

Zusammenfassung

Microsoft Technologien

Michael Wieland

Hochschule für Technik Rapperswil

28. August 2017

Mitmachen

Falls du an diesem Dokument mitarbeiten möchtest, kannst du es auf GitHub unter <https://github.com/michiwieland/hsr-zusammenfassungen> forken.

Lizenz

"THE BEER-WARE LICENSE" (Revision 42): <michi.wieland@hotmail.com> 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 me a beer in return. Michael Wieland

Inhaltsverzeichnis

1. Der Heilige Gral	6
1.1. Reference oder Value	6
1.2. Lamdas	7
1.3. Delegates, Events	7
1.4. Extension Methods	7
1.5. LINQ	9
1.6. Entity Framework	10
1.7. WCF	11
1.7.1. Server	11
1.7.2. Client	13
2. .NET Framework	14
2.1. CLR: Common Language Runtime	14
2.2. CTS: Common Type System	14
2.3. CLS: Common Language Specification	14
2.4. MSIL: Microsoft Intermediate Language	15
2.5. JIT: Just in Time Compilation	16
2.6. Assembly / Komponenten	16
2.6.1. Module	16
2.6.2. References	16
2.7. Kompilierung	17
2.8. Garbage Collection	17
2.8.1. Generationen	17
2.8.2. Deterministic Finalization	18
2.8.3. Finalizer	18
2.8.4. Object Pinning	18
2.8.5. Weak References	19
2.8.6. Memory Leaks	19
3. Visual Studio 15	20
3.1. Solution	20
3.2. Umbenennen	20
3.3. Ordnerstruktur	20
4. C# Grundlagen	21
4.1. Unterschiede zu Java	21
4.2. Naming Conventions	21
4.3. Sichtbarkeiten	21
4.4. Operatoren	22
4.5. Pre-, Post-Inkrement	23
4.6. Statements	23
4.6.1. If Else If Else	23
4.6.2. Switch Case	23
4.6.3. Loops	23
4.6.4. Kommentare	23
4.7. Datentypen	24
4.7.1. Casts	25
4.7.2. Reference Types / Referenztypen	26

4.7.3. Value Types / Werttypen	26
4.8. Nullable Types	27
4.9. Boxing / Unboxing	28
4.10. Object	28
4.11. String	28
4.12. Arrays	29
4.13. Indexer	29
4.14. List	29
4.15. Namespaces	29
5. Variablen und Properties	30
5.1. Konstanten	30
5.2. ReadOnly	30
5.3. Properties	30
5.3.1. Auto Properties	30
5.3.2. Properties direkt initialisieren	31
6. Methoden	32
6.1. Overloading	32
6.2. Call by value	32
6.3. Call by reference	32
6.4. Out Parameter	32
6.5. Params Array	32
6.6. Optionale Parameter (Default Values)	33
6.7. Named Parameter	33
6.8. Virtual	33
6.9. Override	33
6.10. New	34
6.11. Sealed	34
7. Klassen, Structs	35
7.1. Klassen	35
7.1.1. Type Casts	35
7.1.2. Operatoren Überladen	35
7.1.3. Partial Class	36
7.2. Abstrakte Klassen	36
7.3. Sealed Klassen	36
7.4. Statische Klassen	36
7.5. Structs	37
7.6. Konstruktoren	37
7.7. Initialisierungsreihenfolge	38
7.8. Destruktoren	39
7.9. Operator Overloading	39
8. Interfaces	40
9. Enum	41
10. Generics	42
10.1. Type Constraints	42

10.2. Typprüfungen	42
11. Delegates	43
11.1. Multicast Delegates	44
11.2. Anonyme Methoden	44
11.3. Events	45
11.4. EventHandler	46
12. Lamdas	47
12.1. Closure	47
13. Iteratoren	48
13.1. Foreach Loop	48
13.2. Iterator Interface	48
13.3. Iterator Methoden und Yield Return	49
13.4. Extension Methods	50
14. Exceptions	51
14.1. Exception Filter	51
15. LINQ: Language Integrated Query	52
15.1. Extensions Syntax (Fluent Syntax)	53
15.1.1. LINQ Extension Methods	53
15.1.2. SelectMany	54
15.2. Query Expressions Syntax	55
15.2.1. Gruppierung	56
15.2.2. Inner Joins	57
15.2.3. Group Joins	57
15.2.4. Left Outer Joins	57
15.2.5. Let	58
15.2.6. Select Many	58
15.2.7. Left Outer Join mit Select Many	58
16. Direct Initialization	59
16.1. Object Initializers	59
16.2. Collection Initializers	59
16.3. VAR: Anonymous Types	59
17. Entity Framework	60
17.1. OR Mapping	61
17.1.1. Vererbung Mappings	61
17.2. Database First	62
17.3. Code First	62
17.3.1. Attribute	62
17.4. Model Builder	63
17.5. Lazy-, Eager-Loading	63
17.6. DB Context	64
17.7. Entity Key (Object Identity)	64
17.8. Optimistic Concurrency	64

18. WCF: Windows Communication Foundation	65
18.1. Gemeinsame Assembly	65
18.2. Kommunikation	65
18.3. Endpoint	66
18.4. MEX: Meta Data Exchange Endpoint	66
18.5. Contracts	67
18.6. Vererbung	68
18.7. Serialisierung von Referenzen	69
18.8. Faults	69
18.9. Service Hosting	70
18.10 Asynchron und Synchron Kommunikation	71
18.11 Instanzmodelle	72
18.12 Reliability	73
18.13 Concurrency	73
18.14 Transaktionen	73
18.15 Throttling	74
19. Reflection	75
19.1. Type Discovery	75
19.2. Member auslesen	75
19.3. Field Information	76
19.4. Property Information	76
19.5. Method Info	77
19.6. Constructor Info	77
19.7. Attributes	77
19.8. Instanz von Klasse erstellen	78
19.9. Praktisches Beispiel für den Einsatz von Reflection	78
20. Attributes	79
20.1. Anwendungsfälle	79
20.2. Typen	79
20.3. Eigene Attribute	80
A. Listings	82
B. Abbildungsverzeichnis	83
C. Tabellenverzeichnis	84

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.

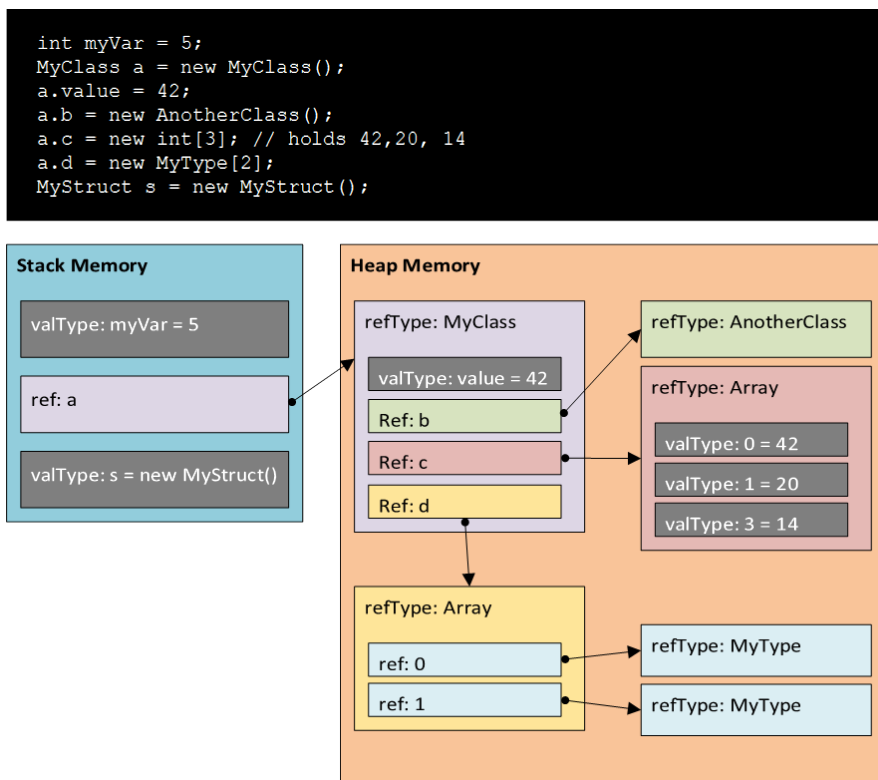


Abbildung 1: Referenz und Value Typen

1.2. Lamdas

- Lamdas sind anonyme lokale Funktionen welche auch als Argument oder Rückgabewert von Funktionen verwendet werden können.
- Lamdas werden in `Func<[param_type], [return_type]> myLamda;` gespeichert, wobei der letzte Typ in den spitzen Klammern der Rückgabe Typ ist.

```
1 //verschachtelte Lamda
2 Func<int,Func<int,string>> curry = a => b => (a + b).ToString();
```

1.3. Delegates, Events

- Der erste Parameter ist bei EventHandler immer immer das `this` Objekt!
- In einem Event können mehrere Lamda/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 // defie 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 DB-Context 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 Framework

- Es werden aktuell über 30 Sprachen unterstützt
- Der Source Code wird in die Intermediate Language (IL: 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)

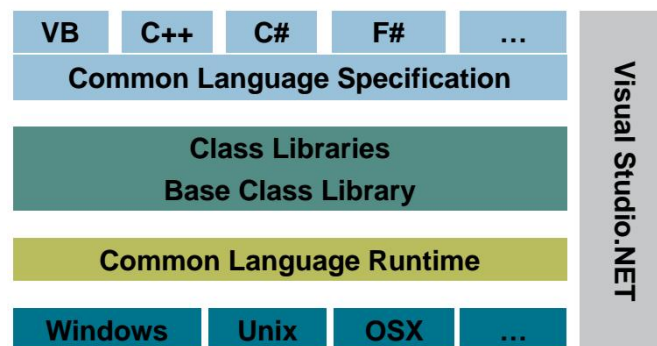


Abbildung 2: .NET Framework Architektur

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 Threading. 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.

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.

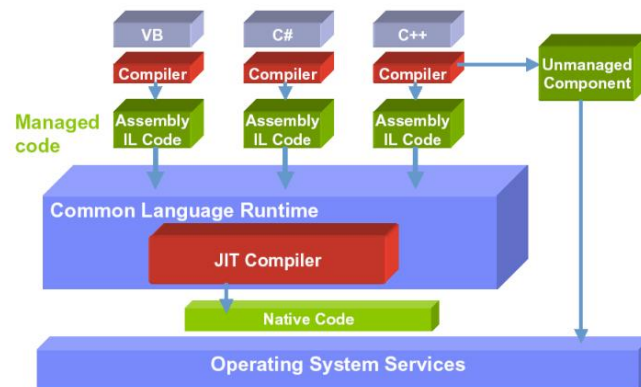


Abbildung 3: CLR: Common Language Runtime Architektur

2.4. MSIL: Microsoft Intermediate Language

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

1. Sprachspezifischer Kompilier 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.)

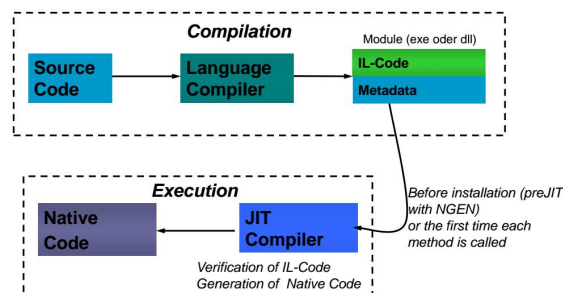


Abbildung 4: MSIL Kompilierung

2.5. JIT: Just in Time Compilation

Bei der JIT Kompilierung wird die aufgerufene Methode vor dem Methodenaufwurf kompiliert und der IL-Code durch nativen Code ersetzt.

2.6. Assembly / Komponenten

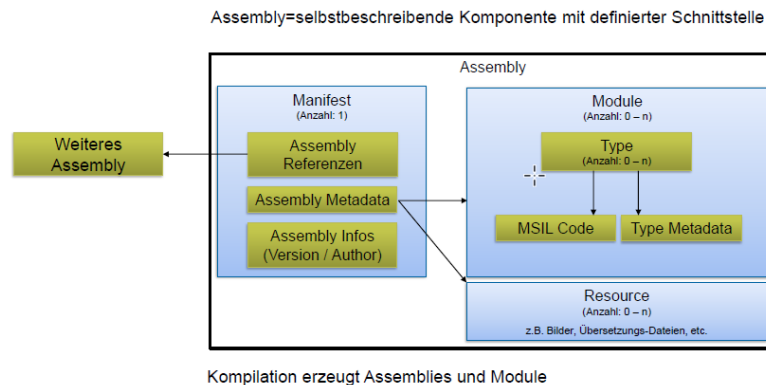


Abbildung 5: Assembly Übersicht

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.

2.6.2. References

Referenzen zeigen auf eine externe Library.

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

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. Visual Studio 15

3.1. Solution

Eine Solution besteht aus mehreren Projekten.

3.2. 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

3.3. 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.

4. C# Grundlagen

4.1. 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){ .. }`)

4.2. 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

4.3. 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

Tabelle 2: Sichtbarkeiten

Typ	Sichtbarkeit	Member (default)	Member (zulässig)
class	public, internal(default)	private	public, protected, internal, private, protected internal
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

4.4. 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 6: Operatoren Präzedenz

4.5. 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
```

4.6. Statements

4.6.1. If Else If Else

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

4.6.2. Switch Case

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

4.6.3. Loops

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

4.6.4. Kommentare

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

4.7. 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 7: Primitive Typen

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

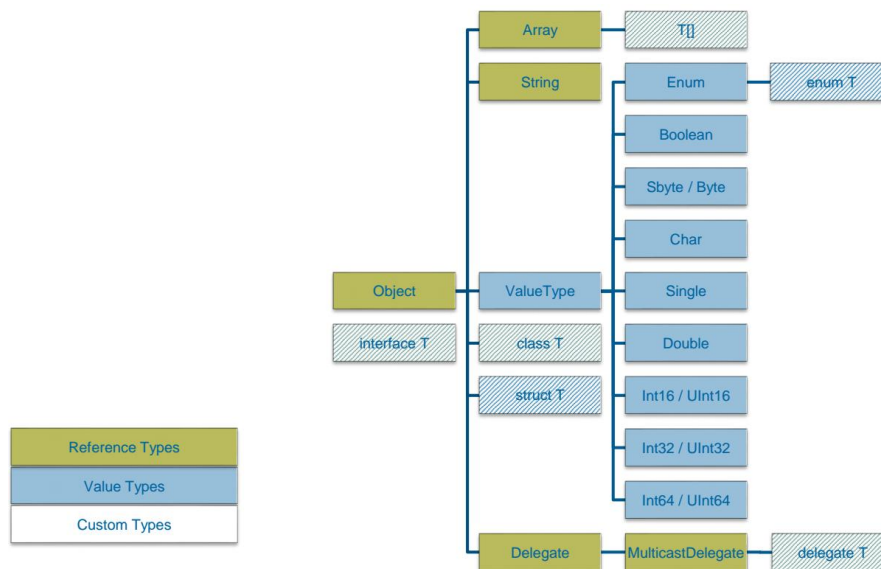
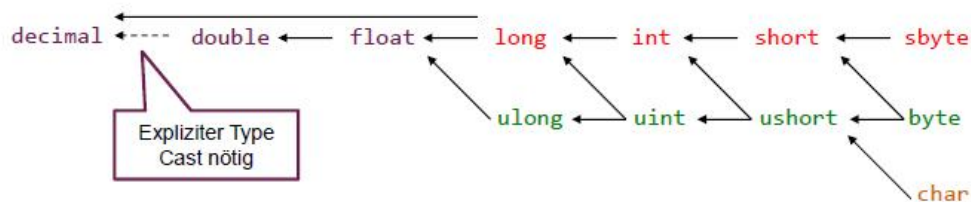


Abbildung 8: Datentypen

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 9: Default Values

4.7.1. Casts



■ Erlaubt

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

■ Nicht erlaubt

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

Abbildung 10: Casts

4.7.2. 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**)

4.7.3. 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
```

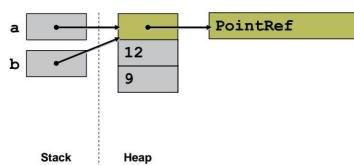


Abbildung 11: Referenztypen

```
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
```

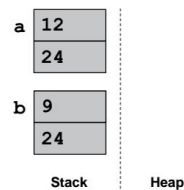


Abbildung 12: Werttypen

4.8. 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

```
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 // 10
16 d ?? 1000 // 1000
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
```

4.9. 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;

```

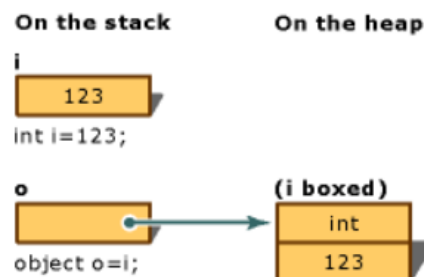


Abbildung 13: Boxing

4.10. Object

- **object** ist ein Alias für System.Object

4.11. String

- **string** ist ein Alias für System.String
- String ist ein Reference Type
- Wie in Java ist ein String nicht modifizierbar
- 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
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

```

4.12. Arrays

```

1  int[] array1 = new int[5]; // deklaration value type
2  int[] array2 = {1,2,3,4,5}; // deklaration & wertedefinition
3  int[] array3 = int[] {1,2,3,4,5,6}; // vereinfachte syntax ohne typ
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 (Speichereffizienter und schneller im Zugriff)
9  int[,] multiDim1 = new int[2,3];
10 int[,] multiDim2 = { {1,2,3} , {4,5,6} };
11
12 // Jagged Arrays
13 int[][] jaggedArray = new int[6][];
14 jaggedArray[0] = new int[4] { 1, 2, 3, 4 }

```

4.13. 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]

```

4.14. 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);

```

4.15. 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)

```

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

```

5. Variablen und Properties

5.1. Konstanten

Der Wert einer Konstante muss zur Compilezeit verfügbar sein.

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

5.2. 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 }
```

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

5.3.1. 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;
```

5.3.2. Properties direkt initialisieren

Properties können bei der Objekt erstellung direkt initialisiert werden.

```
1 MyClass mc = new MyClass() {  
2     Length = 1;  
3     Width = 2;  
4 }
```

6. Methoden

6.1. Overloading

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

```

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);

```

6.2. Call by value

Es wird eine Kopie des Stack Inhalts übergeben

```

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

```

6.3. Call by reference

Adresse der Variable wird übergeben. Mit dem **ref** Keyword können auch Werttypen als Referenz übergeben werden.

```

1 void IncRef(ref int x) {}
2 int value=3;
3 IncRef(ref value); // pass reference, value must be initialized (default param value
    possible too)

```

6.4. 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. Das **out** Keyword muss beim Aufrufer und bei der Methode deklariert werden.

```

1 static void Init(out int a) {
2     a = 10;
3 }
4
5 // usage
6 int value;
7 Init(out value);
8 // value is now 10

```

6.5. 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);

```

6.6. 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 ){ .. }
```

6.7. Named Parameter

Optionale Parameter können über den Namen identifiziert und übergeben werden.

```

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

6.8. 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)

6.9. 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
```

6.10. 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
```

6.11. Sealed

Mit **sealed** weiss der Compiler, dass die Methode (kein Overriding) oder Klasse (keine Vererbung) nicht mehr verändert wird. Es ist das Pendant zum Java **final**.

```
1 public sealed void MyFunc()
```

7. Klassen, Structs

7.1. 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

```

1 public class MyClass : Base
2
3 // check if class is instance of
4 Sub s = new Sub();
5 if (s is Base) {}

```

7.1.1. 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
7
8 // check if class is instance of
9 Sub s = new Sub();
10 if (s is Base) {}

```

7.1.2. 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 }

```

7.1.3. 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();

```

7.2. 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 }

```

7.3. 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 }

```

7.4. 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 }

```

7.5. 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
 - Ist “immutable” (nicht der default)
 - Wird nicht häufig geboxt
 - Ist entweder kurzlebig oder wird meist in andere Objekte eingebettet

```
1 public struct MyStruct {}  
2  
3 ComplexNumber i = new ComplexNumber(2, 3);
```

7.6. Konstruktoren

Man unterscheidet zwischen statischen und nicht-statischen Konstruktoren.

static Ist nicht von aussen verfügbar und wird für Initialisierungsarbeiten verwendet werden. Er wird nur für die erste Instanz aufgerufen

nicht statisch Der normale Konstruktor

```
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 }
```

7.7. 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)

```

Sub s1 = new Sub();
// Sub > subStaticValue
// Sub > Statischer Konstruktor
// Sub > subValue
// Base > baseStaticValue
// Base > Statischer Konstruktor
// Base > baseValue
// Base > Konstruktor
// Sub > Konstruktor
Sub s2 = new Sub();
// Sub > subValue
// Base > baseValue
// Base > Konstruktor
// Sub > Konstruktor

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() { }
}

```

Abbildung 14: Initialisierungs-Reihenfolge (mit Vererbung)

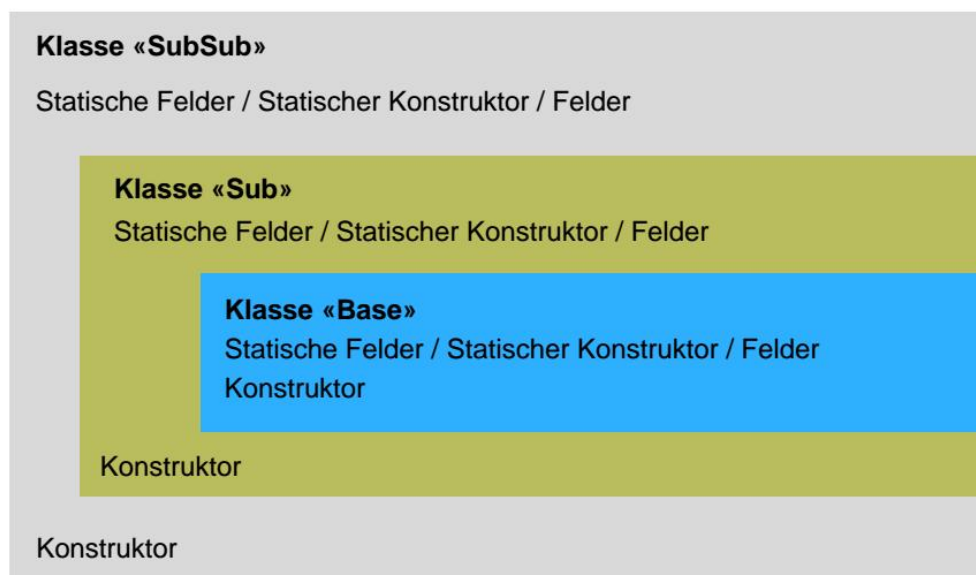


Abbildung 15: Initialisierungsreihenfolge

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

Abbildung 16: Konstruktoraufrufe

7.8. Destruktoren

Der Destruktor ruft im Hintergrund die Methode Finalize auf.

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

7.9. 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 17: Operatoren Überladen

8. Interfaces

- Name beginnt mit grossem I
- Member sind implizit `abstract virtual`
- Member dürfen nicht `static` oder ausprogrammiert sein
- `override` ist nicht nötig

```
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. 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);
9
10 // Print all Enum Types
11 foreach(string name in Enum.GetNames(typeof(Days))) {
12     Console.WriteLine(name);
13 }
```

10. Generics

Generics können in Klassen, Structs und Delegates verwendet werden. Generics sind für Value Types schneller, 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 }
```

10.1. 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>
```

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

Tabelle 4: Type Constraints

```
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 }
```

10.2. Typprüfungen

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

11. Delegates

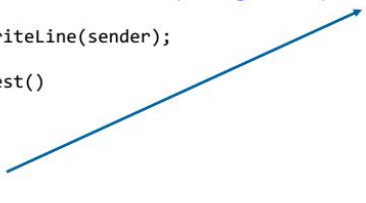
Ein Delegate ist ein eigener **Referenztyp** und wird daher grundsätzlich ausserhalb von Klassen definiert. Jedes Delegate erbt von der Klasse `MulticastDelegate`. Er bietet eine Vereinfachung von Interfaces. Delegates können als Ersatz für das Factory und Template Method Pattern verwendet werden.

- Zuweisung: `DelegateType myDelegate = obj.Method;`
- Aufruf: `object result = myDelegate(params);`

```

1 public delegate int Calculator(int a, int b) {
2     public class Test {
3         Calculator add;
4
5         public Test {
6             // anonymous delegate
7             add = delegate(int a, int b) {
8                 return a+b;
9             }
10        }
11    }
12 }
13
14 int res = add(3,4); // 7

```



```

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 18: Übersicht Delegates und Lamdas

11.1. 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 Methode werden dann nacheinander aufgerufen. (Intern Linked List). Lamdas werden vom Compiler als Delegate abgebildet.

```

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)

```

11.2. Anonyme Methoden

Anonyme Methoden sind immer in-place

```

1 list.ForEach(delegate(int i){
2     Console.Write(i);
3 });

```

11.3. 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 }

```

11.4. EventHandler

Anstelle eines eigenen EventHandler kann man auch den bestehenden `EventHandler<EventArgs>` nutzen. Der erste Parameter ist dann immer das `this` Objekt!

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 }
```

12. Lamdas

- Lamdas können mehrere 0 oder mehrere Parameter haben
- Der Typ der Parameter darf weggelassen werden
- Man unterscheidet zwischen **Expression Lamdas** (Mit geschweiften Klammern) und **Statement Lamdas** (einzelner Ausdruck, kein return nötig). Ein Lamda 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");

```

12.1. Closure

Der Zugriff auf lokale Variablen aus dem Lamda 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 lamda 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

```

13. Iteratoren

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

13.1. Foreach Loop

Der Foreach Loop ist in C# gleich wie in Java mit dem Unterschied, dass anstatt einem Doppelpunkt das Keyword **in** verwendet wird. Die Collection, über welche geloopt wird, muss `IEnumerable` resp. `IEnumerable<T>` implementieren.

```
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 }
```

13.2. Iterator Interface

```
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 }
```

13.3. Iterator Methoden und Yield Return

- Eine Iterator Methode muss die Signatur `public IEnumerator<int> GetEnumerator()` haben
- `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) { .. }

```

13.4. 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

```
1 public static class MyExtensions {
2     public static IEnumerable<T> HsrWhere<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> HsrOfType<T> (this IEnumerable source) {
11        foreach (object item in source) {
12            if (item is T ) {
13                yield return (T)item;
14            }
15        }
16    }
17 }
```

14. Exceptions

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");

```

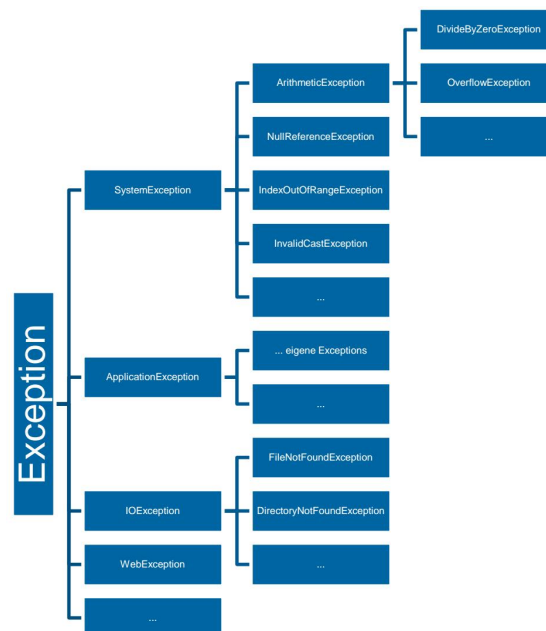


Abbildung 19: Exception Klassen

14.1. Exception Filter

```

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

```

15. 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 ?? Code) Auch LINQ ist reines Compiler Feature.

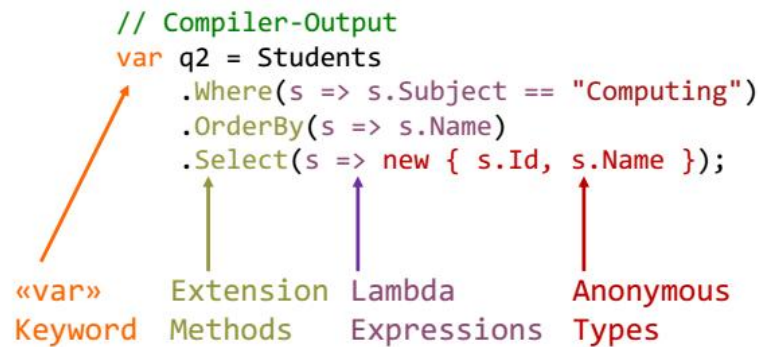


Abbildung 20: LINQ Komponenten

```

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 / Lambda 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 / Lambda 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);

```

15.1. Extensions Syntax (Fluent Syntax)

15.1.1. 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     }

```

15.1.2. 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);
```

15.2. 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

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 21: Query Operatoren

15.2.1. Gruppierung

```
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 };
```

15.2.2. Inner Joins

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

```
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;
```

15.2.3. Group Joins

Ein Group Join verwendet die into Expression.

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

15.2.4. Left Outer Joins

```
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     }
```

15.2.5. 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 }
```

15.2.6. 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;
```

15.2.7. 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     };
```

16. Direct Initialization

16.1. 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 });

```

16.2. 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 };

```

16.3. 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.

17. Entity Framework

Unter .NET kommt ADO.NET als Entity Framework zum Einsatz. Man unterscheidet zwischen zwei Varianten wie die Entitätsklassen/Datenbanken erstellt werden können. Das Entity Framework muss mit NuGet installiert werden.

Designer Centric 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 Centric Man erstellt die Model Klassen und lässt die Datenbank automatisch generieren.

```

1  // models
2  public class Blog
3  {
4      public int BlogId { get; set; }
5      public string Name { get; set; }
6      public string Url { get; set; }
7      // Navigation Property
8      public virtual ICollection<Post> Posts { get; set; } = new HashSet<Post>();
9  }
10
11  // Datenkonsistenz wird von der Businessklasse selbstaendig sichergestellt
12  public class Post {
13      public Blog blog {get; set; }
14  }

```

```

1  // define DB context
2  public class BlogDB : DbContext
3  {
4      public BlogDB() : base("name=ErstesBeispiel"){
5
6          // CreateDatabaseIfNotExists (default)
7          // DropCreateDatabaseIfModelChanges
8          // DropCreateDatabaseAlways
9          // MigrateDatabaseToLatestVersion
10         Database.SetInitializer<BlogDB>(new DropCreateDatabaseAlways<DbContext>());
11     }
12
13     public virtual DbSet<Blog> Blogs { get; set; }
14     public virtual DbSet<Post> Posts { get; set; }
15 }

```

```

1  // use
2  using (var db = new BlogDB()) {
3      //create blog and posts
4      var newBlog = new Blog { Name = "FirstBlog" };
5      db.Blogs.Add(newBlog);
6      newBlog.Posts.Add(new Post() { Title = "Post1" });
7      newBlog.Posts.Add(new Post { Title = "Post2" });
8
9      //save all changes to the database
10     db.SaveChanges();
11
12     // Display all Blogs from the database
13     var query = from blog in db.Blogs
14                 orderby blog.Name
15                 select new { blog.Name, NrPosts = blog.Posts.Count };
16

```

```

17 Console.WriteLine("All blogs in the database:");
18 foreach (var item in query) {
19     Console.WriteLine("{0} : {1}", item.Name, item.NrPosts);
20 }
21
22 //update post
23 var p = (from post in db.Posts select post).FirstOrDefault();
24 if (p != null) {
25     p.Title = "Modified";
26 }
27 db.SaveChanges();
28
29 //delete blog
30 var b = (from blog in db.Blogs select blog).FirstOrDefault();
31 if (b != null) {
32     db.Blogs.Remove(b);
33 }
34
35 //Blog is removed from Database, exists as a detached POCO
36 db.SaveChanges();
37 }

```

Listing 7: app.config

```

1 </configuration>
2 ...
3 <connectionStrings>
4     <add name="ErstesBeispiel" connectionString="data source=(localdb)\mssqllocaldb;
5         initial catalog=ErstesBeispiel;integrated
6         security=True;MultipleActiveResultSets=True;App=EntityFramework"
7         providerName="System.Data.SqlClient" />
8 </connectionStrings>
9 </configuration>

```

17.1. OR Mapping

Ein OR-Mapper stellt die Verbindung zwischen Datenbank und Klassen 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.

Entity Ein Objekt mit einem Key (z.B ID)

Association Definiert eine Assotiation zwischen Entitäten (z.B Navigation Properties)

17.1.1. Vererbung Mappings

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

- **Table per Hierarchy:** Der Standard. Alles in eine Tabelle
- **Table per concrete Type:** Tabelle für Parent und Tabelle für Childs inkl. Felder des Parents
- **Table per Concrete Type:** Tabelle für Parent und Tabellen für Childs mit Verweis auf Parent. Child Tabelle enthält nur die ergänzenden Felder.

17.2. Database First

Beim Database First Ansatz wird die Datenbank aus einem *.edmx Model heraus generiert .Das Model besteht aus drei Teilen:

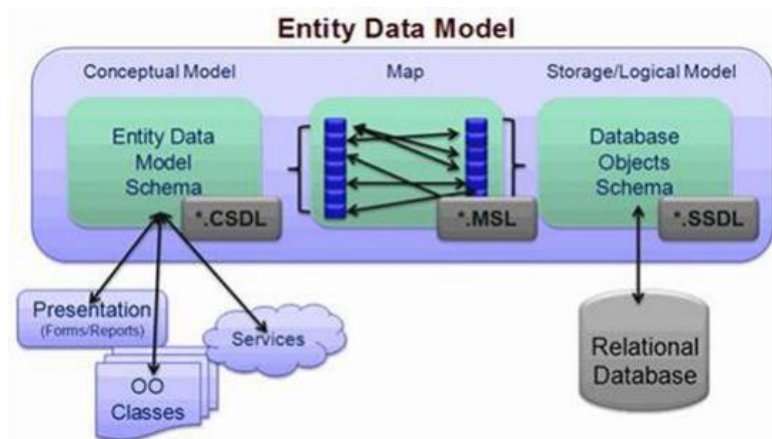


Abbildung 22: Entity Data Model

SSDL: Store Schema Definition Language

Definiert die Struktur des Datenspeichers (z.B. SQL Server Datenbank). Beinhaltet Entities (Tabellen, View), Properties (Spalten), Keys (Primarykeys), Associations (Fremdschlüssel, Beziehungen), Functions (Stored Procedures, Stored Functions).

CSDL: Conceptual Schema Definition Language

Definiert das konzeptuelle Modell und beinhaltet Entitäten, Properties / Fields, Beziehungen und Funktionen.

MSL: Mapping Specification Language

Mapping zwischen den Artefakten in SSDL und CSDL.

17.3. Code First

17.3.1. Attribute

```

1 [Table("orders")]
2 class Order {
3
4     // implizit the Id oder [ClassName]Id is primary key
5     [Key]
6     [Column("id")]
7     public Guid Id { get; set; }
8
9     [Column("customer")]
10    public Guid CustomerId { get; set; }
11
12    [Column("productName"), Required, StringLength(50)]
13    public string Name { get; set; }
14
15    // implizit the Id oder [ClassName]Id is foreign key

```

```

16     [ForeignKey("CustomerId"), InverseProperty("Orders")]
17     public Customer Customer { get; set; }
18
19     // Deklariert das andere Ende der Relation
20     [InverseProperty("Products")]
21     public decimal Price { get; set; }
22
23     [NotMapped]
24     public decimal PriceTotal {
25         get { return this.Items.Sum(x => x.Subtotal); }
26     }
27 }

```

17.4. Model Builder

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

```

1 protected override void OnModelCreating(DbModelBuilder modelBuilder) {
2     base.OnModelCreating(modelBuilder);
3
4     var product = modelBuilder.Entity<Product>();
5     product.ToTable("products");
6     product.HasKey(x => x.Id);
7     product.Property(x => x.Id).HasColumnName("id");
8 }

```

17.5. Lazy-, Eager-Loading

Es wird standardmässig Lazy Loading verwendet.

Lazy Loading Daten werden erst geladen, wenn sie dereferenziert werden. z.B erst wenn effektiv auf die Membervariable (Liste aus mehreren Items) zugegriffen wird. Für das implizite Lazy Loading müssen die Methoden **virtual** definiert werden.

Eager Loading Das komplette Objekt wird geladen.

```

1 // lazy loading
2 using (var context = new NorthwindEntities()) {
3     context.Configuration.LazyLoadingEnabled = true; //default = true
4     var query = from c in context.Customers
5                 where c.CustomerID == "ALFKI"
6                 select c;
7     var cust = query.FirstOrDefault(); //1. load customer
8
9     if (cust == null) return;
10    Console.WriteLine("Customer Id {0}", cust.CustomerID);
11    foreach (var order in cust.Orders) { //2. load orders
12        Console.WriteLine(" --- Order Id {0}", order.OrderID);
13    }
14 }
15
16 // eager loading
17 using (var context = new NorthwindEntities()) {
18     var query = from c in context.Customers
19                 where c.CustomerID == "ALFKI"
20                 select c;

```

```
21     var cust = query.Single(); //1. Load customer
22
23     //2. Explicitly load Orders
24     context.Entry(cust)
25         .Collection(c => c.Orders)
26         .Load();
27     foreach (var order in cust.Orders) { . . . }
28 }
```

17.6. DB Context

Der DB Context ist für die Persistierung von Objekten zuständig. Es stellt change-Tracking und Caching zur Verfügung und übernimmt das Handling von Entity Keys. Im DB Context werden alle CRUD Operationen definiert. Über ihn ist die Methode `saveChanges()` verfügbar.

17.7. Entity Key (Object Identity)

Der Entity Key ist der Primärschlüssel/Fremdschlüssel in der objektorientierten Welt. Er ist Initial auf default(T) (meist 0) gesetzt und wird beim ersten Speichern des Objekts in der Datenbank mit dem DB PK überschrieben. Wenn ein Objekt im gleichen Context geladen wird, gibt der Context das gecachte Objekt zurück. Die Objekte werden anhand ihrem Entity Key gecached.

17.8. Optimistic Concurrency

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

- Version Number pro Record (Timestamp): Beim Laden wird die Versionsnummer als Sessionzustand vermerkt.

```
1 using (var context = new ExtendedNorthwindEntities()) {
2     var cat = context.Categories.First(e => e.CategoryName == "X");
3     cat.Description = "BlaBla";
4     try {
5         int affected = context.SaveChanges();
6     } catch (DbUpdateConcurrencyException) {
7         var entry = context.Entry(cat);
8         entry.Reload();
9
10        // And saving the changes again
11        context.SaveChanges();
12    }
13 }
```

18. WCF: Windows Communication Foundation

WCF wird verwendet um zwischen Prozessen zu kommunizieren die auch nicht auf dem selben Gerät laufen müssen. Dabei geht es besonders um die Kommunikation zwischen Applikationen, da bei Serveranfragen heutzutage eher auf REST gesetzt wird. WCF ist serviceorientiert und kommt direkt mit dem Visual Studio. Es verwendet standardmässig SOAP, wobei auch andere Protokolle verwendet werden können.

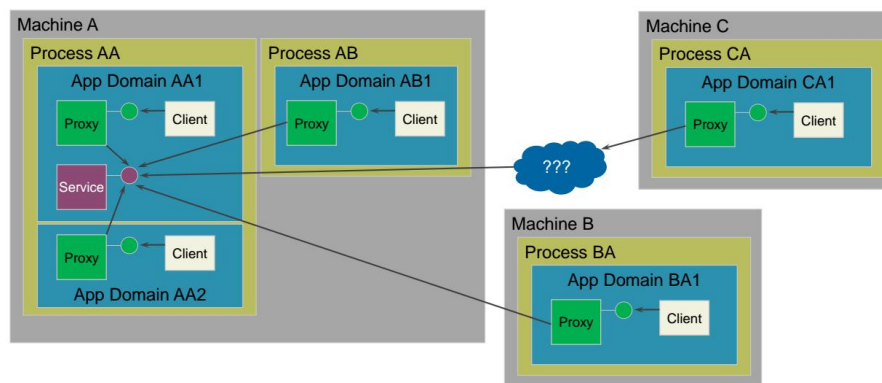


Abbildung 23: WCF Architektur

18.1. Gemeinsame Assembly

Der Einsatz von gemeinsamen Assemblies, erlaubt eine direkte Programmierung gegen ein Interface, was die Entwicklung vereinfacht. Im Gegensatz zu Services, muss die Service Referenz nicht bei jeder Änderung neu generiert werden. Dafür kann ein gemeinsames Assembly nur von .NET Sprachen genutzt werden, ein Service aber auch von anderen Programmiersprachen. Ein weiterer Nachteil ist das verteilen von gemeinsamen Assemblies auf mehreren Geräten.

18.2. Kommunikation

- Es gibt immer einen Client (`ClientBase<T>`) und einen Service (`ServiceHost`)
- Ein Service kann mehrere Endpoints anbieten (Ports). Auch der Client hat einen Endpoint. Zwischen den beiden Endpoints werden dann Messages ausgetauscht.
- Die Typinformation kann beim Client verloren gehen, da aus kompatibilitätsgründen die generische Varianten von z.B Collections verwendet wird. Ebenfalls werden aus den Properties Methoden generiert, da andere Programmiersprachen, dieses Konstrukt nicht kennen
- Es sollte nie die automatisch generierten Klassen des EntityFramework über den Service angeboten werden, da sich bei jeder Generierung die Schnittstelle ändern würde. Die Schnittstelle sollte als eigenes Objekt modelliert werden. Somit hat man die volle Kontrolle welche Felder nach aussen gereicht werden.

18.3. Endpoint

Hinweis 18.1: Prüfung

12_WCF: Slide 10,11: Mögliche Frage. Warum konnte keine Verbindung aufgebaut werden. Da Binding Definitionen nicht matchen.

Ein Service Endpoint wird benötigt um den Service zugreifbar zu machen. Jeder Endpoint besteht aus drei Elementen

A: Address URL / URI (Where)

B: Binding Definition Übertragungskanal (How). Stimmen die Service Eigenschaften nicht mit den Bindings überein, kann keine Verbindung aufgebaut werden.

- Kommunikationsmuster (Synchron, Asynchron, Duplex)
- Encoding (Text, Binary)
- Transport Methode (HTTP, TCP/UDP, Pipes)
- Security
- Reliability (Ordered, Unordered, E2E)
- Transaction Management
- Client und Server müssen zwingend das gleiche Binding verwenden. Die Binding werden über MEX publiziert.

C: Contract Interface Definition, Daten Transfer Objekte / Messages (What)

18.4. MEX: Meta Data Exchange Endpoint

Ein Service kann ein MEX Endpoint zur Verfügung stellen, der die Spezifikation der Schnittstelle zurückliefert. (**WSDL**: Web Service Description Language). Visual Studio kann dann gemäss dem WSDL den Client Code generieren. Wenn der Service ändert, muss natürlich auch der Client Code erneut generiert werden.

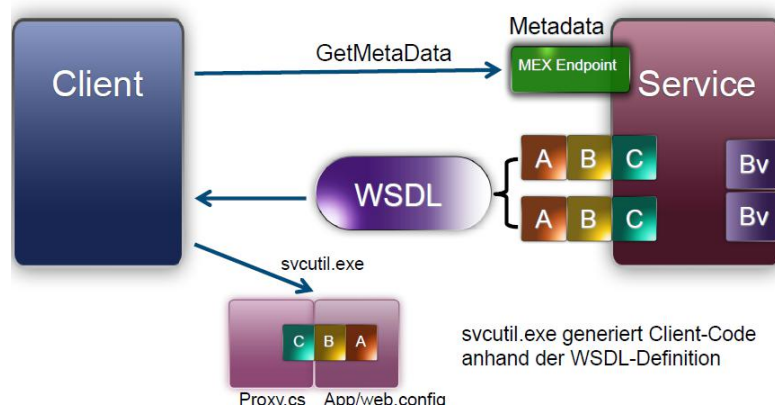


Abbildung 24: Metadata Exchange Point

18.5. Contracts

Hinweis 18.2: Prüfung

Kommt ganz bestimmt an der Prüfung. Setzen der korrekten Attribute für die Service Definition

Es gibt drei Arten von Contracts, die die Kommunikation zwischen Client und Server definieren. Die Contracts sollten immer explizit definiert werden, da standardmässig alle Felder im Object Graph serialisiert werden. (auch Passwörter, etc.)

Service Contract Definiert Operationen, Verhalten und das Interaktionsmodell. Was macht der Service?

Data Contract Definiert übertragenen Datenstrukturen und Versionierung. Welche Objekte verwendet der Service? Definiert das Serialisierungsformat.

Message Contract Definiert die Messagestruktur (Header, Body). Wie sehen die SOAP Nachrichten aus?

Listing 8: Service Contract

```

1 [ServiceContract]
2 public interface ITimeService {
3     [OperationContract]
4     string GetTime();
5
6     // custom name
7     [OperationContract(name="GetTimeDescr")]
8     TimeDescData GetTimeDesc();
9
10    // async
11    [OperationContract(IsOneWay=true)]
12    void StoreTime();
13 }
14
15 public class TimeService : ITimeService {
16     public string GetTime() {
17         return DateTime.Now.ToString();
18     }
19     public TimeDescData GetTimeDesc() {
20         return new TimeDescData();
21     }
22 }
```

Listing 9: Data Contract

```

1 [DataContract]
2 public class TimeDescData {
3     public TimeDescData() {
4         TimeLong =
5         DateTime.Now.ToLongDateString();
6         TimeShort =
7         DateTime.Now.ToShortDateString();
8     }
9
10    [DataMember]
11    public string TimeShort { get; set; }
```

```

12
13     [DataMember]
14     public string TimeLong { get; set; }
15 }
16
17
18 [DataContract]
19 public enum TestType {
20     [EnumMember]
21     A,
22     [EnumMember]
23     B
24 }

```

18.6. Vererbung

Bei vererbten DataContracts muss mit `KnownType()` explizit deklariert werden, welche Subklassen ebenfalls serialisiert werden.

Listing 10: Data Contract Vererbung

```

1 [ServiceContract]
2 public interface IClassroomService {
3     [OperationContract]
4     List<Student> GetStudents();
5 }
6
7 [KnownType(typeof(TiredStudent))]
8 [KnownType(typeof(BoredStudent))]
9 [DataContract]
10 public class Student { }
11
12 [DataContract]
13 public class TiredStudent : Student { }
14
15 [DataContract]
16 public class BoredStudent : Student { }

```

Wenn der Known Type hinter einem Interface versteckt ist, muss dieses explizit mit dem Keyword `ServiceKnownType` angegeben werden.

Listing 11: Service Known Data Contract Vererbung

```

1 [ServiceContract]
2 public interface IClassroomService {
3     [ServiceKnownType(typeof(Teacher))]
4     [OperationContract]
5     List<ITeacher> GetTeachers();
6 }
7
8 [DataContract]
9 public class Teacher : ITeacher { }
10 public interface ITeacher { }

```

18.7. Serialisierung von Referenzen

Standardmässig werden alle public Variablen separat serialisiert. Man kann mit dem Keyword `IsReference = true` aber erzwingen, dass **Referenzierte Objekte** sich auch nach der Serialisierung referenzieren.

Listing 12: Serialisierung von Referenzen

```

1 [DataContract]
2 public class MainDto {
3     public MainDto() {
4         Content1 = new ContentDto();
5         Content2 = Content1; // Reference
6     }
7     [DataMember]
8     public ContentDto Content1;
9     [DataMember]
10    public ContentDto Content2;
11 }
12
13 [DataContract(IsReference = true)]
14 public class ContentDto {
15     ..
16 }
```

18.8. Faults

Da Exceptions nicht serialisierbar sind, müssen als **FaultContract** definiert werden. Damit können Exceptions an den Client übermittelt werden. Man macht dabei gebrauch von der Klasse `FaultException` rsp. `FaultException<>`. Damit FaultContracts verwendet werden können, darf der `OperationContract` **nicht asynchron** definiert sein (`[OperationContract(IsOneWay=true)]`). Ansonsten käme die Exception nie an.

Listing 13: Data Contract

```

1 // Service Definition
2 [ServiceContract]
3 public interface ICalculator {
4     [OperationContract]
5     int Multiply(int n1, int n2);
6
7     [OperationContract]
8     [FaultContract(typeof(MathFault))]
9     int Divide(int n1, int n2);
10 }
11
12 // Server
13 try {
14     return n1 / n2;
15 } catch (DivideByZeroException) {
16     MathFault mf = new MathFault {
17         Operation = "division",
18         ProblemType = "divide by zero"
19     };
20     throw new FaultException<MathFault>(mf);
21 }
22
23 // Client
24 catch (FaultException<MathFault> e) { .. }
```

18.9. Service Hosting

Der WCF Service kann entweder in der `App.config` deklariert oder direkt im Code erstellt werden

Listing 14: App.config

```

1 <services>
2   <service behaviorConfiguration="TimeService.TimeServiceBehavior"
3     name="TimeService.TimeService">
4     <endpoint address="" binding="basicHttpBinding"
5       contract="TimeService.ITimeService"/>
6     <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>
7     <host>
8       <baseAddresses>
9         <add baseAddress="http://localhost:8732/TimeService"/>
10      </baseAddresses>
11    </host>
12  </service>
13 </services>
14
15 <!-- Definition des behaviorConfiguration des Services -->
16 <behaviors>
17   <serviceBehaviors>
18     <behavior name="TimeService.TimeServiceBehavior">
19       <serviceMetadata httpGetEnabled="True"/>
20       <serviceDebug includeExceptionDetailInFaults="False"/>
21     </behavior>
22   </serviceBehaviors>
23 </behaviors>
24
25 // usage (immer die Klasse, nie das Interface) -> korreliert mit service name im
26 // App.config
27 ServiceHost myHost = new ServiceHost(typeof(TimeService.TimeService))

```

Listing 15: Service Hosting (Code)

```

1 Uri myAddress = new Uri("http://localhost:8732/TimeService");
2 BasicHttpBinding myBinding = new BasicHttpBinding();
3 using (ServiceHost myHost = new ServiceHost(typeof(TimeService.TimeService),
4   myAddress))
5 {
6   myHost.AddServiceEndpoint(typeof(TimeService.ITimeService), myBinding, myAddress);
7   myHost.Open();
8   Console.WriteLine("Service ready. Hit <RETURN> to quit.");
9   Console.ReadLine();
10 }

```

18.10. Asynchron und Synchrone Kommunikation

Listing 16: Data Contract

```
1 // default synchron
2 public interface ICalculator {
3     [OperationContract]
4     int Add(int n1, int n2);
5 }
6
7 // One way async call
8 [ServiceContract]
9 public interface ICalculator {
10     [OperationContract(IsOneWay = true)]
11     void StoreProblem(ComplexProblem p);
12 }
13
14 // Duplex with async callbacks
15 [ServiceContract(SessionMode = SessionMode.Required, CallbackContract =
16     typeof(ICalcCallback))]
17 public interface ICalculatorDuplex {
18     [OperationContract(IsOneWay = true)]
19     void AddTo(double n);
20 }
21
22 public class CalculatorService : ICalculatorDuplex {
23     public void AddTo(double n){
24         result += n;
25         Callback.Result(result);
26     }
27     ICalcCallback Callback {
28         get {
29             return OperationContext.Current.GetCallbackChannel<ICalcCallback>();
30         }
31     }
32 }
```

18.11. Instanzmodelle

Es gibt drei verschiedenen Instanzmodelle. Das Instanzmodell wird bei der Service **IMPLEMENTATION** angegeben.

1. **Single**: Ein Service Objekt für sämtliche Clients (Singleton). Der Status ist gebunden an die Service Instanz und deren Laufzeit.
2. **Per Call**: Ein neues Service Objekt pro Call / vollkommen stateless
3. **Per Session**: Ein neues Service Objekt pro Session / Status gebunden an Client Session und deren Laufzeit. Der Session Mode wird auf dem INTERFACE angegeben. (Workflow getriebene Anwendungen)

Listing 17: Instanzmodelle

```

1  [ServiceBehavior(
2      InstanceContextMode
3      = InstanceContextMode.Single
4      InstanceContextMode.PerCall
5      InstanceContextMode.PerSession)]
6  public class CounterService : ICounterService {
7      public void InitializeCounter(int value) { /* ... */ }
8      public void Increment() { /* ... */ }
9      public void EndCounting() { /* ... */ }
10 }
11
12 // Per Session Spezialfall
13 [ServiceContract(
14     SessionMode = SessionMode.Allowed
15     SessionMode.NotAllowed
16     SessionMode.Required)]
17 public interface ICounterService {
18
19     // Default: IsInitiating = true
20     [OperationContract]
21     void InitializeCounter(int value);
22
23     [OperationContract(IsInitiating = false)]
24     void Increment();
25
26     [OperationContract(
27         IsInitiating = false,
28         IsTerminating = true)]
29     void EndCounting();
30 }

```

18.12. Reliability

Listing 18: WCF Reliability

```

1 [DeliveryRequirements(
2   RequireOrderedDelivery = true,
3   QueuedDeliveryRequirements
4     = QueuedDeliveryRequirementsMode.Allowed
5       QueuedDeliveryRequirementsMode.NotAllowed
6       QueuedDeliveryRequirementsMode.Required)]
7 public class CounterService : ICounterService {
8   public void InitializeCounter(int value) { /* ... */ }
9   public void Increment() { /* ... */ }
10  public void EndCounting() { /* ... */ }
11 }

```

18.13. Concurrency

Die Concurrency wird auf der Service Implementierung angegeben.

Listing 19: WCF Concurrency

```

1 [ServiceBehavior(
2   ConcurrencyMode
3     = ConcurrencyMode.Single
4       ConcurrencyMode.Multiple
5       ConcurrencyMode.Reentrant
6   public class CounterService : ICounterService {
7     public void InitializeCounter(int value) { /* ... */ }
8     public void Increment() { /* ... */ }
9     public void EndCounting() { /* ... */ }
10  }

```

18.14. Transaktionen

```

1 // interface
2 [ServiceContract]
3 interface IBankTransaction {
4   [TransactionFlow(
5     TransactionFlowOption.NotAllowed
6     TransactionFlowOption.Allowed
7     TransactionFlowOption.Mandatory)]
8   [OperationContract]
9   void Transfer(Account from, Account to, decimal amount);
10
11   [TransactionFlow(TransactionFlowOption.Mandatory)]
12   [OperationContract]
13   void TransferTxSafe(Acoount from, Account to, decimal amount);
14 }
15
16 // service implementierung
17 [ServiceBehavior(TransactionAutoCompleteOnSessionClose = ture,
18   ReleaseServiceInstanceOnTransactionComplete = true)]
19 public class BankTransaction : IBankTransaction {
20   [OperationBehavior(
21     TransactionScopeRequired=false,
22     TrnasactionAutoCompelte = true)]
23   public void Transfer(Account from, Account to, decimal amout) {

```

```

24     ..
25 }
26
27 [OperationBehavior(
28     TransactionScopeRequired=true),
29     TransactionAutoComplete=false]
30 public void TransferTxSafe(Account ffrom, Account to, decimal amount) {
31     ..
32     OperationContext.Current.SetTransactionComplete();
33 }
34 }
35
36 //client code
37 using (TransactionScope ts = new TransactionScope()) {
38     client.TransferTxSafe(acct1, acct2, 10);
39     client.TransferTxSafe(acct1, acct3, 20);
40
41     ts.Complete();
42 }

```

```

1 <bindings>
2   <wsHttpBinding>
3     <binding name="SampleBinding" transactionFlow="true" />
4   </wsHttpBinding>
5 </bindings>

```

18.15. Throttling

Listing 20: Service Throttling

```

1 <service
2   type="Calculator"
3   behaviorConfiguration="CalculatorBehavior">
4   <!-- endpoint definitions /-->
5 </service>
6 <behaviors>
7   <behavior configurationName="CalculatorBehavior" >
8     <serviceThrottling
9       maxConcurrentCalls="10"
10      maxConnections="10"
11      maxInstances="10"
12      maxPendingOperations="10" />
13     <serviceMetadata
14       httpGetEnabled="false" />
15     <serviceDebug
16       includeExceptionDetailInFaults="true" />
17   </behavior>
18 </behaviors>

```

19. Reflection

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

Grosser **Nachteil** von der Verwendung von Reflection ist, dass der Compiler nicht alle Type-Checks machen kann. Beispielsweise wird per Reflection eine Library geladen so wird erst bei Laufzeit ein Fehler geworfen sollte auf eine nicht existierende Methode oder Property zugegriffen werden. Weiter ist die Performance Einbussen von Reflection nicht zu vernachlässigen. Besonders, wenn die Reflection-Analyse in verschachtelte Schleifen immer wieder aufgerufen wird.

Listing 21: Reflection

```

1 this.GetType() // implemented on object
2 typeof(MyClass) || typeof(int)
3 assbly.GetType("TheNamespace.TheClass") //get known class type

```

19.1. Type Discovery

Suche alle Typen in einem Assembly.

Listing 22: Reflection: Type Discovery

```

1 Assembly a01 = Assembly.Load("mscorlib, PublicKeyToken=b77a5c561934e089,
   Culture=neutral, Version=4.0.0.0");
2 Type[] t01 = a01.GetTypes();
3 foreach (Type type in t01)
4 {
5     Console.WriteLine(type);
6     MemberInfo[] mInfos = type.GetMembers();
7     foreach (var mi in mInfos)
8     {
9         Console.WriteLine(
10             "\t{0}\t{1}",
11             mi.MemberType,
12             mi);
13     }
14 }

```

19.2. Member auslesen

Das Auslesen von Members kann mit `BindingFlags` gefiltert werden.

Listing 23: 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 );

```

19.3. Field Information

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

Listing 24: Reflection: Field Info

```

1 Type type = typeof(Counter);
2
3 Counter c = new Counter(1);
4
5 // All Fields
6 FieldInfo[] fiAll = type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic);
7
8 // Specific Field
9 FieldInfo fi = type.GetField("countValue",
10     BindingFlags.Instance |
11     BindingFlags.NonPublic);
12
13 int val01 = (int) fi.GetValue(c);
14 c.Increment();
15 int val02 = (int) fi.GetValue(c);
16 fi.SetValue(c, -999);

```

19.4. 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 25: 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 pi.SetValue(c, -999);

```

19.5. 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 26: 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);

```

19.6. Constructor Info

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

Listing 27: 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
14         Type[0], null);
15 Counter c02 = (Counter)ci02.Invoke(null);

```

19.7. Attributes

Listing 28: 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));

```

19.8. Instanz von Klasse erstellen

Mit Reflection kann eine Library geladen werden und daraus auch gerade eine Instanz erstellt werden. Folgend ein Beispiel.

```

1 var ass=Assembly.LoadFrom("Tiere.dll");
2 var t = ass.GetType("Tiere.Katze");
3 //Create Instance from Class t using default constructor with one string paramter
4 var k=Activator.CreateInstance(t, new object[] {"Mitzi"});
5 var m = t.GetMethod("MausFangen");
6 //Invoke method MausFangen with no arguments => Invoke expects array with arguments
7 m.Invoke(k, new object[]{});

```

19.9. Praktisches Beispiel für den Einsatz von Reflection

Im Folgendem Beispiel wird mittels Reflection ein OR Mapper erstellt. Dabei werden Attribute vom Model gelesen und ein hier nicht genauer erläutertes CommandObject erstellt.

```

1 public static ObjectCommand ToObjectCommand<T>(this T viewModel,
2     RelatedObjectRequesterAction relatedObjectRequesterAction = null)
3 {
4     var command = new ObjectCommand();
5     Type type = typeof(T);
6     PropertyInfo[] allProperties = type.GetProperties();
7     foreach (var property in allProperties)
8     {
9         if (property.GetValue(viewModel, null) == null) continue;
10        var propertyInfo = viewModel.GetType().GetProperty(property.Name);
11        if (propertyInfo != null)
12        {
13            var updateFieldProperty =
14                (IsNotUpdateFieldAttribute)propertyInfo.GetCustomAttributes(typeof(IsNotUpdateFieldAttribute),
15                    false).FirstOrDefault();
16            if (updateFieldProperty != null && updateFieldProperty.NotUpdateField) continue;
17        }
18        if (property.HasHeatRelatedAttribute() && relatedObjectRequesterAction != null)
19        {
20            command.LinkToExistent.Add(property.GetLinkEntryElementFromViewModel(viewModel));
21            command.LinkToExistent.Last().RelatedObjectId =
22                relatedObjectRequesterAction(
23                    (string)property.GetValue(viewModel, null),
24                    property.GetCustomAttribute<HEATRelatedFieldAttribute>()).Id;
25        }
26        else
27        {
28            command.FieldValues.Add(property.GetFieldValueFromViewModel(viewModel));
29        }
30    }
31    return command;
32 }

```

20. Attributes

Attributes sind das C# Pendant zu den Java Annotations. Bei Attributen geht es um die aspektorientierte 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 29: 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 // Beliebig viele Attribute
12 [DataContract][Serializabel] <=> [DataContract, Serializabel]
13
14 // Parameter
15 [DataContract]
16 [DataContract(Name="MyParam")] // Named Param
17 [Obsolete("Alt!", true)] // Positional Param
18 [Obsolete("Alt!", IsError=true)] // Mixed
```

20.1. Anwendungsfälle

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

20.2. Typen

Man unterscheidet zwei Typen von Attributen

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

20.3. 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.

Listing 30: Custom Attributes

```

1 // declaration
2 [AttributeUsage(
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")]

```

Listing 31: 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 {
21     public string Name { get; set; }
22     public CsvAttribute(string name) {
23         Name = name;
24     }
25 }
26
27 public interface IStringFilter {
28     string Filter(string arg);
29 }
30

```

```
31 public class UppercaseAttribute : Attribute : IStringFilter {  
32     public string Filter(string arg) {  
33         return arg.ToUpper();  
34     }  
35 }  
36  
37 public class LowercaseAttribute : Attribute : IStringFilter {  
38     public string Filter(string arg) {  
39         return arg.ToLower();  
40     }  
41 }
```

A. Listings

1.	Data Transfer Objects (DTO)	11
2.	Service Interface	11
3.	Service Implementation	12
4.	Service Hosting via XML	12
5.	Service Hosting via Code	13
6.	C# Event Handler	46
7.	app.config	61
8.	Service Contract	67
9.	Data Contract	67
10.	Data Contract Vererbung	68
11.	Service Known Data Contract Vererbung	68
12.	Serialisierung von Referenzen	69
13.	Data Contract	69
14.	App.config	70
15.	Service Hosting (Code)	70
16.	Data Contract	71
17.	Instanzmodelle	72
18.	WCF Reliability	73
19.	WCF Concurrency	73
20.	Service Throttling	74
21.	Reflection	75
22.	Reflection: Type Discovery	75
23.	Reflection: Members auslesen	75
24.	Reflection: Field Info	76
25.	Reflection: Property Info	76
26.	Reflection: Method Info	77
27.	Reflection: Constructor Info	77
28.	Reflection: Attributes	77
29.	Attributes	79
30.	Custom Attributes	80
31.	CSV Filter	80

B. Abbildungsverzeichnis

1.	Referenz und Value Typen	6
2.	.NET Framework Architektur	14
3.	CLR: Common Language Runtime Architektur	15
4.	MSIL Kompilierung	15
5.	Assembly Übersicht	16
6.	Operatoren Präzedenz	22
7.	Primitive Typen	24
8.	Datentypen	24
9.	Default Values	25
10.	Casts	25
11.	Referenztypen	26
12.	Werttypen	26
13.	Boxing	28
14.	Initialisierungs-Reihenfolge (mit Vererbung)	38
15.	Initialisierungsreihenfolge	38
16.	Konstruktoraufrufe	39
17.	Operatoren Überladen	39
18.	Übersicht Delegates und Lamdas	43
19.	Exception Klassen	51
20.	LINQ Komponenten	52
21.	Query Operatoren	55
22.	Entity Data Model	62
23.	WCF Architektur	65
24.	Metadata Exchange Point	66

C. Tabellenverzeichnis

1.	Naming Conventions	21
2.	Sichtbarkeiten	22
3.	Standard Sichtbarkeiten von Typen	22
4.	Type Constraints	42