# C# für Dummies

HS2021

**Marco Agostini, Dominik Ehrle**


Computer Science
University of Applied Sciences of Eastern Switzerland
September 2021

# Contents

# 1 .NET Grundlagen

## 1.1 Memory Layout

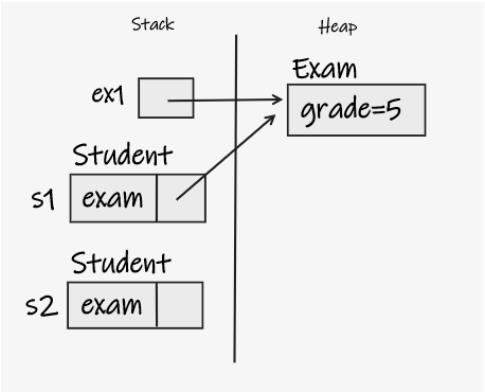
Klassen werden auf dem Heap angelegt und sind implizit public. Structs werden auf dem Stack angelegt und sind implizit public.

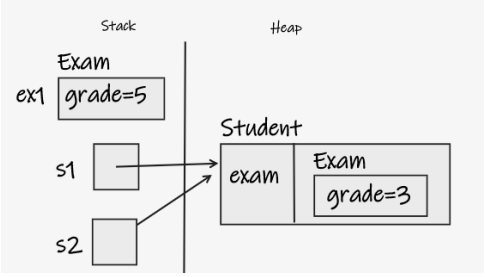
```
// Example 1
public struct Student
{
  public Exam exam;
}
public class Exam
{
  public int grade;
}
```

```
// Example 2
public class Student
{
  public Exam Math;
}
public struct Exam
{
  public int Grade;
}
```

```
public static void Test()
{
  Exam ex1 = new Exam();
  ex1.grade = 3;

  Student s1 = new
   Student();
  Student s2 = s1;
  s1.exam = ex1;
  ex1.grade = 5;
}
```




# 2 C# Grundlagen

## 2.1 Enumeration

```
Volume vMed = Volume.Medium;
string volumeString = "High";
Volume volHigh = ParseVolume(volumeString);

static Volume ParseVolume(string volume)  {
    return (Volume)Enum.Parse(typeof(Volume), volume);
}
enum Volume { Unknown = 10, Low, Medium, High };
```

## 2.2 Object

```
public class Student
{
    private string firstName;
    private string lastName;
    public Student(string firstName, string lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public override string ToString() {
        return firstName+"," +lastName;
    }
}
```

### 2.3 Arrays

```
int[] arr1 = { 1, 2, 3, 4, 5 };
int[] arr2 = new int[] { 1, 3, 5, 7, 9 };
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };
int[][] jaggedArray = new int[6][];
```

### 2.4 String

```
string path = @"C:\Temp\Hello.txt";
Console.WriteLine(path.ToUpper());
Console.WriteLine(path.Replace(@"\","/"));
string[] split = path.Split('\\');
```

## 3 Klassen & Structs

### 3.1 Constructor

```
class Book {
  private string title;
  private string author;
  private bool available;
  public Book(string title, string author, bool available) {
    this.title = title;
    this.author = author;
    this.available = available;
    }
  public Book(string title, string author) : this(title, author, true) { }
  public Book(string title) : this (title, "anonymous") { }
  public Book() : this("untitled") { }
}
```

### 3.2 Indexer

```
class BookList {
  private string[,] books = {
    {"The Green Mile", "Stephen King"},
    {"It", "Stephen King"},
    {"Misery", "Stephen King"},
    {"The Raven", "Edgar Allan Poe"}
  };
  public string this[int index] => $"title: {books[index, 0]} / author: {
   books[index, 1]}";
}
```

### 3.3 Operator

```
class Vector {
  public int X { get; set; }
  public int Y { get; set; }
  public Vector(int x, int y) {
    X = x;
    Y = y;
  }
  public static Vector operator +(Vector v1, Vector v2) {
    return new Vector(v1.X + v2.X, v1.Y + v2.Y);
  }
}
```

### 3.4 Properties

## 4 Vererbung

**virtual** Erlaubt die Überschreibung einer Methode, Property, Indexer, Event mittels override.

**override** Überschreibt explizit die vererbte Methode, Property, Indexer, Event. Ohne Override wird Dynamic Binding unterbrochen.

**new** Unterbricht Dynamic Binding und definiert Methode, Property, Indexer, Event neu.

```
class Vehicle {
    public virtual void WhoAreYou() { Console.WriteLine("Vehicle"); }
}
class Car : Vehicle  {
    public override void WhoAreYou() { Console.WriteLine("Car"); }
}
class Racecar : Car {
    public new virtual void WhoAreYou() { Console.WriteLine("Racecar"); }
}
class F1car : Racecar {
    public override void WhoAreYou() { Console.WriteLine("F1car"); }
}
```

### 4.1 Interfaces

```
interface ISequence {
  void Add(object x); // Method
  string Name { get; } // Property
  object this[int i] { get; set; } // Indexer
  event EventHandler OnAdd; // Event
}

class List : ISequence {
  public void Add(object x) { /* ... */ }
  public string Name { get { /* ... */ } }
  public object this[int i] { get { /* ... */ } set { /* ... */ } }
  public event EventHandler OnAdd;
}
```

## 5 Delegates

Ein Delegate verbindet einen Aufrufer zur Laufzeit mit seiner Zielmethode.

```
delegate int Comparer(object x, object y);
class Car
{
    public string Brand { get; }
    public int EngineSize { get; set; }
    public int WheelSize { get; set; }

    public Car (string Brand, int EngineSize, int WheelSize)
    {
        this.Brand = Brand;
        this.EngineSize = EngineSize;
        this.WheelSize = WheelSize;
    }

    public static int CompareEngine(object x, object y)
    {
        Car c1 = (Car)x;
```

```
        Car c2 = (Car)y;
        if (c1.EngineSize < c2.EngineSize) return -1;
        else if (c1.EngineSize > c2.EngineSize) return 1;
        else return 0;
    }

    public static void CompareCar(Car x, Car y, Comparer compare)
    {
        int result = compare(x, y);
        Console.WriteLine(result);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Car c1 = new Car("Ferrari", 4, 20);
        Car c2 = new Car("Lamborghini", 12, 20);
        Car.CompareCar(c1, c2, Car.CompareEngine);
    }
}
```

### 5.1 Multicast Delegate

Jedes Delegate ist auch ein Multicast Delegate.

```
public delegate void Notifier(string Person);

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public static void sayHi(string sender)
    {
        Console.WriteLine("Hello {0}", sender);
    }

    public static void sayCiao(string sender)
    {
        Console.WriteLine("Ciao {0}", sender);
    }
}
static void Main(string[] args)
{
    Notifier n1 = sayHi;
    n1 += sayCiao;
    n1 += sayCiao;
    n1.Invoke("Marco Agostini");
}
```

## 6 Events

```
    // 1. Define Delegate
    public delegate void RaceEventHandler(object source);

    // 2. Define Publisher
```

```
5   public class RaceController
6   {
7       // 3. Define an event based on Delegate
8       public event RaceEventHandler RaceChangeEvent;
9       public void ChangeRaceState()
10      {
11          RaceChangeEvent?.Invoke(this);
12      }
13  }
14
15  // 5. Write Subscribers
16  public class Car
17  {
18      public int number { get; }
19      public bool IsRunning { get; set; } = false;
20      public Car(int number) { this.number = number; }
21      public void ChangeCarState(object source)
22      {
23          if (!IsRunning)
24          {
25              IsRunning = true;
26              Console.WriteLine("Car with number {0} has started", number);
27          } else {
28              IsRunning = false;
29              Console.WriteLine("Car with number {0} has stopped", number);
30          }
31      }
32  }
33
34  public static void Main(string[] args)
35  {
36      RaceController rh1 = new RaceController();
37      Car c1 = new Car(1);
38      Car c2 = new Car(2);
39
40      // Add Subscribers to Event
41      rh1.RaceChangeEvent += c1.ChangeCarState;
42      rh1.RaceChangeEvent += c2.ChangeCarState;
43
44      rh1.ChangeRaceState();
45      rh1.ChangeRaceState();
46
47      rh1.RaceChangeEvent -= c1.ChangeCarState;
48      rh1.RaceChangeEvent -= c2.ChangeCarState;
49  }
```

# 7 Generics

## 7.1 Type Constraints

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

```
static class MyHelpers
    {
        static TDest CopyTo<TSource, TDest, TElement>(TSource source)
          // Type Constraints for this Operation
            where TSource : IEnumerable<TElement>
            where TDest : IList<TElement>, new()
        {
            TDest dest = new TDest();
            foreach (TElement element in source)
            {
                dest.Add(element);
            }
            return dest;
        }
    }
```

## 7.2 Vererbung

Generische Klassen können von anderen generischen Klassen erben.

```
class MyList<T> : List { }
// Weitergabe des Typparameters an die Basisklasse
class MyList<T> : List<T> { }
// Konkretisierte generische Basisklasse
class MyIntList : List<int> { }
// Mischform
class MyIntKeyDict<t> : Dictionary<int, T> { }
```

## 7.3 Nullable Types

Structs können in der Theorie nicht `null` sein. `default` retourniert den default Wert für den Parametertyp. Vergleiche mit `x == null` Refernce Type=true, false, Value Types=false (Compilerfehler wenn Struct).

```
public void NullExamples<T>()
{
  T x1 = null; // Compilerfehler
  T x2 = 0; // Compilerfehler
  T x3 = default(T); // OK
  T x4 = default; // OK
}
```

### 7.3.1 Nullable Struct

Value Types können dank Generics `null` zugewiesen werden.

**HasValue==true** Liefert den Wert, der gespeichert ist

**HasValue==false** `System.InvalidOperationException`

```
public struct Nullable<T> where T : struct
{
  public Nullable(T value);
  public Nullable();
  public bool HasValue { get; }
  public T Value { get; }
}
```

Danke dem Compiler kann die `T?` Syntax verwendet werden.

```
int? x = 123;
double? y = 1.0;
// Compiler-Output
Nullable<int> x = 123;
Nullable<double> y = 1.0;
```