

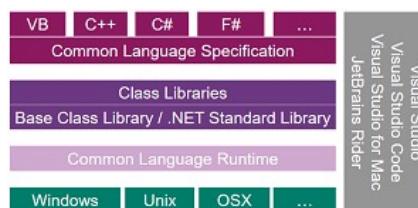
.NET Grundlagen

.NET Geschichte



- .NET Framework:** nicht mehr weitergeführt. Bis 2019 mit Version 4.8
- .NET Core:** Multiplatform Implementierung, existiert ab 2016, heisst ab 2020 nur noch .NET und löst Framework ab
- .NET Standard:** Bietet eine einheitliche Base Class Library (Standard API) für verschiedene Implementationen (Framework, Core, Xamarin, ...). Jede Implementation verwendet eine gewisse .NET Standard Version.
 - Ziel: Konsistenz und implementation "neutraler" cross-platform Libraries.
 - Kompatibilität mit jeweils verwendeter Core / Framework Version muss gegeben sein.
 - Bei Verwendung in eigenen Libraries: je tiefer die .NET Standard Version, desto einfacher einzubinden, aber desto weniger API Funktionen zur Verfügung.

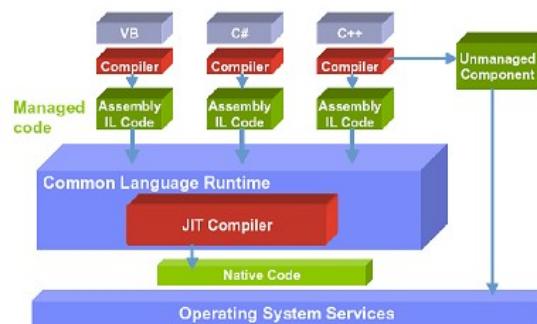
.NET Architektur



Common Language Runtime (CLR)

Laufzeitumgebung für .NET Code, analog Java Virtual Machine JVM

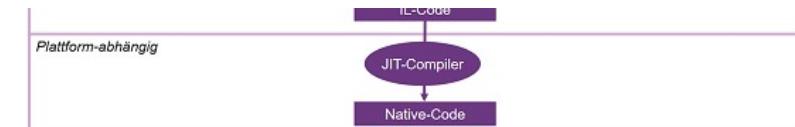
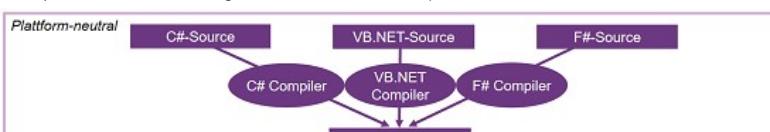
- JIT Compiler (Just-In-Time) übersetzt IL Code in Maschinencode
- Speicherverwaltung mit Garbage Collection
- Common Language Specification CLS: ermöglicht cross-language development dank MSIL, sprachübergreifendes Debugging, Type Checking und Code Verification der MSIL (Intermediate Language)
- Common Type System CTS
- Class Loader lädt Klassen-Code zur Laufzeit
- Code Access Security
- Exceptions
- Thread-Verwaltung
- COM-Interoperabilität (Interaktion mit unmanaged Code)



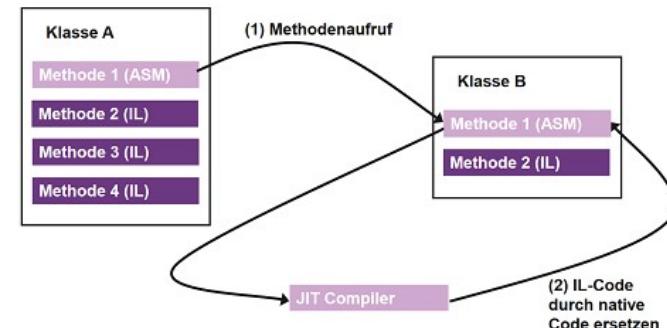
Microsoft Intermediate Language (MSIL)

MSIL ist eine vorkomplizierte Zwischensprache, und beinhaltet das Common Type System CTS. Sie ist

- Prozessor-unabhängig bzw. Plattformneutral und dadurch portabel
- Assembler-ähnlich
- Sprach-Unabhängig innerhalb der .NET Familie
- Typsicher (Type- und weitere Security-Checks beim Laden des Codes)
- Nachteil Effizienz (wird durch JIT teils wettgemacht, der zur Laufzeit spezifische Maschineninstruktionen nutzen kann)



Just In Time Compilation (JIT)



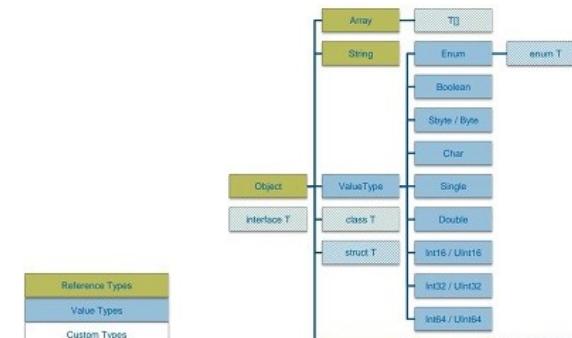
Drei Typen der Kompilierung

- Pre-JIT: Alles vor Ausführung kompiliert
- Normal-JIT: Kompilierung von Methoden bei Aufruf
- Econo-JIT: Wie Normal aber mit Cleanup-Mechanismus

Common Type System (CTS)

Das einheitliche Typensystem der .NET Umgebung ist in der CLR integriert, somit losgelöst von der einzelnen Programmiersprache. Das Typsystem ist Single-Rooted - das heisst alles ist von `System.Object` abgeleitet, auch alle Primitivtypen.

Reference Types sind alle gängigen Objekte, Klassen etc. (auch Strings), Value Types sind alle Basistypen: `sbyte, byte, short, ushort, int, uint, long, ulong, float, double, decimal, bool, char`



Polymorphismus ist somit für ausnahmslos alle Typen möglich. Methode, die `System.Object` erwartet, kann einen "normalen" Int annehmen.

Reference Types

Class erzeugt einen eigenen Reference Type. Auf dem Stack wird ein Pointer angelegt, der auf das effektive Objekt im Heap zeigt. null beschreibt immer eine Referenz ohne Adresse / Ziel.

Beinhaltet auch immer eine Referenz auf den Typen.

```
class PointRef { public int X, Y; }
PointRef a; // erzeugt im Stack eine leere Referenz
a = new PointRef(); // definiert die Referenz im Stack, erzeugt auf dem Heap ein leeres Objekt (Default-Initialisierung von Value Types)
a.X = 12;
a.Y = 24; // jeweiliger Speicherbereich im Heap wird angepasst
```

Zuweisung: Objekt-Referenz wird kopiert. Anpassung an einem Objekt verursacht auch Änderungen am anderen Objekt.

```
PointRef a = new PointRef();
a.X = 12;
a.Y = 24;
PointRef b = a;
b.X = 9;
Console.WriteLine(a.X); // Gibt 9 aus
```

Value Types

Struct erzeugt einen eigenen Value Type. Das Objekt liegt direkt auf dem Stack, **oder inline im Reference- oder Value Type**.

Konstruktor wird vom Compiler automatisch generiert, macht Default-Initialisierung (ausfüllen des Speichers). Structs können nicht abgeleitet werden (sealed).

null bei Value Types wird immer mit 0 oder false oder der Default-Wert verwendet. Effektiver null - Wert nicht möglich.

Vorteile:

- effiziente Specherausnutzung
- effizienter Zugriff
- keine Garbage Collection nötig, wird nach Ende der Methode abgeräumt

```
struct PointVal { public int X, Y; }
PointVal a; // Im Speicher noch nichts passiert
a = new PointVal(); // Speicher auf dem Stack wird alloziert und Default-Initialisiert
a.X = 12;
a.Y = 24; // Werte auf dem Stack werden angepasst
```

Zuweisung: gesamter Wert auf dem Stack wird kopiert.

Keyword	Aliased Type	Java	Wertebereich
sbyte	System.SByte	byte	-128 .. 127
byte	System.Byte		0 .. 255
short	System.Int16	short	-32'768 .. 32'767
ushort	System.UInt16		0 .. 65'535
int	System.Int32	int	-2'148'483'648 .. 2'147'483'647
uint	System.UInt32		0 .. 4'294'967'295
long	System.Int64	long	-2^63 .. 2^63-1
ulong	System.UInt64		0 .. 2^64-1
char	System.Char	float	+1.5E-45 .. +3.4E38 (32bit)
float	System.Single	double	+5E-324 .. +1.7E308 (64bit)
double	System.Double		+1E-28 .. +7.9E28 (128bit)
bool	System.Boolean	boolean	true, false
decimal	System.Decimal	char	Unicode-Zeichen

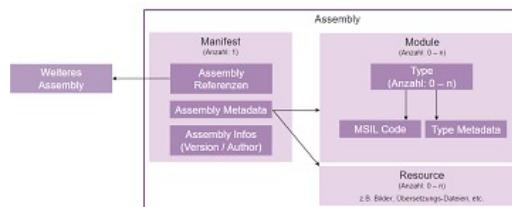
Boxing / Unboxing

Beschreibt das automatische umwandeln von Value Types in Reference Types und zurück. Immer dann, wenn ein Value Type (Struct) als Instanz einem Reference Type zugewiesen wird.

```
System.Int32 i1 = 12; // Value Type
System.Object obj = i1; // Implizites Boxing zum Reference Type
System.Int32 i2 = (System.Int32)obj; // Unboxing muss explizit gecastet werden
```

Assemblies

Ergebnis der Kompilation, eine Deployment- oder Ausführungseinheit. Vom Typ Executable (*.exe) oder Library (*.dll). Können dynamisch geladen werden, definiert Typ-Scope. Kleinste versionierbare Einheit. Entspricht einem JAR-File in Java.



Module und Metadaten

Enthält Code (MSIL) und Metadaten. Metadaten enthalten alle Aspekte des Codes ausser Programmlogik (Definitionen von Klassen, Methoden, Feldern etc.) und können mittels Reflection abgefragt werden.

- Anwendungen in der CLR: Typsicherheit, Memory Management, JIT Compilation
- Anwendungen der IDE: Object-Browser, IntelliSense
- Tools zur Analyse: IL Disassembler (IL DASM), viele Drittanbieter tools
- Erweiterbare Programmsysteme (Generische Ansätze, Dynamic Binding)

.NET Class Library

- Basisklassen mit System-Funktionen
- ADO.NET / Entity Framework für DB-Zugriff
- ASP.NET (Core) Web-Programmierung
- XML, Dateisystemzugriff
- WPF und Windows Forms (GUI)

CLI Interface

Teil des .NET Core SDK dient als Basis für high-level Tools wie IDEs. Command Struktur lautet ähnlich wie Git:

dotnet[.exe] <verb> <argument> --<option> <param> -- Argument zum Verb (eines), Parameter zur Option (0-n).

Möglichkeiten (Auszug):

new	Initialize .NET projects.
restore	Restore dependencies specified in the .NET project.
run	Compiles and immediately executes a .NET project.
build	Builds a .NET project.
publish	Publishes a .NET project for deployment (including the runtime).
test	Runs unit tests using the test runner specified in the project.
pack	Creates a NuGet package.
migrate	Migrates a project.json based project to a msbuild based project.
clean	Clean build outputs(s).
sln	Modify solution (SLN) files.
add	Add reference from the project.
remove	Remove reference from the project.
list	List references of a .NET project.
nuget	Provides additional NuGet commands.
msbuild	Runs Microsoft Build Engine (MSBuild).
vstest	Runs Microsoft Test Execution Command Line Tool.
store	Stores the specified assemblies in the runtime store.
tool	Install or work with tools that extend the .NET experience.
build-server	Interact with servers started by a build.
help	Show help.

Projekte & Referenzen

Projekt-Datei wird seit Version 1.0 als XML verwaltet (*.csproj neu seit 2017). Build Engine von Microsoft heisst MSBuild , kann direkt oder auch via .NET CLI verwendet werden.

- <Property*> : Settings
- <Item*> : zu kompiliende Items
- <Target*> : Sequenz auszuführender Schritte, TargetFramework definiert die zu kompilierende Ziel-API.

Alte Variante vor 2017 war extrem gross und sperrig. Betreffen nur .NET Framework 4.8 und älter.

Referenzen können im Projekt auf verschiedene Arten geführt werden.

- Vorkompiliertes Assembly: Muss im File System verfügbar sein
- NuGet Package: Externe Dependency (nuget.org)
- Visual Studio Projekt: in selber Solution vorhanden
- .NET Core oder Standard SDK (default, zwingend: Microsoft.NETCore.App oder NETStandard.Library)

NuGet Packages

Neuer Standard für Packagin von Applikationen. Teilt die .NET Funktionalität in verschiedene kleinere Packages auf. Erlaubt unterschiedliche Release-Zyklen, erhöht die Kompatibilität durch Kapselung von spezifischen Komponenten, verkleinert die Deployment-Einheiten.

Der Dateityp *.nupkg (zip) beinhaltet Assemblies, Manifest mit Infos zu Package Id, Titel und Beschreibung, Versions-Informationen, Dependencies etc.

Ablauf mit Kompilierung (dotnet build), Paketierung (dotnet pack), Veröffentlichung (dotnet nuget push), Konsumption (dotnet add package).

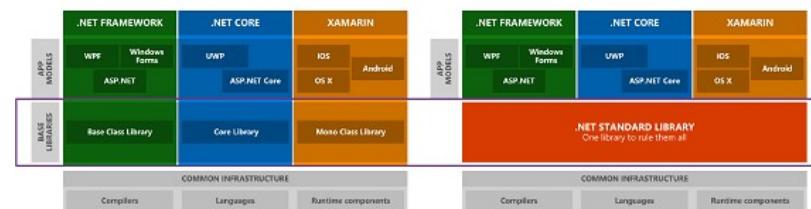
Wichtige Packages sind

- System.Runtime (Object, String, Array, Action, Func, ...)
- System.Collections (Generische Listen wie List, Dictionary< TKey, TValue >, ...)
- System.Net.Http (HttpClient, HttpResponseMessage, ...)
- System.IO.FileSystem (File, Directory, ...)
- System.Linq (Enumerable, ILookup< TKey, TElement >, ...)
- System.Reflection (Assembly, TypeInfo, MethodInfo, ...)

.NET Standard

Wurde mit .NET Core eingeführt, diente als Brücke zwischen .NET Framework und .NET Core. Motivation:

- Implementation von "neutralen" Libraries für Cross-Platform Entwicklung. Können von Core und Framework verwendet werden. Base Class Libraries, Beispiel: Zugriff Filesystem / Sockets, Serialisierung von XML, und und...
- Konsistenz zwischen verschiedenen Frameworks (Fragmentierung minimal halten)



.NET Standard listet minimal zu unterstützende APIs (Klassen und Methoden) auf.

Kompatibilität: Jede .NET Implementation unterstützt eine maximale .NET Standard Version.

Bei eigenen Libraries: je höher die Version, desto mehr APIs können verwendet werden. Je tiefer, desto einfacher einzubinden.

C# Grundlagen

Syntax, neu gegenüber Java

- Referenzparameter
- Objekte am Stack (Structs)
- Blockmatrizen
- Enumerationstypen
- Uniformes Typsystem
- goto
- Systemnahes Programmieren
- Versionierung

"Syntactic Sugar"

- Komponentenunterstützung: Properties, Events
- Delegates
- Indexers
- Operator Overloading
- foreach-Iterator
- Boxing/Unboxing
- Attributes (Annotation)
- ...

Naming Guidelines

PascalCase für so gut wie alles: Namespace, Klassen/Structs, Interface, Enum, Delegates, Methoden, Properties, Events

camelCase für Felder, lokale Variablen und Parameter

Sichtbarkeit

Attribut	Wirkung
public	überall
private	innerhalb Typ
protected	innerhalb Typ oder Ableitung davon
internal	Innerhalb Assembly
protected internal	Innerhalb Typ oder abgeleiteter Klasse oder Assembly
private protected	Innerhalb Typ oder abgeleiteter Klasse, wenn diese im gleichen Assembly ist

Standards

Typ	Standard	Zulässig (nicht nested types)	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	--	--

Namespaces

Strukturiert den Quellcode, ist hierarchisch aufgebaut und nicht an physikalische Strukturen gebunden. Mehrere Namespaces im selben File oder Files pro Namespace sind möglich, Unterscheidung von Namespace und Ordnerstruktur auch.

Beinhaltet andere Namespaces, Klassen, Interfaces, Structs, Enums, Delegates

Import eines anderen Namespaces mittels `using System;` -> macht die Verwendung vom `System.*` Präfix unnötig.

Alias Namen sind möglich: `using F = System.Windows.Forms;`

File-Scope Namespaces

Erlaubt das entfernen von Klammern, nur ein Namespace pro File erlaubt.

```
// Klassische Variante: File1.cs
namespace OstDemo
{
    class X { }
}

// File-Scope NS ab C# 10: File1.cs
namespace OstDemo;
Class X { }
    // Compiler-Fehler bei neuem Namespace im selben File
    namespace OstDemo2 { }
```

Globale Using Direktive

Meist in einem File `GlobalUsings.cs`, mit `using static Azure.Core` kann das using für das ganze Projekt definiert werden. Verkleinert

Boilerplate Code im Header. Alternative im `.csproj`:

```
<ItemGroup>
    <Using Include="Azure.Core" />
</ItemGroup>
```

Implicit Global Usings

Können im `.csproj` aktiviert werden, beinhaltet abhängig vom gewählten SDK verschiedene vordefinierte Usings.

```
<Project Sdk="Microsoft.NET.Sdk"> <!-- SDK Version -->
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net6.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings> <!-- Aktivierung -->
    </PropertyGroup>
</Project>
```

Main-Methode

Einstiegspunkt eines Programms, zwingend für Executable Types. Klassischerweise nur 1x vorhanden, falls mehrere muss in `.csproj` ein Tag `<StartupObject>` definiert werden. Programmalaufzeit ist genau so lange wie die Main-Methode.

Meist in `Program.cs` in der Klasse `Program`.

Anforderungen:

- Sichtbarkeit ist nicht relevant
- Methode Main muss static sein, Klasse aber nicht
- **Rückgabetypen**: void , int , Task und Task<int> , wobei int immer der Program Exit Code ist.
- **Parametertypen**: keine oder string[] für Command-Line Argumente

Zugriff auf Argumente

- über string[] Parameter (meist string[] args)
- über statische Methode `System.Environment.GetCommandLineArgs()`
- flexibler CmdLine Parser: NuGet Package `System.CommandLine`

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

Top-Level Statements

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

- 1x pro Assembly möglich
- Argumente heißen immer args
- Exit Codes sind erlaubt
- using werden vorher definiert
- Typen werden nachher definiert

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

Enumerationstypen

Liste vordefinierter Konstanten inklusive Wert (default Int32, leitet implizit davon ab). Werte werden von 0 weg aufsteigend vergeben oder können explizit gesetzt werden. Können mit Cast auf int verwendet werden. Duplikate sind Okay, werden beim Cast Int zu Enum Type einfach nie verwendet.

```
enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday = 15 }
Days today = Days.Sunday;
int sundayValue = (int)Days.Sunday;
```

```
// Enums können auch andere Typen implementieren, bsp. zur Speicheroptimierung.
enum Days : byte { Sunday, Monday, Tuesday, Wednesday }
```

```
// Parsing unter Verwendung der Klasse 'Enum'
Days day1 = (Days)Enum.Parse(typeof(Days), "Monday"); // non-generic
```

```
Days day2;
bool success1 = Enum.TryParse("Monday", out day2); // generic
```

```
bool success2 = Enum.TryParse("Monday", out Days day3); // generic, ab C# 7.0
```

```
// Alle Werte eines Enums ausgeben
foreach(string name in Enum.GetNames(typeof(Days)))
{
    Console.WriteLine(name);
}
```

Object

Basisklasse aller Typen, `object` ist ein Alias von `System.Object`.

Erlaubt als Parameter dynamische Methoden. Zuweisung von allen Typen kompatibel, da `System.Object` die gemeinsame Basisklasse aller Typen ist.

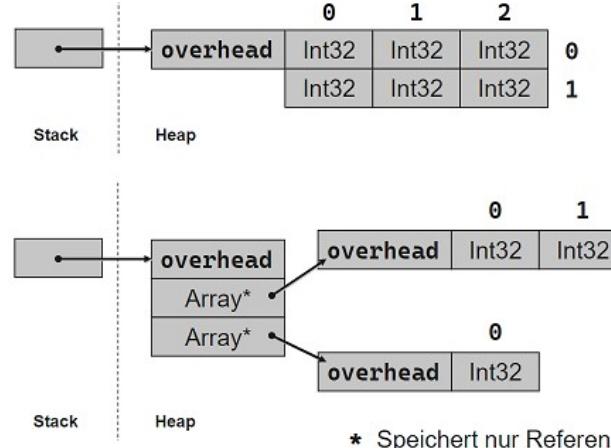
Verfügbare Methoden:

```
public Object(); // Konstruktor
public virtual bool Equals(object obj);
public static bool Equals(object objA, object objB);
public virtual int GetHashCode();
public Type GetType();
protected object MemberwiseClone();
public static bool ReferenceEquals(object objA, object objB);
public virtual string ToString();
```

Arrays

Einfachste Datenstruktur für Listen. Sind zero-Based, letztes Element ist immer `length-1`. Die Klasse `Array` ist ein Referenztyp und wird somit immer auf dem Heap angelegt. Alle Werte sind nach Instanzierung initialisiert (false, 0, null, etc.)

Mehrdimensionales Array rechteckig (Blockmatrix) vs. ausgefranzt (jagged).



```
// Eindimensional
int[] array1 = new int[5];           // Deklaration (value type)
object[] arrayA = new object[5];      // Deklaration (reference type, genau gleich)
int[] array2 = new int[] { 1, 2, 3, 4 }; // Deklaration & Definition
int[] array3 = int[] { 1, 2, 3 };      // vereinfacht ohne new, nur bei deklaration möglich
int[] array4 = { 1, 2, 3 };           // vereinfacht ohne new und ohne Typ
```

```
// Mehrdimensional (rechteckig, Blockmatrix)
int[,] multiDim1 = new int[2,3];
int[,] multiDim2 = { {1,2,3},{1,2,3} };
// Mehrdimensional (jagged)
int[][] jaggedArray = new int[ [6][];
jaggedArray[0] = new int[] { 1, 2, 3, 4 };
```

Länge wird im rechteckigen mehrdimensionalen Array ausmultipliziert, wenn `array1.Length` aufgerufen wird, d.h. die totale Anzahl Speicherplätze ausgegeben (6 im Screenshot oben). Einzelne Dimensionen müssen mit `array1.GetLength(0)` oder `array1.GetLength(1)` abgefragt werden.

Bei jagged Arrays liefert `array2.Length` effektiv die Länge der 0. Dimension (2 im Screenshot oben).

Vorteile von Blockmatrizen:

- Speicherplatz-Effizienz, da weniger Referenzen
- Schneller alloziert, da ein grosser Block verwendet werden kann
- Schnellere Garbage Collection
- Schnellerer Zugriff nicht wirklich, da allgemein optimiert wird

Strings

Datenstruktur für Zeichenketten. Referenztyp, verhält sich aber wie ein Value Type. `string` ist Alias für `System.String`. String selber ist nicht modifizierbar, da interne Optimierung geschieht. Reguläre Verkettung mit + Operator möglich, auch mit Zeilenumbruch.

Wertevergleich mit == und Equals wurde überschrieben, um den Inhalt zu vergleichen statt die Referenz. Indexierung möglich. **Nicht \0**

Terminiert. Länge mit Property `Length` abrufbar. Backslash escapen entweder mit @"C:\Temp" oder "C:\\Temp".

String Interpolation

```
x // File 1partial class MyClass{ partial void Test1Initialize(); partial void Test1Cleanup(); } // File 2partial class MyClass{ public void Test2() { } partial void Test1Initialize() { /Implementation/ } } csharp
string s1 = $"{DateTime.Now}: {"Hello"}";
string s2 = $"{DateTime.Now}: {(DateTime.Now.Hour < 18 ? "Hello" : "Good Evening")};
```

String Interning

Strings werden intern wiederverwendet, d.h. bei gleichem Text wird nur eine neue Referenz auf dasselbe Ziel erstellt.

```
string s2 = String.Copy(string s1); erstellt eine echte Kopie mit anderer Memory-Adresse.
```

Symbol

Identifiers

Alle Unicode Zeichen erlaubt. Case Sensitive. @ für Verwendung von Schlüsselwörtern als Identifier: int @while ist eine gültige Variable.

Unicode Escape Sequenzen sind erlaubt, z.B. \u03c0 anstelle vom Symbol "pi".

Schlüsselwörter

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

Kommentare

```
// Single Line
/* Multi Line */
/// Dokumentation auf Typen, Feldern, Methoden, Properties etc.
```

Primitivtypen

Ganzahlen

Bestimmung des Typen:

- 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}: 24L
- (hex) "0x" hexDigit{hexDigit}{Suffix}: 0abcdL
- (bin) "0b" [0|1]{[0|1]}{Suffix}: 0b1001101u

Fliesskommazahlen

Bestimmung des Typen

- ohne Suffix: double
- Suffix f | F: float
- Suffix d | D: double
- Suffix m | M: decimal

Lesbarkeit/Trennzeichen

Underscore erlaubt zur optischen Strukturierung, ausser am Anfang oder Ende einer Zahl. Wird vom Compiler entfernt.

```
object number = 9_876_543_210 // = 9876543210
```

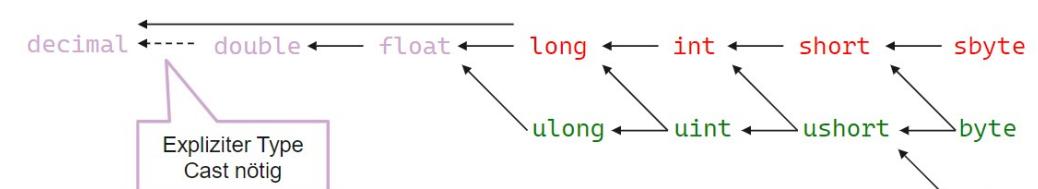
Zeichenketten

String: "test", Char: 't'

Escape Sequenzen:

- \' für '
- \" für "
- \\ für \
- \0 für \000
- \a für 0x0007 (alert)
- \b für 0x0008 (backspace)
- \f für 0x000c (form feed)
- \n für 0x000a (new line)
- \r für 0x000d (carriage return)
- \t für 0x0009 (horizontal tab)
- \v für 0x000b (vertical tab)

Typkompatibilität



\char

Statements

Leeres Statement: Semikolon ist ein Terminator, kein Separator.

; wie auch ; ; ; ist jederzeit erlaubt.

Zuweisung, Methodenaufruf, Return Statement mit oder ohne Wert,

- if
- switch

ganzzahlige, numerische Datentypen, Character und String, Enumeration.

break beendet den einzelnen case - Block (sonst fallthrough),

default wird ausgeführt wenn nichts zutrifft

- while und do-while

while (true) { }

do {} while (true)

- for und foreach

for (int i = 1; i <= 3; i++)

foreach (int x in array)

- Jumps

break - aktuellen Loop beenden

continue - zur nächsten Loop-Iteration

goto "myLabel" - Sprung zu myLabel: (nicht in einen Block, nicht aus dem finally Block eines try-Statements)

goto "case" - innerhalb Switch Statement

Klassen und Structs

Klasse

Reference Type (wird auf dem Heap abgelegt)

- Basisklassen, Abgeleitete Klassen
- Implementation von Interfaces

Konstruktor mit optionalen Parametern

Initialisierung von Feldern in der Klassendefinition erlaubt.

Instanzierung

Speicher wird

- alloziert
- initialisiert

Klassische Initialisierung: Stack s = new Stack(10);

Target-Typed "new"-Expression: Stack s = new(10);

Struct

Soll immer einen einzelnen Wert repräsentieren, speziell wenn dieser aus mehreren Teilen besteht (Vektor, Point, ...).

Ist immutable, wenn Änderungen nötig sind wird ein neuer Struct erzeugt.

- Kurzlebig oder in andere Objekte eingebettet
- wird initialisiert
- Kann nicht ableiten oder als Basisklasse verwendet werden
- Kann Interfaces implementieren

Felder

- Feld (int): initialisierung in deklaration optional, struct felder dürfen nicht initialisiert werden
- Konstante (cons int): muss initialisierungswert haben
- readonly-Feld (readonly int): in deklaration oder konstruktor initialisiert, wert darf später nicht verändert werden.

Nested Types

struct, klasse, interface, enum, delegate: Für spezifische Hilfsklassen verwendet.

Zugriff:

- Äussere Klasse sieht **public** members der inneren
- Innere Klasse sieht **alle** members der äusseren
- Fremde Klasse sieht **public** members der inneren

Statische Klassen

Können nicht instanziert werden.

Nur statische Members erlaubt

Bsp: Sammlung von Werten oder Funktionalität (Math.Pi, Math.Cos())

Statische Usings

using static System.Console um bsp. Console.WriteLine() zu WriteLine() zu kürzen.

Namenskonflikte müssen manuell aufgelöst werden.

Statische Methoden

Prozedur/Aktion (ohne Rückgabewert) Funktion (mit Rückgabewert)

Innerhalb der Klasse ohne Instanz aufrufbar (Print()) Außerhalb der Klasse eine Instanz nötig (mc.Print())

Methodenparameter

Value-Parameter

Kopie des Stack-Inhaltes wird übergeben: Wert (Struct, Int) oder Heap-Referenz (normale Klassen).

Änderungen an "nicht-Klassen" werden ausserhalb der Methode nicht persistiert!

Reference-Parameter

Adresse der Variable wird übergeben. Diese muss initialisiert sein. Nur benötigt für Value Types.

Out-Parameter

Muss initialisiert werden, kann seit C# 7.0 im Methoden-Header gemacht werden.

```
void Init(out int a, out int b) { a = 1; b = 2 } Init(out int a1, out int b1);
```

Discarding von out-Parametern möglich mit Underscore:

```
bool success = int.TryParse("123a", out _);
```

Params-Array

- Erlaubt beliebig viele Parameter
- Muss am Schluss der Deklaration stehen
- nur einmal erlaubt pro Methode
- Wird in der Methode wie ein normales Array verwendet

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

Optionale Parameter

Mittels Zuweisung eines Default Values in der Methodendefinition wird ein Parameter optional.

```
private void Sort(
    int[] array,
    int from = 0,
    int to = -1,
    bool ascending = true,
    bool ignoreCase = false
) { ... }
```

Named Parameter

Parameter können mit Namen bezeichnet unsortiert mitgegeben werden, Compiler sortiert anschliessend wieder korrekt:

```
Sort(a, ascending: false, ignoreCase: true)
```

Überladen von Methoden

Mehrere Methoden mit gleichem Namen: Unterschiede müssen vorhanden sein in einem von:

- Anzahl Parameter
- Parametertypen
- Arten (ref/out, spezifisch C#)

Rückgabewert ist zur Unterscheidung **nicht** relevant.

Properties

Reines Compiler-Konstrukt zur Vereinfachung von Get/Set Methoden. Wie Public Fields, können aber Logik beinhalten beim Lesen und Schreiben.

Standard Property:

```
private _length; // Backing-Field
public int Length
{
    get { return _length; }
    set { _length = value; }
}
```

oder als Kurzform:

```
public int LengthAuto { get; set; } // Auto-Property
```

auto-implemented Property:

```
public int LengthInit { get; set; } = 5;
```

Backing-Field wird auf 5 gesetzt.

```
public int LengthInit { get; }
```

Backing-Field wird auf readonly gesetzt.

Objekt-Initialisierung

Properties können direkt initialisiert werden. Dazu eine geschweifte Klammer nach der Klasseninstanzierung verwenden.

```
MyClass mc = new (
    Length = 1,
    Width = 2
)
```

Init-Onlysetters

Property muss während der Initialisierung gesetzt werden. Dabei ist Schreibzugriff auf eigene readonly-Felder des Typ möglich.

```
public int LengthInitOnly { get; init; }
```

```
MyClass mc = new (
    LengthInitOnly = 42
);
```

Expression Bodied Members

Bei Methoden mit genau einem Statement.

```
public class Examples {
    private int _value;

    // Constructors / Destructors (C# 7.0)
    public Examples(int v) => _value = v;
    ~Examples() => _value = 0;

    // Methods (C# 6.0)
    public int Sum(int x, int y) => x + y;
    public int GetZero() => 0;
    public void Print() => Console.WriteLine("...");

    // Properties (C# 6.0)
    public int Zero => 0;
}
```

```

public int Bla => Sum(Zero, 2);

// Getters/Setters (C# 7.0)
public int Value {
    get => _value;
    set => _value = value;
}

```

Indexers

Überladen des Index-Operators, um Klassen mit Array-Verhalten zu erzeugen.

- Read- und/oder Write-Only möglich, weglassen von `get{} / set{}`
- Schlüsselwort `this` definiert den Indexer
- Schlüsselwort `value` für Zugriff Wert im Setter
- Indexwert kann überladen werden (mehrere Typen)

```

class MyClass {
    private string[] _arr = new string[10];

    public string this[int index] {
        get { return _arr[index]; }
        set { _arr[index] = value; }
    }

    public int this[string index] {
        get
        {
            for(int i = 0; i < _arr.Length; i++)
            {
                if(_arr[i] == search) return i;
            }
            return -1;
        }
    }
}

```

Konstruktoren

Gleicher Name wie Klasse, kein Rückgabetyp nötig.

Privater Konstruktor: Kann nur intern verwendet werden, damit wird kein Default-Konstruktor erzeugt.

Aufruf anderer Konstruktoren derselben Klasse mit `this`. Aufruf des Basisklassen-Konstruktors mit `base`.

Default-Konstruktor

Konstruktor ohne Parameter. Kann bei Klasse überschrieben werden, wird ansonsten vom Compiler generiert. Wird bei Struct immer vom Compiler generiert.

Automatisch generierter Konstruktor setzt alle Felder auf **Standard-Werte** (`null`, `0`, oder `false`).

Standardwerte können manuell verwendet werden

```

int x = default(int);
int y = default(); // ab C# 7.1 möglich

```

Statischer Konstruktor

- Zwingend parameterlos
- Keine Sichtbarkeit
- Nur einmal erlaubt
- Wird genau einmal ausgeführt:
 - bei erster Instanzierung des Typen
 - bei erstem Zugriff auf ein statisches Feld

```

Class MyClass()
{
    static MyClass()
    {
        // ...
    }
}

```

Destruktoren

Finalizer bei Java Bei Klassen erlaubt, bei Structs verboten (keine Möglichkeit zum Ausführen von Logik beim Abräumen)

```

class MyClass()
{
    ~MyClass()
    {
        // Freigabe von Ressourcen ...
    }
}

```

Initialisierungs-Reihenfolge

Klasse Sub: Statische Felder -> Statischer Konstruktor -> Felder

--> **Klasse Base:** Statische Felder -> Statischer Konstruktor -> Felder
--> --> **Klasse Object:** Statische Felder -> Statischer Konstruktor -> Felder
Klasse Base: Konstruktor
Konstruktoren in Ober- oder Unterklassen
Impliziter und expliziter Aufruf des Basisklassen-Konstruktor

```

class Sub : Base
{
    public Sub(int x)
        :base(x)
    {
        // ...
    }
}

```

Operator Overloading

Methode muss `static` sein, erzeugt eine eigene Instanz.

```

public static Point operator + (Point a, Point b)
{
    return new (a._x + b._x, a._y + b._y);
}

```

Partielle Klassen

Klassen, Structs, Interfaces

Definition von einem Typen in mehreren Files möglich.

Zweck:

- Arbeiten mit Generatoren
- Aufsplitten grosser Dateien
- Arbeiten mit schlechter Codequalität

Selbe Sichtbarkeit, Keyword `partial` zwingend bei allen Klassenteilen Implementation von Interfaces, `sealed` kann auch über alle Teile verwendet werden.

Partielle Methoden

Ermöglicht Benutzer-Definierte *Hooks* in generiertem Code.

Funktioniert in Klassen / Structs (muss ebenso `partial` sein).

Kann, muss aber nicht implementiert werden:

```

// File 1
partial class MyClass
{
    partial void Test1Initialize();
    partial void Test1Cleanup();
}

// File 2
partial class MyClass
{
    public void Test2() { }
    partial void Test1Initialize() { /*Implementation*/ }
}

```

Vererbung

Grundlagen

Regeln:

- Nur eine Basisklasse, beliebig viele Interfaces
- Structs können nicht erweitert werden oder erben, aber Interfaces implementieren
- Alle Klassen sind direkt oder indirekt von `System.Object` abgeleitet
- Structs sind über Boxing mit `System.Object` kompatibel

Unterschieden wird zwischen dem **statischen Typ** der Variable und dem **dynamischen Typ** des Objekts auf dem Heap.

Zuweisung ist erlaubt wenn statisch = dynamisch oder statisch Basisklasse von dynamisch, nicht erlaubt wenn statisch Subklasse von dynamisch, oder beide nicht in derselben Vererbungshierarchie sind.

Typprüfungen

- `<var> is <Class>` liefert true oder false zurück, wenn der Typ identisch oder eine Subklasse davon ist.
- Explizite Type Casts mit `(<class>)<var>` weisen den Compiler an, einen Typ in einen anderen umzuwandeln
- Umwandlung mit `<var> as <Class>` bewirkt den Output `obj is T ? (T)obj : (T)null`
- Prüfen mit casted als `if (a is SubSub something) { .. } => SubSub something = default; if (a is SubSub) { something = (SubSub) a }`. Arbeitet also mit `default` statt `null` wie "as"-Operator.

Klassen

Methoden

Subklasse kann Members der Basisklasse überschreiben. Schlüsselwort `virtual` macht Basis-Methode überschreibbar, `override` um in Subklasse effektiv zu überschreiben. Wichtig: beide müssen immer in Kombination angegeben werden, sonst wird nichts überschrieben. Signatur der Methoden muss exakt gleich sein.

Kombination `virtual` nicht möglich mit `static`, `private`, `override`, auch nicht mit `abstract` da dies `virtual` impliziert.

Dynamic Binding

Erlaubt das Aufrufen einer Methode vom dynamischen Typ, auch wenn der statische Typ ein anderer ist. FALLS die Methode im statischen Typ `virtual` markiert ist, suche Vererbungshierarchie von oben nach unten nach konkretester Methode mit Schlüsselwort `override`.

Methoden überdecken: wenn gewünscht, Keyword `new` verwenden.

Ablauf der Methodenwahl: Für jeden Type der Hierarchie zwischen static type und dynamic type: wenn `override` Method vorhanden, wähle diese, sobald eine non-`override` methode (`new`) gefunden wird, breche ab und verwende die zuletzt gefundene Methode.

Achtung: Gecastet wird immer der Static Type!

Abstrakte Klassen

Mischung aus Klasse und Interface. Kann nicht direkt instanziiert werden. Kann beliebig viele Interfaces implementieren, wenn alle Members deklariert sind. Abgeleitete Klassen müssen alle abstrakten Members implementieren, Abstrakte Members können nur innerhalb abstrakter Klassen deklariert werden.

Abstrakte Methoden

Kein Anweisungsteil, sondern direkt mit ; abgeschlossen: `public abstract void Add(object x);`. Implizit `virtual`, dürfen nicht `static` oder `virtual` deklariert werden.

Abstrakte Properties

Kein Anweisungsteil, get und set mit Semikolon abgeschlossen. Implizit `virtual`, dürfen nicht `static` oder `virtual` deklariert werden. `get/set` Kombination muss bei Implementation identisch sein.

Versiegelte Klassen

Keyword `sealed` verhindert das Ableiten einer Klasse, analog Java `final`. Aspekte: Sicherheit (kein versehentliches Erweitern), Performance (Methoden können statisch gebunden werden). Verwendung bei Klassen und deren Methoden, Properties, Indexern und Events.

Versiegelte Members

Verhindert das Überschreiben eines bestimmten Klassenmembers. Kann nicht in einer abstrakten Klasse vorkommen. Ableitung einer Abstrakten Klasse kann Members `public override sealed` definieren, um weitere Ableitungen zu verhindern. `public new virtual` desselben Members ist in abgeleiteten Klassen aber wiederum möglich.

Interfaces

Kann nicht direkt instanziiert werden, keine Sichtbarkeit auf Members. Kann andere Interfaces erweitern. Members sind implizit `abstract` `virtual`. Dürfen nicht `static` sein oder ausprogrammiert werden. Name beginnt mit grossem I .

Erlaubte Inhalte: Methoden, Properties, Indexers, Events

Implementieren

- Klasse kann beliebig viele Interfaces implementieren
- Alle Interface-Members müssen auf der Klasse vorhanden sein, entweder direkt implementiert oder von anderer Basisklasse geerbt
- `override` ist nicht nötig, ausser Basisklasse definiert den gleichen Member
- Kombination mit `virtual` und `abstract` ist erlaubt
- Implementierte Interface-Members müssen `public` und nicht `static` sein.

Verwenden

```
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) { /* Implementierung */ }
}
```

```
public string Name { get { /* Implementierung */ } }
public object this[int i] { get { /* Implementierung */ } set { /* Implementierung */ } }
public event EventHandler OnAdd;

// Zuweisung/Verwendung
List list1 = new List();
list1.Add("Hello");
ISequence list2 = new List();
list2.Add("Hello");
// Typumwandlung
list1 = (List)list2;
list1 = list2 as List;
list2 = list1;
// Typprüfung
if (list1 is ISequence) { /* ... */ }
```

Naming Clashes

Zwei Interfaces könnten dieselben Member definieren, was zu einer Kollision führt.

Variante 1: Selbe Signatur, selber Rückgabetyp, und Logik ist identisch. Dann kann die Methode ganz normal implementiert werden, ohne weitere Angaben.

Variante 2: Selbe Signatur, selber Rückgabetyp. Logisch ist unterschiedlich. Dann müssen die Methoden zur Implementierung explizit benannt werden. Bsp. `void ISequence.Add(object x) { ... }` `void IShoppingCart.Add(object x) { ... }`

Variante 3: Selbe Signatur, selber Rückgabetyp, Logisch ist unterschiedlich aber ein Default-Verhalten ist definierbar. Dann kann eine Methode 'normal' und eine Methode explizit implementiert werden. Bsp. `void Add(object x) { ... }` `void IShoppingCart.Add(object x) { ... }`

Fragile Base Class Problem

Das Markieren von `virtual` und `override` verhindert, dass bei nachträglich hinzugefügten Methoden in einer Basisklasse ein Konflikt mit allenfalls schon vorhandenen Methoden in der Subklasse mit demselben Namen entstehen. Schlimmstenfalls wirft der Compiler eine Warnung, wenn der Code neu kompiliert wird. In Java beispielsweise führt das ohne Warnung zu anderem Verhalten.

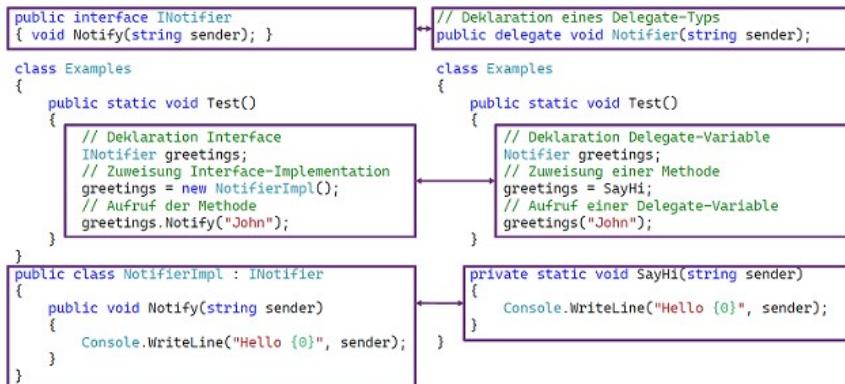
Delegates, Events, Lambdas

"Normale" Delegates

Verwendungszwecke: Methoden als Funktionsparameter, Callback / Observer Pattern

Delegates sind typsichere Funktionspointer, abgeleitet von `System.Delegate`, somit eine Referenz auf 0-n Methoden. Analog Function Pointers in C++. Deklaration geschieht auf Namespace-Level. Jeder Delegate wird vom Compiler als Klasse generiert (abgeleitet von `MulticastDelegate`) und kann danach instanziert und zugewiesen werden. Null-Zuweisung ist möglich, wirft bei Ausführung `NullReferenceException`. Inversion of Control, Möglichkeit auch mit Interfaces.

Delegates sind eine explizite Vereinfachung von Interfaces mit genau einer Methode. Zuweisung einer Methode zur Delegate Variable entspricht der Instanzierung eines Interfaces.



Eine Variable vom Typ Delegate hat 3 Felder

- Target : Klasseninstanz, wenn die angegebene Methode nicht statisch ist.
- Method : Methode die zugewiesen wurde / ausgeführt wird
- Prev : siehe Multicast Delegates

```
// Verwendung von Delegate zur Übergabe von Funktionen als Methodenparameter
public delegate void Comparer(object x, object y);
class Program {
    static int CompareFraction(object x, object y) { ... }
    static int CompareString(object x, object y)
    { return (string)x ).CompareTo(y); }

    static void Sort(object[] a, Comparer compare) /* do some comparison using delegate */
    Comparer cf = CompareFraction; // Methode zu Delegate zuweisen
    Fraction[] fractions = new Fraction[] { new(1, 2), new(3,4) };
    Sort(fractions, cf) // Delegate übergeben
}
```

Wenn die Methode in derselben Klasse ist wie die Delegate Variable, ist `obj "this"` und kann weggelassen werden. Bei statischer Methode der Klassenname.

```
// Zuweisungs-Syntax
DelegateType delegateVar = obj.Method;
```

Zugewiesene Methode

- darf nicht abstract sein
- darf virtual, override und/oder new sein
- muss identisch sein in der Signatur (inkl. Anzahl, Art und Typen der Parameter, Rückgabetyp) wie im Delegate definiert

```
// Aufruf-Syntax
object result = delegateVar.[.Invoke](params); // Mit Rückgabetyp
if (delegateVar != null) { delegateVar(params); } // Mit Null-Check
delegateVar?.Invoke(params); // Standard-Variante ab C# 6.0 mit Null-Check
```

Multicast Delegates

Mehrere registrierte Methoden werden synchron in der registrierten Reihenfolge ausgeführt. Exception führt zum Abbruch aller restlichen Methoden. Invoke-Methode regelt die Ausführung der jeweiligen Zielmethode.

Prev Feld: Referenz auf vorheriges (Multicast-)Delegate aus der Linked List.

```
Notifier greetings;
greetings = SayHi;
greetings += SayCiao; // fügt eine weitere Methode der Ausführungskette hinzu
greetings -= SayHi; // sucht "von hinten" in der linked List nach dem ersten Match und entfernt ihn.
Notifier a = SayHi; Notifier b = SayCiao;
Notifier c = a + b; // (Notifier)Delegate.Combine(a, b);
```

Compiler wandelt jeden "normal" definierten Delegate in eine eigene Klasse um, die von `MulticastDelegate` ableitet. Bei Methoden mit Rückgabewerten wird jeweils nur der letzte Wert behalten.

Events

• Base-Library Code / Version 1

• Client Code

```

public delegate void ClickEventHandler
    (object sender, ClickEventArgs e);
public class ClickEventArgs : EventArgs
{
    public string MouseButton { get; set; }
}

public class Button
{
    public event ClickEventHandler OnClick;
}

```

```

public class Usage
{
    public void Test()
    {
        Button b = new();
        b.OnClick += OnClick;
    }
    private void OnClick(
        object sender,
        ClickEventArgs eventArgs)
    {
        // Action
    }
}

```

Button: Observable / Usage: Observer

Definiert wird immer Event sowie ein Delegate, der die Struktur der Methode angibt, welche auf den Event hin aufgerufen wird (bzw. zum Event Handling angemeldet werden kann).

Der Delegate hat standardmäßig `-Handler` im Namen und befindet sich wiederum auf Namespace Level. Der Event gibt an, welchen Delegate er "verwendet" und wird an bestimmten Stellen im Code via `EventName?.Invoke(params)` ausgelöst.

```

public delegate void TickEventHandler (int ticks, int interval);
public class Clock
{
    // Selber definierter Event
    public event TickEventHandler OnTickEvent;
    // Compiler-generierter Output
    private TickEventHandler OnTickEvent; // Delegate Instanzierung wird private
    // Subscribe/Unsubscribe Logik wird public generiert, Zugriff via Event (+/-=)
    public void add_OnTickEvent(TickEventHandler h) { OnTickEvent += h; }
    public void remove_OnTickEvent(TickEventHandler h) { OnTickEvent -= h; }
}

```

Standardmäßige Delegate Parameter

- `sender` Absender übergibt bei Aufruf des Delegates / Events «`this`» mit. In UIs meist ein Button oder ähnliches.
- `EventArgs` : Argumente des Events werden als Instanz einer von `EventArgs` abgeleiteten Klasse übergeben, falls benötigt. Klasse enthält 0-n Properties, die weitere Angaben zum Event machen. Beispiel: welche Maustaste beim Click, welche Zeit, welche Position...

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

```

public class ClickEventArgs : EventArgs /* kann beliebig um weitere parameter ergänzt werden*/
{
    public string MouseButton { get; set; }
}

public class Button
{
    public event ClickEventHandler OnClick; // "Returntype" definiert Delegate, der den Event "handeln" kann
}
// Verwendung
public void Test() { Button b = new(); b.OnClick += OnClick; }
private void OnClick(object sender, ClickEventArgs eventArgs) { /* do something */ }

```

Zur Verwendung im "Client Code" muss nur eine entsprechende Methode die zum Delegate passt erstellt werden, und diese auf den Event einer Observable-Instanz registriert werden.

Lambda Expressions

Zuweisung von Methoden zu Delegates oder Events müssen nicht zwingend benannt sein, die entsprechenden Methoden können als Lambdas In-Place definiert werden.

```

Func<int, bool> fe = i => i % 2 == 0; // Expression Lambda: "Func<param, param, .., return type>"
Func<int, bool> fs = i =>
{
    int rest = i%2;
    return rest == 0;
} // Statement Lambda, mehrere Zeilen mit { } eingepackt

```

Varianten von Parametern

```

p01 = () => true; // Kein Parameter
p02 = (a) => true; p02 = a => true; // Ein Parameter (Klammer optional)
p03 = (a, b) => true; // etc...
p04 = (ref int a) => true; // Ref oder Out Parameter auch möglich

```

Statement Lambdas: theoretisch unbegrenzte Anzahl Statements, idealerweise aber nicht mehr als 2-3 verwenden. Sonst besser eine named Methode inkl. Tests.

```

Func< int, int, int, bool> p04;
↓   ↓   ↓
p04 = (a, b, c) => true;

```

Closure

Problem: Lambda Expression hat grundsätzlich keinen Zugriff auf die Variablen der umgebenden Methode/Klasse.

Lösung Closure: Compiler generiert eine Hilfsklasse, welche die Logik des Delegates enthält, inklusive aller Variablen, auf die das Delegate zugreift.

Generics

Grundlagen

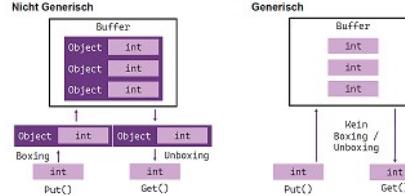
Typsichere Datenstrukturen ohne Festlegung auf einen fixen Typen. Anstelle dessen wird `<T>` verwendet. Bei der Verwendung wird `T` jeweils als spezifischer Typ deklariert. Hat gegenüber der Verwendung von `object` den Vorteil der Typsicherheit. Weiter hohe Wiederverwendbarkeit und Performance.

Hauptanwendungsfall: Collections. Ohne Generics müsste eine Variante pro Collection für jeden einzelnen Typ implementiert werden. Oder bei Verwendung von `Object` wären ständig Boxing/Typumwandlungen nötig.

Generisch sein kann *Klasse / Struct / Interface / Delegate / Event*. Sowie einzelne Methoden, auch wenn die Klasse nicht generisch ist.

Deklaration von `T` immer nach dem Namen des Elements.

Standard-Namensgebung bei einem generischen Parameter: "T", bei mehreren: "T1", "T2" etc. oder "TElement", "TKey" etc.



Type Constraints

Verschiedene Varianten von Constraints sind auf generischen Typen möglich. Angabe jeweils direkt nach bzw. auf der nächsten Zeile nach der Definition. Compiler prüft den Constraint. Kombinationen von verschiedenen Constraints kommagetrennt, pro T-Parameter eine where-Klausel mit Leerzeichen getrennt.

```
class OrderedBuffer<TElement, TPriority>
    where TPriority : IComparable, SomeConstraint
    where TElement : SomeConstraint
{ ... }
```

Constraint	Bedeutet
where T : struct	T muss ein Value Type sein, heisst liegt auf dem Stack oder inline im anderen Objekt.
where T : class	T muss ein Reference Type sein, also Klassen oder auch Interfaces, Delegates. Liegt auf dem Heap.
where T : new()	T muss einen parameterlosen "public" Konstruktor haben, kann also instanziert werden. <i>Muss immer zuletzt angeführt werden, wenn mit anderen Constraints kombiniert.</i>
where T : "ClassName"	T muss von der angegebenen Klasse ableiten, bietet also alle Members.
where T : "InterfaceName"	T muss das Interface implementieren
where T : TOther	T muss identisch sein wie oder ableiten von TOther -> bei Extension Methods verwendet
where T : class?	T muss ein Nullable Reference Type sein. (Klassen, Interfaces, Delegates)
where T : not null	T muss ein Non-Nullable Value Type oder Reference Type sein

Motivation: möglichst wenige Einschränkung, nur genug um die gewünschte Logik zu implementieren.

Vererbung

Verschiedene Varianten möglich:

```
class MyList<T> : List { } // Erben von normalen Klassen
class MyList<T> : List<T> { } // Weitergabe des Typparameters an generische Basisklasse
class MyIntList : List<int> { } // Konkreisierte generische Basisklasse
class MyIntKeyDict<T> : Dictionary<int, T> { } // Mischform
// Achtung: Typparameter werden nicht vererbt
class myList : List<T> { } // Compilerfehler!
```

Zuweisungen

Zuweisung einer Instanz eines generischen Typen an nicht-generische Basistypen ist immer möglich. Wenn die Basisklasse schon generisch ist, müssen Typparameter auf jeden Fall kompatibel sein.

```
class MyList<T>
class MyList<T> : MyList<T>
class myDict<TKey, TValue> : MyList<TKey>
class Examples
{
    public void Test()
    {
        MyList<int> l1 = new MyList<int>();
        MyList<int> l2 = new MyDict<int, float>();
        // Compilerfehler
        MyList<int> l3 = new MyList<float>();
        MyList<object> l4 = new MyList<float>(); // wg. Co-/Kontravarianz, einfügen von Objects
    }
}
```

Methoden überschreiben

```
class Buffer<T> { public virtual void Put(T x) } // Basis
// Konkreisierte Basisklasse
class MyIntBuffer : Buffer<int>
{ public override void Put(int x) { ... } }
// Generische Vererbung
class MyBuffer<T> : Buffer <T>
{ public override void Put(T x) { ... } }
```

Typprüfungen und Type Casts

Können beide wie mit normalen Typen durchgeführt werden. Prüfung identisch wie für normale Typen mit `is`-Operator, Cast mit Klammer oder `as`-Operator. Reflection unterstützt abfrage von konkretisierten und nicht konkretisierten Typparametern: `Type t = typeof(Buffer<int>)`.

Generic Type Inference

Bei Methoden kann der Typparameter weggelassen werden, wenn `T` ein formaler Parameter ist. Damit wird der Typ vom Compiler ermittelt. Rückgabewert zählt nicht. Methode `public void Print<T>(T t) { }` kann direkt aufgerufen werden als `Print(12)`. Methode `public T Get<T>() { }` muss immer der Typ mitgegeben werden mit `Get<int>()`.

Konkretisierung auf CLR Ebene

Bei Value Types wird für jeden neuen Typen eine eigene Klasse mit konkreter Implementierung erstellt -> Performance. Für Reference Types wird einmal eine Klasse mit Typ `<object>` erzeugt und für jeden anderen Typen wiederverwendet.

Generische Delegates

Auch Delegates unterstützen generische Typen.

```
public delegate void Action<T> (T i);
public class MyClass
{
    public static void PrintValues<T>(T i) { Console.WriteLine("Value {0}", i); }
}
public class FunctionParameterTest
{
    static void ForAll<T>(T[] array, Action<T> a)
    {
        Console.WriteLine("ForAll called...");
        if (a == null) { return; }
        foreach (T t in array) { a(t); }
    }
}
```

In .NET vorimplementierte Delegates (in/out => Ko-/Kontravarianz)

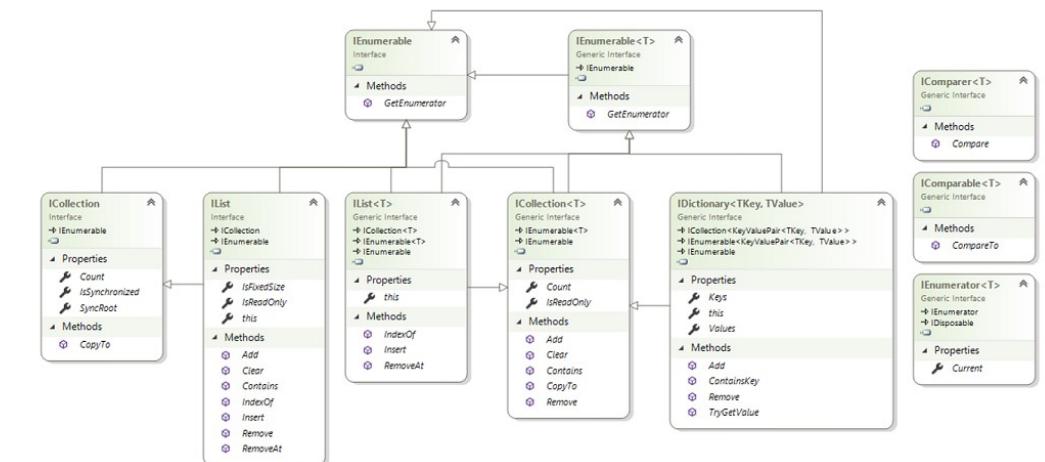
- public delegate void Action<in T, out T> (in T obj1, out T obj2)
- public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)
- public delegate bool Predicate<in T1, in T2>(T1 obj1, T2 obj2) (spezialfall vom Func Delegate. Rückgabetyp fixiert auf Boolean)

Event Handler Delegate:

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

Generische Collections

Namespace System.Collections.Generic	Namespace System.Collections
List<T> / IList<T>	ArrayList / IList
SortedList<TKey, TValue>	SortedList
Dictionary / IDictionary<TKey, TValue>	Hashtable / IDictionary
SortedDictionary<TKey, TValue>	SortedList
LinkedList<T>	-
Stack<T>	Stack
Queue<T>	Queue
IEnumerable<T> / IEnumerator<T>	IEnumerable / IEnumerator
ICollection<T>	ICollection



Ko- und Kontravarianz

Nicht prüfungsrelevant.

Nullable, Record Types

Default Operator

default(T) / default Liefert einen Default-Wert für jeden angegebenen Typen. Bei Reference Types: null , bei Value Types 0 , \0 oder false .

Nullable

Null = ein Null Pointer, also eine Referenz die (noch) keine Zuweisung hat.

ValueTypes

Ein int kann normalerweise nur 0 sein, nicht null . Mit dem ? Operator kann jeder Value Type auch null sein. Meist bei Datenverbindungen benötigt, da dort jeder Wert auch "nicht vorhanden" sein kann.

System.Nullable<T>.HasValue um auf Null zu prüfen. Einfacher Wrapper, der den Null-Zustand für Value Types abbildet. Zugriff auf die Variable, wenn der Zustand Null ist, wirft eine InvalidOperationException .

```
public struct Nullable<T> where T : struct
{
    public Nullable(T value);
    public bool HasValue { get; }
    public T Value { get; }

    int? x = 123;           // entspricht einem System.Nullable<int>
    int x1 = x.HasValue ? x.Value : default;   // Umwandlung zurück zum Int in 3 Varianten
    int x2 = x.GetValueOrDefault();
    int x3 = x.GetValueOrDefault(-1);
    int i = 123; int? x = i; int j = (int)x;   // Mögliche InvalidOperationException.
```

Reference Types

Nullable Reference Type: Muss enabled oder disabled werden. Geschieht im .csproj File oder im Code mit #nullable enable oder #nullable disable , bzw. #nullable restore für Projekt-Default.

```
<PropertyGroup>
<!-->
<Nullable>[disable|enable]</Nullable>
<!-->
</PropertyGroup>
```

Wenn Nullable disabled: Jeder Reference Type kann jederzeit auch null sein. **Wenn Nullable enabled:** Wenn ein Reference Type null sein kann, muss er mit dem ? Operator versehen werden.

```
string? x = null;
string y = null; // Compiler-Fehler.
```

Null-Checks

x is null / is not null prüft in jedem Fall auf eine Null-Referenz. Bei Value Types wird HasValue geprüft, bei Reference Types gilt object.ReferenceEquals(x, null);

x == null wahrscheinlich auch, der == Operator könnte aber überschrieben worden sein.

Weitere Operatoren

```
int? x = null;
int i = x ?? -1; // null-coalescing operator: entweder Wert von x oder -1 wenn null

int? x = null;
i ??= -1; // null-coalescing assignment-operator: wenn i = null dann -1 zuweisen

Action a = null;
a?.Invoke(); // null-conditional operator: rechten Teil ausführen, wenn nicht null

int? x = null;
int y = x!; // null-forgiving operator: übersteuert den Compiler (wenn du denkst, du weißt es besser)
```

Record Types

Reines Compiler-Feature zur Darstellung von Klassen, die Daten repräsentieren. Ziele:

- Reduziert den Boilerplate Code
- Vereinfacht die Arbeit mit Nullable Reference Types
- Record Types sollen immutable sein

```
// Beispiel Positional Syntax, kompakteste Schreibweise. Definition von Werten in Klammern nach Header
public record [class|struct] Person(
    int id,
    string name
);
// Beispiel "normale" Syntax, mehr manuelle Implementation. Automatisch: ToString und Equality wird generiert
public record Person
{
    public Person() : this(0, "") { }
    public Person(int id, string name) { Id = id; Name = name; }

    public int Id { get; init; } // Init Only Properties
    public string Name { get; init; }
}
// Mischung von beiden Varianten ist möglich.
```

Der Compiler generiert dazu bestimmte Members:

- Konstruktor
- Properties (init-only)
- Value equality
- Darstellung mit ToString
- Vererbung (z.B. Equality) wird berücksichtigt

Weitere Features

Vererbung: Basisklasse entweder System.Object oder auch ein Record.

```
public abstract record Person(int Id);
public record SpecialPerson(int Id, string Name) : Person(Id);
// Legt das Id Property in der SpecialPerson Klasse nicht neu an.
```

Value Equality: Vergleich mit == oder .Equals() vergleicht jedes einzelne Property, involviert auch Properties allfälliger Basisklassen. Nur ReferenceEquals() vergleicht wirklich die Memory Adresse.

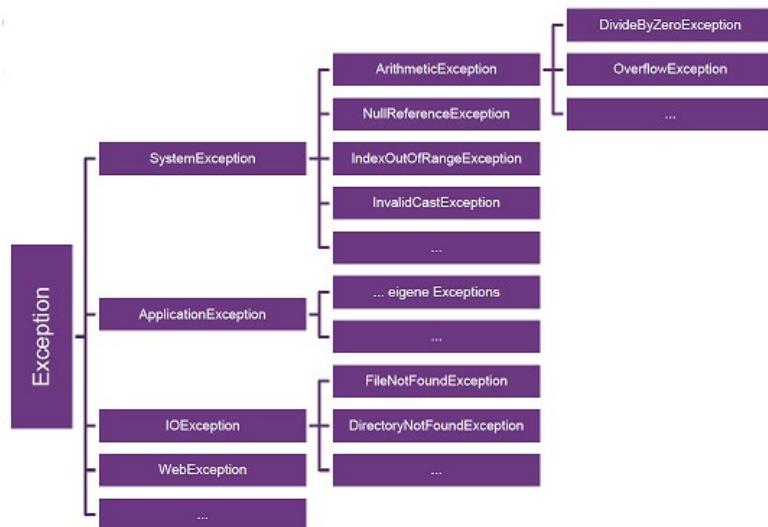
Nondestructive Mutation: Vereinfacht das Erzeugen leicht modifizierter Kopien.

```
Person p1 = new(0, "Ava");
Person p2 = p1 with { Id = 1 }; // Gleicher Name, andere Id
```

Exceptions

Behandeln unerwartete Programmzustände oder Ausnahmeverhalten zur Laufzeit. Best Practices dazu sind

- Wenn möglich Voreinstellungen prüfen um Exceptions zu vermeiden
- Exceptions sind "Fehlercodes" vorzuziehen
- Konkrete Fehlerbeschreibung: möglichst konkrete Klassen verwenden, möglichst detaillierte Beschreibung des Fehlers
- Fehlerbeschreibung NICHT über "Web-Schnittstellen" übermitteln, offenbart Internas und erhöht die Verletzbarkeit.
- Wichtig: Aufräumen nach Exceptions! (offene Sockets, File Handles, Transaktionen etc.)



Beispiel/Catch

```

try { ... something ... }
catch (FileNotFoundException e)           // Exception inklusive Name wenn verwendet
{ Console.WriteLine("{0} not found", e.FileName) }
catch (IOException)                      // Exception nur mit Typ
{ Console.WriteLine("IO exception occurred") }
catch                                // Jegliche Exception
{ Console.WriteLine("Unknown error occurred") }
finally
{ /* Wird immer ausgeführt */ }
  
```

Basisklasse System.Exception

```

public class Exception: ISerializable, _Exception
{
    public Exception();
    public Exception(string message);
    public Exception(string message, Exception innerException);

    public Exception InnerException { get; }    // Verschachtelung von Exceptions möglich
    public virtual string Message { get; }        // Fehlermeldung
    public virtual string Source { get; set; }     // Name der Applikation, Objekt, Framework
    public virtual string StackTrace { get; set; } // Methodenaufrufkette als String
    public MethodBase TargetSite { get; }          // Ausgeführt Codeteil, der den Fehler verursacht

    public override string ToString();            // Fehlermeldung und StackTrace als String
}
  
```

Throw

Werfen einer Exception, geschieht implizit bei ungültigen Operationen, bei Methodenaufrufen welche eine unbehandelte Exception werfen oder explizit mittels throw-Statement.

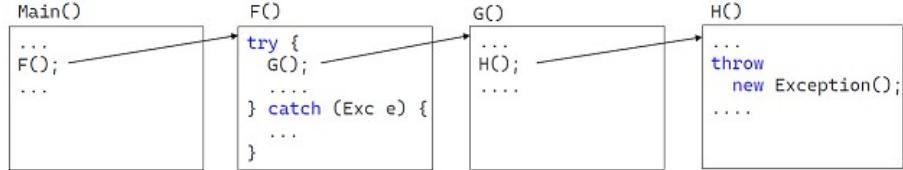
```

throw new Exception("An error occurred");

try { ... }
catch (Exception e)
{ throw e; }      // beginnt neuen Stack Trace
{ throw; }         // Stack Trace bleibt erhalten
  
```

Suche nach Catch-Klausel

Call stack wird rückwärts nach passender Catch Klausel durchsucht. Programmabbruch mit Fehlermeldung und Stack-Trace, falls keine Passende gefunden. (Multicast-) Delegates werden dabei wie normale Methoden behandelt, die nacheinander ausgeführt werden.



Exception Filters

Catch-Block kann auch nur unter definierten Bedingungen ausgeführt werden. when -Klausel erwartet eine Boolean Expression.

```

try { Console.WriteLine("Drink some beer") }
catch (Exception e) when (DateTime.Now.Hour < 16) { /* Kein Bier vor vier! */ }
  
```

Unchecked Exceptions

In C# kann jede Exception auch unbehandelt bleiben. Java benötigt dazu throws -Klausel in Methodensignatur.

Argumente prüfen

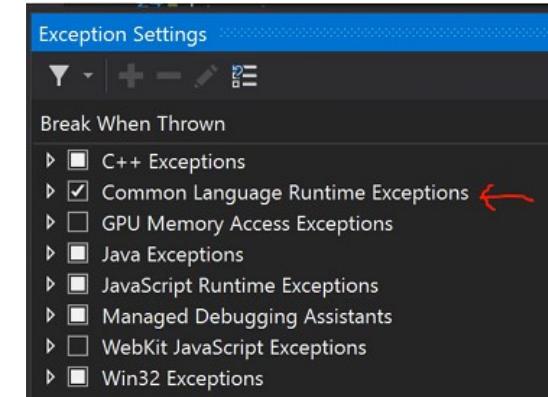
Refactoring-stabile Variante um Argumente auf Null-Werte oder Out Of Range zu prüfen:

```

string Replicate(string s, int nTimes)
{
    if (s == null)
        throw new ArgumentNullException(nameof(s));
}
  
```

First Chance Exception

Debugger wird genau an der Stelle angehalten, wo die Exception auftritt. Standardmäßig sonst dort, wo das Programm abstürzt.



Extension Methods

Erlauben das erweitern von bestehenden Klassen aus Anwendersicht. Die Signatur der Klasse wird jedoch nicht verändert. Zur Deklaration:

- Methode muss in statischer Klasse sein
- Methode muss statisch sein
- Erster Parameter definiert, auf welcher Klasse die Methode verfügbar ist und hat ein this Schlüsselwort voranstehtend.

```

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

Der Compiler wandelt den Aufruf von 1.ToStringSafe() in ExtensionMethods.ToStringSafe(1) um. Typsicherheit ist damit gewährleistet.

Regeln:

- Erlaubt auf Klassen, Structs, Interfaces, Delegates, Enumerators und Arrays
- Kein Zugriff auf interne Members
- Methode ist nur sichtbar, wenn der Namespace der beinhaltenden Klasse importiert wurde
- Bei Konflikt mit einer Methode der Zielklasse verliert die Extension Method immer

Iteratoren

Iterator/Enumerator und Iterable/Enumerable bedeuten jeweils dasselbe.

Zum Iterieren über Collections werden for each Loops verwendet. Jumps sind möglich mit continue (unterbricht aktuelle Iteration) und break (unterbricht gesamten Loop).

```
foreach (ElementType elem in collection)
{ statement }
```

Kriterien für die Collection: Muss IEnumerable/IEnumerable<T> implementieren oder einfach die nötigen Members implementiert haben:

Compiler-Output für ein foreach Loop:

```
IEnumerator enumerator = list.GetEnumerator();
try
{
    while (enumerator.MoveNext())
    {
        int i = (int)enumerator.Current;
        if (i == 3) continue; // Ursprüngliche Logik im foreach
        if (i == 5) break; // ...
        Console.WriteLine(i); // ...
    }
} finally
{
    IDisposable disposable = enumerator as IDisposable;
    if (disposable != null)
        disposable.Dispose();
}
```

IEnumerable/IEnumerator Interfaces

Definiert die Logik, wie in einer Collection "vorwärts gesprungen" wird. Array: Speicher x bytes nach vorne, Linked List: folge der Referenz.

- Hat eine Methode GetEnumerator() mit Rückgabewert einer Klasse e (Enumerator)
- Enumerator e hat eine Methode MoveNext() mit Rückgabewert bool . Zeigt jeweils auf ein Objekt in der Collection. Zeiger startet vor der Liste, wird beim ersten Aufruf auf das erste Listenelement verschoben.
- e hat ein Property Current , das jeweils das aktuelle Element zurückgibt.

Jeder Enumerator ist entweder allgemein (Rückgabetyp object , .NET 1.0) oder generisch (Rückgabetyp T , leitet jeweils vom nicht-generischen ab).

Wichtig: Rückgabetyp ist nicht Teil der Methodensignatur bezüglich Vererbung/Überladung.

Zwei verschiedene Methoden müssen definiert werden object Current {get;} T Current {get;}

MehrfaZ-Zugriff

Iterator kapselt den State (wo bin ich) der aktuellen Traversierung der Liste. Die Collection darf dabei aber nicht verändert werden. Dies erlaubt unterschiedlichen Threads Arbeit mit derselben Liste gleichzeitig.

Implementationsbeispiel

Jeweils zwei Klassen werden benötigt, die einmal IEnumerable und einmal IEnumerator implementieren. IEnumerable wäre jeweils die Collection, Enumerator beinhaltet die Traversierungslogik.

```
// Nicht generisch
class MyList : IEnumerable
{
    public IEnumerator GetEnumerator() { return new MyEnumerator(); }
    class MyEnumerator : IEnumerator // geschachtelt!
    {
        public object Current { get { /* ... */ } }
        public bool MoveNext() { /* ... */ }
        public void Reset() { /* ... */ }
    }
}

// Generisch
class MyIntList : IEnumerable<int>
{
    public IEnumerator<int> GetEnumerator() { return new MyEnumerator(); }
    IEnumerator IEnumerable.GetEnumerator() { return GetEnumerator(); } // Zusätzlich -> Aufruf auf Generisch.
    class MyEnumerator : IEnumerator // geschachtelt!
    {
        public int Current { get { /* ... */ } }
        object IEnumerator.Current { get { /* ... */ } } // Zusätzlich
        public bool MoveNext() { /* ... */ }
        public void Reset() { /* ... */ }
    }
}
```

Keyword yield

Iterator yield ersetzt spezifische implementation von IEnumerator. Innerhalb der Methode public IEnumerator GetEnumerator() kann jeweils mit einem yield return statement der Wert für die Nächste Iteration definiert werden. Typ des Return-Wertes muss mit der Signatur kompatibel sein. yield break terminiert die Iteration.

```
class MyIntList // ...standardmäßig wird yield return innerhalb eines Loops verwendet...
{
    public IEnumerator<int> GetEnumerator()
    {
        yield return 1;
        yield return 2;
        yield break;
        yield return 3; // wird nie aufgerufen..
    }
}
```

Compiler generiert im Hintergrund ein vollständiger Enumerator als State Machine (mit int __state, int __current) die jeweils die aktuelle Stelle im GetEnumerator kapselt. Somit kann beim nächsten Aufruf genau dort weitergemacht werden, wo bei der letzten Iteration aufgehört wurde.

Spezifische Iteratoren

Ermöglicht das festlegen von verschiedenen Iteratoren pro Klasse. Standarditerator wird aufgerufen via foreach (int elem in list) . Ein Spezifischer Iterator kann mit einer separaten Methode implementiert werden, der dann nicht mehr ein IEnumerator Objekt zurückgibt, sondern ein IEnumurable, das selber wiederum eine GetEnumerator() Methode besitzt.

```
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];
    }
    // spezifischer Iterator
    // Kann als Methode (hier) oder auch als Property (Logik innerhalb get {}) umgesetzt werden (keine Parameter)
    public IEnumurable<int> Range(int from, int to)
    {
        for (int i = from; i < to; i++)
            yield return _x[i];
    }
}
foreach (int elem in list.Range(2, 7)) // Aufruf
```

Deferred Evaluation

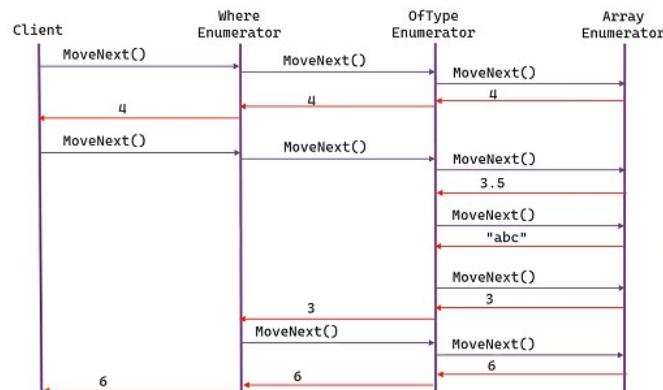
Die Iterator-Methode GetEnumerator() wird erst ausgeführt, wenn MoveNext() aufgerufen wird. Dies passiert im foreach Loop implizit erst dann, wenn effektiv das erste Element abgefragt wird. Somit wird nur so viel Arbeit gemacht, wie effektiv nötig. Dieses Konzept wird bei allen yield Aufrufen verwendet.

Implementation von Query Operatoren

Werden deferred evaluiert, verwenden Extension Method Syntax und können mit dem . -Operator verkettet werden, da sie ihrerseits wieder ein IEnumurable zurückliefern.

```
static class OstExtensions
{
    public static IEnumurable<T> OstWhere<T>(this IEnumurable<T> source, Predicate<T> predicate)
    {
        foreach (T item in source)
        {
            if (predicate(item)) yield return item;
        }
    }
    public static IEnumurable<T> OstOfType<T>(this IEnumurable source)
    {
        foreach (object item in source)
        {
            if (item is T) yield return (T)item;
        }
    }
}
// Anwendung
object[] list = { 4, 3.5, "abc", 3, 6 };
IEnumurable<int> res = list
    .OstOfType<int>()
    .OstWhere(k => k % 2 == 0);
foreach (int i in res) { Console.WriteLine(i); } // Filtern der Elemente in 'res' geschieht erst hier
```

Ablauf der Iteratoren im obigen Beispiel



LINQ

Language INtegrated Query. Erlaubt es, auf beliebigen Datenquellen (Objekte, XML, Datenbanken etc.) SQL-ähnliche Abfragen zu schreiben.

- Compiler-Geprüft
- Typsicher
- Führt die Lambda-Expression ein
- erlaubt deklarative Programmierstil mit "Anonymous Types" und "Object Initializers"
- redundante Typinformationen können im Code weggelassen werden

Namespaces: `System.Xml.Linq`, `System.Data.Linq`, etc.

Architektur

```
List<string> cities = new() { "Rapperswil", "Luzern", "Buchs", "Lausanne" };

IEnumerator<string> enumerator = cities.GetEnumerator();
string elem;
while (enumerator.MoveNext())
{
    elem = enumerator.Current;
    Console.WriteLine(elem);
}
```

LINQ Providers: LINQ To Objects, to Entities (SQL Server), to SQL (veraltet), to XML, to anything..

Array in Query wiedergegeben: `SELECT * FROM table;`

Query benötigt zum speichern ein Objekt mit Interface `IEnumerable<T>`, um über eine Collection iterieren zu können. LINQ Syntax wird vom Compiler auf Variante Extension Methods umgewandelt.

```
string[] cities = { "Bern", "Basel", "Luzern", "Rapperswil", "Buchs"};

// LINQ Query Expression: Reine Selektion auf ganzes Objekt
IQueryable<string> q1 =
    from c in cities
    select c;

// Extension Method Variante
IQueryable<string> l1 = cities.Select(c => c);

// LINQ Query Expression
IQueryable<string> q2 =
    from c in cities
    where c.StartsWith("B")
    orderby c
    select c;

// Extension Method Variante
IQueryable<string> l2 = cities
    .Where(c => c.StartsWith("B"))
    .OrderBy(c => c)
    .Select()
```

Extension Methods

Gesamter Funktionsumfang von LINQ ist beinhaltet in Klasse `System.Linq.Enumerable`. Extension Methods erlauben Ausführung auf einem bestehenden Objekt. Erster Parameter der Definition mit `this` Keyword beschreibt dieses Objekt.

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TSource> Skip<TSource>(
            this IEnumerable<TSource> source, Func<TSource, bool> predicate)
        { ... }
    }

    // Beispiel Verwendung
    int[] numbers = { 1, 4, 2, 9, 13, 8, 9, 0, -6, 12 };
    IQueryable<int> res = numbers
        .Skip(2)
        .Take(4)
        .Select(k => k * k);
    // res = { 4, 81, 256, 64 }
```

Deferred Evaluation

Query wird erst ausgeführt, wenn auch wirklich eine Abfrage auf dem Enumerator ausgeführt wird mit `MoveNext()`, dies wird mittels `yield return` implementiert. Klappt nur mit `IEnumerable` Typen, andernfalls wird Immediate Evaluation gemacht (Rückgabetyp Liste o.ä.).

Immediate Evaluation

Immer dann, wenn der Rückgabetyp kein `IEnumerable` ist oder eine Aggregierung über alle Elemente gemacht wird.

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

Object Initializers

Erlaubt das Instanziieren und Initialisieren einer Klasse in einem einzigen Statement, auch ohne passenden Konstruktor. Reine Compilertechnologie, wird aufgeschlüsselt in einzelne Zuweisungsstatements.

```
new Test { member = value, ... }; // Default Konstruktor
new Test(value) { member = value, ... }; // Konstruktor mit Argumenten

// Beispiel innerhalb LINQ
class Student { ... }

public void Test()
{
    int[] ids = { 2009001, 2009002, 2009003 };
    IEnumerables<Student> students = ids
        .Select(n => new Student { Id = n });
}
```

Collection Initializers

Dasselbe gilt für Collections:

```
// Listen
List<int> list1 = new() { 1, 2, 3, 4 };

// Dictionaries, Hashmap, HashSet
Dictionary<int, string> dict1 = new()
{
    {1, "a"},
    {2, "b"},
    {3, "c"}
};
// oder
dict1 = new()
{
    [1] = "a",
    [2] = "b",
    [3] = "c"
}
```

```
// Compiler-Output
List<int> l1 = new();
l1.Add(1);
l1.Add(2);
l1.Add(3);
l1.Add(4);

Dictionary<int, string> d1 = new();
d1.Add(1, "a");
d1.Add(2, "b");
d1.Add(3, "c");

d1 = new();
d1[1] = "a";
d1[2] = "b";
d1[3] = "c";
```

```
public void Test()
{
    List<int> l1 = new() { 1, 2, 3, 4 };

    Dictionary<int, string> d1 = new()
    {
        { 1, "a" },
        { 2, "b" },
        { 3, "c" }
    };
    d1 = new()
    {
        [1] = "a",
        [2] = "b",
        [3] = "c"
    };
}
```

Können in Kombination verwendet werden, um komplexe Objektstrukturen aufzubauen.

```
// Macht zwar keinen Sinn, aber hey :
Dictionary<int, Student> s = new()
{
    { 2009001, new("John") {
        Id = 2009001,
        Subject = "Computing" } },
    { 2009002, new() {
        name = "Ann", Id = 2009002,
        Subject = "Mathematics" } }
};
```

Keyword "var"

Definiert einen Typ, dessen Namen man nicht kennt. Kommt ausschließlich in Kombination mit `var` zum Einsatz. Compiler ermittelt, was für ein Typ zugewiesen wird (Type Inference).

```
// Vereinfachung der Lesbarkeit
Dictionary<string, IComparable<object>> dict1 = new Dictionary<string, IComparable<object>>();
var dict1 = new Dictionary<string, IComparable<object>>();

// Erlaubt anonyme Typen
var a = new { Id = 1, Name = "John"};
```

Anonymous Types

Erzeugung eines strukturierten Wertes, ohne dass der Typ explizit selber angegeben werden muss. Meist für Projektion in LINQ Queries verwendet. Ziel: Zwischenresultate, die nur für eine einzelne Abfrage gelten, müssen mit möglichst wenig Aufwand zwischengespeichert werden.

werden.

Syntax:

```
new { "p1" = "v1", "p2" = "v2", ... } : der Propertyname p1 - pn ist frei wählbar.
▪ Alle Properties sind readonly
▪ Implizite und explizite Angabe der Property Namen können gemischt werden
▪ Leitet von System.Object ab, Compiler generiert Methoden equals(), GetHashCode() und ToString()
▪ Kann nur einer Variable vom Typ var zugewiesen werden.
▪ Name des Typen wird vom Compiler so generiert, dass eine Kollision mit User-Vergebenen Names unmöglich ist (beginnend mit <> ).
```

Query Expressions

LINQ Query Expressions werden in drei Schritten ausgeführt.

```
// 1. Datenquelle auswählen
int[] numbers = { 0, 1, 2, 3, 4, 5 }
// 2. Query erstellen
var numQuery = from num in numbers
    where (num % 2) == 0
    select num;
// 3. Query ausführen
foreach (int num in numQuery)
{
    Console.WriteLine("{0, 1} ", num);
}
```

Schlüsselwort	Bedeutung	Speziell
from	Definiert Range-Variable und Datenquelle	Beginnt immer das Query
where	Filter	
orderby field [asc desc]	Sortierung	
select	Projektion	Beendet das Query, wenn kein group vorhanden ist
group	Gruppierung in eine Sequenz von Gruppenelementen	Beendet das Query
join	Verknüpfung zweier Datenquellen	
let	Definition von Hilfsvariablen	

Extension Methoden

Standard	Positional	Set Operations (bei Joins)
Select	FirstOrDefault]	Distinct
Where	SingleOrDefault]	Union
OrderBy[Descending]	ElementAt	Intersection
ThenBy[Descending]	Take / Skip	Except
GroupBy	TakeWhile / SkipWhile	Repeat
Join / GroupJoin	Reverse	
Count / Sum / Min / Max / Average		

Range Variablen

Entstehen durch Klauseln from, join, into . Sind Readonly, und nur innerhalb der Expression bis zur nächsten into Klausel sichtbar. Typ ist immer T bei Datenquelle IEnumerable<T> .

Gruppierung

Bringt eine flache Liste in eine strukturierte Form. Ausgabe IGrouping<TKey, TElement> implementiert ein IEnumerable mit zusätzlichem Namen (Key). Dieser repräsentiert das Attribut, welches gruppiert wurde bzw. bei allen Elementen im IEnumerable gleich ist. Zugriff auf die Members genau wie bei IEnumerable Typen.

```
var q = from s in Students
    group s.Name by s.Subject;
// Ergibt zuerst ein Key Value Pair
{
    {Computing, John},
    {Computing, Sue},
    {Mathematics, Ann}
}

// Danach ein IGrouping<TKey, TElement>
{
    Computing, [ John, Sue ],
    Mathematics, [ Ann ]
}

// Ausgabe mit doppeltem ForEach Loop
foreach (var group in q)
{
    Console.WriteLine(group.Key);
    foreach (var name in group) { Console.WriteLine(" " + name); }
}
```

// Direkte weiterverwendung mit "into"

```
var q = from s in Students
    group s.Name by s.Subject into g // ab hier kann nur noch g verwendet werden
    select new
    {
        Field = g.Key,
        N = g.Count()
    };

```

InnerJoins

Datensätze, die auf der anderen Tabelle keine Einträge besitzen, fallen weg.

```
// Explizit - besser für Performance
// on a.X equals b.Y
var q = from s in Students
    join m in Markings
        on s.Id equals m.StudentId
    select s.Name + ", " + m.Course + ", " + m.Mark;

// Implizit - gesamtes Kreuzprodukt wird anschliessend gefiltert
// where a.X == b.Y
var q = from s in Students
    from m in Markings
    where s.Id == m.StudentId
    select s.Name + ", " + m.Course + ", " + m.Mark;
```

Group Joins

Pro Student wird eine Liste für alle Markings erstellt. Ursprüngliche Tabelle bleibt sichtbar.

```
var q = from s in Students
    join m in Markings
        on s.Id equals m.StudentId
    into list // Enthält die Werte der zweiten Tabelle
    select new
    {
        Name = s.Name,
        marks = list // Enthält die Werte der zweiten Tabelle
    };
// Ausgabe
foreach (var group in q)
{
    Console.WriteLine(group.Name);
    foreach (var m in group.Marks) { Console.WriteLine(m.Course); }
}
```

LeftOuterJoins

Wie Inner bzw. Group Join, aber Elemente, für die kein Äquivalent gefunden wird, bleiben bestehen. Zwei from Statements nötig.

```
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);
```

Let Klausel

Zur Definition von Hilfsvariablen, die berechnet werden können. Lebt für den Rest des Queries.

```
var result = from s in Students
    let year = s.Id / 1000 // Hilfsvariable berechnen
    where year == 2009 // Hilfsvariable filtern
    select s.Name + " " + year.ToString(); // Hilfsvariable ausgeben
```

SelectMany

Erleichtert das Zusammenfassen verschachtelter Listen, z.B. List<List<string>> . Innere Listen von verschiedenen Elementen werden so als eine grosse Liste dargestellt.

```
// Extension Method
var q1 = list.SelectMany(s => s.SomeList);
// Query Syntax
var q2 = from segment in list
    from token in segment
    select token;
```

Tasks, Async/Await

Tasks

Platzhalter für ein Resultat, welches noch nicht bekannt ist, hat einen Rückgabewert. Repräsentiert eine asynchrone Operation. Convenience Wrapper um einen C# Thread (= 1 OS Thread) herum. Vereinfacht Programmfluss mit async/await.

```
Task<int> t1 = Task.Factory.StartNew(() => { Thread.Sleep(2000); return 1; });
Task<int> t2 = Task.Run(() => { Thread.Sleep(2000); return 1; });
```

Synchrone Waits

Return aus der Methode nachdem die gesamte Logik durchlaufen wurde. Blockieren den Aufrufer / aktuellen Thread (Problematisch in GUI Applikationen!). Beispiele: t1.Wait(), Task.WaitAll(t1, t2), t1.Result, t1.GetAwaiter().GetResult()

```
while (!t1.IsCompleted) // Busy Wait (bad Idea!)
{
    /* Do some stuff */
    int result1 = t1.Result;
    t1.Wait(); Task.WaitAll(t1, t2); // Explicit Wait
    int result2 = t1.Result;
    int result3 = t1.GetAwaiter().GetResult(); // Awaiter, optimierteres Exception Handling
```

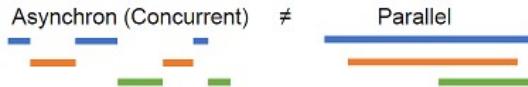
Asynchrone Waits

Ruft eine Methode auf, ohne auf das Resultat zu warten. Möglichkeit zur Benachrichtigung via Callbacks, oder Rückgabe eines Task - Objektes das abgefragt werden kann.

Continuations werden erst dann ausgeführt, wenn das Resultat vorhanden ist. Sind somit nicht-blockierend.

```
Task<int> t1 = GetSomeCustomerIdAsync();
t1.ContinueWith(id => // id: Result Wert vom Task, sobald vorhanden...
{
    Task<string> t2 = GetOrdersAsync(id.Result); // nicht mehr blockierend
    t2.ContinueWith(order => // order: Result Wert vom Task, sobald vorhanden...
        Console.WriteLine(order.Result) // nicht mehr blockierend
    );
});
```

Merke: Asynchron nutzt denselben Thread für verschiedene Aufgaben. Parallel nutzt verschiedene Threads.



Frühstück asynchron kochen heisst, mögliche Wartezeit für die nächste Aufgabe zu nutzen. Parallel kochen heisst, 3 Köche stehen vor einer anderen Pfanne und warten. Geht auch nicht schneller...

async/await

Keyword `async` markiert eine Methode als asynchron. Rückgabetypen sind dabei eingeschränkt auf `Task` (Methode ohne Rückgabewert), `Task<T>` (Methode mit Rückgabewert vom Typ `T`), oder `void` (fire and forget).

Alles was nach dem Keyword `await` folgt, wird vom Compiler in eine Continuation umgewandelt. Keyword ist nur in `async`-Methoden erlaubt.

`Task.WhenAll(t1, t2)` wandelt das abwarten von mehreren Tasks in einen einzigen Task um, der erst beendet ist, wenn alle Subtasks auch fertig sind:

```
string[] allResults = await Task.WhenAll(t1, t2)
```

Overhead für Task bei ca 10ms (?) -> lohnt sich erst bei grösseren Tasks.

.. eine asynchrone Methode (`computePiAsync()`) zu implementieren, bedeutet NICHT dass die Methode an sich `async` sein muss. Sie gibt einfach einen Task zurück.

Cancellation Support

Integriertes Programmiermodell für das Abbrechen von lange dauernder Programmlogik. Ein `CancellationToken` wird durch die gesamte Aufrufkette durchgereicht. Ist bei jeder asynchrone Methode der letzte Parameter (Best Practice).

Kann erstellt werden über die Klasse `CancellationTokenSource`. Wenn der Abbruch via Source ausgelöst wird, wird für alle ihre Tokens ein Abbruch angefordert.

```
CancellationTokenSource cts = new(); // Source wird instanziiert
CancellationToken ct1 = cts.Token; // Emittierung eines Token via Property
cts.Cancel(); // Bricht alle laufenden Operationen ab
```

Beispiel:

```
CancellationTokenSource cts = new();
CancellationToken ct = cts.Token;
Task t1 = LongRunning(1000, ct);
Task t2 = LongRunning(2000, ct);
Task t3 = Longrunning(3000, default); // independent, not cancellable

await Task.Delay(2000, ct);
cts.Cancel();

async Task LongRunning(int ms, CancellationToken ct)
{
```

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

Auslösen von `cts.Cancel()` wird meist mit Ctrl-C auf der Konsole verknüpft.

```
Console.CancelKeyPress += (_, e) =>
{
    cts.Cancel();
    e.Cancel = true;
};
```

EntityFrameworkCore

Ein providerbasiertes OR Mapping Framework. Neue Provider sind als Pakete über NuGet verfügbar und müssen nicht per se von Microsoft sein. Gängige Provider sind MS SQL Server, SQLite, PostgreSQL, In-Memory Provider.

Basis-Funktionalitäten von EF Core beinhalten:

- Mapping von Entitäten
- CRUD-Operationen
- Key-Generierung
- Caching (Performance)
- Change Tracking (Logs)
- Optimistic Concurrency
- Transaction Management
- Command Line Interface

ORMapping - Object/Relational Mapping

ORMapping umfasst immer 3 Komponenten:

1. Objektgraph / **EntityType** / Konzeptionelles Modell (im Code, normale .NET Klassen)
2. **Storage Entity** / Datenquelle / Relationales Modell (Tabelle oder Collection)
3. Mapping zwischen 1 und 2

EntityType

Entspricht einer normalen Klasse im .NET Code, beinhaltet

- Properties
- Entity Keys und Alternate Keys
- Relationships/Associations
- Foreign Keys

Eine **Entity** bezeichnet einen strukturierten Datensatz mit einem Key (OO: Objekt).

Gruppieren in Entity Sets, welche alle Instanzen eines Entity Types (OO: Klasse) beinhalten.

Speziell für OR Mapping: Entitäten können voneinander erben, ungleich in relationalen Datenbanken.

Eine **Relationship** oder Association definiert eine Beziehung. Besitzt jeweils zwei Enden mit Kardinalitäten, Abgebildet durch Navigation Properties oder Association Sets (EF spezifisch) bzw. Foreign Keys.

StorageEntity

Kann eines von folgenden sein

- Tabelle in einer relationalen DB
- View
- Stored Procedures
- (Graph, Collection, ...)

, und beinhaltet *Columns, Primary Keys, Unique Key Constraints, Foreign Keys*.

Mapping

Mapping von Entities, Property zu Column, etc. Muss nicht immer 1:1 sein.

Erweiterte Szenarien:

- Vererbung
- Table Split, Merge möglich
- Complex Types (Bsp. Kapselung von Adresse als eigenes Objekt innerhalb der Customer Tabelle)

3 verschiedene Ansätze zum Mapping von Objekten

Database First

Generiert das Mapping und die Objekte auf Basis einer bestehenden Datenbank. Wird schon seit Anfang von EF und EF Core unterstützt.

Benötigt zusätzliche Projektreferenzen mit `dotnet add package`

`Microsoft.EntityFrameworkCore.Tools`

`Microsoft.EntityFrameworkCore.Design`

`Microsoft.EntityFrameworkCore.SqlServer`

Scaffolding des Codes ausführen mit folgendem Command (-o erstellt die Code Klassen in einem neuen Unterordner)

`dotnet ef dbcontext scaffold <connectionstring> <provider als NuGet Paket> -o Models`

ModelFirst

Generiert die Datenbank, das Mapping und die Objekte basierend auf einem Model (in EF Core nicht verfügbar, hier nicht relevant)

CodeFirst

Generiert das Mapping und die Datenbank auf Basis des Codes (EF 5.0+, EF Core 1.0+), unter Verwendung der 3 verschiedenen Syntaxen.

Ansatz	Funktionsweise
By Convention	Verwendet Klassennamen pluralized: Klasse <code>Category</code> -> Tabelle <code>Categories</code> Property wird nach Name übernommen

Ansatz	Funktionsweise
By Fluent API	<p>Funktionsweise</p> <p>Verwendet <code>ModelBuilder</code> Objekt in überschriebener Methode <code>OnModelCreating</code></p> <pre>[...].Entity<Category>().ToTable("Categories") .HasKey(c => c.Id).HasColumnName("Id")</pre>
By Attributes	<p>Mapping - Providerunabhängig</p> <p>Include/Exclude von Entities</p> <pre>modelBuilder.Entity<Category>(); modelBuilder.Ignore<Metadata>(); // Fluent API [NotMapped] public class Metadata { /* ... */ }</pre> <p>Include/Exclude von Properties</p> <p>Convention: Alle public Properties mit getter/setter werden gemappt.</p> <pre>modelBuilder.Entity<Category>() .Property(e => e.Name); modelBuilder.Entity<Category>() .Ignore(e => e.LoadedFromDatabase); // Fluent API public class Category { [NotMapped] // Data Annotations public string LoadedFromDatabase { get; set; }</pre> <p>Keys</p> <p>Convention: Property mit dem Namen <code>[Entity]Id</code></p> <pre>modelBuilder.Entity<Category>() // Fluent API .HasKey(e => e.Id); // ODER .HasKey(e => new { e.Id, e.Name }) // FluentAPI: einzige Variante für Composite Keys public class Category { [Key] // Data Annotations public int Id { get; set; }</pre> <p>Required/Optional</p> <p>Convention: Value Types werden NOT NULL, Nullable Value Types und Reference Types werden NULL.</p> <pre>modelBuilder.Entity<Category>() // Fluent API .Property(e => e.Name) .IsRequired() // ODER .IsOptional(); public class Category { [Required] // Data Annotations public string name { get; set; }</pre> <p>StringLength</p> <p>Convention: Keine Restriktion auf normalen String Feldern (<code>NVARCHAR(MAX)</code>), 450 Zeichen bei Keys (Primärschlüssel).</p> <pre>modelBuilder.Entity<Category>() // Fluent API .Property(e => e.Name) .HasMaxLength(200); public class Category { [MaxLength(200)] // Data Annotations public string Name { get; set; }</pre> <p>Unicode</p> <p>In Datenbank ist NVARCHAR immer Unicode, ein "normaler" VARCHAR kann nicht by Convention erstellt werden.</p> <pre>modelBuilder.Entity<Category>() // Fluent API .Property(e => e.Name) .IsUnicode(false); public class Category { [Unicode(false)] // Data Annotations public string Name { get; set; }</pre> <p>Precision</p> <p>Verwendet zwei Parameter. Precision: Stellen insgesamt, Scale: Anzahl Nachkommastellen (innerhalb Precision, Angabe optional)</p> <p>Convention: pro Datentyp vom Provider festgelegt.</p>

Ansatz	Funktionsweise
By Convention	Verwendet Klassennamen pluralized: Klasse <code>Category</code> -> Tabelle <code>Categories</code> Property wird nach Name übernommen
By Attributes	Attribut auf Klasse / Property definiert: <code>[Table("Categories")]</code> <code>[Key, Column("Id")]</code>

```

modelBuilder.Entity<Category>()      // Fluent API
    .Property(e => e.Price)
    .HasPrecision(10, 2);
public class Category
{
    [Precision(10, 2)]           // Data Annotations
    public decimal Price { get; set; }
}

```

Indexe

Convention: Werden bei Foreign Keys automatisch erstellt.

```

modelBuilder.Entity<Category>()
    .HasIndex(e => e.Name);        // Fluent API: Non-unique Index
modelBuilder.Entity<Category>()
    .HasIndex(e => new { e.name, e.IsActive }) // Fluent API: Multi-Column Index
    .IsUnique();                     // Fluent API: Unique Index

[Index(nameof(Name), nameof(IsActive), IsUnique = true)]    // Data Annotations (Kombination)
public class Category
{
    public int Id { get; set; }
    public string Name { get; set; }
    public bool? IsActive { get; set; }
}

```

EntityType Configuration

Zur Strukturierung von FluentAPI Definitionen kann pro Entity Type jeweils eine Instanz von `IEntityTypeConfiguration<T>` verwendet werden. Somit kann die Definition zur jeweiligen Entity separat geführt werden, und im `DbContext` wird dann nur noch die entsprechende Instanz aufgerufen.

```

internal class CategoryTypeConfig : IEntityTypeConfiguration<Category> // Separate Definition
{
    public void Configure(EntityTypeBuilder<Category> builder)
    {
        builder
            .Property(p => p.TimeStamp)
            .IsRowVersion();
    }
}

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

```

Mapping-Relational(MS SQL Server)

Tabellen

Name der Tabelle zwingend, name des Schemas optional.

```

public class ShopContext : DbContext
{
    // Convention
    public DbSet<Category> Categories { get; set; }

    // Fluent API
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Category>().ToTable("Category", schema: "dbo");
    }
}

// Data Annotations
[Table("Category", Schema = "dbo")]
public class Category { public int id; ... }

```

Spalten

Mapping by Convention: Spaltenname = Property-Name

```

public class ShopContext : DbContext
{
    // Fluent API
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Category>()
            .Property(e => e.Name)
            .HasColumnName("CategoryName", order: 1);
    }
}

// Data Annotations
public class Category
{
    public int id { get; set; }
}

```

```

    [Column("CategoryName", Order = 1)]
    public string Name { get; set; }
}

```

Datentypen/Default Values

Mapping by Convention: Bekannte Datatype Mappings von C# zu Microsoft SQL Server werden implizit ausgeführt.

C#	Microsoft SQL Server
int	INT
string	NVARCHAR(MAX)
decimal	DECIMAL(18, 2)
float	REAL
byte[]	VARBINARY(MAX)
DateTime	DATETIME
bool	BIT
byte	TINYINT
short	SMALLINT
long	BIGINT
double	FLOAT
char / sbyte / object / etc.	No mapping

```

public class ShopContext : DbContext
{
    // Fluent API
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Category>()
            .Property(e => e.Name)
            .HasColumnName("CategoryName")
            .HasColumnType("NVARCHAR(500)")
            .HasDefaultValue("undefined");
    }

    // Data Annotations (keine Default Values möglich)
    public class Category{
        public int id { get; set; }
        [Column("CategoryName", TypeName = "NVARCHAR(500)")]
        public string Name { get; set; }
    }
}

```

Relationships (are complex)

Auch Assoziation genannt. Wird in der Datenbank mit einer Fremdschlüsselbeziehung dargestellt.

Principal Entity beinhaltet den Primärschlüssel, Eltern-Element der Beziehung. **Dependent Entity** beinhaltet den Fremdschlüssel, Kind-Element der Beziehung.

Navigation Property besteht nur im Code, ist eine Referenz auf das andere Objekt zusätzlich zum Key. Kann eine Referenz (Kardinalität 1) oder eine Collection (Kardinalität n) sein.

Conventions Eine Relationship wird erstellt wenn ein Navigation Property auf einem Typ vorhanden ist. Das bedeutet immer dann, wenn ein Property nicht einen "Scalar Type" auf der Datenbank hat, also ein Grundtyp der direkt in ein Feld umgewandelt werden kann.

```

public class Category // Beispiel: Fully Defined Relationship in 3 Teilen
{
    public int Id { get; set; }
    public ICollection<Products> Products { get; set; } // 1/3: Navigation Property
                                                               // mit n-Kardinalität
}

public class Product
{
    public int Id { get; set; }
    public int CategoryId { get; set; } // 2/3: Foreign Key
    public Category Category { get; set; } // 3/3: Navigation Property mit 1-Kardinalität
}

```

Fluent API

```

// Fully Defined
modelBuilder.Entity<Post>()
    .HasOne(p => p.Blog)          // Navigation Property auf dieser Entity
    .WithMany(b => b.Posts)        // Inverse Navigation auf gegenüberliegender Entity
    .HasForeignKey(p => p.BlogId) // Foreign Key explizit gesetzt
    .HasConstraintName("FK_Post_BlogId");

```

Data Annotations

```

public class Category
{
    public int Id { get; set; }
    public ICollection<Product> Products { get; set; } // Navigation Property
                                                       // mit n-Kardinalität
}

public class Product
{
    public int Id { get; set; }
    public int CategoryId { get; set; }
    [ForeignKey(nameof(CategoryId))]
    public Category Category { get; set; } // Referenz auf Foreign Key
                                         // Navigation Property mit 1-Kardinalität
}

```

Mehrere Relationships auf dieselbe Zieltabelle

```

public class Post
{ ... }
public class User
{
    public string UserID { get; set; }
    ...
    [InverseProperty("Author")]
    public List<Post> AuthoredPosts { get; set; }
    [InverseProperty("Contributor")]
    public List<Post> ContributedPosts { get; set; }
}

```

Many-to-Many Relationships beinhalten auf beiden Seiten ein Collection Navigation Property, wenn sie bei Convention erkannt werden sollen. Nur mit FluentAPI möglich.

Wenn im FluentAPI .UsingEntity(...) angegeben wird, wird in der Datenbank eine zusätzliche Tabelle mit einem sogenannten **Join Entity Type** erstellt, mit Foreign Keys zu beiden Entities des Relationships. Diese Entity dient zum Halten von Informationen zur Verbindung. Konfiguration zur Join Entity kann via FluentAPI gemacht werden

```

modelBuilder
    .Entity<Post>()
    .HasMany(p => p.Tags)
    .WithMany(p => p.Posts)
    .UsingEntity(j => j.ToTable("PostTags")); // Join Entity Types

```

Shadow Foreign Key: Wenn der Foreign Key nicht explizit definiert ist, wird er automatisch erstellt.

Single Navigation Property: Ein Navigation Property reicht aus, um eine Relationship by Convention zu definieren, solange es eindeutig ist.

```

modelBuilder.Entity<Product>()
    .HasOne<Category>()
    .WithMany(b => b.Products);

```

On Delete Cascade ist standardmäßig Cascade Delete bei required Relationships und ClientSetNull für optionale Relationships. ClientSetNull versucht nur, aktuell im Memory geladene dependent entities zu aktualisieren, alle anderen werden nicht verändert.

In FluentAPI kann das DeleteBehavior angepasst werden:

```

modelBuilder.Entity<Product>()
    .HasOne(...).WithMany(...)
    .onDelete(DeleteBehavior.Cascade|SetNull|Restrict);

```

ComputedColumns

Möglich via FluentAPI:

```

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

```

Datenbankoperationen

Database-Context

Das Objekt `DbContext` ist vorgesehen, um jeweils eine *Unit of Work* auszuführen. Die Lifetime des Objekts ist damit jeweils sehr kurz. Änderungen auf Entities werden nachverfolgt und erst mit dem Aufruf von `context.SaveChanges()` oder `context.SaveChangesAsync()` in die Datenbank geschrieben.

Konzeptionell werden die beiden *Design Patterns Repository* und *Unit of Work* umgesetzt.

Funktionalität **zur Design Time:** Model Definieren, Konfigurationen, Database Migrations,

Funktionalität **zur Runtime:** Connections verwalten (Pooling), CRUD Operationen ausführen, Change Tracking, Caching, Transaction Management

Do's:

- innerhalb einem `using`-Statement verwenden
- bei Web Applikationen eine Instanz pro Request
- im GUI eine Instanz pro Formular

Don'ts:

- Instanz zu lange leben lassen (Limitierte Anzahl Connections im Client Connection Pool, Change Tracking hat Grenzen in der Effizienz)
- Instanz sharen (ist nicht Thread-Safe, Exception an einem Ort kann gesamte Instanz unbrauchbar machen)

```

using (ShopContext context = new()) // Context Instanziieren, Change Tracker initialisieren
{
    Category category = await context // Abfragen mit 'LINQ to Entities'
        .Categories
        .SingleAsync(c => c.Id == 1);
    category.Name = $"{category.Name} / Changed"; // Daten anpassen
    await context.SaveChangesAsync(); // Sync zurück in DB

    var categories = context.Categories;
    foreach (Category c in categories) { Console.WriteLine(c.Name); } // Deferred Abfrage
} // Context schliessen: Cache invalidieren, DB-Verbindung zurück in Connection Pool

```

LINQ To Entities

Entity Framework generiert die Queries, Datenbank führt diese aus. LINQ to Entities Queries werden zu Expression Trees kompiliert. Nicht alle .NET Expressions können auf Datenbank-Syntax übersetzt werden. Der generierte SQL Output ist je nach Formulierung unterschiedlich, nicht immer optimal.

Insert

```

using ShopContext context = new();
Category cat = new() { Name = "Notebooks" };
// 3 Alternative Varianten um die neue Kategorie hinzuzufügen
context.Add(cat);
context.Categories.Add(cat);
context.Entry(cat).State = EntityState.Added;
await context.SaveChangesAsync(); // Führt SQL aus

```

Update

```

using ShopContext context = new();
Category cat = await context
    .Categories
    .FirstAsync();
cat.Name = "Changed";
await context.SaveChangesAsync(); // Change im Memory
// Führt SQL aus

```

Delete

```

using ShopContext context = new();
Category cat = await context
    .Categories
    .FirstAsync(c => c.Name == "Notebooks");
// 3 Alternative Varianten zum Löschen
context.Remove(cat);
context.Categories.Remove(cat);
context.Entry(cat).State = EntityState.Deleted;
await context.SaveChangesAsync(); // Führt SQL aus

```

Multiple Operations

```

using ShopContext context = new();
Category cat1 = new { Name = "Notebooks" };
context.Add(cat1);

Category cat2 = new { Name = "Displays" };
context.Add(cat2);

Category cat3 = await context
    .Categories
    .SingleAsync(c => c.Id == 1);
cat3.Name = "Changed";
await context.SaveChangesAsync(); // Führt SQL aus

```

CUD von Assoziationen

```

order.Customer = customer; // Anpassen von Navigation Properties
customer1.Orders.Add(order); // Hinzufügen / Entfernen von Elementen in Collection Navigation Properties
customer2.Orders.Remove(order);
order.CustomerId = 1; // Setzen des Foreign Keys (einige Varianten, die keine weiteren DB-Zugriffe macht)

```

Context - Change Tracking

Change Tracker registriert alle Änderungen an Entities die aus der DB geladen werden. Agiert komplett offline, keine Live-Checks auf der Datenbank. Die möglichen States sind *Added*, *Deleted*, *Modified*, *Unchanged*.

`context.Add(cat); context.Categories.Add(cat);` berücksichtigen ganzen Objekt-Graphen, `context.Entry(cat).State = EntityState.Added;` nur die jeweilige Entity.

State Entries: beinhaltet Informationen über

- Status des Objekt
- Info pro Property: Geändert (boolean), Originalwert, Aktueller Wert
- Entity-spezifische Ladefunktionen
- "Load Referenced Entities"

Optimistic Concurrency

Basiert auf der Annahme, dass ein Objekt zwischen Laden und Speichern eines Records nicht anderweitig verändert wurde. Dies wird vor dem Speichern eines Datensatzes geprüft.

Alternative Pessimistic Concurrency: Datensatz wird für die Dauer der Verarbeitung gesperrt. Probleme dabei sind Deadlocks, Orphaned Locks und schlechte Performance.

Die Erkennung von Konflikten beim Speichern kann mit zwei Varianten erfolgen, Timestamp oder ConcurrencyToken. Beide Varianten laden beim Lesen des Datensatzes weitere Informationen mit, die beim Schreiben zum Vergleich wieder verwendet werden können.

Timestamp oder Row Version

SQL versucht, eine Änderung in die Datenbank zu schreiben, und prüft in der Where Clause auf den Timestamp. `@@ROWCOUNT` gibt an, wie viele Datensätze vom letzten SQL Statement verändert wurden. Wenn der Wert 0 ist, weiss Entity Framework, dass ein Concurrency Fehler passiert ist und wirft eine entsprechende Exception.

```

UPDATE [categories] SET [name] = @p0
WHERE [Id] = @p1 AND [Timestamp] = @p2; /* Timestamp oder Versions-Check */
SELECT [Timestamp] FROM [Categories]
WHERE @@ROWCOUNT == 1 AND [Id] = @p1

```

Das Feld, welches im Code als Timestamp oder RowVersion definiert wird, muss ein automatisches Inkrementieren bei jeder Änderung unterstützen.

```

modelBuilder.[...]
    .Property(p => p.Timestamp)
    .IsRowVersion();           // Fluent API
[Timestamp]                  // Data Annotation
public byte[] Timestamp { get; set; }

```

ConcurrencyToken

Ähnlich wie beim Timestamp werden gewisse Spalten definiert, deren Wert vor dem Speichern geprüft wird.

```

modelBuilder.[...]
    .Property(p => p.Name)
    .IsConcurrencyToken();      // Fluent API
[ConcurrencyCheck]          // Data Annotations
public string Name { get; set; }

```

Sind diese noch gleich wie zum Zeitpunkt des Lesens, wird das Update gemacht. Andernfalls wird von Entity Framework genauso im SQL via ROWCOUNT auf Erfolg oder Fehler geprüft.

```

UPDATE [Products] SET [NAME] = @p0
WHERE [Id] = @p1
AND [Name] = @p2 AND [Price] IS NULL; /* Concurrency Token Check */

```

Varianten zur Konfliktbehandlung

Exception beinhaltet sämtliche wichtigen Informationen: Aktuelle Werte, Original-Werte, Datenbank-Werte.

- Exception fangen, beginne von vorne
- Benutzer Fragen, Vergleich von allen Werten
- Versuch einer Autokorrektur
- Blind überschreiben (nogo!)

```

try { /* ... */ }
catch (DbUpdateConcurrencyException ex)
{
    foreach (EntityEntry entry in ex.Entries)
    {
        if (entry.Entity is Product)
        {
            var proposedValues = entry.CurrentValues;
            var databaseValues = await entry.GetDatabaseValuesAsync();
            foreach (IProperty property in proposedValues.Properties)
            {
                object proposedValue = proposedValues[property];
                object databaseValue = databaseValues[property];
                // TODO: decide which value should be written to table
                // proposedValues[property] = <value to be saved>;
            }
            // Refresh original values to bypass next concurrency check
            entry.OriginalValues.SetValues(databaseValues);
        }
        else { /* ... */ }
    }
}

```

Ladestrategien

Eager Loading (Default)

Assoziationen werden nicht direkt geladen, Include()-Statement nötig. Generiert ein JOIN Statement in derselben SQL Abfrage. Queries werden schnell kompliziert.

```

Order order = await context
    .Orders
    .Include(o => o.Customer) // eager loading
    .Include(o => o.Items)
    .ThenInclude(oi => oi.Product) // cascaded eager loading
    .FirstAsync();
var customer = order
    .Customer; // customer is not "null"
var items = order
    .Items; // items is not "null"

```

Explicit Loading

Assoziationen werden explizit nachgeladen, Collections werden komplett geladen. Pro nachladen separate SQL Abfrage.

```

Order order = await context
    .Orders
    .FirstAsync();                                // << DB Abfrage
await context
    .Entry(order)
    .Reference(o => o.Customer);               // Parents
    .LoadAsync();                                // << DB Abfrage
await context
    .Entry(order)
    .Collection(o => o.Items);                 // Collections
    .Query()
    .Where(oi => oi.QuantityOrdered > 1) // Filter (inkl. .Query() voraus)
    .LoadAsync();

```

Lazy Loading

Assoziationen werden bei Zugriff auf Property automatisch nachgeladen. Collections komplett aber mit Filtermöglichkeit. Separate Abfrage. **Problem:** Logik für Nachladen muss auf dem Property im get implementiert sein. Dies entweder manuell mittels LazyLoader (Dependency Injection) oder automatisch mit Microsoft.EntityFrameworkCore.Proxies .

```

Order lazyOrder;
using (ShopContext context = new())
{
    lazyOrder = await context
        .Orders
        .FirstAsync();
    var customer = lazyOrder.Customer; // << DB Abfrage
    var items = lazyOrder.Items;     // << DB Abfrage
    var items2 = lazyOrder.Items;
}
var fail = lazyOrder.SomeProperty; // Nachladen nur mit aktivem DbContext möglich

```

Implementation mit Proxies:

```

public class ShopContext : DbContext // zwingend public, NICHT sealed
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseLazyLoadingProxies();
    }
    public class Order // zwingend public, NICHT sealed
    {
        public int Id { get; set; }
        public virtual Customer Customer { get; set; }
        public virtual ICollection<OrderItem> Items { get; set; }
    }
}

```

Anhang

LocalDBConnection

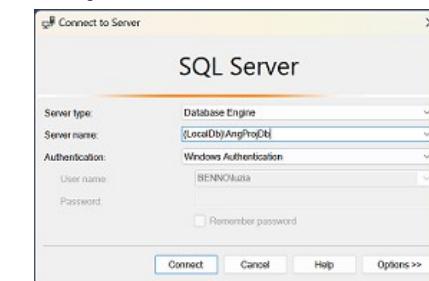
SqlLocalDB.exe create AngProjDb , und im File AppSettings.json den Connection String hinterlegen:

```

{
    "ConnectionStrings": {
        "AngProjDatabase": "Server=(localdb)\AngProjDb;Database=AngProj;Trusted_Connection=True;"
    }
}

```

Im SQL Server Management Studio kann mit folgendem Servernamen auf die Datenbank verbunden werden: (LocalDb)\AngProjDb



OR Mapping Advanced, Migrations

Verschiedene Punkte können in relationalen Datenbanken nicht direkt 1:1 umgesetzt werden wie im Code.

Vererbung

Vorteile: Constraints können über Vererbung einfach abgebildet werden. Bsp: Feld x darf für einen abgeleiteten Typen nicht gesetzt sein, beim anderen ist es zwingend.

Potenzielle Probleme führen dazu, dass diese Varianten oftmals gar nicht angewendet werden:

- Ungenutzte Felder / Speicherplatz
- Zu viele Tabellen
- Inkonsistenzen

Verschiedene Ansätze existieren zur Modellierung von Vererbung in einer Datenbank:

Table per Hierarchy (EF Core Standard)

Nur über DB Context definierbar. Abgeleitete Klassen werden als separate DbSet-Properties der Basisklasse definiert, oder via Model Builder separat:

```
modelBuilder.Entity<Product>
    .HasBaseType<Product>();
```

In der Datenbank wird ein Feld als Diskriminator eingefügt, standardmäßig der Name der Klasse. Dies kann bei Refactorings problematisch sein. Deshalb kann der Diskriminator auch übersteuert werden, dann muss jeder instanzierbare Typ definiert sein. Die Basisklasse kann abstrakt definiert werden.

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

Table per Type

Vorteil: keine NULL-Felder, Überblick über alle Produkte Nachteil: immer Left Joins nötig, wenn alle Informationen benötigt werden.

Tabelle	Felder
Products	Id, Name, Description, Price
MobilePhones	Id (FK), OperatingSystem, SupportsUmts
Tablets	Id (FK), HasKeyboard

Table per Concrete Type

Vorteil: Einzelne Produkte sind einfach abrufbar Nachteil: Gesamtüberblick über alle Produkte ist schwieriger

Tabelle	Felder
Products	Id, Name, Description, Price
MobilePhones	Id, Name, Description, Price, OperatingSystem, SupportsUmts
Tablets	Id, Name, Description, Price, HasKeyboard

Entity/Table Splitting

Mittels Table Splitting können unter anderem mehrere Entitäten auf eine Tabelle gemappt werden. Inheritance (Table per Hierarchy) ist nur eine Variante davon.

```
xxxxxxxxxx var currentAssembly = Assembly.GetExecutingAssembly(); var classInfo = currentAssembly.GetType(); foreach (var classType in classInfo) // Auf jeder Klasse im Assembly{ if(classType.IsDefined(typeof(TableAttribute))) // Wenn das gesuchte Attribut gesetzt ist { Console.WriteLine($"\\nKlasse: {classType.Name}"); foreach (var a in classType.CustomAttributes) // für jedes Attribut { if (a.AttributeType.ToString() == "1_2_Attribute.TableAttribute") // das gesuchte ausgeben Console.WriteLine($"\\nAttribut-Argument: [{a.ConstructorArguments[0].ToString()}]"); } } }cschar
```

Zwei Entitäten auf die gleiche Tabelle mappen. Nur mit FluentAPI möglich. Performance: ermöglicht Abfrage von entweder nur Header Daten oder allen Details

```
public class ShopContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<CustomerDetail>(e => {
            e.ToTable("Customers");
            // Map all Detail-Properties
        });
        modelBuilder.Entity<Customer>(e => {
            e.ToTable("Customers");
            // Map Properties
            e.HasOne(o => o.Details)
                .WithOne()
                .HasForeignKey<CustomerDetail>(o => o.Id);
        });
    }
}
```

OwnedTypes

Owned Type hat keinen Primary Key, deshalb keine Convention möglich. Felder des Owned Property in der Tabelle erhalten einen zusammengesetzten Namen aus `OwnedType_Property`: bsp. `Address_Street`

```
public class ShopContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Customer>()
```

```
        .OwnsOne(b => b.Address); // Variante FluentAPI
    }

    public class Customer {
        public int Id { get; set; }
        public Address Address { get; set; }
    }
} // Variante Data Annotations
public class Address { ... }
```

Möglich ist auch, mehrere oder verschachtelte Owned Types zu verwenden.

```
public class Customer {
    public int Id { get; set; }
    public Address BillingAddress { get; set; }
    public Address ShippingAddress { get; set; }
}
/* Ergibt:
BillingAddress_street  (nvarchar(MAX))
BillingAddress_City   (nvarchar(MAX))
ShippingAddress_Street (nvarchar(MAX))
ShippingAddress_City  (nvarchar(MAX))
*/
```

Database Migrations

Naiver Ansatz: Änderungen der Datenbank in einem Change Script festhalten. Probleme dabei:

- keine manuelle Einflussnahme
- schwierig, sobald Daten vorhanden sind
- Umbenennen von Spalten/Tabellen wie DROP / CREATE -> möglicher Datenverlust
- Hinzufügen einer NOT NULL Spalte zwingt auch bestehende Datensätze, NOT NULL einzuhalten

EF Core Ansatz: Modell anpassen, dann eine Migration erstellen. Diese wird als C# Klasse implementiert, die anschließend reviewed und allenfalls korrigiert werden kann. Jede Migration implementiert die Funktion `up()` und `down()` und kann als Artefakt in der Versionsverwaltung abgespeichert werden. Deployment wie auch Rollback einfach auszuführen.

Erstellte Files

Migrations</timestamp>\ Migration.cs Eigentliche Migration Migrations</timestamp>_Migration.Designer.cs Metadaten für Entity Framework Migrations</contextClassName>\ModelSnapshot.cs Snapshot / Basis für nächste Migration

Tabelle

`dbo.__EFMigrationsHistory` zeigt eine Liste der aktuell auf die Datenbank angewendeten Migrations.

Befehle

CLI [dotnet ef ...]	Package Manager	Beschreibung
database drop	Drop-Database	Löscht die gesamte Datenbank
database update	Update-Database	Setzt den aktuellen Stand der Codebase in der Datenbank um
database update Init		Setzt die Datenbank auf den Stand der angegebenen Migration
migrations add Init	Add-Migration <name>	Legt eine neue Migration an zum aktuellen Stand der Codebase
migrations list		Zeigt eine Liste von verfügbaren Migrations an
migrations remove	Remove-Migration	Löscht die letzte Migration
migrations script <migration-start> <migration-end>	Script-Migration	Gibt auf der Konsole ein SQL-Skript aus, das die Änderungen zwischen Migration-Start und Migration-End ausführt. Vorwärts und Rückwärts möglich.
dbcontext info	Get-DbContext	Liefert Informationen über einen DbContext Typen
dbcontext list		Listet verfügbare DbContext Typen
dbcontext scaffold	Scaffold-DbContext	Erstellt einen DbContext und Entity Types für eine bestehende Datenbank
	Get-Help EntityFramework	Informationen über Entity Framework Commands

C# API

Die C# API bietet diverse Operationen um Migrations programmatisch anzuwenden.

```
using(AngrProjContext context = new())
{
    DatabaseFacade database = context.Database;

    // "Dev" Ansatz (löschen / neu erstellen)
    await database.EnsureDeletedAsync();
    await database.EnsureCreatedAsync();

    // Automatische Migration auf neuesten Stand
    await database.MigrateAsync();

    // Migrations abfragen
    IEnumerable<string> migrations;
    migrations = database.GetMigrations();
    migrations = database.GetPendingMigrations();
    migrations = database.GetAppliedMigrations();

    // Migration ausführen
    IMigrator m = context.GetService<IMigrator>();
    m.Migrate("<MigrationName>");
}
```

gRPC

Überblick

Alternative zu REST: ProtoBuf wird eher für Server-to-Server Kommunikation verwendet (z.B. für Microservices). Fokus dabei ist hohe Performance. Als Frontend-API bessere Alternativen sind REST oder GraphQL. ProtoBuf wird auch als Replacement für Windows Communication Foundation WCF verwendet.

Basistechnologien:

- HTTP/2
- Interface Definition Language (IDL): Google Protocol Buffers

Remote Procedure Call

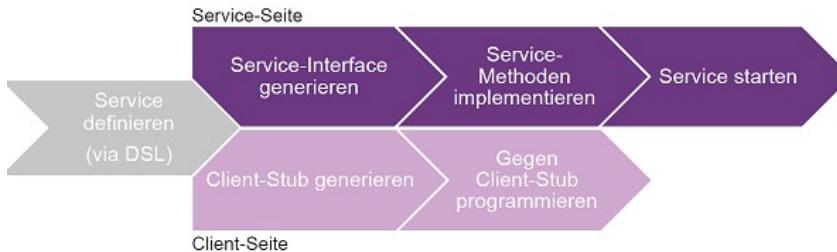
Auch RPC, ermöglicht Client-Server Kommunikation. Wird in fast allen verteilten Systemen verwendet. Einfacher TCP Call kennt das Konzept einer Methode (Procedure) nicht.

Wird in fast allen verteilten Systemen verwendet. Verschiedene Implementationen: WCF, Cobra, JSON-RPC, SOAP, NFS, Apache Thrift / Avro, D-Bus.

gRPC Grundprinzipien

- Einfache Service Definition (IDL, Sprachunabhängig)
- Problemlos skalierbar (stateless)
- Bi-direktionales Streaming
- Integrierte Authentisierungsmechanismen
- Unterstützt verschiedene Sprachen (Java, .NET, C++, Dart, Go, Kotlin, Node, Python, ...)

Developer Workflow



Architektur

gRPC ist ein Software Development Kit. Plattform- und Sprachneutral. Visual Studio (Code) Integration bietet Features wie automatische Code Generierung, Intelli-Sense sowie Debugging & Logging. Kommunikation über HTTP/2.

HTTP/2

Ermöglicht Multiplexing von HTTP Calls pro TCP/IP Connection, verringert Overhead für Disconnect/Reconnect massiv (8 Connections Limit oft vom Browser vorgegeben). Bidirektionales Streaming ermöglicht paralleles Senden und Empfangen. HTTP/2 und gRPC basieren vollständig auf HTTPS, Kommunikation ist also standardmäßig verschlüsselt. Header Compression optimiert die zu übertragenden Daten noch einmal.

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 ¹
Streaming	Client / Server / Bidirektional	Client / Server
Browser Support	Nein (benötigt grpc-web ²)	Ja
Security	Transport / HTTPS (zwingend)	Transport / HTTPS (optional)
Client Generierung	Ja	Third-party Tooling OpenAPI Specification und Client Generator (AutoRest ³ / NSwag ⁴)

Protocol Buffers (ProtoBuf) Basics

Umfang

- IDL: Subform einer Domain Specific Language. Beschreibt ein Service Interface
- Data Model: Beschreibt Messages bzw. Request- und Response Objekte
- Wire Format: Beschreibt das Binärformat zur Übertragung
- Serialisierungs- und Deserialisierungsmechanismen
- Service-Versionierung

Wertetypen

.proto	C#	Default	Beschreibung
double	double	0	
float	float	0	

.proto	C#	Default	Beschreibung
int32	int	0	Variable-length encoding. Inefficient for negative numbers (use sint32 instead)
int64	long	0	Variable-length encoding. Inefficient for negative numbers (use sint64 instead)
bool	bool	false	
string	string		String must always contain UTF-8 encoded or 7-bit ASCII text.
bytes	ByteString		May contain an arbitrary sequence of bytes.
			weitere wie sint32/64, uint32/64, fixed32/64

Proto-Files

Hat jeweils die Dateiendung .proto. Das File enthält im Header allgemeine Informationen. Der Inhalt besteht anschließend aus 0-n Services und 1 oder mehr Service-Methoden (Messages) pro Service. Die Felder eines Message Types beinhalten jeweils:

- Typ : Skalarer Wertetyp, anderer Message Type, Enumeration
- Unique Name : Lower Snake Case
- Unique Field Number : Zur Versionierung bzw. als Identifikator für das Binärformat. Wertebereich 1 - ca. 500'000, mit Ausnahme: 19'000-19'999, diese sind vom Protokoll reserviert.

```

syntax = "proto3";
option csharp_namespace = "_01_BasicExample"; // verwendet für generierten C# Code
package Greet; // Package-Name
import "protos/_base.proto" // Message Types können wiederverwendet werden

service Greeter {
    rpc SayHello (HelloRequest)
        returns (HelloReply); // Service
        // Service Methode
}

message HelloRequest {
    string name = 1;
    repeated string words = 1; // Message Type
    // Field
    // Repeated Field: Liste von Werten
}

message HelloReply {
    string message = 1;
    reserved 1, 3, 20 to 30;
    reserved "page_number"; // Reservierte Versionierung
    // Reservierte Feldnamen, für spätere Verwendung
}

enum Color {
    RED = 0;
    GREEN = 1; // Enumeration: kann entweder in einer Message oder im Root des Files
    // definiert werden. Wert 0 ist zwingend = default value.
}
  
```

Service Methoden beinhalten immer nur genau 1 Parameter und 1 Rückgabetypr. Leere Werte werden mit google.protobuf.Empty definiert (benötigt import "google/protobuf/empty.proto").

Advanced Types

Verwendung

Compiler: protoc.exe ist Bestandteil vom NuGetPackage Grpc.Tools , zusammen mit einigen Plugins zur Generierung von C# Code aus .proto -Files. Damit wird der Proto Compiler direkt von Visual Studio in die Build Chain eingefügt.

Proto-Compiler Output in <projectRoot>\obj\Debug\netcoreappX.X , nützlich bei Build Fehlern.

Einbindung im .csproj File von Server und Client Code, Einstellungen möglich im Visual Studio bei Klick auf das Proto-File, unter Properties (F4).

```

<Project Sdk="Microsoft.NET.Sdk.Web">
    <PropertyGroup>
        <TargetFramework>netcoreapp3.0</TargetFramework>
        <RootNamespace>BasicExample</RootNamespace>
    </PropertyGroup>
    <ItemGroup>
        <ProtoBuf Include=<relative-path>/greet.proto" GrpcServices="[Server|Client]">
            <Link>Protos/greet.proto</Link> <!-- Ort der Verlinkung -->
        </ProtoBuf>
    </ItemGroup>
    <ItemGroup>
        <PackageReference Include="Grpc.AspNetCore" Version="." /> <!-- Server -->
        <PackageReference Include="Google.Protobuf" Version="." /> <!-- Client -->
        <PackageReference Include="Grpc.Net.Client" Version="." /> <!-- Client -->
        <PackageReference Include="Grpc.Tools" Version="." /> <!-- Client -->
    </ItemGroup>
</Project>
  
```

Serverprojekt: Automatisch vom Typ ASP.NET Core Projekt, mit dotnet new grpc oder via Visual Studio. Packages:

Grpc.AspNetCore(.Server)

Clientprojekt: Beliebiger Typ, Proto-File als Kopie oder Link eingebunden. Packages: Grpc.Net.Client , Google.Protobuf , Grpc.Tools

Generierter Code

Pro Service wird eine abstrakte Basisklasse erzeugt:

```

public static partial class Greeter { // Gründe unklar für die äussere Klasse
    public abstract partial class GreeterBase { /* ... */ } // <-- zu verwenden!
}
  
```

Manuell zwingend: Implementieren von allen Methoden der statischen Basisklasse.

```

public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
        => Task.FromResult(new HelloReply { Message = "Hello " + request.Name } );
}

```

Server: Service muss beim Startup registriert werden:

```

WebApplicationBuilder builder = WebApplication.CreateBuilder(args); // ASP.NET spezifisch
builder.Services.AddGrpc(); // GRPC für Dependency Injection konfigurieren
WebApplication app = builder.Build();
app.MapGrpcService<GreeterService>(); // Registrieren jedes Service

```

Beispiel: Customer Service

Proto-File

```

// Customers
service CustomerService {
    rpc GetCustomers (google.protobuf.Empty)
        returns (GetCustomersResponse);
    rpc GetCustomer (GetCustomerRequest)
        returns (GetCustomerResponse);
}

message GetCustomersResponse {
    repeated CustomerResponse data = 1;
}
message GetCustomerResponse {
    CustomerResponse data = 1;
}
message GetCustomerRequest {
    int32 id_filter = 1;
    bool include_orders = 2;
}
message CustomerResponse {
    int32 id = 1;
    string first_name = 2;
    string last_name = 3;
    Gender gender = 4;
    repeated OrderResponse orders = 10;
}
enum Gender { UNKNOWN = 0; FEMALE = 1; MALE = 2; }

// Orders
service OrderService {
    rpc GetOrders (GetOrdersRequest)
        returns (GetOrdersResponse);
}
message GetOrdersRequest {
    int32 customer_id_filter = 1;
}
message GetOrdersResponse {
    repeated OrderResponse data = 1;
}
message OrderResponse {
    string product_name = 1;
    int32 quantity = 2;
    double price = 3;
}

```

CustomerService

```

public class MyCustomerService : CustomerService.CustomerServiceBase
{
    public override Task<GetCustomersResponse> GetCustomers(
        Empty request, ServerCallContext context)
    { /* ... */ }
    public override Task<GetCustomerResponse> GetCustomer(
        GetCustomerRequest request, ServerCallContext context)
    { /* ... */ }
}

```

OrderService

```

public class MyOrderService : OrderService.OrderServiceBase
{
    public override Task<GetOrdersResponse> GetOrders(
        GetOrdersRequest request, ServerCallContext context)
    { /* ... */ }
}

```

Client-Implementation

```

// The port number (5001) must match the port of the gRPC server.
GrpcChannel channel = GrpcChannel.ForAddress("https://localhost:5001");

// Customer service calls
CustomerService.CustomerServiceClient customerClient = new(channel);

```

```

Empty request1 = new();
GetCustomersResponse response1 = await customerClient.GetCustomersAsync(request1);
Console.WriteLine(response1);

GetCustomerRequest request2 = new() { IdFilter = 1 };
GetCustomerResponse response2 = await customerClient.GetCustomerAsync(request2);
Console.WriteLine(response2);

request2.IncludeOrders = true;
response2 = await customerClient.GetCustomerAsync(request2);
Console.WriteLine(response2);

// Order service calls
OrderService.OrderServiceClient orderClient = new(channel);

GetOrdersRequest request3 = new() { CustomerIdFilter = 1 };
GetOrdersResponse response3 = await orderClient.GetOrdersAsync(request3);
Console.WriteLine(response3);

```

gRPC Advanced Topics

Exception Handling

Weitergabe einer Exception vom Server auf den Client via gRPC nicht einfach so möglich: aus Sicherheitsgründen, wegen fehlender Serialisierungslogik einer Exception, wegen möglicherweise unterschiedlicher Programmiersprachen von Server und Client.

```
public class RpcException : Exception
{
    public RpcException(Status status);
    public RpcException(Status status, string message);
    public RpcException(Status status, Metadata trailers);
    public RpcException(Status status, Metadata trailers, string message);

    public Status Status { get; }
    public StatusCode StatusCode { get; }
    public Metadata Trailers { get; } // Möglichkeit, weitere Details an den Client zu übermitteln
}

public struct Status
{
    public static readonly Status DefaultSuccess = new Status(StatusCode.OK, "");
    public static readonly Status DefaultCancelled = new Status(StatusCode.Cancelled, "");
    public Status(StatusCode statusCode, string detail);
    public StatusCode StatusCode { get; }
    public string Detail { get; }
    public override string ToString();
}
```

Standard-Mechanismus auf Übertragungskanal: Nutzung von **HTTP Status Codes**. Convenience Features seitens .NET: **RpcException**. Jede mögliche Exception innerhalb des Servers sollte in eine RpcException verwandelt werden. Somit kann die .NET gRPC Implementation diese in eine saubere HTTP Response verwandeln.

Status Code	Beschreibung
OK	Kein Fehler, alles okay.
Cancelled	Operation wurde abgebrochen (meist durch Client).
Unknown	Fehler kann nicht ermittelt werden / ist unbekannt. Default wenn Exception nicht behandelt wird.
InvalidArgument	Ungültige Argumente beim Aufruf angegeben.
DeadlineExceeded	Deadline überschritten bevor Anfrage erfolgreich beendet wurde.
NotFound	Angefragte Ressource wurde nicht gefunden.
AlreadyExists	Zu erstellende Ressource existiert bereits.
PermissionDenied	Aufrufer hat keine Berechtigung um die Operation auszuführen.
Unauthenticated	Aufrufer ist nicht authentifiziert (angemeldet).
ResourceExhausted	Eine Ressource ist aufgebraucht (Per-User-Quota / Speicherplatz / etc.)
FailedPrecondition	Vorbedingungen sind falsch (ungültiger Systemstatus, keine Datenbankverbindung, Bestellung abschliessen obwohl bereits abgeschlossen).
Aborted	Anfrage wurde abgebrochen (Concurrency Issues, Transaktionsabbruch, etc.).
OutOfRange	Ungültiger Range bei Anfrage (Geburtsdatum in der Zukunft, etc.).
Unimplemented	Methode wurde (noch) nicht implementiert.
Internal	Allgemeiner interner Serverfehler
Unavailable	Service ist aktuell nicht verfügbar.
DataLoss	Datenverlust oder korrupte Daten

Wird von der Server Runtime eine nicht-RPC-Exception gefangen, wird diese allgemein in eine StatusCode Unknown RPC-Exception umgewandelt mit leerem Trailer. Eine RPC-Exception wird jeweils direkt übertragen.

```
public override Task<Empty> NotFound(Empty request, ServerCallContext context)
{
    throw new RpcException(
        new Status(
            StatusCode.NotFound, "Something was not found.")
        //Optional: Trailers mitgeben
        new Metadata
        {
            { "error-details", "Here are some more Details" },
            { "error-obj" + Metadata.BinaryHeaderSuffix, Encoding.UTF8.GetBytes("..payload..") }
            // Metadata.BinaryHeaderSuffix fügt dem Namen "-bin" hinzu für binary-Data
        }
    );
}
```

Common Mistakes

Unimplemented kann heissen:

- Service in Startup nicht registriert mit `endpoints.MapGrpcService<...>();`
- Methode in Proto File definiert aber nicht implementiert (überschrieben).

Unknown kann auf einen fehlenden Catch-Block für aufgetretene Fehler hindeuten.

Clientseite

Exceptions können mit when-Klauseln abgefangen werden:

```
try { /* gRPC call(s) */ }
// Communication exceptions
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) {}
// Other stuff
catch (Exception) { /* ... */ }
```

Well Known Types

Typ	Beschreibung
Empty	import "google/protobuf/empty.proto"
Timestamp	import "google/protobuf/timestamp.proto", immer auf UTC ohne daylight saving bezogen.
ByteString	Typ bytes ohne Import in proto Files verwendbar. in C#: var bs = ByteString.CopyFrom("X", Encoding.Unicode);

Repeated Fields

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

RepeatedResponse response = new();
response.results.Add("Hello");
string[] arr = { "A", "B" }
response.results.AddRange(arr);
```

Map Fields

```
message MapResponse { map<int32, string> results = 1 }

MapResponse response = new();
response.results.Add(1, "Hello"); // Single Add
// Verwendung wie ein IDictionary
bool exists = response.results.ContainsKey(1);
string result= response.results[1];

OneOf
Lässt eine Auswahl von Typen zu, liefert aber immer nur einen davon zurück. Logik in generierter Klasse implementiert, dass immer nur einer der beiden Werte gesetzt sein kann.

message OneofResponse {
    oneof results {
        string imageUrl = 1;
        bytes imageData = 2;
    }
}

OneofResponse response = new() { imageUrl = "http://..." };
var rc = response.ResultsCase; // imageUrl -> wird in C# als Enum implementiert
string s = response.ImageUrl; // "http://..."
ByteString bs = response.ImageData; // default
```

Any

```
import "google/protobuf/any.proto"
message AnyResponse { google.protobuf.Any results = 1; }

AnyResponse response = new(); // Response Objekt
response.Results = Any.Pack(new CustomerResponse()); // Objekt befüllen

bool isCust = response.Results.Is(CustomerResponse.Descriptor);
bool success = response.Results.TryUnpack(out CustomerResponse parsed); // safe unpack
response.Results.Unpack<CustomerResponse>(); // unsafe unpack
```

Configuration und Logging

Serverseitig

Option	Default	Beschreibung
MaxSendMessageSize	null	Maximale Message-Größe beim Senden (Exception falls grösser).
MaxReceiveMessageSize	4 MB	Maximale Message-Größe beim Empfangen (Exception falls grösser).
EnableDetailedErrors	false	Wenn true werden Details zu Exceptions an den Client übermittelt. Achtung: Möglicher Security Flaw
CompressionProviders	gzip	Kompressions-Algorithmus

```

.Services
    .AddGrpc(options => // Global
    {
        options.MaxSendMessageSize = 5 * 1024 * 1024; // 5 MB
        options.MaxReceiveMessageSize = 2 * 1024 * 1024; // 2 MB
        options.EnableDetailedErrors = true;
        options.CompressionProviders = new List<ICompressionProvider> { new
GzipCompressionProvider(CompressionLevel.Optimal) };
    })
.AddServiceOptions<AdvancedService>(options => // Specific Service
{
    options.MaxSendMessageSize = 1 * 1024 * 1024; // 1 MB
});

```

Clientseitig

Option	Default	Beschreibung
HttpClient	Objekt	HttpClient für gRPC Verbindung. Kann verändert werden z.B. um weitere HTTP Headers zu schicken.
DisposeHttpClient	false	Wenn true und Verwendung eines Custom HttpClient (siehe oben) wird der Client mit dem gRPC Channel disposed
LoggerFactory	null	Logger Factory für gRPC Calls
MaxSendMessageSize	null	Maximale Message-Grösse beim Senden (Exception falls grösser).
MaxReceiveMessageSize	4MB	Maximale Message-Grösse beim Empfangen (Exception falls grösser).
Credentials	null	Credentials für die Authentifizierung am Server
CompressionProviders	gzip	Kompressions-Algorithmus

```

GrpcChannel channel = GrpcChannel.ForAddress("https://localhost:5001",
new GrpcChannelOptions
{
    MaxSendMessageSize = 2 * 1024 * 1024, // 2 MB
    MaxReceiveMessageSize = 5 * 1024 * 1024, // 5 MB
    // ...
});

```

Logging

via appsettings.json

```

{
    "Logging": {
        "LogLevel": {
            "Default": "Warning",
            "Microsoft.Hosting.Lifetime": "Information",
            "Grpc": "Debug"
        }
    },
    "AllowedHosts": "*",
    "Kestrel": {
        "EndpointDefaults": {
            "Protocols": "Http2"
        }
    }
}

```

via Program.cs

```

builder.Logging.AddFilter("Grpc", LogLevel.Debug);

```

gRPC-Streams

Performance-Thema in gRPC. 3 verschiedene Modi: Server > Client, Client > Server, Bidirektional.

End-To-End Reliability, garantiertes Ausliefern der Nachrichten Ordered Delivery: Reihenfolge gewährleistet

Anwendungen: Messaging, Games, Live-Resultate, Smart Home Devices etc.

Synchrones Lesen vs. Streaming

ProtobufImplementation

```
service FileStreamingService {
    rpc Readfiles (google.protobuf.Empty)          // Server Streaming Call
        returns (stream FileDto);
    rpc Sendfiles (stream FileDto)                 // Client Streaming Call
        returns (google.protobuf.Empty);
    rpc Roundtripfiles (stream FileDto)           // Bidirektional
        returns (stream FileDto);
}

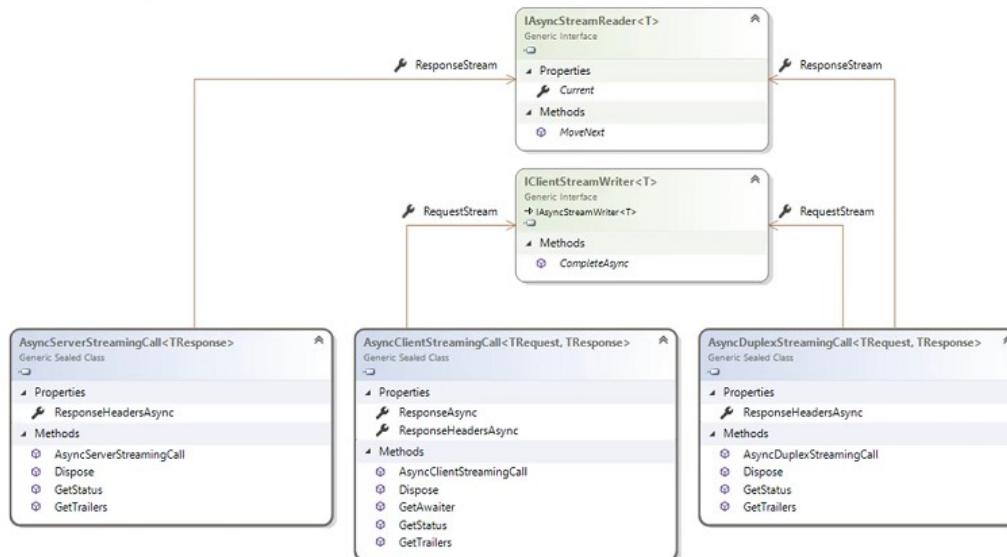
message FileDto {
    string file_name = 1;
    int32 line = 2;
    string content = 3;
}
```

C# Implementation

API Klassen

AsyncServerStreamingCall<TResponse> AsyncClientStreamingCall<TRequest, TResponse> AsyncDuplexStreamingCall<TRequest, TResponse>

Interfaces: IAsyncStreamReader<T>, IClientStreamWriter<T>



Server Streaming Call

```
// Service
public override async Task ReadFiles(
    Empty request,
    IServerStreamWriter<FileDto> responseStream,
    ServerCallContext context)
{
    string [] files = Directory.GetFiles(@"...");
    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 responseStream.WriteAsync(reply);
        }
    }
}
```

```
        }
    }
}

// Client
using AsyncServerStreamingCall<FileDto> call = client.ReadFiles(new Empty());
await foreach (FileDto message in call.ResponseStream.ReadAllAsync())
{
    WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Content: {message.Content}");
}
```

Client Streaming Call

```
// Service
public override async Task<Empty> SendFiles(IAsyncStreamReader<FileDto> requestStream, ServerCallContext context)
{
    await foreach (FileDto message in requestStream.ReadAllAsync())
    {
        WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Line Content: {message.Content}");
    }
}

// Client
using AsyncClientStreamingCall<FileDto, Empty> call = client.SendFiles();
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)
Empty result = await call; // Wait until service method is terminated / Get the result
```

Bidirectional/Duplex Streaming Call

```
// Service
public override async Task RoundtripFiles(
    IAsyncStreamReader<FileDto> requestStream,
    IClientStreamWriter<FileDto> responseStream,
    ServerCallContext context)
{
    await foreach (FileDto message in requestStream.ReadAllAsync())
    {
        await responseStream.WriteAsync(message);
        WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Line Content: {message.Content}");
    }
}

// Client
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}, 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);
    }
}

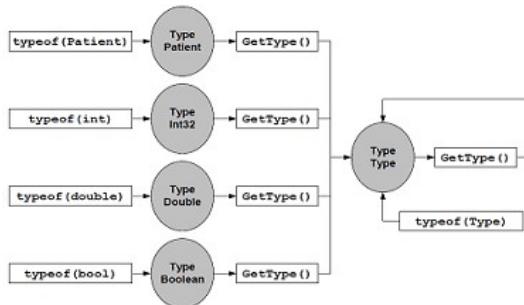
// Required!
await call.RequestStream.CompleteAsync(); // No more messages to come (server exits foreach-Loop)
await readTask; // Wait until service method is terminated / all messages are received by client
```

Reflection

Anwendung: Dynamische Programmierung

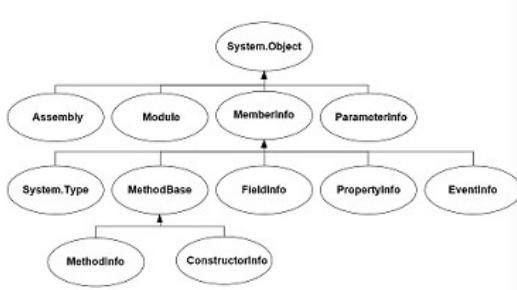
- Metadaten darstellen: Intellisense, Object Browser, etc0.
- Type Discovery: Suchen und Instanziieren von Typen, Zugriff auf Dynamische Datenstrukturen (bsp. JavaScript-Objekte)
- Late Binding von Methoden oder Properties: Suche nach Methoden via String, Ausführen "on the fly"
- Reflection Emit: Erstellen von Typen inkl. Members zur Laufzeit. Nicht in .NET Core portiert worden.

Basics



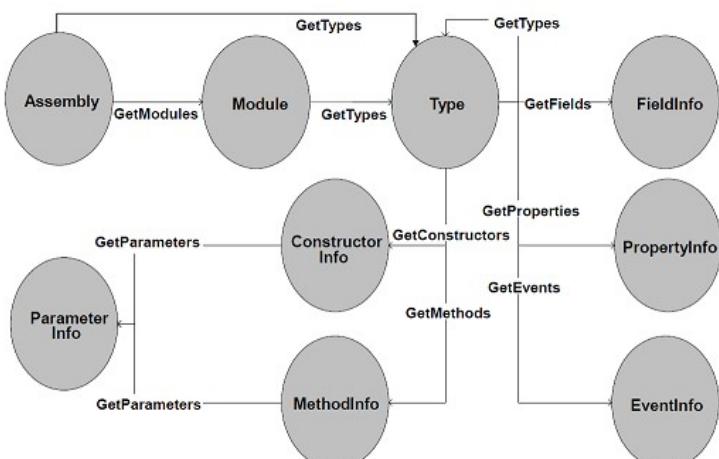
Alle Typen in C# sind selbstbeschreibend. Erfolgt über eine Instanz der Klasse `System.Type`, die z.B. die Klasse `String` beschreibt. Kann programmatisch abgefragt werden nach Visibility, Properties, alle weiteren möglichen Schlüsselwörter. Meist wird mit der Ableitung `System.RuntimeType` gearbeitet.

Vererbungshierarchie: Befindet sich im Assembly `mscorlib`, Namespace `System.Reflection` wenn nicht anders angegeben. `MemberInfo` ist abstrakte Basisklasse die gemeinsame Informationen der abgeleiteten Typen zusammenfasst. Genauso `MethodBase`.



`System.Type` beschreibt alle möglichen Eigenschaften eines Typen, unter Anderem:

- `FullName`
- `IsAbstract`
- Klasse oder Struct
- `GetFields`: Liste von Feldern / Klassenvariablen



Objekttyp `BindingFlags` ist als Bitmask implementiert und können mittels Oder-Verknüpfung kombiniert werden. Dienen zum Filtern von

verschiedenen Infos.

TypeDiscovery

`BindingFlags`: `Public/NonPublic`, `Static/Instance`, `DeclaredOnly`. Defaultmäßig werden nur Public Information ausgegeben.

```
Type[] t01 = assembly.GetTypes();
foreach (Type type in t01)
{
    Console.WriteLine(type);
    // Alle Arten von Members
    MemberInfo[] mInfos = type.GetMembers();
    // dynamische Abfrage von Members
    BindingFlags bf = BindingFlags.Public | BindingFlags.Static;
    MemberInfo[] mInfos = type.FindMembers(MemberTypes.Method, bf, Type.FilterName, "Get*");
    // Ausgabe
    foreach (memberInfo mi in mInfos)
        { Console.WriteLine("\t{0}\t{1}", mi.MemberType, mi); }
}
// 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()
//     ...
// ...
```

FieldInfo

Beschreibt ein Feld auf seiner Klasse mit Name, Typ, Sichtbarkeit etc. Ermöglicht direkten Lese- / Schreibzugriff an API vorbei. Casts zum jeweiligen Typen nötig.

```
Fieldinfo[] fiAll = type.GetFields( BindingFlags.NonPublic );
int value = (int)fi.GetValue(object obj); // Instanz des Objekts
fi.SetValue(object obj, object value); // Instanz des Objekts und neuer Wert
```

PropertyInfo

(Lese- / Schreibzugriff prüfen mit `CanRead` / `CanWrite`)

```
PropertyInfo[] piAll = type.GetProperties(); // Alle Properties
PropertyInfo pi = type.GetProperty("SomeProperty"); // Spezifisches Property
// Permissions prüfen mit pi.CanRead und pi.CanWrite
```

MethodInfo/ConstructorInfo

Methoden können aufgelistet, nach Namen oder auch nach Parameter-Typen gefunden werden:

```
MethodInfo[] miAll = type.GetMethods();
MethodInfo mi = type.GetMethod("Increment");
// Abfrage mit Parametern
Type[] paramtypes = {typeof(int), typeof(string)};
MethodInfo miAbs = type.GetMethods("name", paramTypes);
// Aufruf mit Parametern
object[] @params = { -1, "test" };
miAbs.Invoke(type, @params);

// Konstruktoren
ConstructorInfo[] ciAll = type.GetConstructors();
ConstructorInfo ci01 = type.GetConstructor(new[] { typeof(int) }); // Type Array paramTypes on the fly definiert
// Aufruf
Counter c01 = (Counter)ci01.Invoke(new object[] { 12 }); // Parameter array on the fly definiert

// Konstruktor ausführen via Activator
Counter c02 = (Counter)Activator.CreateInstance(
    typeof(Counter), 12 // mehr parameter mit komma getrennt..
);
// oder bei public default constructors
Counter c03 = Activator.CreateInstance<Counter>();


```

BindingFlags

`Public / NonPublic`, `Static / Instance`, `DeclaredOnly` (nicht vererbt)
Deaktivierung von Reflection-Zugriffen => Code Access Security CAS

Weitere Beispiele

Assemblies

```
FileInfo fi = new "../../../../Tiere.dll";
Assembly tiere = Assembly.LoadFile(fi.FullName);
Assembly specificAssem = Assembly.Load("mscorlib");
Assembly assemFromType = t.Assembly;
Assembly currentAssem = Assembly.GetExecutingAssembly(); // Aktuell ausgeführtes Assembly
```

// Laden von dll-File via Pfad

// Laden von dll via Name

// Assembly eines spezifischen Typs

// Aktuell ausgeführtes Assembly

Objekte Instanziieren und Methoden ausführen

```
FileInfo fi = new "../../../../Tiere.dll";
Assembly tiere = Assembly.LoadFile(fi.FullName);
```

```

Type katzeTyp = tiere.GetType("Tiere.Katze");
ConstructorInfo[] ciAll = katzeTyp.GetConstructors();

ConstructorInfo c01 = ciAll[0];
Console.WriteLine(c01); // Void .ctor(System.String)

object mina = c01.Invoke(new[] {"Mina"});xi

MethodInfo[] miAll = katzeTyp.GetMethods();
Type[] paramTypes = { };
MethodInfo miMausFangen = katzeTyp.GetMethod("MausFangen", paramTypes);

for (int i = 0; i < 5; i++)
{
    miMausFangen.Invoke(mina, null);
    Console.WriteLine(mina); // Die Katze Mina hat erst eine Maus gefangen. ...
}

```

Attributesabfragen

ICustomAttributeProvider wird instantiiert von Assembly/Module, Type, MethodInfo, ParameterInfo

```

var currentAssembly = Assembly.GetExecutingAssembly();
var classInfo = currentAssembly.GetTypes();

foreach (var classType in classInfo) // Auf jeder Klasse im Assembly
{
    if (classType.IsDefined(typeof(TableAttribute)))
        // Wenn das gesuchte Attribut gesetzt ist
    {
        Console.WriteLine($"\\nKlasse: {classType.Name}");
        foreach (var a in classType.CustomAttributes) // für jedes Attribut
        {
            if (a.AttributeType.ToString() == "_1._2_Attribute.TableAttribute")
                // das gesuchte anzeigen
                Console.WriteLine($"Attribute-Argument1: {a.ConstructorArguments[0].ToString()}");
        }
    }
}

```

Attributes

Jede Klasse die von Attribute erbt. Eigenes Attribut AttributeUsage auf der neuen Klasse definiert die Verwendung. Klassename hat Suffix -Attribute, in der Verwendung aber nicht mehr. Parameter entweder Positional oder Named.

```

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Field, AllowMultiple=true)]
public class TableAttribute : Attribute
{
    public TableAttribute(string tableName)
    {
        TableName = tableName;
    }
    public string TableName { get; set; }
}
[Table("tblFilme")]
public class Filme { ... }

```

Beispiel CSV-Mapper: Auf jedem Property einer Klasse kann angegeben werden, wie der Name der Spalte im CSV lauten soll und ob auf dem Text eine Transformation (upper- oder lowercase) angewendet werden soll.

```

// Verwendung
public class Address
{
    [CsvName("Name"), Uppercase]
    public string Name { get; set; }

    [CsvName("Strasse"), Lowercase]
    public string Street { get; set; }
}
// Definition
public class CsvNameAttribute : Attribute
{
    public string Name { get; set; }
    public CsvNameAttribute(string name) { Name = name; }
}
public interface IStringFilter { string Filter (string arg); }
public class UppercaseAttribute : Attribute, IStringFilter
{
    public string Filter (string arg) { return arg.ToUpper(); }
}

```