C# Programmierung - Sprache und .NET Grundlagen

Programmieren in C#

Organisation

- Kursablauf
  - 4 Module pro Tag
  - Modul: Theorie → Live Coding → Übungen
- Code wird auf GitHub hochgeladen
- Folien im Premiumportal
- Unklarheiten/Fragen: **Sofort** unterbrechen und Nachfragen

ppedv AG

- Firmensitz in Burghausen
- Schulungen für
  - Programmiersprachen
  - KI-Tools
  - nahezu alle Microsoft-Technologien
- Website: www.ppedv.de
- Blog-Artikel: https://blog.ppedv.de/
- Events: https://ppedv.de/Events/

Vorstellungsrunde

- Trainer
- Teilnehmer
  - Persönliches, Firma
  - Programmiererfahrung
  - Wünsche

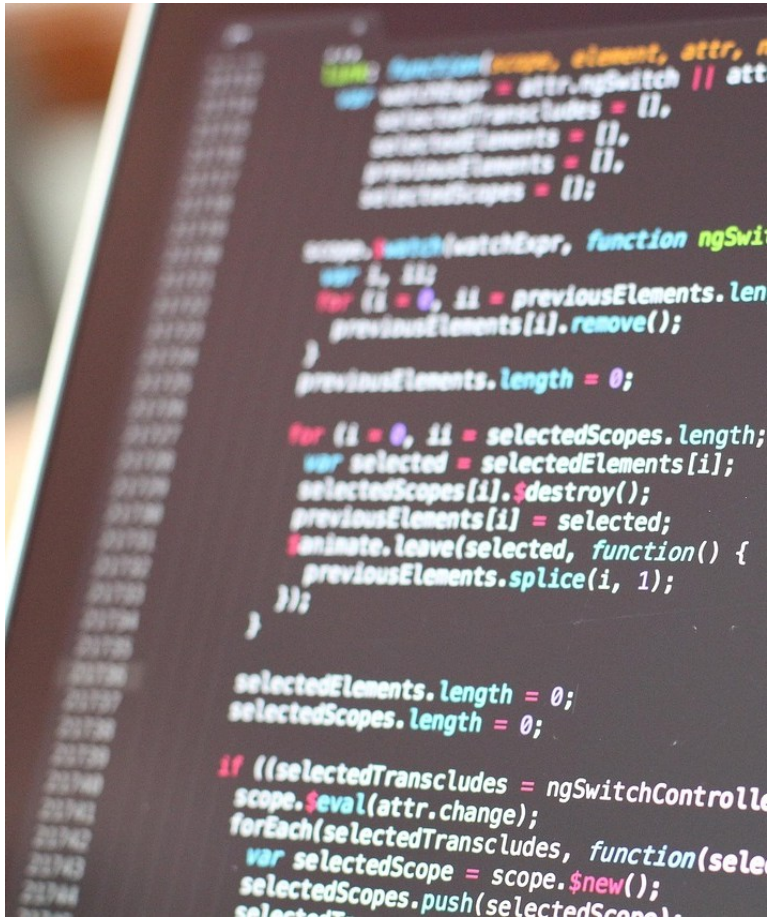
Agenda

- Tag 1
  - C# Grundlagen (Variablen, Konvertierungen, Arithmetik)
  - Konsolenanwendungen mit Ein-/Ausgabe
  - Arrays und Aufzählungstypen
  - Schleifen, Kontrollstrukturen, Enums
- Tag 2
  - Funktionen und Parameter
  - OOP: Klassen und Objekte
  - Vererbung
- Tag 3
  - Typen, Polymorphismus
  - Interfaces
  - Generische Datentypen
  - Linq und Lambda, Erweiterungmethoden
- Tag 4
  - Exception-Handling, Unit-Tests
  - Benutzeroberfläche gestalten
  - Datenzugriff
  - Sonstige Themen

Grundlagen Programmieren C#

Was ist Programmieren?

- **Definition:**
  Anweisungen schreiben, die ein Computer ausführen kann
- **Klare, schrittweise Anweisungen:**
  wie ein Kochrezept, oder eine Bauanleitung
- **Programmiersprachen:**
  viele Verschiedene, jede mit eigenen Stärken (C#, Python, Java, C++, …)



Was ist Quellcode?

- **Definition:**
  textbasierte Anweisungen, die ein Programm definieren
- **Von Menschen zu Computer:**
  Quellcode (Sourcecode) wird von Entwicklern geschrieben und vom Computer interpretiert

- **Beispiel:**
  Quellcode sagt dem Computer er soll den Text „Hello World“ ausgeben

```
Console.WriteLine("Hello World!");
```

Was ist die Konsole?

- textbasierte Schnittstelle zur **Eingabe** und **Ausgabe** von Daten
- Einfache **Kommunikation** zwischen Benutzer und Programm
- Verwendung in C#:
  `Console.[Funktion]`

Ergebnis in der Konsole

- Beispiel Konsolenausgabe:

```
Console.WriteLine("Hello World!");
```

Allgemeines zu C#

- Weiterentwicklung von C und C++
  - 1972: C
  - 1979: C++
  - 2001: C#
- Merkmale
  - Case-sensitive
  - Objektorientiert
  - Garbage-Collection
  - CLR - Common Language Runtime
  - JIT – Just in Time Compiler

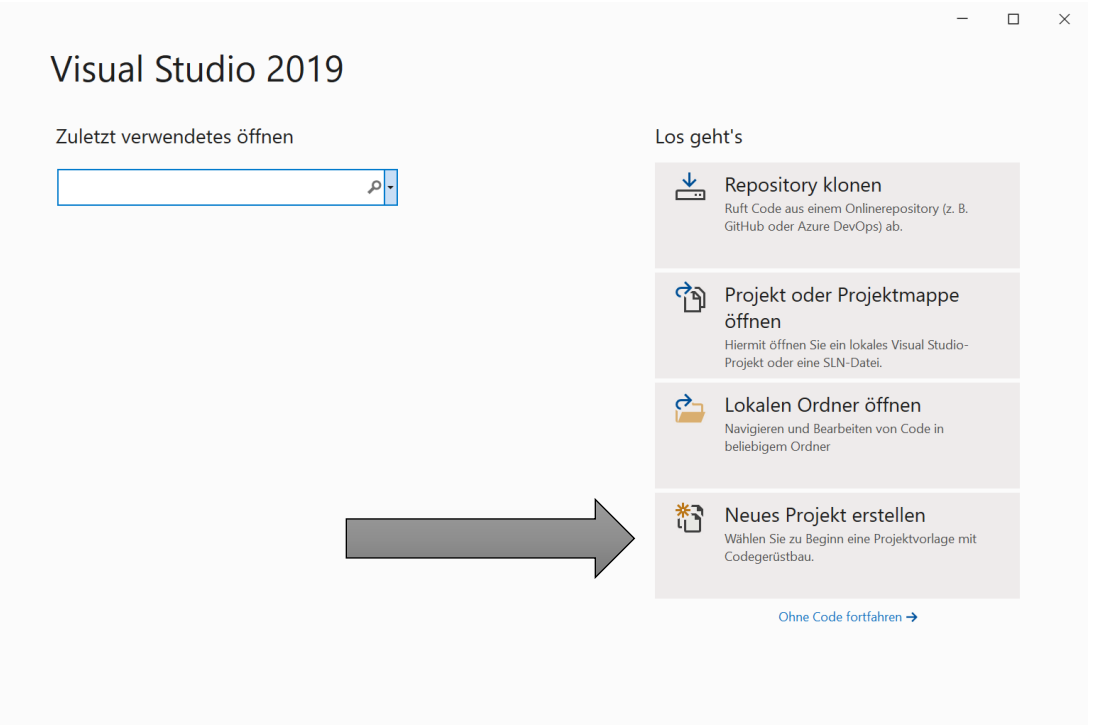
Allgemeines zu .NET

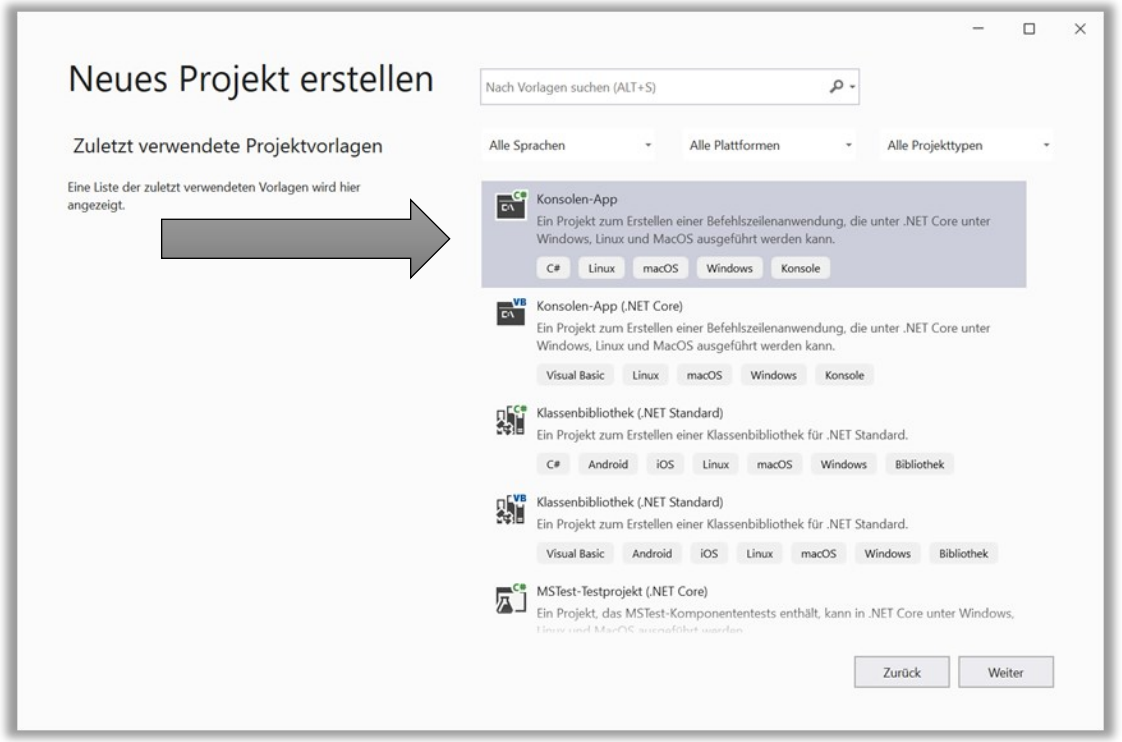
- Anwendung
  - .NET 8.0
    - Microsoft .NET (Windows Desktop)
    - .NET Core
  - UWP – Windows 10 Apps
  - XAMARIN/MAUI – iOS/Android
  - ASP.NET – Webserver
  - Spiele-Programmierung (Unity)
- Entwicklungsumgebungen
  - Visual Studio (aktuell Version 2022)
  - Visual Studio Code
  - Xamarin Studio
  - Unity

Einstieg in Visual Studio

Visual Studio 2019

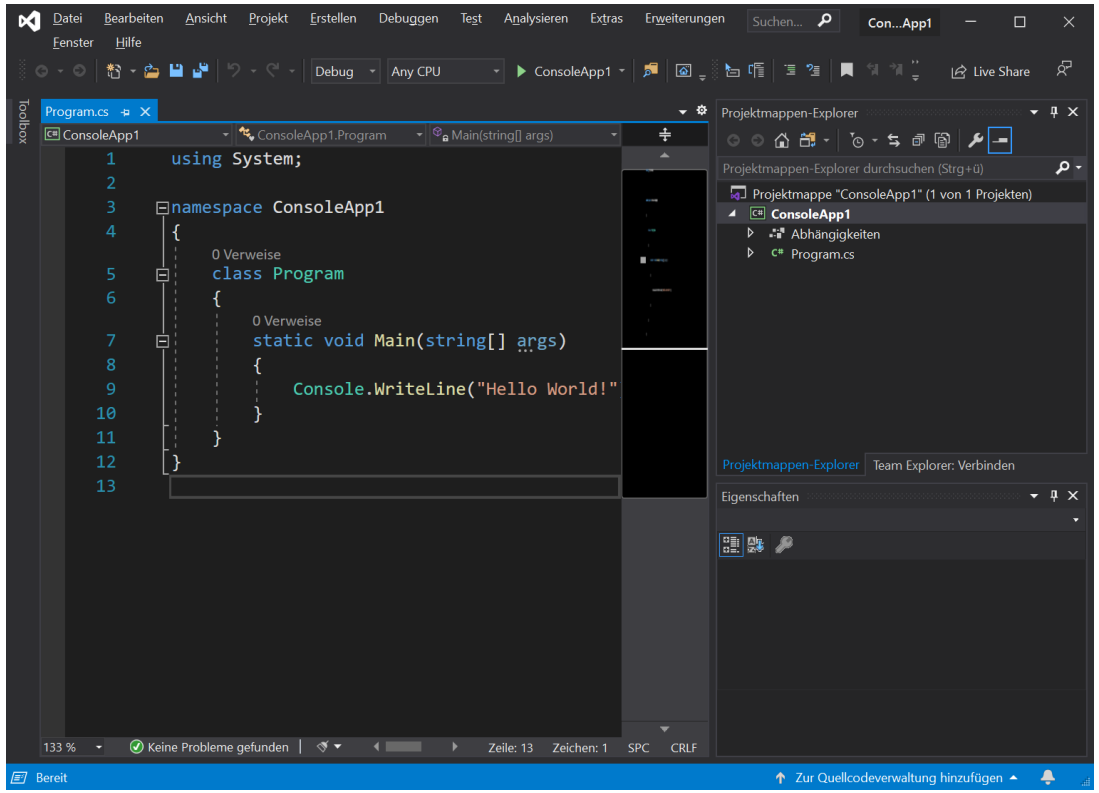
App





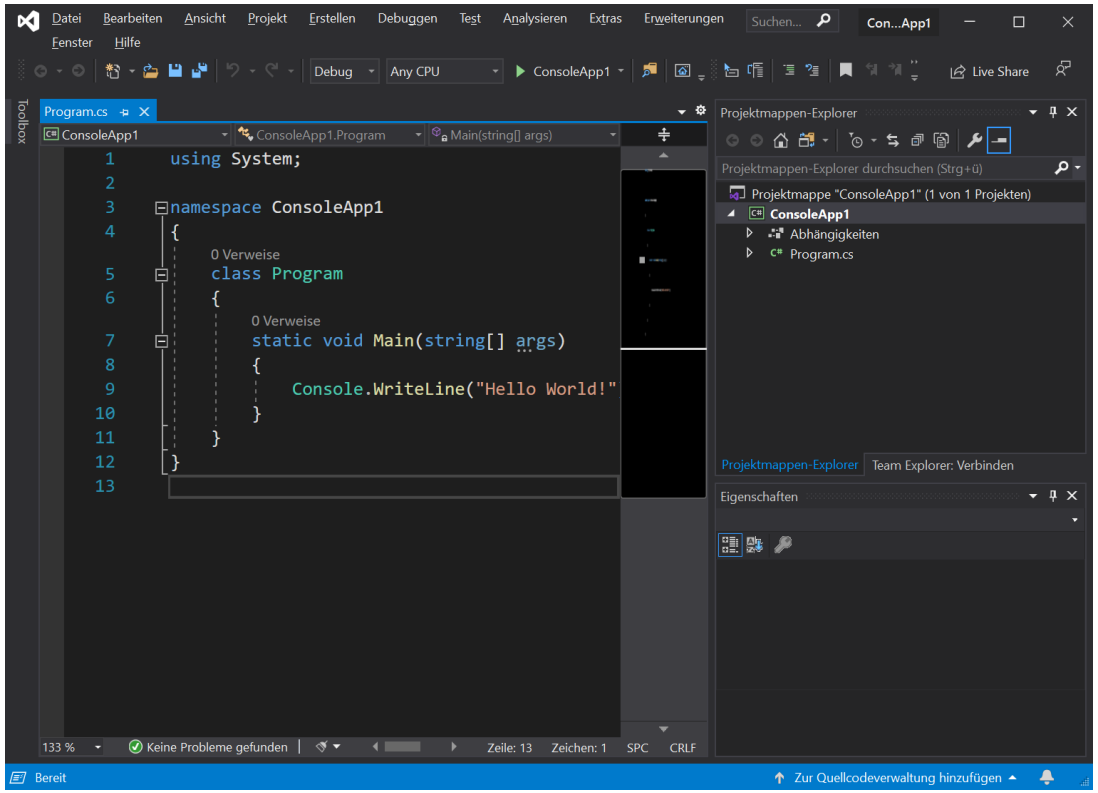
Einstieg in Visual Studio

- VS erstellt Projektmappen (eng: Solutions)
- Eine Projektmappe kann beliebig viele Projekte beinhalten
- Jedes Projekt ist in der Regel einem Applikations-‚Typ' zugeordnet (z.B. C#-Konsolenanwendung, Xamarin-Forms-Anwendung, etc.)



Lab: Erstes Programm in C#

- VS erstellt Projektmappen (eng: Solutions)
- Eine Projektmappe kann beliebig viele Projekte beinhalten
- Jedes Projekt ist in der Regel einem Applikations-‚Typ' zugeordnet (z.B. C#-Konsolenanwendung, Xamarin-Forms-Anwendung, etc.)



Variablen, Datentypen, Konvertierung und Operatoren

Sprachgrundlagen C#

- Befehle

```
Console.WriteLine("Hello World");
Console.ReadKey();
```

- Zuweisungen

```
int Nummer = 20;
string Wort = "Wort";
```

- Kommentare

```
int Nummer = 20; //Einzeiliger Kommentar

/* Hier steht nun
   Text und der
   soll kommentiert wurde. */

///Jede Zeile
///soll kommentiert werden
///wenn ich einen Umbruch mache
```

Variablen und Datentypen

- Variablen sind Behälter für Werte
- Deklaration und Benutzung von Variablen:

```
int Alter; //Deklaration
Alter = 20; //Integer mit Wert 20
int DoppeltesAlter = Alter * 2; //DoppeltesAlter wird berechnet
string Stadt = "Berlin"; //Zeichenkette "Berlin"
Console.WriteLine(Stadt); //"Berlin" in Konsole schreiben
```

Datentypen

Schlüsselwort	Nativer Typ	Beschreibung
char	System.Char	Einzelnes Textzeichen (Unicode-Zeichen 16 Bit)
string	System.String	Text (max. 4GB)
byte	System.Byte	8-Bit-Integer (0 bis +255)
short	System.Short	16-Bit-Integer (-32.768 bis +32.767)
int	System.Int32	32-Bit-Integer (-2.147.483.648 bis +2.147.483.647)
long	System.Int64	64-Bit-Integer (-9,22E19 bis +9,22E19)
float	System.Single	32-Bit-Gleitkommazahl (+/-1,5E-45 bis +/- 3,4E38)
double	System.Double	64-Bit-Gleitkommazahl (+/-5,0E-324 bis +/- 1,7E308)
decimal	System.Decimal	128-Bit-Gleitkommazahl (+/-1,0E-28 bis +/- 7,9E28)
bool	System.Boolean	Boolescher Wert (true oder false)

- Standardgemäß:
  - Ohne Nachkommastellen -> int
  - Mit Nachkommastellen -> double

Strings und Konsolenausgaben

Variablenverwendung:

```
int alterVonMax = 20;
string ausgabe;

// Stringverknüpfung
ausgabe = "Alter von Max: " + alterVonMax;
// $-Prefix
ausgabe = $"Max ist {alterVonMax} Jahre alt.";

Console.WriteLine(ausgabe);
```

Direktausgabe:

```
int alterVonMax = 20;


// Stringverknüpfung
Console.WriteLine("Alter von Max: " + alterVonMax);
// $-Prefix
Console.WriteLine($"Max ist {alterVonMax} Jahre alt.");
// Indexschreibweise
Console.WriteLine("Max ist {0} Jahre alt.", alterVonMax);
```

Konsolenausgabe:

```
Max ist 20 Jahre alt.
```

Formatierung von Strings

Escape-Sequenzen:

Einleitung durch Backslash: \

Beispiele:

\n	neue Zeile
\t	horizontaler Tabulator

```
Console.WriteLine($"Max ist \n{alterVonMax} Jahre \n\talt.");
```

Verbatim-String:

Einleitung durch @

```
Console.WriteLine(@"Max ist
{0} Jahre
        alt.", alterVonMax)
```

Konsolenausgabe (in beiden Fällen):

```
Max ist
20 Jahre
        alt.
```

Konsoleneingaben

Einlesen von Strings

```
string strEingabe = Console.ReadLine();
        //Konsoleneingabe wird in strEingabe gespeichert
int intEingabe = int.Parse(Console.ReadLine());
        //Konsoleneingabe wird (wenn möglich) als Integer in intEingabe gespeichert
double doubleEingabe = double.Parse(Console.ReadLine());
        //Konsoleneingabe wird (wenn möglich) als Double in doubleEingabe gespeichert
```

Einlesen von einzelnem Zeichen (Character)

```
Console.WriteLine("Zum Beenden beliebige Taste drücken: ");
Console.ReadKey();
```

Konvertierungen

- Zahl zu Zeichenkette

```
//Implizit
string satz = "Wert der Zahl: " + zahl;

//Explizit
string satz = zahl.ToString();
```

- Zeichenkette zu Zahl

```
//nur per Hilfsfunktion
int zahl = int.Parse("123");
double zahl2 = double.Parse("123.456");
```

tryParse() ist besser, weil es überprüft, ob der String konvertiert werden kann

- Zahl zu Zahl

```
int ganzzahl = 25;
double kommazahl = ganzzahl; //Implizit
ganzzahl = (int) kommazahl;  //Explizit
```

Mathematische Operatoren und Funktionen

- a + b Plus
- a – b Minus
- a * b Multiplikation
- a / b Division
- a % b Modulo (Rest der Division)
- a++/a-- a = a + 1 / a = a - 1
- a += 2 a = a + 2
- b /= 2 b = b / 2

- Math.Round(2.5)
  - Gerundete Zahl, bei .5 runde auf nächste gerade Zahl
- Math.Max(a,b)
  - Ermittle Maximum von beiden Zahlen
- Math.Min(a,b)
  - Ermittle Minimum von beiden Zahlen

Arrays und Bedingungen

Arrays

- Aufzählung gleichartiger Datentypen

```
int[] zahlen = new int[] { 2, 4, 6, 8, 10 }; //Länge = 5 / max. Index = 4
string[] buchstaben = new string[] { "A", "B", "C" }; //Länge = 3 / max. Index = 2
double[] kommazahlen = new double[10]; //Länge = 10 / max. Index = 9
```

- Zugriff erfolgt über nullbasierenden Index

```
zahlen[2]; // 6
buchstaben[1]; // "B"
buchstaben[2] = "Z"; // { "A", "B", "Z" }
```

Funktionen und Eigenschaften von Arrays

```
int[] zahlen = new int[] { 2, 9, 5 };
```

Funktionsname	Beschreibung	Beispiel
Length	Gibt die Anzahl der Elemente im Array zurück	zahlen.Length; //3
Contains(value)	Prüft ob Array ein Element enthält	zahlen.Contains(5); // true
Max()	Gibt den größten Wert im Array zurück	zahlen.Max(); //9
Min()	Gibt den niedrigsten Wert im Array zurück	zahlen.Min(); //2
First()	Gibt das erste Element im Array zurück	zahlen.First(); //2
Last()	Gibt das letzte Element im Array zurück	zahlen.Last(); //5
Sum()	Gibt die Summe aller Elemente zurück	zahlen.Sum(); //16

Mehrdimensionale Arrays

- Ein Array kann sich über mehrere Dimensionen erstrecken

```
//Deklaration eines mehrdimensionalen Arrays
int[,] zweiDimensionen;

//zweidimensionales Array mit zwei Zeilen und fuenf Spalten
zweiDimensionen = new int[,] { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9, 10 } };

//zweidimensionales Array mit vier Zeilen und drei Spalten
zweiDimensionen = new int[4, 3];

//Zugriff über den Index
int wert = zweiDimensionen[1, 2];

//Zuweisung über den Index
zweiDimensionen[2, 1] = 5;
```

Funktionen und Eigenschaften von mehrdimensionalen Arrays

```
int[,,] array3D;
array3D = new int[2, 3, 2] { { { 1, 2 }, { 3, 4 }, { 5, 6 } }, { { 7, 8 }, { 9, 10 }, { 11, 12 } } };
```

Funktionsname	Beschreibung	Beispiel
Length	Gibt die Anzahl aller Elemente im Array zurück	array3D.Length; // 12
Rank	Gibt die Anzahl der Dimensionen zurück	array3D.Rank; // 3

GetLength(int)	Gibt die Länge der jeweiligen Dimension zurück	array3D.GetLength(0);//2
		array3D.GetLength(1);//3
		array3D.GetLength(2);//2

Boolesche Logik

- Jede Operation ergibt entweder **true** oder **false**

```
a == b;      //a gleich b
a != b;      //a ungleich b
a < b;       //a kleiner b
a > b;       //a größer b
a <= b;      //a kleiner oder gleich b
a && b;      //a und b sind wahr
a || b;      //a oder b ist wahr
a ^ b;       //entweder a oder b
!a;          //true wird zu false, false zu true
```

Bedingungen

```
if (Bedingung A)
{
    //Anweisungen wenn Bedingung A wahr
}
else if (Bedingung B)
{
    //Anweisungen wenn Bedingung A falsch und B wahr
}
else
{
    //Anweisungen wenn Bedingung A und B falsch sind
}
```

Schleifen, Enums, Switch

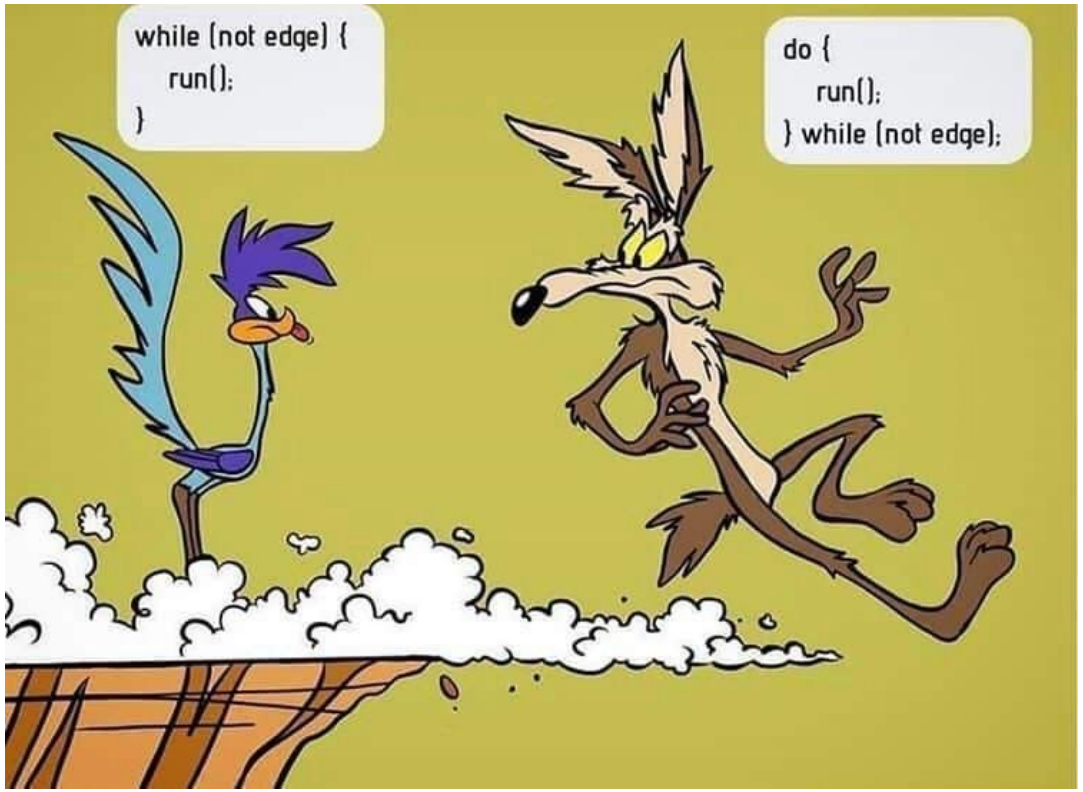
Bedingungsgesteuerte Schleifen

kopfgesteuerte Schleifen

```
while (Bedingung)
{
    //prüft vor jedem Durchlauf
    //ob Bedingung true ist
}
```

fußgesteuerte Schleifen

```
do
{
    //führt erst aus und prüft dann
    //beginnt von vorne wenn
    //Bedingung true ist
} while (Bedingung);
```



https://khalidabuhakmeh.com/coding-the-road-runner-while-loop-meme

Zählerschleife (for-Schleife)

```
for (Laufindex-Initialisierung; Laufindex-Bedingung; Laufindex-Anweisung)
{
    //Anweisung pro Durchgang
}
```

- inkrementale for-Schleife

```
int dauer = 10;
for (int i = 0; i<dauer; i++)
{

}
```

- dekrementale for-Schleife

```
int dauer = 10;
for (int i = dauer; i>0; i--)
{

}
```

Array mit ..

.. for-Schleife durchlaufen

```
for(int i=0;i<zahlen.Length;i++)
{
    Console.WriteLine(zahlen[i]);
}
```

.. foreach-Schleife durchlaufen

```
foreach(var item in zahlen)
{
    Console.WriteLine(item);
}
```

Enums

- eigener Datentyp mit fest vorgegebenen Zuständen
- jeder Zustand entspricht einem Integer

```
enum Wochentag { Mo = 1, Di = 2, ..., Sa = 6, So = 7 };

Wochentag tag = Wochentag.Mi;

if (tag == Wochentag.Mi) {
//...
}
```

switch-Anweisung

```
switch (tag)
{
    case Wochentag.Mo:
        Console.WriteLine("Wochenanfang");
        break;
    case Wochentag.Di:
    case Wochentag.Mi:
    case Wochentag.Do:
        Console.WriteLine("Normaler Wochentag");
        break;
    case Wochentag.Fr:
    case Wochentag.Sa:
    case Wochentag.So:
        Console.WriteLine("Wochenende");
        break;
    default:
        Console.WriteLine("Da ist wohl etwas schiefgelaufen...");
        break;
}
```

```
switch (obst)
{
    case "Orange":
    case "Mandariene":
        farbe = "Orange";
        break;
    case "Zitrone":
    case "Banane":
        farbe = "Gelb";
        break;
    case "Kirsche":
        farbe = "Rot";
        break;
    case string o when o.Length = 0:
        farbe = "*fehlerhafte Eingabe*";
    default:
        farbe = "unbekannt";
        break;
}
```

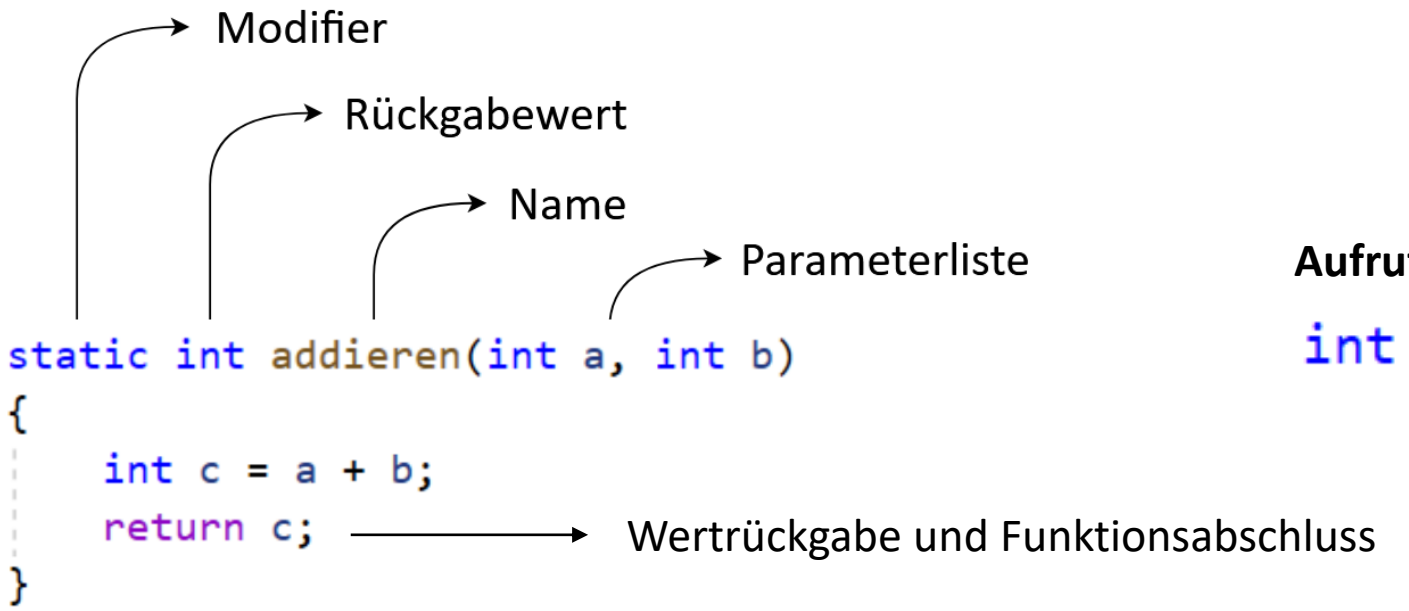
Funktionen

Methoden (Funktionen)

```
static void Main(string[] args)
{

    int zahl1 = 1;
    int zahl2 = 2;
    int summe = zahl1 + zahl2;
    Console.WriteLine($"Die Summe von {zahl1} und {zahl2} ist {summe}.");

    zahl1 = 5;
    zahl2 = 9;
    summe = zahl1 + zahl2;
    Console.WriteLine($"Die Summe von {zahl1} und {zahl2} ist {summe}.");

    zahl1 = 2;
    zahl2 = 7;
    summe = zahl1 + zahl2;
    Console.WriteLine($"Die Summe von {zahl1} und {zahl2} ist {summe}.");

    Console.ReadKey();
}
```

```
static void Main(string[] args)
{
    SummeAusgeben(1, 2);
    SummeAusgeben(5, 9);
    SummeAusgeben(2, 7);
    Console.ReadKey();
}

3 Verweise
static void SummeAusgeben(int zahl1, int zahl2)
{
    int summe = zahl1 + zahl2;
    Console.WriteLine($"Die Summe von {zahl1} und {zahl2} ist {summe}.");
}
```

Methoden (Funktionen)



Aufruf:

```
int summe = addieren(1, 2);
```

Methoden (Funktionen)

- Modifikatoren (Modifiers) definieren die Zugriffsmöglichkeiten auf oder den Zustand der Funktion

  z.B.: `static` `public` `private`

- Der Rückgabewert gibt den Datentyp der Rückgabe an

  z.B.: `int` `Random` `void`

- Über den Bezeichner wird die Funktion aufgerufen

- Die Parameterliste gibt die benötigten Übergabeparameter mit Datentyp und Bezeichner an

Überladung von Methoden

- ein Funktionsname kann in derselben Klasse mehrfach benutzt werden
- jede Funktion muss eine einmalige Parameterliste haben
- die Rückgabewerte können unterschiedlich sein

```
int addieren(int a, int b)
{
    int summe = a + b;
    return summe;
}
```

```
int addieren(int a, int b, int c)
{
    int summe = a + b + c;
    return summe;
}
```

Spezielle Parametertypen

- Automatische Arrays: Aus beliebig vielen Übergaben eines Typs wird automatisch ein Array erstellt

```
public static int BildeSumme(params int[] summanden) {…}
```

Aufrufbeispiele: `BildeSumme(2,5,7);` `BildeSumme(1,2,3,4,5,6,7)`

- Optionale Parameter: Ein Parameter erhält eine Vorbelegung, welche überschrieben werden kann

```
public static int Addiere(int a, int b, int c = 0) {…}
```

Aufrufbeispiele: `Addiere(2,5);` `Addiere(1,2,3)`

Spezielle Parametertypen - out

- eine Funktion kann immer nur einen Rückgabewert haben
- mittels out kann mehr als ein Wert zurückgegeben werden
- Parameter wird in der Signatur deklariert und muss in der Funktion initialisiert werden
- der Aufruf erfolgt ebenfalls mit dem Schlüsselwort

```
int berechne(int zahl1, int zahl2, out int min, out int max, out int avg)
{
    int summe = zahl1 + zahl2;

    if (zahl1 < zahl2)
    {
        min = zahl1;
        max = zahl2;
    }
    else
    {
        min = zahl2;
        max = zahl1;
    }
    avg = summe / 2;

    return summe;
}
```

```
int min;
int max;
int avg;
int summe = berechne(3, 5, out min, out max, out avg);

//summe = 8
//min = 3
//max = 5
//avg = 4
```

Objektorientierte Programmierung OOP

Objektorientierte Programmierung

- Merkmale:
  - Unterteilung des Codes in funktionelle Einheiten (Klassen)
  - Erstellung (Instanziierung) unabhängiger Objekte aus den Klassen
  - Referenzierung abhängiger Objekte innerhalb von Variablen

- Vorteile:
  - Code wiederverwendbar
  - Bessere Strukturierung und Lesbarkeit
  - erhöhte Sicherheit durch Zugriffsmodifizierer

Namespace und Klassen

Namespace

- dient zur Organisation von Klassen und Methoden
- können geschachtelt werden
- über using-Anweisungen vereinfachter Zugriff auf Code

Klasse

- beschreiben die Struktur von Objekten
- von einer Klasse können mehrere Objekte erzeugt werden
- bestehen aus
  - Feldern(Membervariablen)
  - Eigenschaften (Properties)
  - Funktionen (Methoden)
  - Konstruktor/Destruktor

Namespace und Klassen

```
namespace MeinNamespace
{
    class MeineKlasse
    {
        private int feld;

        public int Eigenschaft { get; set; }

        public void Methode()
        {
        }

        public MeineKlasse()
        {
            feld = 0;
        }

        ~MeineKlasse()
        {
        }
    }
}
```

Felder (Fields)

- Felder sind Variablen eines beliebigen Typs innerhalb einer Klasse
- die Felder sollten generell nur privat und nicht von extern verfügbar sein
- der Zugriff auf Felder sollte über „Get“- und „Set“-Methoden erfolgen
- „speichern“ in der Regel die Daten auf die zugegriffen werden soll

```
public class Person
{
    private string vorname;

    public void SetVorname(string vorname)
    {
        this.vorname = vorname;
    }

    public string GetVorname()
    {
        return this.vorname;
    }
}
```

Eigenschaften (Properties)

- vereinfachen das Anlegen der „Get“- und „Set“-Methoden

```
private string vorname;

public void SetVorname(string vorname)
{
    this.vorname = vorname;
}

public string GetVorname()
{
    return this.vorname;
}
```

```
public string Vorname { get; set; }
```

Eigenschaften (Properties)

```
//öffentliches Lesen und Schreiben erlauben
public string Vorname { get; set; }

//Variable kann nur noch intern gesetzt werden
public string Nachname { get; private set; }

//Variable kann von Außen nur überschrieben werden
public string Geheim { private get; set; }
```

```
private int alter;
0 Verweise
public int Alter
{
    get
    {
        return alter;
    }
    set
    {
        if (value > 0)
        {
            alter = value;
        }
    }
}
```

Standardwerte

- Standardwert einem Feld zuweisen

```
class MeineKlasse
{
    private string meinFeld = "Ich habe einen Standardwert!";
    public string MeineEigenschaft
    {
        get { return meinFeld; }
        set { meinFeld = value; }
    }
}
```

Standardwerte

- Standardwert einer Auto-Eigenschaft zuweisen

```
class MeineKlasse
{
    string MeineEigenschaft { get; set; } = "Ich habe einen Standardwert!";

    public MeineKlasse()
    {
    }
}
```

- Standardwert bei Objekterstellung zuweisen

```
class MeineKlasse
{
    string MeineEigenschaft { get; set; }

    public MeineKlasse()
    {
        MeineEigenschaft = "Ich habe einen Standardwert!";
    }
}
```

Konstruktor

- legt den Startzustand des Objektes nach der Initialisierung fest
- eine Klasse kann mehrere Konstruktoren haben (unterschiedliche Parameter)
- wenn kein Konstruktor festgelegt wurde, wird automatisch ein parameterloser Standard-Konstruktor erzeugt (bspw. „Person()“)

```
public Person(string vorname, string nachname)
{
    this.Vorname = vorname;
    this.Nachname = nachname;
}

public Person(string vorname, string nachname, DateTime geburtstag) : this(vorname, nachname)
{
    this.Geburtstag = geburtstag;
}
```

Objekte (Instanzen)

- ein Objekt ist ein Speicherblock, der nach dem Entwurf einer Klasse aufgebaut wird

- es kann mehrere Objekte einer Klasse geben

- Objekte werden mit dem Schlüsselwort „new“ und einer Funktion des Klassennamens (Konstruktor) erzeugt

```
MeineKlasse meinObjekt = new MeineKlasse();
```

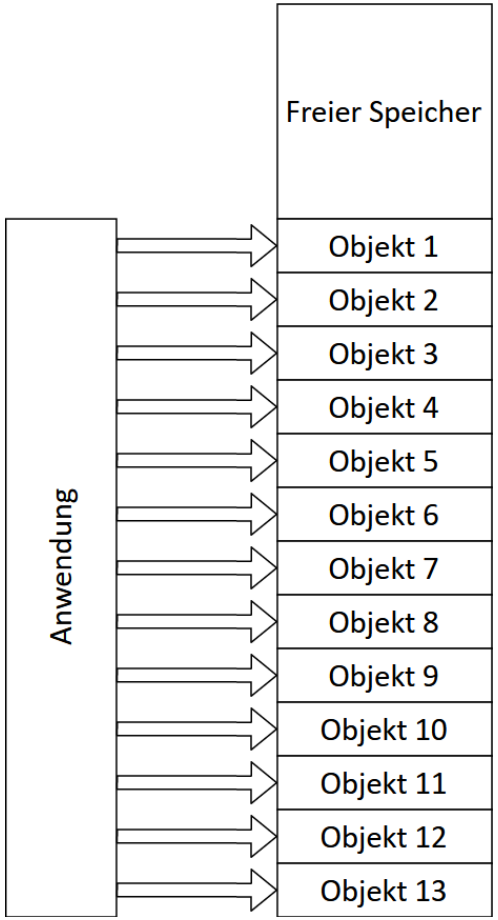
Garbage Collection, Statische Member, Datumswerte Werte und Referenzen, Null

Garbage Collection

- verwaltet die Belegung und Freigabe von Arbeitsspeicher
- überprüft ob Objekte noch verwendet werden
- beginnt diese zu „zerstören“ um den Speicher freizugeben
- läuft komplett eigenständig im Hintergrund

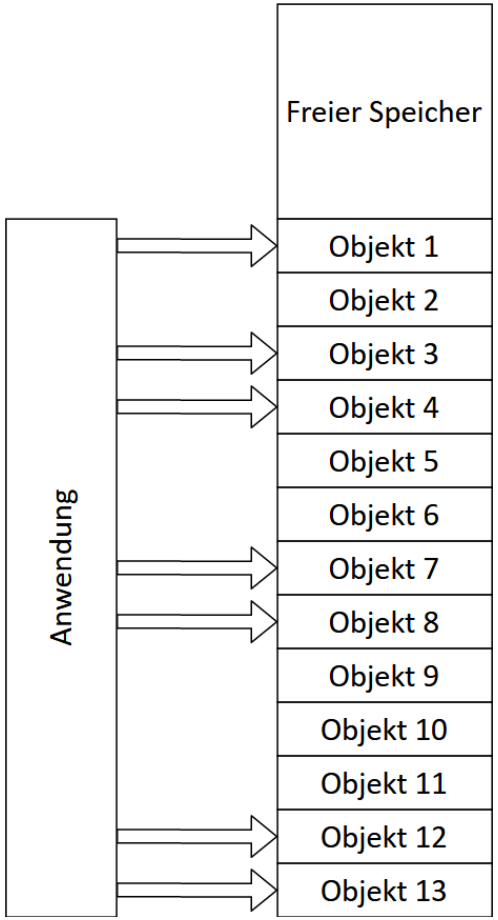
- kümmert sich nur um verwaltete Ressourcen (!)

Garbage Collection



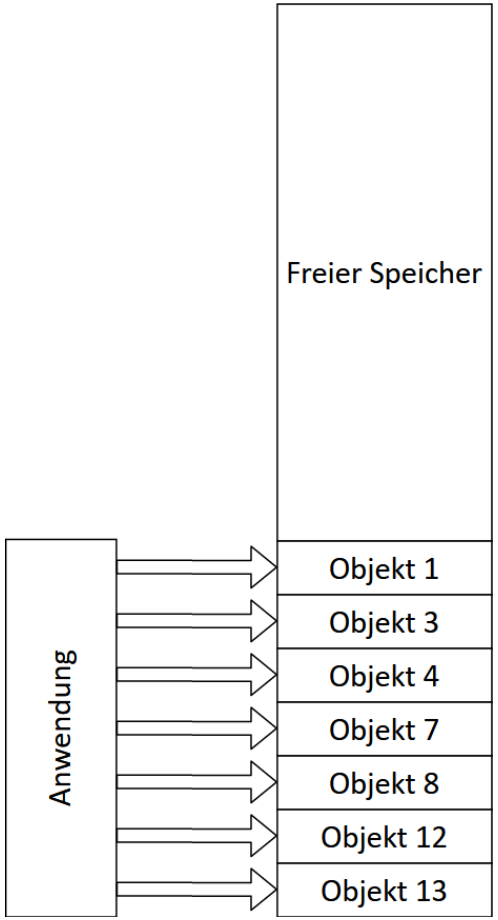
- die Anwendung hat mehrere Objekte im Speicher, welche alle noch benötigt werden

Garbage Collection



- die Anwendung benötigt einige Objekte nicht mehr, diese sind aber noch im Speicher vorhanden

Garbage Collection



- die GC filtert genau diese Objekte heraus und „zerstört“ sie, sodass der Speicher wieder freigegeben wird

Destruktor (Finalizer)

- wird aufgerufen wenn ein Objekt vom Garbage Collector zerstört wird
- kann nur einmal pro Klasse festgelegt werden
- Funktionsname besteht aus einer Tilde (~) und dem Klassennamen
- kann selber Anweisungen ausführen

```
~Person()
{
    Console.WriteLine("Destruktoraufruf: das Objekt wird nun zerstört");
}
```

Statische Member

- gelten für die Klasse und nicht für ein Objekt dieser Klasse
- Zugriff erfolgt über den Klassenname
- statische Member können nicht auf nicht-statische Member der Klasse zugreifen
- Jede Eigenschaft (Property) existiert nur ein Mal für die Klasse („systemweit“)
- klassische Beispiele

```
Console.WriteLine("Text");

string eingabe = Console.ReadLine();

DateTime heute = DateTime.Now;
```

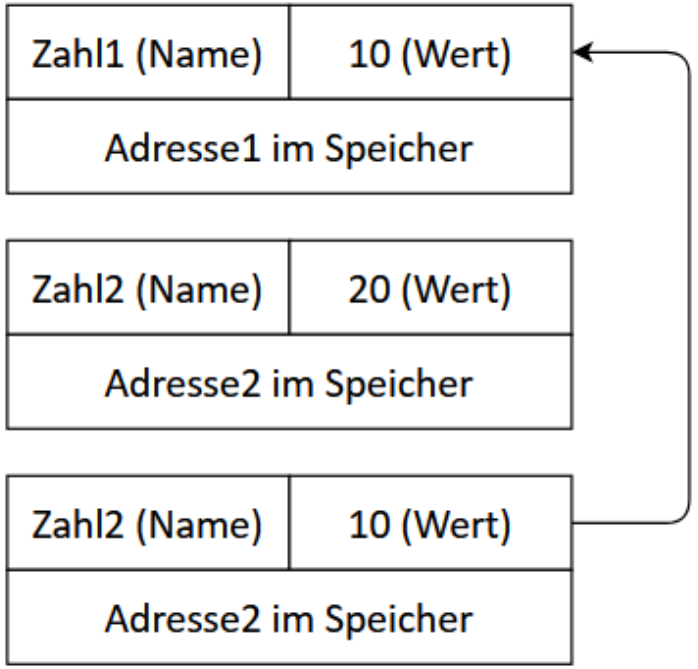
Wertetypen und Referenztypen

- Wertetypen
  - werden mit einer festen Adresse im Speicher hinterlegt
  - der Wert wird beim Ändern an genau der Stelle im Speicher geändert
- Referenztypen
  - werden einmalig im Speicher hinterlegt
  - bei Änderung bekommen sie eine neue Adresse
  - können einer bereits genutzten Adresse zugewiesen werden

Wertetypen und Referenztypen

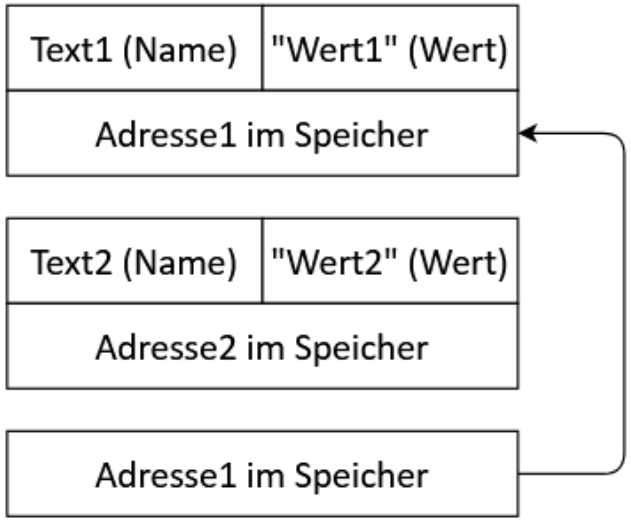
Wertetypen

```
int Zahl1 = 10;
int Zahl2 = 20;
Zahl2 = Zahl1;
```



Referenztypen

```
string Text1 = "Wert1";
string Text2 = "Wert2";
Text2 = Text1;
```



Schlüsselwort - ref

- gleicher Hintergrund wie out
- Parameter muss zuvor initialisiert werden
- Wert des Parameters kann direkt verwendet werden
- Aufruf ebenfalls mit Schlüsselwort

Schlüsselwort - ref

```
int addiere(int zahl1, int zahl2, ref int anzahlAdditionen)
{
    int summe = zahl1 + zahl2;
    anzahlAdditionen += 1;

    return summe;
}

int anzahl = 0;
int ergebnis = 0;
ergebnis = addiere(ergebnis, 3, ref anzahl); //ergebnis = 3 anzahl = 1
ergebnis = addiere(ergebnis, 8, ref anzahl); //ergebnis = 11 anzahl = 2
ergebnis = addiere(ergebnis, 45, ref anzahl); //ergebnis = 56 anzahl = 3
```

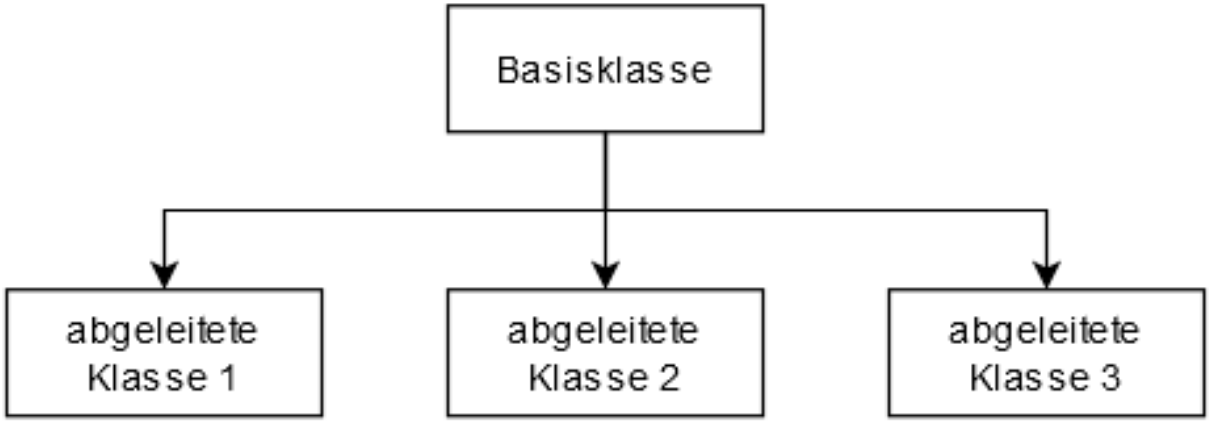
Null

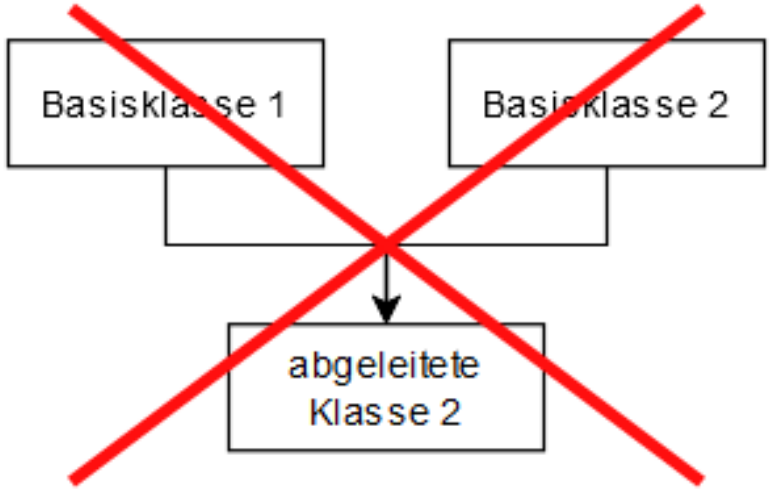
- Standardwert wenn eine Variable keinen Wert hat
- Ausnahmen: int, double, bool, …
- Mit einer if überprüfen

Vererbung und Zugriffsmodifizierer

Vererbung (Ableitung)

- Klassen können voneinander erben
- übernehmen Eigenschaften und Methoden der Basisklasse*
- Es kann an beliebig viele Klassen vererbt werden
- Es kann nur von jeweils einer Klasse geerbt werden





*nicht alle Member werden übernommen(bspw. kein Konstruktor/Destruktor)

Konstruktoren in abgeleiteten Klassen

- beim Initialisieren einer neuen Instanz wird immer erst der Konstruktor der Basisklasse ausgeführt
- wenn der Konstruktor der Basisklasse Parameter enthält, muss dieser explizit mit „: base(*Parameter*)“ angegeben werden

```
public class Lebewesen
{
    1-Verweis
    public int Alter { get; set; }
    1-Verweis
    public Lebewesen(int alter)
    {
        this.Alter = alter;
    }
}

1-Verweis
public class Mensch : Lebewesen
{
    1-Verweis
    public string Wohnort { get; set; }

    0 Verweise
    public Mensch(int alter, string wohnort) : base(alter)
    {
        this.Wohnort = wohnort;
    }
}
```

virtuelle Member

- verhalten sich wie normale Member und können auch so aufgerufen werden
- können von einer abgeleiteten Klasse mit dem Schlüsselwort „override“ überschrieben werden
- mit Schlüsselwort „base“ kann auf den Basisklassenmember zugegriffen werden

```
public class Lebewesen
{
    public virtual string WasBinIch()
    {
        return "Ich bin ein Lebewesen";
    }
}

public class Mensch : Lebewesen
{
    public override string WasBinIch()
    {
        string wasBinIch = base.WasBinIch();
        wasBinIch += " auf zwei Beinen und kann sprechen";
        return wasBinIch;
    }
}

meinMensch.WasBinIch();
//"Ich bin ein Lebewesen auf zwei Beinen und kann sprechen"
```

Überschreibung verhindern

- man kann durchgehend virtuelle Member überschreiben, auch wenn eine Klasse von einer schon abgeleiteten Klasse abgeleitet ist
- dies kann man verbinden indem man das Schlüsselwort „sealed“ verwendet und somit den neu ableitenden Klassen das Überschreiben unterbindet

```
public class Lebewesen
{
    public virtual string WasBinIch()
    {
        return "Ich bin ein Lebewesen";
    }
}

public class Mensch : Lebewesen
{
    public sealed override string WasBinIch()
    {
        return "Ich bin ein Mensch";
    }
}

public class Kind : Mensch
{
    //kann nicht überschrieben werden
    public override string WasBinIch()
    {
        return "Ich bin ein Kind";
    }
}
```

Zugriffsmodifizierer (Modifier)

- können für Felder, Eigenschaften, Methoden und ganze Klassen verwendet werden

Modifier	Zugriff
public	von außerhalb der Klasse
private	nur innerhalb der Klasse
protected	innerhalb der Klasse und in allen abgeleiteten Klassen
internal	nur die aktuellen Assembly
protected internal	nur die aktuellen Assembly oder abgeleitete Klasse
private protected	nur abgeleitete Klassen innerhalb der Assembly

Polymorphismus, Typen

Polymorphismus

- ist die „dritte Säule der OOP“ (Kapslung, Vererbung, Polymorphismus)
- griechisch für „Vielgestaltigkeit“

Objekte abgeleiteter Klasse können immer als Objekte der Mutterklassen betrachtet werden

```
Lebewesen menschAlsLebewesen = new Mensch();
```

Variablentyp vs. Laufzeittyp

```
Mensch mensch = new Mensch()
Variablentyp variablenbezeichner = new Laufzeittyp()
```

- Der Variablentyp bestimmt den Typ einer Variablen und dadurch deren möglichen Inhalt
- Der Laufzeittyp bezeichnet den Typ eines Objekts und kann über die Methode `.GetType()`erfragt werden

Typtest durch GetType und typeof-Operator

- durch „.GetType()“ wird der Laufzeittyp eines Objektes ermittelt
- der typeof Operator ruft die System.Type-Instanz für einen Typ ab

```
Lebewesen menschAlsLebewesen = new Mensch();

if (menschAlsLebewesen.GetType() == typeof(Lebewesen))
//false

if (menschAlsLebewesen.GetType() == typeof(Mensch))
//true
```

Typtest durch is-Operator

- der is-Operator prüft ob der Laufzeittyp mit einem angegebenen Typ kompatibel ist
- gibt „true“ zurück, wenn das Objekt auch als der erfragte Typ betrachtet werden kann / wenn eine Ableitung existiert

```
Lebewesen lebewesen = new Lebewesen();
Mensch mensch = new Mensch();

if (lebewesen is Lebewesen) //true
if (lebewesen is Mensch) //false
if (mensch is Lebewesen) //true
if (mensch is Mensch) //true
if (mensch is object) //true
```

virtuelle Member

- es wird immer auf den Member der jeweiligen Instanz zugegriffen
- die Instanz kann auch in die Basisklasse gecastet werden, der Aufruf geschieht dennoch auf die eigentliche Instanz
- in Basisklasse casten ergibt Sinn, wenn mehrere Objekte unterschiedlicher abgeleiteter Klassen verarbeiten werden

```
Lebewesen lebewesen = new Lebewesen();
lebewesen.WasBinIch();
//"Ich bin ein Lebewesen"

Mensch mensch = new Mensch();
mensch.WasBinIch();
//"Ich bin ein Lebewesen auf zwei Beinen und kann sprechen"

Lebewesen menschAlsLebewesen = (Lebewesen)mensch;
menschAlsLebewesen.WasBinIch();
//"Ich bin ein Lebewesen auf zwei Beinen und kann sprechen"
```

Ausblenden von Basisklassenmembern

- Member aus der Basisklasse können auch ausgeblendet werden
- das Schlüsselwort „new“ vor den Rückgabewert setzen und den gleichen Membernamen verwenden
- wenn eine Instanz in die Basisklasse gecastet wird, kann diese nun nicht mehr auf das neue Member zugreifen

```
public class Lebewesen
{
    public virtual string WasBinIch()
    {
        return "Ich bin ein Lebewesen";
    }
}

public class Mensch : Lebewesen
{
    public new string WasBinIch()
    {
        return "Ich bin ein Mensch";

    }
}
Mensch mensch = new Mensch();
Lebewesen menschAlsLebewesen = (Lebewesen)mensch;

mensch.WasBinIch(); //"Ich bin ein Mensch"
menschAlsLebewesen.WasBinIch(); //"Ich bin ein Lebewesen"
```

abstrakte Member und Klassen

- sind nur Signaturen und geben den Aufbau vor, können aber nicht selber initialisiert werden
- müssen von abgeleiteten Klassen überschrieben werden
- wenn ein Member abstrakt sein soll, muss die Klasse als abstrakt markiert werden

```
public abstract class Fahrzeug
{
    public abstract bool IstFahrbereit();
}

public class Fahrrad : Fahrzeug
{
    public int Luft { get; set; }

    public override bool IstFahrbereit()
    {
        return (Luft > 0);
    }
}
```

abstrakte Member und Klassen

- können nicht „sealed“ verwenden, da sie sich gegenseitig ausschließen
- eine abgeleitete Klasse muss alle abstrakten Member implementieren
- abstrakte Methoden haben keinen Körper sondern werden nur deklariert

Interfaces

Interface (Schnittstelle)

- beschreiben **Eigenschaften**, die erbende Klassen haben müssen
- gibt nur die Struktur von Eigenschaften und Methoden an
  → es definiert Member, ohne diese zu implementieren
- eine Klasse kann von einem oder mehreren Interfaces erben und muss alle Member implementieren
- es kann **kein Objekt** von einem Interface erstellt werden

- alle Member sind standardmäßig public und abstract
- Zugriffsmodifizierer können nicht explizit benutzt werden (kein private, protected oder internal)
- können keine Felder definieren (enthält keine Speicherung von Daten)

Interfaces beschreiben meistens Eigenschaften zum Beispiel Objekt ist „fahrbar“

```
// Schnittstelle IDriveable (etwas ist fahrbar)
interface IDriveable
{
    // Fahrbare Objekte müssen die Eigenschaften Name und Geschwindigkeit haben
    // Fahrbare Objekte können beschleunigen und bremsen

    // Definition einer Eigenschaft Name (lesbar & veränderbar)
    string Name { get; set; }

    // Definition einer Eigenschaft Geschwindigkeit (nur lesbar)
    double Speed { get; }
    // Definition einer Methode zum Beschleunigen
    void Accelerate(double deltaSpeed);

    // Definition einer Methode zum Bremsen
    void Brake(double deltaSpeed);
}
```

Klassen können auch mehrere Interfaces implementieren.

```
// Klasse Car implementiert Schnittstellen IDriveable (Auto ist fahrbar) und ILightable (Auto kann leuchten)
class Car : IDriveable , ILightable
{
    // Eigenschaften Name und Speed von IDriveable
    public string Name { get; set; }
    public double Speed { get; private set; } = 0; // nur veränderbar in der Car Klasse; Standardwert 0

    // Eigenschaft IsLightsOn von ILightable
    public bool IsLightsOn { get; set; }

    // Methode Accelerate und Break von IDriveable müssen implementiert werden
    public void Accelerate(double deltaSpeed)
    {
        Speed += deltaSpeed;
    }
    public void Brake(double deltaSpeed)
    {
        Speed -= deltaSpeed;
    }
}
```

Methoden und statische Properties

- Statische Member sind nur über das Interface aufrufbar
- Wenn sie eine Standartimplementierung besitzen dürfen überschrieben werden

```
interface IDriveable
{
    string Name { get; set; }
    double Speed { get; }

    // Statische Konstante für die maximale Geschwindigkeit
    static double MaxSpeed { get; set; } = 300;

    void Accelerate(double deltaSpeed);
    void Brake(double deltaSpeed);
}
```

```
class Car : IDriveable , ILightable
{
    public string Name { get; set; }
    public double Speed { get; private set; } = 0;
    public bool AreLightsOn { get; set; }

    // Methode Accelerate und Break von IDriveable
    public void Accelerate(double deltaSpeed)
    {
        // Statische Konstante von IDriveable wird verwendet
        if (Speed + deltaSpeed <= IDriveable.MaxSpeed)
        {
            Speed += deltaSpeed;
        } else
        {
            Speed = IDriveable.MaxSpeed;
        }
    }
    public void Brake(double deltaSpeed)
    {
        Speed -= deltaSpeed;
    }
}
```

implizierte Implementierung

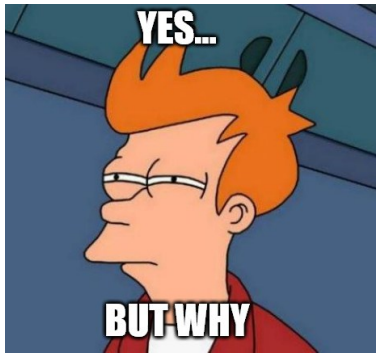
- Zugriff auf Member direkt über den Objekttyp

```
// Klasse Program
class Program
{
    // Main Methode
    static void Main(string[] args)
    {
        // Neues Car-Objekt erstellen
        Car car = new Car();
        // Zugriff auf Accelerate-Methode von IDriveable über Car-Objekt
        car.Accelerate(10);
    }
}
```

explizierte Implementierung

- die Member können keine Zugriffsmodifizierer verwenden
- Zugriff nur über Konvertierung in den Typ des Interfaces

```
class Car : IDriveable
{
    public string Name { get; set; }
    public double Speed { get; private set; } = 0;

    // Explizite Implementierung von IDriveable.Accelerate
    void IDriveable.Accelerate(double deltaSpeed)
    {
        Speed += deltaSpeed;
    }
}

Car car = new Car();
// Konvertierung von Car zu IDriveable
IDriveable driveableCar = (IDriveable) car;
driveableCar.Accelerate(10);
```



explizierte Implementierung: Beispiel FlyingCar

```
// (etwas ist fahrbar)
interface IDriveable
{
    string Name { get; set; }
    double Speed { get; }
    void Accelerate(double deltaSpeed);
    void Brake(double deltaSpeed);
}
```

```
// (etwas ist flugfähig)
interface IFlyable
{
    string Name { get; set; }
    double Altitude { get; }
    void Accelerate(double deltaSpeed);
    void Brake(double deltaSpeed);
}
```

explizierte Implementierung: Beispiel FlyingCar

```
// Klasse FlyingCar implementiert beide Schnittstellen
class FlyingCar : IDriveable, IFlyable
{
    public string Name { get; set; }
    public double Speed { get; private set; }
    public double Altitude { get; private set; }

    // Explizite Implementierung von IDriveable.Accelerate
    void IDriveable.Accelerate(double deltaSpeed)
    {
        Speed += deltaSpeed;
    }

    // Explizite Implementierung von IDriveable.Brake
    void IDriveable.Brake(double deltaSpeed)
    {
        Speed -= deltaSpeed;
    }

    // Explizite Implementierung von IFlyable.Accelerate
    void IFlyable.Accelerate(double deltaSpeed)
    {
        Altitude += deltaSpeed;
    }

    // Explizite Implementierung von IFlyable.Brake
    void IFlyable.Brake(double deltaSpeed)
    {
        Altitude -= deltaSpeed;
    }
}
```

```
static void Main()
{
    // Neues Objekt fylingCar
    FlyingCar flyingCar = new FlyingCar();

    // Casten von flyingCar zu IDriveable und IFlyable
    IDriveable driveable =  (IDriveable) flyingCar;
    IFlyable flyable = (IFlyable) flyingCar;

    // die gleichnamigen Methoden können jetzt aufgerufen werden
    driveable.Accelerate(100);
    flyable.Accelerate(100);
}
```

Fehlerbehandlung und UnitTests

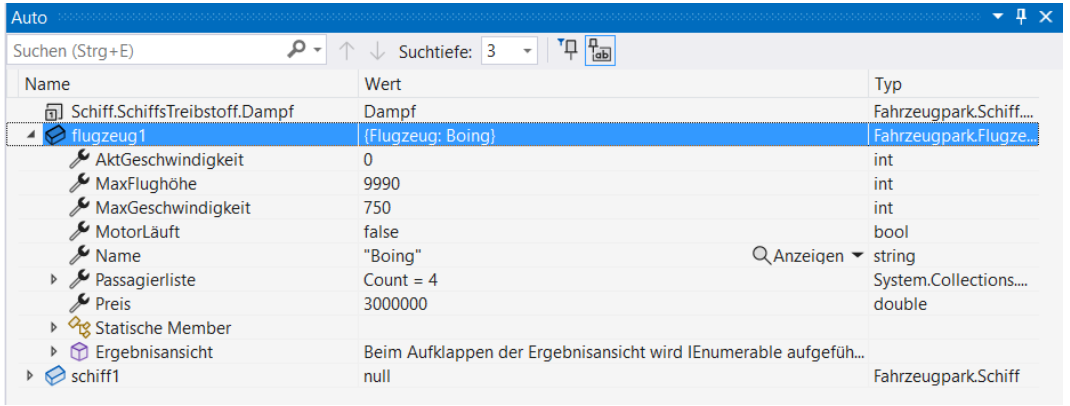
Debugging

- Haltepunkte pausieren die Ausführung BEVOR die gewählte Zeile ausgeführt wird

```
114
115     Flugzeug flugzeug1 = new Flugzeug("Boing", 750, 3000000, 9990);
116     Schiff schiff1 = new Schiff("Titanic", 40, 3500000, Schiff.SchiffsTreibstoff.Dampf);
117
118     BeladeFahrzeuge(pkw1, flugzeug1);
```

  - Einzelschritt (F11): Schrittweise Ausführung mit Sprung in Methodendefinitionen
  - Prozedurschritt (F10): Schrittweise Ausführung auf aktueller Ebene

- Auto- und Lokalfenster beinhalten aktuelle Variablenbelegungen



Fehlerbehandlung

- mit einem try-catch-Block können Fehler abgefangen und behandelt werden
- try-Block
  - eigentliche Anwendung
- catch-Block
  - Fehler werden abgefangen und behandelt
- finally-Block
  - ist optional
  - wird garantiert ausgeführt

```
try
{
    int Eingabe = int.Parse(Console.ReadLine());
}
catch (FormatException e)
{
    Console.WriteLine("Bitte eine Zahl eingeben.");
    return;
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
    return;
}
finally
{
    Console.WriteLine("Danke");
}
```

eigene Exception erstellen

- jede neue Exception ist von der Klasse „Exception“ abgeleitet
- durch „catch (Exception)“ werden somit alle nicht zuvor abgefangenen Fehlermeldungen behandelt

```
public class MyException : Exception
{
    public MyException(string message) : base(message)
    {

    }
}
```

```
try
{
    //einen Fehler werfen
    throw new MyException("Eine MyException wurde geworfen!");
}
catch (MyException e) //über ein catch-Block den Fehler abfangen
{
    Console.WriteLine(e.Message);
    //Eine MyException wurde geworfen!
}
catch (Exception e) //alle anderen Exceptinos werden abgefangen
{
    Console.WriteLine(e.Message);
}
finally
{
    Console.WriteLine("try-catch-Block ist fertig..");
}
```

UnitTest - Komponententest

- mit einem UnitTest können Funktionen auf eine definierte Eingabe und eine erwartete Rückgabe getestet werden

- ein extra Projekt, welches in VisualStudio erstellt werden kann

Testklasse stellen

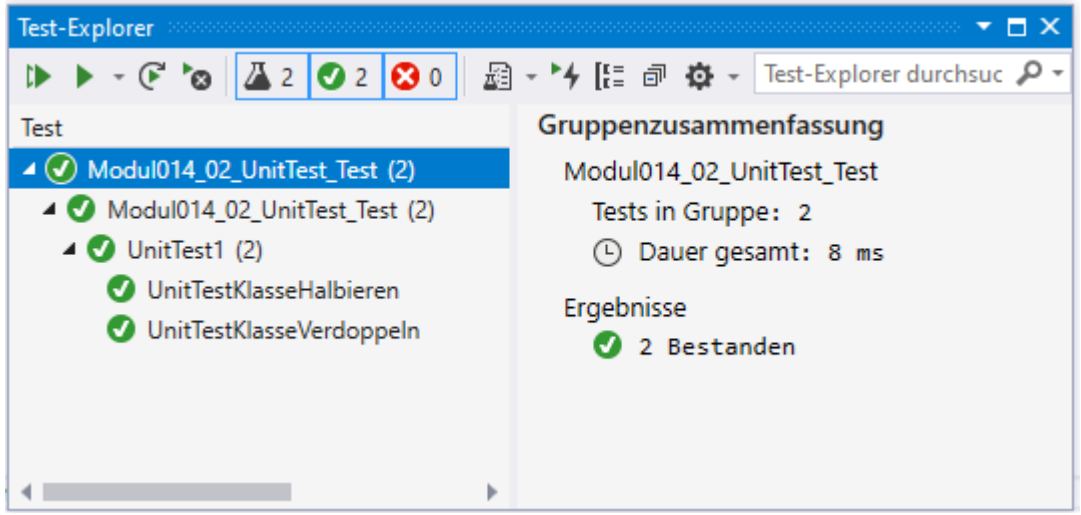
- eine Klasse muss mit dem Attribut „[TestClass]“ versehen werden, damit VS diese als Testklasse erkennt

- eine Methode muss mit dem Attribut „[TestMethod]“ versehen werden, damit VS diese als Testmethode erkennt und bei einem UnitTest aufruft

```
[TestClass]
0 Verweise
public class UnitTest1
{
    [TestMethod]
    0 Verweise
    public void UnitTestKlasseHalbieren()
    {
        UnitTestKlasse unitTestKlasse = new UnitTestKlasse();
        for (int i = 0; i < 100; i++)
        {
            int halbiert = unitTestKlasse.Halbieren(i);
            Assert.AreEqual(i / 2, halbiert);
        }
    }
}
```

UnitTest ausführen

- um einen UnitTest zu starten klickt man im Menü „Test“ auf den Punkt „Alle Test ausführen“
- alternativ mithilfe der Tastenkombination [STRG]+[R] [STRG]+[A]
- im Test-Explorer werden dann die Ergebnisse angezeigt



Generische Listen

generische Datentypen

- durch generische Datentypen kann der Datentyp von Membern (Methoden/Eigenschaften) variabel gestaltet werden
- generische Typen werden mit dem Platzhalter „T“ festgelegt
- innerhalb der Klasse können dann Objekte des Datentyps T erzeugt werden

- oft für Auflistungen benutzt, welche egal bei welchem Datentyp das gleiche Grundverhalten aufweisen sollen

List<T>

- eine Liste von Objekten <T>, auf die per Index zugegriffen werden kann

- vordefinierte Methoden zum Hinzufügen, Entfernen, Durchsuchen, Sortieren und Bearbeiten

```
List<string> stringListe = new List<string>();
stringListe.Add("Das");
stringListe.Add("ist");
stringListe.Add("eine");
stringListe.Add("Liste");
stringListe.Add("mit");
stringListe.Add("strings");
foreach (var item in stringListe)
{
    Console.WriteLine(item);
}

//Das
//ist
//eine
//Liste
//mit
//strings
```

List<T> - nützliche Funktionen

Funktionsname	Beschreibung
Add(T)	fügt der Liste ein Objekt hinzu
Remove(T)	entfernt das erste Objekt T
Clear()	entfernt alle Elemente aus der Liste
Find(Predicate<T>)	gibt ein Objekt zurück, welches dem Predicate<T> entspricht
Contains(T)	gibt zurück ob das Objekt T in der Liste enthalten ist
Exist(Predicate<T>)	gibt zurück ob ein Objekt, welches dem Predicate<T> entspricht, existiert
Sort(Comparison<T>)	sortiert die Liste mithilfe der angegebenen Comparison<T>

- Die komplexeren Methoden (Find(), Contains(), etc.) finden sich als Erweiterungsmethoden im Namespace `System.Linq`

spezielle Listen – Dictionary<TKey, TValue>

- Entspricht ‚Tabelle mit 2 Spalten‘
- Ordnet jeweils einem Key (TKey) ein Value (TValue) zu
- Add(TKey, TValue): Element hinzufügen
- über Key-Angabe Zugriff auf den Value
- ContainsKey(TKey) prüft, ob ein Key vorhanden ist
- ContainsValue(TValue) prüft, ob ein Value vorhanden ist

```
Dictionary<int, string> stringDictionary = new Dictionary<int, string>();
stringDictionary.Add(5, "Element 1");
stringDictionary.Add(10, "Element 2");
string value10 = stringDictionary[10]; // "Element 2"
stringDictionary.ContainsKey(5); //true
stringDictionary.ContainsValue("Element 3"); //false
```

Linq und Erweiterungsmethoden

Linq

- **L**anguage **In**tegrated **Q**uery
- Collections verarbeiten
  - alle Klassen die von IEnumerable erben (List, Array, Dictionary, …)
- SQL-ähnliche Verwendung (filtern, sortieren, summieren, …)
- Lambda-Expressions (=>)
- verketten von einzelnen Linq-Abfragen
- verändern **nicht** die originale Collection
  - Zuweisung auf die originale Collection um Änderungen zu „Übernehmen“

Aufbau

```
List<int> list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
          list.Where(e => e % 2 == 0).ToList();
```

Originale Collection — Linq-Funktion — Lambda-Expression — Linq-Funktion

- Finde alle geraden Zahlen und erzeuge eine neue Liste
- mit Variablenname (e) und => eine Bedingung festlegen
- abschließend mit ToList() das aus Where(…) resultierende IEnumerable<int> zu einer List<int> konvertieren

System.Linq.Enumerable.ToList<TSource> returned	Count = 4	System.Collections.Generic.List<int>
[0]	2	int
[1]	4	int
[2]	6	int
[3]	8	int

Einfache Linq-Funktionen

```
List<int> list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Funktionsname	Beschreibung	Ergebnis
Average	Gibt den Durchschnitt der Liste zurück	5
Min/Max	Gibt den kleinsten/größten Wert der Liste zurück	1, 9
Sum	Gibt die Summe der Liste zurück	45
First	Gibt das erste Element der Liste zurück, wirft eine Exception wenn kein Element gefunden wird	1
FirstOrDefault	Erstes Element, null wenn kein Element gefunden wird	1
Last	Gibt das letzte Element der Liste zurück, wirft eine Exception wenn kein Element gefunden wird	9
LastOrDefault	Letztes Element, null wenn kein Element gefunden wird	9

Vergleich Linq-Schreibweisen

```
List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

List<int> geradeForEach = new List<int>();
foreach (int i in list)
    if (i % 2 == 0)
        geradeForEach.Add(i);

List<int> geradeSQL = (from i in list
                       where i % 2 == 0
                       select i).ToList();

List<int> geradeLinq = list.Where(e => e % 2 == 0).ToList();
```

Linq mit Objektliste

```
List<Fahrzeug> fahrzeuge = new List<Fahrzeug>()...;
```

Funktionsname	Beschreibung	Ergebnis
Where	Filtert die Liste nach einem oder mehreren Kriterien	IEnumerable<Fahrzeug>
Select	Projiziert die Liste auf eine neue Liste nach einem Kriterium	IEnumerable<?>
OrderBy und OrderByDescending	Gibt eine neue Liste sortiert (auf- oder absteigend) nach einem Kriterium zurück	IEnumerable<Fahrzeug>
All/Any	Überprüft ob alle/mindestens ein Element in die Liste die angegebene Bedingung erfüllt	bool
Count/LongCount	Zähle die Elemente in der Liste anhand eines/mehreren Kriterien	int/long
MinBy/MaxBy	Gibt das Objekt mit dem kleinsten/größten Wert anhand eines Kriteriums zurück	Fahrzeug
Skip/Take	Überspringt X Elemente, nimmt die nächsten X Elemente. In Kombination verwenden um Teile der Liste zu nehmen	

Linq mit Objektliste (2)

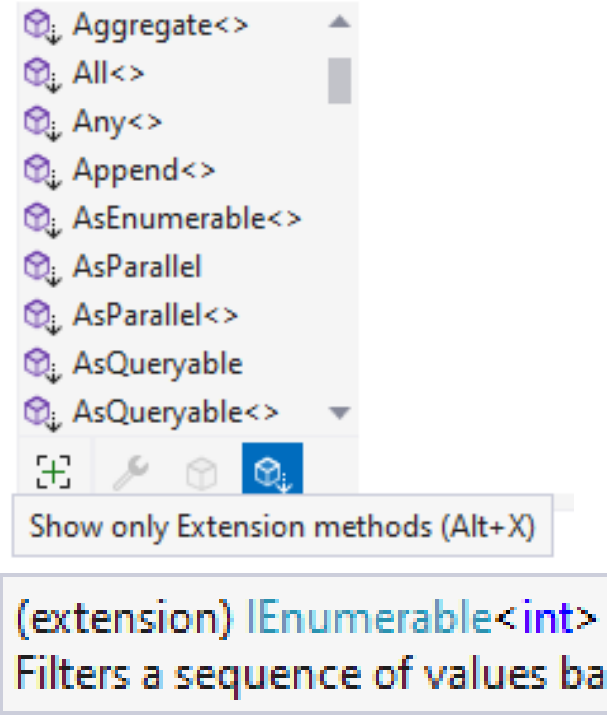
Funktionsname	Beschreibung	Ergebnis
SequenceEqual	Überprüft, ob zwei Listen die selben Elemente haben	bool
Reverse	Erzeugt eine neue Liste die die originale Liste nur umgekehrt ist	IEnumerable<Fahrzeug>
Concat	Verbindet zwei Listen zu einer neuen Liste	IEnumerable<Fahrzeug>
Prepend/Append	Erzeugt eine neue Liste und fügt das Element am Anfang/Ende an	IEnumerable<Fahrzeug>
Except	Gibt eine neue Liste zurück mit Elementen die nur in einer der beiden Listen ist	IEnumerable<Fahrzeug>
Intersect	Gibt die Schnittmenge zwischen den beiden Listen zurück	IEnumerable<Fahrzeug>
SelectMany	Glättet eine Liste von Listen auf eine einzelne Liste herunter	IEnumerable<Fahrzeug>
Chunk	Teilt die Liste auf Teile mit der angegebenen Größe auf	IEnumerable<Fahrzeug[]>
GroupBy	Gruppiert die Liste nach einem Kriterium	IEnumerable<IGrouping<?, Fahrzeug>>
Aggregate	Wendet für jedes Element eine Funktion auf einen Aggregator an	? (Typ des Aggregators)

Konvertierungsfunktionen

- Originale Collection in eine andere Form umwandeln
  - List ↔ Array
  - IEnumerable → List
  - IEnumerable<IGrouping<?, Fahrzeug>> (GroupBy) → Dictionary
- ToArray
- ToDictionary
- ToHashSet
- ToList
- AsEnumerable

Erweiterungsmethoden

- alle Linq Methoden sind Erweiterungsmethoden
- beliebige Methoden an Typen anhängen
- gekennzeichnet mit Würfel + Pfeil nach unten oder mit (extension) bei Methodenvorschau



- Aufbau
  - müssen in einer statischen Klasse definiert werden
  - müssen selbst auch statisch sein
  - mit `this` <Typ> sich auf einen Typen beziehen

Eigene Erweiterungsmethode

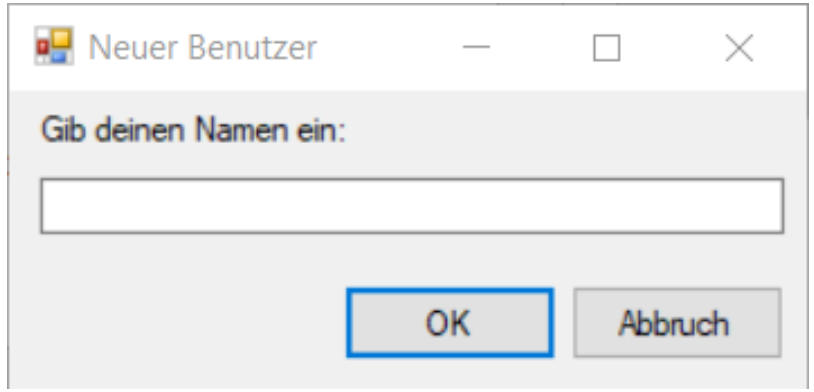
```
public static int GetQuersumme(this int zahl)
{
    int summe = 0;
    string zahlAlsString = zahl.ToString();
    for(int i=0;i<zahlAlsString.Length; i++)
    {
        summe += (int) char.GetNumericValue(zahlAlsString[i]);
    }
    return summe;
}

Console.WriteLine(10232.GetQuersumme()); //Gibt 8 zurück
```

This zeigt an für welche Klasse eine Erweiterungsmethode definiert werden soll

C# mit GUI
am Beispiel von Windows Forms

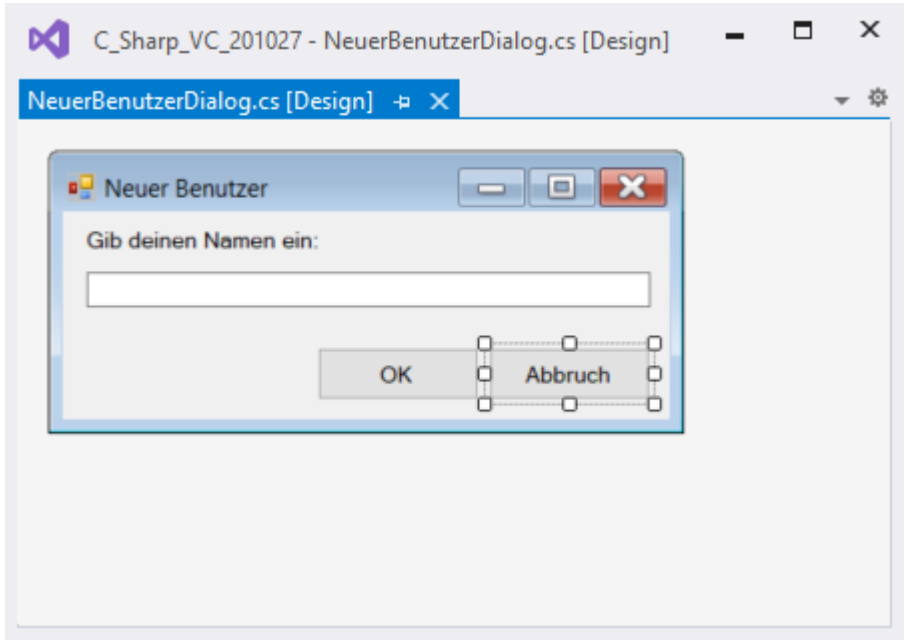
Windows Forms

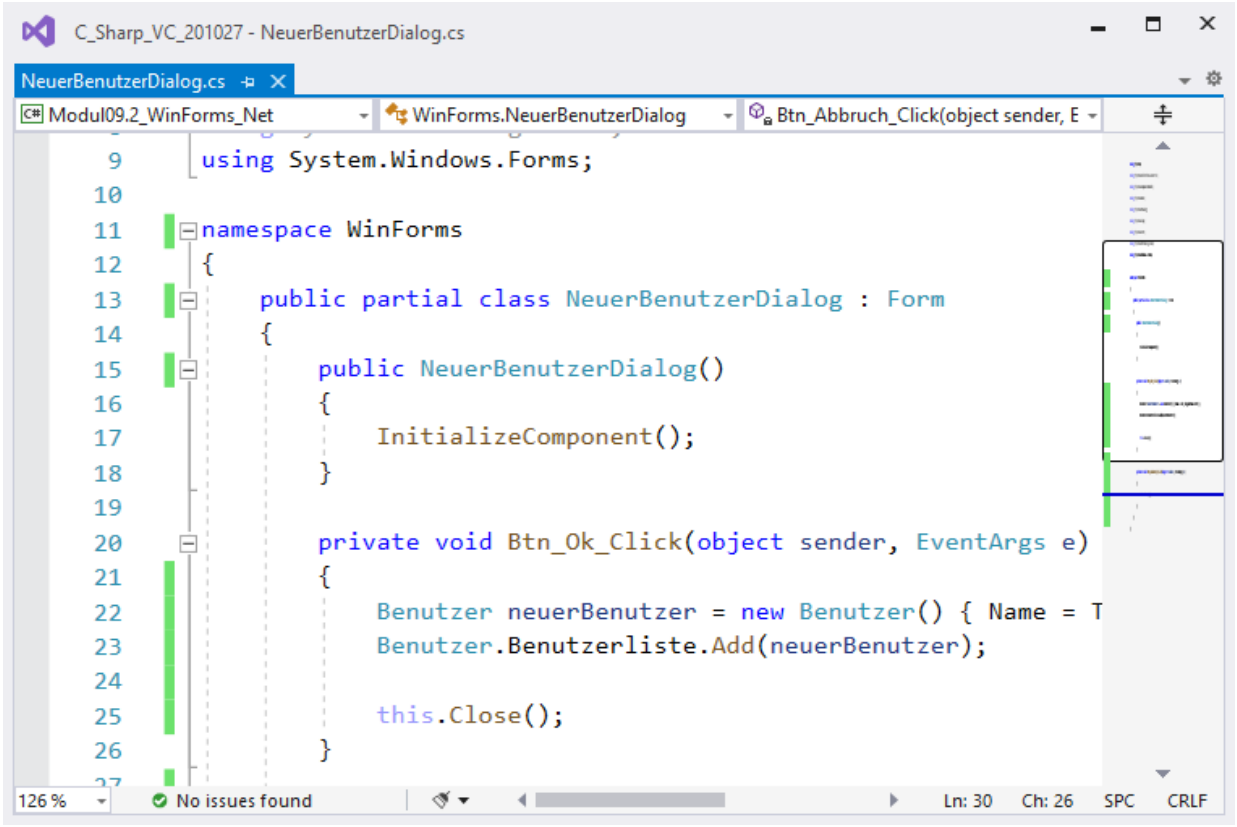
- Windows Forms ist das UI-Framework für Desktop-Programme
  - alternatives UI-Framework: Windows Presentation Foundation (WPF)
- Form: Klasse zur Darstellung eines Windows-Fensters
- Event-basierte Userinteraktion
- Design mittels visuellem Designer



Windows Forms

- ‚zweigeteilte' Bearbeitung:
  - GUI-Design im visuellen Designer
  - Logik im ‚Code-Behind'



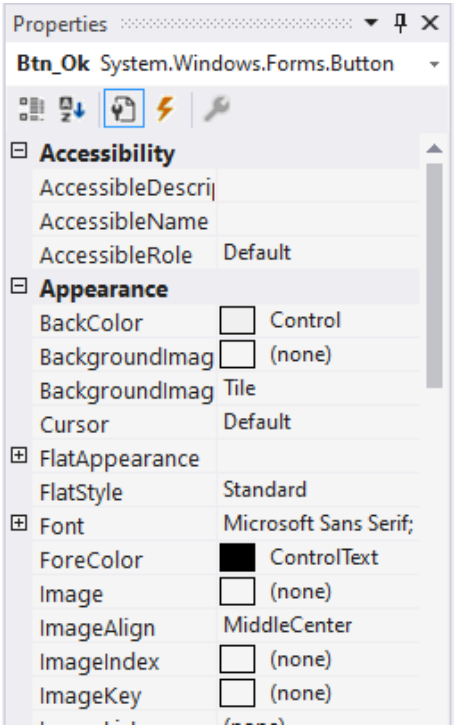


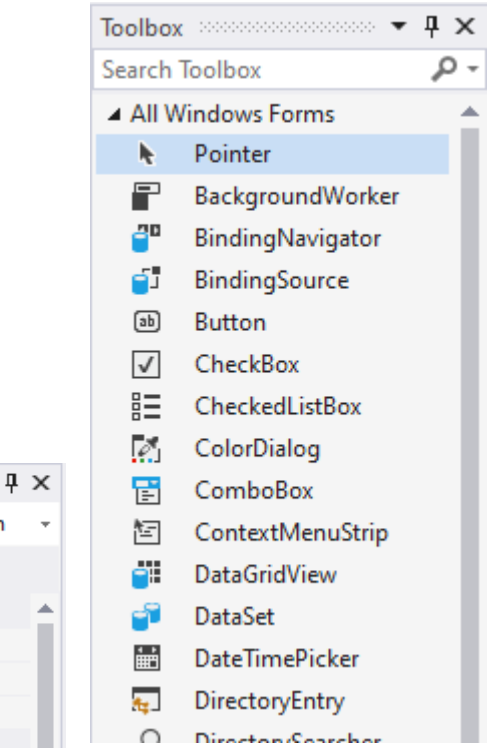
partielle Klassen und Methoden

- Klassen und Methoden können auch in unterschiedlichen Quelldateien aufgeteilt werden
- jede Quelldatei enthält Teile der Typ- oder Methodendefinition
- werden bei der Kompilierung der Anwendung kombiniert
- jede Teilklasse muss als partial markiert werden
- partielle Methoden können nur den Rückgabewert „void“ haben
- Schlüsselwort „partial“ vor dem Schlüsselwort „class“ oder „void“ stehen
- Methoden müssen privat sein und können nur einmal implementiert werden

Windows Forms mit Visual Studio

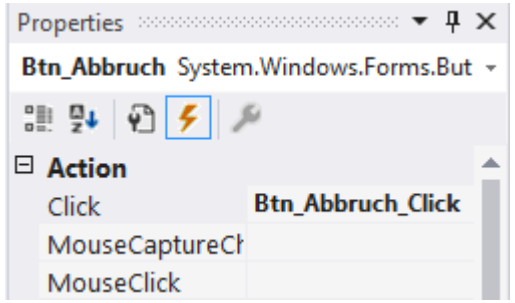
- ‚Toolbox' beinhaltet die GUI-Objekte
  - Drag'n'Drop zum Plazieren im Designer
- ‚Eigenschaften'-Fenster gewährt Zugriff auf die Properties des merkierten GUI-Objekts





Eigenschaften und Events

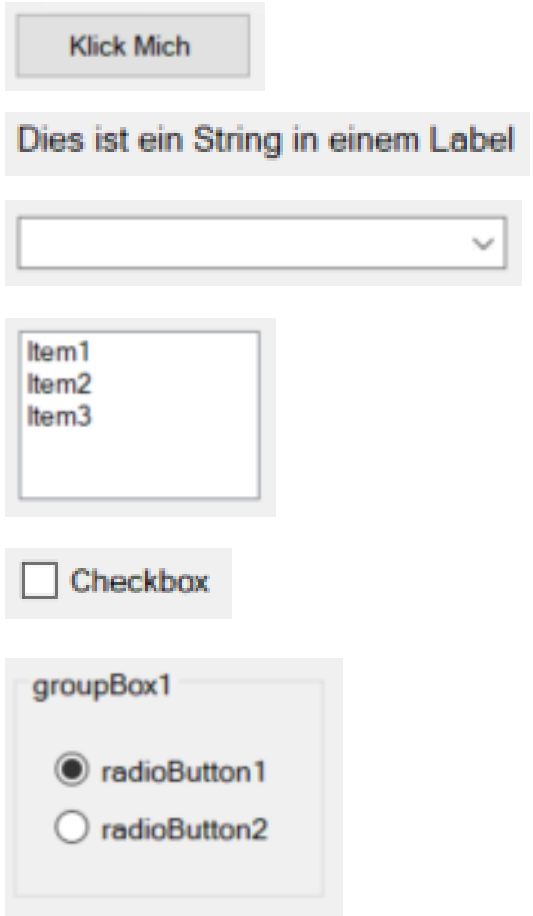
- Im ‚Eigenschaften'- Fenster ( ) findet sich eine Auflistung aller zugänglichen Eigenschaften des im Designer markierten Objekts
  - (Name) definiert den Variablen-Bezeichner, unter dem das Objekt aufgerufen werden kann
- Im ‚Events'-Fenster ( ) befindet sich eine Liste der verfügbaren Events
  - Events müssen mit einer EventHandler-Methode im Code-Behind verknüpft sein




```
private void Btn_Abbruch_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Wichtige GUI-Objekte

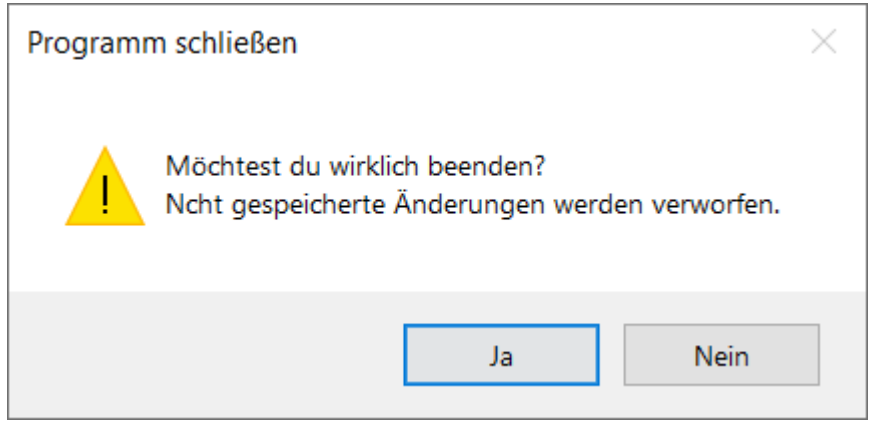
- Button
- Label
- ComboBox
- ListBox
- CheckBox
- RadioButton



- GUI-Klassen erben (indirekt) von `System.Windows.Forms.Control`
- Neben den Ererbten hat jede GUI-Klasse spezifische, klassen-eigene Eigenschaften und Events

MessageBox

- Message-Boxen sind kleine, automatisch generierte Dialogfenster
- Für User-Nachfragen oder Informationen
- Aufruf erfolgt über statische Methode `MessageBox.Show()`
- Rückgabe vom Enum-Typ DialogResult entspicht geklicktem Button




```
string dialogText = "Möchtest du wirklich beenden?\nNcht gespeicherte Änderungen werden verworfen.";
DialogResult result = MessageBox.Show(dialogText, "Programm schließen", MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
if (result == DialogResult.Yes)
    this.Close();
```

Schließen und Öffnen von Forms

- `this.Close()` schließt das aktuelle Form
- `Form.Show()` öffnet ein Form als gleichberechtigtes Fenster
- `Form.ShowDialog()` öffnet ein Form als Dialog-Fenster
  - Dialogfenster müssen bearbeitet und geschlossen werden, bevor der User ein anderes Fenster bearbeiten kann
  - ShowDialog gibt DialogResult zurück

Dateien und Verzeichnisse

Dateien

- .NET hat fertige Klassen zum erstellen, lesen, schreiben und löschen von Dateien

- StreamWriter und StreamReader sind die Standardklassen für den Dateienzugriff
- können unterschiedliche Kodierungen abbilden
- Funktionen sind ähnlich wie bei der Konsole

- Streams müssen geöffnet und auch geschlossen werden
  - GC greift hier nicht ein, da geöffnete Dateien als „unmanaged ressource“ gelten

StreamWriter – System.IO

- mit einem StreamWriter können Dateien erstellt und Inhalt in diese geschrieben werden
- über die Eigenschaft Encoding kann die Kodierung der Ausgabe festgelegt werden
- mit Write() und WriteLine() können Inhalte an den Stream übergeben werden

```
StreamWriter sw = new StreamWriter("Testdatei.txt");
sw.WriteLine("Testzeile1");
sw.WriteLine("Testzeile2");
sw.Close();
```

StreamReader – System.IO

- mit einem StreamReader können Inhalte einer Datei ausgelesen werden
- über die Eigenschaft EndOfStream kann geprüft werden ob der Stream noch weitere Daten enthält
- mit Read() und ReadLine() können Inhalte aus dem Stream gelesen werden

```
StreamReader sr = new StreamReader("Testdatei.txt");
while (!sr.EndOfStream)
{
    string zeile = sr.ReadLine();
}
sr.Close();
```

Using-Statement

- Klassen, welche externe Resourcen wie Dateien oder Datenbankserver verwalten, sollten das IDisposable-Interface implementieren
- Dispose() räumt alle externen Resourcen auf, d.h. Dateien schließen oder DB-Verbindung lösen
- Mittels dem using-Statement wird Dispose() automatisch aufgerufen sobald das Ende des Using-Blockes erreicht wurde

```
using(StreamReader reader = new StreamReader("test.txt"))
{
    //Lese-Operationen
    //Am Ende des Blockes wird automatisch reader.Dispose() aufgerufen
}
```

SaveFileDialog – System.Windows.Forms

- mit einem SaveFileDialog kann ein Pfad und ein Dateiname ausgewählt werden
- die Eigenschaft Filter ermöglicht es nur bestimmte Dateiendungen zuzulassen
- über die Eigenschaft FileName kann man eine Pfadangabe vordefinieren bzw. den gewählten Pfad abrufen
- In WPF: using Microsoft.Win32

```
SaveFileDialog saveFileDialog = new SaveFileDialog();
saveFileDialog.FileName = "Personen.pl";
saveFileDialog.Filter = "Personenliste|*.pl|Textdokument|*.txt";
if (saveFileDialog.ShowDialog() == DialogResult.OK)
{
    string pfadZuDatei = saveFileDialog.FileName;
}
```

OpenFileDialog – System.Windows.Forms

- gleiche Funktionalität wie SaveFileDialog

- der Dateipfad muss existieren um den Dialog zu bestätigen
- In WPF: using Microsoft.Win32

```
OpenFileDialog openFileDialog = new OpenFileDialog();
openFileDialog.FileName = "Personen.pl";
openFileDialog.Filter = "Personenliste|*.pl|Textdokument|*.txt";
if (openFileDialog.ShowDialog() == DialogResult.OK)
{
    string pfadZuDatei = openFileDialog.FileName;
}
```

JSON – Newtonsoft.Json (NuGet-Packet)

- ein Objekt kann mit dem aktuellen Zustand bspw. in eine Datei geschrieben und später wieder eingelesen werden
- die Klasse JsonConvert kümmert sich um die Konvertierung der Inhalte
- mit der Methode SerializeObject der Klasse JsonConvert kann ein beliebiges Objekt in einen JSON-String konvertiert werden (serialisiert)
- mit der Methode DeserializeObject mit der Typangabe <T> der Klasse JsonConvert kann ein beliebiges Objekt in einen JSON-String konvertiert werden (serialisiert)

```
List<string> strings = new List<string> { "Wert1", "Wert2", "Wert3" };

string jsonString = JsonConvert.SerializeObject(strings);

List<string> stringListe = JsonConvert.DeserializeObject<List<string>>(jsonString);
```

JSON – System.Text.Json (ab .NET Core 3.0)

- neuer JSON Konverter direkt in .NET Core enthalten
- weniger Funktion als Newtonsoft.Json
- eine referenzierte Schleife führt zu einer Fehlermeldung
- um einiges schneller als Newtonsoft.Json
- ist jedoch Typabhängig (int => int only)
  Case sensitiv, Eigenschaftsnamen nur mit doppeltem Anführungszeichen

```
List<string> strings = new List<string> { "Wert1", "Wert2", "Wert3" };

string jsonString = JsonSerializer.Serialize(strings);

List<string> stringListe = JsonSerializer.Deserialize<List<string>>(jsonString);
```

XML – System.Xml.Serialization

- mit einem XmlSerializer kann ein Objekt mithilfe eines TextWriters (bspw. StreamWriter) direkt in eine Datei geschrieben werden
- im Unterschied zu den JSON-Klassen muss bei XML ein eigenes Objekt der Klasse XmlSerializer erzeugt werden, welches im Konstruktor eine Instanz des Typs des zu serialisierenden Objektes haben muss
- über die Methode Serialize des XmlSerializer Objektes wird der TextWriter sowie das zu serialisierende Objekt übergeben

```
List<string> liste = new List<string> { "Wert1", "Wert2", "Wert3" };

XmlSerializer serializer = new XmlSerializer(typeof(List<string>));

StreamWriter sw = new StreamWriter("liste.xml");
serializer.Serialize(sw, liste);
sw.Close();
```

Delegates und Events

Delegates

- Delegate-Variablen speichern Referenzen auf andere Methoden
- die Referenzen können zur Laufzeit hinzugefügt oder entfernt werden
- die referenzierten Methoden müssen die gleiche Signatur (gleicher Aufbau) wie der Delegate-Typ haben

```
public delegate string MeinDelegateTyp(string param1, int param2);

public string Vorstellung(string name, int alter)
{
    return $"Hallo, ich heisse {name} und bin {alter} Jahre alt.";
}

MeinDelegateTyp meinDelegate = new MeinDelegateTyp(Vorstellung);
meinDelegate("Max Mustermann", 37);
//Hallo, ich heisse Max Mustermann und bin 37 Jahre alt.
```

Delegates

- Jede Delegate-Variable kann beliebig viele Methoden referenzieren, welche nacheinander ausgeführt werden
- Methoden können durch += hinzugefügt und durch -= herausgelöst werden

```
public delegate void MeinDelegateTyp(string param1, int param2);

public void VorstellungDeu(string name, int alter)
{
    Console.WriteLine($"Hallo, ich heisse {name} und bin {alter} Jahre alt.");
}
public void VorstellungEng(string name, int age)
{
    Console.WriteLine($"Hello, my name is {name} and I am {age} years old.");
}
public void VorstellungFra(string nom, int age)
{
    Console.WriteLine($"Bonjour, je m'appelle {nom} et j'ai {age} ans.");
}
public void VorstellungSpa(string nombre, int anos)
{
    Console.WriteLine($"Hola, mi nombre es {nombre} y tengo {anos} años.");
}
```

```
MeinDelegateTyp meinDelegate = new MeinDelegateTyp(VorstellungDeu);
meinDelegate += new MeinDelegateTyp(VorstellungEng);
meinDelegate += new MeinDelegateTyp(VorstellungFra);
meinDelegate += new MeinDelegateTyp(VorstellungSpa);

meinDelegate("Max Mustermann", 37);
//Hallo, ich heisse Max Mustermann und bin 37 Jahre alt.
//Hello, my name is Max Mustermann and I am 37 years old.
//Bonjour, je m'appelle Max Mustermann et j'ai 37 ans.
//Hola, mi nombre es Max Mustermann y tengo 37 años.
```

vordefinierte Delegate-Typen

Name	Rückgabetyp	Parameter	Beispiel für referenzierbare Methode

Action
Action<in T1>
Action<in T1, in T2>
…
	void	0-16	public void PrintInteger(int p1)
			{
			  Console.Write(p1);
			}
			
			Action<int> a = PrintInteger

Predicate<in T>	bool	1	public bool CheckForZero(int p1)
			{
			  return p1 == 0;
			}
			
			Predicate<int> p = CheckForZero

Func<out TResult>
Func<in T1, out TResult>
Func<in T1, in T2, out TResult>
…
	TResult	0-16	public string SumToString(int p1, int p2)
			{
			  return (p1 + p2).ToString();
			}
			
			Func<int, int, string> f = SumToString

anonyme Methoden

- Methoden ohne Namen nennt man anonyme Methoden
- Methoden können einem Delegate zugewiesen werden, ohne diese vorher zu deklarieren

```
Func<int, int> meineFunc = delegate (int param1)
{
    return param1 + 5;
};
```

- ‚Wegwerfmethoden' für einmaligen Gebrauch, z.B. zur Übergabe zusätzlicher Logik an Methoden

```
List<string> Städteliste = new List<string>() { "München", "Berlin", "Hamburg", "Hannover", "Köln" };
string gefundeneStadt = Städteliste.Find
    (
        delegate (string stadt)
        {
            return stadt.StartsWith('B');              // => gefundeneStadt = "Berlin"
        }
    );
```

Lambda - Ausdrücke

- Kurzschreibweise für anonyme Methoden

```
Func<int, int, int> Addiere = delegate(int x, int y) {return x+y;}
```

```
Func<int, int , int> Addiere = (int x, int y) => {return x+y;}
```

```
Func<int, int , int> Addiere = (x, y) => x+y;
```

EventHandler

- ein Event ist eine Delegate-Variable, an der sich beliebig viele Methoden (EventListener) an- und abmelden können
- Delegate-Typ ist i.d.R. EventHandler
  - Erwartete Übergabeparameter: object, EventArgs
- Kein ‚Überschreiben' durch Neuzuweisung möglich

```
event EventHandler onClick;

onClick += delegate (object sender, EventArgs eventArgs)
{
    Console.WriteLine("EventBehandlung");
};

onClick(this, EventArgs.Empty);
```