.NET Technologien

HS2021

Marco Agostini

Computer Science
University of Applied Sciences of Eastern Switzerland
September 2021

Contents

1 Introduction

1.1 .NET Framework

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

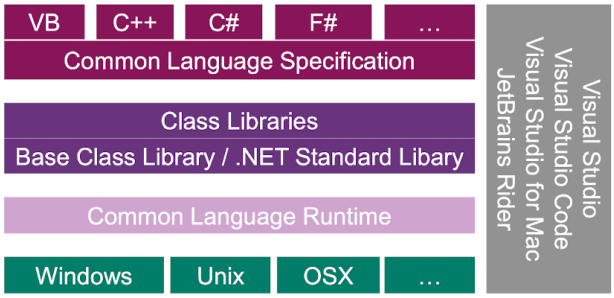


Figure 1: .NET Framework Architektur

1.2 CLR: Common Language Runtime

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

1.3 CTS: Common Type System

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

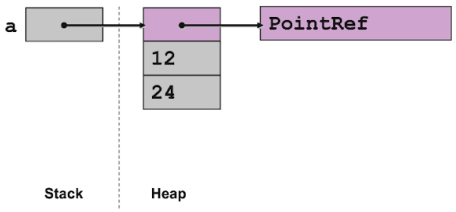
1.3.1 Reference- und Value Typen

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

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

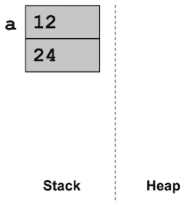
```
class PointRef { public int x, y; }

PointRef a;
a = new PointRef();
a.x = 12;
a.y = 24;
```




```
struct PointVal { public int x, y; }

PointVal a;
a = new PointVal();
a.x = 12;
a.y = 24;
```



1.3.2 Boxing und Unboxing

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

1.4 CLS: Common Language Specification

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

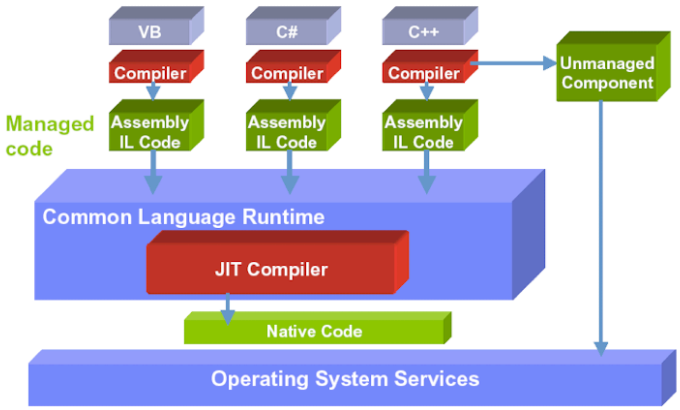


Figure 2: CLR: Common Language Runtime Architecture

1.5 Microsoft Intermediate Language (MSIL)

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

1. Sprachspezifischer Kompilier kompiliert nach MSIL
2. Just In Time Compiler (JIT) Compiler aus dem CLR kompiliert in nativen plattformabhängigen Code.

- Vorteile
  - Portabilität (Nicht-Intel-Prozessoren, Unix etc.)
  - Typensicherheit (Beim laden des Code können Typen- Sicherheit und weiter Security-Checks durchgeführt werden.
- Nachteile
  - Laufzeiteffizient (Kann durch den JIT-Compiler wettgemacht werden)

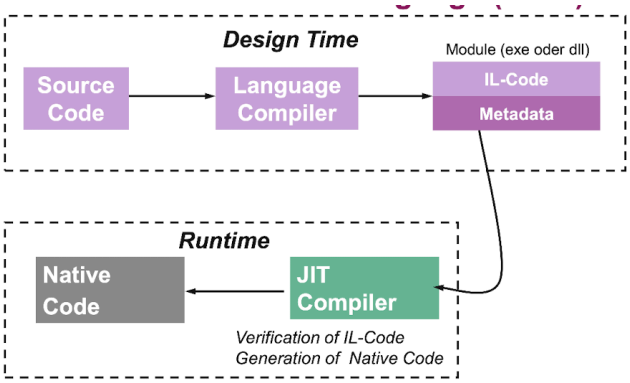


Figure 3: MSIL Kompilierung

1.6 Assemblies

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

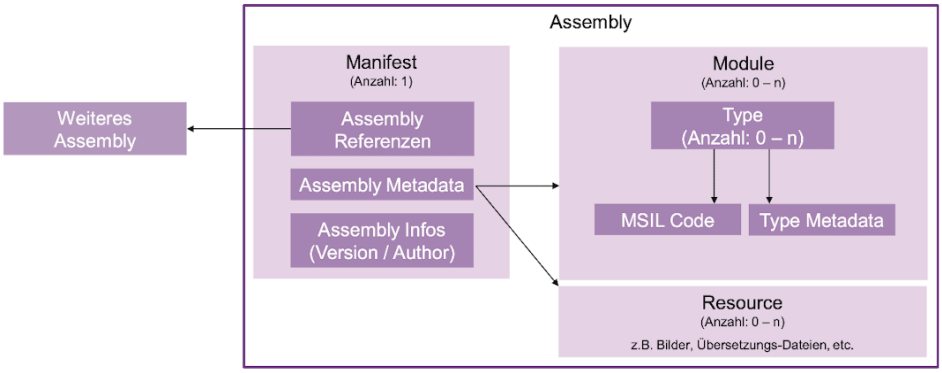


Figure 4: Assembly Überblick

1.6.1 Modules & Metadata

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

2 Projekte und Referenzen in Visual Studio

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

2.1 Projektdateien

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

2.2 Referenzen

Im Projektfile wird zwischen verschiednen Referenzen unterschieden.

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

2.3 Packages & NuGet

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

3 C# Grundlagen

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

3.1 Naming Guidelines

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

Figure 5: Naming Guidelines

3.2 Sichtabrkeitsattribute

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

Figure 6: Sichtabrkeitsattribute

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

* Gilt nicht für «nested types»

Figure 7: Standard Sichtbarkeiten von Typen

3.3 Primitivtypen

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

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

Typenkompatibilität

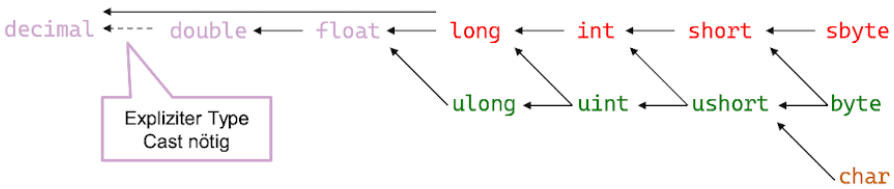


Figure 8: Typenkompatibilität

3.4 Statements

If Statements

```
int value = 5;
if (0 >= value && value < 9) {
  /* ... */
} else if (value > 10) {
  /* ... */
} else {
  Console.WriteLine("Invalid: " +
    value);
}
```

Switch Statement

```
string ctry = "Germany"; string
    language;
switch (ctry) {
  case "England":
  case "USA":
    language = "English"; break;
  case "Germany":
  case "Austria":
  case "Switzerland":
    language = "German"; break;
  case null:
    Console.WriteLine("Coutry null");
    break;
  default:
    Console.WriteLine("Unknown" + ctry
    ); break;
}
```

Loops

```
// Kopfgesteuert
int i = 1;
int sum = 0;
while (i <= 3)
{
  sum += i;
  i++;
}
// Fussgesteuert
int i = 1;
int sum = 0;
do
{
  sum += i;
  i++;
} while (i <= 3)
// Foreach
int[] a = { 3, 17, 4, 8, 2, 29 };
foreach (int x in a)
{
  sum += x;
}
```

Jumps

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

3.5 Namespaces

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

3.6 Main Methode

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

3.6.1 Top-Level Satements

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

- Nur 1x pro Assemlby erlaubt
- Arugmente heissen fix args
- Exit Codes erlaubt
- VOR dem top-level statements können usings definiert werden.
- NACH dem top-level satements können Typen definiert werden.

```
using System;
for (int i = 0; i < args.Length; i++) {
  ConsoleWriter.Write(args, i);
{
Class ConsoleWriter {
  public static void Write(string[] args, in t) // Top Level Satement
  {
    Console.WriteLine("Arg {i} O {args[i]}");
  }
}
```

3.7 Enumerationstypen

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

```
enum Days { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
Days today = Days.Monday;
if (today == Days.Monday) { /* ... */ }
enum Days { Monday = 10, Tuesday, Wednesday, Thursday, Friday, Saturday };
enum Days:byte { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
    Saturday };
// Parsing
bool success3 = Enum.TryParse("Monday", out Days day3);
```

3.8 Object

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

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

3.9 Arays

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

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

3.10 Strings

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

Interning

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

```
string s1 = "Hello ";
string s2 = "World";
string s3 = s1 + s2;
// Unveraenderbarkeit wird durch den Compiler sichergesetllt
string s1 = "Hello ";
string s2 = "World";
string s3 = s1 + s2; // string s3 = System.String.Concat(s1, s2);
s3 += "!";           // s3 = System.String.Concat(s3, "!");
// String Interpolation
string s3 = $"{DateTime.Now}: {"Hello"}";
string s4 = $"{DateTime.Now}: {(DateTime.Now.Hour < 18 ? "Hello" : "Good
    Evening")}";
```

3.11 Symbole

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

4 Klassen und Structs

4.1 Klassen

Sind Referenz Typen und werden auf dem Heap angelegt.

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

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

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

Statische Klassen
Regeln: Nur statische Members erlaubt, kann nicht instanziert werden, sind "sealed".

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

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

4.1.1 Partielle Klassen

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

```
// File1.cs
partial class MyClass {
public void Test1() { } }
// File2.cs
partial class MyClass {
public void Test2() { } }
// Verwendung
MyClass mc = new MyClass();
mc.Test1();
mc.Test2();
```

4.1.2 Abstrakte Klassen

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

4.2 Structs

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

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

Felder
Felder dürfen im Struct nicht initialisiert werden!

```
struct Point {
  int x;
  int y;
  public Point(int x, int y)
  {
  this.x = x; this.y = y;
  }
  public void MoveX(int x) { /* ... */ }
  public void MoveY(int y) { /* ... */ }
}
Point p = new Point(2, 3);
```

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

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

4.3 Properties

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

```
class MyClass {
  private int length; // Backing field
  // Property
  public int Length {
    get { return length; }
    set { length = value; }
  }
  // Compiler Output
  public int get_Length() { return length; }
  public void set_Length(int value) { length = value; }
```

4.4 Auto-Implemented Properties

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

```
// Auto Property - backing filed gets generated automatically
public int LengthAuto { get; set; }
public int LengthInitializes {get; /* set; */ };
```

4.5 Initialisierte Properties

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

```
MyClass mc = new MyClass()
{
  Length = 1,
  Width = 2
};
// Compiler Output
MyClass mc = new MyClass();
mc.Length = 1;
mc.Width = 2;
```

4.6 Indexer

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

```
class SampleCollection<T> {
   private T[] arr = new T[100];
   // Define the indexer to allow client code to use [] notation.
   public T this[int i] {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}
class Program
{
   static void Main() {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
```

4.7 Konstruktoren

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

Default-Konstruktor Klasse

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

Default-Konstruktor Klasse

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

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

Figure 9: Standard-Werte

4.7.1 Statische Konstruktoren

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

4.7.2 Dekonstruktoren

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

```
class MyClass {
  ~MyClass() {
    // Freigabe von File-Handles etc.
  }
}
```

4.7.3 Initalisierungs-Reihenfolge

4.8 Operatoren - Überladung

5 Methoden

5.1 Statische Methoden

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

5.2 Value Parameter

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

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

5.3 Reference Parameter

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

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

5.4 Out-Parameter

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

```
static void Init(out int val) {
  val = 100;
}
int value;
Init(out value);
```

5.5 In Parameter

5.6 Params-Array

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

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

5.7 Optionale Parameter (Default Values)

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

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

5.8 Named Parameter

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

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

5.9 Überladung

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

```
void Test(int x) { /* ... */ }
void Test(char x) { /* ... */ }
void Test(int x, long y) { /* ... */ }
void Test(long x, int y) { /* ... */ }
void Test(ref int x) { /* ... */ }
```

5.10 Lamda Expressions

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

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

6 Vererbung

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

6.1 Typprüfungen

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

```
class Base { }
class Sub : Base { }
class SubSub : Sub { }
public static void Test() {
  SubSub a = new SubSub();
  if (a is SubSub) { /* ... */ }  // True
  if (a is Sub) { /* ... */ }   // True
  if (a is Base) { /* ... */ }  // True
  a = null;
  if (a is SubSub) { /* ... */ }  // False / NULL
}
```

6.2 Type Casts

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

```
class Base { }
class Sub : Base { }
class SubSub : Sub { }
public static void Test() {
  Base b = new SubSub();
  Sub s = b as Sub; // as keyword for the compiler
}
```

6.3 Methoden

Die Subklasse kann Members der Basisklasse überschreiben: Methoden, Properties, Indexer. Schlüsselword "virtual" um Basis-Methode überschreibbar zu machen. Schlüsselwort "override" um die Basis-Methode zu überschreiben. Members sind per default NICHT virtual und override! Regeln: Signatur muss identisch sein, gleiche Rückgabewert, gleiche Sichtbarkeit.
Achtung!
Virtual kann nicht mit: static, abstract, private, override verwendet werden!

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

6.4 Interfaces

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

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

6.4.1 Interfaces Implementieren

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

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

7 Delegates & Events

7.1 Delegates

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

```
public delegate void Notifier(string sender);
class Examples {
  public static void Test() {
  // Deklaration Delegate-Variable
  Notifier greetings;
  // Zuweisung einer Methode
  greetings = new Notifier(SayHi);
  // Kurzform
  greetings = SayHi;
  // Aufruf einer Delegate-Variable
  greetings("John");
}
private static void SayHi(string sender) {
  Console.WriteLine("Hello {0}", sender);
}
```

7.2 Multicast Delegates

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

7.3 Events

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

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

7.4 Anonyme Methoden

Anonyme Methoden sind immer in-place.

```
class AnonymousMethods {
  int sum = 0;
  void SumUp(int i) { sum += i; }
  void Print(int i) { Console.WriteLine(i); }
  void Foo() {
    List<int> list = new List<int>();
    list.ForEach(SumUp);
    list.ForEach(Print);
  }
}
```

8 Generics

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

8.1 Type Constraints

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

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

Figure 10: Type Constraints

8.2 Vererbung

Generische Klassen können von anderen generischen Klassen erben.

8.3 Nullable Types

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

9 Iteratoren

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

9.1 Interfaces

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

```
// Nicht generische Variante
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
public interface IEnumerator
{
  object Current { get; }
  bool MoveNext();
  void Reset();
}
// Generische Variante
public interface IEnumerable<out T> : IEnumerable {
    IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> : IDisposable, IEnumerator {
  T Current { get; }
  // Weitere Members werden vererbt
}
```

9.2 Zugriff

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

9.3 Iterator-Methoden

9.3.1 yield return

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

```
yield return "expression";
```

9.4 Spezifische Iteratoren

9.5 Extension Methode

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

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

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

9.6 Deferred Evaluation

Werte werden erst dann berechnet, wenn sie abgefragt werden.

10 Exceptions

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

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

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

```
FileStream s = null;
try {
  s = new FileStream(@"C:\Temp\Test.txt", FileMode.Open);
} catch (FileNotFoundException e) {
    Console.WriteLine("{0} not found", e.FileName);
} catch (IOException) {
  Console.WriteLine("IO exception occurred");
} catch {
  Console.WriteLine("Unknown error occurred");
} finally {
  if (s != null) s.Close();
}
```

10.1 Klasse System.Exception

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

Poperties

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

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

10.2 Rethrowing

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

```
try {
  throw new Exception("Failure");
} catch (Exception e) {
  throw;
}
```

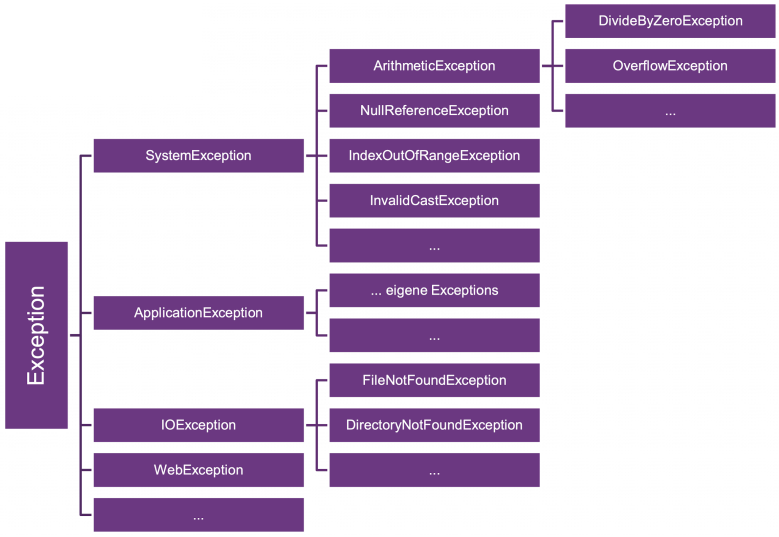


Figure 11: Exception-Typen

10.3 Exception-Klassen

10.4 Catch-Klausel

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

10.5 Exception Filters

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

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

11 Language Integrated Query (LINQ)

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

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

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

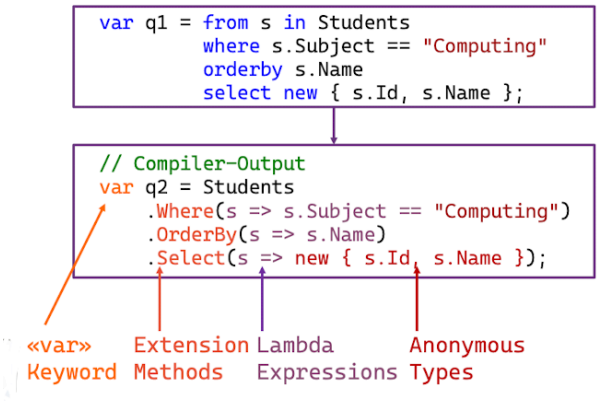


Figure 12: Compilerumwandlung zu Extension Methods

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

```
// Query Expression syntax
var queryWashingtonSorted =
    from e in employees
    where e.State == "WA"
    orderby e.Name descending
    select new
    { e.Name, e.Address };
// Query Extension Method / Lambda syntax
queryWashingtonSorted = employees
    .Where(e => e.State == "WA")
    .OrderByDescending(e => e.Name)
    .Select(e => new
    { e.Name, e.Address });
```

11.1 LINX Extension Methods

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

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

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

11.2 Expression-Bodies Members

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

```
public class Examples {
private int value;
  // Constructors / Destructors (C# 7.0)
  public Examples(int v) => this.value = v;
  ~Examples() => this.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 => this.value;
    set => this.value = value;
  }
}
```

11.3 Query Expressions Synax

```
// 1. Datenquelle waehlen
int[]numbers={0,1,2,3,4,5,6};

// 2. Query erstellen
var numQuery = from num in numbers
  where (num % 2) == 0
  select num;

// 3. Query ausfuehren
foreach (int num in numQuery) { Console.Write("{0,1} ", num); }
```

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

11.4 Gruppierung

```
// q: IEnumerable<IGrouping<string, string>>
var q = from s in Students
  group s.Name by s.Subject;
}

```

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

Figure 13: Query Operatoren / Extension Methoden

```
// Gruppierung mit direkter Wiederverwendung
var q = from s in Students
  group s.Name by s.Subject into g
  select new {
      Field = g.Key,
      N = g.Count()
   };

// Anz. Bestellungen pro Datum
from best in Bestellungen
group best by best.Datum into datumGroup
orderby datumGroup.Key
select new {
   Datum = datumGroup.Key,
   Anzahl = datumGroup.Count()
};
```

11.5 Inner Joins

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

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

11.6 Group Joins

Ein Group Join verwendet die into Expression.

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

11.7 Left Outer Joins

```
var q = from s in Students
  join m in Markings on s.Id equals m.StudentId into match
  from sm in match.DefaultIfEmpty()
```

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

11.8 Select Many

Erleichtert das Zusammenfassen verschachtelter Listen.

12 Direct Initialization

12.1 Object Initializers

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

```
Student s1 = new Student("John") {
  Id = 2009001, // Set public field
  Subject = "Computing" // Set property
};
Student s2 = new Student {
  Name = "Ann",
  Id = 2009002,
  Subject = "Mathematics"
};
```

12.2 Collection Initializers

Ist das selbe wie Objekte Initializers, jedoch mit Listen.

```
List<int> l1 = new List<int> { 1, 2, 3, 4 };
Dictionary<int, string> d1 = new Dictionary<int, string>
  {1,"a"},
  {2,"b"},
  {3,"c"}
};
d1 = new Dictionary<int, string> {
  [1] = "a",
  [2] = "b",
  [3] = "c"
};

object s = new Dictionary<int, Student> {
  { 2009001, new Student("John") {
    Id = 2009001,
    Subject = "Computing" } },
  { 2009002, new Student {
      Name = "Ann", Id = 2009002,
        Subject = "Mathematics" } }
};
```

12.3 Anonymous Types (Let)

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

13 Tasks

Leichtgewichtige Variante eines Threads und repräsentiert eine asynchrone Operation.

Task

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

Thread

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

14 Entity Framework Core

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

Database First Generieren eines Modells aus einer vorhandenen Datenbank.
Code First Manuelles Codieren eines Modells, das der Datenbank entspricht.

```
// Models
public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }

        public List<Post> Posts { get; } = new();
    }

    public class Post
    {
        public Blog Blog { get; set; }
    }
}
```

```
// DB context
public class DatabaseContext : DbContext
{
    public DbSet<Author> Authors { get; set; }
    public DbSet<Book> Books { get; set; }

  public BlogDB() : base("name=ErstesBeispiel") {
    Database.SetInitializer<BlogDB>(new DropCreateDatabaseAlways<DbContext
    >());
  }
}
```

```
// use the databse
using (var db = new BloggingContext()) {
  // Note: This sample requires the database to be created before running.
    Console.WriteLine($"Database path: {db.DbPath}.");

    // Create
    Console.WriteLine("Inserting a new blog");
    db.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
    db.SaveChanges();

    // Read
    Console.WriteLine("Querying for a blog");
    var blog = db.Blogs
      .OrderBy(b => b.BlogId)
        .First();

     // Update
     Console.WriteLine("Updating the blog and adding a post");
     blog.Url = "https://devblogs.microsoft.com/dotnet";
     blog.Posts.Add(
      new Post { Title = "Hello World", Content = "I wrote an app using EF
    Core!" });
      db.SaveChanges();

     // Delete
```

```
25      Console.WriteLine("Delete the blog");
26      db.Remove(blog);
27      db.SaveChanges();
28  }
```

14.1 Code First

14.1.1 Attribute / Data Annotations

```
1  public class Angestellter
2    {
3              [Key, Column("PersNr")]
4              public int Id { get; set; }
5              [MaxLength(20), Required]
6              public string Name { get; set; }
7              [Column("Tel")]
8              public int? Telefonnummer { get; set; }
9              [Column("Salaer", TypeName = "DECIMAL(7,2)")]
10             public decimal Salaer { get; set; }
11             [MaxLength(20)]
12             public string Wohnort { get; set; }
13             [Column(TypeName = "DATETIME")]
14             public DateTime? Eintrittsdatum { get; set; }
15             [Column("AbtNr")]
16             public int? AbteilungId { get; set; }
17             [Column(TypeName = "DECIMAL(7,2)")]
18             public decimal? Bonus { get; set; }
19             [Column("Chef")]
20             public int? ChefId { get; set; }
21
22             [ForeignKey(nameof(AbteilungId))]
23             public virtual Abteilung Abteilung { get; set; }
24
25         [ForeignKey(nameof(ChefId))]
26         public virtual Angestellter Chef { get; set; }
27
28         [InverseProperty(nameof(Chef))]
29         public virtual ICollection<Angestellter> Unterstellte { get; set; }
     = new List<Angestellter>();
30   }
```

14.2 Model Builder

Mit dem Model Builder kann deklarativ festgelegt werden, wie das Model generiert werden soll. Das Resultat ist das selbe wie mit der Attribut Variante.

14.3 Lazy-, Eager-Loading

Es wird standardmässig Lazy Loading verwendet.
Lazy Loading: Daten werden erst geladen, wenn sie dereferenziert werden. z.B erst wenn effektiv auf die Membervariable (Liste aus mehreren Items) zugegriffen wird. Für das implizite Lazy Loading müssen die Methoden virtual definiert werden.
Eager Loading: Das komplette Objekt wird geladen.

14.4 DB Context

Der DBContext ist das Herzstück des Entity Frameworks. Er ist die Verbindung zwischen unseren Entitätsklassen und der Datenbank. Der DBContext ist verantwortlich für die Datenbankinteraktionen wie das Abfra-

gen der Datenbank und das Laden der Daten in den Speicher als Entität. Er verfolgt auch die an der Entität vorgenommenen Änderungen und speichert die Änderungen in der Datenbank.

14.5 Optimistic Concurrency

Annahme: Zwischen laden und speichern eines Datensatzes wird dieser nicht verändert. Zwei Möglichkeiten um die Änderungen zu detektieren: Timestamp / Row Version, Concurrency Tokens / Daten-Versionen. Diese werden im DBContext pro Entity definiert.

Timestamp Pro Record Timestamp / Row Version. Timestamp ist teil des Objektes

Concurrency Token Beim laden der Daten die originalwerte wegkopieren und beim überschreiben einen Vergleich machen, ob die Daten verändert wurden.

15 Google Remote Procedure Call (gRPC)

Der neue Standard-Technologie für die Backend-Kommunikation in .NET. Primär Server-to-Server Kommunikation im Fokus (Microservices). Hohe Performance von zentraler Bedeutung. Nicht als Frontend-API gedacht. Ist der Ersatz für Windows Communication Foundation.

Kommunikationsprotokoll HTTP/2

Interface Definition Language Google Protocol Buffers

15.1 Architektur

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

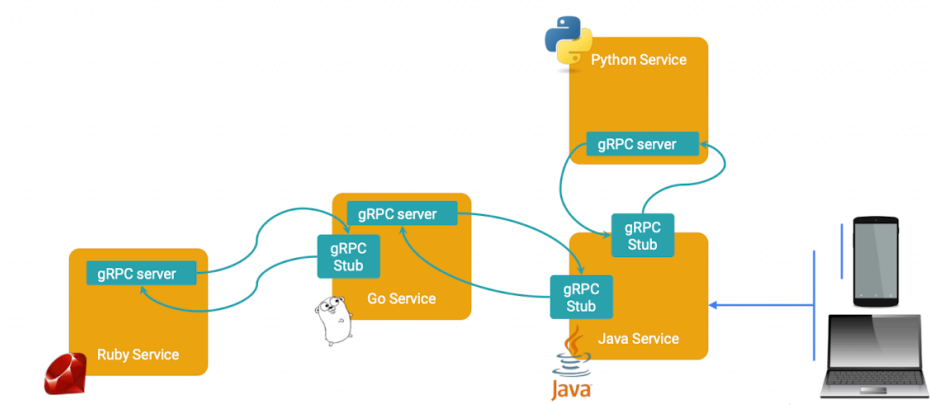


Figure 14: Architekturbeispeil

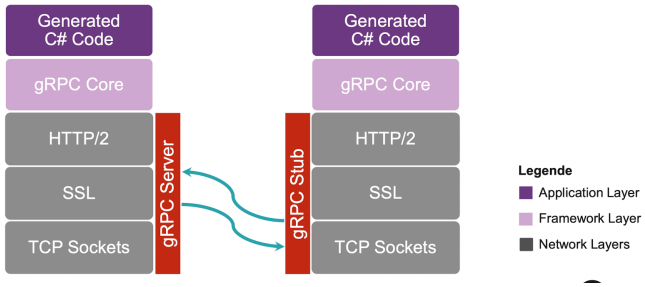


Figure 15: Kommunikationsstack

HTTP/2 Features

Multiplexing Mehrere gRPC Calls pro TCP/IP Session

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

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

15.2 Protocol Buffers

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

Proto Files
Datei-Endung "*.proto". Service-Methoden haben immer einen Parameter und einen Rückgabewert.

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

15.3 Streams

Es werden drei Modi untersützt: Server Streaming (Sever - Client), Client Streaming (Client - Server), Bi-directional / Duplex Streaming Call. Es werden zwei verschiedene Modi von Lesen unterstützt: Synchrones und asynchrones lesen.

15.4 Beispielapplikation

16 Reflection

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

16.1 Type-Discovery

Suche aller Typen in einem Assembly.

```
Assembly a01 = Assembly.Load("mscorlib");

Type[] t01 = a01.GetTypes();
foreach (Type type in t01)
{
  Console.WriteLine(type);
  MemberInfo[] mInfos = type.GetMembers();
  foreach (var mi in mInfos)
  {
    Console.WriteLine(
      "\t{0}\t{1}",
      mi.MemberType,
      mi);
  }
}
```

16.2 Member auslesen

```
Type type = typeof(Counter);
MemberInfo[] miAll = type.GetMembers();
foreach (MemberInfo mi in miAll)
{
  Console.WriteLine(
  "{0} is a {1}",
  mi, mi.MemberType);
}
Console.WriteLine("----------");
PropertyInfo[] piAll = type.GetProperties();
foreach (PropertyInfo pi in piAll)
{
  Console.WriteLine(
  "{0} is a {1}",
  pi, pi.PropertyType);
}

// Filter members according to BindingFlag or FilterName
Type type = typeof(Assembly); BindingFlags bf =
   BindingFlags.Public |
   BindingFlags.Static |
   BindingFlags.NonPublic |
   BindingFlags.Instance |
   BindingFlags.DeclaredOnly;

System.Reflection.MemberInfo[] miFound = type.FindMembers(
  MemberTypes.Method, bf, Type.FilterName, "Get*"
);
```

16.3 Field Information

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

16.4 Property Info

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

16.5 Mehtod Info

Die Method Info beschreibt eine Methode einer Klasse (Name, Parameter, Rückgabewert, Sicht- barkeit). Sie leitet von Klasse MethodBase ab. Die Methode wird mit Invoke() aufgerufen.

16.6 Constructor Info

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

17 Attributes

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

17.1 Anwendungfälle

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

17.2 Custom Attributes