Entity Framework Core 9 – Datenbankzugriff mit .NET

Inhaltliche Vorbereitungen:

1. SQL-Server
2. C# mit ADO.NET und LINQ

Dozent: Dr. Thomas Hager, Berlin,
im Auftrag der Firma GFU Cyrus AG
20.10.2025 – 22.10.2025

Inhaltliche Übersicht:
Objektrelationales Mapping (ORM) mit Entity Framework Core

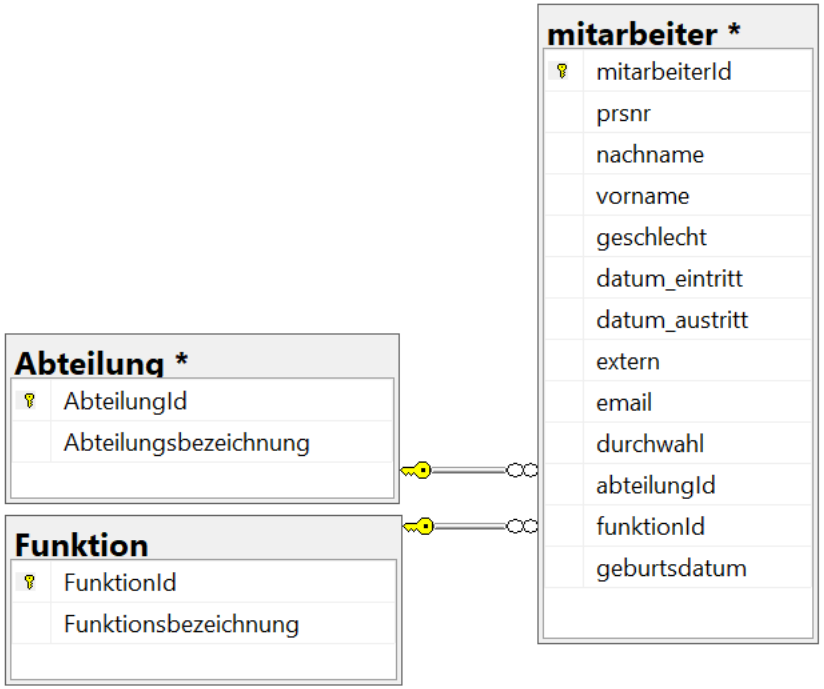
- Wichtige Voraussetzungen und Grundlagen für ORM
  - Das Konzept der Relationalen Datenbanken am Beispiel des SQL Servers
    - Tabellen-Strukturen, Datentypen, referentielle Integrität und andere CONSTRAINTS
    - T-SQL-Erweiterung von SQL: Functions, Stored Procedures, Calculated Columns
  - C#
    - Das objektorientierte Paradigma
    - Die integrierte Abfragesprache LINQ
    - Delegaten, Lambda-Ausdrücke
    - ADO.NET als Vorläufer und Basis von Entity Framework (EF) und Entity Framework Core (EFC)
- **Einführung in das Objektrelationale Mapping mit Entity Framework Core**
  - Der Widerspruch zwischen den beiden Konzepten – Auflösung per ORM
  - Das Grundkonzept des Objekt-Relationalen-Mappings (ORM)
  - Architektur von Entity Framework Core-basierten Anwendungen

Im Folgenden beschäftigen wir uns mit einigen fachlichen Voraussetzungen für EF Core, die zugleich Grundlagen des ORM mit EF Core sind

EINRICHTUNG EINER SQL SERVER TESTDATENBANK ADOTEST MIT DREI TABELLEN, ANSCHLIEßEND EINER C#-CONSOLEN-APP ZUM ZUGRIFF AUF DIE DATEN UND TESTS MIT LINQ

Elemente des Konzepts relationaler Datenbanken (am Beispiel des SQL-Servers)

- Stichworte:
  - Tabelle als Baustein relationaler Datenbanken
  - Tabellenstruktur (Feldnamen, Felddatentypen, Indexe, Constraints).
    Spezielle Regeln für Primärschlüssel-Felder
  - Primär- und Fremdschlüssel, Referentielle Integrität
  - SQL als Abfrage- und Manipulationssprache, Verhältnis von SQL und T-SQL



Beispiel: Abteilung und Funktion sind Nachschlage- bzw. Lookup-Tabellen. Mitarbeiter ist eine Stammdatentabelle Abteilung und Funktion stehen im Verhältnis 1:n zu Mitarbeitern

Skript zur Generierung einer Datenbank und von Datenbank-Objekten

- Wiederherstellen der Datenbank aus dem Backup ADONET.bak
- Oder: Manuelles Anlegen und Nutzen der Datenbank, anschließend Generierung von Tabellen

```
CREATE DATABASE ADOTest;

USE ADOTest;
```

```
CREATE TABLE [dbo].[Abteilung](
[AbteilungId] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[Abteilungsbezeichnung] [nvarchar](20) NOT NULL);

CREATE TABLE [dbo].[Funktion](
[FunktionId] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[Funktionsbezeichnung] [nvarchar](50) NOT NULL);
```

```
CREATE TABLE [dbo].[mitarbeiter](
[mitarbeiterId] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[prsnr] [varchar](4) NULL,
[nachname] [nvarchar](50) NOT NULL,
[vorname] [nvarchar](50) NULL,
[geschlecht] [varchar](1) NULL,
[datum_eintritt] [date] NULL,
[datum_austritt] [date] NULL,
[extern] [bit] NULL,
[email] [varchar](50) NULL,
[durchwahl] [varchar](50) NULL,
[abteilungId] [int] NULL,
[funktionId] [int] NULL,
[geburtsdatum] [date] NULL);
```

Skript zur Generierung einer Datenbank und von Datenbank-Objekten - Ergänzungen

- Festlegung der referentiellen Beziehungen zwischen Abteilung und Mitarbeiter (1:n) sowie Funktion und Mitarbeiter (1:n)

```
ALTER TABLE [dbo].[Mitarbeiter] ADD  CONSTRAINT [FK_Person_Abteilung]
FOREIGN KEY([AbteilungId]) REFERENCES [dbo].[Abteilung] ([AbteilungId]);

ALTER TABLE [dbo].[Mitarbeiter] ADD  CONSTRAINT [FK_Person_Funktion]
FOREIGN KEY([FunktionId]) REFERENCES [dbo].[Funktion] ([FunktionId]);
```

- Weitere Constraints (Beschränkungen) und Default-Werte (Standard-Werte)

```
-- Weitere Constraints
ALTER TABLE [dbo].[Mitarbeiter]  WITH CHECK ADD CHECK  (([Geschlecht]='m' OR [Geschlecht]='w' OR
[Geschlecht]='d' OR [Geschlecht]='x'));

-- Default-Werte
ALTER TABLE [dbo].[Mitarbeiter] ADD  DEFAULT ((1)) FOR [FunktionId];
ALTER TABLE [dbo].[Mitarbeiter] ADD  DEFAULT ((1)) FOR [AbteilungId];
ALTER TABLE [dbo].[Mitarbeiter] ADD  DEFAULT (getdate()) FOR [DatumErstellung];
```

Die Datentypen in den SQL-Server-Tabellen

- Tabellen in relationalen DB werden streng strukturiert. Für jede Spalte wird ein eindeutiger Spaltenname/Feldname und ein eindeutiger Datentyp vergeben:

Tabelle mitarbeiter

Spaltenname	Datentyp	NULL-Werte zulassen
mitarbeiterId	int	☐
prsnr	varchar(4)	☑
nachname	nvarchar(50)	☐
vorname	nvarchar(50)	☑
geschlecht	varchar(1)	☑
datum_eintritt	date	☑
datum_austritt	date	☑
extern	bit	☑
email	varchar(50)	☑
durchwahl	varchar(50)	☑
abteilungId	int	☑
funktionId	int	☑
niederlassungId	int	☑

Tabelle funktion

Spaltenname	Datentyp	NULL-Werte zulassen
funktionId	int	☐
funktionsbezeichnung	nvarchar(45)	☐

Tabelle projektergebnis

Spaltenname	Datentyp	NULL-Werte zulassen
projektergebnisId	int	☐
projektId	int	☐
nettopreis	decimal(10, 2)	☑
mehrwertsteuersatz	float	☑
rechnungstellung	date	☑
zahlungseingang	date	☑

Datentypen im SQL-Server – Numerische Werte

Microsoft SQL Server - Datentyp	Typ	Bemerkung zum Wertebereich
bit		bit-Wertebereich: 0,1, null
tinyint	Ganzzahl	von 0 bis 255 oder von -128 bis 127
smallint	Ganzzahl	von 0 bis 65.535 oder von -32.768 bis 32.767
int	Ganzzahl	0 bis ~4,3 Mill. oder von -2.147.483.648 bis 2.147.483.647
bigint	Ganzzahl	von 0 bis 2^64-1 oder von -(2^63) bis (2^63)-1
real	Gleitkommazahl	Fließkommazahl mit Vorzeichen. Wertebereich von -(1,79769×10308) bis -(2.22507×10-308), 0 und 2.22507×10-308 bis 1,79769×10308
float	Gleitkommazahl	Fließkommazahl mit Vorzeichen. Wertebereich von -(3,402823466×1038) bis -(1,175494351×10-38), 0 und 1,175494351×10-38 bis 3,402823466×1038
decimal, numeric	Dezimalzahl	Numerische Daten mit fester Genauigkeit und Dezimalstellenanzahl von 10^38 1 bis 10^38 1, numeric ist ein Synonym für decimal
money	Dezimalzahl	Währungsdatenwerte zwischen 2^63 (922.337.203.685.477,5808) und 2^63 - 1 (+922.337.203.685.477,5807) mit der Genauigkeit eines Zehntausendstels der Währungseinheit.
smallmoney	Dezimalzahl	Währungsdatenwerte von 214.748,3648 bis +214.748,3647 mit der Genauigkeit eines Zehntausendstels der Währungseinheit

Datentypen im SQL-Server – Zeichenketten, Datum/Zeit, Binär

Datentypen MS SQL (wot.at)

Microsoft SQL Server - Datentyp	Typ	Bemerkung zum Wertebereich
char(n), varchar(n), text	Zeichenfolge	n kann spezifiziert werden (n>=1 bis max). char reserviert Speicher für genau n Zeichen, varchar ermöglicht flexible Speicherung bis zu n Zeichen
nchar(n), nvarchar(n), ntext		n kann spezifiziert werden (n>=1 bis max). nchar reserviert Speicher für genau n Zeichen, nvarchar ermöglicht flexible Speicherung bis zu n Zeichen. Nchar, nvarchar und ntext ermöglichen die Speicherung von unicode-Zeichen.
datetime2(n)	Datum/Zeit	Datums- und Zeitdaten zwischen dem 1. Januar 1753 und dem 31. Dezember 9999 mit einer Genauigkeit von 300stel-Sekunden, also 3,33 Millisekunden. Liefert mit 7 sehr genaue Datum-Zeitangaben. Mit Precision 0 übliches Datum-Zeitformat (YYYY-MM-DD hh:mm:ss)
smalldatetime	Datum/Zeit	Datums- und Zeitdaten zwischen dem 1. Januar 1900 und dem 6. Juni 2079 mit einer Genauigkeit von einer Minute
date	Datum	
time	Zeit	
varbinary(n)	Binär	n kann spezifiziert werden (n>=1 bis max)
Xml	Xml	Dient der Darstellung und Manipulation von XML-Daten
hierarchieId	Binär	Dient der Darstellung und Manipulation hierarchischer Beziehungen
json	nvarchar(max)	Dient der Speicherung und Manipulation komplexer Datenstrukturen im json-Format

Festlegungen von Constraints (Beschränkungen)

- In Tabellen können aus inhaltlichen Gründen einzelne Spalten bestimmten Beschränkungen/Constraints unterliegen

_	**Beschreibung**
NOT NULL	Sichert, dass eine Spalte keine NULL-Werte enthält, erzwingt die Eingabe
UNIQUE	Sichert, dass kein Wert in der Spalte mehrfach auftritt
PRIMARY KEY	Kombiniert NOT NULL und UNIQUE. Jeder Datensatz in der Tabelle erhält einen eindeutigen Identifikator
FOREIGN KEY	Beschreibt referentielle Beziehungen zwischen Tabellen
CHECK	Sichert, dass die Werte in den Spaltenfeldern bestimmten Bedingungen genügen
DEFAULT	Setzt einen DEFAULT-Wert (Standardwert), der automatisch eingetragen wird, wenn kein anderer Wert eingetragen wurde
CREATE INDEX	Ermöglicht die Beschleunigung von Suchoperationen

- Beispiele für PRIMARY KEY, NOT NULL, DEFAULT und CHECK:

```
create table person (
personId int identity(1,1) primary key,
nachname nvarchar(50) not null, -- Erzwingt Eingabe!
vorname nvarchar(50),
geschlecht varchar(1) constraint chk_anrede check (geschlecht in ('m','w', 'd',,x')),
gehalt money default 0 constraint chk_gehalt check (gehalt>=0 and gehalt<=10000),
datum date default getdate());
```

Die Eigenschaft IDENTITY

- Im Beispiel wird das Feld personid mit der Eigenschaft IDENTITY versehen
- Die Eigenschaft IDENTITY(1,1) evtl. auch IDENTITY(m,n) mit m und n konkreten Zahlen (m ist ein Startwert, n ist die Schrittweite) teilt dem DBMS mit, dass bei jedem neuen Eintrag in die Tabelle der Wert des Feldes automatisch erhöht werden soll (INCREMENT, AUTOWERT)
- Dadurch wird gesichert, dass jeder neue Datensatz automatisch einen eindeutigen Identifikator erhält
- Der Standard-Startwert ist 1, kann aber auch größer sein.
- Die Standard-Schrittweite ist 1, kann aber auch größer sein
- Die Eigenschaft IDENTITY ist immer an einen ganzzahligen Datentyp gebunden (im allgemeinen integer, bigint)
- In anderen Datenbanken findet man für die gleiche Eigenschaft die Bezeichnungen Autowert, Auto_Increment, Serial

Die Eigenschaft GUID (Globale Unique ID)

- Normalerweise verwenden wir für die ID einer Tabelle den Datentyp int oder – bei sehr vielen Datensätzen – bigint sowie den CONSTRAINT Identity und die Festlegung als Primärschlüssel
- Es gibt Fälle, in denen man eine **Globale Unique ID** auf der Basis des Datentyps uniqueidentifier einführt, etwa wenn neue Datensätze zum gleichen Schema auf unterschiedlichen Maschinen generiert werden und zusammengeführt werden müssen.

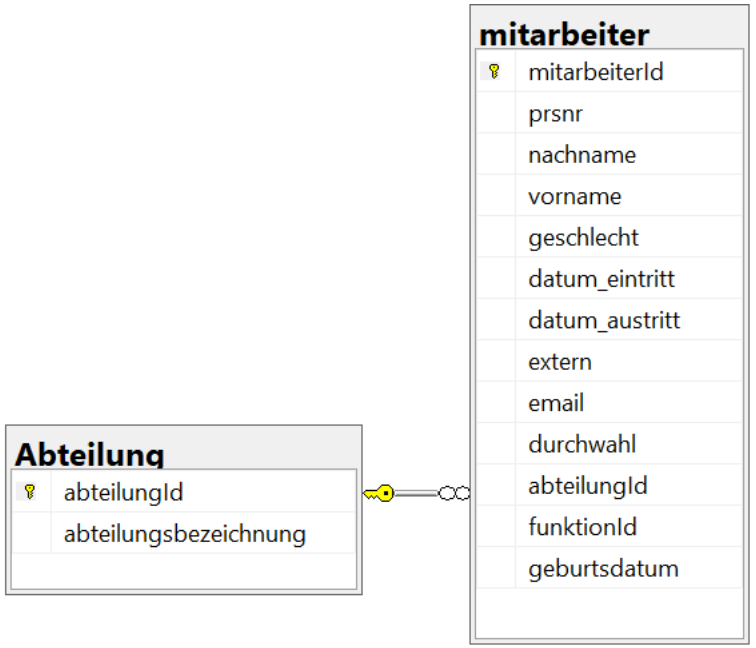
```
CREATE TABLE Kunde
(
 KundeID uniqueidentifier NOT NULL DEFAULT newid() PRIMARY KEY,
 Firma NVARCHAR(100) NOT NULL,
 Webseite VARCHAR(50) NOT NULL,
 Ansprechpartner NVARCHAR(60) NOT NULL,
 Strasse_HsNr NVARCHAR(30) NOT NULL,
 Ort NVARCHAR(30) NOT NULL,
 PLZ VARCHAR(10) NOT NULL,
 Land NVARCHAR(20) NOT NULL,
 Telefon VARCHAR(15) NOT NULL,
 EMail VARCHAR(30) NULL
);
```

```
SELECT NEWID() as GUID
```

```
GUID
7BE27C19-3EB5-4337-8541-16398F40F139
```

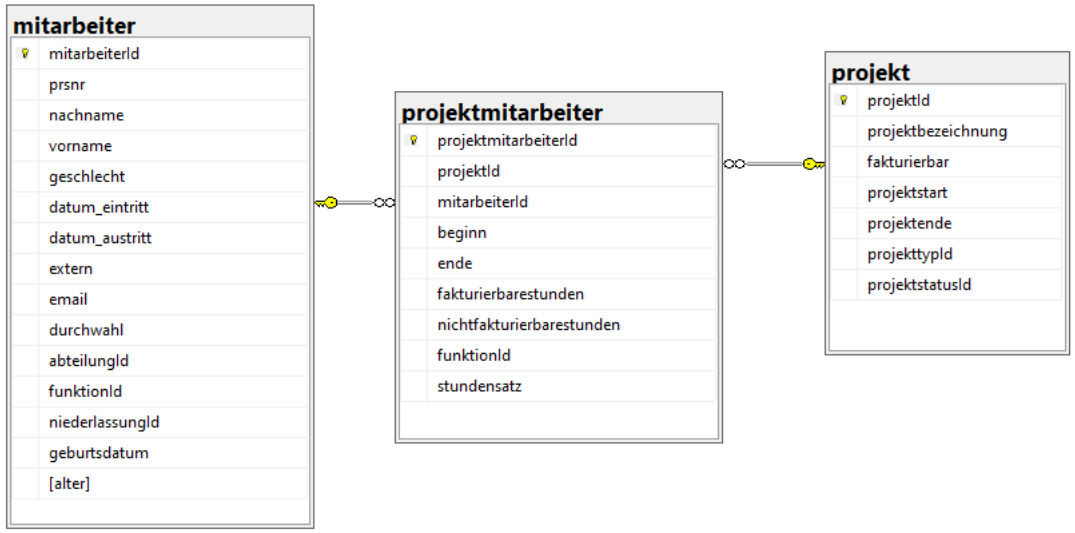
Die relationalen Beziehungen zwischen den Tabellen (1:n)

- Die Beziehungen zwischen Tabellen bilden reale Geschäftslogiken ab. Beispiele:
- 1:n – Beziehungen finden wir z.B. im Verhältnis von Abteilung und Mitarbeiter: Mehrere Mitarbeiter (1..n) gehören zu genau einer Abteilung. Ein Mitarbeiter kann nicht mehreren Abteilungen (gleichzeitig) angehören.



Die relationalen Beziehungen zwischen den Tabellen (m:n)

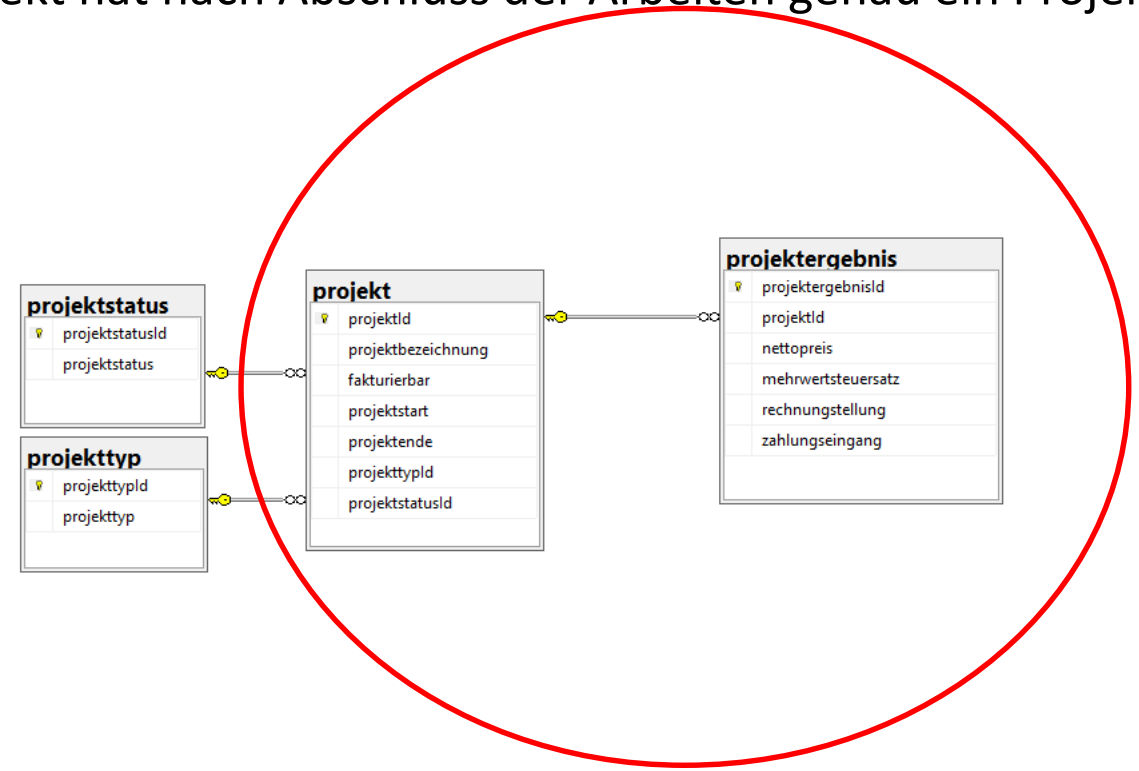
- Die Beziehungen zwischen Tabellen bilden reale Geschäftslogiken ab. Beispiele:
- m:n – Beziehungen: Solche Beziehungen finden wir z.B., wenn wir die Beziehungen zwischen Mitarbeiter (eines Unternehmens) und den Projekten (des Unternehmens) untersuchen und dabei auch den zeitlichen Aspekt berücksichtigen
  - Ein Mitarbeiter kann nicht oder gleichzeitig oder zu unterschiedlichen Zeiten in Projekten mitwirken (0…n)
  - Ein Projekt kann nicht oder gleichzeitig oder zu unterschiedlichen Zeiten mehrere Mitarbeiter beschäftigen (0…m)
  - Es kann also auch Mitarbeiter geben, die gar nicht in Projekten arbeiten (Buchhaltung …)
  - Es kann auch Projekte geben, denen (noch) keine Mitarbeiter zugeordnet wurde (neues Projekt in der Planungsphase)



Die m:n – Beziehung zwischen Projekt und Mitarbeiter wird in zwei Beziehungen 1:n (Mitarbeiter:Projektmitarbeiter), 1:m (Projekt-Projektmitarbeiter) aufgelöst

Die relationalen Beziehungen zwischen den Tabellen (1:1)

- Die Beziehungen zwischen Tabellen bilden reale Geschäftslogiken ab. Beispiele:
- 1:1 – Beziehungen: Ein Projekt hat nach Abschluss der Arbeiten genau ein Projektergebnis



Eingabe von Daten mittels INSERT-Befehlen

- Beispieldaten für die zwei Master-Tabellen Abteilung und Funktion:

```
INSERT [dbo].[Abteilung] ([Abteilungsbezeichnung]) VALUES (N'Nicht zugewiesen'),
(N'Geschäftsleitung'),(N'Rechnungswesen'),(N'Vertrieb/Marketing'),(N'Produktion');

INSERT [dbo].[Funktion] ([Funktionsbezeichnung]) VALUES (N'Nicht_zugewiesen'),
(N'Geschäftsführer_in'),(N'Bereichsleiter_in'),(N,Mitarbeiter_in'), (N'Berater_in'),(N'Entwickler_in');
```

- Beispieldaten für die Detail-Tabelle Mitarbeiter:

```
INSERT INTO dbo.mitarbeiter
(prsnr,nachname,vorname,geschlecht,datum_eintritt,datum_austritt,extern,email,durchwahl,abteilungId,funktionId,geburtsdatum)
VALUES
('1001','Paul','Gabriele','w','2006-01-01',Null,0,'gpaul@projektdb.de','20',2,2,'1993-08-23'),
('1002','Unterlauf','Karin','w','2012-05-15',Null,0,'kunterlauf@projektdb.de','22',2,4,'1970-02-06'),
('1003','Schiefner','Emi','w','2010-10-01',Null,0,'eschiefner@projektdb.de','21',2,7,'1980-05-14'),
('1004','Geist','Ottmar','m','2010-10-15',Null,0,'ogeist@projektdb.de','4567',2,6,'1979-05-13'),
('1005','Wendisch','Horst','m','2006-01-02',Null,0,'hwendisch@projektdb.de','123',2,6,'1981-05-14') usw.
```

- Da wir die Tabellen mit der IDENTITY-Constraint angelegt haben, müssen wir den Insert-Befehlen jeweils eine Ein- und Ausschaltanweisung voran- bzw. nachstellen:

```
SET IDENTITY_INSERT dbo.mitarbeiter ON;
-- Hier stehen die Insert-Anweisungen
SET IDENTITY_INSERT dbo.mitarbeiter OFF;
```

Ergänzungen: Beispiele für gespeicherte Abfragen (Views), eigene Funktionen

- Sicht/View vMitarbeiter

```
create view vMitarbeiter as
select m.MitarbeiterId,m.Nachname,m.Vorname,m.Geschlecht, m.Geburtsdatum,
a.Abteilungsbezeichnung [Abteilung], f.Funktionsbezeichnung [Funktion], m.DatumErstellung
from mitarbeiter m
inner join funktion f on f.FunktionId=m.FunktionId
inner join Abteilung a on a.AbteilungId=m.AbteilungId;
```

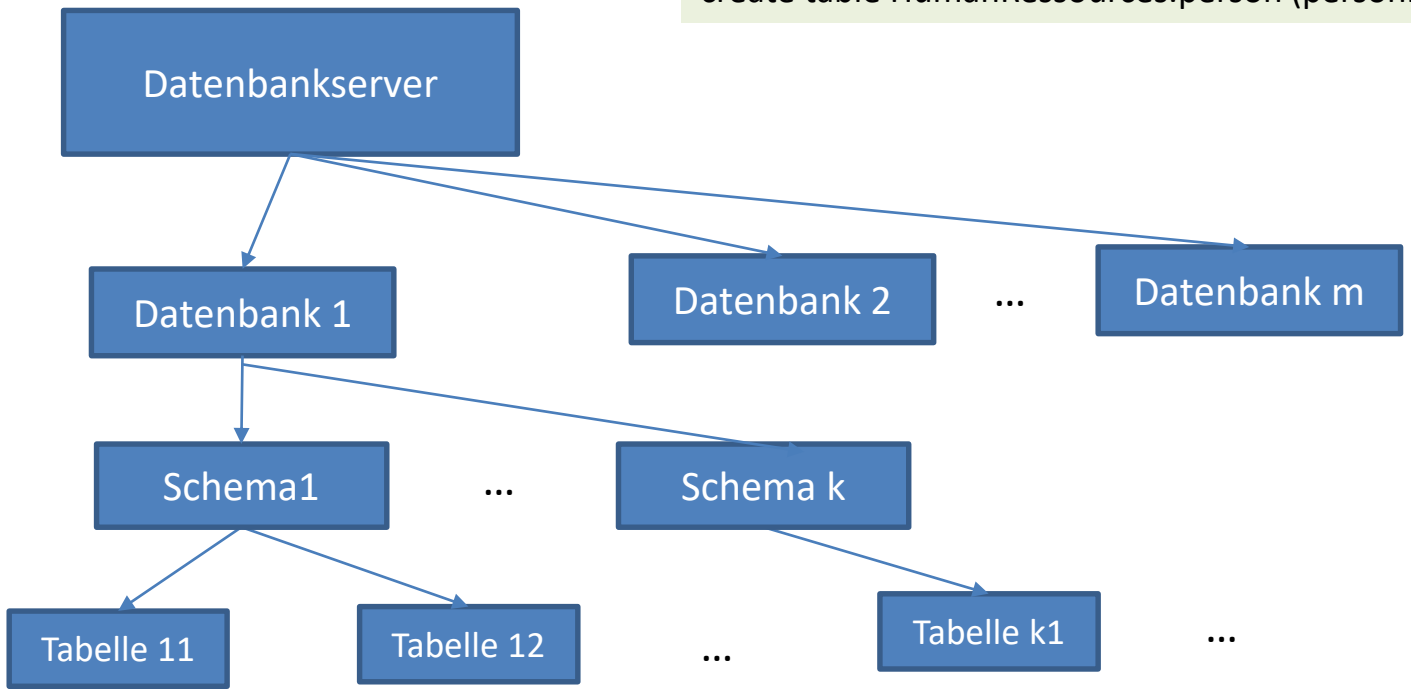
- Funktion zur Berechnung des Alters

```
create function dbo.udfBerechneAlter(@dt date)
returns int
begin
        declare @dtaktuell date = getdate();
        declare @alter int = datediff(year, @dt, @dtaktuell);
        declare @dtgeburtstagInDiesemJahr date = dateadd(year, @alter, @dt);
        if @dtgeburtstagInDiesemJahr > @dtaktuell
                set @alter = @alter-1;
        return @alter;
end;
```

Ergänzung: Schema

- Schemas sind Gliederungen unterhalb der Datenbank-Ebene. Sie treten als Präfix der Datenbank-Tabellen (oder Views oder Funktionen …) auf
- Das Standard-Schema ist dbo, es wird automatisch vergeben. Beispiel: die drei Tabellen in der Datenbank ADOTest sind dbo.mitarbeiter, dbo.funktion, dbo.abteilung
- Von dbo abweichende Schemas können eingeführt werden, um inhaltliche Bereiche in einer Datenbank voneinander zu unterscheiden:

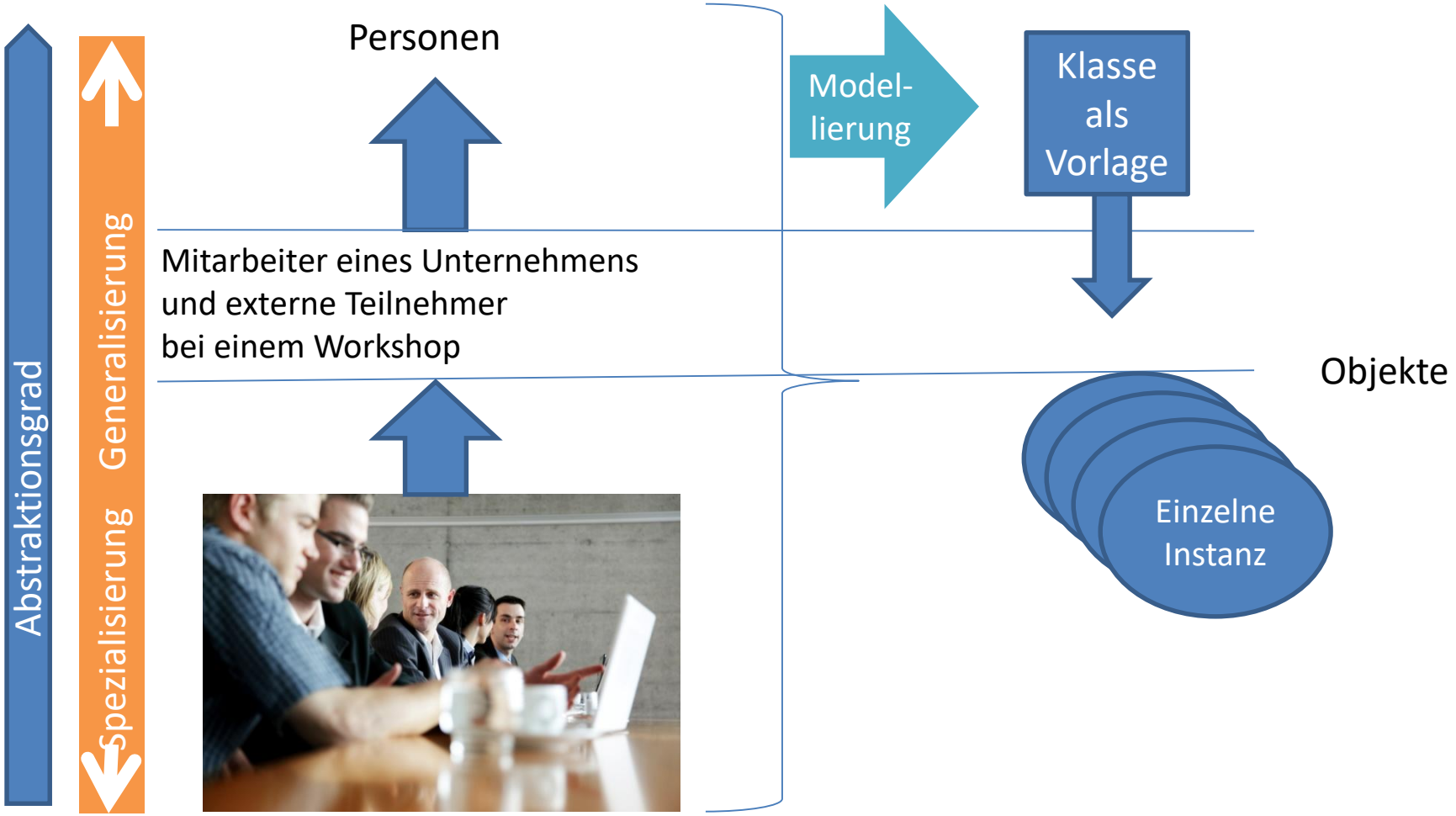
```
create schema HumanRessources;
create table HumanRessources.person (personid int identity(1,1) primary key, nachname nvarchar(59));
```



Im Folgenden

ELEMENTE DER OBJEKTORIENTIERTEN PROGRAMMIERUNG MIT C#

Das Klassenkonzept von C#: Von der Klasse zum Objekt

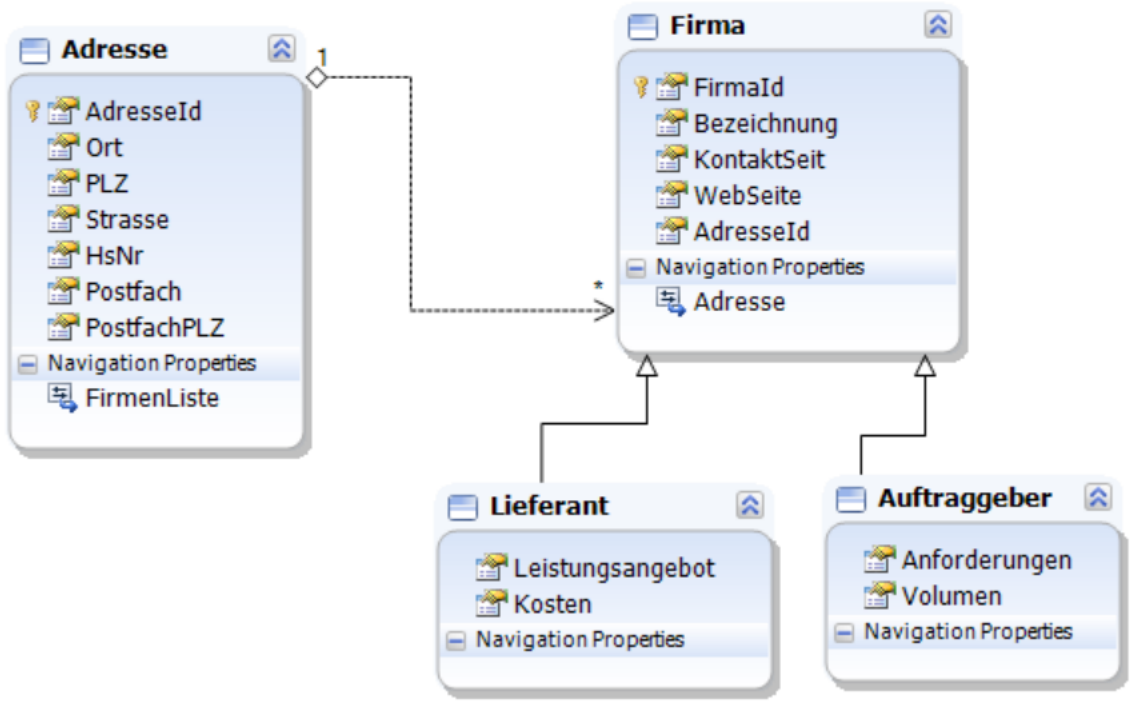


Das Konzept der Objektorientierten Programmierung

- Datenkapselung
  - Eine Klasse erlaubt in der Regel fremden Klassen keinen direkten Zugriff auf ihre Zustandsdaten. So wird das Risiko für das Auftreten inkonsistenter Zustände reduziert. Außerdem kann der Klassendesigner Implementierungsdetails ohne Nebenwirkungen auf andere Klassen ändern.
- Vererbung
  - Aus einer vorhandenen Klasse lassen sich zur Lösung neuer Aufgaben spezialisierte Klassen ableiten, die alle Member der Basisklasse erben. Hier findet eine Wiederverwendung von Software ohne lästiges und fehleranfälliges Kopieren von Quellcode statt. Beim Design der abgeleiteten Klasse kann man sich darauf beschränken, neue Member zu definieren oder bei manchen Erbstücken (z. B. Methoden) Modifikationen zur Anpassung an die neue Aufgabe vorzunehmen.
- Polymorphie
  - Über Referenzvariablen vom Typ einer Basisklasse lassen sich auch Objekte von abgeleiteten Klassen verwalten, wobei selbstverständlich nur solche Methoden aufgerufen werden dürfen, die schon in der Basisklasse definiert sind. Ist eine solche Methode in abgeleiteten Klassen unterschiedlich implementiert, führt jedes per Basisklassenreferenz angesprochene Objekt sein angepasstes Verhalten aus. Derselbe Methodenaufruf hat also unterschiedliche (polymorphe) Verhaltensweisen zur Folge. Welche Methode ausgeführt wird, entscheidet sich erst zur Laufzeit (späte Bindung). Dank Polymorphie ist eine lose Kopplung von Klassen möglich, und die Wiederverwendbarkeit von vorhandenem Code wird verbessert
- Assoziation
  - Klassen können auch direkt miteinander in Beziehung stehen und können damit auch auf Properties oder Methoden der assoziierten Klasse zugreifen

Vererbung

- Im Beispiel ist Firma eine Basisklasse, Lieferant und Auftraggeber erben die Basismerkmale (Properties) von dieser Basisklasse und erweitern diese Merkmale durch weitere, jeweils spezifische Merkmale



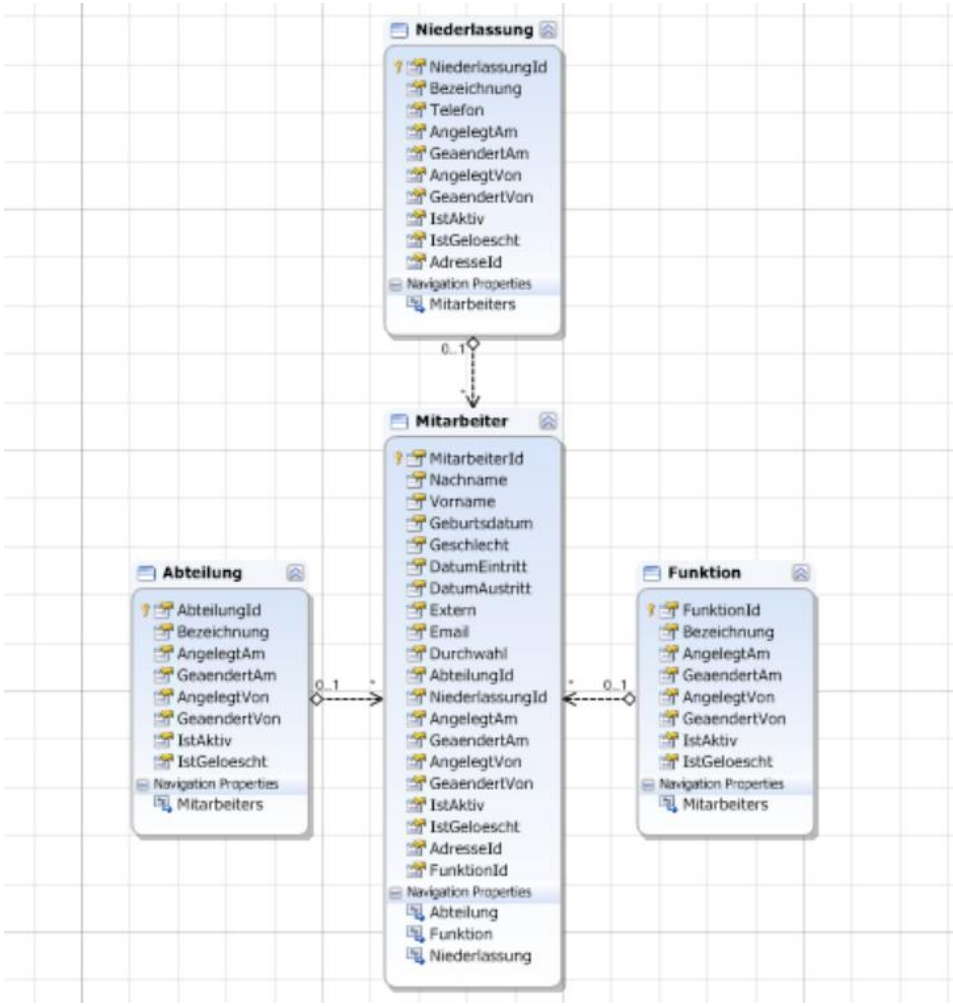
Weitere Konstrukte: Abstrakte Klassen, Interfaces (Schnittstellen) sowie Structs, Records

- In C# können Klassen nur von genau einer (konkreten oder abstrakten) Klasse erben!
- Abstrakte Klasse: Eine Klasse, die Properties und Methoden für andere, „konkrete“ Klassen bereit stellt, selbst aber nicht als Vorlage für Objekte dienen kann
- Interface: Ein Konstrukt, das die Signatur von Properties und Methoden bereit stellt. Eine Klasse, die ein Interface implementiert, muss diese Signaturen realisieren. In C# kann eine Klasse beliebig viele Interfaces implementieren
- Struct: Im Unterschied zu Klassen (= Referenz-Typen) sind Structs Valuetypen. Structs können z.B. nicht vererben oder erben.
- Record (= Datensatz): wurde eingeführt, um die Erzeugung unveränderlicher Objekte zu erleichtern

```
public record Personrecord(string FirstName, string LastName);
```

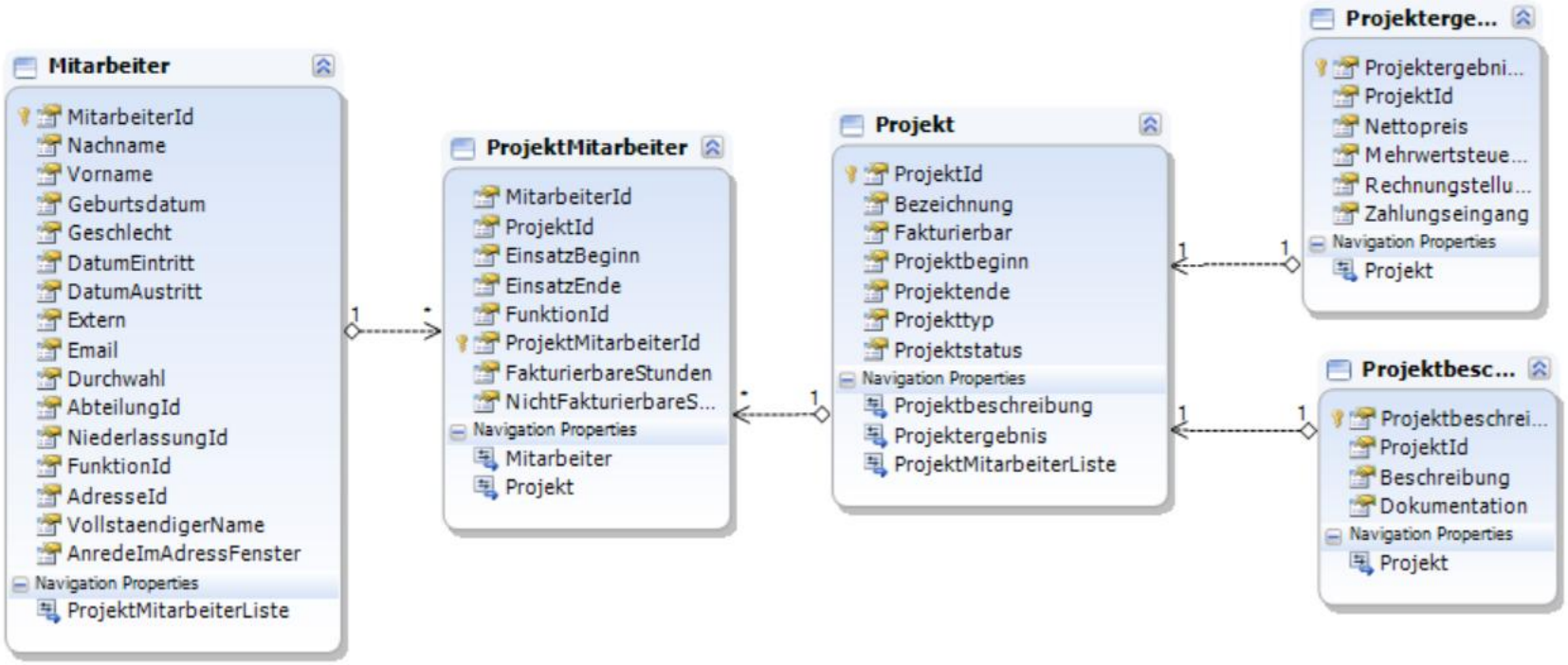
Assoziation: Beziehungen zwischen Klassen

- Assoziation 0,1 …*: Die Klassen Abteilung, Funktion und Niederlassung stehen jeweils mit der Klasse Mitarbeiter in einer unidirektionalen oder bidirektionalen Beziehung (unidirektional: Abteilung kann z.B. eine E-Mailliste aller zugehörigen Mitarbeiter anfordern, bidirektional: auch ein Mitarbeiter kann auf Abteilungsmerkmale zugreifen). Man spricht in diesem Zusammenhang auch von Navigationseigenschaften



Assoziation: Komplexe Beziehungen zwischen Klassen

- Assoziation 1 …* - * … 1: Die Klassen Mitarbeiter und Projekt stehen miteinander in Beziehung durch Vermittlung einer Zuordnungsklasse Projektmitarbeiter. Dadurch wird eine m:n – Beziehung zwischen Mitarbeiter und Projekt realisiert. Zum Projekt wie auch zum Mitarbeiter gibt es daher immer eine ProjektmitarbeiterListe.
- Assoziation 1… 1: Ein Projektobjekt ist immer mit maximal einem Projektergebnisobjekt und einem Projektbeschreibungobjekt verknüpft



LESENDER UND SCHREIBENDER ZUGRIFF AUF DIE DATENBANK MIT ADO.NET

Lesender und schreibender Zugriff auf die Datenbank mit ADO.NET

- Der lesende und schreibende Zugriff auf Datenbanken gehört zu den zentralen Aufgaben der Datenverarbeitung
- Microsoft stellte mit der Technologie in den 2000ern mit ADO.NET ein leistungsfähiges Framework für diese Aufgaben zu Verfügung, das sich für den Zugriff auf relationale Datenbanken wie MS SQL Server, MySQL, DB2, PostgreSQL, Oracle aber auch auf Datenquellen wie Excel, Access eignet.

- Die zentralen Bibliotheken sind
  - im Fall des SQL Servers → Microsoft.Data.SqlClient (löst System.Data.SqlClient ab)
  - Im Fall von OleDB-Quellen (Access, Excel oder per ODBC definierte Quellen) → System.Data.OleDb

Zugriff auf eine Datenbank (am Beispiel SQL Server)

Der Zugriff auf eine Datenbank mit C# besteht aus folgenden Schritten:

- Verbindung aufnehmen zur Datenbank: Connection-Klasse, speziell SQLConnection (aus Microsoft.Data.SqlClient)
  - Absetzen eines SQL-Befehls an die Datenbank: Command-Klasse, speziell SQLCommand
    - Auswerten des SQL-Befehls: Execute
      - Handelt es sich um einen Select-Befehl, werden Daten zurückgegeben und von einem SqlReader oder von SqlDataSets aufgenommen
      - In allen anderen Fällen (Insert, Update, Delete, Anstoßen einer Stored Procedure erfolgt keine Datenrückgabe, bestenfalls als Meldung über die Anzahl der betroffenen Zeilen
  - Beenden des Command-Blocks
- Verbindung zur Datenbank schließen

Verbindungsaufnahme (am Beispiel des SQL Servers)

Die Verbindungsaufnahme wird durch einen so genannten Connection String gesteuert, der die wichtigsten Parameter für die Verbindungsaufnahme enthält:

```
private static string sqlConnectionString
          = "Data Source=localhost;Initial Catalog=ADOTest;
          Integrated Security=True";TrustServerCertificate=Yes;Encrypt=False;“
```

Ein Connection String besteht aus Angaben zum Server, zur Datenbank und aus Anmeldedaten (Trusted Connection oder Username und Passwort)

Wir werden später sehen, wie wir den Connectionstring in eine externe Konfigurationsdatei einbetten (appsettings.json), auf die dann unsere Bibliotheken zur Laufzeit zugreifen werden:

```
"DBConnectionString": "data source=localhost;initial catalog=ProjektDB01;
          Integrated Security=true;TrustServerCertificate=Yes;Encrypt=False;"
```

Verbindungsaufnahme (am Beispiel des SQL Servers)

- Verbindung aufnehmen zur Datenbank: Connection-Klasse, speziell SQLConnection (aus Microsoft.Data.SqlClient)
  - Absetzen eines SQL-Befehls an die Datenbank: Command-Klasse, speziell SQLCommand
    - Auswerten des SQL-Befehls: Execute
      - Handelt es sich um einen Select-Befehl, werden Daten zurückgegeben und von einem SqlReader oder von SqlDataSets aufgenommen
      - In allen anderen Fällen (Insert, Update, Delete, Anstoßen einer Stored Procedure erfolgt keine Datenrückgabe, bestenfalls als Meldung über die Anzahl der betroffenen Zeilen
  - Beenden des Command-Blocks
- Verbindung zur Datenbank schließen

- Das SQLConnection-Objekt: Mittels einer using-Anweisung wird ein scope geschaffen, in dem das Öffnen der Verbindung probiert wird

```
using (var con = new SqlConnection(connectionString))
{
        try {
            con.Open();
                        …
        }
        catch (SqlException ex)
        {
                        ….
        }
}
```

- Am Ende der Abarbeitung wird die Verbindung automatisch geschlossen (using-Anweisung!)

Das Command-Object
(am Beispiel des SQL Servers)

- Verbindung aufnehmen zur Datenbank: Connection-Klasse, speziell SQLConnection (aus Microsoft.Data.SqlClient)
  - Absetzen eines SQL-Befehls an die Datenbank: Command-Klasse, speziell SQLCommand
    - Auswerten des SQL-Befehls: Execute
      - Handelt es sich um einen Select-Befehl, werden Daten zurückgegeben und von einem SqlReader oder von SqlDataSets aufgenommen
      - In allen anderen Fällen (Insert, Update, Delete, Anstoßen einer Stored Procedure erfolgt keine Datenrückgabe, bestenfalls als Meldung über die Anzahl der betroffenen Zeilen
  - Beenden des Command-Blocks
- Verbindung zur Datenbank schließen

Das SqlCommand-Objekt: Mittels einer using-Anweisung wird ein weiteres scope geschaffen, in dem das Command-Objekt initialisiert wird: Dem Command-Objekt wird das Connection-Objekt der Sql-Befehl (Variable sqlStatement) übergeben:

string sqlStatement

```
using (var con= new SqlConnection(connectionString))
{
        try {
                con.Open();
                using (var cmd = new SqlCommand(sqlStatement, con))
                {
                        …
                } //Hier wird das Command-Objekt geschlossen
            }
            catch (SqlException ex)
            {
                        ….
            }
} //Hier wird die Verbindung geschlossen
```

Lesender Zugriff
(am Beispiel des SQL Servers)

- Verbindung aufnehmen zur Datenbank: Connection-Klasse, speziell SQLConnection (aus Microsoft.Data.SqlClient)
  - Absetzen eines SQL-Befehls an die Datenbank: Command-Klasse, speziell SQLCommand
    - Auswerten des SQL-Befehls: Execute
      - Handelt es sich um einen Select-Befehl, werden Daten zurückgegeben und von einem SqlReader oder von SqlDataSets aufgenommen
      - In allen anderen Fällen (Insert, Update, Delete, Anstoßen einer Stored Procedure erfolgt keine Datenrückgabe, bestenfalls als Meldung über die Anzahl der betroffenen Zeilen
  - Beenden des Command-Blocks
- Verbindung zur Datenbank schließen

Das SqlDataReader-Objekt

Über das Command-Objekt wird das Lesen des Schemas der Tabelle (oder Sicht) und anschließend der Daten veranlasst. Die Daten werden dem Reader übergeben. Das Reader-Objekt muss direkt ausgewertet werden, da es die Verbindung nicht „überlebt“

```
using (SqlConnection con = new SqlConnection(connectionString))
{
        try {
                con.Open();
                using (SqlCommand cmd =
                                new SqlCommand(sqlStatement, con))
                {
                        var oReader= cmd.ExecuteReader();
                        While oReader.read()
                        {
                                //Hier werden die Daten gesammelt
                        }

                }
            }
            catch (SqlException ex)
            {
                        ....
            }
}
```

Lesender Zugriff und Rückgabe eines DataSets (am Beispiel des SQL Servers)

- Das DataSet-Objekt ist deutlich stärker als das Reader-Objekt, da es trotz der geschlossenen Verbindung im Arbeitsspeicher gehalten wird und deshalb separat ausgewertet, evtl. verändert und evtl. an die Datenbank zurückgegeben werden kann
- Als Mittler zwischen der Datenbank und dem Dataset-Objekt dient der DataAdapter. Das SqlDataAdapter-Objekt öffnet die Verbindung zur Datenbank im Bedarfsfall. Es weist das SqlCommand-Objekt an, das es aus seiner SelectCommand-Eigenschaft ermittelt, die Schemainformationen des Resultsets einzuholen, das zuvor beispielsweise durch eine SELECT-Anweisung abgefragt wurde. Es baut aus diesen Schemainformationen die Grundstruktur eines neuen DataTable-Objekt auf. Es verwendet anschließend ein SqlDataReader-Objekt, das durch das SqlCommand-Objekt angelegt wurde, um das DataTable-Objekt mit Daten zu füllen.
- Es schließt die Verbindung zur Datenbank.

Lesender Zugriff und Rückgabe eines DataSets (am Beispiel des SQL Servers)

- Verbindung aufnehmen zur Datenbank: Connection-Klasse, speziell SQLConnection (aus Microsoft.Data.SqlClient)
  - Absetzen eines SQL-Befehls an die Datenbank: Command-Klasse, speziell SQLCommand
    - Auswerten des SQL-Befehls: Execute
      - Handelt es sich um einen Select-Befehl, werden Daten zurückgegeben und von einem SqlReader oder von SqlDataSets aufgenommen
      - In allen anderen Fällen (Insert, Update, Delete, Anstoßen einer Stored Procedure erfolgt keine Datenrückgabe, bestenfalls als Meldung über die Anzahl der betroffenen Zeilen
  - Beenden des Command-Blocks
- Verbindung zur Datenbank schließen

```
{
  DataSet ds = null;
   using (var con = new SqlConnection(connectionString))
   {
     con.Open();
      using (var cmd = new SqlCommand(sqlStatement, con))
      {
        using (var dap = new SqlDataAdapter())
        {
           dap.SelectCommand = cmd;
           dap.TableMappings.Add(tableName, sqlStatement);
           ds = new DataSet();
           ds.CaseSensitive = true;
           dap.Fill(ds);
           return ds;
        }
      }
   }
}
```

Änderung von Daten (am Beispiel des SQL Servers)

- Verbindung aufnehmen zur Datenbank: Connection-Klasse, speziell SQLConnection (aus Microsoft.Data.SqlClient)
  - Absetzen eines SQL-Befehls an die Datenbank: Command-Klasse, speziell SQLCommand
    - Auswerten des SQL-Befehls: Execute
      - Handelt es sich um einen Select-Befehl, werden Daten zurückgegeben und von einem SqlReader oder von SqlDataSets aufgenommen
      - In allen anderen Fällen (Insert, Update, Delete, Anstoßen einer Stored Procedure erfolgt keine Datenrückgabe, bestenfalls als Meldung über die Anzahl der betroffenen Zeilen
  - Beenden des Command-Blocks
- Verbindung zur Datenbank schließen

```
using (var con = new SqlConnection(sqlConnectionString))
{
          con.Open();
          using var cmd = new SqlCommand(sqlStatement, con))
          {
                try
                {
                    nRows = cmd.ExecuteNonQuery();
                    msg = string.Format($"Es sind {nRows} Datensätze betroffen");
                }
                catch (Exception ex)
                {
                    errmsg += "\n" + ex.Message.ToString();
                    success = false;
                }
          }
}
```

Ein Insert, Update, Delete-Befehl oder auch das Anstoßen einer Stored Procedure erfolgt mit der Methode ExecuteNonQuery() des Command-Objekts

Für jede Operation muss das SQL-Statement einzeln aufgebaut werden. Dabei müssen die Properties der C#-Klassen auf die Tabellen und Spalten der SQL-Datenbank und die C#-Datentypen auf die SQL-Server-Datenbank-Felddatentypen gemappt werden. Bei Abfragen (Read) erfolgt das in umgekehrter Richtung.

SqlParameter

- Verbindung aufnehmen zur Datenbank: Connection-Klasse, speziell SQLConnection (aus Microsoft.Data.SqlClient)
  - Absetzen eines SQL-Befehls an die Datenbank: Command-Klasse, speziell SQLCommand
    - Auswerten des SQL-Befehls: Execute
      - Handelt es sich um einen Select-Befehl, werden Daten zurückgegeben und von einem SqlReader oder von SqlDataSets aufgenommen
      - In allen anderen Fällen (Insert, Update, Delete, Anstoßen einer Stored Procedure erfolgt keine Datenrückgabe, bestenfalls als Meldung über die Anzahl der betroffenen Zeilen
  - Beenden des Command-Blocks
- Verbindung zur Datenbank schließen

- Zur Vermeidung bösartiger Angriffe (SQL Injektion) sollte man in den Statements mit Parametern arbeiten. Statt zu schreiben

```
SqlCommand cmd = new SqlCommand( "select * from Teilnehmer where Herkunftsland= ‚USA‘", con);
```

- führen Sie Parameter ein:

```
SqlCommand cmd = new SqlCommand( "select * from Teilnehmer where Herkunftsland=
@Herkunftsland, con);
```

- Solche Parameter werden durch ein Parameterobjekt übergeben:

```
SqlParameter param=new SqlParameter();
param.ParameterName=@Herkunftsland;
param.Value=„USA“
```

- Innerhalb des Command-Blocks werden die Parameter dann übergeben:

```
cmd.Parameters.Add(param);
```

Das Klassenmodell und das (mehr oder weniger manuelle) Mapping zwischen Datenbank-Tabellen und Klassen im klassischen ADO.NET-Verfahren

- Die drei mit den Tabellen korrespondierenden C#-Klassen werden zunächst unabhängig entwickelt

```
public class Abteilung
{
    public int AbteilungId { get; set; }
    public string Abteilungsbezeichnung { get; set; }

}
```

```
public class Funktion
{
   public int FunktionId { get; set; }
   public string Funktionsbezeichnung { get; set; }
}
```

- sowie eine Enumeration:

```
public enum EGeschlecht
{
   w, m, d, x
}
```

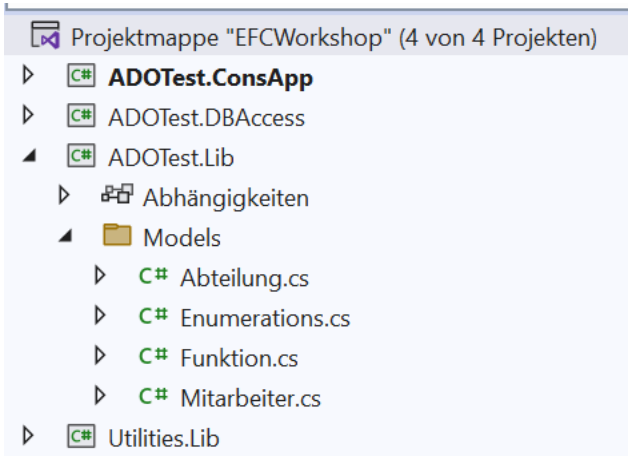
```
public  class Mitarbeiter
{

   public int MitarbeiterId { get; set; }
   public string prsnr { get; set; }
   public string Vorname? { get; set; }
   public string Nachname { get; set; }
   public EGeschlecht Geschlecht { get; set; }
   public DateOnly Geburtsdatum { get; set; }
   public DateOnly? DatumEintritt { get; set; }
   public DateOnly? DatumAustritt { get; set; }
   public Boolean Extern { get; set; }
   public string? Email { get; set; }
   public string? Durchwahl { get; set; }

   public Funktion Funktion { get; set; }
   public Abteilung Abteilung { get; set; }

}
```

Im Folgenden

EINE BEISPIELANWENDUNG ZUM STUDIUM VON SQL SERVER-TECHNOLOGIEN UND C#-KONZEPTEN, INSBESONDERE ADO.NET UND LINQ

Der Aufbau unser ADOTest-Anwendung

- Wir führen eine Projektmappe EFCWorkshop ein
- Die ADO-Test-Anwendung besteht aus einer Konsolen-Applikation mit der Main-Klasse
  - Modell-Bibliothek (ADOTest.Lib) mit den Klassen Mitarbeiter, Funktion, Abteilung
  - einer Datenzugriffsbibliothek (ADOTest.DBAccess) mit einer Hilfsklasse DBAccess, die die Methoden für den Datenbankzugriff enthält.
  - Einer Utility-Bibliothek, die die Display-Funktion und evtl. weitere übergreifende Methoden enthält

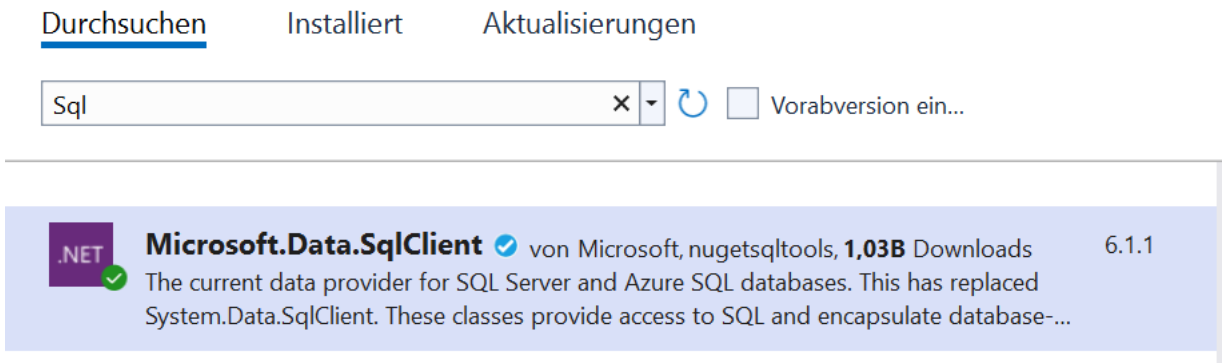


Abhängigkeiten in ADOTest.ConsApp::
1. Projektabhängigkeit ADOTest.DBAccess
2. Projektabhängigkeit ADOTest.Lib
3. Projektabhängigkeit Utilitties.Lib

Abhängigkeiten in ADOTest.DBAccess:
1. Microsoft.Data.SqlClient
2. Projektabhängigkeit ADOTest.Lib

Vorbereitungen zum Zugriff zur Datenbank in Visual Studio

- In der Bibliothek ADOTest.DBAccess benötigen wir einen Projektverweis auf die Modell-Klassen (ADOTest.Lib.Models)
- Ferner müssen wir das Paket System.Data.SqlClient installieren. Das stellt das klassische „Framework“ für den Datenbankzugriff dar. Hierzu wird der NuGet-Paketmanager aufgerufen:



- Zum Aufbau einer Verbindung benötigen wir entsprechende Angaben über den SQL Server und die Datenbank, die wir einer statischen Variablen zuweisen:

```
private static string sqlConnectionString
          = "Data Source=localhost;Initial Catalog=ADOTest;
          Integrated Security=True";TrustServerCertificate=Yes;Encrypt=False;
```

Lesender Zugriff auf die Daten mittels ADO.NET
Beispiel: Laden einer Abteilungsliste

```
public static List<Abteilung> GetDBAbteilungsListe(string sqlStatement)
{
    List<Abteilung> abtListe = new List<Abteilung>();
    using (var con = new SqlConnection(sqlConnectionString))
    {
          con.Open();
          using (var cmd = new SqlCommand(sqlStatement, con))
          {
              using (System.Data.SqlDataReader oReader = cmd.ExecuteReader())
              {
                    if (oReader.HasRows)
                    {
                          while (oReader.Read())
                          {
                              Abteilung ab = new Abteilung()
                              {
                                  AbteilungId = oReader.GetInt32(0),
                                  Abteilungsbezeichnung = oReader.GetString(1)
                              };
                              abtListe.Add(ab);
                          }
                    }
              }
          }
   }
   return abtListe;
}
```

Hier findet das „Mapping“ statt!

Lesender Zugriff auf die Daten mittels ADO.NET
Beispiel: Laden einer Mitarbeiterliste

```
public static List<Mitarbeiter> GetDBMitarbeiterListe(string sqlStatement = "SELECT [mitarbeiterId], [nachname], [vorname], [geschlecht], [geburtsdatum], [abteilungId],[funktionId] FROM [dbo].[mitarbeiter];")
{
    List<Mitarbeiter> mitListe = new List<Mitarbeiter>();
    using (var con = new SqlConnection(sqlConnectionString))
    {
          con.Open();
          using (var cmd = new SqlCommand(sqlStatement, con))
          {
                    using (SqlDataReader oReader = cmd.ExecuteReader())
                    {
                        if (oReader.HasRows)
                        {
                            while (oReader.Read())
                            {
                                Mitarbeiter mit = GetMitarbeiter(oReader);
                                mitListe.Add(mit);
                            }
                        }
                    }
          }
    }
    return mitListe;
}
```

Hier findet das „Mapping“ per Hilfsfunktionen statt!

Hilfsfunktionen für das Mapping

- Die Zuordnung der Feldwerte aus den Tabellenfeldern zu den Properties der Klasse erfolgt über diese Hilfsfunktion:

```
private static Mitarbeiter GetMitarbeiter(SqlDataReader oReader)
{
    Funktion fkt = new Funktion(){
                    FunktionId = oReader.GetInt32(6),
                      Abteilungsbezeichnung = GetValueFromTable("Funktion", "Funktionsbezeichnung", "FunktionId=" + fkt.FunktionId.ToString())
    };

    Abteilung abt = new Abteilung(){
                    AbteilungId = oReader.GetInt32(5),
                    Abteilungsbezeichnung = GetValueFromTable("Abteilung", "Abteilungsbezeichnung", "AbteilungId=" + abt.AbteilungId.ToString())
    };

    Mitarbeiter mit = new Mitarbeiter(){
       MitarbeiterId = oReader.GetInt32(0),
       Nachname = oReader.GetString(1),
       Vorname = oReader.GetString(2),
       Geburtsdatum =DateOnly.FromDateTime(oReader.GetDateTime(3)),
       Geschlecht = ToEnum<Enumerations.EGeschlecht>(oReader.GetString(4), Enumerations.EGeschlecht.w),
       Funktion = fkt,
       Abteilung = abt
     };
     return mit;
}
```

Hilfsfunktionen für das Mapping

- Eine häufig auftretende Aufgabe ist die Gewinnung eines Feldwertes in einer Mastertabelle aus der Kenntnis des Fremdschlüsselwertes: `select abteilungsbezeichnung from abteilung where abteilungid=@id;`

```
public static string GetValueFromTable(string table, string fieldname, string criterion)
{
    string wert = "";
    string sqlStatement = "select " + fieldname + " from " + table + " where " + criterion;
    using (var con = new SqlConnection(sqlConnectionString))
    {
        con.Open();
        using (var cmd = new SqlCommand(sqlStatement, con))
        {
             using (var oReader = cmd.ExecuteReader())
             {
                 if (fieldname != null && oReader.HasRows)
                 {
                 while (oReader.Read())
                       wert = oReader.GetString(0);
              }
          }
        }
     }
     return wert;
}
```

Probleme mit ADO.NET

- Bei der traditionellen ADO.NET - Arbeitsweise mit untypisierten DataSets werden Tabellen und Spalten durch Indexer-Argumente vom Zeichenfolgentyp angesprochen, und auf die Werte einer Tabellenzeile greift man über die DataRow-Eigenschaft ItemArray mit dem Datentyp Object[] zu, was lästige und fehleranfällige Typumwandlungen erfordert.
- Probleme des Datenzugriffs mit schwacher Typisierung werden auch durch typisierte DataSets überwunden. Allerdings bleibt bei diesem Ansatz der Paradigmenbruch zwischen der objektorientierten Programmierung einerseits und der Bearbeitung von Tabellenzeilen andererseits bestehen.

Im Folgenden:

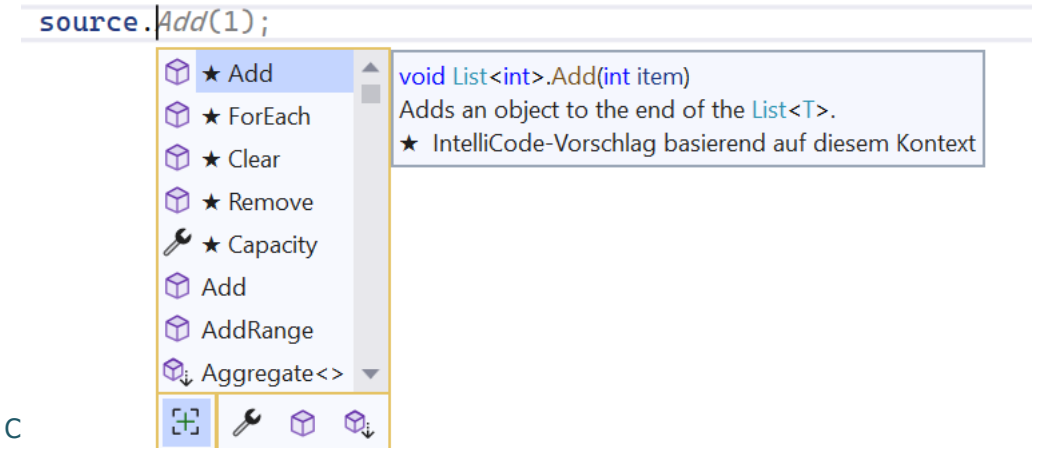
AUSWERTUNG DER DATEN MIT LINQ

Auswertung der Daten mit Hilfe von LINQ (Language Integrated Query) to Objects

- Seit C# 3.0 ist eine einheitliche, für diverse Datenquellen geeignete, an SQL angelehnte Abfragetechnik namens LINQ (Language Integrated Query) verfügbar.
- Mit LINQ-Methoden können Auswertungen von Kollektions-Objekten oder Arrays vorgenommen werden.
- Wir werden sogenannte LINQ-to-Objects-Abfragen oder lokale Abfragen durchführen, später, wenn wir Entity Framework Core kennengelernt haben, werden wir LINQ-to-Entities-Abfragen durchführen, um Informationen aus SQL-Datenbanken zu beziehen
- Der Hintergrund sind Erweiterungsmethoden im Namensraum System.Linq für alle Typen, welche die Schnittstelle IEnumerable implementieren (z.B. solche Typen wie List<T>, Array<T>)
- Hat man z.B. eine Liste definiert

```
List<int> source = new List<int> { 1, 2, 3, 4, 5, 6 };
```

kann man nach dem Setzen des Punktes hinter `source` aus der Liste der Erweiterungsmethoden wählen.



Von der klassischen Methode zum Delegaten, über Delegaten (Action) zu Lambda

- Aufgabe: Beim Durchlaufen einer Liste sollen die Elemente auf der Konsole angezeigt werden

```
List<int> source = new List<int> { 1, 2, 3, 4, 5, 6 }
```

```
public static void Display(dynamic x)
{
        Console.WriteLine(x); //Schreibe auf Console
}
```

- Klassisch:

```
foreach (var element in source)
{
    Display(element.ToString());
}
```

- Jetzt mit dem Delegaten Action:

```
Action<int> act = Display;
source.ForEach(act);
```

- Und mit Lambda-Notation:

```
source.ForEach(x => Display(x));
```

Beispiele für funktionale Programmierung

Dasselbe mit einer Liste von Mitarbeitern

- Dieses Beispiel zeigt die LINQ-Technik für eine Liste aus Mitarbeitern

```
private static void LINQBeispiel02()
{
            List<Mitarbeiter> source = new List<Mitarbeiter> {
                new Mitarbeiter { MitarbeiterId = 1, prsnr= "1001", Nachname = "Paul", Vorname = "Gabriele",
                    Geschlecht = EGeschlecht.w, Geburtsdatum = new DateTime(1993, 08, 23) },
                new Mitarbeiter { MitarbeiterId = 2, prsnr = "1002", Nachname = "O‘Reilley", Vorname = "Charles",
                    Geschlecht = EGeschlecht.w, Geburtsdatum = new DateTime(1970, 02, 06) },
                new Mitarbeiter { MitarbeiterId = 3, prsnr = "1003", Nachname = "Makarian", Vorname = "Emi",
                    Geschlecht = EGeschlecht.w, Geburtsdatum = new DateTime(1980, 05, 14) }
            };

            Display("Klassisches Durchlaufen der Mitarbeiterliste { m1, m2, m3 } mit Kriterium Geburtsjahr >=1980 ");
            foreach (var m in source)
            {
                if (m.Geburtsdatum.Year>=1980) {
                    Display($"{m.Nachname}, {m.Vorname}");
                }
            }
            //Jetzt mit LINQ und Lambda-Notation:
            Display("\nDurchlaufen der Mitarbeiterliste  mit LINQ, mit Kriterium Geburtsjahr >=1980“);
            source
                .Where(m => m.Geburtsdatum.Year >= 1980)
                .ToList()
                .ForEach(m => Display($"{m.Nachname}, {m.Vorname}"));
}
```

Auswertung von Datenbank-Daten – „Klassisch“ oder mittels LINQ (Language Integrated Query)

- Nach dem Zugriff auf die Daten in einer Datenbank-Tabelle, z.B. Mitarbeiter, können diese angezeigt und weiter ausgewertet werden.
- Beispiel: Uns interessieren alle Mitarbeiter in der Abteilung Infrastruktur mit ihrer jeweiligen Funktion

```
string suchbegriff= "Infra";
var mitListe = DBAccess.GetDBMitarbeiterListe
```

- „Klassisches“ Vorgehen:

```
foreach (var m in mitListe)
{
     if (suchBegriff != "" && m.Abteilung.Abteilungsbezeichnung.Contains(suchbegriff))
     {
          Display($" -- Suchbegriff {suchBegriff} gefunden!");
          Display($"{m.Nachname}, {m.Vorname}, Funktion: {m.Funktion.Funktionsbezeichnung}");
     }
}
```

- Query-Notation:

```
var mitListe2 = from m in DBAccess.GetDBMitarbeiterListe()
                where (m.Geburtsdatum.Value.Year >= 1990)
                orderby (m.Abteilung.Abteilungsbezeichnung, m.Nachname)
                select m;
```

- Vorgehen mit LINQ-To-Objects und Lambda-Notation:

```
var mitListe3 = DBAccess.GetDBMitarbeiterListe()
                .Where(m => m.Abteilung.Abteilungsbezeichnung.Contains(suchBegriff))
                .OrderBy(m => m.Nachname)
                .ToList()
                .ForEach(m => Display($"{m.Nachname}, {m.Vorname}. Funktion: {m.Funktion.Funktionsbezeichnung}"));
```

Zusammenfassung LINQ

Typische Aufgaben	Beispiele für LINQ-Syntax
Sortieren	OrderBy, OrderByDescending, Reverse
Filtern	Where
Einzelnes Element selektieren	First, FirstOrDefault, Find
Projektion (Auswahl einzelner Felder)	Select
Aggregation	Max, Min, Sum, Count, Average
Partition	Skip, Take
Logischer Ausdruck (Test auf Wahr oder Falsch)	Any, All, Contains, Vergleichsoperatoren