

**Zusammenfassung**

# **Microsoft Technologien (.NET)**

J.Hürzeler, S.Moll, S.Dellsperger, J.Klaiber, M.Wieland

Fachhochschule OST

8. Dezember 2022

## **Lizenz**

"THE BEER-WARE LICENSE" (Revision 42): Joshua, Sharon, Severin, Julian and Michael wrote this file. As long as you retain this notice you can do whatever you want with this stuff. If we meet some day, and you think this stuff is worth it, you can buy us a beer in return.

## Inhaltsverzeichnis

<b>1</b>	<b>Der Heilige Gral</b>	<b>7</b>
1.1	Reference oder Value . . . . .	7
1.2	Lamdas . . . . .	8
1.3	Delegates, Events . . . . .	8
1.4	Extension Methods . . . . .	8
1.5	LINQ . . . . .	10
1.6	Entity Framework . . . . .	11
1.7	WCF . . . . .	12
1.7.1	Server . . . . .	12
1.7.2	Client . . . . .	14
<b>2</b>	<b>.NET</b>	<b>15</b>
2.1	CLR: Common Language Runtime . . . . .	15
2.2	CTS: Common Type System . . . . .	15
2.3	CLS: Common Language Specification . . . . .	16
2.4	MSIL: Microsoft Intermediate Language . . . . .	16
2.5	JIT: Just in Time Compilation . . . . .	16
2.6	Assembly / Komponenten . . . . .	17
2.6.1	Module . . . . .	18
2.6.2	References . . . . .	18
2.6.3	Packages . . . . .	18
2.7	Kompilierung . . . . .	19
2.8	Garbage Collection . . . . .	19
2.8.1	Generationen . . . . .	19
2.8.2	Deterministic Finalization . . . . .	20
2.8.3	Finalizer . . . . .	20
2.8.4	Object Pinning . . . . .	20
2.8.5	Weak References . . . . .	21
2.8.6	Memory Leaks . . . . .	21
<b>3</b>	<b>.NET Standard</b>	<b>22</b>
<b>4</b>	<b>Command Line Interface CLI</b>	<b>23</b>
4.1	Kommandos . . . . .	23
<b>5</b>	<b>Visual Studio 22</b>	<b>24</b>
5.1	Solution . . . . .	24
5.2	Umbenennen . . . . .	24
5.3	Ordnerstruktur . . . . .	24
5.4	Projekt-Dateien . . . . .	24
<b>6</b>	<b>C# Grundlagen</b>	<b>25</b>
6.1	Unterschiede zu Java . . . . .	25
6.2	Naming Conventions . . . . .	25
6.3	Sichtbarkeiten . . . . .	25
6.4	Operatoren . . . . .	26
6.5	Pre-, Post-Inkrement . . . . .	28

6.6	Statements . . . . .	28
6.6.1	If Else If Else . . . . .	28
6.6.2	Switch Case . . . . .	28
6.6.3	Loops . . . . .	28
6.6.4	Kommentare . . . . .	28
6.7	Datentypen . . . . .	29
6.7.1	Casts . . . . .	30
6.7.2	Reference Types / Referenztypen . . . . .	31
6.7.3	Value Types / Werttypen . . . . .	31
6.8	Nullable Types . . . . .	32
6.9	Boxing / Unboxing . . . . .	33
6.10	Object . . . . .	33
6.11	String . . . . .	33
6.12	Arrays . . . . .	34
6.13	Indexer . . . . .	34
6.14	List . . . . .	34
6.15	Namespaces . . . . .	35
6.16	Main Methode . . . . .	35
<b>7</b>	<b>Variablen und Properties</b>	<b>37</b>
7.1	Konstanten . . . . .	37
7.2	ReadOnly . . . . .	37
7.3	Properties . . . . .	37
7.3.1	Auto Properties . . . . .	37
7.3.2	Properties direkt initialisieren . . . . .	38
7.3.3	Abstrakte Properties/Indexes . . . . .	38
<b>8</b>	<b>Methoden</b>	<b>39</b>
8.1	Overloading . . . . .	39
8.2	Call by value . . . . .	39
8.3	Call by reference . . . . .	39
8.4	Out Parameter . . . . .	39
8.5	Params Array . . . . .	40
8.6	Optionale Parameter (Default Values) . . . . .	40
8.7	Named Parameter . . . . .	40
8.8	Virtual . . . . .	40
8.9	Override . . . . .	40
8.10	Methoden überdecken mit New . . . . .	41
8.11	Dynamic Binding . . . . .	41
8.12	Abstrakte Methoden . . . . .	41
8.13	Sealed . . . . .	41
<b>9</b>	<b>Klassen, Structs</b>	<b>43</b>
9.1	Klassen . . . . .	43
9.1.1	Type Casts . . . . .	43
9.1.2	Typprüfung . . . . .	43
9.1.3	Typprüfungen mit implizitem Type Cast . . . . .	43
9.1.4	Operatoren Überladen . . . . .	44
9.1.5	Partial Class . . . . .	44

9.2	Abstrakte Klassen . . . . .	44
9.3	Sealed Klassen . . . . .	45
9.4	Statische Klassen . . . . .	45
9.5	Structs . . . . .	46
9.6	Konstruktoren . . . . .	46
9.7	Initialisierungsreihenfolge . . . . .	47
9.8	Destruktoren . . . . .	48
9.9	Operator Overloading . . . . .	49
<b>10</b>	<b>Interfaces</b>	<b>50</b>
10.1	Interface Naming Clashes . . . . .	50
<b>11</b>	<b>Enum</b>	<b>51</b>
<b>12</b>	<b>Generics</b>	<b>52</b>
12.1	Type Constraints . . . . .	52
12.1.1	Beispiele . . . . .	52
12.2	Typprüfungen . . . . .	53
12.3	Generische Vererbung . . . . .	53
12.4	Generische Delegates . . . . .	54
12.5	Generische Collections . . . . .	54
<b>13</b>	<b>Nullability</b>	<b>55</b>
13.1	Default operator literal . . . . .	55
13.2	Structs / Value Types . . . . .	55
13.2.1	Struct <Nullable> . . . . .	55
13.2.2	T? Syntax . . . . .	56
13.2.3	Sicheres Lesen & Type Casts . . . . .	56
13.3	Klassen / Reference Types . . . . .	56
13.3.1	Syntax . . . . .	56
13.4	Syntactic sugar . . . . .	57
13.4.1	is null is not null (Pattern matching) . . . . .	57
13.4.2	?? (Null-Coalescing Operator) . . . . .	57
13.4.3	??= (Null-Coalescing Assignment) . . . . .	57
13.4.4	?. (Null-Conditional Operator) . . . . .	57
<b>14</b>	<b>Record Types</b>	<b>58</b>
14.1	Deklarationsmöglichkeiten . . . . .	58
14.2	Value Equality . . . . .	59
14.3	Nondestructive mutation . . . . .	59
<b>15</b>	<b>Delegates</b>	<b>60</b>
15.1	Multicast Delegates . . . . .	61
15.2	Funktionsparameter . . . . .	62
15.3	Anonyme Methoden . . . . .	63
15.4	Callbacks . . . . .	63
15.5	Events . . . . .	65
15.6	EventHandler . . . . .	66

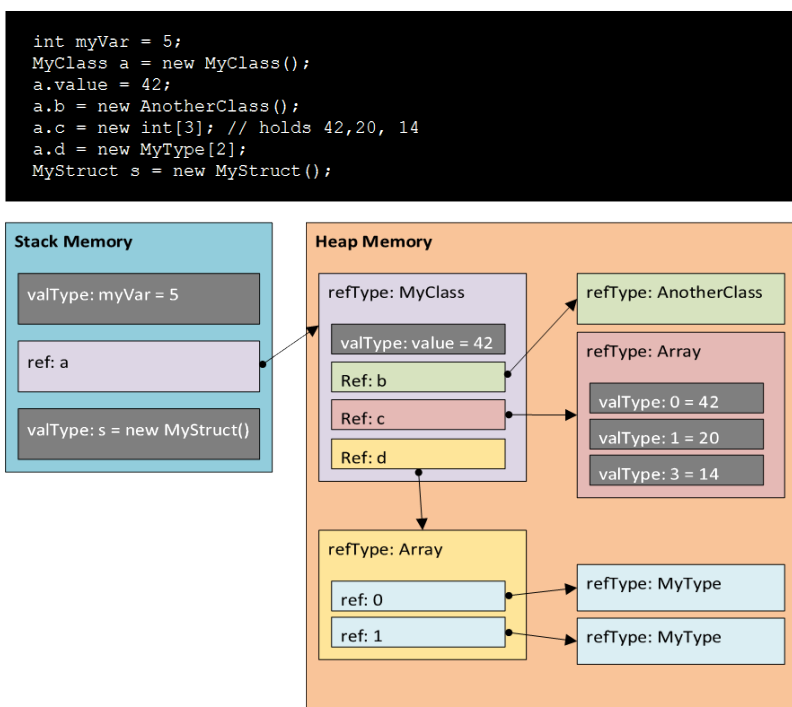
<b>16 Lambdas</b>	<b>67</b>
16.1 Closure . . . . .	67
<b>17 Iteratoren</b>	<b>68</b>
17.1 Foreach Loop . . . . .	68
17.2 Iterator Interfaces . . . . .	68
17.3 Iterator Methoden und Yield Return . . . . .	69
<b>18 Extension Methods</b>	<b>71</b>
18.1 Deferred Evaluation . . . . .	71
<b>19 Exceptions</b>	<b>73</b>
19.1 Exception Filter . . . . .	73
<b>20 Direct Initialization</b>	<b>75</b>
20.1 Object Initializers . . . . .	75
20.2 Collection Initializers . . . . .	75
20.3 Kombination aus Object und Collection Initializers . . . . .	75
20.4 VAR: Anonymous Types . . . . .	76
<b>21 LINQ: Language Integrated Query</b>	<b>77</b>
21.1 Lambda Expressions . . . . .	77
21.2 Extension Methods und Deferred Evaluation . . . . .	78
21.3 Extension Methods und Immediate Evaluation . . . . .	78
21.4 Extensions Syntax (Fluent Syntax) . . . . .	79
21.4.1 LINQ Extension Methods . . . . .	79
21.4.2 SelectMany . . . . .	79
21.5 Query Expressions Syntax . . . . .	80
21.5.1 Range Variabeln . . . . .	81
21.5.2 Gruppierung . . . . .	81
21.5.3 Inner Joins . . . . .	82
21.5.4 Group Joins . . . . .	82
21.5.5 Left Outer Joins . . . . .	82
21.5.6 Let . . . . .	83
21.5.7 Select Many . . . . .	83
21.5.8 Left Outer Join mit Select Many . . . . .	83
<b>22 Expression-Bodied Members</b>	<b>84</b>
<b>23 Tasks</b>	<b>85</b>
23.1 Task vs Thread . . . . .	85
23.2 Task API (synchrone waits) . . . . .	85
<b>24 ASYNC / AWAIT</b>	<b>86</b>
24.1 Synchron vs. Asynchron . . . . .	86
24.2 async / await . . . . .	86
24.3 Cancellation . . . . .	87
24.3.1 Cancellation Token Source . . . . .	88

<b>25 Entity Framework</b>	<b>89</b>
25.1 OR Mapping	89
25.1.1 Modell	90
25.1.2 Relationale DB (SQL Server)	92
25.2 Database Context	95
25.2.1 Klasse "DbContext"	95
25.2.2 LINQ to Entities	96
25.2.3 CUD Operationen (Create, Update, Delete)	96
25.2.4 CUD von Assoziationen	97
25.2.5 Change Tracking	98
25.3 Lazy-, Eager-Loading	99
25.4 Optimistic Concurrency	100
25.4.1 Erkennung von Konflikten	100
25.5 Inheritance	101
25.6 Database Migration	102
25.7 Data Type Mappings	103
<b>26 GRPC: Google Remote Procedure Call</b>	<b>104</b>
26.1 Protocol Buffers	104
26.2 gRPC C# API	106
26.3 Beispiel Customer Service	107
26.4 Streams	108
26.5 Special Types	111
26.6 Exception Handling	112
26.7 Cancellation	113
<b>27 Reflection</b>	<b>114</b>
27.1 Anwendungen	114
27.2 Type Discovery	114
27.3 Member auslesen	115
27.4 Field Information	115
27.5 Property Information	116
27.6 Method Info	116
27.7 Constructor Info	116
27.8 Example of Reflection Usage	117
27.9 Attributes	117
<b>28 Attributes</b>	<b>118</b>
28.1 Anwendungsfälle	118
28.2 Typen	118
28.3 Eigene Attribute	119
28.4 Reflection Emit	120

# 1 Der Heilige Gral

## 1.1 Reference oder Value

- Man unterscheidet zwischen Referenz- (Klassen) und Value Typen (Structs, Enum und primitive Datentypen)
- Bei Referenztypen liegt die Referenz auf dem Stack und das eigentliche Objekt auf dem Heap.
- Bei der Parameterübergabe bei Value wird eine Kopie angelegt. Bei Referenztypen wird einfach nur die Referenz auf dem Stack kopiert (nicht aber das Objekt!)
- Für die Parameterübergabe by Reference ist das Keyword **ref** notwendig
- Ein **out** Parameter verhält sich wie ein **ref** Parameter, mit dem Unterschied, dass er nicht initialisiert sein muss.
- Strings werden auf dem Heap als **char** Arrays alloziert.
- Boxing nennt man die automatische Umwandlung von einem Value in einen Referenz Type (implizit).
- Unboxing ist die automatische Umwandlung von einem Referent in einen Value Type (explizit).



## 1.2 Lamdas

- Lamdas werden in `Func<[param_type], [return_type]> myLamda`; gespeichert, wobei der letzte Typ in den spitzen Klammern der Rückgabe Typ ist.

## 1.3 Delegates, Events

Delegates sind typsichere Funktions-Pointer, wobei die Typsicherheit vom Compiler gewährleistet wird.

- Der erste Parameter ist bei EventHandler immer immer das `this` Objekt!
- In einem Event können mehrere Lambda/Funktionen registriert werden (`+=`)
- Wird ein Delegate in einer `Func<T>` gespeichert kann das Delegate von überall verwendet werden. Das `event` Keyword macht das Delegate privat und generiert public Methoden für die Registrierung und Deregistrierung.

---

```

1 // define event handler, where event happens (z.B Schalter)
2 public event EventHandler<MyEventArgs> MyEventHandler;
3 // define event args
4 public MyEventArgs : EventArgs {
5     public string Value {get; set; }
6 }
7
8 // register a function to the event
9 // function is called, when event happens
10 MyEventHandler += (o, e) => {
11     // do anything
12 }
13
14 // Invoke EventHandler
15 MyEventHandler?.Invoke(this, new MyEventArgs() {
16     Value = "test"
17 });

```

---

```

1 // without event args
2 public event Action<bool> MyEvent;
3 MyEvent?.Invoke(this, true);
4 // called function with bool param
5 public void EventHappens(bool state) { this.Light = state; }
6 MyEvent += Light.EventHappens; // register

```

---

## 1.4 Extension Methods

- Eine Extension Method **und** die Wrapper Klasse müssen `static` sein und der erste Parameter der Methode `this` als Prefix haben.
- Der erste Parameter definiert die Klasse, welche erweitert wird

---

```

1 using MyExtensions; // in callee
2
3 // simple iterator
4 public static class MyExtensions {
5     public static IEnumerable<T> Ext1<T>(this IEnumerable<T> input) {
6         foreach (T item in input) {

```

---



```
7         yield return item;  
8     }  
9 }  
10 }
```

---

## 1.5 LINQ

- Das Select Statement gibt ein Objekt vom Typ `IEnumerable<T>` eines anonymen Types mit den jeweiligen Feldern zurück.
- Nützliche Funktionen sind `g.Count()`, `g.Average(e => e.Amount)`, `g.Sum(e => e.Amount)`, `x.Min(x => x.Price)`, `x.Max(x => x.Price)`

---

```

1 // extension syntax
2 var query = myArray
3   .Where(e => e.Name.StartsWith("a") && e.Name.EndsWith("b"))
4   .GroupBy(e => e.Department)
5   .OrderBy(e => e.Name)
6   .Select(e => new {
7       Name = e.Name,
8       Department = (e.Department == null) ? "empty" : e.Department
9   })
10  .ToList();
11
12 // query syntax
13 var query = from e in myArray
14             from d in e.departments
15             where e.StartsWith("a")
16             group e by e.Name into mygroup [where mygroup.Count() > 3]
17             orderby e.Name, d.Name // order by two fields
18             select new {
19                 Name = mygroup.Key,
20                 Department = d.Name
21             };
22
23 // inner join (==)
24 var innerJoinQuery =
25     from c in categories
26     join p in products on c.ID equals p.CategoryID // or compound 'from' over nav prop
27     select new {
28         ProductName = p.Name,
29         Category = c.Name
30     };
31
32 // group join (into)
33 var innerGroupJoinQuery =
34     from c in categories
35     join p in products on c.ID equals p.CategoryID into prodGroup
36     select new {
37         CategoryName = c.Name,
38         Products = prodGroup.Count()
39     };
40
41 // left outer join (DefaultIfEmpty() combined with group join)
42 var leftOuterJoinQuery =
43     from c in categories
44     join p in products on c.ID equals p.CategoryID into prodGroup
45     from item in prodGroup.DefaultIfEmpty(
46         new Product { // set default
47             Name = String.Empty,
48             CategoryID = 0
49         })
50     select new {
51         CatName = c.Name,
52         ProdName = item.Name
53     };

```

---

## 1.6 Entity Framework

- Über den `DbContext` findet die Kommunikation mit der Datenbank statt. Er ist für die Persistierung und Transaktionshandling verantwortlich. Jedes persistente Objekt ist dem DB Kontext zugeordnet, was Caching und Tracking von Änderungen erlaubt.
- Der Entity Key ist die OO Representation des Primary/Foreign Key. Er wird vom `DbContext` gesetzt und hat beim Erzeugen den Default Wert seines Types. Sobald die OO Representation in der DB gespeichert wird, wird der Entity Key mit dem Primary Key aus der DB überschrieben.
- Für die Sicherstellung der referenziellen Integrität sind die Business Klasse selber zuständig.
- Das Entity Framework verwendet standardmässig **Lazy Loading**. Das bedeutet, dass die Daten erst geladen werden, wenn sie explizit dereferenziert werden. Die Navigation Property muss beim Lazy Loading **virtual** sein!
- Beim **Eager Loading** wird das komplette Objekt mit einer `Include("A.B")` Anweisung geladen.

---

```
1 // lazy loading (navigation property needs to be virtual)
2 public class Blog {
3     public int BlogId { get; set; }
4     public string Name { get; set; }
5     public string Url { get; set; }
6     public string Tags { get; set; }
7
8     // allows lazy loading
9     public virtual ICollection<Post> Posts { get; set; }
10 }
11
12 // eager loading (load everything at one using Include())
13 using (var context = new BloggingContext()) {
14     var blogs1 = context.Blogs
15         .Include(b => b.Posts)
16         .ToList();
17
18     var blogs2 = context.Blogs
19         .Include("Posts")
20         .ToList();
21 }
22
23 // disable lazy loading globally
24 public BloggingContext() {
25     this.Configuration.LazyLoadingEnabled = false;
26 }
```

---

## 1.7 WCF

- Client und Server müssen das gleiche Binding haben. Dieses wird über den Metadata Exchange publiziert (MEX).
- Standardmässig werden alle public Properties/Felder eines DTO nach einander serialisiert.
- Der Service kann entweder direkt im Code im XML definiert werden.
- Der Client kommuniziert immer über einen Proxy mit dem Service. Der Proxy kann generiert werden (Properties werden in Getter,Setter gewandelt, Listen Typinformationen gehen verloren)

### 1.7.1 Server

Listing 1: Data Transfer Objects (DTO)

---

```

1 [DataContract]
2 [KnownType(typeof(DerivedA))]
3 [KnownType(typeof(DerivedB))]
4 public class AModelClass {
5     [DataMember]
6     public string Name {get; set;}
7 }
8
9 [DataContract]
10 public class DerivedA : AModelClass {
11     [DataMember]
12     public string Name {get; set;}
13 }
14
15 [DataContract]
16 public class DerivedB : AModelClass, IInterface {
17     [DataMember]
18     public string Name {get; set;}
19 }
20
21 [DataContract]
22 public enum MyEnum {
23     [EnumMember]
24     A,
25     [EnumMember]
26     Bs
27 }

```

---

Listing 2: Service Interface

---

```

1 // (may without callback)
2 [ServiceContract(
3     CallbackContract=typeof(IMyCallback),
4     SessionMode=SessionMode.Allowed)]
5 public interface IMyServiceInterface {
6     List<AModelClass> Models {
7         [OperationContract(IsOneWay=false)]
8         get;
9     }
10
11     [OperationContract]
12     void GetModelById(int id);

```

---

```

13
14     [ServiceKnownType(typeof(DerivedB))]
15     [OperationContract]
16     List<IInterface> getDerivedB();
17 }
18
19 // Callback Interface
20 public interface IMyCallback {
21     [OperationContract(IsOneWay=true)]
22     void PassResult(AModelClass model, bool success);
23 }

```

Listing 3: Service Implementation

```

1 // Service Implementierung
2 [ServiceBehaviour(InstanceContextMode=InstanceContextMode.Single)]
3 public class MyService : IMyServiceInterface {
4     private IMyCallback callback = ...;
5     private List<AModelClass> models = new List<Models>();
6     public List<AModelClass> Models {
7         get { return models; }
8     }
9
10    public void GetModelById(int id) {
11        Model model = models.Where(m => m.id == id);
12        callback.PassResult(model, true);
13    }
14
15    public List<IInterface> getDerivedB() {
16        return new List<DerivedB>();
17    }
18 }

```

```

1 // Usage (immer die Klasse, nie das Interface!)
2 ServiceHost myHost = new ServiceHost(typeof([namespace].MyService))

```

Listing 4: Service Hosting via XML

```

1 <services>
2   <service name="[namespace].MyService">
3     <!-- Endpoint: http://localhost:8732/MyService/ -->
4     <endpoint address="" binding="basicHttpBinding"
5       contract="[namespace].IMyServiceInterface"/>
6     <!-- Endpoint: http://localhost:8732/MyService/mex -->
7     <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>
8     <host>
9       <baseAddresses>
10        <add baseAddress="http://localhost:8732/MyService/" />
11      </baseAddresses>
12    </host>
13  </service>
14 </services>

```

Listing 5: Service Hosting via Code

---

```
1 Uri address = new Uri("http://localhost:8732/MyService");
2 BasicHttpBinding binding = new BasicHttpBinding();
3
4 using(ServiceHost host = new ServiceHost(typeof([namespace].MyService), address)) {
5     host.AddServiceEndpoint(typeof([namespace].IMyServiceInterface), binding, address);
6     host.Open();
7     Console.WriteLine("Service ready");
8 }
```

---

### 1.7.2 Client

---

```
1 // name must match with xml name
2 var factory = new ChannelFactory<IMyServiceInterface>("MyService");
3 IMyServiceInterface proxy = factory.CreateChannel();
4 // use
5 proxy.GetDerivedB();
```

---

---

```
1 <xml? version="1.0"?>
2 <configuration>
3     <system.serviceModel>
4         <client>
5             <endpoint
6                 address="http://localhost:8732/MyService"
7                 binding="basicHttpBinding"
8                 contract="[namespace].IMyServiceInterface"
9                 name="MyService" />
10        </client>
11    </system.serviceModel>
12 </configuration>
```

---

## 2 .NET

- Es werden aktuell über 30 Sprachen unterstützt
- Der Source Code wird in die Microsoft Intermediate Language (MSIL: Ähnlich wie Assembler, vergleichbar mit Java Bytecode) kompiliert
- 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)

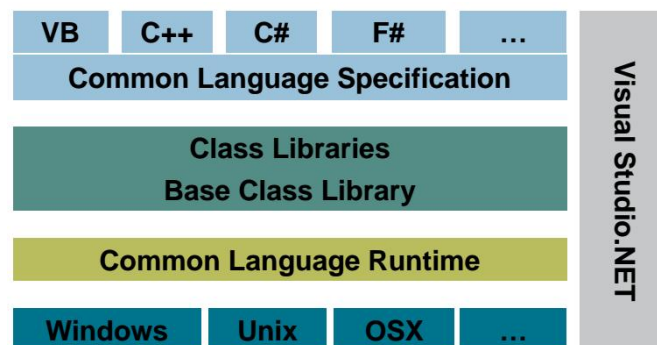


Abbildung 1: .NET Framework Architektur

### 2.1 CLR: Common Language Runtime

Die Common Language Runtime (CLR) umfasst mehrere Funktionen wie z.B. Just In Time Compilation für die Übersetzung von Intermediate Language Code in Maschinencode. Man versteht unter dem CLR ein sprachunabhängiges, abstrahiertes Betriebssystem. Es ist verantwortlich für das Memory Management, Class Loading, Garbage Collection, Exceptions, Type Checking, Code Verification des IL-Codes, Threading, Debugging und COM-Interoperabilität. Die CLR ist mit der Java VM vergleichbar.

### 2.2 CTS: Common Type System

Das Common Type System (CTS) ist ein einheitliches Typensystem für alle .NET Programmiersprachen. CTS ist integriert in CLR. Mittels Reflection ist ein programmatisches Abfragen des Typensystems möglich. (Erweiterbar mittels "Custom Attributes")

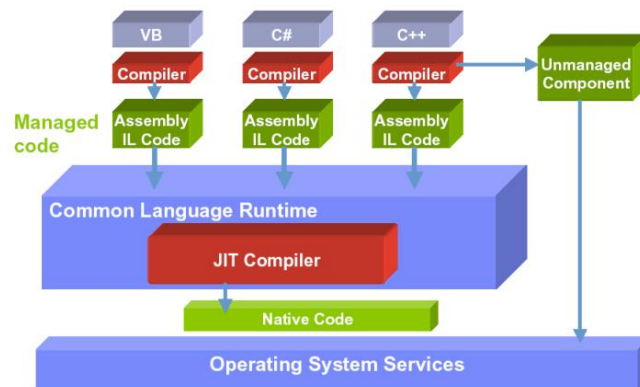


Abbildung 2: CLR: Common Language Runtime Architektur

## 2.3 CLS: Common Language Specification

Die Common Language Specification (CLS) sind allgemeine Regeln für die sprachübergreifende Entwicklung im .NET Framework. CLS kompatible Bibliotheken können in allen .NET Sprachen verwendet werden.

## 2.4 MSIL: Microsoft Intermediate Language

Microsoft Intermediate Language (MSIL) ist eine **prozessor-, und sprachunabhängige** Zwischensprache die Assembler ähnelt.

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

### Vorteile

- Portabilität
- Typsicherheit: Beim Laden des Codes können Typensicherheits und Security Checks durchgeführt werden.

### Nachteile

- Performance (kann verbessert werden, wenn JIT Compiler prozessorabhängige Hardwarebeschleunigung nutzt.)

## 2.5 JIT: Just in Time Compilation

Bei der JIT Kompilierung wird die aufgerufene Methode vor dem Methodenaufruf kompiliert und der IL-Code durch nativen Code ersetzt. Es gibt drei Typen von JIT-Compilern:

- Pre-JIT: Gesamter Code vor Ausführung (z.B mit NGEN)
- Normal-JIT (Siehe Diagramm)
- Econo-JIT: Wie Normal-JIT, aber mit Cleanup.



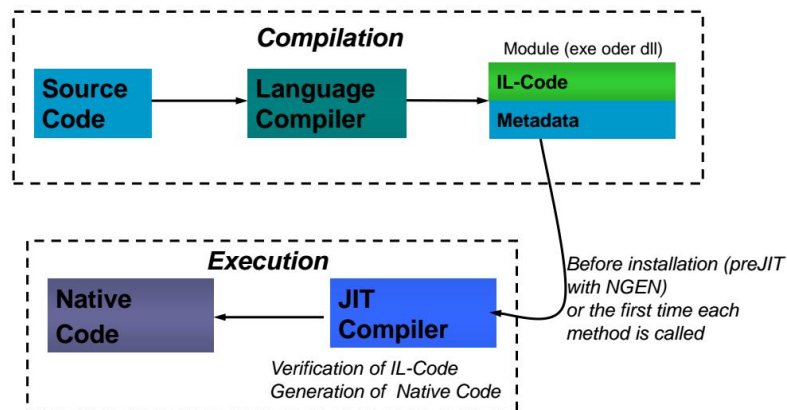
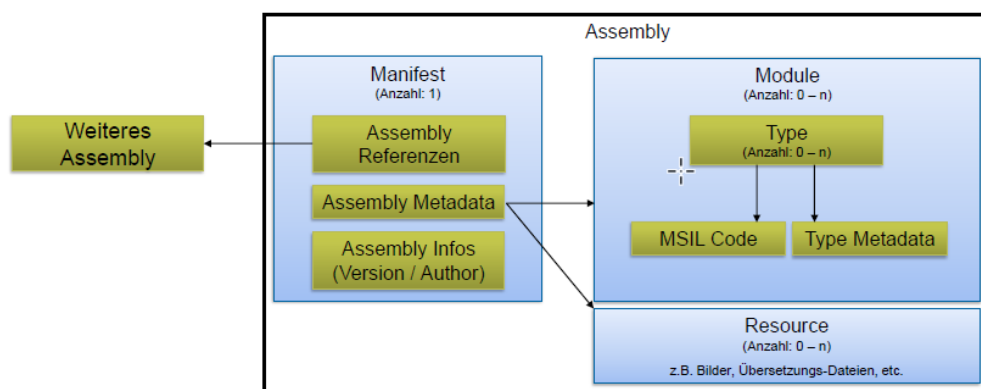


Abbildung 3: MSIL Kompilierung

## 2.6 Assembly / Komponenten

Assembly=selbstbeschreibende Komponente mit definierter Schnittstelle



Kompilation erzeugt Assemblies und Module

Abbildung 4: Assembly Übersicht

Ein Assembly kann mit einem JAR File verglichen werden. Ein Assembly enthält MSIL-Code, Typ und Assembly Metadaten, Manifest mit strong names (Version/Author) und Referenzen auf andere Assemblies. Ein Assembly **kann aus mehreren Modulen bestehen, standardmässig** enthält ein Assembly aber **genau ein Modul**. Assemblies können nicht geschachtelt werden!

**Private Assembly** Private Assembly werden über einen Dateipfad referenziert und sind ansonsten nirgends registriert. Sie werden meist nur von einer Applikation genutzt.

**Shared Assembly** Shared Assemblies verfügen über einen Strong Name (eindeutige Bezeichnung: Bez, Version, Culture, Public Key) und liegen im Global Assembly Cache (GAC). Ein Shared Assembly steht allen Applikationen zur Verfügung. Es sollte nicht zu viele Versionen im GAC registriert werden. (DLL Hell). Für die registriert wird das Command Line Tool `gacutil.exe` verwendet.

### 2.6.1 Module

Die Kompilation erzeugt ein Modul mit Code / MSIL und Metadaten. Die Metadaten beschreiben alle Aspekte des Codes ausser der Programmlogik. (Klassen, Methoden und Feld Definitionen) Diese Metadaten können mit Reflektion abgefragt werden. Die Metadaten werden von Analysetools (IL Dissassembler), IDEs (IntelliSense, Object-Browser) als auch von der CLR (Typsicherheitverifikation, Memory Management, JIT Compilation) verwendet.

### 2.6.2 References

Referenzen zeigen auf eine externe Library.

Referenzen werden beim CSC mittels `csc /target:exe /r:MyDLL.dll Program.cs` eingefügt. Es gibt verschiedene Arten:

- Vorkompiliertes Assembly
  - Im File System
  - Debugging nicht verfügbar
  - Navigation nur auf Metadaten-Ebenen
- NuGet package
  - Externe Dependency (nuget.org)
  - Debugging nicht verfügbar
  - Navigation nur auf Metadaten-Ebenen
- Visual Studio Projekt
  - In gleicher Solution vorhanden
  - Debugging und Navigation verfügbar
- .NET Core oder .NET Standard SDK
  - Zwingend
  - Normalerweise: "Microsoft.NETCore.App"
  - Bei .NET Standard "NETStandard.Library"

### 2.6.3 Packages

.NET wird in kleineren Packages ausgeliefert und ist somit kein Monolithisches Framework mehr. Es wird aufgeteilt in diverse NuGet Packages. Dies erlaubt unterschiedliche Releasezyklen, erhöhung der Kompatibilität und kleinere Deployment-Einheiten. Zu den wichtigen Packages gehören System.Runtime, System.Collections, System.Net.Http, System.IO.FileSystem, System.Linq und System.Reflection.

#### NuGet

Die \*.nupkg Datei enthält alle Libraries in mehreren Versionen sowie die Manifest/Metadaten (Package Identifier, Titel, Beschreibung, Versions-Informationen, Dependencies, etc.). Es ist als Zip-Datei gespeichert.

Jeder kann Packages in der NuGet Gallery ([www.nuget.org](http://www.nuget.org)) veröffentlichen.

## 2.7 Kompilierung

Zur Kompilierung wird der CSharp Compiler (CSC) verwendet.

---

```
1 // Create Executable: ClassA.exe
2 csc.exe /target:exe ClassA.cs
3
4 // Create Lib: ClassA.dll
5 csc.exe /target:library ClassA.cs
6
7 // Create Executable, referencing a Lib
8 csc.exe /target:exe
9     /out:Programm.exe
10    /r:ClassA.dll // or /r:System.Windows.Forms.dll (GAC)
11    ClassB.cs ClassC.cs
12 // Ergibt = Program.exe
```

---

## 2.8 Garbage Collection

Der Garbage Collector löscht Objekte auf dem Heap, die nicht mehr über eine Root-Referenz referenziert werden. (Mark and Sweep) Wie in Java weiss man nicht wenn der GC aufgerufen wird (**nicht deterministisch**). Er kann aber mit der Methode GC.Collect() manuell aufgerufen werden. Der Ablauf ist immer gleich:

1. Alle Objekte als Garbage betrachten
2. Alle reachable Objekte markieren
3. Alle nicht markierten Objete freigeben
4. Speicher kompaktieren

Die Garbage Collection started, sobald eine dieser Bedingungen wahr ist

- System hat zu wenig Arbeitsspeicher
- Allozierte Objekte im Heap übersteigen einen Schwellwert
- GC.Collect Methode wird aufgerufen.

**Root Referenzen** Root-Referenzen sind statische Felder und aktive lokale Variablen auf dem Stack.

### 2.8.1 Generationen

Objekte werden in drei Generationen aufgeteilt: Zuerst werden die Objekte der 0ten Generation abgeräumt.

- Generation 0: Objekte wurden seit dem letzten GC Durchlauf neu erstellt (z.B lokale Variablen)
- Generation 1: Objekte die einen GC Durchlauf überlebt haben (z.B Members)
- Generation 2: Objekte die mehr als einen GC Durchlauf überlebt haben.

### 2.8.2 Deterministic Finalization

Objekte sollten wenn nötig mit dem Interface `IDisposable` und der `void Dispose()` Methode finalisiert werden und nur wenn nötig mit einem Destruktor. Man spricht von Deterministic Finalization, wenn der Programmierer für die Freigabe der unmanaged Ressourcen zuständig ist und diese explizit über `Dispose()` freigibt. Dazu muss die `Dispose()` Methode überschrieben werden. Mit `using` wird der Aufruf von `Dispose()` implizit sichergestellt. Deterministic Finalization sollte bei allen I/O Klassen verwendet werden.

- Dateisystem Zugriffe
- Netzwerk Kommunikation
- Datenbank Anbindung

---

```

1 public class DataAccess : IDisposable {
2     private DbConnection connection;
3     public DataAccess() {
4         connection = new SqlConnection();
5     }
6
7     ~DataAccess() {
8         // backup
9         connection.Dispose();
10    }
11
12    public void Dispose() {
13        // suppress GC, as we just want to call dispose
14        System.GC.SuppressFinalize(this);
15        connection.Dispose();
16        // Call base.Dispose(); if necessary
17    }
18 }
19
20 using (DataAccess dataAccess = new DataAccess()) {
21     // work with dataAccess
22 }

```

---

### 2.8.3 Finalizer

Der Gebrauch von herkömmlichen Finalizer ist nicht deterministisch (man weiss nicht wann der GC aufgerufen wird). Der Garbage Collector arbeitet viel effizienter wenn kein Destruktor/Finalizer vorhanden ist. Einflüsse auf den GC Aufruf sind folgende:

- Gerade verfügbarem Speicher
- Generation des aktuellen Objektes
- Reihenfolge in der Finalization Queue
- Manuell oder automatisch getriggert
- Kann auch abhängig von der .NET Runtime Version sein

### 2.8.4 Object Pinning

Der GC kompaktiert Speicher bei Bedarf. Mit dem Keyword `fixed` kann dies unterbunden werden. (schlechte Performance)

### 2.8.5 Weak References

Wird eine strong Referenz (default) auf null gesetzt, wird es irgendwann vom GC abgeräumt. Auf das null objekt kann nicht mehr zugegriffen werden. Mit Weak Referenzen kann man immer noch auf das Objekt zugreifen, bis es vom GC abgeräumt wird. Mit der Methode `TryGetTarget(out sr)` kann man auf das alte Objekt zugreifen und dieses wiederherstellen. Wurde das Objekt abgeräumt, muss es neu erstellt werden.

### 2.8.6 Memory Leaks

Memory Leaks entstehen, wenn z.B ein Event Listener nicht abgeräumt wird. Objekte welche aus einer anonymen Methode oder Lamda Ausdruck innerhalb eines Event Listener noch referenziert werden, werden nicht abgeräumt. Gleiches gilt für alle IDisposable Objekte, bei denen `Dispose()` nicht aufgerufen wurde. (z.B DB Connection)

---

```

1  // interface
2  public interface IDisposable {
3      void Dispose();
4  }
5
6  // deterministic finalization
7  public class DataAccess : IDisposable {
8      private DbConnection connection;
9      public DataAccess() {
10         connection = new SqlConnection();
11     }
12
13     ~DataAccess() {
14         connection.Dispose();
15     }
16
17     public void Dispose() {
18         System.GC.SuppressFinalize(this);
19         connection.Dispose();
20     }
21 }
22
23 class MyClass {
24     // call disposal
25     DataAccess dataAccess = new DataAccess() ;
26     dataAccess.Dispose();
27
28     // implicit Disposal call with using
29     // Multiple usings possible
30     // syntactic sugar, compiles to try-finally with Dispose call
31     using (DataAccess dataAccess = new DataAccess())
32     using (SQLParser parser = new SQLParser()) {
33         ..
34     }
35
36     // or with same type
37     using (DataAccess da1 = new DataAccess(), DataAccess da2 = new DataAccess()) {
38         ..
39     }
40 }

```

---

### 3 .NET Standard

Der .NET Standard bietet die Brücke zwischen den ehemaligen Versionen .NET Framework (Windows) und .NET Core (Universal). Wobei es inzwischen nur noch .NET gibt. Es bietet die gemeinsamen Schnittstellen an, die bei beiden Versionen verwendet werden können. Es werden die minimal zu unterstützten APIs (Klassen und Methoden) pro Version definiert. Als Entwickler kann man sein package kompatibel zu einer bestimmten Standard Version machen. Diese Libraries sind dann Cross-Plattform. Man hält die Fragmentierung der Framework minimal und reduziert Pre-Compiler-Anweisungen.

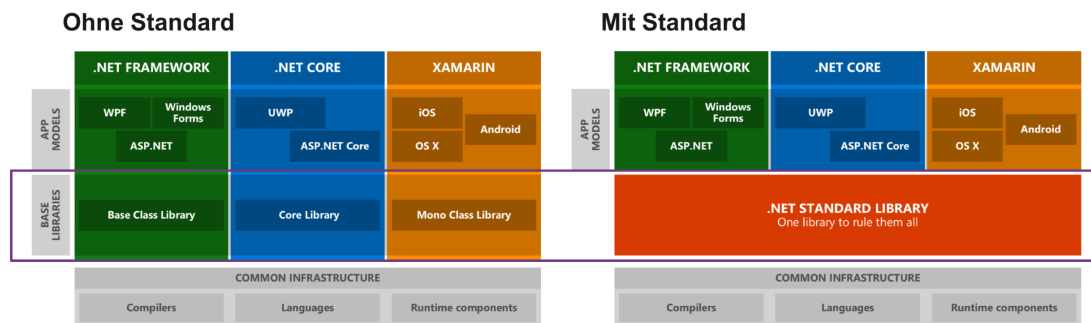


Abbildung 5: Vergleich ohne und mit .NET Standard

Je höher die Version desto mehr .NET Apis, desto tiefer, einfacher einzubinden. Jede .NET Implementation unterstützt eine andere maximale .NET Standard version. So wird beispielsweise .NET Standard 2.1 von .NET Framework nicht mehr supported, sondern nur noch von .NET Core 3.0 und .NET.

.NET Standard	1.0 <sup>1</sup>	1.1 <sup>1</sup>	1.2 <sup>1</sup>	1.3 <sup>1</sup>	1.4 <sup>1</sup>	1.5 <sup>1</sup>	1.6 <sup>1</sup>	2.0 <sup>2</sup>	2.1 <sup>3</sup>
.NET	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework <sup>1</sup>	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 <sup>2</sup>	4.6.1 <sup>2</sup>	4.6.1 <sup>2</sup>	N/A <sup>3</sup>
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	TBD
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2021.2.0b6

Abbildung 6: Übersicht der .NET-Implementation und deren kompatiblen .NET Standard Versionen

## 4 Command Line Interface CLI

Die CLI ist Teil des .NET Core SDK. Es ist die Basis für die high-level Tools (Visual Studio, Rider, etc.) Aufgerufen wird es mit `dotnet[.exe] <Verb> <argument> --<option> <param>`.

### 4.1 Kommandos

**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 output(s).

**sln** Modify solution (SLN) files.

**add** Add reference to 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.

## 5 Visual Studio 22

### 5.1 Solution

Eine Solution besteht aus mehreren Projekten.

### 5.2 Umbenennen

Folgende Objekte müssen manuell umbenannt werden

- Ordner in der das Projekt liegt
  1. Manuelle Anpassung des Ordner Names in File-System
  2. Manuelles Anpassen der \*.sln-Datei
- Name des Assemblies
  - Rechts-Klick auf Projekt > Properties > Application > Assembly name
- Name des Default Namespaces (wird bei neuen Classen verwendet)
  - Rechts-Klick auf Projekt > Properties > Application > Default namespace

### 5.3 Ordnerstruktur

Jeder Projektordner enthält folgende zwei Verzeichnisse

**bin\<BuildKonfiguration>**

Beinhaltet das fertige, gelinkte Kompilat

**obj\<BuildKonfiguration>**

Beihaltet Files welche während der Kompilierung erzeugt werden und für die Erstellung eines Assemblies nötig sind.

### 5.4 Projekt-Dateien

Die Projekte werden als XML-Datei verwaltet in einer \*.csproj Datei. Es beschreibt was alles Kompiliert werden muss, etc. Die Projektdateien werden von Build Engines intepretiert. Es gibt diverse Gruppen. Da gibt es Property-Groups (Settings), Item-Groups (Zu kompiliertende Items), Target-Groups (Weitere Buildsteps)



## 6 C# Grundlagen

### 6.1 Unterschiede zu Java

- Es gibt Structs, welche wie Klassen sind (jedoch Wertetypen)
- Es gibt Properties (spez. Getter und Setter) und Indexer (erweiterter Array Zugriff)
- Andere Syntax bei den Konstruktoren
- Es gibt Operator Overloading
- Parameterübergabe kann explizit by value oder by reference sein (auch für Wertetypen)
- Es gibt partielle Klassen und Methoden für Generatoren
- Es heisst `NullReferenceException` und nicht `NullPointerException`
- Es heisst `base` und nicht `super`
- Konstruktorparameter können direkt dem Parent übergeben werden. (`public Derived(int x): base(x){ .. }`)

### 6.2 Naming Conventions

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

Tabelle 1: Naming Conventions

### 6.3 Sichtbarkeiten

- Abgeleitete Klasse/Interfaces dürfen nicht die grössere Sichtbarkeit als ihren Basistyp haben (z.B Parent "internal" und Sub "public")
- Member Typen müssen mindestens gleich sichtbar wie der Typ selbst sein
- Standard Sichtbarkeit ist `internal`
- Interface Member dürfen keine Angaben zur Sichtbarkeit haben.

Attribut	Beschreibung
public	Überall sichtbar
private	Innerhalb des jeweiligen Typen sichtbar (Klasse/Struct)
protected	Innerhalb des jeweiligen Typen oder abgeleiteten Klasse sichtbar (Klasse/Struct)
internal	Innerhalb des jeweiligen Assemblies sichtbar
protected internal	Kombination aus internal und protected
private protected*	Innerhalb des jeweiligen Typen oder abgeleiteter Klasse sichtbar, wenn diese im gleichem Ass

Tabelle 2: Sichtbarkeiten

Typ	Sichtbarkeit	Member (default)	Member (zulässig)
class	public, internal(default)	private	public, protected, internal, private, protected internal, private protected
struct	public, internal(default)	private	public, internal, private
enum	public, internal(default)	public	-
interface	public, internal(default)	public	-
delegate	public, internal(default)	-	-

Tabelle 3: Standard Sichtbarkeiten von Typen

## 6.4 Operatoren

Category (by precedence)	Operator(s)	Associativity
Primary	x.y f(x) a[x] x++ x-- new typeof default checked:	left
Unary	+ - ! ~ ++x --x (T)x	right
Multiplicative	* / %	left
Additive	+ -	left
Shift	<< >>	left
Relational	< > <= >= is as	left
Equality	== !=	right
Logical AND	&	left
Logical XOR	^	left
Logical OR		left
Conditional AND	&&	left
Conditional OR		left
Null Coalescing	??	left
Ternary	?:	right
Assignment	= *= /= %= += -= <<= >>= &= ^=  = =>	right

Abbildung 7: Operatoren Präzedenz

## 6.5 Pre-, Post-Inkrmenet

---

```
1 // post increment
2 int a = 1;
3 int b = a++; // a=2, b=1
4
5 // pre increment
6 a = 1;
7 b = ++a; // a=2, b=2
```

---

## 6.6 Statements

### 6.6.1 If Else If Else

---

```
1 if () {
2 } else if () {
3 } else {}
```

---

### 6.6.2 Switch Case

---

```
1 switch() {
2 case:
3 case: break;
4 }
```

---

### 6.6.3 Loops

---

```
1 while() {}
2 do {} while ();
3 for (int = 1; i <= myList.Count(); i++) {}
4 foreach(int x in y);
```

---

### 6.6.4 Kommentare

---

```
1 // Single Line Comment
2 /* Multiline Comment */
3 /// Dokumentation
```

---

## 6.7 Datentypen

Numerische Datentypen können einen der folgenden Literale haben

Literal	Typ	Java	Wertebereich
sbyte	System.SByte	byte	-128 .. 127
byte	System.Byte	---	0 .. 255
short	System.Int16	short	-32768 .. 32767
ushort	System.UInt16	---	0 .. 65535
int	System.Int32	int	-2147483648 .. 2147483647
uint	System.UInt32	---	0 .. 4294967295
long	System.Int64	long	$-2^{63} .. 2^{63}-1$
ulong	System.UInt64	---	$0 .. 2^{64}-1$
float	System.Single	float	$\pm 1.5E-45 .. \pm 3.4E38$ (32 Bit)
double	System.Double	double	$\pm 5E-324 .. \pm 1.7E308$ (64 Bit)
decimal	System.Decimal	---	$\pm 1E-28 .. \pm 7.9E28$ (128 Bit)
bool	System.Boolean	boolean	true, false
char	System.Char	char	Unicode-Zeichen

Abbildung 8: Primitive Typen

- u/U: unsigned (signed Variablen können nur mit einem cast einer unsigned Variablen zugewiesen werden)
- l/L: long
- f/F: float

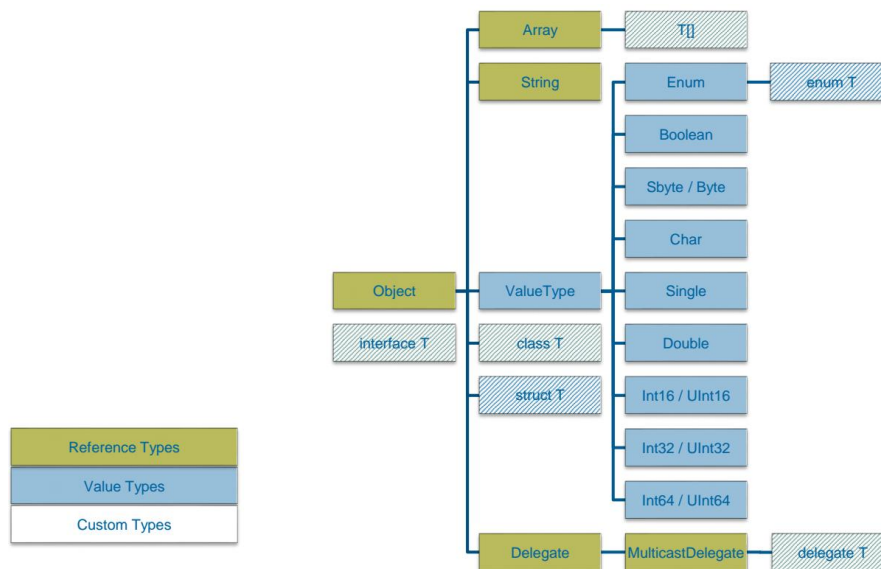
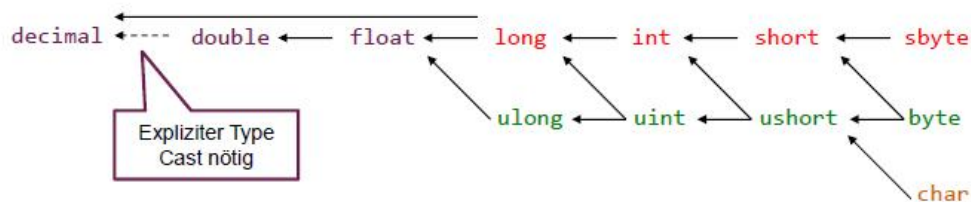


Abbildung 9: Datentypen

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

Abbildung 10: Default Values

### 6.7.1 Casts



#### ■ Erlaubt

- `intVariable = shortVariable`
- `intVariable = charVariable`
- `floatVariable = charVariable`
- `decimalVariable = (decimal)doubleVariable`
- `byteVariable = (byte)longVariable`
- [...]

#### ■ Nicht erlaubt

- `shortVariable = intVariable`
- `charVariable = intVariable`
- `charVariable = floatVariable`
- [...]

Abbildung 11: Casts

### 6.7.2 Reference Types / Referenztypen

- Sind auf dem Heap gespeichert, wobei die Variable an sich auf dem Stack liegt
- Die Referenzen werden automatisch vom Garbage Collector aufgeräumt
- Wird ein Reference Type einer Methode übergeben, wird die Objekt referenz kopiert. (sofern nicht **ref**)

### 6.7.3 Value Types / Werttypen

- Sind auf dem Stack gespeichert
- Primitive Datentypen, Struct und System.Enum
- Wird eine Value Type Variable einer weiteren Value Type Variable zugewiesen, wird der Wert kopiert. Gleiches gilt für die Methodenparameter by Value.

```
PointRef a = new PointRef();
a.x = 12;
a.y = 24;
PointRef b = a;
b.x = 9;
Console.WriteLine(a.x); // Prints 9
```

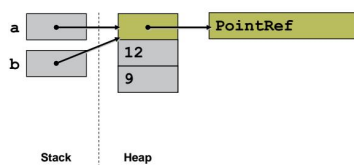


Abbildung 12: (Referenztypen)

```
PointVal a = new PointVal();
a.x = 12;
a.y = 24;
PointVal b = a;
b.x = 9;
Console.WriteLine(a.x); // Prints 12
Console.WriteLine(b.x); // Prints 9
```

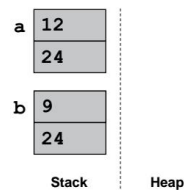


Abbildung 13: (Werttypen)

## 6.8 Nullable Types

- Der ? Operator erlaubt es Null Werte einem Wertetyp zuzuweisen. Der Typ ist dann `Nullable<T>`
- Arithmetisch Ausdrücke mit Null ergeben immer `null`
- Vergleiche mit Null sind immer `false`. Ausnahme `null == null`
- Der ?? Operator erlaubt es einen Default Wert anzugeben, falls die Variable leer ist. Der zurückgegebene Typ ist dann kein Nullable-Type mehr(z.B `int`)

---

```

1  int a = 0;
2  bool b = false;
3  int? c = 10;
4  int? d = null;
5  int? e = null;
6
7  c + a // 10, typeof int?
8  a + null // null
9  a < c //true
10 a + null < c // false
11 a > null // false
12 (a + c - e) * 9898 + 1000 // null
13 d // null
14 d == d // true
15 c ?? 1000 // wenn null dann 1000 ansonsten value --> gibt 10
16 d ?? 1000 // wenn null dann 1000 ansonsten value --> gibt 1000 (weil d == null)
17
18 -----
19
20 int a = 1;
21 int? b = 2;
22 int? c = null;
23
24 a+1; // 2
25 a+b; // 3
26 a+c; // null
27 a < b; // True
28 a < c; // False
29 a + null; // null
30 a + null < b; // False
31 a + null < c; // False
32 a + null == c; // True
33
34 -----
35 // Sicherer MethodChaining:
36 string s = GetNullableInt()?.ToString(); // Liefert null, wenn variable links null,
    ansonsten string
37 // Sicherer Delegate Aufruf:
38 Action a = Console.WriteLine;
39 a?.Invoke(); // ruft delegate auf, wenn a != null
40 //Typprüfung
41 myVar is Type<T> // liefert bool
42 // Casts identisch wie bei normalen Typen
43 (Type<T>)myVar // liefert Type<T>
44 myVar as Type<T> // liefert Type<T>

```

---



## 6.9 Boxing / Unboxing

Beim Boxing werden Value Typen implizit in Referenztypen konvertiert. Das Unboxing erfolgt immer explizit.

---

```

1 // boxing
2 int i = 123;
3 object o = i;
4
5 // unboxing
6 o = 123;
7 i = (int) o;

```

---

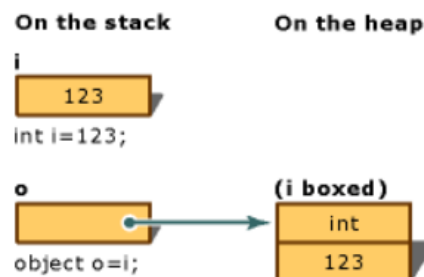


Abbildung 14: Boxing

## 6.10 Object

- **object** ist ein Alias für System.Object. Es ist die Basisklasse aller Typen.

## 6.11 String

- **string** ist ein Alias für System.String
- String ist ein Reference Type
- Wie in Java ist ein String **nicht modifizierbar**, jedoch Verkettung mit + möglich
- Mit dem @ vor dem String Literal kann der String Sonderzeichen enthalten, die nicht escaped werden müssen.

---

```

1 // Escape: The "File" can be \t found at \\server\share
2 @"The ""File"" can be \t found at \\server\share"
3
4 // Formatieren (konkatenieren)
5 string f = string.Format("A={0} and B={1}", a, b);
6
7 // Kopieren
8 string s2 = string.Copy(s1);
9
10 // Vergleichen
11 s1.Equals(s2) // Inhalt wird verglichen, nicht die Referenz
12 s1 == s2 // Inhalt wird verglichen, nicht die Referenz
13 s1.CompareTo(s2); // -1, 0, +1
14 string.ReferenceEquals(s1, s2); // Achtung: String Pooling, nach Copy = False

```

---

## 6.12 Arrays

Einfachste Datenstruktur für Listen, bei welcher die Länge aller Dimensionen bei der Instanzierung bekannt ist. Wie in anderen Sprachen ist das Array zero-based(Index: [0 - (n-1)]), ein Referenztyp (zeigt auf Heap) und alle Werte nach Instanzierung initialisiert(false, 0, null, etc.).

---

```

1  int[] array1 = new int[5]; // deklaration value type
2  int[] array2 = new int[] {1,2,3,4,5}; // deklaration & wertedefinition
3  int[] array3 = int[] {1,2,3,4,5,6}; // vereinfachte syntax ohne new
4  int[] array4 = {1,2,3,4,5,6}; // vereinfachte syntax ohne Typ / new
5  object[] array5 = new object[5]; // deklaration ref type
6  array1.Length // Get Length
7
8  // Blockmatrizen (Mehrdimensionale Arrays (rechteckig)) (Speichereffizienter,
   // schnelleres Allokieren, schnellere Garbage Collection und vermeintlich schneller
   // im Zugriff (Boundary Check wird nur bei 1-dimensionalen Array optimiert))
9  int[,] array1 = new int[3,2]; // Deklaration
10 int length = array1.Length; // Liefert 6
11 int length0 = array1.GetLength(0); // Liefert 3 (Laenge 0. Dimension)
12 int length1 = array1.GetLength(1); // Liefert 2 (Laenge 1. Dimension)
13 int[,] multiDim2 = { {1,2,3} , {4,5,6} }; // Deklaration & Wertdefinition
14
15 // Jagged Arrays (ausgefranst)
16 int[][] jaggedArray = new int[6][]; // Deklaration
17 jaggedArray[0] = new int[4] { 1, 2, 3, 4 } // Wertdefinition

```

---

## 6.13 Indexer

Ein Indexer erlaubt einfachen Zugriff auf ein Array. Er wird mit dem Keyword **this** erstellt.

---

```

1  class BookList {
2      private string[,] books ={{},{},{}};
3
4      public string this[int i1, int i2] {
5          get { return books[i1, i2]; }
6          set { books[i1, i2] = value; }
7      }
8  }
9
10 // access
11 bookList[0, 0]

```

---

## 6.14 List

---

```

1  var myList = new List<int>() { 1, 2, 3, 4, 5 }; // using System.Collections.Generic;
2  myList.Count(); // Or Property when var x = myList.Count;
3  myList.Add(6);
4  myList.Remove(4);
5  myList.Contains(6); // True
6  myList.ForEach(n => Console.WriteLine(n)); // 1,2,3,5,6
7  myList.Clear();
8  myList[4];
9  myList.IndexOf(4);

```

---

## 6.15 Namespaces

Namespaces entspricht dem Package in Java und lässt den Code hierarchisch strukturieren. Ein Namespace ist nicht an die physische Struktur gebunden (in Java schon). Ein file kann mehrere Namespaces beinhalten und ein Namespace kann in verschiedenen Files definiert sein. Er kann andere Namespaces, Klassen, Interfaces, Structs, Enums und Delegates enthalten.

---

```

1 namespace A {
2     using C;
3     public class A : Base {
4         C.externMethod();
5     }
6 }
```

---

Neu, wenn nur ein Namespace in einer Datei folgende Schreibweise präferiert. Spart in jeder Zeile 4 Zeichen (File-scoped Namespace). Nur ein Namespace pro File ist so dann erlaubt:

---

```

1 namespace A;
2 //.... Code ....
```

---

Namespace können auch mittels Alias-Namen geladen werden:

---

```

1 using F = System.Windows.Forms;
2 //.... Code ....
3 F.Button b;
```

---

Weiter gibt es auch globale Imports (Meist Datei GlobalUsings.cs). Dies geht auch implizit via \*csproj-Datei. Nicht jede SDK supported diese Funktion:

---

```

1 using static Azure.Core;
```

---

## 6.16 Main Methode

Die Main Methode ist die Einstiegspunkt in die Anwendung. Sie ist zwingend für Executables (Console Application, Windows Application, etc.). Darf genau einmal Vorkommen. Wenn es mehrere gibt, muss mann in der csproj-Datei explizit angeben, welche verwendet werden soll. Sie befindet sich meist in der Datei Program.cs. Sie muss folgende Signatur haben:

---

```

1 // Examles
2 static void Main() { }
3 static int Main() { }
4 static void Main(string[] args) { }
5 static int Main(string[] args) { }
6 static async Task Main() { }
7 static async Task<int> Main() { }
8 static async Task Main(string[] args) { }
9 static async Task<int> Main(string[] args) { }
```

---

Auf Arbumente kann unterschiedlich zugegriffen werden. Per Default greift man über ein string[]-Parameter zu. Ohne diesen Parameter ist es auch möglich über die statische Methode System.Environment.GetCommandLineArgs(); darauf zuzugreifen. Neu kann auch die Main-Methode als entry-Point weggelassen werden dank Top-level Statements. Die Regeln sind folgende:

- Nur 1x pro Assembly erlaubt
- Argumente heissen fix args

- Exit Codes erlaubt: `return someIntValue;`
- VOR dem top-level Statements können usings definiert werden
- NACH dem top-level Statements können Typen definiert werden

## 7 Variablen und Properties

### 7.1 Konstanten

Der Wert einer Konstante muss zur Compilezeit verfügbar sein.

```
1 const long size = int.MaxValue;
```

### 7.2 ReadOnly

ReadOnly Felder müssen in der Deklaration oder im Konstruktor initialisiert werden. ReadOnly Variablen sind äquivalent mit Java final Felder

```
1 readonly DateTime date1 = DateTime.Now;
2
3 class Test {
4     private readonly int myProp;
5     public int MyProp {
6         get { return myProp; }
7     }
8
9     public Test() {
10         myProp = 42;
11     }
12 }
```

### 7.3 Properties

Eine Property ist ein Wrapper um Getter und Setter. Get und Set können einzeln weggelassen werden. (read-only, write-only) Bei Set besteht zudem die Möglichkeit das Flag private zu setzen.

```
1 // Backing Field
2 private int lenght;
3
4 public int Length {
5     get { return lenght; }
6     // private is optional
7     private set { lenght = value; }
8 }
9
10 MyClass mc = new MyClass();
11 mc.Length = 12;
12 int length= mc.Length;
```

#### 7.3.1 Auto Properties

Bei Auto Properties wird das Backing Field sowie die zugehörigen Getter und Setter automatisch generiert.

```
1 // Auto Property: Backing field is auto generated
2 public int LengthAuto { get; set; }
3 public int LengthInitializes {get; /* set; */ } = 5;
```

### 7.3.2 Properties direkt initialisieren

Properties können bei der Objekt erstellung direkt initialisiert werden.

---

```
1 class MyClass
2 {
3     private int length; // Backing-Field
4
5     // Property
6     public int Length
7     {
8         get {return length; }
9         set { length = value; }
10    }
11    public int Width { get; set; }
12 }
13
14 MyClass mc = new MyClass() {
15     Length = 1;
16     Width = 2;
17 }
```

---

### 7.3.3 Abstrakte Properties/Indexes

Abstrakte Properties/Indexes haben kein Anweisungsteil. Get und Set werden analog der Auto Properties mit einem Semikolon abgeschlossen. Wichtig ist das bei der Implementation Get und Set Kombination identisch sein muss.

## 8 Methoden

Im C# hat man zwei verschiedene Ausprägungen von Methoden:

- Prozedur/Aktion: ohne Rückgabewert
- Funktion: mit Rückgabewert

### 8.1 Overloading

Methoden können **überladen** werden. (Unterschiedliche Anzahl Parameter, Unterschiedliche Typen, Unterschiedliche Parametertypen (ref/out) aber immer gleicher Name). Rückgabewert ist **kein** Unterscheidungsmerkmal.

---

```

1 public static void Foo(int x);
2 public static void Foo(doubly y);
3 public static void Foo(int x, int y);
4 public static void Foo(params int[] x); // params array = normales array
5
6 // sollte man nicht machen. Design Problem!
7 public static void Foo(int ref x);
8 public static void Foo(int out x);

```

---

### 8.2 Call by value

Es wird eine Kopie des Stack Inhalts übergeben

---

```

1 void IncVal(int x){x = x + 1;}
2 int val = 3;
3 IncVal(val); // val = 3;

```

---

### 8.3 Call by reference

Adresse der Variable wird übergeben. Mit dem **ref** Keyword können auch Werttypen als Referenz übergeben werden. Wichtig die Variable muss zuerst initialisiert werden.

---

```

1 void IncRef(ref int x) {x++; }
2 int value=3; //value must be initialized first
3 IncRef(ref value); // pass reference, value = 4;

```

---

### 8.4 Out Parameter

Das **out** Keyword erlaubt es Werte by Reference zu übergeben. Es funktioniert wie das **ref** Keyword, mit dem Unterschied, dass die Variable nicht im Vorhinein initialisiert werden muss. Es ist ebenfalls möglich, die Variable mit dem **out** direkt beim Methodenaufruf zu deklarieren. Das **out** Keyword muss beim Aufrufer und bei der Methode deklariert werden.

---

```

1 static void Init(out int a) {
2     a = 10;
3 }
4 // usage
5 int value1
6 Init(out value1) //value1 = 10;
7 // declaration in method call
8 Init(out int value2); // value2 is now 10

```

---

## 8.5 Params Array

Erlaubt beliebig viele Parameter. Das params Array muss am Ende der Deklaration stehen.

---

```
1 void Sum(out int sum, params int[] values) { .. }
2 Sum(out sum2, 1,2,3,4);
```

---

## 8.6 Optionale Parameter (Default Values)

Erlaubt ermöglicht Zuweisung eines Default Values. Die Optionalen Parameter dürfen erst am Schluss deklariert werden. Default Werte können bei **out** und **ref** Parameter nicht verwendet werden.

---

```
1 private void Sort(
2     int[] array,           // Erforderlich
3     int from = 0,         // Optional
4     int to = -1,          // Optional
5     bool ascending = true, // Optional
6     bool ignoreCase = false // Optional
7 ){ .. }
```

---

## 8.7 Named Parameter

Optionale Parameter können über den Namen identifiziert und übergeben werden.

---

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

---

## 8.8 Virtual

Bei C# wird alles statisch gebunden. Mit dem Keyword **virtual**, wird dynamisch gebunden. Bei einer virtuellen Methode wird deshalb die überschriebene Methode in der Subklasse aufgerufen. Virtual kann nicht mit folgenden Keywords verwendet werden.

- **static**
- **abstract** (implizit virtual)
- **override** (implizit virtual)

## 8.9 Override

Mit dem Keyword **override** können **virtual** Methoden überschrieben werden. Die Signatur muss dabei identisch sein. Man spricht von dynamischem Binding.

---

```
1 public class Base {
2     public virtual void Invoke() {
3         Console.WriteLine("Base");
4     }
5 }
6 public class Derived : Base {
7     public override void Invoke() {
8         Console.WriteLine("Derived");
9     }
10 }
11
```

---



---

```

12 Base a = new Base();
13 Base b = new Derived();
14 Derived c = new Derived();
15
16 a.Invoke(); // base
17 b.Invoke(); // derived
18 c.Invoke(); // derived

```

---

## 8.10 Methoden überdecken mit New

Mit **new** weiss der Compiler, dass der Member bewusst überdeckt wurde. Man spricht von **statischem Binding**. Es wird immer die Methode des statischen Typs ausgeführt. New kann **nicht** mit **override** verwendet werden, jedoch mit **virtual**.

---

```

1 public class Base {
2     public void Invoke() {
3         Console.WriteLine("Base");
4     }
5 }
6 public class Derived : Base {
7     public new void Invoke() {
8         Console.WriteLine("Derived");
9     }
10 }
11
12 Base a = new Base();
13 Base b = new Derived();
14 Derived c = new Derived();
15
16 a.Invoke(); // base
17 b.Invoke(); // base
18 c.Invoke(); // derived

```

---

## 8.11 Dynamic Binding

Man spricht von Dynamic Binding, wenn die Methoden des dynamischen Typs aufgerufen werden. Dazu gibt es ein vereinfachtes Regelwerk:

- Falls der dynamische Typ konkreter als der statische Typ und die Methode **virtual** ist.
- Suche Vererbungs-Hierarchie von oben nach unten nach konkreter Methode mit Schlüsselwort **override**

## 8.12 Abstrakte Methoden

Abstrakte Methoden haben statt dem Anweisungsteil ein Semikolon und sind implizit **virtual**, dürfen jedoch **nicht static** oder **virtual** sein.

---

```

1 abstract class Sequence { public abstract void Add(object x); }

```

---

## 8.13 Sealed

Mit **sealed** weiss der Compiler, dass die Methode (kein Overriding) oder Klasse (keine Vererbung) nicht mehr verändert wird. Bei der Methode muss es in Kombination mit **override** verwendet

werden. Das Überdecken von versiegelten Members mit **new** ist erlaubt Es ist das Pendant zum Java **final**.

---

```
1 public override sealed void MyFunc(); //works
2 public sealed void MyFunc2(); // Compiler-Error
```

---

## 9 Klassen, Structs

### 9.1 Klassen

- Klassen sind Referenztypen die auf dem Heap abgelegt werden
- Klassen können ineinander verschachtelt sein. Die Inner Class hat dabei Zugriff auf alle Member der Outer Class. Die Inner Class wird mit `OuterClass.InnerClass inner = new OuterClass.InnerClass();` initialisiert.
- Klassen können statisch sein. Statische Klassen können nicht abgeleitet werden. Es gibt auch statische Imports `using static System.Math`
- Hat immer einen Default Konstruktor, sofern nicht ein anderer definiert wurde.
- Links steht immer der statische Typ und rechts der dynamische

#### 9.1.1 Type Casts

Type Casts können mit den runden Klammern oder mit dem Keyword `as` gemacht werden. `as` liefert `null` zurück, wenn nicht gecasted werden kann (anstatt eine Exception zu werfen).

---

```

1 // type cast
2 Base b = new Sub();
3 Sub s = (Sub) b; // could throw InvalidCastException
4
5 // cast with 'as'
6 Sub s = b as Sub; // returns null if cast not possible

```

---

#### 9.1.2 Typprüfung

Die Typprüfung gibt `true` wenn:

- Typ von obj identisch wie "Tist (exakt gleicher Typ)
- Typ von obj eine Sub-Klasse von "Tist

---

```

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 }

```

---

#### 9.1.3 Typprüfungen mit implizitem Type Cast

Ebenfalls ist es möglich ein dynamischen Typ zu prüfen und direkt zu casten.

---

```

1 public class Animal { public string Sound { get; } = "Sound like an Animal"; }
2 public class Dog : Animal { public new string Sound { get; } = "Barking like a Dog"; }
3
4 Animal animalJoe = new Dog();
5     Console.WriteLine(animalJoe.Sound); // "Sounds like an Animal"
6     if (animalJoe is Dog dogJoe) {
7         Console.WriteLine(dogJoe.Sound); // "Barking like a Dog"
8     }

```

---

### 9.1.4 Operatoren Überladen

---

```

1 public class Vector{
2     private int x, y;
3
4     public Vector(int x, int y) {};
5
6     // must be static
7     public static Vector operator + (Vector a, Vector b) {
8         return new Vector(a.x + b.x, a.y + b.y);
9     }
10 }

```

---

### 9.1.5 Partial Class

Das **partial** Keyword erlaubt die Definition in mehreren Files. Es sind auch partielle Methoden möglich

---

```

1 partial class MyClass { public void Test1() { .. } }
2 partial class MyClass { public void Test2() { .. } }
3
4 MyClass mc = new MyClass();

```

---

## 9.2 Abstrakte Klassen

- Abstrakte Klassen können nicht direkt instanziiert werden.
- Alle abstrakte Member müssen implementiert sein (**override**)

---

```

1 abstract class Sequence {
2     public abstract void Add(object x); // implicit virtual, no implementation
3     public abstract string Name { get; }; // property
4     public abstract object this[int i]{ get; set; }; // indexer
5     public abstract event EventHandler OnAdd; // Event;
6
7     public override string ToString() {return Name; };
8 }
9
10 class List : Sequence {
11     public override void Add(object x)
12 }

```

---

### 9.3 Sealed Klassen

Von versiegelten "sealed" Klassen kann nicht abgeleitet werden. Es verhält sich also wie das `final` bei Java.

---

```
1 sealed class Sequence {  
2     // members can also be sealed:  
3     public sealed void X();  
4 }  
5  
6 class List : Sequence {} // Compiler error  
7  
8 class Sequence {  
9     // cannot be overwritten, but 'new' is possible  
10    public sealed void X();  
11 }
```

---

### 9.4 Statische Klassen

Statische Klassen sind implizit `sealed`. Sie dürfen nur statische Member enthalten und können nicht instanziiert werden.

---

```
1 static class MyMath {  
2     public const double Pi = 3.14159;  
3     public static double Sin(Double x) { .. }  
4 }
```

---

## 9.5 Structs

- Structs sind Valuetypen die auf Stack liegen
- Structs sind Valuetype und können deshalb nie **null** sein.
- Structs können weder vererben noch erben. (Interfaces sind aber möglich)
- Structs benötigen weniger Speicherplatz wie Klassen
- Es gibt keinen parameterlosen Konstruktor!
- Struct Felder dürfen nicht initialisiert werden
- Structs sollten in folgenden Fällen verwendet werden
  - Repräsentiert einen einzelnen Wert
  - Instanzgrösse ist kleiner als 16 Byte (128 Bit)
  - Ist immutable (nicht der default)
  - Wird nicht häufig geboxt
  - Ist entweder kurzlebig oder wird meist in andere Objekte eingebettet

---

```

1  public struct FinalPoint
2  {
3      public readonly int x;
4      public readonly int y;
5      public FinalPoint(int x, int y) //Konstruktor nur mit Parameter
6      {
7          this.x = x;
8          this.y = y;
9      }
10 }
11 //Verwendung
12 FinalPoint FinalPoint1 = new FinalPoint(1,2);

```

---

## 9.6 Konstruktoren

Es gibt verschiedene Konstruktoren:

**statische Konstruktoren** Ist nicht von aussen verfügbar und wird für Initialisierungsarbeiten verwendet werden. Er wird nur für die erste Instanz aufgerufen. Dieser Konstruktor ist zwingend parameterlos und die Sichtbarkeit darf nicht angegeben werden. (keyword: *static*)

**nicht statische Konstruktoren** Der normale Konstruktor

**private Konstruktoren** Können nur intern verwendet werden. Gibt es einen privaten Konstruktor, so wird **kein** Default Konstruktor erzeugt.

---

```

1  public class MyClass {
2      // call super constructor
3      public MyClass(int a) : base(a) {}
4
5      // call constructor in same class
6      public MyClass(int a) : this(a, false) {}
7

```

---

```

8   public MyClass(int a, boolean b) {}
9
10  // static constructor
11  static MyClass() {}
12  }

```

---

## 9.7 Initialisierungsreihenfolge

1. Statische Felder (Unterklasse zuerst) (nur 1x pro Klasse, falls mehrere Instanzen erzeugt werden!)
2. Statische Konstruktoren (Unterklasse zuerst) (nur 1x pro Klasse, falls mehrere Instanzen erzeugt werden!)
3. Felder (Unterklasse zuerst, in Deklarationsreihenfolge)
4. Konstruktoren (Oberklasse zuerst)

<pre> Sub s1 = new Sub(); // Sub &gt; subStaticValue // Sub &gt; Statischer Konstruktor // Sub &gt; subValue // Base &gt; baseStaticValue // Base &gt; Statischer Konstruktor // Base &gt; baseValue // Base &gt; Konstruktor // Sub &gt; Konstruktor Sub s2 = new Sub(); // Sub &gt; subValue // Base &gt; baseValue // Base &gt; Konstruktor // Sub &gt; Konstruktor </pre>	<pre> class Base {     private static int baseStaticValue = 0;     private int baseValue = 0;     static Base() { }     public Base() { } }  class Sub : Base {     private static int subStaticValue = 0;     private int subValue = 0;     static Sub() { }     public Sub() { } } </pre>
---	---

Abbildung 15: (Initialisierungs-Reihenfolge (mit Vererbung))



Abbildung 16: (Initialisierungsreihenfolge)

Impliziter Aufruf des Basisklassenkonstruktors			Expliziter Aufruf
<pre>class Base { } class Sub : Base {     public Sub(int x) {} } Sub s = new Sub(1);</pre>	<pre>class Base {     public Base() {} } class Sub : Base {     public Sub(int x) {} } Sub s = new Sub(1);</pre>	<pre>class Base {     public Base(int x) {} } class Sub : Base {     public Sub(int x) {} } Sub s = new Sub(1);</pre>	<pre>class Base {     public Base(int x) {} } class Sub : Base {     public Sub(int x)         : base(x) {} } Sub s = new Sub(1);</pre>
Konstruktoraufrufe Okay <ul style="list-style-type: none"> <li>Base()</li> <li>Sub(int x)</li> </ul>	Konstruktoraufrufe Okay <ul style="list-style-type: none"> <li>Base()</li> <li>Sub(int x)</li> </ul>	<b>Compilerfehler</b> Default-Konstruktor für Klasse «Base» wird nicht mehr automatisch erzeugt	Konstruktoraufrufe Okay <ul style="list-style-type: none"> <li>Base(int x)</li> <li>Sub(int x)</li> </ul>

Abbildung 17: (Konstruktoraufrufe)

## 9.8 Destruktoren

Der Destruktor ruft im Hintergrund die Methode Finalize auf.

```
1 class MyClass {
2     ~MyClass() {}
3 }
```



## 9.9 Operator Overloading

Die Methode muss **static** sein und das Keyword **operator** verwenden. **Mindestens 1 Parameter** muss vom Typ der enthaltenen Klasse sein!

---

```

1  class MyClass {
2      public static MyClass operator + (MyClass a, MyClass b) {
3          return new MyClass(a.x + b.x, a.y + b.y);
4      }
5      public static MyClass operator ~(MyClass a) {
6          return new MyClass();
7      }
8
9      public static MyClass operator + (int a, int b) {...} // does not work!
10 }
11
12 // usage
13 MyClass mc3 = mc1 + mc2;
```

---

Folgende Operatoren können überladen werden

Operator	OL	
+ - ! ~ && -- true false	✓	Unäre Operatoren
+ - * / % &   ^ << >>	✓	Binäre Operatoren
==, !=, <, >, <=, >=	✓	Vergleichsoperatoren

Abbildung 18: Operatoren Überladen

## 10 Interfaces

- Kann **nicht** direkt instanziiert werden
- Bei Mitgliedern von einem Interface wird keine Sichtbarkeit angegeben
- Name beginnt mit grossem I
- Mitglieder sind implizit **abstract virtual**
- Mitglieder dürfen nicht **static** oder ausprogrammiert sein
- In der Klasse, welche Interface implementiert, müssen alle Interface-Mitglieder vorhanden sein.
- **override** ist nicht nötig
- Seit C# 8.0 können Interfaces auch Default-Implementierungen haben und so dennoch programmiert werden.

---

```

1  interface ISequence {
2      void Add(object x);
3      string Name { get; }
4      object this[int i] { get; set; }
5      event EventHandler OnAdd;
6  }
7
8  class List : Base, ISequence, I2 {
9      public void Add(object x) { /* ... */ }
10     public string Name { get { /* ... */ } }
11     public object this[int i] { get { /* ... */ } set { /* ... */ } }
12     public event EventHandler OnAdd;
13 }

```

---

### 10.1 Interface Naming Clashes

Es ist möglich, dass zwei implementierte Interfaces dieselben Methoden haben. Dann ist es möglich, die Member-Methoden und/oder ein Default-Verhalten zu implementieren.

---

```

1  class ShoppingCart : ISequence, IShoppingCart {
2      public void Add(object x) { /* ... */ }
3      void ISequence.Add(object x) { /* ... */ }
4      int IShoppingCart.Add(object x) { /* ... */ } // also possible if same return value
5  }

```

---

## 11 Enum

Eine Enumeration ist eine vordefinierte Liste von Konstanten mit einem optionalen Wert. Enum leitet von Int32 ab. Um Speicherplatz zu sparen, könnte man auch von byte, sbyte, short etc erben(Sollte nicht gemacht werden).

---

```

1 enum Days { Sunday = 42, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
2 Days today = Days.Monday;
3 if (today == Days.Monday) { /* ... */ }
4
5 // Two ways to parse Enum out of a String
6 Enum.Parse(typeof(Days), "Monday");
7 MyEnum myEnum;
8 Enum.TryParse("Monday", out myMonday); // returns true if successful
9
10 // Print all Enum Types
11 foreach(string name in Enum.GetNames(typeof(Days))) {
12     Console.WriteLine(name);
13 }

```

---

Enums sind mit Werten definiert. Standardmässig beginnt es mit 0. Jedoch kann das auch angepasst werden.

---

```

1 enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
2 int sundayValue = (int)Days.Sunday; Console.WriteLine("{0} / #{1}.", Days.Sunday,
    sundayValue); // Ausgabe: Sunday / #0
3 int fridayValue = (int)Days.Friday; Console.WriteLine("{0} / #{1}.", Days.Friday,
    fridayValue); // Ausgabe: Friday / #5
4
5 //oder mit eigener Wertedefinition:
6 enum DaysWithValues { Sunday = 10, Monday, Tuesday, Wednesday, Thursday, Friday = 9,
    Saturday };
7
8 int sundayValue = (int)DaysWithValues.Sunday; Console.WriteLine("{0} / #{1}.",
    DaysWithValues.Sunday, sundayValue); // Ausgabe: Sunday / #10
9 int fridayValue = (int)DaysWithValues.Friday; Console.WriteLine("{0} / #{1}.",
    DaysWithValues.Friday, fridayValue); // Ausgabe: Friday / #9

```

---

## 12 Generics

Generics erlauben die Implementationen von generischen Strukturen ohne die Verwendung von **object**. Anstelle von **object** wird **T** verwendet. Generics können in Klassen/Structs, Interfaces, Delegates/Events und Methoden verwendet werden. Während der Verwendung einer der möglichen Strukturen wird **T** beispielsweise als **string** deklariert. Generics sind für Value Types schneller (kein Boxing nötig), bei Reference Type jedoch nicht. (Verglichen mit **object**)

---

```

1 public class Buffer<T> {
2     T[] items;
3     public void Put(T item) { /* ... */ }
4     public T Get() { /* ... */ }
5 }
6
7 //mehrere Typparameter
8
9 public class Buffer<TElement, TPriority> {
10     TElement[] items;
11     TPriority[] priorities;
12     public void Put( TElement item, TPriority prio) { /* ... */ }
13 }

```

---

### 12.1 Type Constraints

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

**Kovarianz** Erlaubt die Zuweisung von stärker abgeleiteten Typen als ursprünglich angegeben

---

```

1 public interface IBuffer<in T>

```

---

**Kontravarianz** Erlaubt die Zuweisung von weniger stark abgeleiteten Typen als ursprünglich angegeben.

---

```

1 public interface IBuffer<out T>

```

---

#### 12.1.1 Beispiele

---

```

1 class MyClass<T, P> where T : IComparable { .. }
2 public T GetInstance<T>() where T : new() {
3     return new T(); // must have default constructor
4 }

```

---

```

1 public void FillList<T>(T source) where T : List<int>, IEnumerable<int> {
2     source.Add(1); source.Add(2); source.Add(3); }

```

---

```

1 class ExamplesCombiningConstraints<T1, T2>
2 where T1 : struct
3 where T2 : Buffer, IEnumerable<T1>, new()
4 { /* ... */ }

```

---

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 <b>parameterlosen</b> "public"Konstruktor haben. Wird benötigt um new T() zu erstellen Dieser Constraint muss wenn mit anderen kombiniert <b>immer zuletzt aufgeführt</b> werden
where T : "ClassName"	T muss von Klasse "ClassName"ableiten.
where T : InterfaceName	T muss Interface InterfaceNameimplementieren.
where T : TOther	T muss identisch sein mit TOther. oder T muss von TOther ableiten.
where T : class?	T muss eine Nullable Reference Type sein. Darunter fallen auch Klassen, Interfaces, Delegates. (Mehr im nächsten Kapitel)
where T : not null	T muss in Non-Nullable Value Type oder Reference Type sein.

Tabelle 4: Type Constraints

## 12.2 Typprüfungen

```
1 Type t = typeof(Buffer<int>); // t.Name = Buffer[System.Int32]
```

## 12.3 Generische Vererbung

Generische Klassen können von anderen generischen Klassen erben:

**Normale Klassen** `class` `MyList<T> : List { }`

**Weitergabe des Typparameters an Basisklasse** `class` `MyList<T> : List<T> { }`

**Konkretisierte generische Basisklasse** `class` `MyIntList : List<int> { }`

**Mischform** `class` `MyIntKeyDict<T> : Dictionary<int, T> { }`

```
1 // Zuweisung mit "normaler" Basisklasse
2 class MyList<T> : MyList { /* ... */ }
3 class MyDict<TKey, TValue> : MyList { /* ... */ }
4 public void Test() {
5     MyList l1 = new MyList<int>();
6     MyList l2 = new MyDict<int, float>();
7     object o1 = new MyList<int>();
8     object o2 = new MyDict<int, float>();
9 }
10
11 // Zuweisung mit generischer Basisklasse
12 class MyList2<T> : MyList<T> { }
13 class MyDict<TKey, TValue> : MyList<TKey> { }
14 public void Test() {
15     MyList<int> l1 = new MyList2<int>();
16     MyList<int> l2 = new MyDict<int, float>();
17     MyList<int> l3 = new MyList<float>(); //Compilerfehler: Typparameter inkompatibel
18     MyList<object> l4 = new MyList<float>(); //Compilerfehler: Typparameter
19                                     inkompatibel
20 }
```

## 12.4 Generische Delegates

Es ist auch möglich Delegates generisch zu machen. Hier einige Beispiele:

---

```
1 public delegate void Action<T>(T i);  
2 public delegate void Action<T1,T2>(T1 obj1, T2 obj2);  
3 public delegate TResult Func<T>(T arg);  
4 public delegate bool Predicate<T>(T obj);
```

---

## 12.5 Generische Collections

Es gibt eine Liste von generischen Collections, hier einige davon:

- `List<T>`
- `SortedList<TKey, TValue>`
- `Dictionary<TKey, TValue>`
- `SortedDictionary<TKey, TValue>`
- `LinkedList<T>`
- `Stack<T>`
- `Queue<T>`

## 13 Nullability

Tony Hoare nennt das Einführen der Null Reference als "Bilion-Dollar-Mistake".

### 13.1 Default operator literal

`default(T)`/ `default` liefert den Default-Wert für angegebene Typen. Bei Reference Types ist die `null`. Bei Value Types ist dies entweder `0`, `false` oder `\0`. Bei Vergleiche (`x is null`) gibt es folgendes Verhalten:

**Reference Types** `true` wenn `x null` ist, ansonsten `false`

**Value Types** `false` in jedem Fall. Compilerfehler wenn `T : struct`

---

```

1 public void NullExamples<T>()
2 {
3     // Nullwerte zuweisen
4     T x1 = null; // Compilerfehler
5     T x2 = 0; // Compilerfehler
6     T x3 = default(T); // OK
7     T x4 = default; // OK (C# 7.1)
8
9     // Nullwerte prüfen
10    if (x1 is null)
11    {
12        /* ... */
13    }
14 }
```

---

### 13.2 Structs / Value Types

Null-Zustand existiert nicht und muss quasi erzwungen werden (Seit 2002 /.NET 2.0 C# 2).

#### 13.2.1 Struct <Nullable>

Value Types kann nun dank Generics "nullfügewiesen werden. Dazu dient die Klasse `System.Nullable<T>`.

Status	HasValue	Value
null	false	?
not null	true	123

Tabelle 5: Werte je nach Status

Wenn auf Value zugegriffen wird und `HasValue false` ist, wird eine `System.InvalidOperationException` geworfen.

---

```

1 public struct Nullable<T>
2 where T : struct
3 {
4     public Nullable(T value);
5
6     public bool HasValue { get; }
7     public T Value { get; }
8 }
```

---

### 13.2.2 T? Syntax

Die T? Syntax dient zur verbesserung der Lesbarkeit. So kann `null` direkt zugewiesen werden.

---

```
1 int? x = 123;
2 int? y = null;
```

---

### 13.2.3 Sicheres Lesen & Type Casts

---

```
1 int? x = null;
2
3 // Klassisch
4 int x1 = x.HasValue
5     ? x.Value
6     : default;
7
8 // Via Methode
9 int x2 = x.GetValueOrDefault();
10
11 // Via Methode inkl. eigenem Default
12 int x3 = x.GetValueOrDefault(-1);
```

---

## 13.3 Klassen / Reference Types

Null-Zustand ist in Runtime abgebildet. Zugriff auf Null References führt zu `NullReferenceException`. Dies ist nicht mehr eliminierbar. Doch Compiler bietet seit C# 8 Vermeidung von Zugriff auf Null Referenzen an. (Seit 2019 / .NET Core 3.0 / C# 8) Compiler generiert Warnungen basierend auf "static flow analysis".

Hinweis: Implementation unterscheidet sich massiv von "Nullable Value Types"

### 13.3.1 Syntax

Die Compiler Warnungen können im .csproj aktiviert werden innerhalb der `<PropertyGroup>` mittels `<Nullable>enable</Nullable>` und deaktiviert mittels `<Nullable>disable</Nullable>`. Die Warnungen können auch im Code via Präprozessor Direktiven mittels `#nullable enable` aktiviert und mit `#nullable disable` deaktiviert werden.

Eine Variable kann mittels `?` als "nullable" deklariert werden. Weiter gibt es den Null-forgiving Operator `!`, welcher den Compiler verspricht, dass die Variable nicht `null` ist. Dies ist sehr gefährlich und sollte nur mit bedacht eingesetzt werden, da Compile Time  $\neq$  Runtime (inkonsistent).

---

```
1 string? nameNull = null;
2 string name = nameNull; // Warning
3
4 if (nameNull is null) // or "name == null"
5 {
6     name = nameNull; // Warning
7     name = nameNull!; // OK, but wrong
8 }
9 else
10 {
11     name = nameNull; // OK
12 }
```

---



## 13.4 Syntactic sugar

### 13.4.1 is null is not null (Pattern matching)

**Value Types** Verwendet `System.Nullable<T>.HasValue`

**Reference Types** Prüft ob es sich um eine Null Referenz handelt mittels `object.ReferenceEquals()`

Der `==` Operator sollte nicht verwendet werden, da er manuell überschrieben worden sein könnte.

---

```

1 int? x = null;
2 string s = null;
3
4 bool b1a = x is null; // true
5 bool b1b = x is not null; // false
6
7 bool b2a = s is null; // true
8 bool b2b = s is not null; // false

```

---

### 13.4.2 ?? (Null-Coalescing Operator)

Binäre Operator. Gibt den Linken Teil zurück, sofern nicht null. Ansonsten wird der Rechte Teil zurückgegeben.

---

```

1 int? x = null;
2 int i = x ?? -1; \\ -1
3
4 \\ Auch möglich
5 int i = x ?? throw new ArgumentNullException(nameof(x));

```

---

### 13.4.3 ??= (Null-Coalescing Assignment)

Setzt die Variable (linker Teil) auf einen gewünschten wert, sofern diese Variable null ist. Keine Operation wenn die Variable ungleich null ist.

---

```

1 int? x = null;
2 int i ??= -1; \\ x hat nun Wert "-1"
3
4 int? x = 10;
5 int i ??= -1; \\ x hat immernoch Wert "10"

```

---

### 13.4.4 ?. (Null-Conditional Operator)

Führt den rechten Teil aus, sofern der linke Teil nicht null ist. Andernfalls wird null geliefert. (Sofern Methode nicht void liefert). Funktioniert auch mit Delegates über `.Invoke()`.

---

```

1 object o = null;
2 Action a = null;
3
4 string s = o?.ToString(); // null
5 a?.Invoke();

```

---

## 14 Record Types

Record Types sind ein reines Compiler Feature und dienen zum erstellen von Datenrepräsentations-Klassen und Reduktion von Boilerplate Code.

**Vereinfacht** Arbeit mit Nullable Reference Types.

**Grundidee** Record Types sollten immutable sein.

---

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

---

Folgender Code wird generiert:

- Konstruktor
- Properties (Immutable)
- Value equality
- Darstellung (ToStringMethode, etc.)
- Vererbung wird berücksichtigt (z.B. Equality)

### 14.1 Deklarationsmöglichkeiten

Eine manuelle Deklaration wie normale Klassen ist möglich. So werden teilweise To-String und Value Equality generiert. Besser ist: Positional Syntax verwenden (Siehe oben)

---

```
1 public record Person
2 {
3     public Person() : this(0, "") { }
4     public Person(int id, string name)
5     {
6         Id = id;
7         Name = name;
8     }
9
10    public int Id { get; init; }
11    public string Name { get; init; }
12 }
13
14 // Anwendungsbeispiel
15 Person p1 = new();
16 Person p2 = new(1, "Mary");
```

---

Man kann auch eine gemischte Variante nutzen. Um so eigene Methoden zu erweitern. Wichtig: Generierte Properties werden als Init-Only setter generiert. Man kann so eine Warnung herausgeben, wenn Name leer.

---

```
1 public record Person(int Id)
2 {
3     public string Name { get; init; }
4     public void DoSomething() { }
5 }
6
7 // Anwendungsbeispiel
8 Person p1 = new(0);
9 p1.Id = 0; // Compilerfehler
10 p1.Name = ""; // Compilerfehler
11 Person p2 = new(0) { Name = "" }; // OK
```

---

## 14.2 Value Equality

Der generierte Code zu Value equality vergleicht alle Properties und macht KEINE Reference-Equality. Involviert auch Properties allfälliger Basisklassen.

---

```
1 public record Person(int Id, string Name);
2
3 // Anwendungsbeispiel
4 Person p1 = new(0, "Mary");
5 Person p2 = new(0, "Mary");
6 bool eq1 = p1 == p2; // true
7 bool eq2 = p1 != p2; // false
8 bool eq3 = p1.Equals(p2); // true
9 bool eq4 = ReferenceEquals(p1, p2); // false
```

---

## 14.3 Nondestructive mutation

Da Immutable, müssen Records kopiert werden, will man was ändern. Dazu gibt es das Schlüsselwort with.

---

```
1 public record Person(int Id, string Name);
2
3 // Anwendungsbeispiel
4 Person p1 = new(0, "Mary");
5
6 Person p2 = p1 with { Id = 1 };
7 bool eq2 = p1 == p2; // false
8
9 Person p3 = p1 with { };
10 bool eq3 = p1 == p3; // true
```

---

## 15 Delegates

Ein Delegate ist ein eigener **Referenztyp** und wird daher grundsätzlich ausserhalb von Klassen definiert. Jedes Delegate erbt von der Klasse `MulticastDelegate`. Er bietet eine Vereinfachung von Interfaces. Delegates können als Ersatz für das Factory und Template Method Pattern verwendet werden. Genutzt werden Delegates vor allem für zwei Dinge:

1. Methoden als Parameter übergeben
2. Definition von Callback-Methoden

Delegates werden so verwendet:

- Deklaration Delegate Typ: `public delegate void Notifier(string sender);`
  - Schlüsselwort `delegate`
  - Definition von einem Rückgabewert (hier `void`)
  - Definition eines Names (hier `Notifier`)
  - Definition von Parametern (hier `string sender`)
- Deklaration der Delegate-Variable: `Notifier greetings;`
- Zuweisung einer Methode:
  - Standard: `greetings = new Notifier(SayHi);`
  - Kurzform: `greetings = SayHi;`
  - Anonym: `greetings = delegate(string sender) { /*... */ }`
- Aufruf: `greetings("John");`

---

```

1 // Deklaration eines Delegate-Typs
2 public delegate void Notifier(string sender);
3
4 class Examples {
5     public static void Test()
6     {
7         // Deklaration Delegate-Variable
8         Notifier greetings;
9         // Zuweisung einer Methode
10        greetings = new Notifier(SayHi);
11        // Kurzform
12        greetings = SayHi;
13        // Aufruf einer Delegate-Variable
14        greetings("John"); // "Hello John"
15
16        Notifier greetings2;
17        // anonyme Methode zugewiesen
18        greetings2 = delegate(string sender) { Console.WriteLine("Ciao {0}", sender); };
19        greetings2("Seppo"); // "Ciao Seppo"
20    }
21
22    private static void SayHi(string sender)
23    {
24        Console.WriteLine("Hello {0}", sender)
25    }
26 }

```

---

```

public delegate void MyDel(string sender);

public class Examples
{
    public void Print(string sender)
    {
        Console.WriteLine(sender);
    }
    public static void PrintStatic(string sender)
    {
        Console.WriteLine(sender);
    }
    public void Test()
    {
        MyDel x1;

        /* ... */
    }
}

```

```

// Standard (Instanz-Methode) C# 1.0 / 2.0
x1 = new MyDel(this.Print);
x1 = this.Print;

// Standard (Statischer Meth) C# 1.0 / 2.0
x1 = new MyDel(Examples.PrintStatic);
x1 = Examples.PrintStatic;

// Anonymous Delegate
x1 = delegate(string sender)
{ Console.WriteLine(sender); };

// Anonymous Delegate (Kurzform)
x1 = delegate { Console.WriteLine("Hello"); };

// Lambda Expression (LINQ / später)
x1 = sender => Console.WriteLine(sender);

// Statement Lambda Expr. (LINQ / später)
x1 = sender => { Console.WriteLine(sender); };

```

Abbildung 19: Delegate Lambda Overview

## 15.1 Multicast Delegates

Jedes Delegate ist auch ein Multicast Delegate. Im Unterschied zum normalen Delegate beinhaltet das Multicast Delegate mehrere Methoden. Weitere Methoden können mit += hinzugefügt und mit -= wieder entfernt werden. Die Methoden werden dann nacheinander aufgerufen. (Intern Linked List). Lambdas werden vom Compiler als Delegate abgebildet. Speichert man den Rückgabewert bei der Ausführung des Delegates ab, entspricht dieser dem **Rückgabewert des letzten Funktionsaufruf**, sofern dieses ein Return-Value hatte.

```

1 // keyword delegate
2 public delegate void Notifier(string sender);
3
4 class Examples {
5     public void Test() {
6         // Deklaration Delegate-Variable
7         Notifier greetings;
8         // Zuweisung einer Methode mit passender Signatur
9         greetings = new Notifier(SayHello);
10        // Kurzform
11        greetings = SayHello;
12        greetings += SayGoodBye;
13        // Aufruf einer Delegate-Variable
14        greetings("John");
15    }
16
17    private void SayHello(string sender) {
18        Console.WriteLine("Hello {0}", sender);
19    }
20
21    private void SayGoodBye(string sender) {
22        Console.WriteLine("Good bye {0}", sender);
23    }
24 }
25
26 // anonymous delegate
27 // inline multicast delegate
28 Calculator calc =
29     delegate (int a, int b) { return a+b;

```

```
30     + delegate (int a, int b) { return a - b};  
31 int res = calc(3,2) // 1 (last call)
```

---

## 15.2 Funktionsparameter

---

```
1  public delegate void Action(int i);  
2  public class MyClass  
3  {  
4      public static void PrintValues(int i)  
5      {  
6          Console.WriteLine("Value {0}", i);  
7      }  
8      public void SumValues(int i) { Sum += i; }  
9      public int Sum { get; private set; }  
10 }  
11 public class FunctionParameterTest  
12 {  
13     static void ForAll(int[] array, Action action)  
14     {  
15         Console.WriteLine("ForAll called...");  
16         if (action == null) { return; }  
17         foreach (int t in array)  
18         {  
19             action(t);  
20         }  
21     }  
22 }  
23  
24 public static void TestSum()  
25 {  
26     MyClass c = new MyClass();  
27     int[] array = { 1, 5, 8, 14, 22 };  
28  
29     // Delegate Variables  
30     Action v1 = c.PrintValues; // Static  
31     Action v2 = c.SumValues; // Instance Method  
32  
33     // Execution  
34     ForAll(array, v1);  
35     ForAll(array, v1);  
36  
37     ForAll(array, v2);  
38     Console.WriteLine("--- Sum {0}", c.Sum);  
39     ForAll(array, v2);  
40     Console.WriteLine("--- Sum {0}", c.Sum);  
41 }  
42 // Konsolen Ausgabe v1  
43 ForAll called...  
44 Value 1  
45 Value 5  
46 Value 8  
47 Value 14  
48 Value 22  
49 ForAll called...  
50 // Konsolen Ausgabe v2  
51 ForAll called...  
52 --- Sum 50  
53 ForAll called...  
54 --- Sum 100
```

---

---

```

1 public static void Test() {
2     Action ToExecute;
3     ToExecute = delegate(string str) { Console.WriteLine(str.ToLower());}; // Add
4         toLower method
5     ToExecute += delegate(string str) { Console.WriteLine(str.ToUpper());}; // add
6         toUpper method
7     ToExecute("Tubel"); //TUBEL tubel
8 }

```

---

### 15.3 Anonyme Methoden

Mit anonymen Methoden ist keine Deklaration einer Methode nötig. Methodencode wird in-place angegeben. Ebenfalls hat eine Methode Zugriff auf lokale Variablen (im unteren Beispiel `sum`).  
**Besser mit Lambdas lösen!**

---

```

1 class AnonymousMethods {
2     void Foo() {
3         list.ForEach(delegate(int i) { Console.WriteLine(i); } );
4         int sum = 0;
5         list.ForEach(delegate(int i) { sum += i; } );
6     }
7 }
8 }

```

---

### 15.4 Callbacks

**Besser mit Events lösen!**

---

```

1 public delegate void TickEventHandler (int ticks, int interval);
2
3 public class Clock
4 {
5     private TickEventHandler OnTickEvent;
6
7     public void add_OnTickEvent(TickEventHandler h)
8     { OnTickEvent += h; }
9     public void remove_OnTickEvent(TickEventHandler h)
10    { OnTickEvent -= h; }
11    private void Tick(object sender, EventArgs e)
12    {
13        ticks++;
14        OnTickEvent?.Invoke(ticks, interval);
15    }
16 }
17
18 public static void Test()
19 {
20     Clock c1 = new Clock(1000);
21     Clock c2 = new Clock(2000);
22
23     ClockObserver t1 = new ClockObserver("01");
24     ClockObserver t2 = new ClockObserver("02");
25
26     //Observers anmelden
27     c1.add_OnTickEvent(t1.OnTickEvent);
28     c2.add_OnTickEvent(t2.OnTickEvent);
29
30     // Warteschlaufe o.Ä

```

```
31
32 //Observers abmelden
33 c1.remove_OnTickEvent(t1.OnTickEvent);
34 c2.remove_OnTickEvent(t2.OnTickEvent);
35 }
```

---



## 15.5 Events

Events sind Instanzen von Delegates, wobei das Delegate implizit **private** ist, damit es das Event nur von intern getriggert werden kann. (Kompiler 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 // 1. define delegate
2 public delegate void TimeEventHandler (object source, CustomEventArgs args);
3
4 // 2. define publisher
5 public class Clock {
6     // 3. define an event based on delegate
7     // compiles to private field with subscribe, unsubscribe methods
8     public event TimeEventHandler OnTimeChangedEvent;
9
10    public void MyAction() {
11        // convetional name
12        OnTimeChanged();
13    }
14
15    // 4. raise event
16    protected virtual void OnTimeChanged() {
17        CustomEventArgs args = new CustomEventArgs() {
18            Custom = new custom(); // ref model
19        }
20        OnTimeChangedEvent?.Invoke(this, args)
21    }
22 }
23
24 // 5. write subscribers
25 public class Subscriber {
26
27     // match with delegate
28     public void OnTimeChanged(object source, CustomEventArgs args) {
29         /*... */
30     }
31 }
32
33 //6. Event Args: Create
34 public class CustomEventArgs : EventArgs {
35     public Custom CustomProp { get; set; }
36 }
37
38 // Model
39 public class Custom {
40     /*... */
41 }
42
43 // 7. use it
44 static void Main(string[] args) {
45     var clock = new Clock(); // publisher
46     var subscriber = new Subscriber(); // subscriber
47
48     // add as many subscriber as needed
49     clock.OnTimeChangedEvent += subscriber.OnTimeChanged;
50
51     // 8. exec
52     clock.MyAction();
53 }

```

---

## 15.6 EventHandler

Ein EventHandler hat folgende Eigenschaften:

**Standard Syntax** `public delegate void AnyHandler(object sender, EventArgs e);`

**1. Parameter `object` sender:**

- Sender des Events
- Absender übergibt bei Aufruf des Delegates / Events `this` mit

**2. Parameter `EventArgs` e:**

- Beliebige Sub-Klasse von `EventArgs`
- Enthält Informationen zum Event (z.B. Linke oder Rechte Maustaste beim Klick, etc.)
- Begründung: Argumente können jeder ergänzt werden ohne Anpassung der Signatur

Anstelle eines eigenen EventHandler kann man auch den bestehenden `EventHandler<EventArgs>` nutzen.

Listing 6: C# Event Handler

```
1 public delegate void ClickEventHandler(obj sender, EventArgs e);
2
3 public class ClickEventArgs : EventArgs {
4     public string MouseButton{get; set;}
5 }
6
7 public class Button {
8     public event ClickEventHandler OnClick;
9 }
10
11 public class Usage {
12     public void Test() {
13         Button b = new Button();
14         // add custom click handler, must match delegate signature
15         b.OnClick += OnClick;
16     }
17
18     // click handler
19     private void OnClick(sender, ClickEventArgs eventargs) {
20         /*...*/
21     }
22 }
```

## 16 Lambdas

- Lambdas können 0 oder mehrere Parameter haben
- Der Typ der Parameter darf weggelassen werden
- Man unterscheidet zwischen **Statement Lambdas** (Mit geschweiften Klammern) und **Expression Lambdas** (einzelner Ausdruck, kein return nötig). Ein Lambda kann als Func<> Typ gespeichert werden.

---

```

1 // Prototype
2 Func<[param_type], [return_type]> myLambda;
3
4 // Expression Lambda
5 Func<int, bool> fe = i => i % 2 == 0;
6
7 // Statement Lambda
8 Func<int, bool> fs = i => {
9     int rest = i%2;
10    bool isRestZero = rest == 0;
11    return isRestZero;
12 };
13
14 // Can be nested
15 Func<string, Func<string, int>> l = (string s) => ((string s2) => s2.Length);
16 var call = l("a")("b");

```

---

### 16.1 Closure

Der Zugriff auf lokale Variablen aus dem Lambda ist erlaubt.

---

```

1 int x = 0;
2 Action a = () => x = 1;
3 Console.WriteLine(x); // Output: 0
4 a();
5 Console.WriteLine(x); // Output: 1

```

---

```

1 // each lambda has its own instance of multiplier
2 public static Func<int, int> GetOp() {
3     int multiplier = 2;
4     Func<int, int> operator = x => x * multiplier++;
5     return operator;
6 }
7
8 var operator = GetOp();
9 oper(2); // 4
10 GetOp()(2); // 4
11 oper(2) // 6

```

---

## 17 Iteratoren

Es sind mehrere Iteratoren zur gleichen Zeit auf eine Liste erlaubt. Die Collection darf während der Iteration nicht verändert werden.

### 17.1 Foreach Loop

Der Foreach Loop ist in C# gleich wie in Java mit dem Unterschied, dass anstatt einem Doppelpunkt das Keyword **in** verwendet wird. In C# hat ein **foreach**-Loop folgende Besonderheiten:

- **continue** Unterbricht die aktuelle Iteration
- **break** Unterbricht gesamten Loop

Die Collection, über welche geloopt wird, muss **IEnumerable** resp. **IEnumerable<T>** implementieren. Eine andere Variante ist, dass die Collection einer Implementation von **IEnumerable** resp. **IEnumerable<T>** ähneln muss. Das bedeutet konkret:

- Collection hat Methode **GetEnumerator()** mit Rückgabewert **e**
- **e** hat eine Methode **MoveNext()** mit Rückgabewert **bool**
- **e** hat ein Property **Current**

---

```

1 int[] list = new int[] { 1, 2, 3, 4, 5, 6 };
2 foreach (int i in list) {
3     if (i == 3) continue;
4     if (i == 5) break;
5     Console.WriteLine(i);
6 } // Ausgabe 1 2 4

```

---

### 17.2 Iterator Interfaces

Beim Thema Iteratoren haben wir zwei Interfaces, welche beteiligt sind:

1. **public interface IEnumerable** mit den Member:
  - **IEnumerator GetEnumerator()** gibt ein **IEnumerator** Objekt **e** zurück.
2. **public interface IEnumerator** mit den Member:
  - **object Current { get; }** gibt aktuelles Element zurück
  - **bool MoveNext();** springt zum nächsten Item
  - **void Reset();** Zurücksetzen des Iterators

Diese Interfaces gibt es auch in generischer Form: **Interface IEnumerable<T> : IEnumerable** und **public interface IEnumerator<out T> : IDisposable, IEnumerator**

---

```

1 // each collection, which implements IEnumerable, supports foreach
2 public interface IEnumerable<out T> : IEnumerable {
3     IEnumerator<T> GetEnumerator();
4 }
5
6 // IEnumerator
7 public interface IEnumerator<T> {
8     T Current { get; }

```

---

```

9     bool MoveNext(); // calls yield return
10    void Reset();
11 }
12
13 public interface IEnumerator<out T> : IDisposable, IEnumerator {
14     T Current { get; }
15 }

```

---

### 17.3 Iterator Methoden und Yield Return

- Eine Iterator Methode muss die Signatur `public IEnumerator<int> GetEnumerator()` haben
- `yield return` Statement gibt den nächsten Wert für die nächste Iteration eines `foreach` Loops zurück
- muss **mindestens ein** `yield return` Statement enthalten
- `IEnumerator.MoveNext()` ruft den nächsten `yield return` in `GetEnumerator()` auf.
- `yield break` terminiert die aktuelle Iteration

---

```

1 class MyIntList {
2     private int[] data = new int[10];
3
4     // Standard Iterator
5     public IEnumerator<int> GetEnumerator() {
6         for (int i = 0; i < data.Length; i++) {
7             yield return data[i];
8         }
9     }
10
11    // Spezifische Iterator-Methode (Rueckgabewert = IEnumerable)
12    public IEnumerable<int> Range(int from, int to) {
13        for (int i = from; i < to; i++) {
14            yield return data[i];
15        }
16    }
17
18    // Spezifisches Iterator-Property (Rueckgabewert = IEnumerable)
19    public IEnumerable<int> Reverse {
20        get {
21            for (int i = data.Length - 1; i >= 0; i--) {
22                yield return data[i];
23            }
24        }
25    }
26
27    // Fibonacci
28    public static IEnumerable<int> Fibonacci(int number) {
29        int a = 0, b = 1;
30
31        yield return a;
32        yield return b;
33
34        for (int i = 0; i <= number; i++)
35        {
36            int temp = a;
37            a = b;
38            b = temp + b;

```

```
39     yield return b;  
40 }  
41 }  
42 }
```

---

```
1 MyIntList list = new MyIntList();  
2  
3 // Aufruf Standard Iterator  
4 foreach (int elem in list) { .. }  
5  
6 // Aufruf spezifische Iterator Methode  
7 foreach (int elem in list.Range(2, 7)) { .. }  
8  
9 // Aufruf Iterator Property  
10 foreach (int elem in list.Reverse) { .. }
```

---

## 18 Extension Methods

- Extension Methods erlauben das Erweitern (aus Anwendersicht) bestehender Klassen
- Extension Methoden können auf Klassen, Structs, Interfaces, Delegates, Enumeratoren und Arrays angewendet werden.
- Extension Methods werden hauptsächlich für das Method Chaining verwendet.
- Eine Extension Method muss **statisch** sein und der erste Parameter der Methode **this** als Prefix haben.
- Der erste Parameter definiert die Klasse, welche erweitert wird
- Kein Zugriff auf interne Members aus Extension Methode

---

```

1 public static class ExentsionMethods
2 {
3     public static string ToStringSafe(this object current)
4     {
5         return current == null ? string.Empty : current.ToString();
6     }
7 }

```

---

### 18.1 Deferred Evaluation

Der Aufruf von einer Methode, welche einen `IEnumerator` zurückgibt, führt diesen **noch nicht** aus. Dies passiert erst implizit im foreach-loop. Mit dem Verwenden von Extension Methods mit Iterationen kann so eine Query-Operation implementiert werden, welche verschachtelt werden kann. Diese haben folgende Eigenschaften:

- Erweitern alle Collections die `IEnumerable` implementieren
- Liefern in der Regel wieder ein `IEnumerable`
- Können mit dem `."`Operator verkettet werden
- Werden "deferred"(aufgeschoben) evaluiert

Mit dem `this`-Kontext, welcher der Extension Methode mitgegeben wird, macht man die Verbindung zu der Collection.

---

```

1 public static class MyExtensions {
2     public static IEnumerable<T> OstWhere<T> (this IEnumerable<T> source, Predicate<T>
3         predicate) {
4         foreach (T item in source) {
5             if (predicate(item)) {
6                 yield return item;
7             }
8         }
9     }
10    public static IEnumerable<T> OstOfType<T> (this IEnumerable source) {
11        foreach (object item in source) {
12            if (item is T) {
13                yield return (T)item;
14            }
15        }
16    }
17 }

```

---

```
15     }
16 }
17 }
18
19 // Anwendung
20
21 object[] list = { 4, 3.5, "abc", 3, 6 };
22 // Extension Method Syntax
23 IEnumerable<int> res = list
24     .HsrOfType<int>()
25     .HsrWhere(delegate (int k) { return k % 2 == 0; });
```

---



## 19 Exceptions

Wie auch in anderen Sprachen behandeln Exceptions unerwartete Programmstati oder Ausnahmeverhalten zur Laufzeit. Sie sind Fehlercodes vorzuziehen.

Es wird pro Exception nur ein catch-Block ausgeführt. Jede Exception muss von System.Exception erben. Es gibt keine throws Anmerkung am Methoden Kopf. In C# sind alle Exception Unchecked Exceptionn (müssen nicht behandelt werden.)

---

```
1 try {  
2     // code to execute  
3 } catch (FileNotFoundException e) {  
4 } catch (IOException) {  
5     // optional var name, if not needed  
6 } catch {  
7     // implizit System.Exception  
8 } finally {  
9     // always executed  
10 }  
11  
12 // throw exception  
13 throw new Exception("An error occurred");  
14 throw new ArgumentNullException(nameof(s)); // nameof zur Ermittlung des ungültigen  
    Parameter  
15  
16 // exception weiterwerfen ( neuen Stack Trace)  
17 catch (Exception e) { throw e; }  
18  
19 // rethrowing (Stack Trace bleibt erhalten)  
20 catch (Exception e) { throw; }
```

---

### 19.1 Exception Filter

Exception Filters sind Catch-Blöcke, welche nur unter definierten Bedingungen ausgeführt werden. Es sind when Klauseln, welche einen **bool** Expression erwarten.

---

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

---

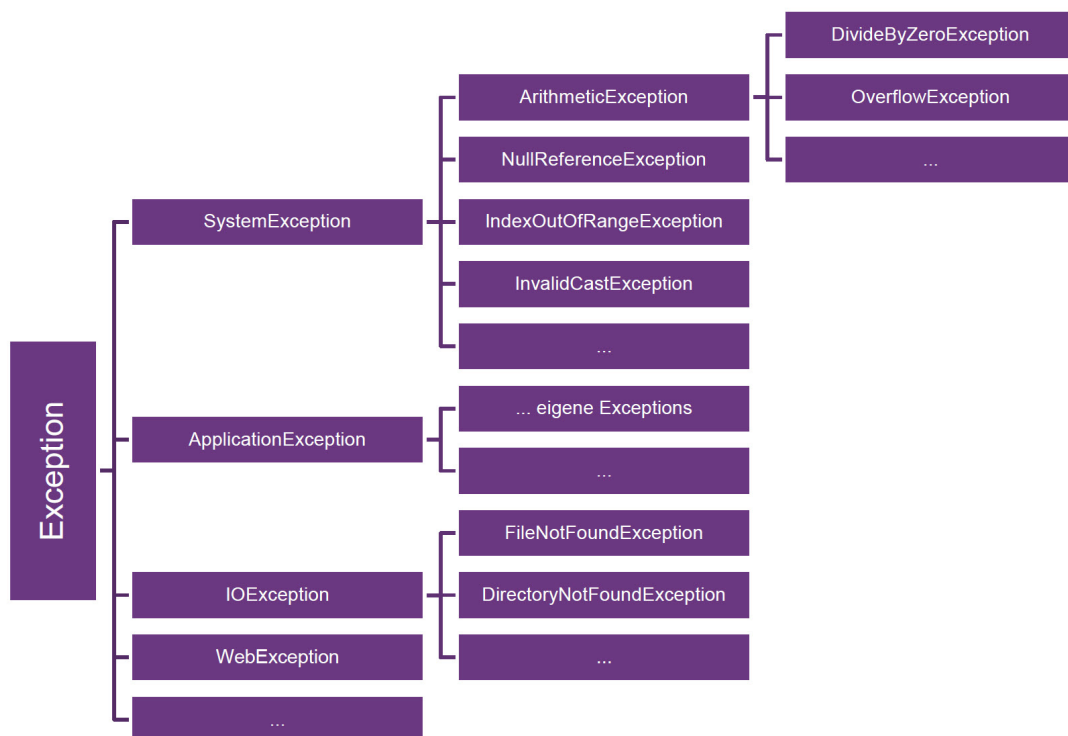


Abbildung 20: Exception Klassen

## 20 Direct Initialization

### 20.1 Object Initializers

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

---

```

1  Student s1 = new Student("John") {
2      Id = 2009001,
3      Subject = "Computing"
4  };
5  Student s2 = new Student {
6      Name = "Ann",
7      Id = 2009002,
8      Subject = "Mathematics"
9  };
10
11 // Object Initializers zusammen mit Lamdas
12 int[] ids = { 2009001, 2009002, 2009003 };
13 IEnumerable<Student> students = ids.Select(n => new Student { Id = n });

```

---

### 20.2 Collection Initializers

Ist das selbe wie Objekte Initializers, jedoch mit Listen.

---

```

1  List<int> l1 = new List<int> { 1, 2, 3, 4 };
2  Dictionary<int, string> d1 = new Dictionary<int, string>
3  {
4      { 1, "a" },
5      { 2, "b" },
6      { 3, "c" }
7  };
8  d1 = new Dictionary<int, string> {
9      [1] = "a",
10     [2] = "b",
11     [3] = "c"
12 };
13
14 object s = new Dictionary<int, Student>
15 {
16     { 2009001, new Student("John") {
17         Id = 2009001,
18         Subject = "Computing" } },
19     { 2009002, new Student {
20         Name = "Ann", Id = 2009002,
21         Subject = "Mathematics" } }
22 };

```

---

### 20.3 Kombination aus Object und Collection Initializers

//Kombination Object und Collection Initializers

---

```

1  object s = new Dictionary < int, Student >
2  {
3      {
4          2009001,
5          new Student("John") {

```

---

```

6      Id = 2009001, Subject = "Computing"
7    },
8  },
9  {
10     2009002,
11     new Student {
12       Name = "Ann", Id = 2009002, Subject = "Mathematics"
13     }
14   }
15 };

```

---

## 20.4 VAR: Anonymous Types

- Mit dem Schlüsselwort **var** wird der Typ vom Compiler herausgefunden
- **var** kann nur für lokale Variablen verwendet werden. Der Einsatz bei Parametern, Klassenvariablen und Properties ist nicht erlaubt.
- Der Typ wird aus der Zuweisung abgeleitet, wobei die Variable zu 100% typensicher bleibt.
- Wird meistens in LINQ Queries verwendet (um Zwischenresultate zu speichern)
- **Wichtig:** Properties von Anonymen Typen sind immer readonly.

---

```

1  class Student {
2    public string Name;
3    public int Id;
4    public string Subject {
5      get;
6      set;
7    }
8    public Student() {}
9    public Student(string name) {
10     Name = name;
11   }
12 }
13 public class Examples {
14   public void Test() {
15     var a = new {
16       Id = 1, Name = "John"
17     };
18     var b = new {
19       a.Id, a.Name
20     };
21     var studentList = new List < Student > ();
22     var q = studentList.GroupBy(s => s.Subject).Select(grp => new {
23       Subject = grp.Key, Count = grp.Count()
24     });
25   }
26 }

```

---

## 21 LINQ: Language Integrated Query

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. (Sie erzeugen den selben *msil* Code) Auch LINQ ist reines Compiler Feature.

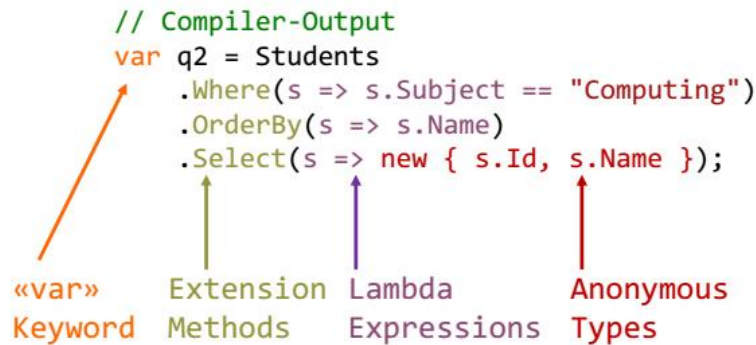


Abbildung 21: LINQ Komponenten

---

```

1 // Query expression syntax
2 var empQuery =
3     from e in employees
4     from d in departments
5     where e.DepId == d.Id
6     orderby d.Name
7     select new { EmployeeName = e.Name, DepartmentName = d.Name };
8
9 // Extension Method / Lamda syntax
10 var empQuery = employees
11     .Join(departments,
12         eKey => eKey.DepId,
13         dKey => dKey.Id, (e, d) => new {
14             EmployeeName = e.Name,
15             DepartmentName = d.Name })
16     .OrderBy(k1 => k1.DepartmentName);
17
18 // Query expression syntax
19 var projList = from p in projects
20 from e in p.Employees
21 orderby p.Name, e.Name
22 select new { Project = p.Name, Employee = e.Name };
23
24 // Extension Method / Lamda syntax
25 var projList = projects
26     .SelectMany(p => p.Employees
27         .Select(e => new { Project = p.Name, Employee = e.Name }))
28     .OrderBy(p => p.Project)
29     .ThenBy(p => p.Employee);

```

---

### 21.1 Lambda Expressions

Eine Lambda Expression ist eine anonyme Methode mit folgende Eigenschaften:

- Keine Implementation einer benannten Methode nötig
- **delegate** Schlüsselwort fällt weg
- Angaben von Parametertypen optional
- Kann auf äussere Variablen zugreifen, aber nicht umgekehrt
- Zwei Ausprägungen
  - Expression Lambdas (nicht im Klammern gefasst, genau ein Ausdruck): `(parameters)=> expression`
  - Statement Lambdas (Beliebig viele Statements in Block gefasst) `(parameters)=> { statements; }`

## 21.2 Extension Methods und Deferred Evaluation

Query Operationen sind ebenfalls mit `yield return` implementiert. Das bedeutet geben ein `IEnumerable<T>` zurück, welches nicht direkt ausgeführt wird. Da es nicht direkt ausgeführt sondern zurückgeschoben wird, spricht man auch von Deffered Evaluation. Daher können die Resultate immer anders sein.

---

```

1 public void TestDeferredEvaluation(){
2     string[] cities = { "Bern", "Basel", "Zürich", "Rapperswil", "Genf" };
3     IEnumerable<string> citiesB = cities.Where(c => c.StartsWith("B"));
4     // Ausführung: 2 Städte (Bern, Basel)
5     foreach (string c in citiesB) { /* ... */ }
6     cities[0] = "Luzern";
7     // Ausführung: 1 Stadt (Basel)
8     foreach (string c in citiesB) { /* ... */ }
9 }

```

---

## 21.3 Extension Methods und Immediate Evaluation

Einige Operatoren führen Queries direkt aus (in der Regel wenn der Rückgabewert nicht `IEnumerable` ist). Resultate werden direkt in Variable gespeichert

- `ToList` / `ToArray`
- `Count` / `First`
- `Sum` / `Average`

---

```

1 public void TestDeferredEvaluation()
2 {
3     string[] cities = { "Bern", "Basel", "Zürich", "Rapperswil", "Genf" };
4     // Ausführung
5     List<string> citiesB = cities.Where(c => c.StartsWith("B")).ToList();
6     int citiesEndl = cities.Where(c => c.EndsWith("l")).Count();
7 }

```

---

## 21.4 Extensions Syntax (Fluent Syntax)

### 21.4.1 LINQ Extension Methods

LINQ bringt in der Klasse `Enumerable` eine Vielzahl an Query Operatoren wie `Where()`, `OrderBy()`, etc. mit sich. Innerhalb der Extension Method kann dann ein z.B ein Lamda übergeben werden. (Predicate)

---

```

1  // needs to be static class
2  public static class Extensions {
3
4      // should be generic
5      public static void HsrForEach<TSource>(this IEnumerable<TSource> source,
6          Action<TSource> action) {
7          foreach (TSource item in source) {
8              action(item);
9          }
10
11     public static IEnumerable<TSource> HsrWhere<TSource>(this IEnumerable<TSource>
12         source, Func<TSource, bool> predicate) {
13         foreach (TSource item in source) {
14             if (predicate(item)) {
15                 // use yield return
16                 yield return item;
17             }
18         }
19
20         // use yield, because we return IEnumerable
21         public static IEnumerable<TResult> HsrOfType<TResult>(this IEnumerable source) {
22         foreach (object item in source) {
23             if (item is TResult) {
24                 yield return (TResult)item;
25             }
26         }
27     }
28
29     public static List<TSource> HsrToList<TSource>(this IEnumerable<TSource> source) {
30         return new List<TSource>(source);
31     }
32
33     public static int HsrSum<TSource>(this IEnumerable<TSource> source, Func<TSource,
34         int> selector) {
35         int sum = 0;
36         foreach (TSource t in source) {
37             sum += selector(t);
38         }
39         return sum;
40     }
41 }

```

---

### 21.4.2 SelectMany

SelectMany erleichtert das zusammenfassen verschachtelter Listen.

---

```

1  var projList = projects
2      .SelectMany(p => p.Employees
3      .Select(e => new { Project = p.Name,
4          Employee = e.Name })))

```

---

```

5      .OrderBy(p => p.Project)
6      .ThenBy(p => p.Employee);

```

## 21.5 Query Expressions Syntax

```

1  // 1. Datenquelle waehlen
2  int[] numbers = { 0, 1, 2, 3, 4, 5, 6 };
3
4  // 2. Query erstellen
5  var numQuery = from num in numbers
6                  where (num % 2) == 0
7                  select num;
8
9  // 3. Query ausfuehren
10 foreach (int num in numQuery) {
11     Console.WriteLine("{0,1} ", num);
12 }

```

- from: Datenquelle
- where: Filter
- orderby: Sortierung
- select: Projektion
- group: Gruppierung in eine Sequenz von Gruppen Elementen
- join: Verknüpfung zweier Datenquellen
- let: Definition von Hilfsvariablen

**Grundregeln:** Query Syntax beginnt immer mit `from` und Query Syntax endet immer mit `select` oder `group`

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 Distinke Elemente zweier Mengen
OrderBy[Descending]	ElementAt Element an numerischer Position	Intersection Überschneidende Elemente zweier Mengen
ThenBy[Descending]	Take / Skip Alle Elemente vor/nach einer numerischen Position	Except Elemente aus Menge A die in Menge B fehlen
GroupBy	TakeWhile / SkipWhile Alle Elemente vor/nach passendem Prädikat	Repeat N-fache Kopie der Liste
Count	Reverse Alle Elemente in umgekehrter Reihenfolge	
Sum / Min / Max / Average		

Abbildung 22: Query Operatoren



### 21.5.1 Range Variablen

Range Variablen entstehen durch `from` oder `join` oder `into` und sind readonly. Im folgenden Beispiel sind die Variablen `s`, `m` und `g` Range Variablen:

---

```
1 from s in Students
2 join m in Markings on s.Id equals m.StudentId
3 group s by s.Subject into g
4 select g;
```

---

### 21.5.2 Gruppierung

Transformation in Key/Value Pairs:

---

```
1 // q: IEnumerable<IGrouping<string, string>>
2 var q = from s in Students
3 group s.Name by s.Subject;
4 foreach (var group in q)
5 {
6     Console.WriteLine(group.Key);
7     foreach (var name in group)
8     {
9         Console.WriteLine(" " + name);
10    }
11 }
12
13 // Gruppierung mit direkter Weiterverarbeitung mittels into
14 var q = from s in Students
15 group s.Name by s.Subject into g
16 select new {
17     Field = g.Key,
18     N = g.Count()
19 };
20
21 foreach (var x in q) {
22     Console.WriteLine(x.Field + ": " + x.N);
23 }
24
25
26 // Anz. Bestellungen pro Datum
27 from best in Bestellungen
28 group best by best.Datum into datumGroup
29 orderby datumGroup.Key
30 select new {
31     Datum = datumGroup.Key,
32     Anzahl = datumGroup.Count()
33 };
```

---

### 21.5.3 Inner Joins

Ein Inner Join nimmt nur jene Ergebnisse, die nicht **null** sind. Es werden zwei Mengen über einen Schlüssel verknüpft. **Wichtig:** `equals` nicht `==` verwenden.

---

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

---

### 21.5.4 Group Joins

Ein Group Join verwendet die `into` Expression. `s` bleibt sichtbar

---

```
1 // Pro "Student" wird eine Liste für alle "Markings" erstellt
2 var q =
3     from s in Students
4     join m in Markings on s.Id equals m.StudentId
5     into list
6     select new
7     {
8         Name = s.Name,
9         Marks = list
10    };
11
12 foreach (var group in q) {
13     Console.WriteLine(group.Name);
14     foreach (var m in group.Marks) {
15         Console.WriteLine(m.Course);
16     }
17 }
```

---

### 21.5.5 Left Outer Joins

Verknüpft zwei Mengen über einen Schlüssel. Wenn kein rechtes Element gefunden wird bleibt linkes Element trotzdem bestehen.

---

```
1 var q = from s in Students
2     join m in Markings on s.Id equals m.StudentId into match
3     from sm in match.DefaultIfEmpty()
4     select s.Name + ", " + (sm == null
5         ? "?"
6         : sm.Course + ", " + sm.Mark);
7
8 foreach (var x in q) {
9     Console.WriteLine(x);
10 }
```

---

```
1 var data = from fd in FlightDetails
2     join pd in PassengersDetails on fd.Flightno equals pd.FlightNo into joinedT
3     from pd in joinedT.DefaultIfEmpty()
4     select new {
5         nr = fd.Flightno,
6         name = fd.FlightName,
7         passengerId = pd == null ? String.Empty : pd.PassengerId,
8         passengerType = pd == null ? String.Empty : pd.PassengerType
9     }
```

---

### 21.5.6 Let

Let erlaubt das Definieren von Hilfsvariablen

---

```
1 var result =  
2     from s in Students  
3     let year = s.Id / 1000  
4     where year == 2009  
5     select s.Name + " " + year.ToString();  
6  
7 foreach (string s in result) {  
8     Console.WriteLine(s);  
9 }
```

---

### 21.5.7 Select Many

Man spricht von Select Many im Query Syntax, wenn das zweite `from` sich auf das Erste bezieht.

---

```
1 var selectMany = from a in MyArray  
2     from b in a.Split() // another array  
3     select b;
```

---

### 21.5.8 Left Outer Join mit Select Many

---

```
1 var projList =  
2     from p in projects  
3     from pl in p.ProjectManager.DefaultIfEmpty()  
4     orderby p.Name  
5     select new {  
6         Project = p.Name,  
7         Manager = (pl==null) ? "-" : pl.Name  
8     };
```

---

## 22 Expression-Bodied Members

Anstelle von einem Block können Expression-Bodied Members verwendet werden. Jedoch dürfen die Blöcke maximal ein Statement enthalten. Das Ganze funktioniert für Methoden/Operatoren, (De-)Konstruktoren, Properties/Indexers.

---

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

---

## 23 Tasks

Ein Task ist eine leichtgewichtige Variante eines Threads und repräsentiert eine asynchrone Operation. Synchrone Waits sind **gefährlich** und **blockieren** den aktuellen Thread! Beispielsweise kann ein Download den ganzen Thread für eine bestimmte Zeit blockieren. Besser mit `async/await` arbeiten.

```

1 Task task = Task
2   .Run(
3     () => { /* Some workload */ }
4   )
5   .ContinueWith( t => 1234 );
6   task.Wait();

```

### 23.1 Task vs Thread

#### Task

- Hat einen Rückgabewert
- Unterstützt «Cancellation» via Token
- Mehrere parallele Operationen in einem Task
- Vereinfachter Programmfluss mit `async / await`
- Verwendet einen Thread Pool
- Task ist eher ein «high level» Konstrukt
  - Weniger Einfluss auf Details
  - Weniger «Handarbeit»

#### Thread

- Kein Rückgabewert (Alternativen aber möglich)
- Keine «Cancellation»
- Nur eine Operation in einem Thread
- Keine Unterstützung für `async / await`
- Thread Pool muss explizit (manuell) verwendet werden
- Thread ist eher ein «low level» Konstrukt
  - Mehr Einfluss auf Details
  - Mehr «Handarbeit»

### 23.2 Task API (synchrone waits)

#### ■ Starten eines Tasks

- Via Factory (bietet weitere Optionen) →
- Via Task direkt startet mit Default-Values →

```

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

```

#### ■ Resultat abwarten (synchrone waits) →

- Busy Wait (don't do this)
  - Zur Info: Task.Result selbst würde auch schon blockieren, bietet aber keine Optionen
- Expliziter Wait() →
  - Unterstützt auch Timeouts
- Via Awaiter →
  - Optimierteres Exception Handling

```

// Busy wait for result (bad idea!)
while (!t1.IsCompleted)
{
    // Do other stuff
}
int result1 = t1.Result;

// Explicit wait
t1.Wait();
int result2 = t1.Result;

// Using awaiter
int result3 = t1.GetAwaiter().GetResult();

```

## 24 ASYNC / AWAIT

### 24.1 Synchron vs. Asynchron

#### Synchrone Operation

- Return aus der Methode nachdem die gesamte Logik durchlaufen wurde
- Blockieren den aktuellen Thread bis diese fertig gelaufen sind

#### Asynchrone Operation

- Ruft eine Methode auf ohne auf das Resultat zu warten
- Möglichkeit zur Benachrichtigung bei Fertigstellung (Callback)
- Oder: Rückgabe eines Task Objektes auf welchem Status abgefragt werden kann

### 24.2 async / await

#### async

- Markiert die Methode als asynchron
- Einschränkung Rückgabewert
  - Task (Task ohne Rückgabewert)
  - Task<T> (Task mit Rückgabewert T)
  - void (Fire and forget, i.d.R nicht verwendet)

#### await

- Alles nach 'await' ist wird vom Compiler zu einer 'Continuation' umgewandelt (es wird also nicht blockiert)
- Nur in 'async' Methoden erlaubt

**Beispiel / Download Content mit await****■ Methode zum Download eines Website-Inhalts**

- Gibt einen (voraussichtlich) noch laufenden Task zurück

**■ Anwendung**

```
Task<string> t1 = DownloadContentAsync("...hst.ch");
Task<string> t2 = DownloadContentAsync("...ost.ch");
```

```
// Do other stuff
```

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

```
// Access result
Console.WriteLine(t1.Result);
Console.WriteLine(t2.Result);
```

Array aller Resultate

Blockiert aktuellen Thread nicht

Zugriff auf einzelne Resultate

```
public static async Task<string> DownloadContentAsync(
    string url)
{
    using (WebClient client = new WebClient())
    {
        return await client.DownloadStringTaskAsync(url);
    }
}

public static Task<string> DownloadContent(
    string url)
{
    return Task.Run(() =>
    {
        using (WebClient client = new WebClient())
        {
            return client.DownloadString(url);
        }
    });
}
```

async Methode

Suffix "Async"

await Schlüsselwort

WebClient Async API

**Vor- und Nachteile async/await**

- + Code sieht immer noch synchron aus
- + Keine Continuations nötig
- + Ersetzt Multithreading für asynchrone Ausführungen
- Overhead ist relativ gross
- Lohnt sich daher erst bei längeren Operationen
- Await nicht erlaubt in lock-Statements

**24.3 Cancellation**

Integriertes Programmiermodell für das Abbrechen von Programmlogik, z.B. Abbruch einer langlaufenden Datenbankoperation, wenn API-Call abgebrochen wurde oder ein Abbruch durch 'CTRL+C'.

Nutzt die Klasse `CancellationToken`. Muss durch die gesamte Aufrufkette durchgereicht werden. Dies als letzter Parameter jeder asynchronen Methode (Best Practices).

```
1 static void ManualCancellation(
2     CancellationToken cancellationToken)
3 {
4     for (int i = 0; i < 100_000; i++)
5     {
6         // Do some work...
7
8         // Variante A
9         if (cancellationToken.IsCancellationRequested) { /* ... */ }
10
11        // Variante B
12        cancellationToken.ThrowIfCancellationRequested();
13    }
14 }
```

### 24.3.1 Cancellation Token Source

Klasse zur Erstellung und Steuerung von CancellationTokens. Kann beliebig viele Tokens emittieren.

---

```
1 CancellationTokenSource cts = new();
2 CancellationToken ct = cts.Token;
3
4 Task t1 = LongRunning(1_000, ct); // 1s tied to CTS
5 Task t2 = LongRunning(3_000, ct); // 3s tied to CTS
6 Task t3 = LongRunning(3_500, default); // 3s independent/not cancellable
7
8 await Task.Delay(2_000, ct);
9
10 Console.WriteLine("Cancelling");
11 cts.Cancel();
12 Console.WriteLine("Canceled");
13
14 async Task LongRunning(int ms, CancellationToken ct)
15 {
16     Console.WriteLine($"{ms}ms Task started.");
17     await Task.Delay(ms, ct);
18     Console.WriteLine($"{ms}ms Task completed.");
19 };
20
21 //Console output:
22 //-----
23 //1000ms Task started.
24 //3000ms Task started.
25 //3500ms Task started.
26 //1000ms Task completed.
27 //Cancelling
28 //Canceled
29 //3500ms Task completed.
```

---



## 25 Entity Framework

Entity Framework Core /EF Core ist ein O/R Mapping Framework und verbindet Objekt-Orientiertes (Domain Model) mit Relationalem (Relational Model). Man unterscheidet zwischen zwei Varianten wie die Entitätsklassen/Datenbanken erstellt werden können.

**Basis-Funktionalitäten** Mapping von Entitäten, CRUD Operationen, Key-Generierung, Caching, Change Tracking, Optimistic Concurrency, Transactions, CLI

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

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

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

### 25.1 OR Mapping

Ein OR-Mapper (Mapping) stellt die Verbindung zwischen Datenbank (Storage Entity) und Klassen (Entity Type) her. Er verbindet die Object ID mit dem Primary Key und bildet auch komplexer gebilde, wie z.B Vererbung ab. Beim Code First Ansatz **muss ein Member Id** **rsp. [ClassName]Id** existieren, damit das Entity Framework über den PrimaryKey bescheid weiss.

**Providers** Diverse relationale SQL Providers. (Von NuGet)

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

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

**Mappingansätze By Convention, By Attributes, By Fluent API** . Zuordnung von Entity Type zu Storage Entity, Property zu Column, Entity Key zu Primary Key, Foreign Key zu Relationship.

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

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

---

```

1 using (ShopContext context = new ShopContext()) {
2     var query = from c in context.Categories           SELECT TOP(2)
3     where c.Name == "Tablets"                         [c].[Id], [c].[Name],
4         [c].[Timestamp]                                FROM [Categories] AS [c]
5     select c;                                           Where [c].[Name] = N'Tablets'
6     Category tablets = query.Single()                  // ----->
7 }
```

---

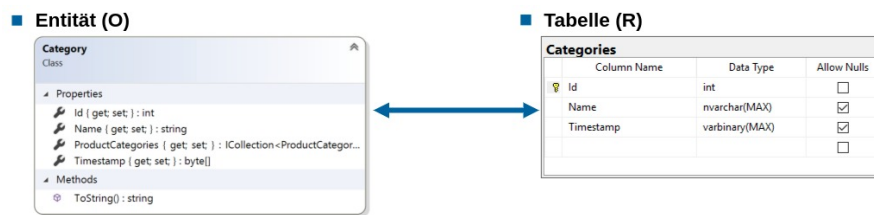


Abbildung 23: OR Mapping Entität &lt;-&gt; Tabelle

### 25.1.1 Modell

**Convention** Automatisches Mapping ohne explizite Konfiguration.

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

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

**Mapping Ansätze:** Database First, Model First, Code First

In den nachfolgenden Beispiele sind jeweils 3 Arten von komplementären Mappings vermischt. Gleiche Mappings werden mehrfach in verschiedenen Arten gezeigt.

#### Include/Exclude Entities

```

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

#### Include/Exclude Properties Convention: Alle public Properties mit Getter/Setter

```

1 public class ShopContext : DbContext {
2     public DbSet<Category> Categories { get; set; }
3     protected override void OnModelCreating(ModelBuilder modelBuilder) {
4         modelBuilder.Entity<Category>()
5             .Property(b => b.Name);
6         modelBuilder.Entity<Category>()
7             .Ignore(b => b.LoadedFromDatabase); // Fluent API - Ignore im Builder
8     }
9 }
```

---

```

9  }
10 public class Category {
11     public int Id { get; set; }
12     public string Name { get; set; }
13     [NotMapped] // Data Annotations
14     public DateTime LoadedFromDatabase { get; set; }
15 }

```

---

**Keys** Convention: Property mit dem Namen "[Entity]Id" (z.b. Category.Id, Category.CategoryId)

---

```

1 public class ShopContext : DbContext {
2     public DbSet<Category> Categories { get; set; }
3     protected override void OnModelCreating(ModelBuilder modelBuilder) {
4         modelBuilder.Entity<Category>()
5             .HasKey(e => e.Id) // Fluent API - Einzige Möglichkeit für
6             .Composite Keys // Data Annotations
7             .IsRequired();
8     }
9     public class Category {
10         [Key] // Data Annotations
11         public int Id { get; set; }
12         public string Name { get; set; }
13     }
14     public class Translation {
15         public string Language { get; set; }
16         public int CategoryId { get; set; }
17     }

```

---

**Required / Optional** Convention: Value Types werden "NOT NULL" (int), Nullable Value Types werden "NULL" (int?), Reference Types werden "NULL"

---

```

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

```

---

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

---

```

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

```

---

**Unicode** Convention: Strings sind immer Unicode (NVARCHAR)

---

```
1 .Property(e => e.Name).IsUnicode(false) // Fluent API
2 [Unicode(false)] // Data Annotations
```

---

**Precision** Convention: Pro Datentyp im Provider festgelegt, Precision: Genauigkeit / Anzahl digits total, Scale: Anzahl Nachkommastellen

---

```
1 .Property(e => e.Name).HasPrecision(10, 2) // Fluent API
2 [Precision(10, 2)] // Data Annotations
```

---

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

---

```
1 modelBuilder.Entity<Category>() // Fluent API - Non-unique Index
2 .HasIndex(b => b.Name);
3 modelBuilder.Entity<Category>() // Fluent API - Unique Index
4 .HasIndex(b => b.Name)
5 .IsUnique();
6 modelBuilder.Entity<Category>() // Fluent API - Multi-column Index
7 .HasIndex(b => new { b.Name, b.IsActive });
8
9 [Index(nameof(Name))] // Data Annotations - Non-unique Index
10 [Index(nameof(Name), IsUnique = true)] // Data Annotations - Unique Index
11 [Index(nameof(Name), nameof(IsActive))] // Data Annotations - Multi-column Index
```

---

### 25.1.2 Relationale DB (SQL Server)

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

---

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

---

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

---

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

---

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

---

```
1 // Fluent API
2 modelBuilder.Entity<Category>()
3 .Property(e => e.Name)
4 .HasColumnName("CategoryName")
5 .HasColumnType("NVARCHAR(500)") // Datentyp-Name des Zielsystems
```

---

```

6         .HasDefaultValue("---"); // Default (Wert/Gueltige SQL Expression)
7 //Data Annotation - Datentyp-Name des Zielsystems, Default Values nicht unterstuetzt.
8 [Column("CategoryName", TypeName = "NVARCHAR(500)"]
9 public string Name { get; set; }

```

---

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

---

```

1 // Fluent API - HasOne/WithMany oder HasMany/WithOne
2 modelBuilder.Entity<Product>()
3     .HasOne(p => p.Category)
4     .WithMany(b => b.Products)
5     .HasForeignKey(p => p.CategoryId)
6     .HasConstraintName("FK_Product_CategoryId");
7
8 public class Category {
9     public int Id { get; set; }
10    public ICollection<Product> Products { get; set; }
11 }
12 public class Product {
13     public int Id { get; set; }
14     public int CategoryId { get; set; }
15     //Data Annotations - Auf Navigation Property wird Foreign Key Property definiert
16     [ForeignKey(nameof(CategoryId))]
17     public Category Category { get; set; }
18 }

```

---

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

---

```

1 // Fluent API - .HasForeignKey weglassen
2 modelBuilder.Entity<Product>()
3     .HasOne(p => p.Category)
4     .WithMany(b => b.Products)
5     .HasConstraintName("FK_Product_CategoryId");
6
7 public class Category {
8     public int Id { get; set; }
9     public ICollection<Product> Products { get; set; }
10 }
11 public class Product {
12     public int Id { get; set; }
13     //Data-Annotation - Foreign Key weglassen
14     public Category Category { get; set; }
15 }

```

---

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

---

```

1 //Fluent API - .HasOne ist anders
2 modelBuilder.Entity<Product>()
3     .HasOne<Category>()
4     .WithMany(b => b.Products)
5     .HasConstraintName("FK_Product_CategoryId");
6
7 public class Category {

```

```

8     public int Id { get; set; }
9     public ICollection<Product> Products { get; set; }
10 }
11 public class Product {
12     //Data Annotations - Foreign Key + Navigation Property weglassen
13     public int Id { get; set; }
14 }

```

---

#### Relationship / Association One-to-Many / Foreign Key Convention: Foreign Key Property

---

```

1 //Fluent API - .HasOne ist anders
2 modelBuilder.Entity<Product>()
3     .HasOne<Category>()
4     .WithMany()
5     .HasForeignKey(p => p.CategoryId)
6     .HasConstraintName("FK_Product_CategoryId");
7
8 public class Category {
9     public int Id { get; set; }
10 }
11 public class Product {
12     //Data Annotations - Foreign Key + Navigation Property weglassen
13     public int Id { get; set; }
14     public int CategoryId { get; set; }
15 }

```

---

#### Relationship / Association - One-to-one Nur Reference Navigation Property, keine Collection Navigation Property, .HasOne(...).WithOne(...)

#### Relationship / Many-to-Many / Ohne Join Entity Type Convention: Wird automatisch erkannt. Mapping-Tabelle wird automatisch generiert Data Annotations: Wird nicht unterstützt

---

```

1 // Fluent API - HasMany/WithMany. Ausgehend von Category oder Product
2 modelBuilder.Entity<Category>()
3     .HasMany(p => p.Products)
4     .WithMany(b => b.Categories);
5
6 public class Category {
7     public int Id { get; set; }
8     public ICollection<Product> Products { get; set; }
9 }
10 public class Product {
11     public int Id { get; set; }
12     public ICollection<Category> Categories { get; set; }
13 }

```

---

#### Relationship / Many-to-Many / Mit Join Entity Type Convention / Data Annotations: Wird nicht unterstützt

---

```

1 // Fluent API - HasMany/WithMany. Ausgehend von Category oder Product
2 modelBuilder.Entity<Category>()
3     .HasMany(c => c.Products)
4     .WithMany(p => p.Categories)
5     .UsingEntity<ProductCategory>()
6         // Right part - ProductCategory > Product
7         pc => pc

```

```

8         .HasOne(e => e.Product)
9         .WithMany(e => e.ProductCategories)
10        .HasForeignKey(e => e.ProductId), // Optional
11        // Left part - ProductCategory > Category
12        pc => pc
13        .HasOne(e => e.Category)
14        .WithMany(e => e.ProductCategories)
15        .HasForeignKey(e => e.CategoryId) // Optional
16    );
17
18    public class Category
19    {
20        public int Id { get; set; }
21        public ICollection<Product> Products { get; set; }
22        public ICollection<ProductCategory> ProductCategories { get; set; }
23    }
24    public class Product
25    {
26        public int Id { get; set; }
27        public ICollection<Category> Categories { get; set; }
28        public ICollection<ProductCategory> ProductCategories { get; set; }
29    }
30    public class ProductCategory
31    {
32        public int ProductId { get; set; } // Optional
33        public Product Product { get; set; }
34        public int CategoryId { get; set; } // Optional
35        public Category Category { get; set; }
36        // Payload Property
37        public bool IsDeleted { get; set; } = false;
38    }

```

## 25.2 Database Context

### 25.2.1 Klasse "DbContext"

Wichtigster Teil des Entity Framework. Kombination zweier Patterns (Repository, Unit of Work). Funktionen:

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

#### DbContext Lifecycle

- **DbContext-Instanzen sollen nicht:** zu lange leben (limitierte Anzahl Connections im Client Connection Pool), geshared werden (ist nicht thread-safe).
- **Empfehlungen:** In einem "using"-Statement verwenden, Web-Applikationen (Instanz pro Request), GUI (Instanz pro Formular), Generell (Instanz pro "unit of work").

```

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

```

### 25.2.2 LINQ to Entities

#### Einfaches Beispiel

---

```

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

```

---

**Query Execution with JOIN** LINQ Query Syntax für Entity Framework: Entity Framework führt keine Queries aus sondern generiert sie nur. Datenbank führt sie dann aus. Nicht alle .NET-Expressions können auf Datenbanksyntax übersetzt werden. Vorallem solche mit eigenen Funktionen sind problematisch. Der Generierte SQL Output ist je nach Formulierung anders (Manchmal optimal, manchmal weniger.) Kann von Version zu Version ändern.

---

	Executed T-SQL Statement
1 //LINQ to Entities	SELECT
2 var query =	[c].[Name] AS [CustomerName],
3 from c in context.Customers	COUNT(*) AS [OrdersCount]
4 join o in context.Orders	FROM
5 on c.Id equals o.CustomerId	[Customers] AS [c]
6 group o by c.Name into cGroup	INNER JOIN
7 where cGroup.Key == "Angela"	[ORDERS] AS [o]
8 select new {	ON [c].[Id] = [o].[CustomerId]
9 CustomerName = cGroup.Key,	GROUP BY [c].[Name]
10 OrdersCount = cGroup.Count()	HAVING [c].[Name] = N'Angela'
11 };	
12 var result = query.ToList();	

---

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

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

#### Insert Entity Framework Core

---

```

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

```

---



**Update** Entity Framework Core

---

```

1 using (ShopContext context = new ShopContext()) {
2     Category cat = context.Categories.First();
3     cat.Name = "Changed";    // Change
4     context.SaveChanges();  // Save SQL is executed here
5 }

```

---

**Delete** Entity Framework Core

---

```

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

```

---

**25.2.4 CUD von Assoziationen**

Durch Anpassung von Navigation Properties `order.Customer = customer`

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


---

```

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

```

---

Setzen des Foreign Keys `order.CustomerId = 1`; -> einzige Variante, welche keine weiteren Datenbankzugriffe benötigt

**Insert Object Graph**


---

```

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

```

---

**Insert Related Entity**


---

```

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

```

---

```

9     context.SaveChanges();
10 }

```

---

### Change Relationship 1

```

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

```

---

### Change Relationship 2

```

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

```

---

## 25.2.5 Change Tracking

Registriert alle Änderungen an getrackten Entities. Aktualisiert den Entity State.

**State Handling** DbContext hat Methoden für das Hinzufügen und das Setzen des States im Change Tracker.

- Add() -> State "Added"
- Remove() -> State "Deleted"
- Update() -> State "Modified"
- Unchanged() -> State "Unchanged"

```

1 // New Record
2 Category cat = new Category { Name = "Laptops" }; // EntityState.Detached
3 context.Add(cat); // EntityState.Added
4 context.SaveChanges(); // EntityState.Unchanged
5 cat.Name = "Notebooks"; // EntityState.Modified
6 context.SaveChanges(); // EntityState.Unchanged
7 context.Remove(cat); // EntityState.Deleted
8 context.SaveChanges(); // EntityState.Unchanged

```

---

## 25.3 Lazy-, Eager-Loading

Es wird standardmässig Eager Loading verwendet.

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

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

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

---

```

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

```

---

```

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

```

---

```

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

```

---

```

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

```

---

```

12 .Query()
13 .Where(oi => oi.QuantityOrdered > 1) // Filter
14 .LoadAsync();

```

---

## 25.4 Optimistic Concurrency

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

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

### 25.4.1 Erkennung von Konflikten

**Pro Record Timestamp / Row Version** wird ein Timestamp oder eine Versionsnummer erstellt.

Beim Laden wird die Versionsnummer als Sessionzustand vermerkt. Validierung: Beim Zurückschreiben der Daten, wird die Session-Versionsnummer mit der Versionsnummer der DB verglichen.

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

#### Timestamp

```

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

```

---

#### Concurrency Token

```

1 public class ShopContext : DbContext {
2     public DbSet<Product> Products { get; set; }
3     protected override void OnModelCreating(ModelBuilder modelBuilder) {
4         modelBuilder.Entity<Product>()
5             .Property(p => p.Name)
6             .IsConcurrencyToken(); // Fluent API
7         modelBuilder.Entity<Product>()
8             .Property(p => p.Price)
9             .IsConcurrencyToken(); // Fluent API
10    }
11 }
12 // --- ODER ---
13 [ConcurrencyCheck] // Data Annotations
14 public string Name { get; set; }

```

---

**Konflikt erzeugen**


---

```

1 try {
2     using (var context1 = new ShopContext())
3     using (var context2 = new ShopContext()) {
4         // Client 1
5         var p1 = context1.Categories.First();
6         p1.Name = "Save 1";
7         // Client 2
8         var p2 = context2.Categories.First();
9         p2.Name = "Save 2";
10        context1.SaveChanges();
11        context2.SaveChanges(); // Fails
12    }
13 } catch (DbUpdateConcurrencyException e) { /* ... */ }
```

---

**Konfliktbehandlung** DbUpdateConcurrencyException beinhaltet fehlerhafte "Entries" (Aktuelle Werte, Originale Werte, Datenbank Werte). **Standardverfahren:** 1. Ignorieren, 2. Benutzer fragen, 3. Autokorrektur.

---

```

1 try { /* ... */ }
2 catch (DbUpdateConcurrencyException ex) {
3     foreach (var entry in ex.Entries) {
4         if (entry.Entity is Product) {
5             var proposedValues = entry.CurrentValues;
6             var databaseValues = entry.GetDatabaseValues();
7             foreach (var property in proposedValues.Properties) {
8                 var proposedValue = proposedValues[property];
9                 var databaseValue = databaseValues[property];
10                // TODO: decide which value should be written to database
11                // proposedValues[property] = <value to be saved>;
12            }
13            // Refresh original values to bypass next concurrency check
14            entry.OriginalValues.SetValues(databaseValues);
15        } else { /* ... */ }
16    }
17 }
```

---

## 25.5 Inheritance

Es gibt drei Varianten wie Vererbungen in der Datenbank abgebildet werden können

**Table per Hierarchy** Der Standard. Alles in eine Tabelle. Diskriminator entscheidet über Typ. Nicht benutzte Felder sind **null**. Nur über DbContext definierbar. Diskriminator zur Unterscheidung notwendig.

---


```

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

---

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

- **Tabelle "Products":** Id, Name, Description, Price



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

Abbildung 24: Table per Hierarchy

- **Tabelle "MobilePhones"**: Id, OperatingSystem, SupportsUmts
- **Tabelle "Tablets"**: Id, HasKeyboard

**Table per concrete Type** Tabelle für Parent (gemeinsame Felder) und Tabellen für Childs (eigene Felder) mit Verweis auf Parent. Nachteil: Duplicate columns

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

## 25.6 Database Migration

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

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

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

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

---

```

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

---

## 25.7 Data Type Mappings

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

Abbildung 25: Data Type Mappings

## 26 gRPC: Google Remote Procedure Call

Neue Standard-Technologie für Backend-Kommunikation in .NET Core. Hohe Performance von zentraler Bedeutung. gRPC ist ein Software Development Kit (SDK), Request-Response-orientiert, Plattform- und Sprach-neutral. Kommunikation über HTTP/2 (Multiplexing (mehrere gRPC Calls pro TCP/IP Connection), Bidirectionales Streaming, Parallele Requests und Responses in einer einzigen TCP Verbindung, Kommunikation wegen HTTPS verschlüsselt).

**RPC** ermöglicht Client / Server Kommunikation und wird in fast allen verteilten Systemen verwendet. **Grundprinzipien:** Einfache Service-Definition, Sprach-Unabhängigkeit, Problemlose Skalierbarkeit, Bi-direktionales Streaming, integrierte Authentisierungsmechanismen

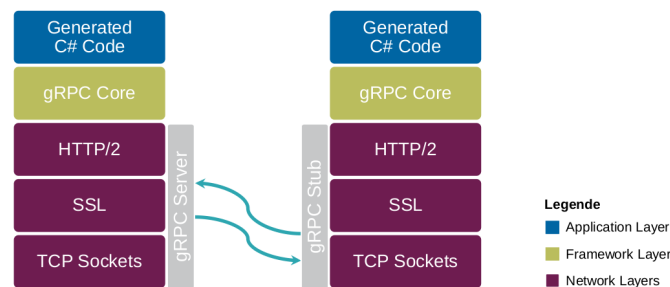


Abbildung 26: gRPC Architektur

### 26.1 Protocol Buffers

**Interface Definition Language (IDL)** Beschreibt ein Service Interface plattform- und sprach-neutral.

**Data Model** Beschreibt Messages resp. Request- und Response-Objekte.

**Wire Format** Beschreibt das Binärformat zur Übertragung.

**Serialisierungs-/Deserialisierungs-Mechanismen**

**Service-Versionierung**

**Protocol Buffers / Fields / Scalar Value Types** .proto und C#

.proto	C#	Default	
double	double	0	
float	float	0	
int32	int	0	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.
int64	long	0	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.
bool	bool	false	
string	string	<empty>	A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than 232.
bytes	ByteString	<empty>	May contain any arbitrary sequence of bytes no longer than 232.
[...]	Weitere Ganzzahltypen wie [u s][int long][32 64] oder fixed[32 64] <a href="https://developers.google.com/protocol-buffers/docs/proto3#scalar">https://developers.google.com/protocol-buffers/docs/proto3#scalar</a>		



**Proto Files** Datei-Endung \*.proto, 1 oder mehr Services und mehrere Service-Methoden pro Service. 1 oder mehr Message Type Fields (Field definiert durch Type, Unique Name, Unique Field Number). Service-Methoden haben immer genau 1 Parameter und 1 Rückgabewert. Null-Werte mit `google.protobuf.Empty`.

---

```

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

```

---

### Fields/Repeated Fields

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

---

```

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

```

---

**Enumerations** Definition innerhalb von Message oder Proto-File root, Enum-Member mit Wert 0 muss existieren (Default Value)

---

```

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

```

---

**Timestamp** `Timestamp.FromDateTime(DateTime.UtcNow);`

---

```

1 message TimestampResponse {
2     repeated google.protobuf.Timestamp results = 1;
3 }
4 Timestamp von = Timestamp.FromDateTime(new DateTime(2021, 07, 15, 0, 0, 0,
    DateTimeKind.Utc));

```

---

**Reserved keyword**

---

```

1 message SearchRequest {
2     reserved 1, 3, 20 to 30;
3     reserved "page_number",
4         "result_per_page";
5
6     string query = 1; // Compilerfehler
7     int32 page_number = 2; // Compilerfehler
8     int32 result_per_page = 3; // Compilerfehler
9 }

```

---

## 26.2 gRPC C# API

### Protocol Buffers Compiler

- protoc.exe mit Plugins für C# Code Generierung
- Automatisch in Build Pipeline eingebunden
- NuGet Package `grpc.Tools`
- Proto-Compiler Output in "obj" Folder

**Aufbau Server Projekt** ASP.NET Core Projekt, erstellen via CLI (`dotnet new grpc`), NuGet Packages (`Grpc.AspNetCore`).

---

```

1 // Server Projekt
2 <Project Sdk="Microsoft.NET.Sdk.Web">
3     <ItemGroup>
4         <Protobuf Include="Protos\greet.proto" GrpcServices="Server"> // zu
5             generierende Klassen (Server)
6         <Link>Protos\greet.proto</Link> // File Link
7     </Protobuf>
8 </ItemGroup>
9 <ItemGroup>
10     <PackageReference Include="Grpc.AspNetCore" Version="..." /> // Packages
11 </ItemGroup>
12 </Project>

```

---

**Aufbau Client Projekt** Beliebiges Projekt, Proto-File als Kopie/Link eibinden, NuGet Packages (`Grpc.Net.Client`, `Google.Protobuf`, `Grpc.Tools`)

---

```

1 // Client Projekt
2 <Project Sdk="Microsoft.NET.Sdk.Web">
3     <ItemGroup>
4         <Protobuf Include="..\<relative_path>\greet.proto" GrpcServices="Client"> //
5             Protobuf Include (relativer Pfad) und zu generierende Klassen (Client)
6         <Link>Protos\greet.proto</Link>

```

---

```

6     </Protobuf>
7 </ItemGroup>
8 <ItemGroup>
9     <PackageReference Include="Google.Protobuf" Version="..." />
10    <PackageReference Include="Grpc.Net.Client" Version="..." />
11    <PackageReference Include="Grpc.Tools" Version="...">[...]
```

---

## 26.3 Beispiel Customer Service

2 Services Customer Service und Order Service

### Service/Proto-File

```

1 // Customer Service
2 service CustomerService {
3     rpc GetCustomers (google.protobuf.Empty) //Liste aller Kunden
4         returns (GetCustomersResponse);
5     rpc GetOrders (GetOrdersRequest) //einzelner Kunde
6         returns (GetOrdersResponse);
7 }
8
9 // Order Service
10 service OrderService {
11     rpc GetCustomer (GetCustomerRequest) //Alle Bestellungen zu einem Kunden
12         returns (GetCustomerResponse);
13 }
```

---

### Messages/Proto-File

```

1 // Customer Messages
2 message GetCustomersResponse {
3     repeated CustomerResponse data = 1;
4 }
5 message GetCustomerResponse {
6     CustomerResponse data = 1;
7 }
8 message GetCustomerRequest {
9     int32 id_filter = 1;
10    bool include_orders = 2;
11 }
12 message CustomerResponse {
13     int32 id = 1;
14     string first_name = 2;
15     string last_name = 3;
16     Gender gender = 4;
17     repeated OrderResponse orders = 10;
18 }
19 enum Gender { UNKNOWN = 0; FEMALE = 1; MALE = 2; }
20
21 // Order Messages
22 message GetOrdersRequest {
23     int32 customer_id_filter = 1;
24 }
25 message GetOrdersResponse {
26     repeated OrderResponse data = 1;
27 }
28 message OrderResponse {
```

```

29     string product_name = 1;
30     int32 quantity = 2;
31     double price = 3;
32 }

```

---

### Service Implementation

```

1 //Customer Service
2 public class MyCustomerService : CustomerService.CustomerServiceBase {
3     public override async Task<GetCustomersResponse> GetCustomers(Empty request,
4         ServerCallContext context) { /* ... */ }
5     public override async Task<GetCustomerResponse> GetCustomer(GetCustomerRequest
6         request, ServerCallContext context) { /* ... */ }
7 }
8 //Order Service
9 public class MyOrderService : OrderService.OrderServiceBase {
10     public override async Task<GetOrdersResponse> GetOrders(GetOrdersRequest request,
11         ServerCallContext context) { /* ... */ }
12 }

```

---

### Client-Implementation (Customer)

```

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

```

---

### Client-Implementation (Order)

```

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

```

---

## 26.4 Streams

**Unterstützt 3 Modi** Server Streaming Call (Server > Client), Client Streaming Call (Client > Server), Bidirectional/Duplex Streaming Call. Schlüsselwort "stream" vor Typenbezeichnung.

**Reliability** End-to-End Reliability (garantiertes Ausliefern der Nachrichten gewährleistet), Ordered Delivery (Reihenfolge gewährleistet)

---

```

1 service FileStreamingService {
2     rpc ReadFiles (google.protobuf.Empty) //Server Streaming Call
3         returns (stream FileDto);
4     rpc SendFiles (stream FileDto) //Client Streaming Call
5         returns (google.protobuf.Empty);
6     rpc RoundtripFiles (stream FileDto) //Bi-directional / Duplex Streaming Call
7         returns (stream FileDto);
8 }
9 message FileDto {
10     string file_name = 1;
11     int32 line = 2;
12     string content = 3;
13 }

```

---

### Server Streaming Call (Server > Client) Service

---

```

1 //Client
2 using (AsyncServerStreamingCall<FileDto> call = client.ReadFiles(new Empty())) {
3     await foreach (FileDto message in call.ResponseStream.ReadAllAsync()) { // Read
4         last written chunk
5         WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Line Content:
6             {message.Content}");
7     }
8 }
9 //Service
10 public override async Task ReadFiles(Empty request, IServerStreamWriter<FileDto>
11     responseStream, ServerCallContext context) {
12     string[] files = Directory.GetFiles(@"...");
13     foreach (string file in files) { // Files Loop
14         string content; int line = 0;
15         using StreamReader reader = File.OpenText(file);
16         while ((content = await reader.ReadLineAsync()) != null) { // Line-Loop
17             line++;
18             FileDto reply = new FileDto {
19                 FileName = file, Line = line, Content = content,
20             };
21             await responseStream.WriteAsync(reply); // Write to Stream
22         }
23     }
24 }

```

---

### Client Streaming Call (Client > Server) Service

---

```

1 //Service
2 public override async Task<Empty> SendFiles(IAsyncStreamReader<FileDto>
3     requestStream, ServerCallContext context) { // Request Stream
4     await foreach (FileDto message in requestStream.ReadAllAsync()) { // Read last
5         written chunk
6         WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Line Content:
7             {message.Content}");
8     }
9     return new Empty(); // Empty Result
10 }
11 //Client
12 using (AsyncClientStreamingCall<FileDto, Empty> call = client.SendFiles()) {

```

```

11 string[] files = Directory.GetFiles(@"Files");
12 foreach (string file in files) { //File-Loop
13     string content; int line = 0;
14     using StreamReader reader = File.OpenText(file);
15     while ((content = await reader.ReadLineAsync()) != null) { // Line-Loop
16         line++;
17         FileDto reply = new FileDto {
18             FileName = file, Line = line, Content = content,
19         };
20         await call.RequestStream.WriteAsync(reply); // Write to Stream
21     }
22 }
23 // Required!
24 await call.RequestStream.CompleteAsync(); // Close Stream
25 await call; // Wait until all requests are submitted
26 }

```

---

### Bi-directional (Client > Server) Client

```

1 //Service
2 public override async Task RoundtripFiles(IAsyncStreamReader<FileDto> requestStream,
3     IServerStreamWriter<FileDto> responseStream, ServerCallContext context) {
4     await foreach (FileDto message in requestStream.ReadAllAsync()) { // Read last
5         // written chunk
6         await responseStream.WriteAsync(message); // Write to Stream
7         WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Line Content:
8             {message.Content}");
9     }
10 }
11 //Client
12 using (AsyncDuplexStreamingCall<FileDto, FileDto> call = client.RoundtripFiles()) {
13     // Read
14     Task readTask = Task.Run(async () => { // Read Task (no await)
15         await foreach (FileDto message in call.ResponseStream.ReadAllAsync()) { // Read
16             // last written chunk
17             WriteLine($"File: {message.FileName}, Line Nr: {message.Line}, Line Content:
18                 {message.Content}");
19         }
20     });
21
22     // Write
23     string[] files = Directory.GetFiles(@"Files");
24     foreach (string file in files) { // File-Loop
25         string content; int line = 0;
26         using StreamReader reader = File.OpenText(file);
27         while ((content = await reader.ReadLineAsync()) != null) { // Line-Loop
28             line++;
29             FileDto reply = new FileDto {
30                 FileName = file, Line = line, Content = content,
31             };
32             await call.RequestStream.WriteAsync(reply); // Write to Stream
33         }
34     }
35     await call.RequestStream.CompleteAsync(); // Close Stream
36     await readTask; // Wait until all requests are submitted
37 }

```

---

## 26.5 Special Types

**Empty** Platzhalter für Null-Werte `var e = new Empty();`

**Timestamp** UTC Zeitstempel - muss UTC sein, es darf nicht `DateTime.Now` verwendet werden.

`Timestamp ts = Timestamp.FromDateTime(DateTime.UtcNow)`

**Collections - Repeated Fields** Generiert ein Repeated Field Property, ist read only

---

```

1 message RepeatedResponse { repeated string results = 1; }
2
3 // Anwendung
4 // Single Add
5 response.Results.Add("Hello");
6 // Multiple Add
7 string[] arr = { "A", "B" };
8 response.Results.AddRange(arr);
9 // Compilerfehler
10 response.Results = new RepeatedField<string>();

```

---

**Collections - Map Fields** Generiert ein Map Field Property, ist read only

---

```

1 message MapResponse { map<int32, string> string results = 1; }
2
3 // Anwendung
4 var response = new MapResponse();
5 response.Results.Add(1, "Hello");
6 response.Results.ContainsKey(1);
7 string result = response.Results[1];

```

---

**Oneof** - Lässt eine Auswahl von Typen zu

---

```

1 message OneofResponse {
2     oneof results {
3         string image_url = 1;
4         bytes image_data = 2;
5     }
6 }

```

---

**Bytes** - Binärer Datentyp

---

```

1 message BinaryResponse {
2     bytes results = 1;
3 }
4
5 // Empty ByteString
6 ByteString bs = ByteString.Empty;
7 // To ByteString
8 bs = ByteString.CopyFromUtf8("X");
9 bs = ByteString.CopyFrom("X", Encoding.Unicode);
10 bs = ByteString.CopyFrom(new byte[0]);
11 bs = ByteString.FromStream(null); // Any Stream
12 // From ByteString
13 string r1 = bs.ToStringUtf8();
14 string r2 = bs.ToString(Encoding.Unicode);
15 byte[] r3 = bs.ToArray();
16 bs.WriteTo(null); // Any Stream

```

---

**Any** - Repräsentiert einen beliebigen Wert `google.protobuf.Any results = 1;`

## 26.6 Exception Handling

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

```

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

```

### Status Codes

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.)

Abbildung 27: Statuscode 1/2

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

Abbildung 28: Statuscode 2/2



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

---

```

1 public override async Task<Empty> Unhandled(Empty request, ServerCallContext context)
2     {
3         throw new Exception("Unhandled Exception");
4     }

```

---

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

---

```

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

```

---

## 26.7 Cancellation

Cancellation Token kann beim Aufruf mitgegeben werden. Parameter wird automatisch generiert. Zugriff auf Server via ServerCallContext.

---

```

1 // Client
2 CancellationTokenSource tokenSource = new();
3 CancellationToken token = tokenSource.Token;
4
5 tokenSource.CancelAfter(1111); // Zeitbasiert. Alternative token.Cancel()
6 await client.DummyAsync(
7     new Empty(),
8     cancellationTokens: token
9 );
10
11 // Zugriff auf Deadline server-seitig
12 public override Task<Empty> Dummy(
13     Empty request, ServerCallContext context)
14 {
15     Console.WriteLine(context.CancellationToken.IsCancellationRequested);
16 }

```

---

## 27 Reflection

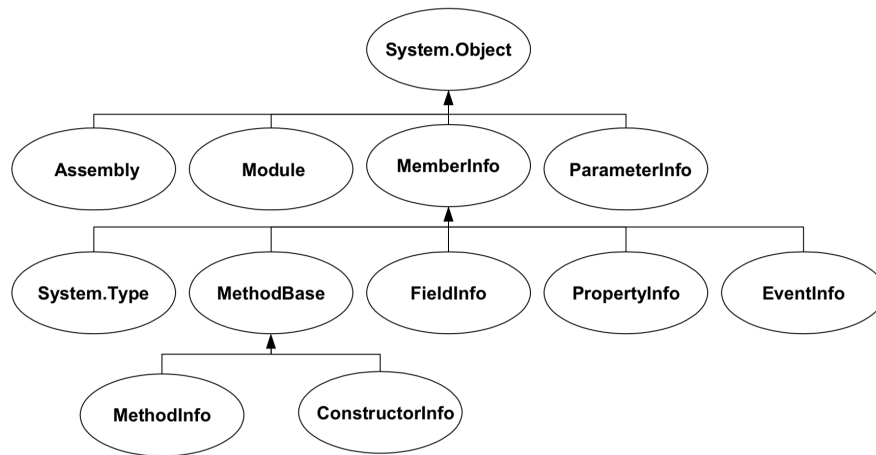


Abbildung 29: Typ-Hierarchie

### 27.1 Anwendungen

**Metadaten erstellen** Darstellung der Metadaten in Tools

**Type Discovery** Suchen und Instanzieren von Typen, Zugriff auf dynamische Datenstrukturen.

**Late Binding (Methods/Properties)** Aufruf von Methoden / Properties nach Type Discovery

**Reflection Emit / Code-Emittierung** Erstellen von Typen inkl. Members zur Laufzeit

**Alle Typen in der Common Language Runtime (CLR) sind selbst-definierend**

**Nicht zugreifbare Members auch einsehbar** z.b. private Felder

**Klasse "System.Type"** Einstiegspunkt aller Reflection-Operationen, repräsentiert einen Typen mit all seinen Eigenschaften, abstrakte Basisklasse, "System.RuntimeType" wird jeweils verwendet.

**Ermitteln von "System.Type" via** `obj.GetType()`, `typeof("classname")`.

---

```

1 this.GetType() // implemented on object
2 typeof(MyClass) || typeof(int)

```

---

### 27.2 Type Discovery

Suche alle Typen in einem Assembly.

Listing 7: Reflection: Type Discovery

---

```

1 Assembly a01 = Assembly.Load("mscorlib, PublicKeyToken=b77a5c561934e089,
  Culture=neutral, Version=4.0.0.0");

```

---

```

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

```

---

### 27.3 Member auslesen

Das Auslesen von Members kann mit BindingFlags gefiltert werden.

Listing 8: Reflection: Members auslesen

```

1 Type type = typeof(Counter);
2 MemberInfo[] miAll = type.GetMembers();
3 foreach (MemberInfo mi in miAll) {
4     Console.WriteLine("{0} is a {1}", mi, mi.MemberType);
5 }
6 Console.WriteLine("-----");
7 PropertyInfo[] piAll = type.GetProperties();
8 foreach (PropertyInfo pi in piAll) {
9     Console.WriteLine("{0} is a {1}", pi, pi.PropertyType);
10 }
11
12 // ex2: filter members according to BindingFlags or Filtername
13 Type type = typeof(Assembly);
14 BindingFlags bf =
15     BindingFlags.Public |
16     BindingFlags.Static |
17     BindingFlags.NonPublic |
18     BindingFlags.Instance |
19     BindingFlags.DeclaredOnly;
20
21 System.Reflection.MemberInfo[] miFound = type.FindMembers(
22     MemberTypes.Method, bf, Type.FilterName, "Get*"
23 );

```

---

### 27.4 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.

Listing 9: Reflection: Field Info

```

1 Type type = typeof(Counter);
2 Counter c = new Counter(1);
3
4 // All Fields
5 FieldInfo[] fiAll = type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic);
6
7 // Specific Field

```

---

```

8  FieldInfo fi = type.GetField("countValue", BindingFlags.Instance |
   BindingFlags.NonPublic);
9
10 int val01 = (int) fi.GetValue(c);
11 c.Increment();
12 int val02 = (int) fi.GetValue(c);
13 fi.SetValue(c, -999);

```

---

## 27.5 Property Information

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

Listing 10: Reflection: Property Info

---

```

1  Type type = typeof(Counter);
2  Counter c = new Counter(1);
3
4  // All Properties
5  PropertyInfo[] piAll = type.GetProperties();
6
7  // Specific Property
8  PropertyInfo pi = type.GetProperty("CountValue");
9
10 int val01 = (int)pi.GetValue(c);
11 c.Increment();
12 int val02 = (int)pi.GetValue(c);
13 if(pi.CanWrite) { pi.SetValue(c, -999); }

```

---

## 27.6 Method Info

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

Listing 11: Reflection: Method Info

---

```

1  Type type = typeof(Counter);
2  Counter c = new Counter(1);
3
4  // All Methods
5  MethodInfo[] miAll = type.GetMethods();
6
7  // Specific Method
8  MethodInfo mi = type.GetMethod("Increment");
9  mi.Invoke(c, null);

```

---

## 27.7 Constructor Info

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

Listing 12: Reflection: Constructor Info

---

```

1  Type type = typeof(Counter);
2  Counter c = new Counter(1);
3

```

---

---

```

4 // All Constructors
5 var ciAll = type.GetConstructors();
6
7 // Specific Constructor Overload 01
8 ConstructorInfo ci01 = type.GetConstructor(new[] { typeof(int) });
9 Counter c01 = (Counter)ci01.Invoke(new object[] { 12 });
10
11 // Specific Constructor Overload 02
12 ConstructorInfo ci02 =
13     type.GetConstructor(BindingFlags.Instance|BindingFlags.NonPublic, null, new
        Type[0], null);
14 Counter c02 = (Counter)ci02.Invoke(null);

```

---

## 27.8 Example of Reflection Usage

---

```

1 using System.Reflection;
2
3 namespace TestReflection {
4     class Program {
5         static void Main(string[] args) {
6             var ass=Assembly.LoadFrom("Autos.dll");
7             var t = ass.GetType("Autos.FastCars");
8             var c=Activator.CreateInstance(t, new object[] { "Lamborghini"});
9             var m = t.GetMethod("AutoFahren");
10            m.Invoke(c, new object[]{});
11        }
12    }
13 }

```

---

## 27.9 Attributes

Listing 13: Reflection: Attributes

---

```

1 Type type = typeof(MyMath);
2
3 // All Class Attributes
4 object[] aiAll = type.GetCustomAttributes(true);
5
6 // Check Definition
7 bool aiDef = type.IsDefined(typeof(BugfixAttribute));

```

---

## 28 Attributes

Attributes sind das C# Pendant zu den Java Annotations. Bei Attributen geht es um die aspekt-orientierte Programmierung. z.B Erweiterung eines Attributes um eine Aspekt Serialisierung, Transactions, etc. Es können auch eigene Attribute geschrieben werden. Diese leiten immer von `System.Attribute` ab. Attribute können mit über Reflection abgefragt werden.

Listing 14: Attributes

```
1 [DataContract, Serializable]
2 [Obsolete]
3 // Etc.
4 public class Auto {
5     [DataMember]
6     public string Marke { get; set; }
7     [DataMember]
8     public string Typ { get; set; }
9 }
10
11 // Beliebige Attribute
12 [DataContract][Serializable] <=> [DataContract, Serializable]
13
14 // Parameter
15 [DataContract] // Ohne Parameter
16 [DataContract(Name="MyParam")] // Named Parameter
17 [Obsolete("Alt!", true)] // Positional Parameter
18 [Obsolete("Alt!", IsError=true)] // Mixed
```

### 28.1 Anwendungsfälle

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

### 28.2 Typen

Man unterscheidet zwei Typen von Attributen

1. Intrinsic Attributes: Kommen bereits mit der CLR mit
2. Custom Attributes: Eigens definierte Attribute

## 28.3 Eigene Attribute

Bei der Deklaration können die Objekte eingegrenzt werden, auf die das Attribute angewendet werden kann. Jedes Attribute muss als Postfix "Attribute" haben. (xxAttribute). Beim Verwenden wird der Postfix jedoch weggelassen.

---

```

1 // declaration
2 [AttributeUsage(           //definiert wo attribute verwendet werden duerfen
3     AttributeTargets.Class |
4     AttributeTargets.Constructor |
5     AttributeTargets.Field |
6     AttributeTargets.Method |
7     AttributeTargets.Property,
8     AllowMultiple = true)]
9 public class BugfixAttribute : Attribute
10 {
11     public BugfixAttribute(int bugId, string programmer, string date) {...}
12     public int BugId { get; }
13     public string Date { get; }
14     public string Programmer { get; }
15     public string Comment { get; set; }
16 }
17
18 // usage
19 [Bugfix(121, "MichaelWieland", "14/12/16")]

```

---

### CSV-Filter

---

```

1 // list
2 var a = new List<Address> {
3     new Address("Hans", "Strasse 16", "8645", "Jona") ,
4     new Address("Hans2", "Strasse 2", "8645", "Jona")
5 }
6 Writer.SaveToCsv(a, @"C:\Temp\test.csv");
7
8 // address
9 public class Address {
10     [CsvName("Name"), Uppercase]
11     public string Name { get; set; }
12     [CsvName("Strasse"), Lowercase]
13     public string Street { get; set; }
14     [CsvName("Plz")]
15     public string Postcode { get; set; }
16     ...
17 }
18
19 // Custom Attributes
20 public class CsvNameAttribute : Attribute { // Mapping eines Properties auf CSV Spalte
21     public string Name { get; set; }
22     public CsvAttribute(string name) {
23         Name = name;
24     }
25 }
26
27 public interface IStringFilter { // beschreibt beliebigen Filter
28     string Filter(string arg);
29 }
30 public class UppercaseAttribute : Attribute : IStringFilter { // Implementation von
31     IStringFilter
32     public string Filter(string arg) {
33         return arg.ToUpper();
34     }
35 }

```

---

```

33     }
34 }
35 public class LowercaseAttribute : Attribute : IStringFilter { // Implementation von
    IStringFilter
36     public string Filter(string arg) {
37         return arg.ToLower();
38     }
39 }

```

---

## 28.4 Reflection Emit

Reflection.Emit erlaubt neue Assemblies und Typen zur Laufzeit zu erzeugen und sofort zu verwenden. Erzeugen von Assemblies, neuen Modulen, neuen Typen, symbolischer Metainformationen für bestehende Module.

### Wichtigste Klassen

- AssemblyBuilder ⇒ Assemblies definieren
- ModuleBuilder ⇒ Module definieren
- TypeBuilder ⇒ Typen definieren
- MethodBuilder ⇒ Methoden definieren
- ILGenerator ⇒ IL-Code erzeugen

### Assembly inkl. Module definieren

---

```

1 AssemblyName assemblyName = new AssemblyName("Hsr.EmitDemoAssembly");
2 AssemblyBuilder assemblyBuilder =
    AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName,
    AssemblyBuilderAccess.RunAndSave);
3 ModuleBuilder moduleBuilder =
    assemblyBuilder.DefineDynamicModule("Hsr.EmitDemoModule",
    "Hsr.EmitDemoAssembly.dll");

```

---

### Klasse "EmitDemo" definieren

---

```

1 TypeBuilder typeBuilder = moduleBuilder.DefineType("EmitDemo", TypeAttributes.Public);

```

---

### Methode "SayHello" definieren

---

```

1 Type[] paramTypes = { typeof(string) };
2 Type retType = typeof(string);
3 MethodBuilder methodBuilder = typeBuilder.DefineMethod("SayHelloTo",
4     MethodAttributes.Public | MethodAttributes.Virtual, retType, paramTypes);

```

---



### Einfügen der MSIL Operationen in die neu erzeugte Methode

---

```
1 ILGenerator ilGen = methodBuilder.GetILGenerator();
2 ilGen.Emit(OpCodes.Ldstr, "Hello ");
3 ilGen.Emit(OpCodes.Ldarg_1);
4 MethodInfo mi = typeof(string).GetMethod("Concat", new[] { typeof(string),
    typeof(string) });
5 ilGen.Emit(OpCodes.Call, mi);
6 ilGen.Emit(OpCodes.Ret);
```

---

### Klasse "EmitDemo" erzeugen

---

```
1 typeBuilder.CreateType();
```

---

### Methode "SayHello" ausführen

---

```
1 MethodInfo method = typeBuilder.GetMethod("SayHelloTo", new[] { typeof(string) });
2 object obj = Activator.CreateInstance(typeBuilder);
3 object ret = method.Invoke(obj, new object[] { "HSR" });
4 Console.WriteLine(ret);
```

---

### Assembly "Hsr.EmitDemoAssembly.dll" abspeichern

---

```
1 assemblyBuilder.Save(assemblyName.Name + ".dll");
```

---