



Generics

Modulübersicht

- Einsatz und Verwendung von Generics
- generische Klassen
- generische Methoden
- Constraints
- Vererbung von generischen Klassen

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 WertetyConstraint<T>     where T : struct { }  
class ReferenztyConstraint<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> { }
```

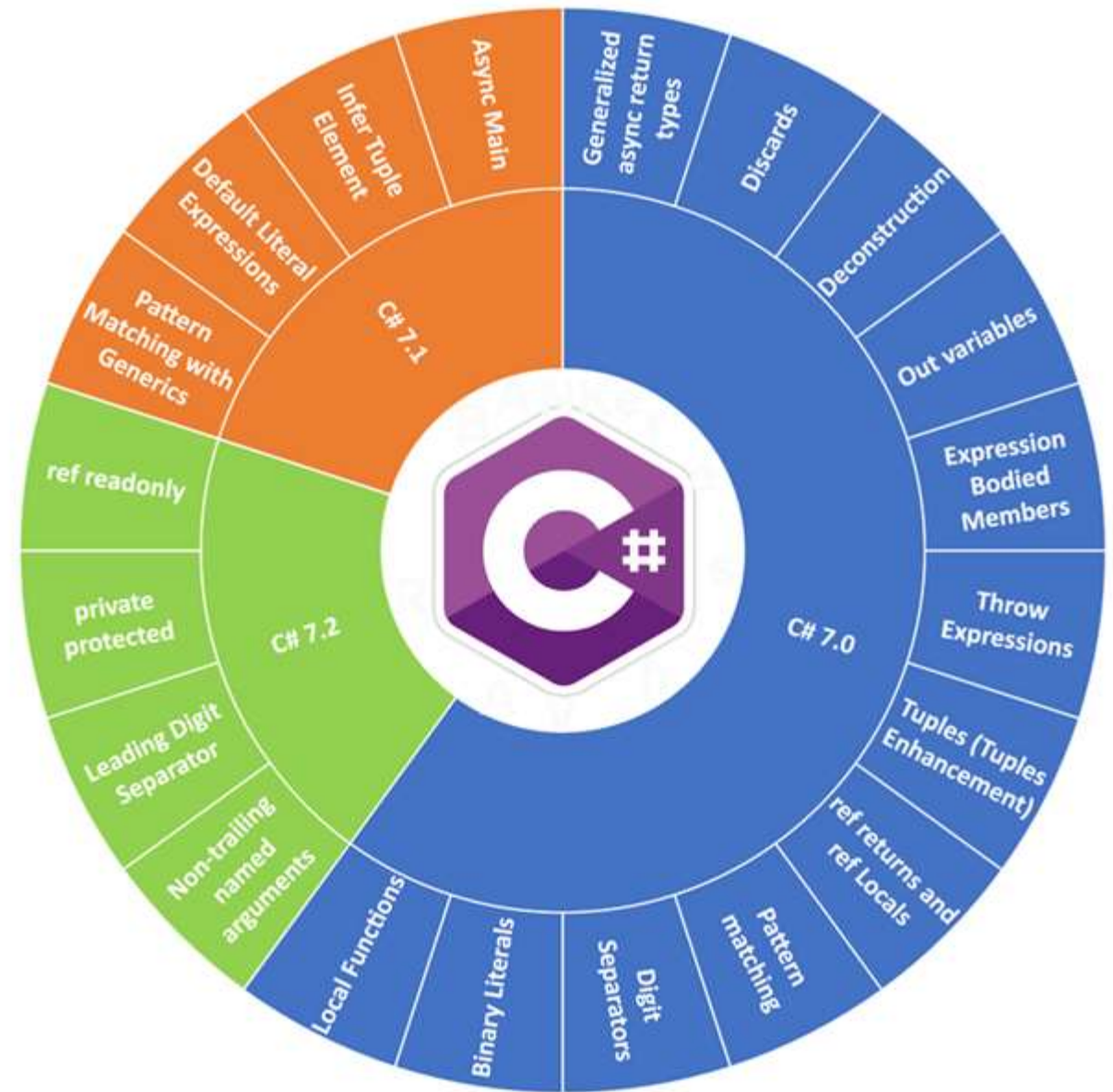
```
class BasisklasseMitEinschränkungen<T> where T : class { }  
class SubklasseOffenMitEinschränkungen<T> : Basisklasse<T> where T : class { }  
class SubklasseGeschlossenMitEinschränkungen : Basisklasse<string> { }
```



Neue Sprachfeatures in C#

Modulübersicht

- C# 7.0
- C# 7.1
- C# 7.2
- C# 7.3
- C# 8.0
- C# 9.0
- C# 10.0...



Defaults

Target framework	version	C# language version default
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

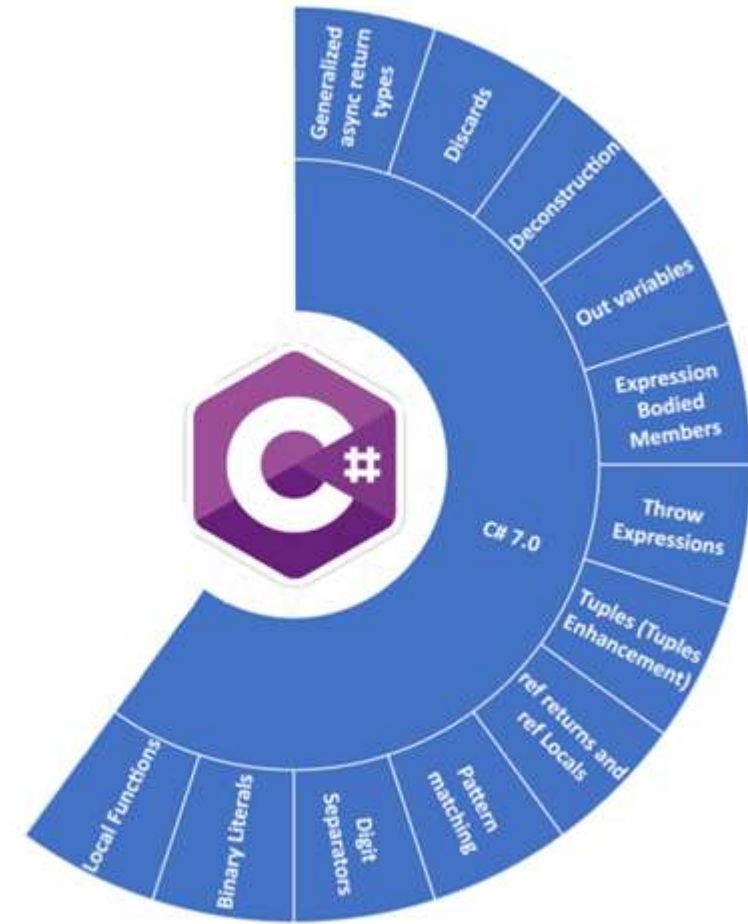
```
<PropertyGroup>  
  <LangVersion>7.2</LangVersion>  
</PropertyGroup>
```



C# 7.0

C# 7.0 – Überblick

- out Variablen
- Pattern Matching
 - is – Operator mit Patterns
 - switch
- Tupel
- lokale Funktionen
- Separator für numerische Werte
- Return by Reference
- ValueTask<T>
- Mehr Expression Bodied Members
- Exceptions in Expressions werfen



C# 7.0 – out

```
private void IntCheck()
{
    string eingabe = "12345";
    int ausgabe; // <- deklaration erforderlich !
    if(Int32.TryParse(eingabe,out ausgabe))
    {
        Console.WriteLine(ausgabe);
    }
}
```

```
private void IntCheck()
{
    string eingabe = "12345";
    if(Int32.TryParse(eingabe,out var ausgabe))
    {
        Console.WriteLine(ausgabe);
    }
}
```

C# 7.0 – Pattern Matching

```
private void SchreibSterneInDieKonsole(object o)
{
    if (o is null) // Constant pattern "null"
        return;
    if (o is int i) // Typenpattern
        Console.Write(new string('*', i));
}
```

C# 7.0 – Pattern Matching mit Switch

```
private void MusterabgleichMitSwitch(Grafik g)
{
    switch (g)
    {
        case Kreis k:
            Console.WriteLine($"Kreis mit dem Radius {k.Radius}");
            break;
        case Rechteck q when (q.Länge == q.Breite):
            Console.WriteLine($"Quadrat mit der Seitenlänge {q.Länge}");
            break;
        case Rechteck r:
            Console.WriteLine($"Rechteck mit der Länge {r.Länge} und Breite {r.Breite}");
            break;
        default:
            Console.WriteLine("Unbekannte Grafik");
            break;
    }
}
```

C# 7.0 – Tupel

```
class Person
{
    public string Vorname { get; set; }
    public string ZweiterVorname { get; set; }
    public string Nachname { get; set; }

    public (string, string, string) VollenNamenAusgabe()
    {
        return (Vorname, !string.IsNullOrEmpty(ZweiterVorname) ? ZweiterVorname : string.Empty, Nachname);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person { Vorname = "Max", Nachname = "Mustermann" };
        var name = p.VollenNamenAusgabe();
        Console.WriteLine($"{name.Item1} {name.Item2} {name.Item3}");
    }
}
```

C# 7.0 – Tupel

```
class Person
{
    public string Vorname { get; set; }
    public string ZweiterVorname { get; set; }
    public string Nachname { get; set; }

    public (string Vorname, string ZweiterVorname, string Nachname) VollenNamenAusgabe()
    {
        return (Vorname, !string.IsNullOrEmpty(ZweiterVorname) ? ZweiterVorname : string.Empty, Nachname);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person { Vorname = "Max", Nachname = "Mustermann" };
        var name = p.VollenNamenAusgabe();
        Console.WriteLine($"{name.Vorname} {name.ZweiterVorname} {name.Nachname}");
    }
}
```


C# 7.0 – Tupel Dekonstruktion

```
public void Main()
{
    Person p = new Person { Vorname = "Max", Nachname = "Mustermann" };

    (string vn, string zv, string nn) = p.VollenNamenAusgeben();    // Variante 1

    (var vn2, var zv2, var nn2) = p.VollenNamenAusgeben();        // Variante 2

    var (vn3, zv3, nn3) = p.VollenNamenAusgeben();                // Variante 3

    string vorname, zweiterVorname, nachname;
    (vorname, zweiterVorname, nachname) = p.VollenNamenAusgeben(); // Variante 4

    Console.WriteLine($"{vorname} {zweiterVorname} {nachname}");
}
```

C# 7.0 – Dekonstruktion von Typen

```
class Kunde
{
    public int ID { get; set; }
    public string Name { get; set; }
    public bool Stammkunde { get; set; }

    public void Deconstruct(out int ID, out string Name, out bool Stammkunde)
    {
        ID = this.ID;
        Name = this.Name;
        Stammkunde = this.Stammkunde;
    }
}

public void Main()
{
    Kunde k = new Kunde { ID = 12, Name = "Max Mustermann", Stammkunde = true };
    var (id, name, stammkunde) = k;
    Console.WriteLine($"{id}: {name} ist Stammkunde:{stammkunde}");
}
```

C# 7.0 – Lokale Funktionen

```
public void WertVerdoppeln()
{
    int wert = 123;
    void LokaleFunktion()
    {
        wert *= 2;
    }
    LokaleFunktion();
    Console.WriteLine(wert);
}
```

```
public int Fibonacci(int x)
{
    return Fib(x - 1).AktuellerWert;

    (int AktuellerWert, int VorherigerWert) Fib(int i)
    {
        if (i == 0) return (1, 0);
        var (p, pp) = Fib(i - 1);
        return (pp + p, p);
    }
}
```

C# 7.0 – Literale

```
public void Literale()
{
    int hex = 0x00FF00;
    byte bin = 0b00001111;
    byte bin2 = 0b0000_1111;
    int hundertdreiundzwanzig = 1_____2_____3;
}
```

C# 7.0 – Rückgabe per Referenz

```
public ref int Zahlensuche(int gesuchteZahl,int[]zahlen)
{
    for (int i = 0; i < zahlen.Length; i++)
    {
        if (zahlen[i] == gesuchteZahl)
            return ref zahlen[i];
    }
    throw new IndexOutOfRangeException();
}

public void Main()
{
    int[] zahlen = { 5, 7, 432, 567, -98, 3, 2 };
    ref int position = ref Zahlensuche(3, zahlen); // Alias für Index 5
    position = 100_000_000;
    Console.WriteLine(zahlen[5]);
}
```

C# 7.0 – ValueTask<T>

- C# 6.0 und darunter erlaubt nur void, Task oder Task<T> für asynchrone Methoden
- ValueTask<T> ist eine alternative für die Referenztypen Task und Task<T> und kann in einigen Situationen die Performance verbessern

C# 7.0 – Mehr Expression Bodied Members

```
class Person
{
    public Person(string Name) => this.Name = Name;
    ~Person() => Name = null;

    private string name;
    public string Name
    {
        get => name;
        set => name = value;
    }
}
```

C# 7.0 – Exceptions in Expressions werfen

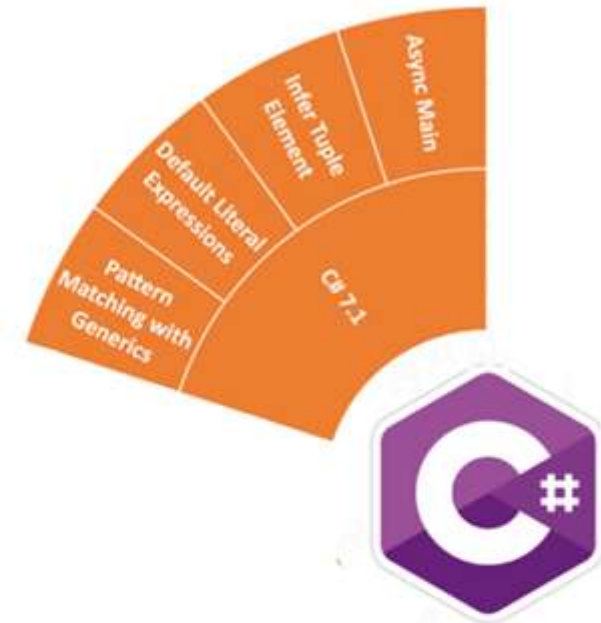
```
class Person
{
    public Person(string Name) => this.Name = Name ?? throw new ArgumentException();
    ~Person() => Name = null;

    public string Name { get; set; }
    public string GetVorname()
    {
        string[] geteilterName = Name.Split(' ');
        return (geteilterName.Length > 0) ? geteilterName[0] : throw new InvalidOperationException();
    }
    public string GetNachname() => throw new NotImplementedException();
}
```


C# 7.1

C# 7.1 – Überblick

- async Main
- Abgeleitete Tupelelementnamen
- default Literale
- Pattern Matching mit Generics



C# 7.1 – async Main

```
class Program
{
    static async Task Main()
    {
        await Task.Delay(2000);

        Console.WriteLine("Hallo Welt");
    }
}
```

C# 7.1 – Abgeleitete Tupelelementnamen

```
static void Main()
{
    int zähler = 5;
    string label = "Farben in diesem Bild";
    var tupel = (zähler: zähler, label: label); // C# 7.0

    var tupel = (zähler, label);                // C# 7.1
}
```

C# 7.1 – default Literale

```
static int Main()
{
    Func<string, bool> whereFunktion = default(Func<string, bool>); // alt

    Func<string, bool> whereFunktion = default;                       // neu
    Func<int> defaultValue = () => default;
    int i = default;
    int[] zahlen = { default, 1, 2, 3 };

    if (i is default)
        Console.WriteLine("i hat den Standardwert");

    return default;
}

private void OptionalerParameter(int i = default)
{
}
}
```

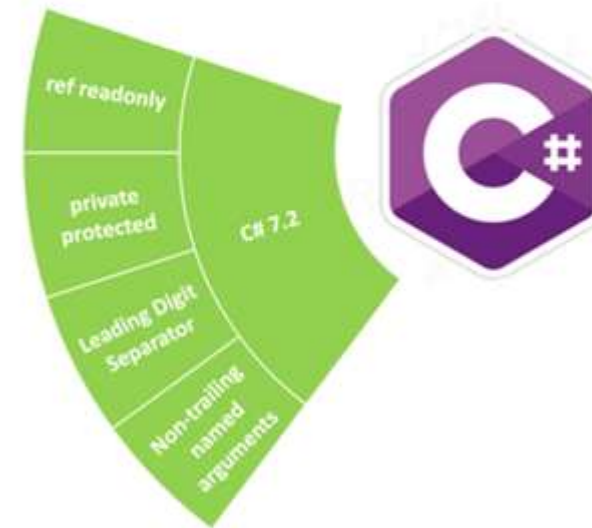
C# 7.1 – Pattern Matching mit Generics

```
static void Print<T>(T input)
{
    switch (input)
    {
        case int i:
            Console.WriteLine($"Integer: {i}");
            break;
        case string s:
            Console.WriteLine($"String: {s}");
            break;
        default:
            Console.WriteLine("Unbekannter Typ");
            break;
    }
}
```

C# 7.2

C# 7.2 – Überblick

- readonly ref
- private protected
- Verbesserung beim Literal-Separator
- Verbesserung bei benannten Argumenten



C# 7.2 – readonly ref

- ref: Variable wird als Referenz übergeben
- out: Variable wird als Referenz übergeben und muss einen neuen Wert zugewiesen bekommen
- in: Variable wird als Referenz übergeben, ist aber readonly !

```
public static void InParameter(in int x)
{
    int y = x++;
}
```

[?] (parameter) in int x

Cannot assign to variable 'in int' because it is a readonly variable

C# 7.2 – private protected

```
class Mitarbeiter
{
    public          int ID { get; set; }
    protected      string Name { get; set; }
    internal        DateTime Geburtsdatum { get; set; }
    private         decimal Gehalt { get; set; }
    protected internal int AbteilungsID { get; set; }

    private protected string Projektname { get; set; }
}
```

C# 7.2 – Verbesserung beim Literal-Separator

```
public void Literale()
{
    int hex = 0x00FF00;
    byte bin = 0b00001111;
    byte bin2 = 0b_0000_1111;
    int hundertdreieundzwanzig = 1____2____3;
}
```

```
public void Literale_CSharp7_2()
{
    int hex = 0x_00FF00;
    byte bin = 0b__0000__1111;
    // ...
}
```

C# 7.2 – Verbesserung bei benannten Argumenten

```
public static long Summe(int zahl1, int zahl2 = default, int zahl3 = default)
{
    return zahl1 + zahl2 + zahl3;
}

public static void Main()
{
    int z1 = 5, z2 = 10, z3 = 15;

    long ergebnis  = Summe(z1, z2, z3);           // geht
    long ergebnis2 = Summe(z1, zahl2: z2, zahl3:z3); // geht
    long ergebnis3 = Summe(z1, zahl2: z2, z3);    // geht seit C# 7.2
}
```

C# 7.3

C# 7.3 – Überblick:

- Neuzuweisung von lokalen ref – Variablen
- Arrayinitialisierer für stackalloc – Arrays
- Neue Constraints für generische Klassen
- Tupel unterstützen nun == und !=
- Attribute für automatisch generierte Felder

Neuzuweisung von lokalen ref - Variablen

```
SehrGroßesStruct sgs = new SehrGroßesStruct();  
SehrGroßesStruct sgs2 = new SehrGroßesStruct();  
  
ref SehrGroßesStruct referenz = ref sgs;  
referenz = ref sgs2;
```

Arrayinitialisierer für stackalloc – Arrays

```
unsafe
```

```
{
```

```
    int* pointerAufArray = stackalloc int[5] { 10, 20, 30, 40, 50 };
```

```
    int* pointerAufArray2 = stackalloc int[] { 10, 20, 30, 40, 50 };
```

```
    Span<int> beliebigeMenge = stackalloc [] { 10, 20, 30, 40, 50 };
```

```
}
```


Neue Constraints für generische Klassen

```
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() { }
```

```
class EnumConstraint<T>           where T : Enum { };  
class DelegateConstraint<T>      where T : Delegate { };  
class UnmanagedConstraint<T>     where T : unmanaged { };
```

Tupel unterstützen nun == und !=

```
var tupel1 = (12, "Hallo Welt");  
var tupel2 = (12, "Hallo Welt");  
  
if(tupel1 == tupel2)  
    Console.WriteLine("Gleich");
```

Attribute für automatisch generierte Felder

```
[Serializable]
class MeineKlasse
{
    [field: NonSerialized]
    public string Vorname { get; set; }
}
```




C# 8.0

C# 8.0 – Überblick:

- Readonly Member
- Mehr Expressions für gängige Patterns -> Switch/Property/Position/Tupel - Pattern
- Using Deklaration
- Statische lokale Funktionen
- Asynchrone Streams
- Indizes und Ranges
- Null-Coalescing Zuweisung
- Verbatim-Strings und String-Interpolation
- Referenztypen ohne null
- Neue Features für Interfaces

Readonly Member

```
public struct Punkt
{
    3 references
    public int X { get; set; }
    3 references
    public int Y { get; set; }
    0 references
    public readonly double Entfernung => Math.Sqrt(X * X + Y * Y); // Verändert keine Werte
    0 references
    public readonly void Berechne(int xOffset, int yOffset) // Verändert Werte
    {
        X += xOffset;
        Y += yOffset;
    }
}
```

 `int Punkt.Y { get; set; }`
Cannot assign to 'Y' because it is read-only

Mehr Expressions für gängige Patterns

- Switch - Pattern

```
public string HeutigerTag()
{
    switch (DateTime.Now.DayOfWeek)
    {
        case DayOfWeek.Monday:
            return "Montag";
        case DayOfWeek.Tuesday:
            return "Dienstag";
        case DayOfWeek.Wednesday:
            return "Mittwoch";
        case DayOfWeek.Thursday:
            return "Donnerstag";
        case DayOfWeek.Friday:
            return "Freitag";
        default:
            return "Wochenende";
    }
}
```

```
public string HeutigerTag()
{
    return DateTime.Now.DayOfWeek switch
    {
        DayOfWeek.Monday => "Montag",
        DayOfWeek.Tuesday => "Dienstag",
        DayOfWeek.Wednesday => "Mittwoch",
        DayOfWeek.Thursday => "Donnerstag",
        DayOfWeek.Friday => "Freitag",
        _ => "Wochenende"
    };
}
```

Mehr Expressions für gängige Patterns

- Property - Pattern

```
public decimal Bonuszahlung(Person mitarbeiter, decimal grundbetrag)
{
    return mitarbeiter switch
    {
        { Alter: 20 } => grundbetrag * 0.50m,
        { Alter: 21 } => grundbetrag * 0.60m,
        { Alter: 22 } => grundbetrag * 0.70m,
        { Alter: 23 } => grundbetrag * 0.80m,
        { Alter: 24 } => grundbetrag * 0.90m,
        { Alter: 25 } => grundbetrag,
        _ => grundbetrag * 2.0m // Alle anderen
    };
}
```


Mehr Expressions für gängige Patterns

- Position - Pattern

```
public decimal Bonuszahlung(Person mitarbeiter, decimal grundbetrag)
{
    return mitarbeiter switch
    {
        var (vn, nn, a) when vn == "Hannes" => grundbetrag * 5.00m,
        var (vn, nn, a) when a < 20 => grundbetrag * 1.25m,
        var (vn, nn, a) when a < 30 => grundbetrag * 1.50m,
        var (vn, nn, a) when a < 40 => grundbetrag * 2.00m,
        var (vn, nn, a) when a < 50 => grundbetrag * 2.50m,
        _ => grundbetrag * 3.0m
    };
}
```

Mehr Expressions für gängige Patterns

- Tupel - Pattern

```
public static string SchereSteinPapier(string element1, string element2)
=> (element1, element2) switch
{
    ("stein", "papier") => "Papier gewinnt",
    ("papier", "stein") => "Papier gewinnt",
    ("stein", "schere") => "Stein gewinnt",
    ("schere", "stein") => "Stein gewinnt",
    ("papier", "schere") => "Schere gewinnt",
    ("schere", "papier") => "Schere gewinnt",
    (_, _) => "Unentschieden"
};
```

Using Deklaration

```
public int SchreibeInEineDateiCSharp7X(string[] zeilen)
{
    int geschriebeneZeilen = 0;

    using (var datei = new StreamWriter("Demo.txt"))
    {
        foreach (string line in zeilen)
        {
            datei.WriteLine(line);
            geschriebeneZeilen++;
        }
    } // .Dispose()

    return geschriebeneZeilen;
}
```

```
public int SchreibeInEineDateiCSharp8(string[] zeilen)
{
    using var datei = new StreamWriter("Demo.txt");

    int geschriebeneZeilen = 0;
    foreach (string line in zeilen)
    {
        datei.WriteLine(line);
        geschriebeneZeilen++;
    }

    return geschriebeneZeilen;
    // .Dispose()
}
```

Statische lokale Funktionen

```
public int Berechnung()  
{  
    int y = 5;  
    int x = 10;  
    return Add(x, y);  
  
    static int Add(int links, int rechts) => links + rechts;  
}
```

Asynchrone Streams

```
public static async IEnumerable<int> GeneriereZahlen()  
{  
    for (int i = 0; i < 20; i++)  
    {  
        await Task.Delay(100);  
        yield return i;  
    }  
}
```

```
public static async void GebeZahlenAus()  
{  
    await foreach (var zahl in GeneriereZahlen())  
    {  
        Console.WriteLine(zahl);  
    }  
}
```

Indizes und Ranges

- Neue Syntax für das Arbeiten mit Sequenzen
 - System.Index: Index in einer Sequenz
 - ^ - Operator: Index relativ zum Ende der Sequenz
 - System.Range: Teilbereich einer Sequenz
 - .. – Operator: Beginn und Ende eines Teilbereiches

```
var wörter = new string[]
{
    "Franz",           // Index vom Anfang   Index vom Ende
    "jagt",             // 0                               ^9
    "im",               // 1                               ^8
    "komplett",         // 2                               ^7
    "verwahrlosten",    // 3                               ^6
    "Taxi",             // 4                               ^5
    "quer",             // 5                               ^4
    "durch",           // 6                               ^3
    "Bayern"           // 7                               ^2
};                    // 8                               ^1
                    // 9 (wörter.Length)       ^0
```

Indizes und Ranges

- Neue Syntax für das Arbeiten mit Sequenzen
 - System.Index: Index in einer Sequenz
 - ^ - Operator: Index relativ zum Ende der Sequenz
 - System.Range: Teilbereich einer Sequenz
 - .. – Operator: Beginn und Ende eines Teilbereiches

```
// Franz jagt im komplett verwahrlosten Taxi quer durch Bayern
```

```
Console.WriteLine($"Das letzte Wort ist {wörter[^1]}");  
var imKomplettVerwahrlostenTaxi = wörter[2..6]; // exkl wörter[6]  
var durchBayern = wörter[^2..^0]; // exkl wörter[^0]  
var alles = wörter[..];  
var ersterTeil = wörter[..4]; // Franz jagt im komplett  
var letzterTeil = wörter[6..]; // quer durch Bayern
```

```
Range phrase = 1..4;  
var text = wörter[phrase]; // jagt im komplett
```

Null-Coalescing Zuweisung

- Zuweisungsoperation nur dann ausführen, wenn der linke Operator null ist

```
List<int> meineZahlen = null;  
int? zahl = null;
```

```
meineZahlen ??= new List<int>(); // Wenn null, dann neue Liste erstellen  
meineZahlen.Add(zahl ??= 17);    // Wenn null, dann 17 zuweisen  
meineZahlen.Add(zahl ??= 20);    // Wenn null, dann 20 zuweisen
```

```
Console.WriteLine(string.Join(" ", meineZahlen));  
// > 17 17
```


Verbatim-Strings und String-Interpolation

```
Console.WriteLine($"Diese Variante funktioniert");  
Console.WriteLine(@"Diese Variante funktioniert nun ebenfalls");
```

Referenztypen ohne null

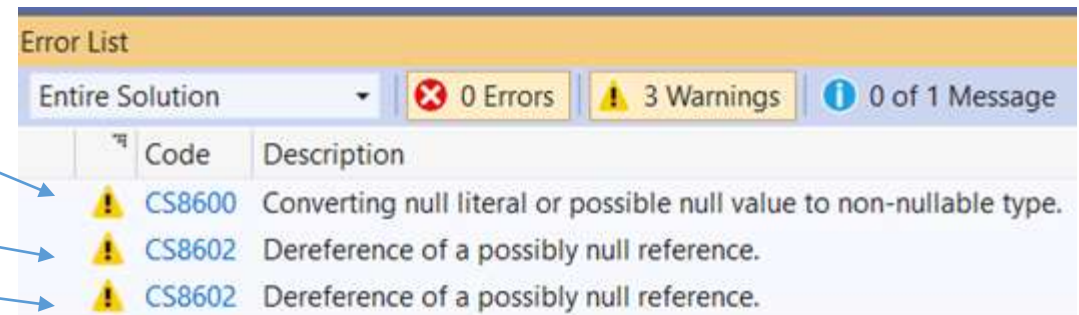
```
static void Main(string[] args)
{
#nullable enable
    string darfNichtNullSein = null;

    string? darfNullSein = null;

    Console.WriteLine(darfNichtNullSein[0]);
    Console.WriteLine(darfNullSein[0]);

    darfNichtNullSein = "Demo";
    Console.WriteLine(darfNichtNullSein);

    if(darfNullSein != null)
        Console.WriteLine(darfNullSein);
#nullable disable
}
```



Error List		
Entire Solution		
0 Errors 3 Warnings 0 of 1 Message		
	Code	Description
!	CS8600	Converting null literal or possible null value to non-nullable type.
!	CS8602	Dereference of a possibly null reference.
!	CS8602	Dereference of a possibly null reference.

Neue Features für Interfaces

- In Interfaces dürfen von nun an folgende Schlüsselwörter und Features genutzt werden:
- Zugriffsmodifizierer:
 - public, private, protected, internal, abstract, virtual
- Features
 - statische Felder und Methoden
 - statische Konstruktoren
 - partielle Interfaces
 - Code in Methoden (-> Standardimplementierung)

Standardimplementierung für Interfaces

```
interface IInterfaceMitDefaultMethode
```

```
{  
    1 reference  
    void MachEtwasInDerKlasse();  
    0 references  
    void MachEtwasImInterface()  
    {  
        Console.WriteLine("Code in einem Interface");  
    }  
}
```

```
class Demo : IInterfaceMitDefaultMethode
```

```
{  
    // MachEtwasImInterface muss nicht implementiert werden !  
    1 reference  
    public void MachEtwasInDerKlasse()  
    {  
        throw new NotImplementedException();  
    }  
}
```

Standardimplementierung für Interfaces

```
interface IInterfaceMitDefaultMethode
```

```
{
```

```
    1 reference
```

```
    void MachEtwasInDerKlasse();
```

```
    0 references
```

```
    void MachEtwasImInterface()
```

```
    {
```

```
        Console.WriteLine("Code in einem Interface");
```

```
    }
```

```
}
```

```
Demo d1 = new Demo();
```

```
d1.
```

- Equals
- GetHashCode
- GetType
- MachEtwasInDerKlasse
- ToString

```
IInterfaceMitDefaultMethode d1 = new Demo();
```

```
d1.
```

- Equals
- GetHashCode
- GetType
- MachEtwasImInterface
- MachEtwasInDerKlasse
- ToString



C# 9.0

C# 9 – Überblick:

- Top-Level Statements
- Target-typed New Expression
- Init only setters
- Patterns
 - Relational Patterns
 - Pattern Combinators
 - Switch Expression
- records

DEMO

Top-Level Statements

- Namespace
 - class
 - static void Main()
- Niemand ruft die Main() auf
- Args sind einfach da
- Darf nur in einer Datei vorkommen



```
1 using System;
2 Console.WriteLine(value: "Hello World!");
3
4 Console.WriteLine(value: $"ARGS: {string.Join(separator: ", ", args)}");
5
6 Hallo();
7
8 void Hallo()
9 {
10     Console.WriteLine(value: "Hallo");
11 }
12
```


Target-typed New Expression

- Alternative zu var

```
AnyClass ac1 = new AnyClass();  
  
var ac2 = new AnyClass();  
  
AnyClass ac3 = new();  
AnyClass ac4 = new(para: "Hannes");
```

Init only setters

- Wert kann nur einmal gesetzt werden
- private set
 - ctor
 - Kann geändert werden
- Gut für records

```
Person p1 = new Person() { Id = 1, Name = "Hannes" };  
p1.Id = 999;
```

```
1 reference  
class Person  
{  
    2 references  
    public int Id { get; init; }  
    1 reference  
    public string Name { get; set; }  
}
```

Patterns

- If / else if / else if / else if / else
- Relational Expressions
 - >, <, >= , <=
- Pattern Combinators
 - is, and, or, not,

```
int temperatur = 4;  
string wetter = temperatur switch  
{  
    < 0 => "Frost",  
    > 0 and < 20 => "Kühl",  
    > 20 and <= 25 => "Warm",  
    > 25 and <= 30 => "Heiß",  
    > 30 => "Hitze",  
    _ => "Unbekannt"  
};
```

```
bool IstBuchstabe(char c) => c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

```
if(wetter is not null)  
{  
    Console.WriteLine(value: "Es gibt Wetter");  
}
```

Records

- Referenztyp
- ToString()
- == prüft alle Werte
- GetHashCode
- Copy & Clone -> with
- Vererbung nur Records

Record definieren

```
public record Person(int Id, string Name);
```

Record benutzen

```
var p1 = new Person(Id: 1, Name: "Hannes");
```

Vom Compiler generierter code:

```
11 references
public class Person : IEquatable<Person>
{
    ...
    8 references
    public int Id { get; init; }

    8 references
    public string Name { get; init; }

    0 references
    public Person(int Id, string Name)
    {
        this.Id = Id;
        this.Name = Name;
    }
    ...
}
```

Init only setters

- ReadOnly after Constructor

```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars} pressure";
}
```



SOLID

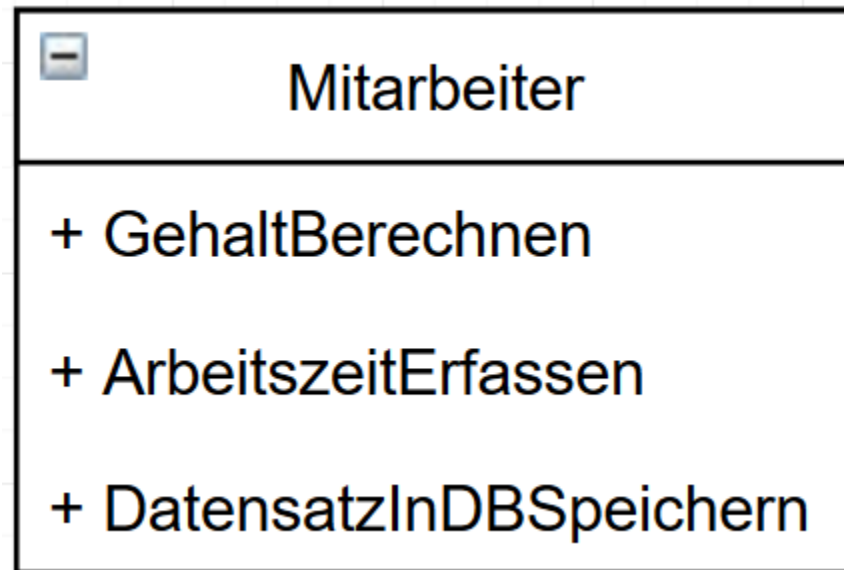
Modulübersicht

- Single-Responsibility-Prinzip
- Open-Closed-Prinzip
- Liskovsches-Substitutionsprinzip
- Interface-Segregation-Prinzip
- Dependency-Inversion-Prinzip

Single-Responsibility-Prinzip

“There should never be more than one reason for a class to change.”

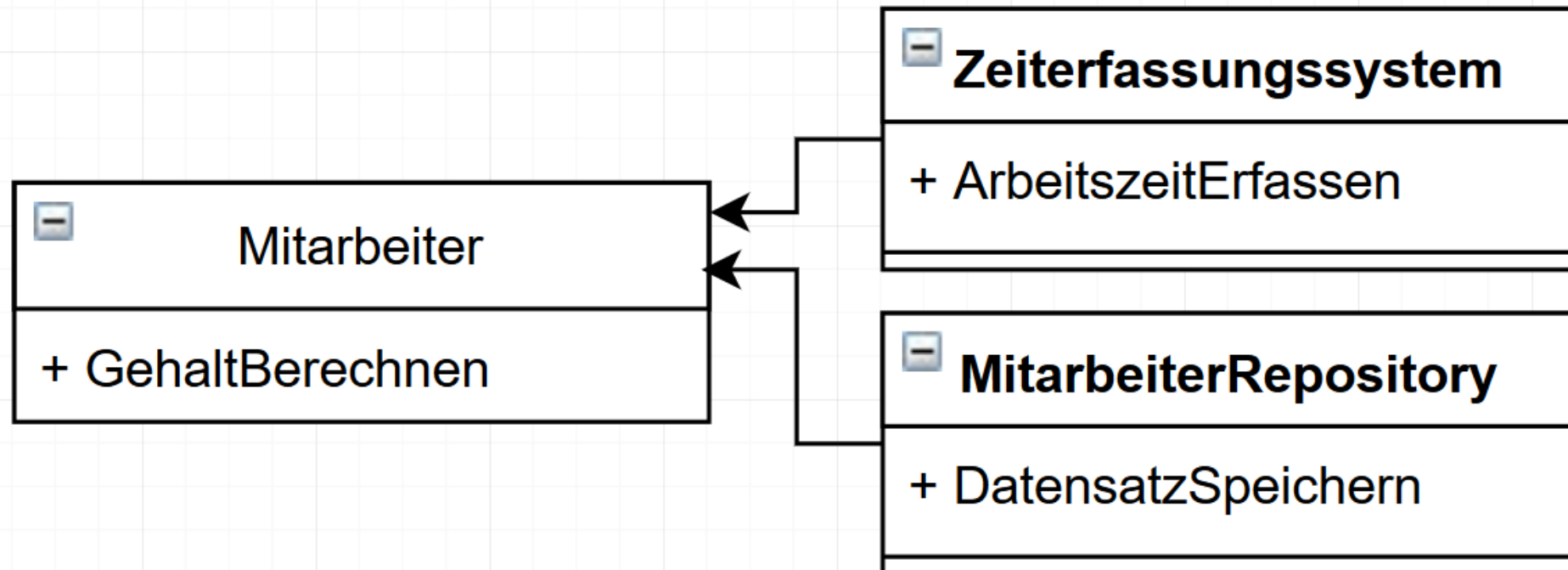
– Robert C. Martin: Agile Software Development: Principles, Patterns, and Practices



Single-Responsibility-Prinzip

“There should never be more than one reason for a class to change.”

– Robert C. Martin: Agile Software Development: Principles, Patterns, and Practices



Open-Closed-Prinzip

“Modules should be both open for extension and closed for modification.”

– Bertrand Meyer: Object Oriented Software Construction

```
static void Main()
{
    ArrayList list = new ArrayList();
    list.Add(new Kreis { Formtyp = GeometrischeForm.Kreis, Radius = 12 });
    list.Add(new Quadrat { Formtyp = GeometrischeForm.Quadrat, Seitenlänge = 21 });

    foreach (Grafik item in list)
    {
        switch(item.Formtyp)
        {
            case GeometrischeForm.Quadrat:
                break; // Logik für das Quadrat
            case GeometrischeForm.Kreis:
                break; // Logik für den Kreis
        }
    }
}
```

Open-Closed-Prinzip

“Modules should be both open for extension and closed for modification.”

– Bertrand Meyer: Object Oriented Software Construction

```
static void Main()
{
    List<Grafik> list = new List<Grafik>();
    list.Add(new Kreis {Radius = 12 });
    list.Add(new Quadrat {Seitenlänge = 21 });

    foreach (Grafik item in list)
    {
        item.Zeichnen();
    }
}
```

Liskovsches-Substitutionsprinzip

„Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be true for object y of type S where S is a subtype of T .“

– Barbara H. Liskov, Jeannette M. Wing: Behavioral Subtyping Using Invariants and Constraints

```
class Rechteck
{
    protected int höhe;
    protected int breite;
    public virtual void SetHöhe(int wert)
    {
        höhe = wert;
    }
    public virtual void SetBreite(int wert)
    {
        breite = wert;
    }
}
```

```
class Quadrat : Rechteck
{
    public override void SetHöhe(int wert)
    {
        höhe = wert;
        breite = wert;
    }
    public override void SetBreite(int wert)
    {
        höhe = wert;
        breite = wert;
    }
}
```

Interface-Segregation-Prinzip

“Clients should not be forced to depend upon interfaces that they do not use.”

– Robert C. Martin: The Interface Segregation Principle

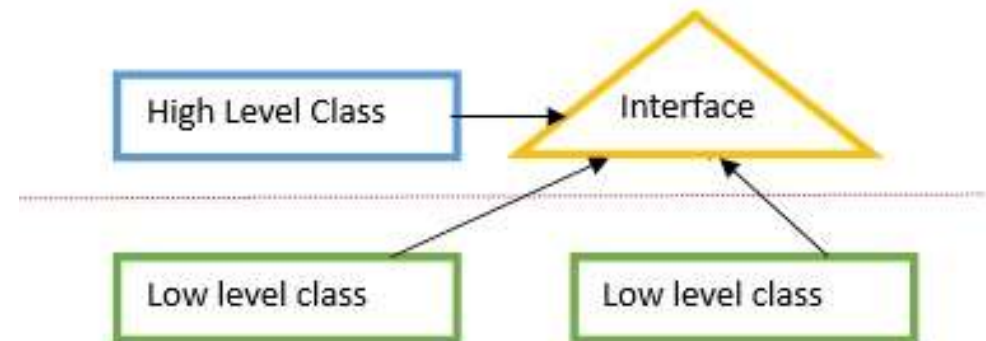
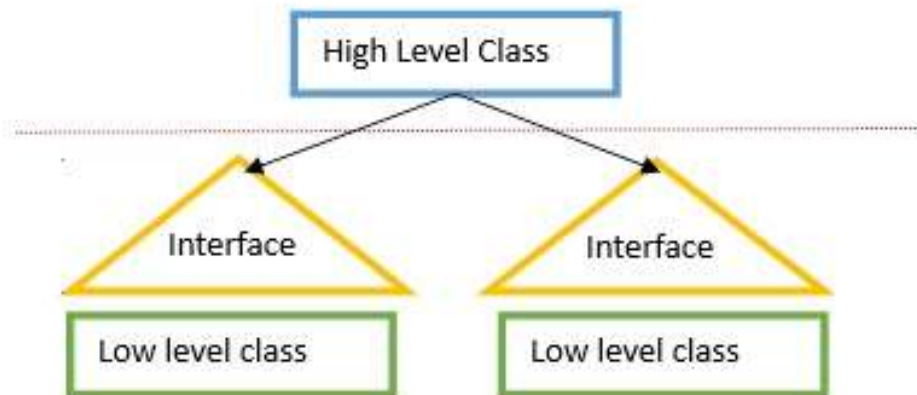


Dependency-Inversion-Prinzip

“A: High-level modules should not depend on low level modules. Both should depend on abstractions.

B: Abstractions should not depend upon details. Details should depend upon abstractions.”

– Robert C. Martin: The Dependency Inversion Principle

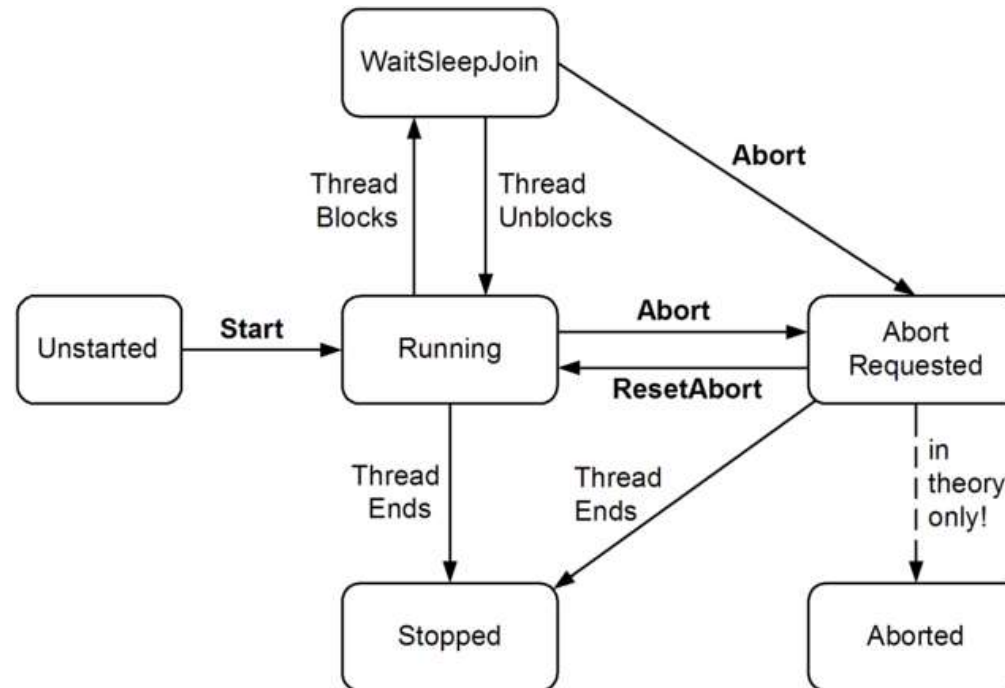




Multithreading

Multithreading

- Ziel: mehrere Handlungsstränge „parallel“ abarbeiten
- Es kann immer nur 1 Thread pro Kern gleichzeitig ausgeführt werden



Thread mit parameterloser Methode

```
public static void Main(string[] args)
{
    Thread t = new Thread(MachEtwas);
    t.Start();

    Console.ReadKey();
}

private static void MachEtwas()
{
    // Logik
}
```

Threads und Parameter

```
public static void Main(string[] args)
{
    Thread t = new Thread(new ParameterizedThreadStart(Berechne));
    t.Start(42);

    Console.ReadKey();
}

private static void Berechne(object c)
{
    // komplizierte Berechnung im Thread
}
```

Thread beenden

```
public static void Main(string[] args)
{
    Thread t = new Thread(MachEtwas);
    t.Start();

    Thread.Sleep(3000);
    t.Abort();
    Console.ReadKey();
}
```

```
private static void MachEtwas()
{
    for (int i = 0; i < 50; i++)
    {
        Thread.Sleep(100);
        Console.Write("zZZZ....");
    }
}
```

Auf einen Thread warten

```
public static void Main(string[] args)
{
    Thread t = new Thread(MachEtwas);
    t.Start();

    t.Join(); // Warten, bis der Thread fertig ist
    Console.WriteLine("---Ende---");
    Console.ReadKey();
}
```

```
private static void MachEtwas()
{
    for (int i = 0; i < 50; i++)
    {
        Thread.Sleep(100);
        Console.Write("zZZZ....");
    }
}
```

ThreadPool

```
public static void Main(string[] args)
{
    ThreadPool.QueueUserWorkItem(MachWas);

    Console.WriteLine("---ENDE---");
    Console.ReadKey();
}

private static void MachWas(object arg1)
{
    // Logik
}
```

Vordergrund- und Hintergrundthreads

- Thread.IsBackground
- Unterschied zwischen Vordergrund- und Hintergrundthreads:
 - Solange Vordergrundthreads laufen, wird die Anwendung von der Runtime ausgeführt.
 - Wenn der letzte Vordergrundthread zu Ende ist, wird die Anwendung geschlossen und alle Hintergrundthreads gestoppt.

```
// Standardfall: Vordergrundthread  
Thread t1 = new Thread(MachEtwas);  
t1.Start();
```

```
// Standardfall: Hintergrundthread  
ThreadPool.QueueUserWorkItem(MachEtwasImThreadPool);
```

Synchronisation

- Problem: mehrere Threads greifen auf die selbe Variable zu
- Lösungen:
 - Monitor / lock() {}
 - Mutex
 - Semaphore
 - System.Collections.Concurrent

Monitor / Lock()

- Monitor ermöglicht das synchronisieren eines Codeblocks
- Zur Synchronisation wird ein beliebiges Objekt benötigt, das immer nur von einem Thread gesperrt werden kann

```
private readonly object lockObject = new object();  
private int betrag;
```

```
public void Erhöhen()  
{  
    Monitor.Enter(lockObject);  
    betrag++;  
    Monitor.Exit(lockObject);  
}
```

```
public void Verringern()  
{  
    lock (lockObject)  
    {  
        betrag--;  
    }  
}
```


Mutex

- Mutex verhält sich wie ein Monitor, erlaubt aber die Verwendung über mehrere Prozesse hinweg und ist daher auch in mehreren Instanzen einer Applikation gemeinsam nutzbar
- Anwendungsfälle:
 - Anwendung nur einmalig starten
 - Inter-Prozess Kommunikation synchronisieren
 - Synchronisieren einer gemeinsamen Aufgabe zweier Prozesse
 - Synchronisation eines gemeinsamen Zugriffs auf Ressourcen aus mehreren Prozessen heraus

Mutex - Beispiel

```
static void Main(string[] args)
{
    Mutex mutex = new Mutex(false, "MeinMutex");
    for (int i = 0; i < 100; i++)
    {
        mutex.WaitOne();
        Thread.Sleep(100);
        Console.WriteLine(i);
        mutex.ReleaseMutex();
    }
    Console.ReadKey();
}
```

Semaphore

- Ein Semaphore beschränkt den Zugriff auf den Codebereich auf eine festgelegte Anzahl von Threads, die gleichzeitig zugreifen dürfen.
- SemaphoreSlim (ab .NET 4.0):
 - Schneller als ein Semaphore, aber nicht Prozessübergreifend
- Vergleich:
 - Semaphore = Mutex mit einstellbarer Anzahl an Threads
 - SemaphoreSlim = Monitor mit einstellbarer Anzahl an Threads

Semaphore - Beispiel

```
class SemaphoreZähler
{
    private static Semaphore semaphore = new Semaphore(3, 3);
    private static int zähler = 0;

    public void MachWas()
    {
        semaphore.WaitOne();
        zähler++;
        Console.WriteLine(zähler);
        zähler--;
        semaphore.Release();
    }
}

static void Main(string[] args)
{
    SemaphoreZähler zähler = new SemaphoreZähler();

    for (int i = 0; i < 500; i++)
        new Thread(zähler.MachWas).Start();

    Console.ReadKey();
}
```

System.Collections.Concurrent

- Beinhaltet Thread-sichere Auflistungen
- Beispiele:
 - ConcurrentBag<T>
 - ConcurrentDictionary<T>
 - ConcurrentStack<T>
 - ConcurrentQueue<T>



Task Parallel Library (TPL)

TPL

- Vereinfacht die Parallelisierung
- Nutzt alle zur Verfügung stehenden Prozessorkerne
- Teilt die Arbeit automatisch auf mehrere Kerne auf und kümmert sich um Low-Level Details
- Klassen:
 - Task<T>
 - Parallel

Parallel

- Parallel.For
 - parallelisierte For-Schleife
 - Nur Increment um 1 möglich !
- Parallel.ForEach
 - parallelisierte ForEach-Schleife
- Parallel.Invoke
 - Führt mehrere Actions parallel aus

Tasks erstellen

Tasks ohne Rückgabe:

```
Task t1 = new Task(MachWas);  
t1.Start();
```

// .NET 4.0

```
Task t2 = Task.Factory.StartNew(MachWas);
```

// .NET 4.5

```
Task t3 = Task.Run(new Action(MachWas));
```

Tasks mit Rückgabe:

```
Task<int> t1 = new Task<int>(GibWasZurück);  
t1.Start();
```

// .NET 4.0

```
Task<int> t2 = Task.Factory.StartNew(GibWasZurück);
```

// .NET 4.5

```
Task<int> t3 = Task.Run(new Func<int>(GibWasZurück));
```

Ergebnis von einem Task auslesen

```
Task<string> t1 = Task<string>.Factory.StartNew(() =>
{
    Thread.Sleep(3000);
    return DateTime.Now.ToLongTimeString();
});
```

```
Console.WriteLine("Ich warte gerade ...");
Console.WriteLine($"Ergebnis: {t1.Result}");
```

Auf einen Task warten

```
Task t = Task.Run(() =>
{
    Console.WriteLine("Sehr lange Aufgabe...");
    Thread.Sleep(5000);
    Console.WriteLine("Aufgabe erledigt");
});
t.Wait();
// Task.WaitAny(t1,t2,t3);
// Task.WaitAll(t1,t2,t3);
```

Einen Task beenden

```
CancellationTokenSource cts = new CancellationTokenSource();

Task t = Task.Run(() =>
{
    Console.WriteLine("Sehr lange Aufgabe...");
    for (int i = 0; i < 100; i++)
    {
        if (cts.Token.IsCancellationRequested)
            break;
        // Alternative:
        // cts.Token.ThrowIfCancellationRequested();

        Console.Write('#');
        Thread.Sleep(100);
    }
    Console.WriteLine("Aufgabe erledigt");
});

Thread.Sleep(2000);
cts.Cancel();
```

Exceptions im Task

```
Task t1 = Task.Run(() => throw new ArgumentException());  
Task<string> t2 = Task<string>.Factory.StartNew(() => throw new FormatException());  
  
// So bekommt man keine Exception mit !  
  
Console.WriteLine("---ENDE---");  
Console.ReadKey();
```

Richtige Vorgehensweise:

```
try  
{  
    // Entweder Warten  
    Task.WaitAll(t1, t2);  
    // oder Result abfragen  
    string ergebnis = t2.Result;  
}  
catch (AggregateException ex)  
{  
    // Alle Exceptions: ex.InnerExceptions  
}
```

Async/Await

