

.NET Technologies

HS2021

Marco Agostini

Computer Science
University of Applied Sciences of Eastern Switzerland
September 2021

Contents

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

6.3	Methoden	17
6.4	Interfaces	18
6.4.1	Interfaces Implementieren	18
7	Delegates & Events	19
7.1	Delegates	19
7.2	Multicast Delegates	19
7.3	Events	19
7.4	Anonyme Methoden	19
8	Generics	20
8.1	Type Constraints	20
8.2	Vererbung	20
8.3	Nullable Types	20
9	Iteratoren	21
9.1	Interfaces	21
9.2	Zugriff	21
9.3	Iterator-Methoden	21
9.3.1	yield return	21
9.4	Spezifische Iteratoren	21
9.5	Extension Methode	21
9.6	Deferred Evaluation	22
10	Exceptions	23
10.1	Klasse System.Exception	23
10.2	Rethrowing	23
10.3	Exception-Klassen	24
10.4	Catch-Klausel	24
10.5	Exception Filters	24
11	Language Integrated Query (LINQ)	25

1 Introduction

1.1 .NET Framework

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

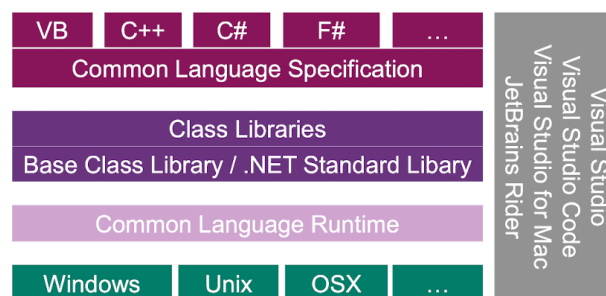


Figure 1: .NET Framework Architektur

1.2 CLR: Common Language Runtime

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

1.3 CTS: Common Type System

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

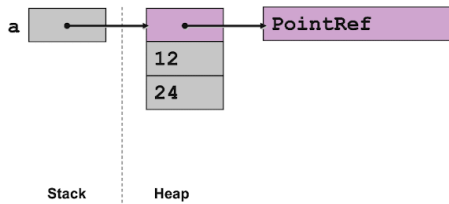
1.3.1 Reference- und Value Typen

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

- Reference Types
 - Werden auf dem Heap gespeichert
 - Variable enthält Referenz
 - Automatisch Garbage Collection
 - Konstruktor erzeugt und initialisiert Objekt
 - Objekt hat eine Referenz auf seine Typenbeschreibung

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

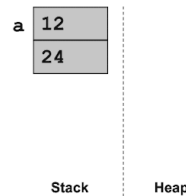
PointRef a;
a = new PointRef();
a.x = 12;
a.y = 24;
```



- Value Types
 - Zur Speicherung Rohen Werte auf dem Stack
 - Konstruktor macht nur eine Initialisierung
 - Boxing: automatische Umwandlung von Reference Type
 - sealed

```
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.

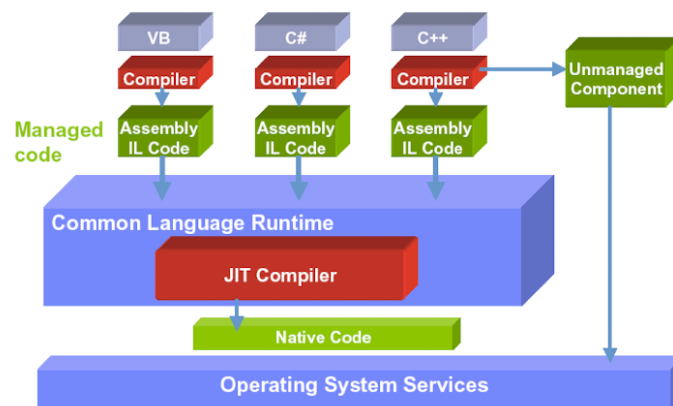


Figure 2: CLR: Common Language Runtime Architecture

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)

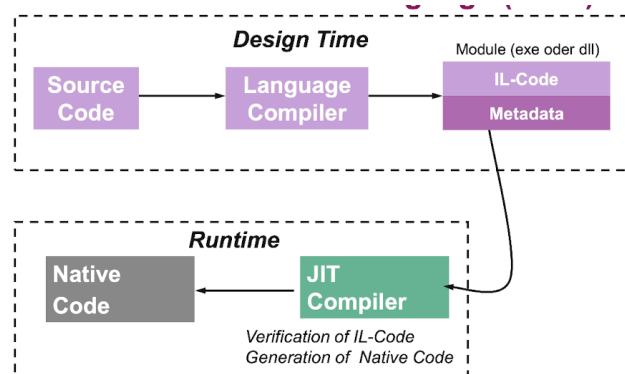


Figure 3: MSIL Kompilierung

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.

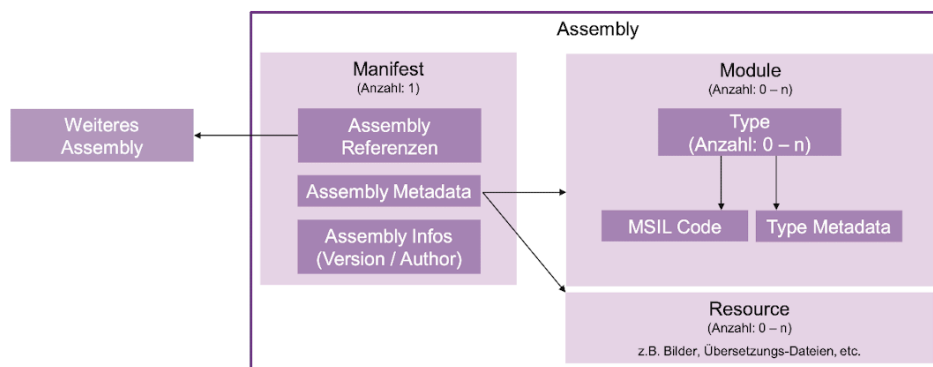


Figure 4: Assembly Überblick

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 unterstützten XML-Elemente und Attribute.

2.2 Referenzen

Im Projektfile wird zwischen verschiedenen Referenzen unterschieden.

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

2.3 Packages & NuGet

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

3 C# Grundlagen

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

3.1 Naming Guidelines

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

Figure 5: Naming Guidelines

3.2 Sichtbarkeitsattribute

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

Figure 6: Sichtbarkeitsattribute

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

* Gilt nicht für «nested types»

Figure 7: Standard Sichtbarkeiten von Typen

3.3 Primitivtypen

Ganzzahlen

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

Fliesskommazahlen

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

Typenkompatibilität

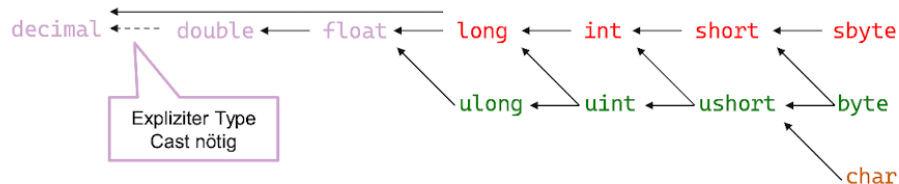


Figure 8: Typenkompatibilität

3.4 Statements

If Statements

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

Switch Statement

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

Loops

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

Jumps

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

3.5 Namespaces

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

3.6 Main Methode

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

3.6.1 Top-Level Statements

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

Regeln:

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

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

3.7 Enumerationstypen

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

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

3.8 Object

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

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

3.9 Arrays

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

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

3.10 Strings

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

Interning

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

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

3.11 Symbole

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

4 Klassen und Structs

4.1 Klassen

Sind Referenz Typen und werden auf dem Heap angelegt.

Vererbung

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

Felder

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

Nested Types

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

Statische Klassen

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

Statische Usings

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

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

4.1.1 Partielle Klassen

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

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

4.1.2 Abstrakte Klassen

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

4.2 Structs

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

Vererbung

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

Felder

Felder dürfen im Struct nicht initialisiert werden!

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

Verwendung

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

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

4.3 Properties

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

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

4.4 Auto-Implemented Properties

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

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

4.5 Initialisierte Properties

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

```

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

```

4.6 Indexer

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

```

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

```

4.7 Konstruktoren

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

Default-Konstruktor Klasse

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

Default-Konstruktor Klasse

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

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

Figure 9: Standard-Werte

4.7.1 Statische Konstruktoren

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

4.7.2 Dekonstruktoren

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

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

4.7.3 Initialisierungs-Reihenfolge

4.8 Operatoren - Überladung

5 Methoden

5.1 Statische Methoden

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

5.2 Value Parameter

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

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

5.3 Reference Parameter

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

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

5.4 Out-Parameter

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

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

5.5 In Parameter

5.6 Params-Array

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

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

5.7 Optionale Parameter (Default Values)

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

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


5.8 Named Parameter

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

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

5.9 Überladung

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

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

6 Vererbung

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

6.1 Typprüfungen

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

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

6.2 Type Casts

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

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

6.3 Methoden

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

Achtung!

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

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

6.4 Interfaces

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

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

6.4.1 Interfaces Implementieren

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

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

7 Delegates & Events

7.1 Delegates

Ein Delegate 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 Delegates wird durch den Namen des Delegates definiert. Verwendung: Methoden können als Parameter übergeben werden, Definition von Callback-Methoden. Eine Zuweisung von null ist erlaubt.

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

7.2 Multicast Delegates

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

7.3 Events

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

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

7.4 Anonyme Methoden

Anonyme Methoden sind immer in-place.

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

8 Generics

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

8.1 Type Constraints

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

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

Figure 10: Type Constraints

8.2 Vererbung

Generische Klassen können von anderen generischen Klassen erben.

8.3 Nullable Types

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

9 Iteratoren

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

9.1 Interfaces

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

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

9.2 Zugriff

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

9.3 Iterator-Methoden

9.3.1 yield return

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

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

9.4 Spezifische Iteratoren

9.5 Extension Methode

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

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

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

9.6 Deferred Evaluation

Werte werden erst dann berechnet, wenn sie abgefragt werden.

10 Exceptions

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

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

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

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

10.1 Klasse System.Exception

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

Properties

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

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

10.2 Rethrowing

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

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

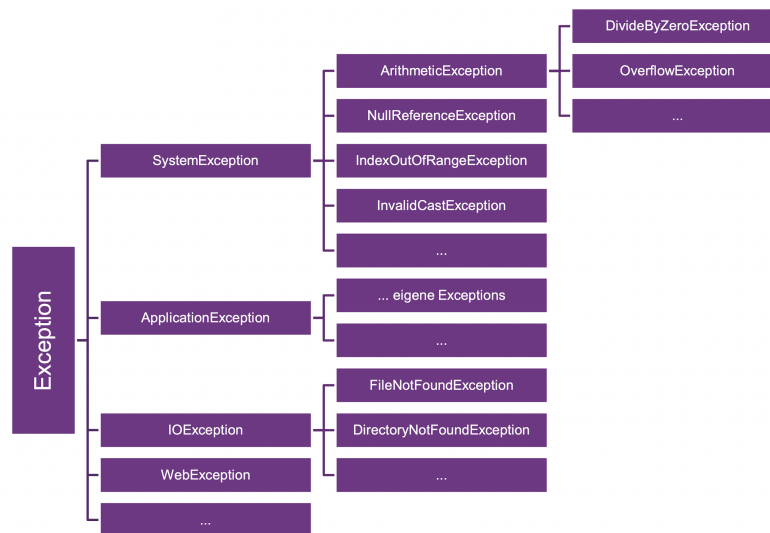



Figure 11: Exception-Typen

10.3 Exception-Klassen

10.4 Catch-Klausel

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

10.5 Exception Filters

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

```

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

```

11 Language Integrated Query (LINQ)

Ist eine Sprach-Integrierte Abfragesprache.

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

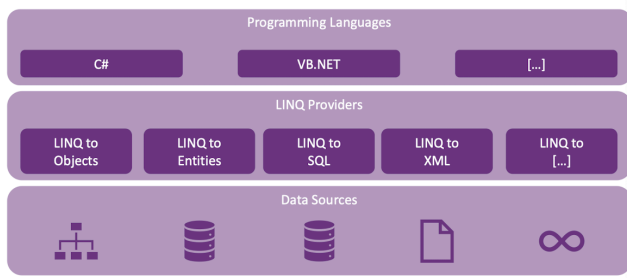


Figure 12: Die Architektur von LINQ