



C# >  
GETTING STARTED

C# >  
ECOSYSTEM

C# >  
SYNTAX

C# >  
DATA TYPES & OPERATORS

C# >  
DEBUGGING

C# >  
ARRAYS & CONTROLS

C# >  
METHODS | FUNCTIONS

DRAFT  
C# >  
NAMESPACES

C# >  
CLASSES

C# >  
INHERITANCE

C# >  
INTERFACES & POLYMORPHISM

DRAFT  
C# >  
STRUCTS

DRAFT  
C# >  
ATTRIBUTES

C# >  
GENERICS

C# & WINDOWS FORMS

C# > DELEGATES,  
ANONUMOUS FUNCTIONS  
EVENT HANDLING

C# >  
ERROR HANDLING

C# & UNIT TESTS

C# >  
FILE ACCESS

C# >  
FORTGESCHRITTENE THEMEN

C# >  
LINQ

C# >  
NEUE FEATURES

C# >  
ASYNC

C# >  
SOLID

C# >  
PREPROCESSOR DIRECTIVES

C# >  
MULTITHREADING

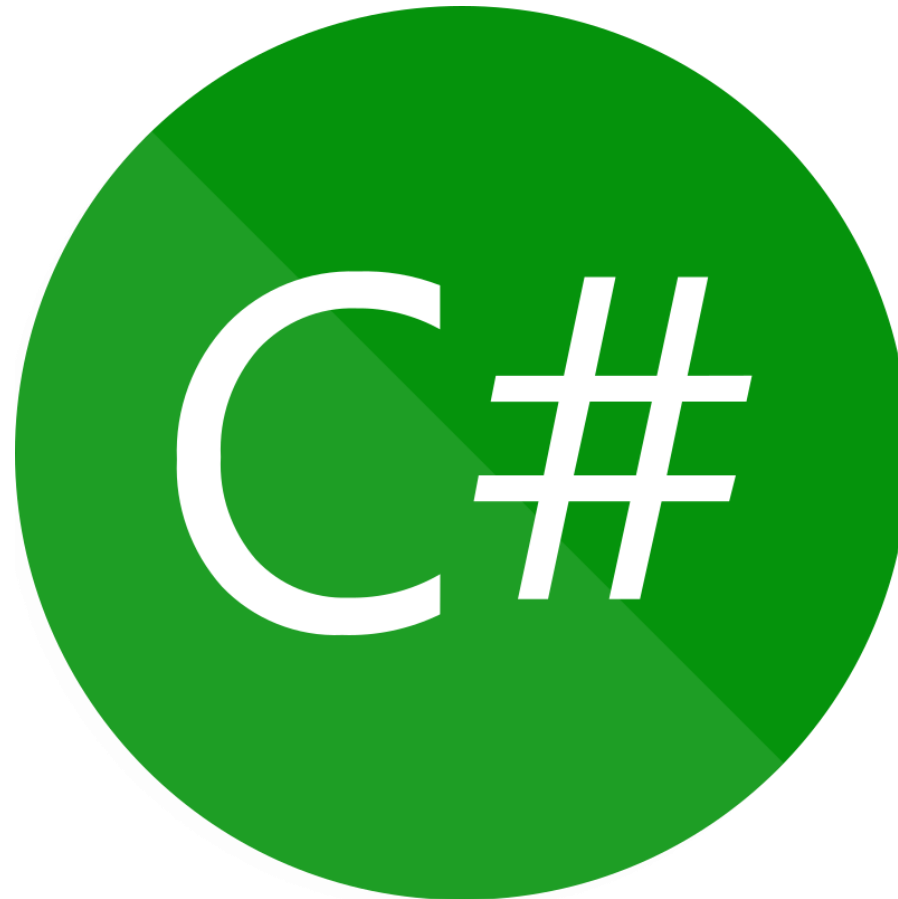
DRAFT  
C# >  
IMPLEMENTATIONS

DRAFT  
C# >  
STYLE GUIDES

DRAFT  
C# >  
TESTING

DRAFT  
C# >  
DEPLOYMENT

DRAFT  
C# >  
FACTS



# C# > GETTING STARTED

# C# > GETTING STARTED > IDEA OF #C#

- WAS IST C#
  - eine Programmiersprache
  - objektorientierte Sprache
  - Weiterentwicklung von C und C++
  - aktuelle Version 8.0 (Sep. 2019) `#checkForUpdates`
- Merkmale
  - Case-sensitive
  - Garbage-Collection

## WOZU IST C#

- Byte-basiert
- Microsoft .NET (Windows Desktop)
- UWP – Windows 10 Apps
- .NET Core – Windows/Linux/Mac (aber dann ohne WinForms...)
- XAMARIN – iOS/Android
- ASP.NET – Webserver
- Spiele-Programmierung (Unity)

# C# > GETTING STARTED > LINKS

- OFFICIAL HOMEPAGE

- <https://docs.microsoft.com/de-de/dotnet/csharp/>

- OFFICIAL DOCS

- <https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/>
  - <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/>

- OFFICIAL BLOG

- <https://devblogs.microsoft.com/dotnet/>

- OFFICIAL CODE

- f12

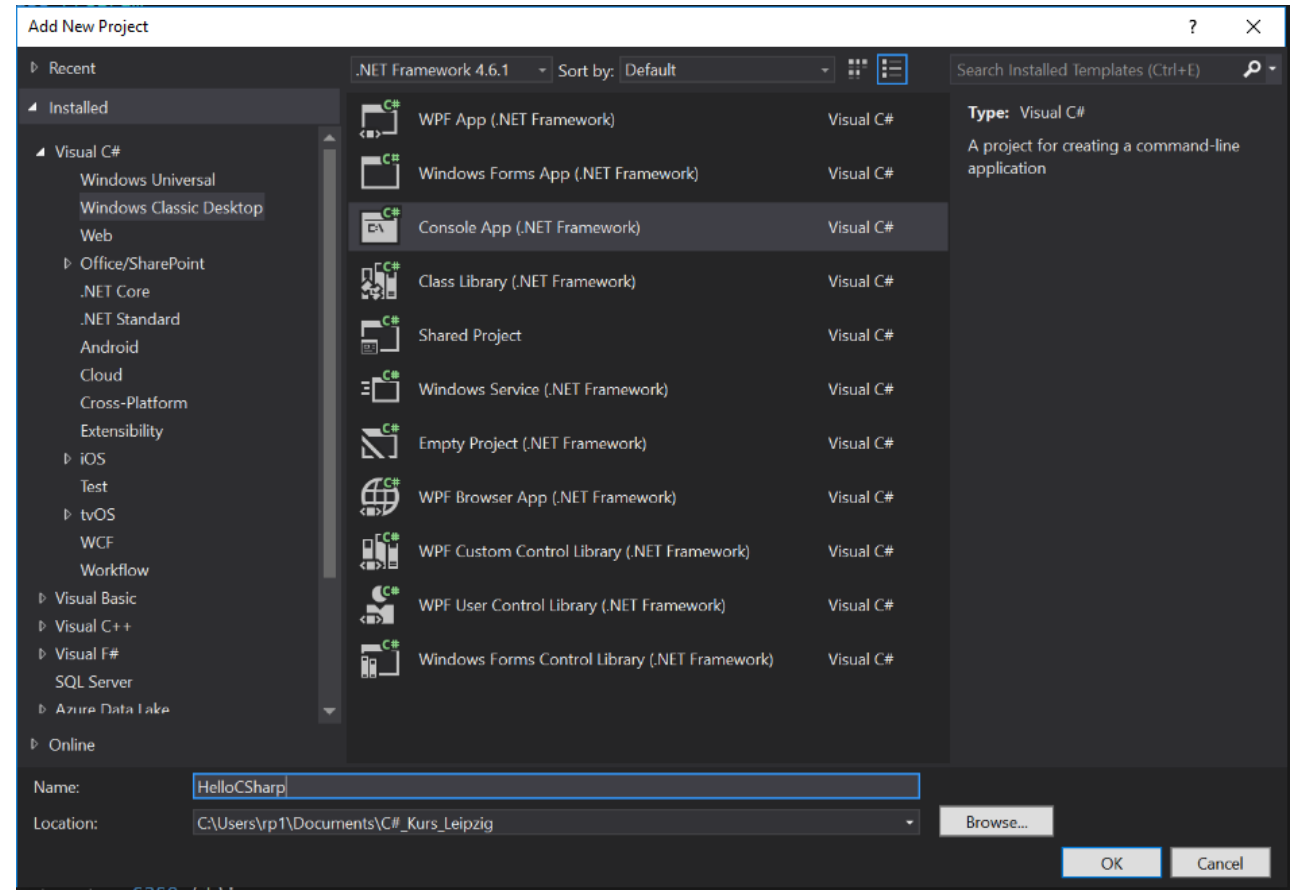
## C# > GETTING STARTED > PRIMARY TOOLS

- eine IDE, z.B. Visual Studio (oder Xamarin Studio oder Unity)
  - SDK (Software Developer Kit), z.B. .NET
    - ein Compiler, z.B. Roslyn

## C# > GETTING STARTED > LAB

- Hello C# Projekt
- Solutions am besten pro Teilnehmer unterschiedlich benennen, um sie im gemeinsamen Repository zu unterscheiden)
- Konsolenanwendung in Visual Studio
- Variablen deklarieren und Text ausgeben

Übung

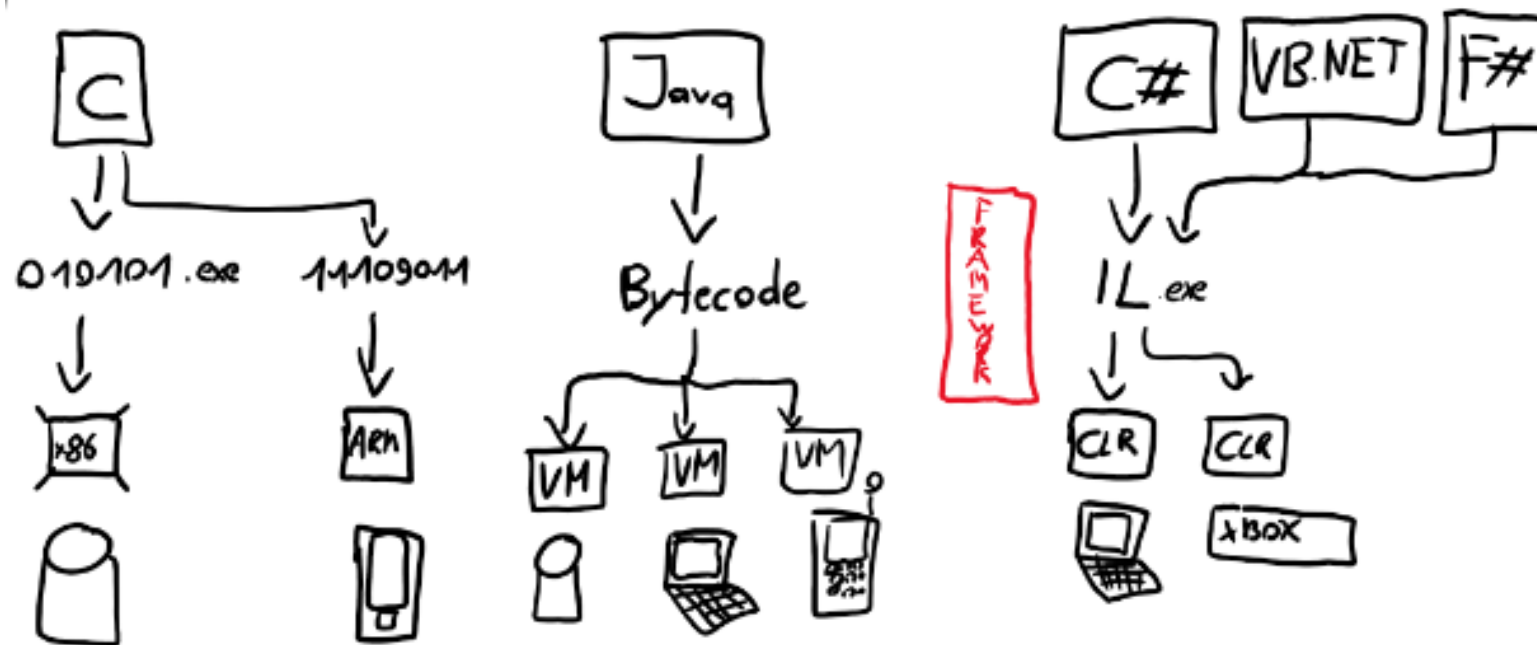


# C# > ECOSYSTEM



# C# > ECOSYSTEM > CLR

- CLR - Common Language Runtime



# C# > SYNTAX

# C# > SYNTAX Sprachgrundlagen

- Befehle / Expressions / Ausdrücke

```
Console.WriteLine(„Hello World“);  
Console.ReadKey();
```

- Zuweisungen / Statements

```
int Nummer = 20;  
string Wort = „Wort“;
```

- Kommentare

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



ctrl k c - comment  
ctrl k u - uncomment

## C# > ESSENTIALS > REGIONS

- weitere Möglichkeit, den Code zu kommentieren
- mit Regions kann man den Code in thematische / funktionale Abschnitte aufteilen
- Regions lassen sich auch verschachteln
- Regions kann man zusammenklappen
- Es gibt keine Tastenkombination in VS für Zusammen-/Aufklappen von Regions
  - Benutze die Erweiterung Collapse Region dafür



region tab tab

```
#region <NameDesCodeabschnitts>  
    // code  
#endregion
```

## C# > ESSENTIALS > VARIABLES & DATA TYPES

- Variablen sind Behälter/Container für Werte
- Syntax beim Deklarieren und evtl Initialisieren von Variablen:
  - <type> <identifier>;
  - <type> <identifier> = <literal>;
  - <type> <identifier> = <expression>;
- Syntax beim Aufrufen von Variablen:
  - <identifier>

```
int Alter; //Deklarieren / Deklaration
Alter = 20; //Initialisierung: Integer mit Wert 20
int Summe = Alter*2; //Summe bekommt Wert 40
string Stadt = "Berlin"; //Zeichenkette "Berlin"
Console.WriteLine(Stadt); //Berlin in Konsole schreiben
```

## C# > ESSENTIALS > VARIABLES & DATA TYPES

- VAR
- steht für variant
- wenn Datentyp unbekannt

```
foreach (var item in myArray) { ... }
```

## C# > ESSENTIALS > SCOPES

- Namensräume? - ja
- Klassen? - ja
- Methoden? - ja
- Geschweifte Klammern? - nein

## C# > ESSENTIALS > VS C# SNIPPETS > OVERVIEW

- Attribute
- checked
- class
- ctor
- cw
- do
- else
- enum
- equals
- Exception
- for
- foreach
- forr
- if
- indexer
- interface
- invoke
- iterator
- iterindex
- lock
- mbox
- namespace
- prop
- propa
- propdb
- propfull
- propg
- sim
- struct
- svm
- switch
- testc
- testm
- try
- tryf
- unchecked
- unsafe
- using
- while
- ~



# **C# >**

# **DATA TYPES & OPERATORS**

# C# > DATA TYPE > IDEA

- WAS SIND DATENTYPEN
  - Attribute für Daten (Metainformationen über Daten)
- WOZU SIND DATENTYPEN
  - Infos für Compiler, wie viel Speicher für die Variablen reserviert werden soll
  - Infos darüber, was hat der Programmierer mit diesen Daten vor
  - in C# kann man über die Datentypen-Schlüsselwörter auch die entsprechende Klasse ansprechen
- BEGRIFFE
  - built-in types / eingebaute Datentypen
  - primitives / primitive Datentypen
  - custom types / benutzerdefinierte (genauer: programmiererdefinierte) Datentypen

# C# > DATA TYPES

Schlüsselwort	native Typen / .NET Typen (auch für F#, VB genutzt)	Beschreibung
<code>char</code>	System.Char	Einzelnes Textzeichen
<code>string</code>	System.String	Text (max. 2GB)
<code>short</code>	System.Short	16-Bit-Integer (0000000000000000)
<code>int</code>	System.Int32	32-Bit-Integer
<code>long</code>	System.Int64	64-Bit-Integer
<code>float</code>	System.Single	16-Bit-Gleitkommazahl
<code>double</code>	System.Double	32-Bit-Gleitkommazahl
<code>decimal</code>	System.Decimal	128-Bit-Gleitkommazahl
<code>bool</code>	System.Boolean	true oder false

Standardgemäß:

- Ohne Nachkommastellen -> `int`
- Mit Nachkommastellen -> `double`

## C# > DATA TYPES > TYPE CHECK

- Operator is
  - Syntax: `<expression> is <type>`
  - gibt bool zurück
- Methode `myVar.GetType()`

```
Alter + Stadt is string
```

# Ausgabe von Strings auf der Konsole

- verschiedene Schreibweisen:

```
int alterVonMax = 20;  
Console.WriteLine("Alter von Max: " + alterVonMax); // old school, vermeiden  
Console.WriteLine("Max ist {0} Jahre alt.", alterVonMax); // Indexplatzhalter  
Console.WriteLine($"Max ist {alterVonMax} Jahre alt."); // #v.6.0+
```



cw tab tab

# Konsoleneingaben verarbeiten

- Einlesen von Strings

```
string stringEingabe = Console.ReadLine();  
int intEingabe = int.Parse(Console.ReadLine());  
double doubleEingabe = double.Parse(Console.ReadLine());
```

- Bemerkung: `int.Parse()` ist nur für strings gedacht
- Einlesen von einzelнем Zeichen (Character)

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

## C# > DATA TYPES > CASTING

- Ungenaueres zum Genaueren - implizit
- Gleitkommazahl-Typen zu Ganzzahl-Typen - Nachkommastellen werden abgeschnitten (2,5665 -> 2)
- Genaueres zum Ungenaueren - muss explizit gecastet werden  
`varVomNeuenDTyp = (NeuerDTyp) varVomAltenDTyp;`

# Konvertierung von Datentypen (Casting)

- Zahl => String

- Implizit:

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

- Explizit:

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

- String => Zahl

- Nur per Hilfsfunktion möglich:

```
int zahl = int.Parse(Console.ReadLine());
```

```
double zahl2 = double.Parse(Console.ReadLine());
```

- Zahl => Zahl

```
int ganzzahl = 25;
```

```
double kommazahl = ganzzahl;
```

```
ganzzahl = (int)kommazahl;
```



# Mathematische Operatoren und Funktionen

- $a + b$  Plus
- $a - b$  Minus
- $a * b$  Multiplikation
- $a / b$  Division
- $a \% b$  Modulo (Rest der Division)
- $a++$   $a = a + 1$
- $a--$   $a = a - 1$
- $a += 2$   $a = a + 2$
- $b /= 2$   $b = b / 2$
- $\text{Math.Round}(2.5)$  Gerundete Zahl, bei .5 runde auf nächste gerade Zahl
- $\text{Math.Max}(a,b)$  Ermittle Maximum von beiden Zahlen
- $\text{Math.Min}(a,b)$  Ermittle Minimum von beiden Zahlen

# Enumeratoren

- dt.: Aufzählungstyp
- Eigenen Datentyp definieren, der nur ganz bestimmte Werte zulässt
- Achtung: Die Enums lassen sich nicht innerhalb von Methoden deklarieren.
- Elemente in Enums heißen Konstanten

```
enum Wochentag { Mo=1, Di=2, Mi, Do, Fr, Sa, So};
```

- Vorteil: bessere Lesbarkeit des Codes

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



enum tab tab

# CS > DATA TYPES & OPERATORS > LAB

- LAB > DEMO

- Dbool
- Dstring
- Dint
- Ddouble
- Denum

- LAB > QUESTIONS (lieber am nächsten Tag)

- s. nächste Folie

- LAB > EXERCISE

- Mini-Rechner

## C# > TYPES & OPERATORS > LAB > QUESTIONS

1. Was ist die Aufgabe der Garbage Collection?
2. Welchen Datentypen kann man für Zeichenketten (Text) verwenden?
3. Eine Variable wurde mit `int x = 2;` initialisiert, welchen Wert hat sie nach folgenden Operationen:  
`x += 2;`  
`x++;`  
`x = x % 2;`
4. Welcher Datentyp hat das Ergebnis der Division eines Integer-Werts mit einem Double-Wert?
5. Mit welcher Funktion kann man einen String in einen Double konvertieren?

# C# > DEBUGGING

# C# > DEBUGGING

- siehe VS > DEBUGGING

# C# >

# ARRAYS & CONTROLS

# Logische Operatoren

- 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
}
```



if tab tab  
else tab tab

# Switch

```
switch (tag)
{
    case Wochentag.Mo:
        Anweisung A;
        break;
    case Wochentag.Di:
        Anweisung B;
    case Wochentag.Mi:
        Anweisung C;
        break;
    default:
        Anweisung D;
        break;
}
```



switch tab tab

# Schleifen

## Kopfgesteuert

**while** (Bedingung)

```
{  
    //wiederholt solange bis Bedingung falsch ist  
}
```

## Fußgesteuert

**do**

```
{  
    //führt mindestens 1 Mal aus  
} while (Bedingung);
```



while tab tab  
do tab tab

## C# > CONTROLS > LOOPS > BREAK & CONTINUE

- Schlüsselwörter innerhalb von Schleifen:
- `break;` //verlasse die Schleife
- `continue;` //springe zum nächsten Schleifendurchlauf (Schleifen-Bedingung wird erneut geprüft)

# Zähl-Schleifen

`for(Initialisierung; Bedingung; Anweisung)`

Hochzählen(Inkrementierung):

```
for (int i = 0; i<10; i++)  
{  
    //Zähle von 0 bis 9  
}
```

Runterzählen(Dekrementierung):

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



for tab tab

# Arrays

## Aufzählung von gleichartigen Datentypen

```
int[] zahlen = new int[] { 2, 4, 5 };  
string[] buchstaben = new string[] { "A", "B", "C" };  
double[] kommazahlen = new double[10];
```

## Zugriff über Index

```
zahlen[2]; //5
```

```
buchstaben[1]; //"B"
```

```
zahlen[2] = 4; //setzt 3. Element im Array auf Wert 4
```

# Array durchlaufen

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

Oder

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



foreach tab tab

Achtung: foreach kann nur Werte auslesen, nicht verändern

# Nützliche Funktionen und Eigenschaften von Arrays

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

Methoden kommen aus Linq!

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



# LAB

- Siehe die drei nächsten Folien

# C# > ARRAYS & CONTROLS > LAB > QUESTIONS

1. Bool a = true und bool b = false, welchen Wahrheitswert ergeben folgende Ausdrücke:
  1. a == b
  2. a || b
  3. !(a && b)
  4. !b
  5. a ^ b
2. Welche der folgenden Array-Initialisierungen ist syntaktisch falsch?
  1. int[] a = new int[20];
  2. int[] b = new int[] {2, 5, 8}
  3. int[] c = new int[];
3. Wie oft wird eine do-while-Schleife mindestens durchlaufen?
4. Was ist der Unterschied zwischen break und continue in einer Schleife?

# LAB > Zahlenratespiel

- Zufallszahl ermitteln

```
//Initialisiere Zufallsgenerator  
Random generator = new Random();  
//Erzeuge Zahl zwischen 1 und 4  
int zufallszahl = generator.Next(1, 5);
```

Übung

## LAB > Zahlenratespiel

- Teil 1
  - Erzeuge eine Zufallszahl zwischen 1 und 5
  - Prüfe Nutzereingabe und gebe aus ob seine Zahl größer, gleich oder kleiner der Zufallszahl ist
- Teil 2
  - Der Nutzer soll solange neue Zahlen eingeben können bis er die richtige Zahl erraten hat
  - Am Ende des Spiels die Anzahl der Versuche ausgeben

Übung

# C# > METHODS | FUNCTIONS

# C# > METHODS | FUNCTIONS > DECLARE & CALL

```
public static int Addiere(int a, int b)
{
    int c = a + b;
    return c;
}
```

Modifier Rückgabewert Name(Parameterliste)

## Aufruf

```
int summe = Addiere(2, 4);
```

# Spezielle Parametertypen: out/ref

- Normalerweise werden Werttypen wie int, bool, string oder double in Funktionen als Wert übergeben
- Mittels out/ref können sie als Referenz übergeben werden

```
public void changeAlter(out int wert)
{
    wert = 6;
}
```

```
//Aufruf
int alter;
changeAlter(out alter); //alter ist jetzt 6
```

- Out behandelt die Variable so als hätte sie noch keinen Wert (nicht initialisiert)
- Mit out kann man eine Variable direkt beim Aufruf anlegen
- Bei Ref muss eine bereits initialisierte Variable übergeben werden

# Spezielle Parametertypen: params

- ermöglicht variable Anzahl von Parametern von einem Typ

```
public static int BildeSumme(params int[] summanden)
{
    int summe = 0;
    foreach (var item in summanden)
    {
        summe += item;
    }
    return summe;
}
```

```
BildeSumme(2, 3, 4);           //ergibt 9
BildeSumme();                  //ergibt 0
BildeSumme(new int[]{3, 3, 4}); //ergibt 10
```



# Spezielle Parametertypen: optional

- Parameter können schon in der Deklaration mit Defaultwerten initialisiert werden, beim Aufruf ist der Parameter dann optional

```
public string StringCombine(string[] strings, char trenner = '/')
{
    string resultString = strings[0];
    for(int i = 1; i < strings.Length; i++)
    {
        resultString += (trenner + strings[i]);
    }
    return resultString;
}

StringCombine(new string[] { "A", "B", "C" });           // "A/B/C"
StringCombine(new string[] { "A", "B", "C" }, '-');     // "A-B-C"
```

- Die optionalen Parameter müssen in der Funktionsdeklaration immer nach den Pflicht-Parametern folgen

## LAB > Fragen

1. Was stimmt nicht an folgenden Funktionsdeklaration?

- `void funktion1(int x, string y = "abc", int z)`
- `void funktion2(params int x)`
- `void funktion3(int x, out y, int x, out y)`

2. Warum sollte man statt fester String-Werte lieber Enumeratoren verwenden, um Zustände des Programms zu verwalten?

DRAFT

C# >

NAMESPACES

# NAMESPACES > INTRO

- WAS SIND NAMESPACES
- WOZU SIND NAMESPACES
  - Methode, Klassen zu organisieren
  - Methode, auf die Klassen schneller zuzugreifen

# NAMESPACES > INTRO

- Stammnamensraum ist global-Namespace
- `global::System` -> `System`
- Namespaces werden mit Hilfe von using-Direktiven verwendet.
- benutzerdefinierte Namensräume sind möglich (Schlüsselwort `namespace`)

# C# > CLASSES

# Objektorientierte Programmierung

- Vorteile:
  - Code wiederverwendbar
  - Bessere Strukturierung und Lesbarkeit
  - Sicherheit (public/private)
- Klassen: beschreiben Struktur von Objekten
- Aus einer Klasse beliebig viele Objekte erzeugbar:
  - `Random random1 = new Random();`
  - `Random random2 = new Random();`
- Klassen bestehen aus
  - Feldern (Membervariablen)
  - Eigenschaften (Properties)
  - Funktionen (Methoden)
  - Konstruktor/Destruktor

# Felder und Methoden

```
public class Person
{
    //Felder
    private string vorname;
    private string nachname;
    //Methoden
    public void SetVorname(string vorname)
    {
        this.vorname = vorname;
    }
    public string GetVorname()
    {
        return this.vorname;
    }
}
```



class tab tab



# Properties

- Kurzschreibweise um Zugriff auf private Membervariablen über Methoden zu definieren

```
public class Person
{
    //Properties
    public string Vorname { get; private set; }
    public string Nachname { get; private set; }

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



prop tab tab  
ctor tab tab

# Varianten von 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; }
```

# Zugriff auf Property genau steuern

```
private int alter; //Membervariable
public int Alter //dazugehörige Property
{
    get
    {
        return alter;
    }
    set
    {
        if(value > 0)
        {
            alter = value;
        }
    }
}
```



propfull tab tab

# Modifizier

- Können für Klassen, Felder, Eigenschaften und Methoden verwendet werden

Modifizier	Zugriff
<code>public</code>	Von Außerhalb der Klasse
<code>private</code>	Nur innerhalb der Klasse
<code>protected</code>	Innerhalb der Klasse und in allen abgeleiteten Klassen
<code>internal</code>	Zugriff nur innerhalb der aktuellen Assembly/des selben Namespaces

# Konstruktoren/Destruktoren

- Legt den Startzustand des Objekts nach Initialisierung fest
- Mehrere Konstruktoren pro Klasse möglich (verschiedene Parameter)

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

- Destruktor wird bei Zerstörung des Objektes aufgerufen

```
~Person()
{
    Console.WriteLine("Destruktor");
}
```



ctor tab tab  
~ tab tab

## Statische Member (Methoden/Eigenschaften)

- Als statisch markierte Methoden oder Eigenschaften gelten für die Klasse selbst und nicht für Objekte der Klasse
- Der Zugriff erfolgt über den Klassennamen
- Jede Eigenschaft existiert nur 1 Mal für die jeweilige Klasse
- Innerhalb von statischen Methoden darf nicht auf nicht-statische Methoden/Eigenschaften der Klasse zugegriffen werden
- WriteLine() und ReadLine() sind statische Methoden der Klasse Console, um auf sie zuzugreifen muss zuvor keine Instanz der Klasse Console erzeugt werden

# LAB

- QUESTIONS
- DEMO
  - Library anlegen
  - Konsumierende App anlegen
  - in der App Rechtsklick > Add > Reference

# Fragen

1. Was ist der Unterschied zwischen private und protected?
2. Welche Zugriffsmöglichkeiten innerhalb und außerhalb der Klasse bieten die folgenden Eigenschaften:

1. `public int Property1 { get; set; }`
2. `public int Property2 { get; private set; }`
3. `private int Property3 { get; set; }`
4. `public int Property4 { get; }`

3. Woran erkennt man eine Konstruktor-Deklaration?



# C# > INHERITANCE

# Vererbung von Klassen

- Alle Eigenschaften und Methoden von Basisklasse übernehmen

```
class CustomRandom : Random
{
    public int NextInclusive(int min, int max)
    {
        return base.Next(min, max+1);
    }
}
```

# Konstruktor in vererbten Klassen

```
public class Lebewesen
```

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

```
public class Mensch : Lebewesen
```

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

Ruft zuerst Basiskonstruktor auf und  
danach Konstruktor der Kindklasse

# Virtuelle Methoden

```
public class Fahrzeug
{
    public virtual string BeschreibeMich() {
        return "Ich bin ein Fahrzeug";
    }
}
public class Auto : Fahrzeug
{
    public override string BeschreibeMich()
    {
        return base.BeschreibeMich() + " und habe 4 Räder";
    }
}
```

# Abstrakte Methoden und Klassen

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

public class Fahrrad : Fahrzeug
{
    public int Luftdruck { get; set; }
    public override bool IstFahrbereit()
    {
        return (Luftdruck > 0);
    }
}
```

# C# > INHERITENCE > LAB

- Fragen

1. Von wie vielen Klassen kann eine Klasse maximal erben?
2. Was ist der Unterschied zwischen abstrakten und virtuellen Methoden?

- DEMO

- DClass zu DLibrary umbenennen
- DUseClass zu DUseLibrary umbenennen

# C# >

# INTERFACES & POLYMORPHISM

# Interfaces

- Geben Schnittstellen für Methoden und Eigenschaften vor, ohne diese zu implementieren
- Unterschied zu abstrakten Klassen: dort können einzelne Methoden bereits implementiert sein
- Eine Klasse kann beliebig viele Interfaces gleichzeitig implementieren

```
public interface IBewegbar
{
    int Räderanzahl { get; set; }

    void Bewegen();
}
```



interface tab tab



# Interface implementieren

```
class Fahrrad : Fahrzeug, IBewegbar
{
    //explizite Schnittstellenimplementierung
    void IBewegbar.Bewegen()
    {
        Console.WriteLine("Bewegen Explizit");
    }
}

class Fahrrad : Fahrzeug, IBewegbar
{
    //normale Implementierung
    public void Bewegen()
    {
        Console.WriteLine("Bewegen Implizit");
    }
}
```

# Polymorphismus

- Objekte einer bestimmten Klasse können entweder als ihr eigener Typ, als beliebiger Basistyp (Klasse von der sie abgeleitet sind) oder als Typ einer Schnittstelle, die sie implementieren, verwendet werden

```
class Fahrrad : Fahrzeug, IBewegbar { ... }
```

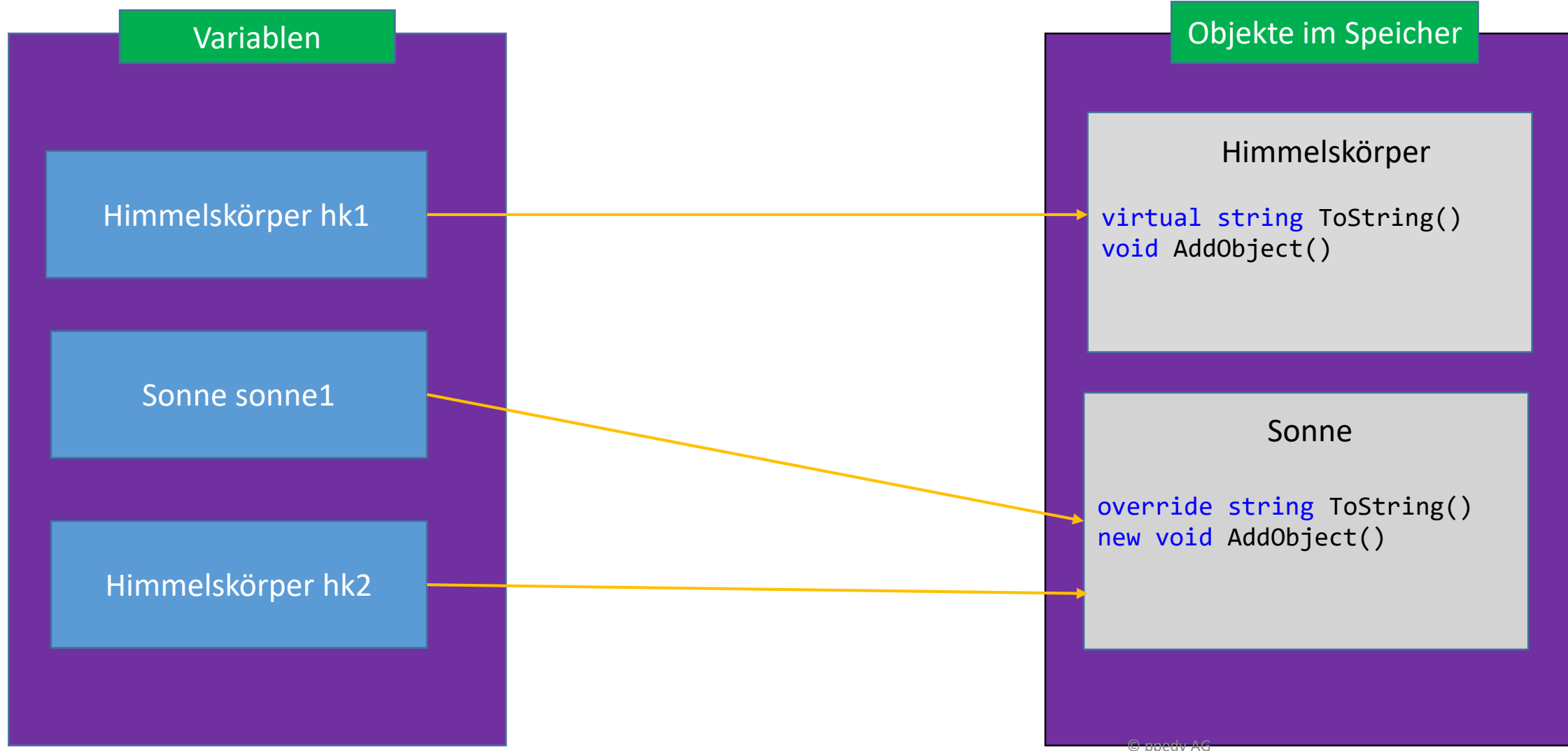
```
Fahrrad fahrrad = new Fahrrad();
```

```
Fahrzeug fahrzeug = fahrrad;
```

```
IBewegbar bewegbar = fahrrad;
```

```
Variablentyp bezeichner = new Laufzeittyp();
```

# VIRTUAL & OVERRIDE VS OVERLOAD



# Typ eines Objektes prüfen

- GetType() ermittelt den Typ des Objektes, auf den eine Variable referenziert

```
Fahrzeug fahrzeug = new Fahrrad();  
if(fahrzeug.GetType() == typeof(Fahrrad)) // true  
if(fahrzeug.GetType() == typeof(Fahrzeug)) // false
```

- is prüft ob ein Objekt Typ einer bestimmten Klasse ist, ein Interface implementiert oder Kind einer Basisklasse ist (fahrzeug steht für Fahrzeug-Objekt und fahrrad für Fahrrad-Objekt)

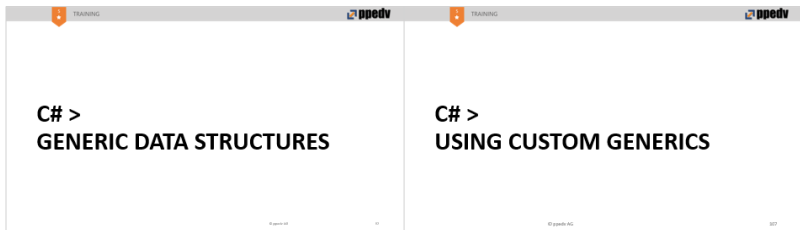
```
if(fahrzeug is Fahrrad) // false  
if(fahrzeug is Fahrzeug) // true  
if(fahrrad is IBewegbar) // true  
if(fahrrad is Fahrzeug) // true  
if(fahrrad is Object) // true
```

# Fragen

1. Wie viele Interfaces kann eine Klasse implementieren?
2. Von welcher Klasse erben alle anderen Klassen automatisch mit?
3. Welche der Folgenden Bedingungen ergibt true?

```
class Fahrzeug {...}  
class Fahrrad : Fahrzeug{...}  
Fahrzeug fz = new Fahrzeug();  
Fahrzeug fr = new Fahrrad();  
if (fz is Fahrzeug){...}  
if (fz is Fahrrad){...}  
if (fr is Fahrzeug){...}  
if (fr.GetType() == typeof(Fahrzeug)){...}
```

# C# > GENERIC



# **C# > GENERIC DATA STRUCTURES**

# Generische Datentypen

```
List<string> StringListe = new List<string>();  
StringListe.Add("1. Eintrag");  
foreach(var item in StringListe)  
{  
    Console.WriteLine(item);  
}
```

Deklaration der Klasse List

```
public class List<T> {  
    public void Add(T item);  
}
```

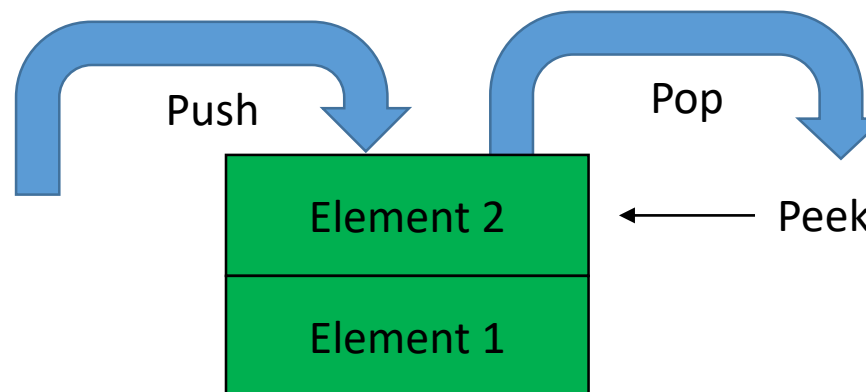
T steht für einen beliebigen Datentypen



# Spezielle Arten von Listen: Stack

- LIFO: Last In First Out

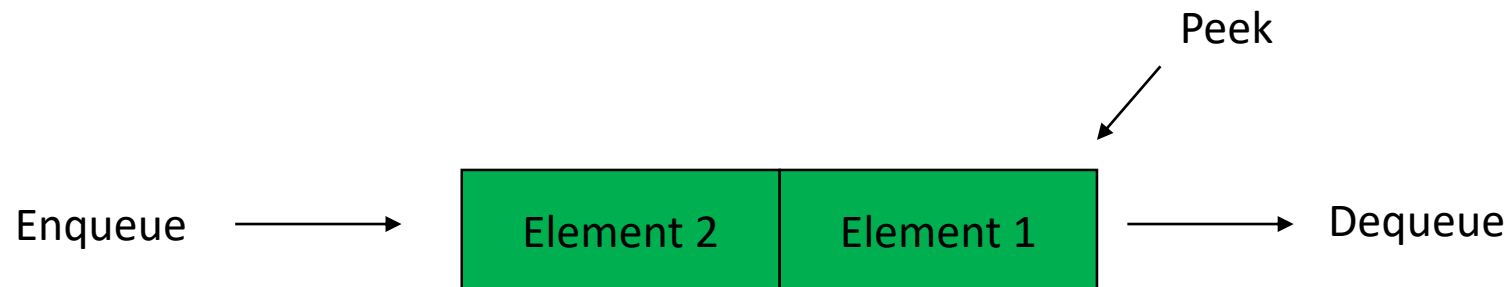
```
Stack<string> stringStack = new Stack<string>();  
stringStack.Push("Element 1");  
stringStack.Push("Element 2");  
stringStack.Peek() //=> "Element 2"  
stringStack.Pop()  //=> Element 2 herunternehmen
```



# Spezielle Arten von Listen: Queue

- FIFO: First In First Out

```
Queue<string> stringQueue = new Queue<string>();  
stringQueue.Enqueue("Element 1");  
stringQueue.Enqueue("Element 2");  
stringQueue.Peek()    //=> "Element 1"  
stringQueue.Dequeue() //=> Element 1 herausnehmen
```



# Spezielle Arten von Listen: Dictionary

- Ordnet einem Wert (**Key**) einen anderen Wert zu (**Value**)

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

Key vom Typ Integer	Value vom Typ String
5	„Element 1“
10	„Element 2“

# Spezielle Arten von Listen: Hashtable

- Ähnlich wie Dictionary, aber Key und Value können einen beliebigen Wertetyp (int, float, char, string) oder Referenztypen haben (List, Stack, eigene Klasse)
- Beim Versuch auf einen nicht-existent Index zuzugreifen, gibt es keine Fehlermeldung

```
Hashtable ht = new Hashtable();  
ht.Add("key1", "value1");  
ht.Add(20, 100);  
ht.Add(10.5, new DateTime(2012, 1, 1));  
  
if(ht[10.5]?.GetType() == typeof(DateTime)) {  
    Console.WriteLine("Jahr: " + ((DateTime)ht[10.5]).Year);  
}
```

# Kopie eines Objektes erstellen (Opt 1): Kopierkonstruktor

Objekte werden standardmäßig immer per Referenz zugewiesen

```
Person person1 = new Person("Alex");
```

```
Person person2 = person1; //beide zeigen auf das gleiche Objekt im Speicher
```

## Kopierkonstruktor definieren:

```
public Person(Person person)
{
    this.Name = person.Name;
}
```

```
Person person2 = new Person(person1);
```

## Kopie eines Objektes erstellen (Opt 2): Clone-Funktion

```
public class Person : ICloneable
{
    public string Name { get; set; }
    public Person Ehepartner { get; set; }
    public object Clone()
    {
        Person newPerson = (Person) this.MemberwiseClone();
        newPerson.Ehepartner =
(Person) this.Ehepartner?.Clone();
        return newPerson;
    }
}
```

# C# > GENERICS > LAB

- QUESTIONS
- DEMO
- EXERCISE

# Fragen

1. Welche der folgenden Typen wird als Wert und welche als Referenz übergeben?
  1. int
  2. Random
  3. bool
  4. List<>
  5. decimal
2. Wie kann man innerhalb von Funktionsparametern erzwingen dass ein Wert auf jeden Fall als Referenz und nicht als Wert übergeben wird?
3. Wie definiert man ein Dictionary welches einem Namen eine Liste von Wohnorten zuordnet?



# C# > USING CUSTOM GENERICS

# Modulübersicht

- Einsatz und Verwendung von Generics
- generische Klassen
- generische Methoden
- Constraints
- Vererbung von generischen Klassen
- Kovarianz / Kontravarianz / Invarianz

# Generics

- Platzhalter für Datentypen in Klassen und Methoden

```
void Method<T>(T obj)
{
    ...
}
```

```
Method<object>(new object());
Method(new object());
Method("Hello World");
```

- Beispiele im .NET Framework:  
List<T>, Stack<T>, Dictionary<TKey, TValue>, Comparer<T>  
Action<T>, Func<T>, EventHandler<T> ...

# Demo

Vergleich zwischen einem selbst implementierten Stack für den Datentyp „object“, einer typenspezifischen Implementierung und einer generischen Implementierung

# Constraints

- Einschränkungen für die Wahl des Datentypen
- Arten von Constraints:

```
class InterfaceConstraint<T>    where T : IDisposable { }  
class KlassenConstraint<T>     where T : Beispielklasse { }  
class WertetypConstraint<T>    where T : struct { }  
class ReferenztypConstraint<T> where T : class { }  
class KonstruktorConstraint<T> where T : new() { }
```

# Vererbung

- Ist die Basisklasse generisch, kann die Subklasse den Typenparameter übernehmen
- Soll die Subklasse nicht generisch sein, kann man einen konkreten Typen einsetzen

```
class Basisklasse<T> { }  
class SubklasseOffen<T> : Basisklasse<T> { }  
class SubklasseGeschlossen: Basisklasse<int> { }
```

```
class BasisklasseMitEinschränkungen<T> where T : class { }  
class SubklasseOffenMitEinschränkungen<T> : Basisklasse<T> where T : class { }  
class SubklasseGeschlossenMitEinschränkungen : Basisklasse<string> { }
```

## Kovarianz (out)

- Eine Instanz eines kovarianten generischen Typen ist implizit zu einem generischen Typen konvertierbar, der einen weniger spezifischen generischen Typenparameter besitzt

```
[TypeDependencyAttribute("System.SZArrayHelper")]  
public interface IEnumerable<out T> : IEnumerable
```

```
IEnumerable<Tier> tiere = Enumerable.Empty<Tier>();  
IEnumerable<Hund> hunde = Enumerable.Empty<Hund>();
```

```
tiere = hunde;  
// hunde = tiere; // Typenfehler
```

## Kontravarianz (in)

- Eine Instanz eines kontravarianten generischen Typen ist implizit zu einem generischen Typen konvertierbar, der einen spezifischeren generischen Typenparameter besitzt

```
public interface IComparer<in T>

ICComparer<Tier> tierVergleicher = null;
ICComparer<Hund> hundVergleicher = null;

hundVergleicher = tierVergleicher;
// tierVergleicher = hundVergleicher; // Typenfehler
```



# Invarianz

- Es besteht keine Beziehung zwischen invarianten Typen, da man sowohl Werte konsumieren als auch produzieren kann

```
[DefaultMember("Item")]  
[TypeDependencyAttribute("System.SZArrayHelper")]  
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
```

```
IList<Hund> hunde = new List<Hund>();  
IList<Tier> tiere = new List<Tier>();  
// tiere = hunde; // Typenfehler
```

```
// Wenn IList kovariant wäre, könnte man Elemente vom falschen Typ hinzufügen  
IList<Tier> zoo = new List<Katze>(); // Wenn das erlaubt wäre...  
zoo.Add(new Hund()); // ... wäre das auch erlaubt
```


```
// Wenn IList kontravariant wäre, könnte man Elemente vom falschen Typ zurückbekommen  
IList<Hund> zwinger = new List<Tier>(); // Wenn das erlaubt wäre ...  
Hund dog = zwinger[0]; // ... wäre das auch erlaubt
```

# **C# > DELEGATES, ANONUMOUS FUNCTIONS EVENT HANDLING**

# Delegates

- Delegate-Variablen speichern Referenzen auf andere Methoden
- Referenzen können zur Laufzeit hinzugefügt/entfernt werden
- Referenzierte Methoden müssen gleiche Signatur haben wie der Delegate-Typ

```
public delegate int MyDelegateTyp(int param1, string param2);  
public int Beispiel(int param1, string param2) {  
    return 2;  
}  
  
MyDelegateTyp myDelegate = new MyDelegateTyp(Beispiel);  
myDelegate.Invoke(5, "Test"); //Rückgabewert: 2
```



# Vordefinierte Delegate-Typen

Name	Rückgabotyp	Parameter	Beispiel für referenzierbare Methode
Action<int>	Void	1 Integer	<pre>public void myAction(int p1) {     Console.Write(p1); }</pre>
Predicate<int>	Bool	1 Integer	<pre>public bool myPredicate(int p1) {     return true; }</pre>
Func<int, int, string>	String	2 Integer	<pre>public string myFunc(int p1, int p2) {     return (p1 + p2).ToString(); }</pre>
Func<string>	String	Keine	<pre>public string myFunc() {     return "Test"; }</pre>

# Anonyme Methoden

- Anonym bedeutet: Methode ohne Namen
- Delegate-Variablen Methoden zuweisen, ohne diese vorher zu deklarieren

```
Func<int, int> myDelegate = new MyDelegates(  
    delegate(int p1) {  
        return 5;  
    }  
);
```

- Kurzschreibweise:

```
Func<int, int> myDelegate = delegate(int p1) {  
    return 5;  
};
```

# EventHandler definieren

- Ein Event ist eine Delegate-Variable, an der sich beliebig viele Methoden (EventListener) anmelden können
- unter C# wird für ein Event-Delegate per Konvention der Typ EventHandler benutzt
- Beim Auslösen des Events werden alle referenzierten Methoden aufgerufen

```
event EventHandler onClick +=  
    delegate (Object sender, EventArgs args)  
    {  
        Console.WriteLine("EventBehandlung");  
    };  
  
onClick(this, EventArgs.Empty); //Aufruf des Events
```

# Timer verwenden

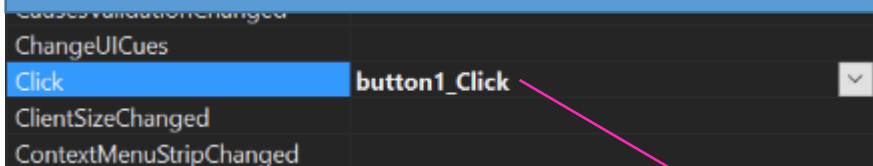
```
using System.Timers;

Timer timer = new Timer();
timer.Elapsed += new ElapsedEventHandler(Countdown);
timer.Interval = 1000;
Timer.Start();

void Countdown(Object sender, ElapsedEventArgs args)
{
    countdownNumber--;
    Console.WriteLine(countdownNumber);
    if(countdownNumber <= 0)
    {
        timer.Stop();
    }
}
```

# Events am Beispiel von WindowsForms

## Properties von button1



## Form1.Designer.cs

```
this.button1.Name = "button1";  
this.button1.Size = new System.Drawing.Size(244, 144);  
this.button1.TabIndex = 1;  
this.button1.Text = "Button 1";  
this.button1.UseVisualStyleBackColor = true;  
this.button1.Click += new System.EventHandler(this.button1_Click);  
//
```

## Form1.cs

```
31  
32  
33 private void button1_Click(object sender, EventArgs e)  
34 {  
35  
36 }  
37  
38  
39
```



# Partielle Klassen am Beispiel WindowsForms

- jedes Formular in einer WindowsForms-Anwendung ist ein Objekt einer von der Klasse Form abgeleiteten Klasse, z.B. Form1
- die Klasse Form1 wird in 2 verschiedenen Dateien deklariert, dies ist möglich durch das Schlüsselwort „partial“

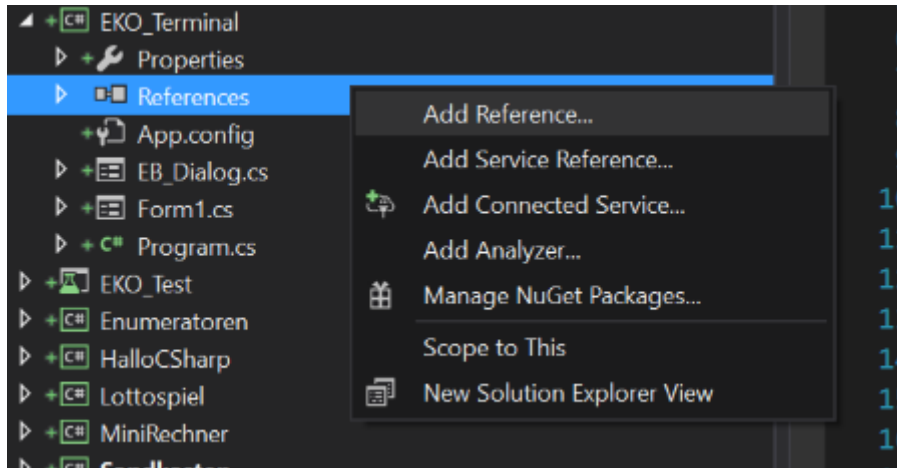
Form1.cs

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

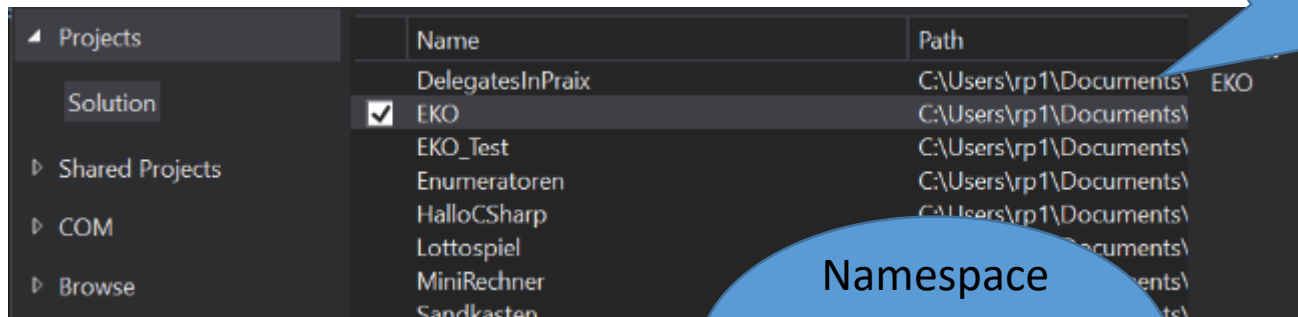
Form1.Designer.cs

```
partial class Form1
{
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.IContainer
```

# Klassen aus anderen Projekt einbinden



Referenz zum  
Projekt  
hinzufügen



Einzubindende  
Projekte  
auswählen

```
using EKO;  
using System;  
using System.Collections.Generic;  
using System.ComponentModel;  
using System.Data;
```

Namespace  
per using in  
jedem Codefile  
einbinden

# LAB

- QUESTIONS
- DEMO
- EXERCISE

# Fragen

- Welche Methode passt zu welchem Delegate-Typen?

```
void methode1(int x)
int methode2(string y)
void methode3(bool x, bool y)
string methode4(List<string> x)
void methode5()
```

```
Func<List<string>, string> delegate1;
Action<bool, bool> delegate2;
Func<string, int> delegate3;
Action delegate4;
Action<int>
```

# C# > ERROR HANDLING

# Exception-Handling

- Mittels try/catch Fehlerfälle in Anwendung abfangen:

```
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");
}
```



try tab tab

# Eigene Exception-Klasse mit Fehlermeldung

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

- Fehler werfen:

```
throw new MyException("Errormessage");
```

- Fehler abfangen:

```
catch (MyException e)
{
    Console.WriteLine(e.Message);
}
```

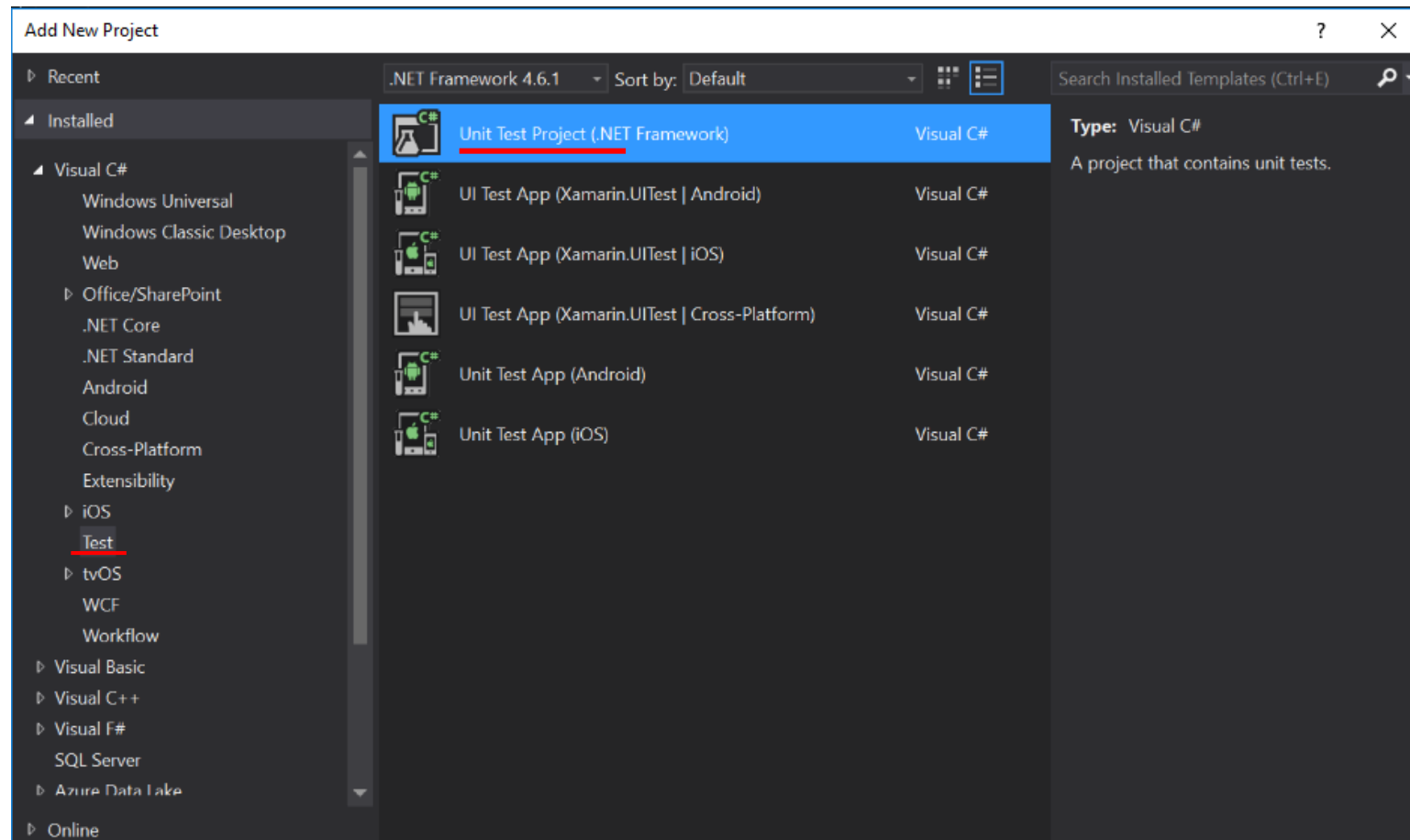
# Fragen

- Wozu benötigt man den Finally-Block in einem try-catch-Block?
- Was passiert wenn man innerhalb eines catch-Blockes erneut eine Exception wirft?



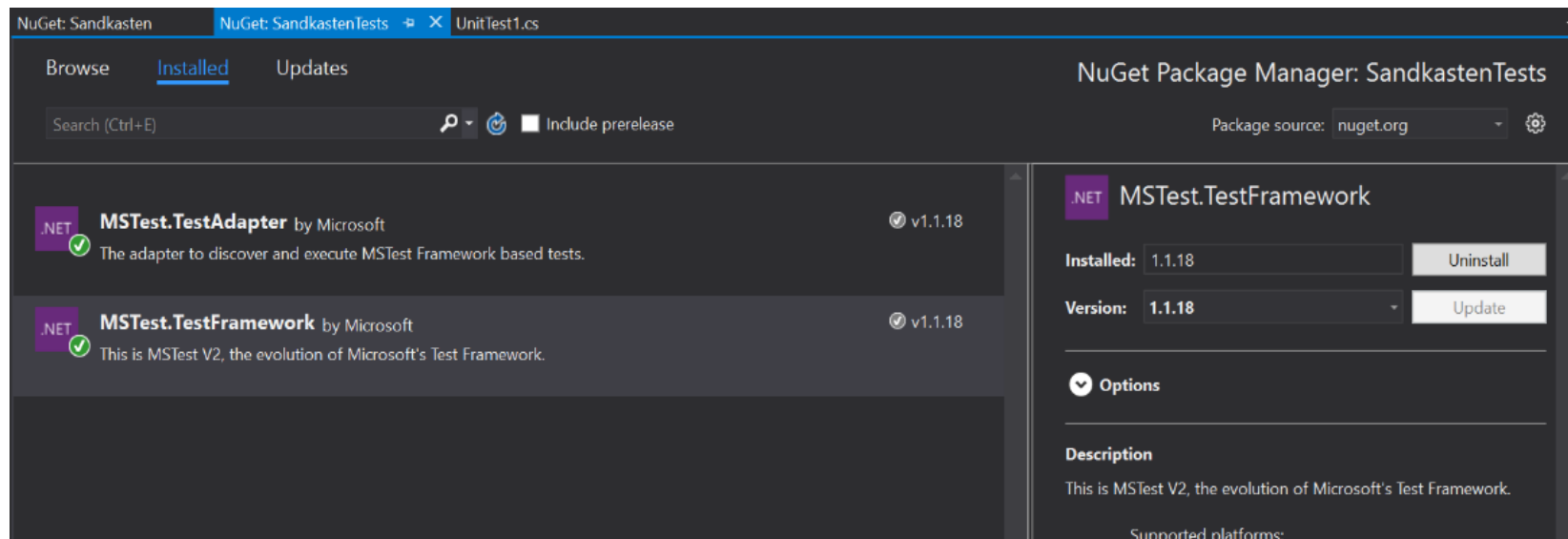
# C# & UNIT TESTS

# UnitTests - Testprojekt anlegen



# UnitTests – NuGet Packete installieren

- MSTest.TestFramework und MSTest.TestAdapter über NuGet Package Manager für das Projekt installieren



- Schließlich noch das zu testende Projekt unter References hinzufügen

# UnitTests – Testklasse schreiben

```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
        KlasseAusProjekt objekt = new KlasseAusProjekt();
        int result = objekt.ZuTestendeMethode(4);
        Assert.IsNotNull(result);
        Assert.AreEqual(2, result);
    }
}
```

- Test laufen lassen über Test->Run->All Tests

# C# & UNIT TESTS > LAB

# C# > FILE ACCESS

# Dateien schreiben/lesen

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

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

# Dateidialog anzeigen

- Dialog zum Datei öffnen bzw. speichern

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

- FileName: ausgewählte (bzw. vorbelegter) Dateiname
- Filter: welche Dateitypen sollen auswählbar sein
- zum Datei Öffnen analog die Klasse OpenFileDialog verwenden



# Objekte als JSON serialisieren

- Serialisierung: Beliebiges Objekt in einen String konvertieren
- Vorteil: Der Zustand eines Objektes kann als String in eine Datei geschrieben und später wieder ausgelesen und zurückkonvertiert (deserialisiert) werden

```
using Newtonsoft.Json; //vorher per NuGet installieren
var daten = new List<string> { „Wert1“, „Wert2“, „Wert3“ };
//.NET-Objekt in JSON-String konvertieren
string result = JsonConvert.SerializeObject(daten);
sw.Write(result);
//JSON-String in .NET-Objekt umwandeln
objekt = JsonConvert.DeserializeObject<List<string>>(sr.ReadToEnd());
```

# Fragen

- Warum muss man beim Deserialisieren das Format, in das deserialisiert werden soll, mit angeben aber beim Serialisieren nicht?
- Wie kann man beim OpenFileDialog die Auswahl auf Text-Dateien einschränken?

**C# >  
LINQ**

# C# > LINQ > #INTRO

- WAS IST LINQ
  - Language Integrated Query / Sprachintegrierte Abfrage
- WOZU IST LINQ
- OFFICIAL DOCS
  - <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/linq/>
  - <https://docs.microsoft.com/de-de/dotnet/csharp/linq/>

## *Exkurs:* LINQ

- Language Integrated Query
- Abstrahierte Abfragesprache auf Objektmengen
- Direkter Bestandteil von C#
  
- 101 LINQ Beispiele
  - <https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b>

# Web API und HTTP

## • Exkurs LINQ

- Language Integrated Query
- Abfragesprachen auf Objektmengen
- Direkter Bestandteil von C# und VB.NET
- Bietet die Möglichkeit, Abfragen abstrakt zu beschreiben, die später durch entsprechende Provider in konkrete Abfragen für die entsprechenden Zielsysteme überführt werden.
- 101 LINQ Beispiele:
  - <https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b>

# LINQ Basics

- Typisierte Daten Aggregationen
  - Mehrzeilig
  - Einzeilig ->auch mehrere Zeilen
- Where
  - Find Key
- Group by
- Sort
- Select
  - New oder Anonym
- Join

```
var queryLondonCustomers = from cust in customers
where cust.City == "London"
select cust;
```

```
var studentNames = studentList.Where(s => s.Age > 18)
    .Select(s => s)
    .Where(st => st.StandardID > 0)
    .Select(s => s.StudentName);
```

# LINQ Basics

- Typisierte Daten Aggregationen
  - Mehrzeilig
  - Einzeilig -> auch mehrere Zeilen
- Where
  - Find Key
- Group by
- Sort
- Select
  - New oder Anonym
- Join

```
var queryLondonCustomers = from cust in customers
where cust.City == "London"
select cust;
```

```
var studentNames = studentList.Where(s => s.Age > 18)
    .Select(s => s)
    .Where(st => st.StandardID > 0)
    .Select(s => s.StudentName);
```



# ASP.NET CORE & LINQ > INTRO

```
// using System.Linq;  
var movies =    from m in _context.Movie  
                select m;
```

# Linq (Language Integrated Query)

- Sammlung von Erweiterungsmethoden zur Abfrage und Sortierung von Elementen innerhalb von Aufzählungstypen (Lists, Arrays, Stacks, Queues ...)
- An SQL-Syntax angelehnt

```
using System.Linq;  
string [] Städte = new string[] { "Leipzig", "Hamburg", "Hannover" };  
var mitH = from stadt in Städte  
           where stadt.StartsWith("H")  
           select stadt;
```

Ergebnis: String-Liste mit Hamburg und Hannover

```
//Linq-Syntax entspricht dieser Schreibweise  
var mitH = Städte.Where(stadt => stadt.StartsWith("H"));
```

# Wichtige Linq-Befehle

Befehl	Funktion	Beispiel
From	Datenquelle festlegen	<code>from item in StringListe</code>
Select	Form des Ergebnisses	<code>from item in StringListe select "new " + item</code>
Where	Ergebnisse filtern	<code>from item in StringListe where item.EndsWith("a")</code>
OrderBy	Elemente absteigend sortieren	<code>from item in StringListe orderby item.Length</code>

# LINQ > CLASS QUERYABLE > METHODS #OVERVIEW

- Aggregate() 3
- All() 1
- Any() 2
- Append() 1
- AsQueryable() 2
- Average() 20
- Cast() 1
- Concat() 1
- Contains() 2
- Count() 2
- DefaultIfEmpty() 2
- Distinct() 2
- ElementAt() 1
- ElementAtOrDefault() 1
- Except() 2
- First() 2
- FirstOrDefault() 2
- GroupBy() 8
- GroupJoin() 2
- Intersect() 2
- Join() 2
- Last() 2
- LastOrDefault() 2
- LongCount() 2
- Max() 2
- Min() 2
- OfType() 1
- OrderBy() 2
- OrderByDescending() 2
- Prepend() 1
- Reverse() 1
- Select() 2
- SelectMany() 4
- SequenceEqual() 2
- Single() 2
- SingleOrDefault() 2
- Skip() 1
- SkipLast() 1
- SkipWhile() 2
- Sum() 20
- Take() 1
- TakeLast() 1
- TakeWhile() 2
- ThenBy() 2
- ThenByDescending() 2
- Union() 2
- Where() 2
- Zip() 2

# C# > LINQ > PACKAGE #OVERVIEW

- System.Linq. ...
  - Enumerable
  - EnumerableExecutor
  - EnumerableExecutor<T>
  - EnumerableQuery
  - EnumerableQuery<T>
  - IGrouping<TKey,TElement>
  - ILookup<TKey,TElement>
  - IOrderedEnumerable<TElement>
  - IOrderedQueryable
  - IOrderedQueryable<T>
  - IQueryable
- IQueryable<T>
- IQueryProvider
- Lookup<TKey,TElement>
- OrderedParallelQuery<TSource>
- ParallelEnumerable
- ParallelExecutionMode
- ParallelMergeOptions
- ParallelQuery
- ParallelQuery<TSource>
- Queryable