
  }
}

[Table("Category", Schema = "dbo")] // (3)
public class Category {
  public int Id { get; set; }
  public string Name { get; set; }
}
```

10.3.2. Spalten

(1) Convention
Spaltenname = Property-Name *(z.B. `Name`)*
Die `Order` *(Reihenfolge der Spalte in DB)* ist nach der Reihenfolge der Properties im Code gegeben.

(2) Fluent API
In `OnModelCreating()`
durch `Entity<T>().Property()`
`.HasColumnName(name, order: order).`

(3) Data Annotations
`[Column(name, Order = order)]`
Annotation an Property.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Category>()
     .Property(e ⇒ e.Name)
     .HasColumnName("CategoryName", order: 1); // (2)
  }
}

public class Category {
  public int Id { get; set; }
  [Column("CategoryName", Order = 1)] // (3)
  public string Name { get; set; }
}
```

10.3.3. Datentypen & Default Values

(1) Convention
Je nach Provider unterschiedlich.
Keine Default Values.

(2) Fluent API
In `OnModelCreating()`
durch `Entity<T>().Property().HasColumnName()`
`.HasColumnType()` *(Datentypname der Ziel-DB)*
`.HasDefaultValue()` *(Wert / Gültige SQL Expression)*.

(3) Data Annotations
`[Column(name, TypeName = type)]` Annotation an Property.
Default Values nicht unterstützt.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Category>()
     .Property(e ⇒ e.Name)
     .HasColumnName("CategoryName")
     .HasColumnType("NVARCHAR(500)")
     .HasDefaultValue("---"); // (2)
  }
}

public class Category {
  public int Id { get; set; }
  [Column("CategoryName", TypeName = "NVARCHAR(500)")]//(3)
  public string Name { get; set; }
}
```

10.3.4. Primary Keys, Default Schema & Computed Columns

Primary Keys

Convention

`PK_<Klassenname>`

Fluent API

```
modelBuilder.Entity<Category>()
  .HasKey(e ⇒ e.Id)
  .HasName("PrimaryKey_Category");
```

Default Schema

Convention

Microsoft SQL Server verwendet «dbo», SQLite kennt keine Schemas

Fluent API

```
modelBuilder.HasDefaultSchema("sales");
```

Computed Columns

Fluent API

```
modelBuilder.Entity<Category>()
  .Property(e ⇒ e.DisplayName)
  .HasComputedColumnSql(
    "[Id] + ' ' + [Name]");
```

10.4. RELATIONSHIPS

Bildet Beziehungen zwischen ***zwei Entitäten*** ab. Sie werden im Modell durch ***Navigation Properties*** und ***Foreign Keys*** dargestellt. In den Beispielen wird eine ***Produkt-zu-Kategorie-Beziehung*** gezeigt.

- ***Collection Navigation Property***: Enthält alle Elemente der N-Beziehung in einer `ICollection`. Damit kann vom 1-Ende zum N-Ende navigiert werden *(Kategorie → Produkt)*.
- ***Reference Navigation Property***: Die Referenz zum 1-Ende. Hat denselben Typ wie das 1-Ende. Damit kann vom N-Ende zum 1-Ende navigiert werden *(Produkt → Kategorie)*.
- ***Foreign Key Property***: Foreign Key auf dem N-Ende. Wird als eigene Property implementiert.

10.4.1. one-to-many / Fully Defined

Eine one-to-many Relationship ist eine ***1-zu-N-Beziehung***. Im Entity Framework gibt es mehrere «Ausbaustufen», je nachdem welche Art von Zugriff von beiden Enden gewünscht ist.

In einer ***Fully Defined one-to-many Beziehung*** sind alle Beziehungs-Elemente vorhanden.

(1) Convention

✓ ***Collection Navigation Property*** *(das 1-Ende)*
✓ ***Reference Navigation Property*** *(das N-Ende)*
✓ ***Foreign Key Property***

(2) Fluent API

In `OnModelCreating()` durch `Entity<T>().Property()` entweder `.HasOne().WithMany()` *(von 1 zu N)* oder `.HasMany().WithOne()` *(von N zu 1)* *(je nachdem welches Ende T ist)* `.HasForeignKey()` `.HasConstraintName("FK_<TableName>_<KeyName>")`

(3) Data Annotations

`[ForeignKey(name)]` Annotation an Navigation Property. Das Property, welches die ID enthält muss davor deklariert werden.

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Product>() // 1-Ende
     .HasOne(p ⇒ p.Category)
     .WithMany(b ⇒ b.Products) // (2)
     .HasForeignKey(p ⇒ p.CategoryId)
     .HasConstraintName("FK_Product_CategoryId");
  }
}
public class Category {
  public int Id { get; set; }
  public ICollection<Product> Products { get; set; }
}
public class Product {
  public int Id { get; set; }
  public int CategoryId { get; set; }
  [ForeignKey(nameof(CategoryId))] // (3)
  public Category Category { get; set; }
}
```

10.4.2. one-to-many / Shadow Foreign Key

In einer ***Shadow Foreign Key one-to-many Beziehung*** wird der Foreign Key weggelassen.

(1) Convention

✓ ***Collection Navigation Property*** *(das 1-Ende)*
✓ ***Reference Navigation Property*** *(das N-Ende)*
× ***Foreign Key Property***

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Product>() // 1-Ende
     .HasOne(p ⇒ p.Category)
     .WithMany(b ⇒ b.Products) // (2)
     //.HasForeignKey(p ⇒ p.CategoryId)
     .HasConstraintName("FK_Product_CategoryId");
  } }
```

(2) Fluent API

In `OnModelCreating()` durch `Entity<T>().Property()` entweder `.HasOne().WithMany()` *(von 1 zu N)* oder `.HasMany().WithOne()` *(von N zu 1)*
(je nachdem welches Ende T ist)
`.HasConstraintName("FK_<TableName>_<KeyName>")`
(ohne Foreign Key)

(3) Data Annotations

Die Foreign-Key-Annotation wird weggelassen

```
public class Category {
  public int Id { get; set; }
  public ICollection<Product> Products { get; set; }
}
public class Product {
  public int Id { get; set; }
  //public int CategoryId { get; set; }
  //[ForeignKey(nameof(CategoryId))] (3)
  public Category Category { get; set; }
}
```

10.4.3. one-to-many / Single Navigation Property

In einer ***Single Navigation Property one-to-many Beziehung*** ist es nicht mehr möglich, vom N-Ende zum 1-Ende zu kommen, da das Reference Navigation Property wegfällt.

(1) Convention

✓ ***Collection Navigation Property*** *(das 1-Ende)*
× ***Reference Navigation Property*** *(das N-Ende)*
× ***Foreign Key Property***

(2) Fluent API

`HasOne()` bzw. `WithOne()` enthält nicht mehr ein Lambda zum 1-Ende, sondern nur noch den Typ dessen als `T`.

(3) Data Annotations

Das Navigation Property wird ebenfalls weggelassen

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Product>() // 1-Ende
     .HasOne<Category>() // Nur noch Verweis auf Typ
     .WithMany(b ⇒ b.Products) // (2)
     //.HasForeignKey(p ⇒ p.CategoryId)
     .HasConstraintName("FK_Product_CategoryId");
  }
}
public class Category {
  public int Id { get; set; }
  public ICollection<Product> Products { get; set; }
}
public class Product {
  public int Id { get; set; }
  //public int CategoryId { get; set; }
  //[ForeignKey(nameof(CategoryId))] (3)
  //public Category Category { get; set; }
}
```

10.4.4. one-to-many / Foreign Key only

In einer ***Foreign Key only one-to-many Beziehung*** ist nur noch der Foreign Key auf dem N-Ende vorhanden.

(1) Convention

× ***Collection Navigation Property*** *(das 1-Ende)*
× ***Reference Navigation Property*** *(das N-Ende)*
✓ ***Foreign Key Property***

(2) Fluent API

`HasOne()` bzw. `WithOne()` enthält nicht mehr ein Lambda zum 1-Ende, sondern nur noch den Typ dessen als `T`.
`WithMany()` bzw. `HasMany()` hat ebenfalls kein Lambda mehr, ist aber nicht generisch.

(3) Data Annotations

Keine

```
public class ShopContext : DbContext {
  public DBSet<Category> Categories { get; set; }
  protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
    modelBuilder.Entity<Product>() // 1-Ende
     .HasOne<Category>()  // Nur noch Verweis auf Typ
     .WithMany()          // Kein Lambda mehr (2)
     .HasForeignKey(p ⇒ p.CategoryId)
     .HasConstraintName("FK_Product_CategoryId");
  }
}
public class Category {
  public int Id { get; set; }
  //public ICollection<Product> Products { get; set; }
}
public class Product {
  public int Id { get; set; }
  public int CategoryId { get; set; }
  //[ForeignKey(nameof(CategoryId))] (3)
  //public Category Category { get; set; }
}
```

10.5. DATABASE CONTEXT

Der Database Context ist der ***wichtigste Teil des Entity Frameworks***. Zur ***Design-Time*** definiert er das Model/OR Mapping, die Konfiguration und stellt ***Database Migrations*** bereit. Während der Runtime verwaltet er den Connection Pool, führt CRUD Operationen aus, stellt Change Tracking bereit, cachet Daten und verwaltet Transaktionen.

`DbContext`-Instanzen sollten nicht zu lange leben. Es gibt eine ***limitierte Anzahl Connections*** im Client Connection Pool und das ***Change Tracking*** wird über die Zeit ***ineffizient***. Auch sollten `DbContext`-Instanzen nicht geteilt werden, da sie nicht Thread-safe sind und eine Exception die Instanz unbrauchbar machen kann. ***Fehleranfälligkeit***, weil Objekte nur an einem Kontext attached.

Empfehlungen: `DbContext` in einem `using`-Statement verwenden, Web-Applikationen sollten eine Instanz pro Request instanzieren, GUI-Applikationen eine pro Formular. Generell ausgedrückt: Eine Instanz pro «Unit of Work»

```
await using ShopContext context = new();
```

10.5.1. LINQ to Entities

Das ***Entity Framework*** führt selbst keine Queries aus, es ***generiert*** nur Queries, welche die DB dann ausführt. Der SQL-Output ist je nach Formulierung der LINQ-Queries unterschiedlich. LINQ-to-Entities-Queries werden zu ***Expression Trees*** kompiliert, welche dann zur Laufzeit ***geparst*** werden und ***SQL-Statements*** generieren. Dies impliziert aber, dass nicht alle .NET Expressions auf DB-Syntax übersetzt werden können – der Provider muss Funktionen bereits kennen, um deren SQL generieren zu können.

```
await context.Categories.SingleAsync(c ⇒ Name.ToLower() == "Büsis"); // Funktioniert
// Funktion dem Provider unbekannt, funktioniert nicht
await context.Categories.SingleAsync(c ⇒ MyHelper.DoSomeShitWithThis(c.Name) == "Büsis");
```

10.5.2. Ablauf

1. `DbContext` instanzieren
   (DB-Verbindung öffnen oder offene Verbindung aus Connection Pool holen, Cache & Change Tracker initialisieren)
2. Abfrage mit LINQ
3. Daten ändern/speichern
   (Change Tracker erkennt Änderung & bereitet SQL-Statement vor)
4. Deferred Abfrage mit LINQ
   (In Abfragesprache übersetzen & Abfrage ausführen)
5. `DbContext` schliessen
   (Cache invalidieren & DB-Verbindung zurück in Connection Pool)

```
await using ShopContext context = new(); // 1.
Category category = await context        // 2.
  .Categories
  .SingleAsync(c ⇒ c.Id == 1);
category.Name = $"{category.Name} changed";

await context.SaveChangesAsync();        // 3.
var categories = context.Categories;     // 4.
foreach (Category c in categories)
  { Console.WriteLine(c.Name); }
// 5. End of Method
```

10.5.3. CUD-Operationen

Der ***`DbContext`*** agiert nach dem ***«Unit of Work»-Pattern***: Beim Laden eines Objektes aus der DB wird ein neuer UoW registriert. Die Änderungen werden ***aufgezeichnet*** und beim Speichern werden alle in einer ***einzigen Transaktionen geschrieben***.

Entities haben einen von fünf States:

- ***Added:*** Das Entity wird vom `DbContext` getracked, existiert aber in der DB noch nicht
- ***Unchanged:*** Entity wird getracked, existiert in der DB, und ihre Properties haben sich gegenüber DB nicht verändert.
- ***Modified***: Entity wird getracked, existiert in der DB, und mindestens ein Property-Wert wurde verändert.
- ***Deleted:*** Entity wird getracked, existiert in der DB, wurde zum Löschen markiert. Wird deshalb gelöscht, wenn `SaveChanges()` zum nächsten Mal ausgeführt wird
- ***Detatched:*** Entity wird nicht vom `DbContext` getracked

Insert

Es gibt 3 verschiedene Möglichkeiten, ein Objekt der DB hinzuzufügen. Davon muss nur eine verwendet werden.

```
await using ShopContext context = new();
Category cat = new() { Name = "Notebook" };

// (1) Nur Kategorie-Objekt ohne Referenzen inserten
context.Entry(cat).State = EntityState.Added;
// Alle Objektreferenzen von Kategorie auch inserten
context.Add(cat); // (2) generisch
context.Categories.Add(cat); // (3) nur für Category

// SQL generieren & ausführen
await context.SaveChangesAsync();
// Neu generierten Primary Key erhalten
int catId = cat.Id;
```

Delete

```
await using ShopContext context = new();
Category cat = await context
  .Categories
  .FirstAsync(c ⇒ c.Name == "Kinderartikel");

// Nur Kategorie-Objekt ohne Referenzen löschen
context.Entry(cat).State = EntityState.Deleted;
// Alle Objektreferenzen von Kategorie auch löschen
context.Remove(cat); // generisch
context.Categories.Remove(cat); // nur für Category

// SQL generieren & ausführen
await context.SaveChangesAsync();
```

Update

```
await using ShopContext context = new();

// Gewünschte Kategorie laden
Category cat await context
  .Categories
  .FirstAsync(c => c.Name == "Tastaturen");

// Änderungen an Objekten durchführen
cat.Name = "Mechanische Tastaturen";

// SQL generieren & ausführen
await context.SaveChangesAsync();
```

Mehrere Operationen (Unit of Work)

Durch UoW können mehrere Operationen vor `SaveChanges` durchgeführt werden.

```
await using ShopContext context = new();
Category c1 = new() { Name = "Notebooks" };
context.Add(c1);
Category c1 = new() { Name = "Ultrawides" };
context.Add(c2);
Category c3 = await context
  .Categories
  .SingleAsync(c ⇒ c.Id == 42);
c3.Name = "Gaming Chairs";
await context.SaveChangesAsync();
```

Batch-Operationen

Delete

```
await using ShopContext context = new();
// Alle Spalten in Categories löschen
await context
  .Categories
  .ExecuteDeleteAsync();
// Löschen mit Filter
await context
  .Categories
  .Where(c ⇒ c.Name == "Billigbier")
  .ExecuteDeleteAsync();
```

Update

```
await using ShopContext context = new();
Category cat await context
  .Categories
  // Filter, wenn gewünscht
  .Where(c => c.Name == "Hundespielzeug")
  .ExecuteUpdateAsync(c ⇒
    c.SetProperty(
      p ⇒ p.Name,
      "Büsi-Spielzeug"
    ) );
```

10.5.4. CUD von Assoziationen

Beziehungen können auf mehrere Arten angepasst werden:

- ***Anpassen des Navigation Properties:*** `order.Customer = customer;`
- ***Hinzufügen/Entfernen in Collection Navigation Properties:*** `customerOrders.Add(order);` bzw. `.Remove(order);`
- ***Setzen des Foreign Keys:*** `order.CustomerId = 1;`

Gesamten Objekt-Graph inserten

```
await using ShopContext context = new();
Customer c = new() {
  Name = "Jannis"
  Orders = new List<Order> {
    new() { ... }
    new() { ... }
  } };
context.Add(c);
await context.SaveChangesAsync();
```

Beziehung für bestehendes Objekt hinzufügen

```
await using ShopContext context = new();
Customer c = await context
  .Customers
  .Include(c ⇒ c.Orders) // Bestehende laden
  .FirstAsync();


c.Orders.Add(new Order());
await context.SaveChangesAsync();
```

Beziehung via Navigation Property ändern

Indirekte Foreign Key-Änderung

```
await using ShopContext context = new();
Order order = await context
  .Orders
  .FirstAsync();
order.Customer = await context
  .Customers
  .FirstAsync(c ⇒ c.Name == "Jannis");
await context.SaveChangesAsync();
```

Beziehung via Foreign Key ändern

Direkte Foreign Key-Änderung

```
await using ShopContext context = new();
Order order = await context
  .Orders
  .FirstAsync();


order.CustomerId = 2;
await context.SaveChangesAsync();
```

10.6. CHANGE TRACKING

Der Change Tracker ***registriert alle Änderungen an getrackten Entities.*** Er aktualisiert den Entity State und agiert komplett ***ohne*** die DB zu überprüfen. Der Change Tracker hat für das ***Setzen*** von jedem State eine ***Methode***. Es gibt immer mindestens 3 Varianten mit dem gleichen Effekt.

- ***`context.Add(), .Remove()` etc.:*** Berücksichtigen den ganzen Objektgraphen, können sämtliche Entities entgegennehmen.
- ***`context.[...].Add(), .Remove()` etc.:*** Berücksichtigen den ganzen Objektgraphen, können nur diese bestimmten Entities entgegennehmen.
- ***`Entry(...).State`:*** Berücksichtigt nur dieses Entity, ohne Referenzen

10.6.1. State Entries

Können über `DbContext.Entry<T>(object)` abgerufen werden.

Inhalt

- Informationen über ***Status*** des Objekts
- Status jedes Properties *(Geändert?, originaler Wert, aktueller Wert)*
- Entity-spezifische ***Ladefunktionen***
- Funktion, um Werte neu von DB zu laden
- Funktion, um referenzierte Entities zu laden *(Lazy Loading)*

```
Category cat = await context
  .Categories
  .FirstAsync();
cat.Name = "Nirvana"; // Entity ändern
EntityEntry<Category> entry = context
  .Entry(cat);
Console.WriteLine(entry.State); // Output: Modified
PropertyEntry<Category, string> en = entry
  .Property(c ⇒ c.Name);
Console.WriteLine(en.IsModified); // true
Console.WriteLine(en.OriginalValue); // Nickelback
Console.WriteLine(en.CurrentValue);  // Nirvana
```

Beispiel State Transition

```
// New Record
Category cat = new() { Name = "Glüehwii" };          // EntityState.Detached (wie untracked file in Git)
context.Add(cat);                                    // EntityState.Added
await context.SaveChangesAsync();                    // EntityState.Unchanged
cat.Name = "Drü Manne Kebab";                        // EntityState.Modified
await context.SaveChangesAsync();                    // EntityState.Unchanged
context.Remove(cat);                                 // EntityState.Deleted
await context.SaveChangesAsync();                    // EntityState.Unchanged (Objekt bleibt im Speicher)
// Load from Database (tracked)
Category catLoaded1 = await context                  // EntityState.Unchanged
  .Categories
  .FirstAsync(c ⇒ Name == "Schoggigipfel");
// Load from Database (untracked)
Category catLoaded2 = await context                  // EntityState.Detached
  .Categories
  .AsNoTracking() // Deaktiviert Change Tracking
  .FirstAsync(c ⇒ Name == "El Töni Mate" );
```

10.7. LADEN VON OBJECT GRAPHS

Object Graphs sind ein Set von ***eigenständigen***, aber ***verbundenen Objekten***, die zusammen eine ***logische Einheit*** bilden. Diese können auf verschiedene Arten geladen werden.

- ***Eager Loading:*** Assoziationen werden nicht geladen, können aber mit `Include()` einzeln hinzugefügt werden. Geschieht in der SQL-Abfrage via Join. Ist der Default.
- ***Explicit Loading:*** Assoziationen werden explizit nachgeladen, Collections werden komplett geladen. Geschieht in separater SQL-Abfrage
- ***Lazy Loading:*** Assoziationen werden bei Property-Zugriff automatisch nachgeladen. Collections werden komplett geladen, aber Filtern ist möglich. Geschieht in separater SQL-Abfrage.

Kein Laden von Referenzen

```
Order order = await context
  .Orders
  .FirstAsync(); // DB-Access

var customer = order.Customer;
// customer is null
var items = order.Items
// items is null
```

Eager Loading

```
Order order = await context
  .Orders
  .Include(o ⇒ o.Customer)
  .Include(o ⇒ o.Items)
    .ThenInclude(oi ⇒ oi.Product)
  .FirstAsync(); // DB-Access

var customer = order.Customer;
// customer is not null
var items = order.Items
// items is not null
```

Explicit Loading

```
Order order = await context
  .Orders
  .FirstAsync(); // DB-Access
await context
  .Entry(order)
  .Reference(o ⇒ o.Customer)
  .LoadAsync(); // DB-Access
await context
  .Entry(order)
  .Collection(o ⇒ o.Items)
  .Query().Where(oi ⇒ Ordered > 1)
  .LoadAsync(); // DB-Access
var customer = order.Customer;
// customer is not null
var items = order.Items
// items is not null
```

Datenbankzugriffe: Immer wenn etwas geholt, gespeichert oder über Daten aus der DB iteriert wird. Bei Lazy Loading zusätzlich, wenn auf Referenzen / andere Entities zugegriffen wird.

10.7.1. Lazy Loading

Modell-Klassen verwenden meist ***Auto-Properties*** *({ get; set; })*. Wie kann beim Zugriff ***zusätzliche Ladelogik*** ausgeführt werden?

1. ***Manuell:*** Auf Auto-Properties verzichten und Logik selbst implementieren
2. ***Proxies:*** Durch Dynamic Binding mit `virtual` auf Navigation Properties.

EF generiert selbst abgeleitete Proxy-Klassen, welche ca. die Logik von Variante 1 beinhalten. Wird die ***DB-Verbindung*** bei Lazy-Loading ***unterbrochen***, kann sie ***nicht*** wiederhergestellt werden.

```cs
public class Order {
  public int Id { get; set; }
  public Customer Customer { get; set; } }
// Variante 1
public class Order {
  public int Id { get; set; }
  public Customer Customer { ... } }
// Variante 2
public class Order {
  public int Id { get; set; }
  public virtual Customer Customer { get; set; } }
public class OrderProxy : Order {
  public override Customer Customer { ... } }
```

10.8. OPTIMISTIC CONCURRENCY

Annahme: Zwischen Laden und Speichern eines Records wird dieser nicht verändert. Erst beim Speichern wird überprüft, ob mittlerweile veränderte Werte auf der DB liegen. Für die Konflikterkennung gibt es ***zwei Varianten***.

Timestamp	***Concurrency Tokens*** (Daten-Versionen)
Pro Record wird ein Timestamp bzw. eine «Row Version» gespeichert. Dieser ist Teil des Datenobjekts. Sie werden bei einer Änderung am Property automatisch aktualisiert.	Beim Ändern der Daten verbleiben die Originaldaten im Objekt. Diese werden beim Speichern mitgegeben und überprüft, ob die Originalwerte dem aktuellen DB-Zustand entsprechen.
`DB-Timestamp == Objekt-Timestamp?` – ***Ja:*** Speichern & Timestamp erhöhen – ***Nein:*** Versionskonflikt	`DB-Daten == Originale Objektdaten?` – ***Ja:*** Speichern – ***Nein:*** Versionskonflikt
Fluent API In `OnModelCreating()` auf `modelBuilder.Entity<T>()` `.Property(p ⇒ p.property).IsRowVersion();` **Data Annotations** `[Timestamp]` Annotation an Property	**Fluent API** In `OnModelCreating()` auf `modelBuilder.Entity<T>()` `.Property(p ⇒ p.property).IsConcurrencyToken();` **Data Annotations** `[ConcurrencyCheck]` Annotation an Property

10.8.1. Konfliktbehebung

Die ***`DbUpdateConcurrencyException`*** beinhaltet fehlerhafte Entries *(Aktuelle Werte, Originale Werte, Werte von DB)*.

Varianten zur Behandlung

1. Fehler ignorieren
2. Benutzer fragen
3. Autokorrektur
   1. Exception fangen
   2. Fehlerhafte Werte analysieren
   3. Objekt korrigieren
   4. Concurrency Tokens / Timestamps aktualisieren
   5. Speichern

```cs
// Code aus Platzgründen nicht asynchron
using ShopContext context1 = new();
using ShopContext context2 = new();
// Client 1
Category p1 = context1.Categories.First();
p1.Name = "Nina";
// Client 2
Category p2 = context2.Categories.First();
p2.Name = "Jannis";

context1.SaveChanges(); // OK
context2.SaveChanges(); // Fails
// DbUpdateConcurrencyException
```

10.9. DATABASE MIGRATION

Sollen Spalten oder Tabellen ***angepasst*** werden, kann es zu verschiedenen Problemen kommen, vor allem wenn ***vorhandene Daten abgeändert*** werden müssen. Mit den Database Migrations von EF werden diese vereinfacht.

Während der Entwicklung

1. Modell anpassen *(Klassen, Properties erstellen/ändern/löschen)*
2. Migration erstellen *(Als C#-Klasse, logischer Code für Up-/Down-Migration)*
3. Migration reviewen und evtl. korrigieren
4. Code der Migration kann zu Git hinzugefügt werden

Deployment

1. Änderungen gemäss Migrations-Reihenfolge auf DB deployen
2. Rollback auf älteren Stand via Down-Migration möglich

Jede Migration wird anhand ihres Namens und ihres Erstellungs-Timestamps identifiziert.

Migrationsdateien im C# Projekt

- ***<timestamp>_<MigrationName>.cs***
  Die eigentliche Migration
- ***<timestamp>_<MigrationName>.Designer.cs***
  Metadaten für Entity Framework
- ***<DbContextClassName>ModelSnapshot.cs***
  Snapshot, welcher als Basis für nächste Migration gilt.

Migrationstabellen auf der DB

dbo._EFMigrationsHistory
Liste aller auf die DB angewendeten Migrationen

Workflow einer Migration

1. ***Erste Migration erstellen:*** `dotnet ef migrations add InitialMigration` *(erstellt Migration mit ihren Dateien in Projekt)*
2. ***Datenbank updaten:*** `dotnet ef database update` *(erstellt `_EFMigrationsHistory` mit `InitialMigration` und führt diese aus)*
3. Weitere Migration erstellen, usw.

10.9.1. Migrations-API

Migrations können innerhalb des Codes automatisiert werden.

- ***Delete / Create:***
  - `EnsureDeleted()` löscht DB
  - `EnsureCreated()` erstellt DB, wenn nicht vorhanden
- ***Migrate:*** DB auf aktuellste Migration migrieren
- Abfrage vorhandener Migrations *(Alle, Pending, Applied)*
- Explizite Migration auf spezifische Version

```
await using ShopContext context = new();
DatabaseFacade database = context.Database;
await database.EnsureDeletedAsync();
await database.EnsureCreatedAsync();
await database.MigrateAsync();
IEnumerable<string> list = database.GetMigrations();
list = await database.GetPendingMigrationsAsync();
list = await database.GetAppliedMigrationsAsync();
IMigrator m = context.GetService<Migrator>();
await m.MigrateAsync("MigrationName");
```

11. GRPC - GOOGLE REMOTE PROCEDURE CALL

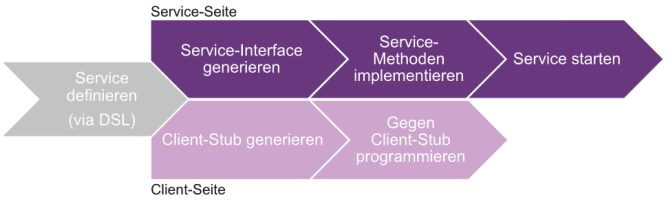
gRPC ist die neue ***Standard-Technologie*** für ***Backend-Kommunikation*** in .NET. Ist eine Erweiterung von RPC (Remote Procedure Call). Erlaubt ***Client / Server Kommunikation***, wird aber primär für ***Server-to-Server*** Kommunikation eingesetzt. ***Hohe Performance*** von zentraler Bedeutung, ***nicht als Frontend-API*** gedacht. Löst Probleme wie Security, Synchronisierung, Data Flow Handling etc.

Grundprinzipien

Einfache Service-Definition, Sprach-unabhängigkeit, Problemlose Skalierbarkeit, Bi-direktionales Streaming, Integrierte Authentisierungsmechanismen.

Verwendet ***HTTP/2*** als Kommunikationsprotokoll *(Unterstützt Multiplexing und bidirektionales Streaming, HTTPS und Header Compression, weniger Overhead weil ACK nicht mehr pro Request sondern einmalig für alle Ressourcen)* und ***Google Protocol Buffers*** *(Protobuf)* als Interface Definition Language *(IDL)*.

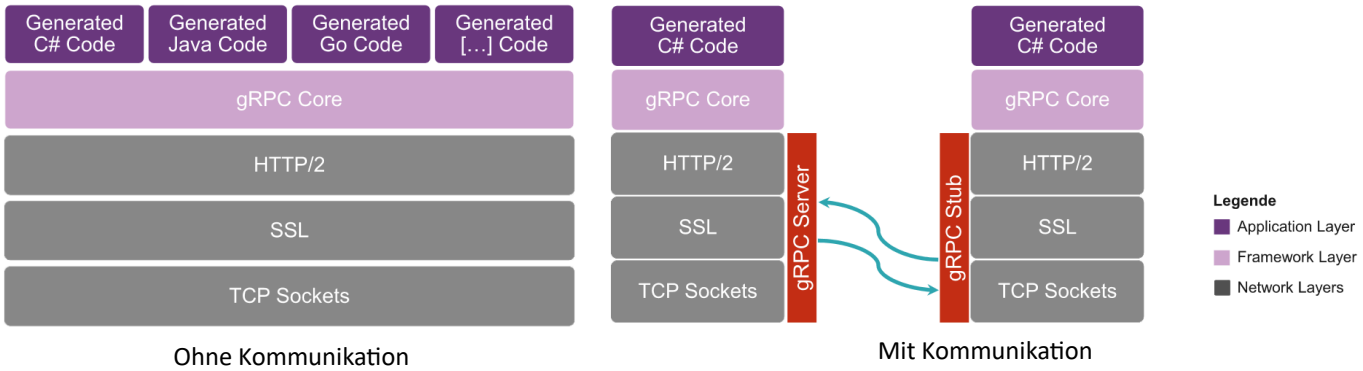
Developer Workflow



11.1. ARCHITEKTUR

gRPC ist ein ***Software Development Kit***. Es ist plattformneutral und eine Visual Studio (Code) Integration existiert.

11.1.1. Aufbau



11.1.2. Beispiel: Service

Interface Definition Language (Protobuf)

```
syntax = "proto3";
option csharp_namespace = "BasicExample";
package Greet;

// Service Definition
service Greeter {
  // Specify Request-Message
  rpc SayHello(HelloRequest)
    // Service Method with Response Message
    returns (HelloReply);
}
// Request message containing the user's name
message HelloRequest {
  // Message Type, "name" is only for humans
  string name = 1;
}
// Response message containing the greetings
message HelloReply {
  // 1 = Unique field ID, Order of encoding
  string message = 1;
}
```

Implementation

Ein gRPC-Service returnt immer einen ***Task.FromResult()*** mit dem entsprechenden Message Type-Objekt.

```
public class GreeterService : Greeter.GreeterBase {
// GreeterBase = Generierte Basisklasse

  // T ist Name einer Message
  public override Task<HelloReply> SayHello(
    HelloRequest request,
    ServerCallContext context
  ) ⇒
    Task.FromResult(
      new HelloReply {
        Message = "Hello " + request.Name
      }
    );
}
```

11.1.3. Beispiel: Client

```
// The Port number (5001) must match the port of the gRPC server.
GrpcChannel channel = GrpcChannel.ForAddress("https://localhost:5001");
Greeter.GreeterClient client = new(channel); // Generated Client Stub

// Remote Procedure Call
try {
  HelloReply reply = await client.SayHelloAsync(new HelloRequest { Name = "GreeterClient" });
  Console.WriteLine($"Greeting: {reply.Message}");
} catch ( ... ) { ... }
```

11.1.4. Vergleich gRPC / REST

Feature	*gRPC*	*REST*
Contract	Required	Optional (OpenAPI)
Transport	HTTP/2	HTTP/1.1
Payload	Protobuf *(small / binary)*	JSON *(larger / human readable)*
Formalisierung	Strikte Spezifikation	Keine oder OpenAPI Specification
Streaming	Client / Server / Bidirektional	Client / Server
Browser Support	Nein *(benötigt grpc-web)*	Ja
Security	Transport / HTTPS *(zwingend)*	Transport / HTTPS *(optional)*
Client Generierung	Ja	Third-party Tooling

11.2. PROTOCOL BUFFERS

11.2.1. Umfang

- ***Interface Definition Language (IDL):*** Eine Subform der Domain Specific Language (DSL). Beschreibt ein Service Interface plattform- und sprach-neutral.
- ***Data-Model:*** Beschreibt Messages *(respektive Request- und Response-Objekte)*
- ***Wire Format:*** Beschreibt das Binärformat zur Übertragung
- ***Serialisierungs- / Deserialisierungs-Mechanismen***
- ***Service-Versionierung***

11.2.2. Proto Files

Der Code im Kapitel «Interface Definition Language (Protobuf)» (Seite 50) ist in einem ***Proto-File*** *(Datei-Endung *.proto)* gespeichert. Ein Proto-File besteht aus folgenden ***Abschnitten:***

- ***Header:*** Allgemeine Definitionen *(Syntax, option, etc)*
- ***Services:*** 0 oder mehr Services, 1 oder mehr Service-Methoden pro Service
- ***Message Types:*** 0 oder mehr Fields, Field definiert sich aus Type, Unique Name und Unique Field Number.

Service-Methoden haben immer genau 1 Parameter und 1 Rückgabewert. ***Null-Werte*** können mit `import "google/protobuf/empty.proto"; ... google.protobuf.Empty` verwendet werden.

11.2.3. Messages / Fields

- Angabe des ***Feldtypen*** *(Skalarer Werttyp, anderer Message Type oder Enumeration)*,
- ***Unique Field Name*** *(Wird für Generatoren verwendet, in Lower Snake Case)*
- ***Unique Field Number*** *(Identifikation für das Binärformat)*.

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

11.2.4. Fields / Repeated Fields

Es gibt zwei Arten von Fields, `singular` und `repeated`. `singular` ist ein skalarer Wert und der Default. `repeated` ist eine Liste von Werten als Strings.

```
message SearchResponse {
  repeated string results = 1;
}
```

11.2.5. Enumerations

Analog zu `enum` in .NET, ***Definition*** innerhalb einer Message oder im Proto-File Root. Enum-Member mit dem ***Wert 0 muss zwingend existieren*** *(Default-Wert)*. Schlüsselwort ***`reserved`*** kann auch für Enumerations verwendet werden.

```
message SearchRequest {
  Color searchColor = 1;
  enum Color {
    RED = 0; // 0 must exist
    GREEN = 1; } }
```

11.2.6. Message Type Composition & Imports

Message Types können ebenfalls als Field verwendet werden. ***Import*** eines *.proto Files über das `import`-Schlüsselwort. Damit können z.B. Enum-Definitionen ausgelagert werden.

```
// File: example.proto
import "protos/_base.proto";
message Search {
  LogicalOperator operator = 2;
}
// File: _base.proto
enum LogicalOperator{ ... }
```

11.2.7. Reserved Fields

Für ***Versionierung*** gedacht. Wiederverwendung wird vom Protocol Buffer Compiler verhindert. Schlüsselwort `reserved` ***verfügbar für Unique Field Name und Unique Field Number*** *(Falls in einer neuen Version ein Feld entfernt wird, wird zukünftige Überschreibung so verhindert und somit Rückwärtskompabilität garantiert)*.

Ranges können mit `"to"` reserviert werden:
`reserved 1 to 3`

```
message SearchRequest {
  reserved 1, 3, 20 to 30;
  reserved "page_number", "result_per_page";

  string query = 1; // Compilerfehler
  int32 page_number = 2; // Compilerfehler
  int32 result_per_page = 3; // Compilerfehler
}
```

11.3. GRPC C# API

11.3.1. Protocol Buffers Compiler

Der Protobuf Compiler wird automatisch in die Build-Pipeline vor dem C# Compiler eingebunden. In Visual Studio kann angegeben werden ob Client & Server, nur eines von beiden oder gar nichts generiert werden soll.

Verantwortlichkeiten: Parsen / Validieren von Proto-Files, Auflösen von Includes, Generieren von C# Source Files.

11.3.2. Aufbau Client / Server

Server-Projekt	***Client-Projekt***
ASP.NET Core Projekt	Beliebiges Projekt
Proto-File als Kopie / Link einbinden	Proto-File als Kopie / Link einbinden
NuGet Packages: Grpc.AspNetCore	**NuGet Packages:** Grpc.Net.Client, Google.Protobuf, Grpc.Tools

Server-Projekt

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <RootNamespace>BasicExample</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <None Remove="Protos\greet.proto" />
  </ItemGroup>

  <ItemGroup>
    <Protobuf Include="Protos\greet.proto"
              GrpcServices="Server">
      <Link>Protos\greet.proto</Link>
    </Protobuf>
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="…" />
  </ItemGroup>
</Project>
```

Protobuf Include

Zu generierende Klassen (Server)

File Link

Packages

Client-Projekt

```
<Project Sdk="Microsoft.NET.Sdk">
   <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <RootNamespace>BasicExampleClient</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <Protobuf Include="..\<relative_path>\greet.proto"
              GrpcServices="Client">
      <Link>Protos\greet.proto</Link>
    </Protobuf>
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Google.Protobuf" Version="…" />
    <PackageReference Include="Grpc.Net.Client" Version="…" />
    <PackageReference Include="Grpc.Tools" Version="…">[...]
    </PackageReference>
  </ItemGroup>
</Project>
```

Protobuf Include (Relativer Pfad)

Zu generierende Klassen (Client)

Packages

11.3.3. Generierter Code

- ***Namespace:*** Proto-Code «`option csharp_namespace = "ProtoExample"`» wird zu C# «`namespace ProtoExample`»
- ***Abstrakte Basisklasse:*** Wird pro Service erzeugt.
  «`service MyService { /* ... */ }`» wird zu «`public static partial class MyService { /* ... */ }`»

Services müssen beim Startup ***registriert*** sein, Generierte Methoden aus abstrakter Basisklasse müssen ***implementiert*** werden, ansonsten gibt es in beiden Fällen Exceptions vom Protobuf-Compiler.

11.3.4. Startup C# API

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
// Add services to the container (Registration of the gRPC Types via Dependency Injection)
builder.Services.AddGrpc();

WebApplication app = builder.Build();

// Configure the HTTP request pipeline (Definition of the endpoints, once per service)
app.MapGrpcService<GreeterService>();
app.Run();
```

11.3.5. Beispiel Customer Service

Architektur-Übersicht

CustomerService

- RPC `GetCustomers`
  - Input: Keiner
  - Output: Kunden-Liste ohne Bestellungen
- RPC `GetCustomer`
  - Input: ID und `IncludeOrders` bool
  - Output: Kunde mit/ohne Bestellungen

OrderService

- RPC `GetOrders`
  - Input: Customer ID
  - Output: Liste der Bestellungen des Kunden

11.4. STREAMS

Streams unterstützen ***drei Modi:***

- Server Streaming Call *(Server → Client)*,
- Client Streaming Call *(Client → Server)*
- Bi-direktional.

Garantiert sowohl ***Auslieferung*** als auch ***Reihenfolge*** der Auslieferung.
Anwendungen: Messaging, Games, Live-Resultate, Smart Home Devices etc.

- ***Synchrones Lesen:*** Resultat wird vom Server erst returnt, wenn alles gelesen wurde.
- ***Asynchrones Lesen:*** Server returnt jede gelesene Zeile direkt.

11.4.1. Protocol Buffers

Schlüsselwort "stream" vor Typbezeichnung. Payload ist eine normale Message.

- ***ReadFiles:*** Server Streaming Call *(stream bei returns)*
- ***SendFiles:*** Client Streaming Call *(stream bei Parameter)*
- ***RoundtripFiles:*** Bi-Direktionaler Streaming Call *(stream bei beiden)*

```
service FileStreamingService {
  rpc ReadFiles (google.protobuf.Empty)
    returns (stream FileDto);
  rpc SendFiles (stream FileDto)
    returns (google.protobuf.Empty);
  rpc RoundtripFiles (stream FileDto)
    returns (stream FileDto);
}
message FileDto {
  string file_name = 1;
  int32 line = 2;
  string content = 3;
}
```

11.4.2. Beispiel: File Streaming Service

Client (`Program.cs`)

```
using AsyncClientStreamingCall<FileDto, Empty> call = client.SendFiles();

// The port number(5001) must match the port of the gRPC server.
GrpcChannel channel = GrpcChannel.ForAddress("https://localhost:5001");
StreamingService.StreamingClient client = new(channel); // Client that every function uses

// Run all streaming functions
await TestServerStreaming(client); await TestClientStreaming(client); await TestBiDirectionalStreaming(client);
WriteLine("Press any key to exit..."); ReadKey();

// Stream files from server
static async Task TestServerStreaming(StreamingService.StreamingClient client) {
    WriteLine(nameof(TestServerStreaming));
    using AsyncServerStreamingCall<FileDto> call = client.ReadFiles(new()); // Call object without Parameter
    await foreach (FileDto message in call.ResponseStream.ReadAllAsync()) { // Read last written chunk
        WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Content: {message.Content}");
    }
}

// Stream files to server
static async Task TestClientStreaming(StreamingService.StreamingClient client) {
  WriteLine(nameof(TestClientStreaming));
  string[] files = Directory.GetFiles(@"Files"); // Get all files in folder
  foreach (string file in files) { // Open every file
      string content; int line = 0; using StreamReader reader = File.OpenText(file);
      while ((content = await reader.ReadLineAsync()) != null) {
          line++; // Read every line
          FileDto reply = new() { FileName = file, Line = line, Content = content, }; // Write into Protobuf
          await call.RequestStream.WriteAsync(reply); // Write protobuf to stream
      }
  }
  // Closing the stream is required
  await call.RequestStream.CompleteAsync(); // No more messages to come (server exits foreach-Loop)
  Empty result = await call; // Wait until service method is terminated / Get the result
}

// Send and receive files at the same time
static async Task TestBiDirectionalStreaming(StreamingService.StreamingClient client) {
  WriteLine(nameof(TestBiDirectionalStreaming));
  using AsyncDuplexStreamingCall<FileDto, FileDto> call = client.RoundtripFiles();
  // Read
  Task readTask = Task.Run(async () => {
      await foreach (FileDto message in call.ResponseStream.ReadAllAsync()) {
          WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Content: {message.Content}");
      }
  });
  // Write
  string[] files = Directory.GetFiles(@"Files");
  foreach (string file in files) {
      string content; int line = 0; using StreamReader reader = File.OpenText(file);
      while ((content = await reader.ReadLineAsync()) != null) {
          line++;
          FileDto reply = new() { FileName = file, Line = line, Content = content, };
          await call.RequestStream.WriteAsync(reply);
      }
  }
  // Required
  await call.RequestStream.CompleteAsync(); // No more messages to come (server exits foreach-Loop)
  await readTask; // Wait until service method is terminated / all messages are received by client
}
```

Server

```
public class StreamingService : FileStreamingService.FileStreamingServiceBase {
    // Read files from disk and send to the client
    public override async Task ReadFiles(
      Empty request, // No parameters
      IServerStreamWriter<FileDto> responseStream,
      ServerCallContext context)
    {
      Empty request, // No parameters
      IServerStreamWriter<FileDto> responseStream,
      ServerCallContext context)
    {
        WriteLine(nameof(ReadFiles));
        string[] files = Directory.GetFiles(@"..\11_FileStreamingFiles");

        foreach (string file in files) {
            string content; int line = 0;
            using StreamReader reader = File.OpenText(file);

            // Read until End of File
            while ((content = await reader.ReadLineAsync()) != null) {
                line++;
                FileDto reply = new() { FileName = file, Line = line, Content = content, };
                await responseStream.WriteAsync(reply);
            }
        }
    }

    // Recieve files that the client sends
    public override async Task<Empty> SendFiles(
      IAsyncStreamReader<FileDto> requestStream,
      ServerCallContext context)
    {
        WriteLine(nameof(SendFiles));
        await foreach (FileDto message in requestStream.ReadAllAsync()) { // Read last written chunk
            WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Content: {message.Content}");
        }
        return new Empty(); // Empty result, nothing to return
    }

    // Send files back to the client
    public override async Task RoundtripFiles(
      IAsyncStreamReader<FileDto> requestStream,
      IServerStreamWriter<FileDto> responseStream,
      ServerCallContext context)
    {
        WriteLine(nameof(RoundtripFiles));
        await foreach (FileDto message in requestStream.ReadAllAsync()) {
            await responseStream.WriteAsync(message);
            WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Content: {message.Content}");
        }
    }
}
```

Proto

```
syntax = "proto3";
import "google/protobuf/empty.proto";
option csharp_namespace = "FileStreaming";
package FileStreaming;

service StreamingService {
  rpc ReadFiles (google.protobuf.Empty) returns (stream FileDto);
  rpc SendFiles (stream FileDto) returns (google.protobuf.Empty);
  rpc RoundtripFiles (stream FileDto) returns (stream FileDto);
}

message FileDto {
  string file_name = 1;
  int32 line = 2;
  string content = 3;
}
```

11.5. EXCEPTION HANDLING

Grundsätzlich immer via ***RpcException***. Basierned auf StatusCodes, Details über «Trailers» möglich.

Status Codes: `OK`, `Cancelled`, `Unknown`, `InvalidArgument`, `DeadlineExceeded`, `NotFound`, `AlreadyExists`, `PermissionDenied`, `Unauthenticated`, `ResourceExhausted`, `FailedPrecondition`, `Aborted`, `OutOfRange`, `Unimplemented`, `Internal`, `Unavailable`, `DataLoss`.

Unbehandelte Exceptions

Exception wird auf Server nicht sauber behandelt. Server Runtime fängt Exception, wirft `RpcException` mit Status Code «`Unknown`».

Behandelte Exceptions mit Trailer

Exception wird auf Server sauber behandelt und korrekt verpackt.

Für die Trailers wird die `Metadata`-Klasse verwendet. Die Informationen werden als Key-Value-Pair-Liste gespeichert.

```
public override Task<Empty> Unhandled(
Empty request, ServerCallContext context) => throw
new Exception("Unhandled Exception");

// behandelt
public override Task<Empty> NotFound(
Empty request, ServerCallContext context) => throw
new RpcException(new Status(
  StatusCode.NotFound, "not found"), new Metadata {
    { "error-details", "The server dislikes you" } }
);
```

11.5.1. Client Side Exception Handling

Analog zur Serverseite: `RpcException`. Kann optional mit `when`-Klausel auf verschiedene Catch-Blöcke verteilt werden.

RpcException umfassen auch alle Fehlersituationen bezüglich ***Kommunikation*** *(Endpoint antwortet nicht, Verbindung bricht ab, etc.)*.

Weitere Exceptions *(other stuff)* sind rein ***client-seitige Probleme*** *(Channel-Variable ist null, etc)*.

Häufige Fehler sind `Unimplemented` *(Service nicht in Startup registriert oder Service-Methode nicht überschrieben)* oder `Unknown` *(Fehlender Catch-Block für aufgetretenen Fehler)*

```
try { /* gRPC calls */ }
catch (RpcException e)
  when (e.StatusCode == StatusCode.Unavailable) {}
catch (RpcException e)
  when (e.StatusCode == StatusCode.NotFound) {}
catch (RpcException e)
  when (e.StatusCode == StatusCode.Aborted) {}
catch (RpcException e) {}
catch (Exception) {} // Other stuff
```

11.6. SPECIAL TYPES

11.6.1. Well Known Types

– ***Empty:*** Platzhalter für Null-Werte

```
import "google/protobuf/empty.proto";
service EmptyService { rpc Dummy (google.protobuf.Empty) returns (google.protobuf.Empty); }
```

```
Empty e = new();
```

– ***Timestamp:*** UTC Zeitstempel

```
import "google/protobuf/timestamp.proto";
message TimestampResponse { google.protobuf.Timestamp results = 1; }
```

```
Timestamp ts = new() { Seconds = DateTime.UtcNow.Second };
```

– ***Bytes / ByteString:*** Binärer Datentyp

```
message BinaryResponse { bytes results = 1; }
```

```
ByteString bs = ByteString.Empty; bs = ByteString.CopyFromUtf8("X");
```

– ***Oneof:*** Lässt eine Auswahl von Typen zu. Beim Auslesen wird nur der zuletzt geschriebene Wert angezeigt, alle anderen sind `default`. Intern als Klasse und Enum *(das aktuelle Property)* implementiert.

```
message OneofResponse { oneof results { string image_url = 1; bytes image_data = 2; } }
```

```
OneofResponse response = new() { ImageUrl = "https://..." }
var rc = response.ResultsCase; // Momentan aktiv: ImageUrl
string s = response.ImageUrl; // https://...
ByteString bs = response.ImageData; // default

response.ImageData = ByteString.CopyFromUtf8("PicOfUrMom"); // neue Response setzen
rc = response.ResultsCase; // Momentan aktiv: ImageData
s = response.ImageUrl; // default
bs = response.ImageData; // PicOfUrMom
```

– ***Any:*** Repräsentiert einen beliebigen Wert.

```
import "google/protobuf/any.proto";
message AnyResponse { google.protobuf.Any results = 1; }
```

```
AnyResponse response = new();
response.Results = Any.Pack(new CustomerResponse()); // Create Pack
bool isCust = response.Results.Is(CustomerResponse.Descriptor); // Type check
bool success = response.Results.TryUnpack(out CustomerResponse parsed); // Safe unpack
```

11.6.2. Collections

– ***Repeated Fields:*** Generiert ein Repeated Field Property. Ist read-only. Etwa equivalent zu `IList`.

```
message RepeatedResponse { repeated string results = 1;}
```

```
RepeatedResponse response = new();
response.Results.Add("Hello"); string[] arr = {"A", "B"}; response.Results.AddRange(arr);
```

– ***Map Fields:*** Generiert ein Map Field Property. Ist read-only. Etwa equivalent zu `IDictionary`.

```
message MapResponse { map<int32, string> results = 1; }
```

```
MapResponse response = new(); response.Results.Add(1, "Hello");
bool exists = response.Results.ContainsKey(1); string result = response.Results[1];
```

11.7. KONFIGURATION & LOGGING

11.7.1. Server-side

Option	*Default*	*Beschreibung*
`MaxSendMessageSize`	`null`	Maximale Message-Grösse beim Senden *(Exception wenn grösser)*
`MaxReceiveMessageSize`	4 MB	Maximale Message-Grösse beim Empfangen *(Exception wenn grösser)*
`EnableDetailedErrors`	`false`	Wenn `true` werden Exception-Details an Client übermittelt *(Mögliches Sicherheitsrisko!)*
`CompressionProviders`	gzip	Kompressions-Algorithmus für Messages

Konfiguration

```
.Services
  .AddGrpc(options => // Globale Konfiguration
  {
    options.MaxSendMessageSize = 5 * 1024 * 1024; // 5 MB
    options.MaxReceiveMessageSize = 2 * 1024 * 1024; // 2 MB
    options.EnableDetailedErrors = true;
    options.CompressionProviders = new List<ICompressionProviders>
    {
      new GzipCompressionProvider(CompressionLevel.Optimal)
    }
  })
  .AddServiceOptions<AdvancedService>(options => // Spezifischer Service
  {
    options.MaxSendMessageSize = 1 * 1024 * 1024; // 1 MB
  })
```

Logging aktivieren in `Program.cs`

```
builder.Logging.AddFilter(
  "Grpc", LogLevel.Debug
);
```

…oder in `appsettings.json`

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning",
      "Grpc": "Debug"
    }
  },
  "AllowedHosts": "*",
}
```

11.7.2. Client-side

Option	*Default*	*Beschreibung*
`HttpClient`	Objekt	HttpClient für gRPC-Verbindung. Kann verändert werden um z.B. weitere HTTP Header zu senden
`DisposeHttpClient`	`false`	Wenn `true` und Verwendung eines eigenen `HttpClient` wird der Client mit dem gRPC-Channel disposed
`LoggerFactory`	`null`	Logging-Schnittstelle für gRPC Calls
`Credentials`	`null`	Authentifizierungs-Credentials am Server

```
GrpcChannel channel = GrpcChannel.ForAddress(
  "https://localhost:5001",
  new GrpcChannelOptions { MaxSendMessageSize = 2 * 1024 * 1024  }
);
```

11.8. DEADLINES & CANCELLATION

Anstatt Timeouts gibt es in gRPC ***Deadlines***. Dabei wird ein Timestamp mitgegeben und bei jeder Stelle überprüft, ob dieser bereits überschritten wurde.

Sie können direkt beim Aufruf mitgegeben werden, entweder über `CallOptions` oder Direkt. Die Deadline muss UTC sein. Bei Überschreiten wird eine `RpcException` ausgelöst. Es kann server-seitig über `ServerCallContext` darauf zugegriffen werden.

Cancellation wird ähnlich gehandhabt: Mit `tokenSource.CancelAfter(1111)` das Timeout setzen, dann als Parameter mitgeben.

```
// Variante 1: Als callOption
CallOptions callOptions = new(
  deadline: DateTime.UtcNow.AddMilliseconds(200));
await client.DummyAsync(new Empty(), callOptions);
// Variante 2: Direkt als Parameter
await client.DummyAsync(new Empty(),
  deadline: DateTime.UtcNow.AddMilliseconds(200));
// Zugriff auf Deadline server-seitig
public override Task<Empty> Dummy(
  Empty request, ServerCallContext context) {
    Console.WriteLine(context.Deadline.ToString("O")
  );
}
```

Deadlines	***Cancellation***
Erwartet, evtl. planbar, quantifizierbar, Schutz vor zu langen Wartezeiten	Unvorhersehbar, Zufällig, Durch unberechenbaren Interrupt ausgelöst *(Abbruch durch User, nicht mehr an Resultat interessiert)*

12. REFLECTION & ATTRIBUTES

12.1. REFLECTION

Reflections ermöglichen es, ***Informationen über geladene Assemblies*** und die darin definierten Typen zur erhalten. Auch können mit Reflections ***Typinstanzen zur Laufzeit*** erstellt werden.

Anwendung:

- Type Discovery *(Suchen und Instanzieren von Typen, Zugriff auf dynamische Datenstrukturen – z.B. JavaScript-Objekte)*
- Late Binding von Methoden/Properties *(Aufruf von Methoden/Properties nach der Type Discovery)*
- Reflection Code-Emittierung *(Erstellen von Typen & Members zur Laufzeit)*

12.1.1. Klasse `System.Type`

Alle Typen in der Common Language Runtime CLR sind ***selbst-definierend***. Die Klasse `System.Type` ist der ***Einstiegspunkt*** aller Reflection-Operationen. ***Repräsentiert*** einen ***Typen*** mit all seinen Eigenschaften. Ist eine ***abstrakte Basisklasse***. Alle Objekte sind Instanzen von Typen. Bildet auch Vererbungshierarchien ab.

`typeof` / `GetType()`

Ermitteln von `System.Type`. Mit `typeof()` erhält man `System.Type` durch Angabe des Typen selbst *(`typeof(int)`, `typeof(List<string>)`)*, durch `.GetType()` den eines Objekts oder Property zur ***Laufzeit*** *(`int i; i.GetType();`)*. `System.Type` beschreibt sich selbst ebenfalls als `System.Type`-Objekt.

Member-Informationen / Typ-Hierarchie

Jeder Klassen-Member hat einen ***eigenen Reflection-Typen***. «`System.Runtime.MemberInfo`» ist abstrakte Basisklasse für Members. Bei der Suche nach Members ist es möglich zu filtern, z.B. nach der Sichtbarkeit *(z.B. `public`)* oder nach bestimmter Memberart *(z.B. Properties)*. Nicht zugreifbare Members wie private Felder sind auch einsehbar. Klassen befinden sich im Assembly «`mscorlib`» im Namespace «`System.Reflection`».

12.1.2. Beispiel Metadaten

Type-Discovery

Suche aller Typen in einem Assembly.

Ausgabe:

```
...
System.Int32
  Method Int32 CompareTo(System.Object)
  Method Int32 CompareTo(Int32)
  Method Boolean Equals(System.Object)
  Method Boolean Equals(Int32)
  Method Int32 GetHashCode()
  Method System.String ToString()
...
```

```
Assembly a01 = Assembly.Load("mscorlib");

Type[] t01 = a01.GetTypes();
foreach (Type type in t01) {
  Console.WriteLine(type);

  MemberInfo[] mInfos = type.GetMembers();
  foreach (MemberInfo mi in mInfos) {
    Console.WriteLine(
      $"\t{mi.MemberType}\t{mi}");
  }
}
```

Alle Members auslesen

Suche aller Members eines Typen.

Ausgabe:

```
...
Int32 get_CountValue() is a Method
Void set_CountValue(int32) is a Method
Void Increment() is a Method
...
----------
Int32 CountValue is a Property
...
```

```
Type type = typeof(Counter);
MemberInfo[] miAll = type.GetMembers();
foreach (MemberInfo mi in miAll) {
  Console.WriteLine($"{mi} is a {mi.MemberType}");
}
Console.WriteLine("----------");
PropertyInfo[] piAll = type.GetProperties();
foreach (PropertyInfo pi in piAll) {
  Console.WriteLine($"{pi} is a {pi.PropertyType}")
}
```

Dynamisches auslesen von Members

Suche spezieller Members eines Typen.

Ausgabe:

```
Assembly GetAssembly(Type) is a Method
Int32 GetHashCode() is a Method
Type GetType_Compat(String, String) is a Method
Assembly GetExecutingAssembly() is a Method
Assembly GetCallingAssembly() is a Method
Assembly GetEntryAssembly() is a Method
...
```

```
Type type = typeof(Assembly);
BindingFlags bf =
  BindingFlags.Public |
  BindingFlags.Static |
  BindingFlags.NonPublic |
  BindingFlags.Instance |
  BindingFlags.DeclaredOnly;
MemberInfo[] miFound = type.FindMembers(
  MemberTypes.Method, bf, Type.FilterName, "Get*"
); // Hier: Filter nach Name einer Methode
```

12.1.3. Member Information

Field Info

Beschreibt ein ***Feld auf einer Klasse*** *(Name, Typ, Sichtbarkeit, etc.)*. Erlaubt lesen / schreiben eines Feldes.

```
object GetValue(object obj);
public void SetValue(object obj, object value);
```

```
Type type = typeof(Counter); Counter c = new(1);
// All Fields
FieldInfo[] fiAll = type.GetFields(
  BindingFlags.Instance | BindingFlags.NonPublic);
// Specific Field
FieldInfo fi = type.GetField("_countValue",
BindingFlags.Instance | BindingFlags.NonPublic);
int val01 = (int) fi.GetValue(c); c.Increment();
int val02 = (int) fi.GetValue(c);
fi.SetValue(c, -999);
```

Property Info

Beschreibt ein ***Property auf einer Klasse*** *(Name, Typ, Sichtbarkeit, Informationen zu Get / Set, etc.)*. Erlaubt lesen / schreiben eines Properties.

```
object GetValue(object obj);
public void SetValue( object obj, object value);
```

```
Type type = typeof(Counter); Counter c = new(1);
// All Properties
PropertyInfo[] piAll = type.GetProperties();
// Specific Property
PropertyInfo pi = type.GetProperty("CountValue");
int val01 = (int) pi.GetValue(c); c.Increment();
int val02 = (int) pi.GetValue(c);
if (pi.CanWrite) { pi.SetValue(c, -999); }
```

Method Info

Beschreibt eine ***Methode*** auf einer Klasse *(Name, Parameter / Rückgabewert, Sichtbarkeit, etc.)*. Leitet von der ***Klasse «MethodBase»*** ab *(Basisklasse für `MethodInfo` und `ConstructorInfo`, Konstruktoren und Methoden sind aus Sicht der Metadaten recht ähnlich)*. Kann über `invoke`-Methode aufgerufen werden.

```
Type type = typeof(Counter); Counter c = new(1);
// All methodes
MethodInfo[] miAll = type.GetMethods();
// Specific Method
MethodInfo mi = type.GetMethod("Increment");
mi.Invoke(c, null);

// Mit Parametern
Type type = typeof(System.Math);
Type[] paramTypes = { typeof(int) };
MethodInfo miA = type.GetMethod("Abs", paramTypes);
object[] @params = { -1 }; // fill array with params
object returnVal = miA.Invoke(type, @params);
```

Constructor Info

Beschreibt einen ***Konstruktoren*** einer Klasse *(Name, Parameter, Sichtbarkeit, etc.)*. Leitet von der ***Klasse «MethodBase»*** ab. Kann über `invoke`-Methode aufgerufen werden.

```
Type type = typeof(Counter);
// All Constructors
ConstructorInfo[] ciAll = type.GetConstructors();
// Specific Constructor Overload 1:
```

Alternative via «Activator»:

```
Counter c03 = (Counter)Activator.CreateInstance(
  typeof(Counter), 12 /*, ... */ );
// Wenn Public Default Constructor existiert
Counter c04 = Activator.CreateInstance<Counter>();
```

```
ConstructorInfo ci01 = type.GetConstructor(
  new[] { typeof(int)} );
Counter c01 = (Counter)ci01.Invoke(
  new object[] { 12 });
// Specific Constructor Overload 2:
ConstructorInfo ci02 = type.GetConstructor(
  BindingFlags.Instance | BindingFlags.NonPublic,
  null, new Type[0], null);
Counter c02 = (Counter)ci02.Invoke(null);
```

12.2. ATTRIBUTES

Entsprechen ***Java Annotations***. Erweitern bestehende Attribute wie «`public`», «`static`», «`abstract`» oder «`sealed`». Ermöglichen ***spezifische Aspekte*** *(Erweiterbare Elemente)*.

Abfrage über Reflection möglich:
`ReflectionAttributes.Attributes.Auto`

Basisklasse: «`System.Attribute`»

```
[DataContract, Serializable]
[Obsolete] // Etc.
public class Auto {
  [DataMember]
  public string Marke { get; set; }
  [DataMember]
  public string Type { get; set; }
}
```

Anwendungsfälle: Object-relational Mapping, Serialisierung, Security und Zugriffssteuerung, Dokumentation, etc.
Arten von Atributen: ***«Intrinsic» Attribute*** *(In CLR definiert und integriert, teilweise vom Compiler berücksichtigt, wie z.B. «obsolete»)* und ***«Custom» Attribute*** *(In Framework Class Library, Selbst definierte Attribute)*

12.2.1. Syntax

Es sind ***beliebig viele*** Attribute möglich, Deklaration entweder ***separat*** *(`[DataContract][Serializable]`)* oder ***komma-separiert*** *(`[DataContract, Serializable]`)*. Je nach Implementation eines Attributes kann es ***mehrfach*** angewandt werden.

Parameter / Werte müssen vom Compiler berechenbar sein: `[Datacontract]`, `[Datacontract(Name = "AutoClass")]`, `[Obsolete("Alt!", true)]`, `[Obsolete("Alt!", IsError = true)]`

12.2.2. Custom Attribute

Im Beispiel wird ein `[BugfixAttribute]` für Dokumentation implementiert. Die Klasse hat selbst das Attribut `[AttributeUsage]`, welches bestimmt, wo ein Attribut verwendet werden kann. Durch `AllowMultiple` kann es mehrfach am selben Member angebracht werden.

```
// Deklaration
[AttributeUsage(
    AttributeTargets.Class |
    AttributeTargets.Constructor |
    AttributeTargets.Field |
    AttributeTargets.Method |
    AttributeTargets.Property,
    AllowMultiple = true)]
public class BugfixAttribute : Attribute {
  public BugfixAttribute(
    int bugId, string programmer, string date) {
    /* ... */
  }
  public int BugId { get; }
  public string Date { get; }
  public string Programmer { get; }
  public string Comment { get; set; }
}
```

```
// Verwendung
[Bugfix(121, "Nina Grässli", "16/01/2025")]
[Bugfix(107, "Jannis Tschan", "17/01/2025",
  Comment = "Some major changes!")]
public class myMath {
  [Bugfix(121, "Nina Grässli", "20/01/2025")]
  public int MyInt { get; set; }

  // Compilerfehler, weil event kein AttributeTarget
  [Bugfix(148, "Jannis Tschan", "23/01/2025")]
  public event Action CalculationDone;
  /* ... */
}
```

Abfrage via Reflection

Attribute können über Reflection abgefragt werden.
Das `ICustomAttributeProvider`-Interface stellt `IsDefined()` *(Prüft, ob ein bestimmtes Attribut vorhanden ist)* und `GetCustomAttributes()` *(Liste aller Attribute)* zur Verfügung.

Abfrage von Attribute-Information über Klasse «`Type`».
«`true`» berücksichtigt auch vererbte Attribute.

```
Type type = typeof(MyMath);
// All Class Attributes
object[] aiAll = type.GetCustomAttributes(true);
IEnumberable<Attribute> aiAllTyped =
  type.GetCustomAttributes(typeof(BugfixAttribute));
// Check Definition
bool aiDef =
  type.IsDefined(typeof(BugfixAttribute));
```