

C#

GITHUB



- Sourcen mit Beispielen zum Skript finden sie unter <https://github.com/florianwachs/AspNetWebservicesCourse>

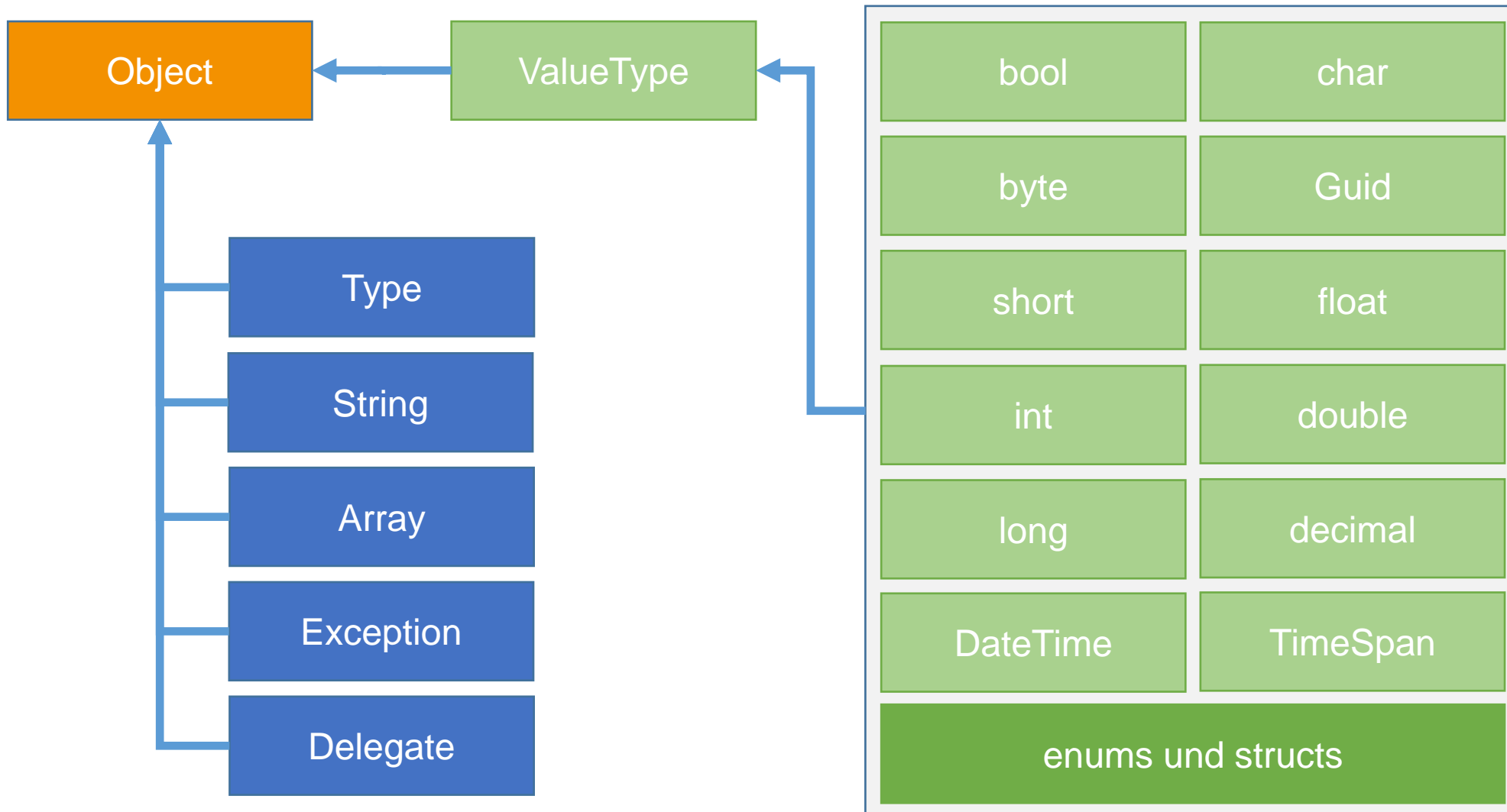
RESSOURCEN

Url	Beschreibung
https://docs.microsoft.com/de-de/dotnet/csharp/	Sehr gute Dokumentation von Microsoft
https://channel9.msdn.com/Series/CSharp-101/?WT.mc_id=Educationalcsharp-c9-scottha	Einführungskurs als Videos
https://docs.microsoft.com/en-us/learn/paths/csharp-first-steps/	Lernkurs C#
https://github.com/florianwachs/FHRWebservices/blob/master/01_csharp/lessons/docs/csharp_cheat_sheet.md	Ein „Cheatsheet“ für die Vorlesung

STATEMENTS

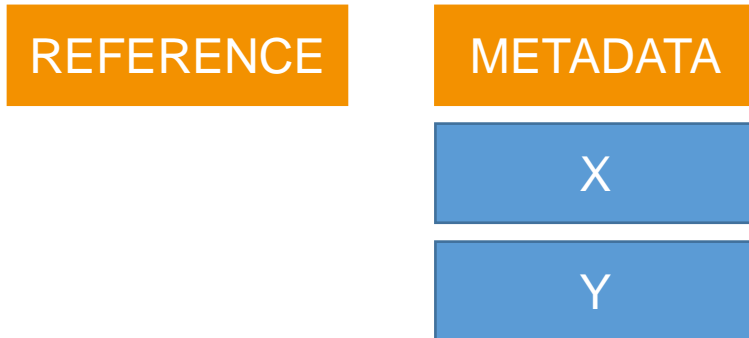
- Sehr ähnlich zu C++ und Java
- **if** (<bool expr>) { ... } else { ... };
- **switch**(<var>) { case <const>: ...; };
- **while** (<bool expr>) { ... };
- **for** (<init>;<bool test>;<modify>) { ... };
- **do** { ... } while (<bool expr>);

DAS TYPE-SYSTEM

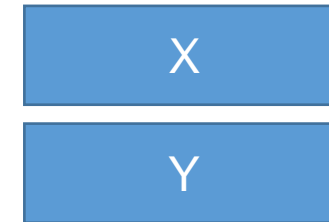


REFERENCE VS. VALUE TYPES

PointClass



PointStructure



REFERENCE-TYPES VS. VALUE-TYPES

- Value-Types sind leichtgewichtige Datencontainer
- Value-Types werden im **Stack** erzeugt und verwaltet
- Reference-Types sind für „rich-objects“ die ein Verhalten abbilden und Referenzen enthalten
- Instanzen eines Reference-Types belegen Speicherplatz auf dem **Heap** und werden vom Garbage Collector verwaltet
- Ein Value-Type wird bei der Zuweisung kopiert, beim Reference-Type wird nur eine neue Referenz auf das gleiche Object erzeugt

OBJECT

- Alle Typen in .NET leiten von Object ab
- Definiert die Methoden `Equals()`, `GetHashCode()`, `ToString()` die von Ableitungen überschrieben werden können
- Definiert die Methoden `GetType()`, `MemberwiseClone()`
- Hat die statischen Methoden `Equals(object a, object b)` und `ReferenceEquals(object a, object b)`

SIMPLE DATATYPES

- Integer Types
 - **byte**, **sbyte** (8bit), **short**, **ushort** (16bit)
 - **int**, **uint** (32bit), **long**, **ulong** (64bit)
- IEEE Floating Point Types
 - **float** (precision of 7 digits)
 - **double** (precision of 15–16 digits)
- Exact Numeric Type
 - **decimal** (28 significant digits)
- Character Types
 - **char** (single character)
 - **string** (rich functionality, by-reference type)
- Boolean Type
 - **bool** (distinct type, **not** interchangeable with **int**)

BOXING / UNBOXING

```
int x = 1;  
// Boxing: int => object  
object boxDHL = x;  
object boxUPS = x;
```

```
var gleichesObjekt = object.ReferenceEquals(boxDHL, boxUPS); // false!
```

```
// Unboxing object => int  
var y = (int)boxDHL;  
var z = (int)boxUPS;
```

NULLABLE TYPES

unassigned values

Die .NET Runtime initialisiert zwar Variablen, der C#-Compiler betrachtet dies jedoch als (Programmier-)Fehler

```
int a;  
Console.WriteLine(a);
```

default oder assign

Das **default** keyword liefert den Default-Wert eines Typs. Bei numerischen Typen ist dies **0**, bei Referenztypen **null**, bei enums das erste Element und bei structs eine default-initialisierte Instanz

```
var a = default(int);  
// oder  
a = 0;  
Console.WriteLine(a);
```

NULLABLE TYPES

```
int? a = default(int?);  
// oder  
a = null;  
  
if (a.HasValue)  
{  
    // Foo  
}  
  
int aOrDefault = a.GetValueOrDefault(10);  
  
// oder  
// null-coalescing operator  
// https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-coalescing-operator  
aOrDefault = a ?? 10;
```

NULLABLE TYPES

- Nur für **Value**-Types
 - int?, long?, decimal?, enums
- Damit können fehlende oder ungültige Werte signalisiert werden
- **HasValue**, **GetValueOrDefault** erleichtern den Umgang mit **null**
- Besonders hilfreich beim Laden von Daten aus einer Datenbank oder in einem ViewModel

VARIABLENDEKLARATION

```
public void Explicit()
{
    int a = 0;
    double b = 2.3;

    // Das m ist nötig um decimal zu erhalten.
    // Als Standard wird double verwendet.

    decimal c = 3.4m;
    string d = "Hallo";

    List<Tuple<int, string, Dictionary<int, string>>> maybeMakeAClass =
    new List<Tuple<int, string, Dictionary<int, string>>>();
}
```

VARIABLENDEKLARATION

```
public void Var()  
{  
var a = 0;  
var b = 2.3;  
  
// Das m Literal ist nötig um decimal zu erhalten.  
// Als Standard wird double verwendet.  
  
var c = 3.4m;  
var d = "Hallo";  
  
var maybeMakeAClass = new List<Tuple<int, string, Dictionary<int, string>>>();  
}
```

VAR

- Implicit Typed Variable
- Der Compiler setzt den Typ während des Kompilierens
- Der Compiler nimmt bei den simplen Datentypen einen default Typ. Es kann aber mit Literalen wie m, f, l übersteuert werden
- Nicht mit var aus JavaScript verwechseln (entspricht eher dem dynamic Keyword von C#)

ENUMERATIONS

```
public enum TaskStates
{
    New, // 0
    Committed, // 1
    InProgress, // 2
    Done, // 3
}
```

Definition

```
public class Task
{
    public TaskStates State { get; set; }

    public Task()
    {
        State = TaskStates.New;
    }
}
```

Verwendung

ENUMERATIONS

```
public void SetNewState(TaskStates newState)
{
    switch (newState)
    {
        case TaskStates.New:
            break;
        case TaskStates.Committed:
            break;
        case TaskStates.InProgress:
            break;
        case TaskStates.Done:
            break;
        default:
            break;
    }
}
```

Verwendung in switch

ENUMERATIONS

Definition

```
// das Flags-Attribut hat nur Auswirkung in der  
ToString()-Methode  
[Flags]  
public enum DocumentOptions  
{  
    ConvertToPDF = 1,  
    SendAsMail = 2,  
    Archive = 4,  
    MarkAsConfidential = 8,  
    Default = ConvertToPDF | Archive,  
}
```

ENUMERATIONS

Verwendung

```
public void Foo(string documentText, DocumentOptions options)
{
    if ((options & DocumentOptions.ConvertToPDF) != 0)
    {
        // foo
    }

    if (options.HasFlag(DocumentOptions.MarkAsConfidential))
    {
        // send to NSA :-)
    }
}
```

Definition

```
[Flags]
public enum DocumentOptions
{
    ConvertToPDF = 1,
    SendAsMail = 2,
    Archive = 4,
    MarkAsConfidential = 8,
    Default = ConvertToPDF | Archive,
}
```

ENUMERATIONS

- Typisierte Aufzählungen (statt int)
- Haben eine implizierte Nummerierung die angepasst werden kann
- können mit boolescher Algebra verwendet werden
- Besonders nützlich in **switch**-Ausdrücken

NAMESPACES

```
namespace FHRWebservices.Basics  
{  
    // Klassen, enums, structs, ...  
}
```

Definition

```
using FHRWebservices.Basics;
```

Einbindung

NAMESPACES

geschachtelt

```
namespace FHRWebservicesSS2015.Basics
{
    namespace Service
    {
        // Klassen, Structs, Enums
    }

    namespace Service.Tools
    {
        // Können auch geschachtelt werden
        namespace Nested
        {
            // Klassen, Structs, Enums
        }
    }
}
```

Flach (Best-Practice)

```
namespace FHRWebservicesSS2015.Basics.Service
{
    // Klassen, Structs, Enums
}

namespace FHRWebservicesSS2015.Basics.Service.Tools
{
    // Klassen, Structs, Enums
}

namespace FHRWebservicesSS2015.Basics.Service.Tools.Nested
{
    // Klassen, Structs, Enums
}
```

NAMESPACES

- Dienen der **semantischen** Code-Strukturierung
- Verhindert Namenskollisionen
- Können geschachtelt werden
- Müssen nicht einer Dateisystem-Struktur folgen
- Können über mehrere Dateien verteilt werden
- über **using [Namespace]** können Namespaces eingebunden werden
 - `using System.Data;`
 - `using System.Collections.Generic;`

PROPERTIES

```
// Properties mit Backing-Field
private int age;
public int Age
{
    get
    {
        return age;
    }

    private set
    {
        age = value;
    }
}
```

- Mischung aus Feld und Methode
- get / set oder beides

PROPERTIES

```
// auto-property
public string FirstName { get; private set; }

// C# 6: initializer
public DateTime Timestamp { get; } = DateTime.UtcNow;

// C# 7: expression
public DateTime Timestamp => DateTime.UtcNow;

public int Age
{
    get => age;
    set
    {
        if (age >= 0)
            age = value;
    }
}
```

PROPERTIES

- Werden vom Compiler in get- und set-Methoden übersetzt
- Automated Properties brauchen kein Backingfield, der Compiler legt eins an
- get und set können unterschiedliche Access-Modifier haben
- Mit C# ab Version 6 können auch Automated Properties schon bei der Definition initialisiert werden
- Ab C# 7 können auch Properties als Expression definiert werden

CLASS

```
public class Person
{
    // Fields
    private long? id;

    // Properties
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime? Created { get; private set; }
    public bool IsNew
    {
        get { return !id.HasValue; }
    }

    // ...
}
```

CLASS| Constructors

```
public class Person
{
    // ...

    // Konstruktoren: Expliziter Default
    public Person()
    {
        Created = DateTime.Now;
    }

    // Custom Konstruktor der den Default-Konstruktor aufruft
    public Person(string firstName, string lastName) : this()
    {
        FirstName = firstName;
        LastName = lastName;
    }

    // ...
}
```

CLASS

```
public class Person
{
    // ...

    // Methoden
    public void Save()
    {
        // Sehr empfehlenswert!
        id = new Random().Next();
    }

    // indexer
    // events
    // delegates
}
```

CLASS

```
// Default Konstruktor
var noName = new Person();
Console.WriteLine(noName.Created);

// Custom Konstruktor
var hansi = new Person("Hansi", "Hintermeier");

// Object-Initializer Syntax
// https://msdn.microsoft.com/en-us/library/bb384062.aspx
var nice = new Person
{
    FirstName = "Jason",
    LastName = "Bourne"
};
```

CLASS| static

```
// Von static Klassen gibt es genau eine Instanz pro AppDomain
public static class StaticClass
{
    public static void Demo1()
    {
        Console.WriteLine(GetUserName());
    }

    // in einer static class müssen alle Member static sein
    public static string GetUserName()
    {
        return Environment.UserName;
    }
}

public class StaticMemberInClass
{
    // in einer non-static class können static Member enthalten sein
    public static string XmlElementName
    {
        get
        {
            return "StaticMember";
        }
    }
}
```


CLASS

- user-defined Type
- Grundbaustein der OOP
- kapselt Fields, Properties, Methods, usw. in einer **semantischen** Einheit
- mit dem **new**-Keyword wird eine neue Instanz einer Klasse erzeugt
- Die Felder und Properties bilden den **Zustand** einer Instanz
- Klassen können in Klassen geschachtelt werden
- **static**
 - Klasse: es gibt pro AppDomain nur eine Instanz der Klasse, die Runtime erzeugt diese automatisch. Alle Member müssen ebenfalls static sein
 - Member: Instanz-Member können in einer non-static Klasse mit static Membern gemischt werden
 - Statische Klassen werden nur für Helper / Tools empfohlen, da sie eine enge Koppelung mit sich bringen die in Unit Tests schwer bis nicht auflösbar ist

ACCESS MODIFIER

- **public** : Keine Zugriffsbeschränkung
- **private**: Sichtbarkeit nur innerhalb des definierenden Types
- **protected**: Sichtbar innerhalb des deklarierenden Types und Ableitungen der Klasse
- **internal**: nur innerhalb der definierenden Assembly sichtbar
- **internal protected**: nur innerhalb der definierenden Assembly oder Ableitungen der Klasse sichtbar
- Können an Typdefinitionen und Membern vergeben werden
- Werden sie weggelassen ist der Standard für Typen **internal** und für Member **private**

VERERBUNG

// Nur Einfachvererbung erlaubt

```
public class Student : Person
{
    public decimal Happiness { get; private set; }

    public void MakeParty()
    {
        Happiness = decimal.MaxValue;
    }

    public void WriteTests()
    {
        Happiness = decimal.MinValue;
    }
}
```

```
var student = new Student
{
    FirstName = "Schakeline Mandy",
    LastName = "Mandy"
};

student.MakeParty();
student.WriteTests();
student.Save();
```

VERERBUNG| base

```
public class Student : Person
{
    // Mit base kann auf die Konstruktoren der Basis zugegriffen werden
    public Student(string firstName, string lastName, decimal motivation)
        : base(firstName, lastName)
    {
        Motivation = motivation;
    }

    public override string ToString()
    {
        // mit base kann auf Implementierungen der Basisklasse
        // zugegriffen werden
        return string.Format("{0} {1}:", base.ToString(), MoodStatus);
    }
}
```

VERERBUNG| abstract

```
// von abstrakten Klassen kann keine
// Instanz erzeugt werden
public abstract class DtoBase
{
    public abstract string ElementName
    {
        get;
    }

    public abstract Task AppendTo(XmlWriter w);
    public abstract void ReadFrom(XElement e);
}
```

```
public class ArticleDto : DtoBase
{
    public override string ElementName
    {
        get { // ... }
    }

    public override Task AppendTo(XmlWriter w)
    {
        // ...
    }

    public override void ReadFrom(XElement e)
    {
        // ...
    }
}
```

VERERBUNG| virtual

```
public abstract class DtoBase
{
    // virtual Methoden bieten eine Implementierung
    // die aber von Ableitungen überschrieben werden
    // können
    public virtual string GetDebugMessage()
    {
        return "Type: " + GetType().Name;
    }
}
```

```
public class ArticleDto : DtoBase
{
    public override string GetDebugMessage()
    {
        return string.Format("{0} {1}", base.GetDebugMessage(), Name);
    }
}
```

VERERBUNG| sealed

```
// sealed unterbindet weitere Ableitungen
public sealed class ArticleDto : DtoBase
{
}
```

```
public class ArticleDto : DtoBase
{
    // sealed unterbindet weiteren override in Ableitungen
    public sealed override string GetDebugMessage()
    {
        return string.Format("{0} {1}", base.GetDebugMessage(), Name);
    }
}
```

VERERBUNG

- Nur einfache Vererbung (bei C++ ist Mehrfachvererbung möglich)
- **base**
 - am Konstruktor: Konstruktoren der Basisklasse aufrufen
 - in Methode / Property: Implementierung der Basisklasse aufrufen
- **abstract**
 - an der Klasse: es kann keine Instanz dieser Klasse mit **new** erzeugt werden
 - an einer Methode / Property: eine Ableitung muss eine Implementierung liefern
 - für abstract Methoden / Properties muss auch die Klasse **abstract** sein
- **virtual**
 - kann an Methoden und Properties definiert werden
 - kann in Ableitungen mit **override** überschrieben werden
- **sealed**
 - an der Klasse: Es kann nicht weiter von der Klasse abgeleitet werden. Hat einen gewissen Performancevorteil
 - an einer Methode / Property: Der Member kann von Ableitungen nicht weiter überschrieben werden

INTERFACES

```
public interface IStudentRepository
{
    IEnumerable<Student> GetAll();
    Student GetById(int id);
}
```

```
public static void PrintStudentsMood(IStudentRepository repo)
{
    foreach (var student in repo.GetAll())
    {
        Console.WriteLine(student.MoodStatus);
    }
}

public static void Demo1()
{
    PrintStudentsMood(new StudentsDuringTestRepository());
    PrintStudentsMood(new StudentsDuringPartyRepository());
}
```

INTERFACES

```
public class StudentsDuringTestRepository : IStudentRepository
{
    public IEnumerable<Student> GetAll()
    {
        return students;
    }

    public Student GetById(int id)
    {
        return students.FirstOrDefault(student => student.Id == id);
    }
}
```

INTERFACES

- Beschreibt ein Verhalten, dass von der implementierenden Struktur oder Klasse erfüllt werden muss
- enthalten keine Implementierung, nur eine Schnittstellenbeschreibung
- **class** und **struct** können beliebig viele Interfaces implementieren
- können von anderen Interfaces ableiten
- die .NET Base Class Library (**BCL**) wird mit hunderten Interfaces ausgeliefert, z.B. `IEnumerable<>`, `IComparable`, `IEnumerable<>`
- Essentiell für eine Vielzahl von **Design Patterns**

CASTING

```
public static void BasicTypes()
{
    int i = 5;
    // wenn von einem Typ mit kleinerem Wertebereich
    // in einen mit größeren zugewiesen wird,
    // ist kein explizites Casting nötig
    long l = i; //long l = (long)i;

    // wenn von einem Typ mit größerem Wertebereich
    // in einen mit kleineren zugewiesen wird,
    // ist ein explizites Casting nötig
    // ACHTUNG: dabei geht bestenfalls Genauigkeit verloren
    double d = 1.2;
    float f = (float)d;
}
```

CASTING

```
public static void ReferenceTypes()
{
    // Student : Person
    var student = new Student();

    // implicit upcast
    Person p = student;

    // wirft einen Fehler
    // student = p;

    student = (Student)p;
}
```

CASTING| as / is

```
public static void AsIs()
{
    IStudentRepository repo = new StudentsDuringPartyRepository();

    try
    {
        var fail = (StudentsDuringTestRepository)repo;
    }
    catch (InvalidCastException)
    {
    }

    // mit is kann getestet werden, ob eine Typkonvertierung erfolgreich wäre
    Console.WriteLine(repo is StudentsDuringTestRepository); // false

    // mit as kann eine Typkonvertierung durchgeführt werden
    // wenn die Typen nicht kompatibel oder null sind, wird auf null evaluiert
    var dasIstNull = repo as StudentsDuringTestRepository;
}
```

CASTING

- Für die ValueTypes im .NET Framework sind bereits sinnvolle **implizite** und **explizite** casts definiert
- Es können auch eigene Type-Conversions in eigenen Typen definiert werden
- Mit **is** und **as** können Typ-Conversions erst geprüft und „sicher“ durchgeführt werden

STRUCTS

```
public struct PointStruct : ICanMove
{
    public int X;
    public int Y;

    public PointStruct(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }

    public void MoveBy(int x, int y)
    {
        X += x;
        Y += y;
    }

    public override string ToString()
    {
        return string.Format("X:{0}, Y:{1}", X, Y);
    }
}
```


STRUCTS

- Können auch Interfaces implementieren
- Sind sealed, dh. Es kann nicht weiter von ihnen abgeleitet werden
- Value-Type
- Werden oft für mathematische Konstrukte verwendet (Point, Money, Vector) wo die Eigenschaften der Value-Types „natürlicher“ als die der Reference-Types sind

BY-VALUE VS. BY-REFERENCE

- Argumente werden standardmäßig **by-value** übergeben
- Eine Kopie des Wertes wird erzeugt, wenn das Argument an die Methode übergeben wird
 - Bei Value-Types: eine komplette Kopie der Struktur
 - Bei Reference-Types: nur die Referenz auf das Objekt wird kopiert

BY-VALUE VS. BY-REFERENCE

```
private static void FooByValue(Student s)
{
    // s ist eine Kopie der Referenz auf s
    // das Objekt lässt sich modifizieren
    s.LastName = "Blaa";

    // das wird nicht funktionieren,
    // da nur die lokale Referenz auf ein
    // neues Objekt zeigt
    s = new Student { FirstName = "HAHAHAHA" };
    Console.WriteLine("Innerhalb von Foo: " +
s.ToString());
}
```

```
var student = new Student { FirstName = "Liese", LastName = "Müller" };
Console.WriteLine("Student vor Foo(): " + student.ToString()); // Liese Müller
FooByValue(student);
Console.WriteLine("Student nach Foo(): " + student.ToString()); // Liese Blaa
```

BY-VALUE VS. BY-REFERENCE

```
private static void FooByReference(ref Student s)
{
    // s ist eine Kopie der Referenz auf s
    // das Objekt lässt sich modifizieren
    s.LastName = "Blaa";

    // das wird nicht funktionieren,
    // da nur die lokale Referenz auf ein
    // neues Objekt zeigt
    s = new Student { FirstName = "HAHAHAHA" };
    Console.WriteLine("Innerhalb von Foo: " + s.ToString());
}
```

```
var student = new Student { FirstName = "Liese", LastName = "Müller" };
Console.WriteLine("Student vor Foo(): " + student.ToString()); // Liese Müller
FooByReference(ref student);
Console.WriteLine("Student nach Foo(): " + student.ToString()); // HAHAHAHA
```

BY-VALUE VS. BY-REFERENCE

- Mit dem **ref**-Modifizier wir angeegeben, das die Parameterübergabe by-reference erfolgen soll
- Der **ref**-Parameter muss sowohl bei der Methodendeklaration als auch beim Aufruf angegeben werden
- Beim Aufruf der Methode muss die übergebene Referenz initialisiert sein

BY-VALUE VS. BY-REFERENCE

```
private static bool SplitNames(string text, out string[] names)
{
    // Muss vor dem Verlassen der Methode zugewiesen werden
    names = null;

    if (!string.IsNullOrEmpty(text))
    {
        names = text.Split();
    }

    return names != null && names.Length > 0;
}
```

```
var test = "Hans Meiser";
string[] names;
if (SplitNames(test, out names))
{
    // foo
}
```

BY-VALUE VS. BY-REFERENCE

- Der **out**-Modifier verhält sich wie der ref-Modifier, bis auf:
 - Der Parameter darf noch nicht zugewiesen worden sein
 - Muss innerhalb der Methode zugewiesen werden, bevor die Methode verlassen wird
- Viele BCL-Typen machen sich out zunutze
 - DateTime.TryParse, decimal.TryParse, Dictionary.TryGetValue

DELEGATES

- Ein Objekt das „weiß“ wie eine Methode aufzurufen ist
- Ein Delegate-Type definiert die Signatur der aufzurufenden Methode
 - return type und Parameter der Methode
- Delegates sind typsicher und können als Parameter eingesetzt werden
- Analog zu einem C-function-pointer

DELEGATES

```
// Definition eines Delegate-Types
```

```
public delegate decimal MathOperation(decimal x, decimal y);
```

```
public static decimal Add(decimal x, decimal y)
{
    return x + y;
}
```

```
public static decimal Multiply(decimal x, decimal y)
{
    return x * y;
}
```

```
// Dem Delegate kann jede Methode zugewiesen werden,
// solange sie der Methodensignatur folgt, die der
// Delegate vorgibt
```

```
MathOperation op = new MathOperation(Add);
```

```
// oder
```

```
op = Add;
```

```
Console.WriteLine(op(2, 4)); // 6
```

```
op = Multiply;
```

```
Console.WriteLine(op(2, 4)); // 8
```

DELEGATES| Generic

- Die BCL liefert eine Reihe von generellen Delegates mit
 - **Func<>**: n-Parameter und ein Rückgabewert
 - **Action<>**: n-Parameter und void Rückgabe
- `delegate TResult Func < out TResult > ();`
- `delegate TResult Func < in T, out TResult > (T arg);`
- `delegate TResult Func < in T1, in T2, out TResult > (T1 arg1, T2 arg2);`
- `delegate void Action ();`
- `delegate void Action < in T > (T arg);`
- `delegate void Action < in T1, in T2 > (T1 arg1, T2 arg2);`
- Geht bis T16

DELEGATES| Generic

```
public static decimal Add(decimal x, decimal y)
{
    return x + y;
}
```

```
// Generic Delegates
// 2 decimal Parameter, decimal Return Type
Func<decimal, decimal, decimal> op = Add;
Console.WriteLine(op(2, 4));
```

LAMBDA

- Eine unbenannte Methodenimplementierung statt einer Delegate-Instanz
- „Syntactic Sugar“, Compiler macht die Arbeit
- `(parameters) => expression-or-statement-block`
- Expression-Block
 - `Func<int,int> sqrt = x => x*x;`
- Statement-Block
 - `Func<int,int> sqrt= x=>{return x*x;};`
- LINQ und Lambdas sind füreinander gemacht...

LAMBDA

```
MathOperation op = (x, y) => x + y;  
Console.WriteLine(op(2, 4)); // 6
```

```
// oder mit Typ Angabe
```

```
MathOperation op2 = (decimal x, decimal y) => x + y;
```

```
// mit Generics
```

```
Func<decimal, decimal, decimal> op3 = (x, y) => x + y;
```

LAMBDA

```
// Collections und Arrays bieten viele Methoden die  
// einen Delegate als Parameter verwenden  
var a = new List<int> { 1, 2, 3, 5, 67, 345, 223334 };  
var biggerThan5 = a.FindAll(n => n > 5);
```

```
// ohne Lambda  
var biggerThan5OhneLambda = a.FindAll(BiggerThan5);
```

```
private bool BiggerThan5(int n)  
{  
    return n > 5;  
}
```

ARRAYS

- Arrays fassen eine feste Anzahl von Elementen eines Typs zusammen
- Elemente des Arrays werden in einem zusammenhängenden Speicherblock abgelegt, was einen effektiven Zugriff erlaubt
- Index ist 0-basiert
- Arrays leiten von **System.Array** ab und bieten einige nützliche Methoden und Eigenschaften

ARRAYS

```
// Arrays werden mit ihrer Größe initialisiert
int[] a = new int[5];

// über einen Indexer kann auf Elemente zugegriffen werden
a[0] = 1;
a[2] = 3;

// der Index ist 0-basiert
var third = a[2];

// array initialization expression
int[] b = { 5, 4, 3, 2, 1 };
```


MULTIDIMENSIONAL ARRAYS

```
var rectangular = new int[3, 3]
{
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 }
};
var valueFirstRowSecondColumn = rectangular[0, 1]; // 1
```

rectangular array

Ein n-dimensionaler Block von Speicher

```
var jagged = new int[3][]
{
    new []{1,2,3},
    new []{4,5},
    new []{6}
};

var val = jagged[1][1]; // 5
```

jagged array

Arrays von Arrays

COLLECTIONS

System.Collection.Generic

List<T>

Dictionary<K,V>

HashSet<T>

System.Collection.Concurrent

[\[MSDN\]](#)

System.Collection

ArrayList

Hashtable

BitArray

Stack

Queue

SortedList

COLLECTIONS

```
var list = new List<int>();  
list.Add(1);  
list.AddRange(new[] { 2, 3, 4, 5 });  
  
for (int i = 0; i < list; i++)  
{  
    var item = list[i];  
    Console.WriteLine(item);  
}  
  
foreach (var item in list)  
{  
    Console.WriteLine(item);  
}
```

COLLECTIONS

```
var map = new Dictionary<string, decimal>
{
    { "EUR", 1 },
    { "USD", 1.1m },
    { "NOK", 8m }
};
```

```
map.Add("CHF", 2);
```

```
var quote = map["CHF"];
map["CHF"] = 2.5m;
```

```
if (map.ContainsKey("AUD")) { }
```

```
decimal q;
if (!map.TryGetValue("AUD", out q))
{
    q = 10;
}
```

```
// C# 7: out-Variables
if(map.TryGetValue("USD", out decimal amount))
{
    q = amount;
}
```

LINQ

- Language Integrated Query
- Framework und Sprachfeatures um typsichere Queries gegen Collections und Arrays schreiben zu können (alles was IEnumerable<T> implementiert)
- Filterungs- und Aggregierungsmöglichkeiten
- <https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b>

LINQ

Fluent-Syntax

```
var test = Enumerable.Range(0, 50);

var methodQuery = test
    .Where(number => (number % 2) == 0)
    .Select(number => Math.Pow(number, 2));

foreach (var number in methodQuery)
{
    Console.Write(number + " ");
}
```

Query-Syntax

```
var test = Enumerable.Range(0, 50);

var query = from number in test
            where (number % 2) == 0
            select Math.Pow(number, 2);

foreach (var number in query)
{
    Console.Write(number + " ");
}
```

Beide Queries werden erst ausgeführt, wenn
iteriert wird

LINQ

```
IStudentRepository repo = new StudentsDuringPartyRepository();

foreach (var student in repo.GetAll()
    .Where(student =>
        student.Motivation > 5.0 && student.FirstName.StartsWith("A"))
    .OrderBy(student => student.FirstName))
{
    // Foo
}
```

INDEXER

```
// Aus C# in a Nutshell
public class Sentence
{
    private string[] words;

    public Sentence(string text)
    {
        words = text.Split();
    }

    public string this[int index]
    {
        get { return words[index]; }
        set { words[index] = value; }
    }

    public override string ToString()
    {
        return string.Join(" ", words);
    }
}
```

```
var t = new Sentence("The quick brown fox");
Console.WriteLine(t[2]); // quick
t[2] = "old";
Console.WriteLine(t.ToString());
```


INDEXER

- Array-like Zugriff auf Klassen und Strukturen die sich wie eine Liste oder ein Dictionary verhalten
- Auch **string** implementiert einen Indexer: `var test = „Hallo“; char a = test[1];`

ATTRIBUTES

- .NET erlaubt das Hinzufügen von zusätzlichen Metadaten zu Types, Members und Assemblies
- Eigener Code kann damit „dekoriert“ werden
- Die BCL hat eine Reihe definiert
 - [Serializable], [Test], [Obsolete], [Flags]
- Ein Attribut alleine hat keine Wirkung, ausführender Code (oder die Runtime) muss die Attribute berücksichtigen

ATTRIBUTES| Definition

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class TaskAttribute : Attribute
{
    public enum Severitiy { Low, Mid, High, Critical }

    public string Description { get; set; }
    public Severitiy Level { get; set; }

    public TaskAttribute(string description)
    {
        this.Description = description;
    }
}
```

ATTRIBUTES| Verwendung

```
// Per Konvention kann man statt TaskAttribute auch nur Task schreiben
[Task("Hier fehlt die Implementierung", Level = TaskAttribute.Severity.Critical)]
[Task("Wofür ist die Klasse da?")]
public class MyClass
{
}
```

ATTRIBUTES| Auswertung

```
var type = typeof(MyClass);
var attrs = type.GetCustomAttributes(typeof(TaskAttribute), true);

if (attrs.Length != 0)
{
    foreach (var attr in attrs)
    {
        var taskAttr = (TaskAttribute)attr;
        Console.WriteLine("Task '{0}' hat die Dringlichkeit {1}",
            taskAttr.Description,
            taskAttr.Level);
    }
}
```

GENERIC| class

```
public class MyAwesomeStack<T>
{
    private const int Max_Size = 100;

    private int position;
    private T[] data = new T[Max_Size];

    public void Push(T item)
    {
        data[position++] = item;
    }

    public T Pop()
    {
        return data[position--];
    }
}
```

```
var myStringStack = new MyAwesomeStack<string>();
myStringStack.Push("hello");

var myIntStack = new MyAwesomeStack<int>();
myIntStack.Push(1);
```

GENERICS| method

```
private static void Swap<T>(ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

```
int x = 5;
int y = 10;
```

```
// man kann die Typenliste angeben
Swap<int>(ref x, ref y);
```

```
// oder es den Compiler machen lassen
Swap(ref x, ref y);
```

GENERICs| constraints

- where T: base-class
 - Base Class Constraint
- where T: interface
 - Interface Constraint
- where T: class
 - Reference-Type Constraint
- where T: struct
 - Value-Type Constraint
- where T: new()
 - Parameterless Constructor Constraint
- where U: T

GENERICs| constraints

```
public class MyFactory<T> where T : new()
{
    public T[] Make(int count)
    {
        return Enumerable.Range(0, count).Select(_ => new T()).ToArray();
    }
}
```

```
public class MyGenericClass<T, U>
    where T : Student, ICanMove
    where U : new()
{
    // T muss von Student ableiten oder Student sein und das ICanMove Interface implementieren
    // U muss einen parameterlosen Konstruktor besitzen
}
```

GENERIC

- Oberstes Ziel: Code Wiederverwendbarkeit
- Sind ein **Template** mit **Typ**-Platzhaltern
- Macht die C# Collections performant und typsicher
- Können in Interfaces, classes, Methoden und structs definiert werden
- Gleiche Logik, nur unterschiedliche Typen
- Constraints können für jeden Typ-Parameter angegeben werden, egal ob in Typdefinition oder Methodendefinition

EXCEPTIONS

```
try
{
    NotDoneYet();
}
catch (InvalidOperationException)
{
    // Fängt nur Exceptions von diesem Typ
}
catch (Exception ex)
{
    // Fängt alle restlichen Exceptions

    // wirft die Exception weiter den Stacktrace rauf
    throw;
}
finally
{
    // Wird auf "jeden" Fall ausgeführt
    // Aufräumen, z.B. File-Handles schließen
}
```

```
public void NotDoneYet()
{
    throw new NotImplementedException("Hab´s doch gesagt!");
}
```

EXCEPTIONS

- Es können eigene Ableitungen von Exceptions erzeugt werden
- Das .NET Framework definiert eine Vielzahl:
`FileNotFoundException`, `InvalidOperationException`,
`NotImplementedException`, `StackOverflowException`
- Können innerhalb eines try-catch Blocks behandelt werden
- Es können beliebig viele catch-Blöcke angegeben werden
- Mit `throw` kann eine Exception weitergeworfen werden
- Exceptions können in andere Exceptions verpackt werden
- Bad Practice: Exceptions nie zur Programmflusssteuerung verwenden

EXPRESSION-BODIED FUNCTION AND PROPERTY (C# 6)

```
// Normale Methode  
private string CalculateBestHero(int inYear)  
{  
    return "Chuck Norris! EVERY YEAR";  
}
```

```
// Methode mit Expression-Body  
private string CalculateBestHero2(int inYear) => "Chuck Norris! EVERY YEAR";
```

EXPRESSION-BODIED FUNCTION AND PROPERTY (C# 6)

```
// Normale berechnete Property
public string CompleteName
{
    get
    {
        return FirstName + LastName;
    }
}

// Property mit Expression-Body
public string CompleteName2 => FirstName + LastName;
```

NULL-CONDITIONAL OPERATOR (C# 6)

```
public static void BeforeCSharp6()
{
    var s = GetStudent(123);

    // Alle Null-Checks durchführen bevor wir auf Count zugreifen können
    int addressCount = s != null
        && s.Contact != null
        && s.Contact.Addresses != null ? s.Contact.Addresses.Count : 0;
}
```

```
public static void WithCSharp6()
{
    var s = GetStudent(123);

    // Code hinter "?" wird nur ausgeführt wenn davor ein Non-Null Wert ermittelt wurde
    int addressCount = s?.Contact?.Addresses?.Count ?? 0;
}
```

STRING INTERPOLATION (C# 6)

```
public static void WithoutInterpolation()
{
    var p = GetPerson();
    // Verwendung von numerierten Platzhaltern im String
    var text = string.Format("Der User {0} {1} wurde am {2} erzeugt.", p.FirstName, p.LastName,
                             p.Created);
    Console.WriteLine(text);
}
```

```
public static void WithInterpolation()
{
    var p = GetPerson();

    // Direkte Verwendung von Variablen im String. "$" ermöglicht die Interpolation
    var text = $"Der User {p.FirstName} {p.LastName} wurde am {p.Created} erzeugt.";
    Console.WriteLine(text);
}
```


ANONYMOUS TYPES

- Innerhalb einer Property oder Methode kann ein anonymer Typ erzeugt werden
- Anonym, da keine explizite Klassendefinition vorhanden sein muss. Der Compiler erzeugt eine Klasse und implementiert Equals(), GetHashCode() und ToString()
- Er kann die Property oder Methode nur durch einen Cast auf **object** verlassen, was aber nicht empfohlen wird
- Das Einsatzgebiet ergibt sich im Zusammenspiel mit LINQ

ANONYMOUS TYPES

```
// Der Compiler implementiert automatisch eine  
// Klasse und implementiert ToString Equals und GetHashCode  
// Die Klasse leitet direkt von Object an  
var person1 = new { FirstName = "Jason", LastName = "Bourne" };  
Console.WriteLine(person1);
```

```
// Für person2 wird die gleiche Klasse  
// verwendet wie für Person 1  
var person2 = new { FirstName = "Jason", LastName = "Bourne" };
```

```
Console.WriteLine(person1.Equals(person2)); // true
```

```
// Für person3 erzeugt der Compiler eine neue  
// Klasse, es kommt auf die Reihenfolge der  
// Properties an  
var person3 = new { LastName = "Bourne", FirstName = "Jason" };
```

```
Console.WriteLine(person1.Equals(person3)); // false
```

ANONYMOUS TYPES

```
var query = Enumerable.Range(0, 5).Select(n => new
{
    Number = n,
    Square = n * n,
    Sqrt = Math.Sqrt(n)
});

foreach (var item in query)
{
    Console.WriteLine("Number: {0}, Square: {1}, Sqrt: {2}", item.Number, item.Square, item.Sqrt);
}
```

ANONYMOUS TYPES

```
var query = from person in GetTestPersons()
            group person by person.FirstName[0] into grp
            orderby grp.Key
            select new { FirstLetter = grp.Key, Personen = grp };

foreach (var gruppe in query)
{
    Console.WriteLine("First Letter: " + gruppe.FirstLetter);
    foreach (var person in gruppe.Personen)
    {
        Console.WriteLine("\t" + person.ToString());
    }
}
```

YIELD

- Das **yield**-Keyword ist eine Erleichterung des Compilers, um einen Enumerator zu erzeugen
- **yield return <Expression>** liefert einen Wert an den Aufrufer zurück. Die Kontrolle über die weitere Iteration wird ebenfalls an den Aufrufer zurück gegeben
- Mit **yield break** kann dem Aufrufer mitgeteilt werden, das nicht weiter iteriert werden kann

YIELD

```
// die Verwendung von yield erspart die Definition
// einer eigenen Klasse die IEnumerator und IEnumerable
implementiert
public static IEnumerable<int> GetRandom(int count)
{
    var rnd = new Random();
    var hardBreak = 100;
    for (int i = 0; i < count; i++)
    {
        if (i == hardBreak)
        {
            yield break;
        }
        // yield return gibt die Kontrolle an
        // den Aufrufer zurück. Erst wenn der Aufrufer
        // wieder weiter iteriert, wird hier weiter gemacht
        yield return rnd.Next();
    }
}
```

YIELD

```
// Obwohl wir max. 10000 Randoms  
// generieren wollen, wird die Generierung  
// nach 5 beendet  
var query = GetRandom(10000).Take(5);  
  
var i = 1;  
foreach (var item in query)  
{  
    Console.WriteLine("{0}: {1}", i++, item);  
}
```

YIELD

Mit yield

```
// die Verwendung von yield erspart die Definition
// einer eigenen Klasse die IEnumerator und IEnumerable implementiert
public static IEnumerable<int> GetRandom(int count)
{
    var rnd = new Random();
    var hardBreak = 100;
    for (int i = 0; i < count; i++)
    {
        if (i == hardBreak)
        {
            yield break;
        }
        // yield return gibt die Kontrolle an
        // den Aufrufer zurück. Erst wenn der Aufrufer
        // wieder weiter iteriert, wird hier weiter gemacht
        yield return rnd.Next();
    }
}
```

Ohne yield

```
// analog zum DemoMitYield: Das müsste ohne yield implementiert werden
// um das gleiche Verhalten zu erhalten
public class RandomEnumerator : IEnumerator<int>, IEnumerable<int>
{
    private readonly int hardBreak = 100;
    private readonly int count;
    private readonly Random rnd;

    private int i;
    private int current;

    public RandomEnumerator(int count)
    {
        this.count = count;
        this.rnd = new Random();
    }

    public int Current
    {
        get
        {
            return current;
        }
    }

    public bool MoveNext()
    {
        var currentIndex = i++;
        var hasNext = currentIndex < count && currentIndex != hardBreak;
        if (hasNext)
        {
            current = rnd.Next();
        }

        return hasNext;
    }

    object System.Collections.IEnumerator.Current
    {
        get { throw new NotImplementedException(); }
    }

    public void Reset()
    {
        throw new NotImplementedException();
    }

    public void Dispose()
    {
    }

    public IEnumerator<int> GetEnumerator()
    {
        return this;
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }
}
```


OPTIONALE PARAMETER

```
public static void WriteToConsole(string msg,  
    ConsoleColor foregroundColor = ConsoleColor.Blue,  
    bool useTimeStamp = false)  
{  
    // ...  
}
```

```
// nur der erforderliche Parameter wurde gesetzt  
// für alle anderen wird der Standard verwendet  
WriteToConsole("Hi there");
```

```
// alle Parameter können auch einfach angegeben werden  
WriteToConsole("Alarmstufe ROT", ConsoleColor.Red, false);
```

[\[MSDN\]](#)

NAMED PARAMETER

```
public static void WriteToConsole(string msg,  
    ConsoleColor foregroundColor = ConsoleColor.Blue,  
    bool useTimeStamp = false)  
{  
    // ...  
}
```

// es können beliebige optionale Parameter als named Parameter gefüllt werden

```
WriteToConsole("What's the time? Oh there it is...", useTimeStamp: true);
```

PARTIAL

PartialClass.generated.cs

```
partial class PartialClass
{
    private int? generatedField;
    public int? GeneratedField
    {
        get
        {
            Getting(ref generatedField);
            return generatedField;
        }
        set
        {
            var old = generatedField;
            if (old != value)
            {
                generatedField = value;
                Changed(old, value);
            }
        }
    }
    // Wenn die Methoden nicht implementiert werden
    // entfernt der Compiler die Aufrufe
    partial void Changed(int? oldValue, int? newValue);
    partial void Getting(ref int? value);
}
```

PartialClass.cs

```
public partial class PartialClass
{
    public DateTime? LastChanged { get; private set; }

    // partial Methode wird implementiert und
    // damit vom Compiler auch berücksichtigt
    partial void Changed(int? oldValue, int? newValue)
    {
        LastChanged = DateTime.Now;
    }
}
```

PARTIAL

- Definition einer Klasse oder Methode kann auf mehrere Dateien aufgeteilt werden
- partial Methoden können nur **void** als Rückgabewert haben
- Sehr nützlich im Zusammenspiel mit Codegeneratoren
- Wird eine partial Methode nicht implementiert, entfernt der Compiler den Aufruf
- Der Compiler fügt alle Partial-Definitionen zu einer Datei zusammen

EXTENSION METHODS

- syntactic sugar für den Aufruf von statischen Methoden
- LINQ nutzt Extension Methods um `IEnumerable<>` zu erweitern
- Extension Methods müssen als statische Klasse mit statischen Methoden implementiert werden. Mit **this** wird der zu erweiternde Typ angegeben
 - `public static string MyCustomDateTimeFormat(this DateTime time);`
- Extension Methods haben nur Zugriff auf die public-Member des erweiterten Types
- Die Klassendefinition wird nicht beeinflusst

EXTENSION METHODS

```
// Extension Methods müssen in einer
// static class definiert sein
// damit sie auf den "erweiterten" Typen angewendet werden
// können, muss diese Klasse entweder im gleichen Namespace
// liegen oder der Namespace per using importiert werden.
public static class PersonExtensions
{
    // mit this wird
    // Extension-Methods sind normale statische Methoden,
    // können also auch weitere Parameter enthalten
    public static string GetFullName(this Person person, string prefix = null)
    {
        // Extension-Methods sind syntactic sugar.
        // Sie erweitern nicht die Klassendefinition,
        // sie können nur auf die public-Member zugreifen
        return prefix + person.FirstName + " " + person.LastName;
    }
}

var p = new Person("Franzi", "Maier");
// Für den Aufrufer sieht es so aus,
// als wäre GetFullName() teil der Klassendefinition
Console.WriteLine(p.GetFullName());
// in Wirklichkeit wird aber nur folgendes aufgerufen
Console.WriteLine(PersonExtensions.GetFullName(p));
```

NEW| Hiding Inherited Members

- Mit new an Members können die Basis-Implementierungen versteckt werden und eine eigene Implementierung durchgeführt werden
- Im Prinzip wird dem Compiler gesagt, verwende diese Implementierung statt der Basis
- Der Einsatz macht nur in Ausnahmefällen Sinn

NEW| Hiding Inherited Members

```
public class CompatPerson : Person
{
    // ohne new würde ein compile-Fehler auftreten
    // da IsNew bereits definiert ist.
    public new bool IsNew
    {
        get { return false; }
    }

    // Auch Methoden der Basis können mit new versteckt werden
    public new void Save()
    {
        Console.WriteLine("Save");
        base.Save();
    }
}
```


USING| Directive

- Import von Namespaces
 - `using System.Linq`
- Definition von Typ oder Namespace Aliasen
 - `using MyVector = System.Tuple<int,int,int>;`
 - Verwendung: `MyVector v = new MyVector(1,2,3);`
 - `using SqlStuff = System.Data.SqlClient;`
- Verwendung mit `IDisposable` Objekten
 - Beispiel folgt

USING| IDisposable

```
namespace System
{
    // Summary:
    //     Defines a method to release allocated resources.
    [ComVisible(true)]
    public interface IDisposable
    {
        // Summary:
        //     Performs application-defined tasks associated with freeing, releasing, or
        //     resetting unmanaged resources.
        void Dispose();
    }
}
```

USING| IDisposable

```
public class MyResource : IDisposable
{
    public MyResource()
    {
    }

    public void DoStuff()
    {
        // Framework Guideline
        EnsureNotDisposed();
    }

    public void Dispose()
    {
        // Framework Guideline:
        // Dispose kann mehrfach aufgerufen werden
        if (!disposed)
        {
            Console.WriteLine("MyResource Aufräumarbeiten");
        }
    }
}
```

USING| IDisposable

- Klassen die interne Ressourcen verwalten oder selbst eine Ressource darstellen sollten das **IDisposable**-Interface implementieren
- In der implementierten **Dispose()**-Methode, werden alle Ressourcen freigegeben, welche vom Objekt für die Aufgabenerfüllung benötigt werden
 - .z.B. File- oder Network-Handles, Datenbankverbindungen
- Regel: Ein Objekt das Disposed wurde darf nicht weiter verwendet werden. Viele Framework-Objekte werfen eine Exception, wenn man es trotzdem macht

USING| IDisposable

- Der Aufruf von Dispose führt nicht automatisch zu einem Garbage Collect
- Es ist viel mehr dem Interesse geschuldet, verwendete Ressourcen möglichst schnell freizugeben und zu signalisieren, dass das Objekt nicht weiter verwendet werden soll
- der Garbage Collector räumt in der Regel etwas später auf (der GC ist ein interessanter eigener Themenkomplex...)

USING| IDisposable

- Guidelines des .NET Frameworks
 - Einmal disposed darf ein Objekt nicht reaktiviert werden. Ein Aufruf der Methoden und Properties sollte eine **ObjectDisposedException** werfen
 - **Dispose()** darf beliebig oft aufgerufen werden
 - Wenn ein Disposable-Objekt selbst Objekte enthält die IDisposable implementieren, so sollte auch Dispose auf diesen Child-Ressourcen aufgerufen werden

USING| Statement

```
using (var res = new MyResource())
{
    Console.WriteLine("Verwende MyResource");
}
```

// Gleichbedeutend zu:

```
MyResource res2 = new MyResource();
try
{
    Console.WriteLine("Verwende MyResource");
}
finally
{
    if (res2 != null)
    {
        res2.Dispose();
    }
}
```

ASYNC / AWAIT

- Zum Verständnis von `async` / `await` muss erst noch Einführung in .NET Threading erfolgen
- Das Feature wurde mit C# 5 hinzugefügt und ermöglicht asynchronen Code auf synchrone Weise zu schreiben
- Gerade bei aufeinander folgenden Serviceaufrufen ist die Erleichterung für den Entwickler enorm

ASYNC / AWAIT

- Disclaimer: Multithreading richtig umzusetzen ist hart und erfordert viel Wissen und Erfahrung. Nachfolgend wird nur das absolute Minimum erklärt, um Async / Await zu verwenden
- Weitere Ressourcen
 - C# in a Nutshell, Albahari
 - Concurrent Programming on Windows, Duffy

ASYNC / AWAIT

- Klasse **Thread** in System.Threading als „Urgestein“ des Threading
- .NET verwendet einen Thread Pool der von der Runtime verwaltet wird
- Verschiedenste Pattern für den Aufruf von asynchronen Methoden
 - APM: Begin[OperationName] End[OperationName]
 - EventBased: WebClient.DownloadStringAsync, WebClient.DownloadStringCompleted

ASYNC / AWAIT

■ Task

- Repräsentiert eine asynchrone Operation
- Können durch Continuations verkettet werden
- Können ein Result liefern oder void sein
- Task <> Thread: Der Task Scheduler entscheidet ob und wie parallel ausgeführt wird
- **Task.Run()** scheduled eine Operation, die standardmäßig im .NET Thread Pool ausgeführt wird
- **Task.Result** : Liefert das Ergebnis des Tasks, blockiert den aktuellen Thread falls das Ergebnis noch nicht vorhanden ist
- **Task.Wait()**: Blockiert den aktuellen Thread, bis der Task beendet ist
- **Task.FromResult()**: Wrappt einen Completed Task um einen Wert

■ TaskScheduler

- Weiß wie ein Task auszuführen ist
- Custom TaskScheduler möglich
- .NET Framework liefert einen Standard Scheduler

ASYNCHRON / AWAIT

- Blockieren ist in der Regel unerwünscht da,
 - in GUI Applikationen die Oberfläche „einfriert“ da keine Nachrichten in der Message Loop des UI-Threads mehr abgearbeitet werden
 - in Webanwendungen eine begrenzte Anzahl an Worker Threads zur Verfügung steht, um eingehende Requests zu bearbeiten
- Optimal: Alles im Hintergrund erledigen und nur das Ergebnis in den Synchronisationskontext (UI-Thread, Worker-Thread) dispatchen

ASYNCHRON / AWAIT

async teilt dem Compiler mit, dass im folgenden Codeblock der **await** nicht als Identifier (z.B. für eine Variable) sondern als Keyword verwendet wird

```
public static async Task<string> GetChucksWisdomAsync(string authToken)
{
    var api = "http://api.icndb.com/jokes/random";
    var client = new HttpClient();

    var rawJson = await client.GetStringAsync(api);
    return GetJokeFromJSON(rawJson);
}
```

await lässt den Compiler eine State-Machine bauen. Liefert GetStringAsync einen Task der bereits Completed ist, wird die Methode weiter ausgeführt. Falls nicht, wird der nach dem **await** folgende Codeblock in eine Task-Continuation gepackt und die Methode wird verlassen. Wenn

DYNAMIC| static vs dynamic binding

```
public class Chuck
{
    public string GetWisdom()
    {
        return "Chuck Norris doesn't need to use AJAX "+
            "because pages are too afraid to postback anyways.";
    }
}
```

DYNAMIC| static vs dynamic binding

static binding

```
Chuck chuck = new Chuck();  
// Zur compile-time steht fest, dass GetWisdom()  
// am Objekt chuck verfügbar ist  
chuck.GetWisdom();  
  
// Durch die Zuweisung auf object gehen die  
// Typinformationen von chuck "verloren"  
object chuckObj = chuck;  
  
// der Compiler sieht nur object und dessen Member  
// folgendes führt zu einem compile-time error  
chuckObj.GetWisdom();
```

dynamic binding

```
Chuck chuck = new Chuck();  
  
chuck.GetWisdom();  
  
object chuckObj = chuck;  
  
// dynamic sagt dem Compiler, dass das typechecking  
// erst zur Laufzeit stattfinden soll  
dynamic chuckDynamic = chuckObj;  
chuckDynamic.GetWisdom();
```

(dynamic expression)
This operation will be resolved at runtime.

DYNAMIC

- Funktioniert nicht bei
 - ExtensionMethods (ist compile-time syntactic sugar)
 - Base-Members die von einer Sub-Klasse versteckt werden
 - Interface-Methoden wenn ein Object zuerst gecastet werden müsste um
- `var != dynamic`
 - **var**: Compiler findet den Typ zur compile-time heraus
 - `var x = „Hallo“ =>` static type ist **string**, runtime type ist **string**
 - **dynamic**: Typ wird zur Laufzeit ermittelt
 - `dynamic x = „Hallo“ =>` static type ist **dynamic**, runtime type ist **string**
- `dynamic != Reflection`
 - `dynamic` hat keinen Zugriff auf private Member
- Performance Hit

DYNAMIC| Reflection-Ersatz

```
private class NotRelated
{
    public void LogStatus()
    {
        Console.ForegroundColor = ConsoleColor.Blue;
        Console.WriteLine("Status looks good");
        Console.ResetColor();
    }
}
```

```
private static void BetterCallLog(dynamic o)
{
    o.LogStatus();
}
```

```
private class NotRelatedEither
{
    public void LogStatus()
    {
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("Could be better");
        Console.ResetColor();
    }
}
```

```
public static void ReplaceReflectionDemo()
{
    var obj1 = new NotRelated();
    var obj2 = new NotRelatedEither();

    BetterCallLog(obj1);
    BetterCallLog(obj2);
}
```

DYNAMIC| ExpandoObject

```
// ExpandoObject ist wie ein PropertyBag / Dictionary
// Member können einfach ergänzt werden
dynamic obj = new ExpandoObject();

obj.Test = "Hi";
obj.Crazy = (Func<bool>)((() => true);

// ExpandoObject implementiert INotifyPropertyChanged
((INotifyPropertyChanged) obj).PropertyChanged += (PropertyChangedEventHandler)((s, e) =>
{
    Console.WriteLine("Change: " + e.PropertyName);
});

Console.WriteLine(obj.Test);
Console.WriteLine(obj.Crazy());

obj.Test = "Servus";

try
{
    Console.WriteLine(obj.GibtsNicht());
}
catch (RuntimeBinderException ex)
{
    Console.WriteLine("Diesen Member gibt es nicht");
}
```

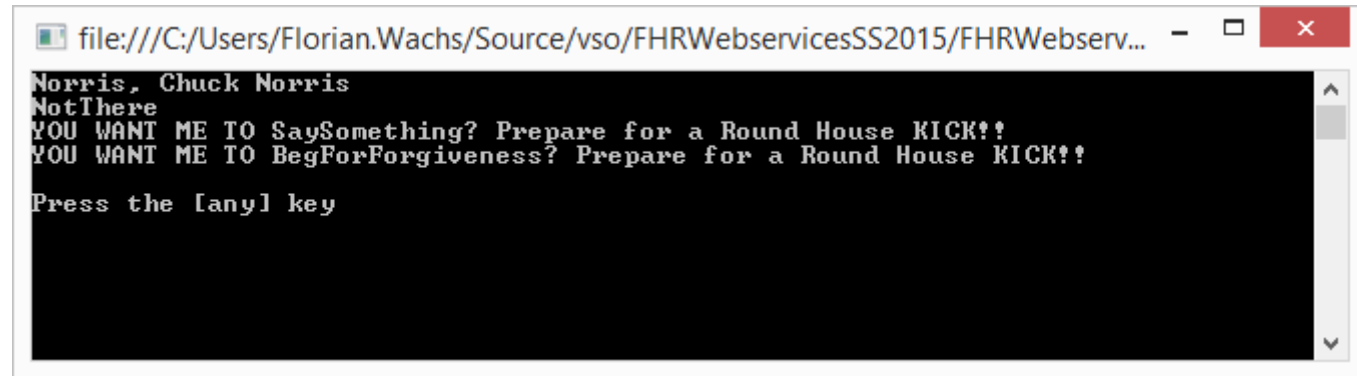
DYNAMIC| DynamicObject

```
public class DynamicChuck : DynamicObject
{
    public override bool TryGetMember(GetMemberBinder binder, out object result)
    {
        result = binder.Name;
        if (binder.Name == "Name")
        {
            result = "Norris, Chuck Norris";
        }
        return true;
    }

    public override bool TryInvokeMember(InvokeMemberBinder binder, object[] args, out object result)
    {
        result = "YOU WANT ME TO " + binder.Name + "? Prepare for a Round House KICK!!";
        return true;
    }
}
```

DYNAMIC| DynamicObject

```
public static void DynamicObjectProviderDemo()  
{  
    dynamic dynamicChuck = new DynamicChuck();  
    Console.WriteLine(dynamicChuck.Name);  
    Console.WriteLine(dynamicChuck.NotThere);  
    Console.WriteLine(dynamicChuck.SaySomething());  
    Console.WriteLine(dynamicChuck.BegForForgiveness());  
}
```



The screenshot shows a console window with the following output:

```
file:///C:/Users/Florian.Wachs/Source/vso/FHRWebservicesSS2015/FHRWebserv...  
Norris, Chuck Norris  
NotThere  
YOU WANT ME TO SaySomething? Prepare for a Round House KICK!!  
YOU WANT ME TO BegForForgiveness? Prepare for a Round House KICK!!  
Press the [any] key
```

DYNAMIC| JSON.NET

```
var rawJson = @"
    {
        ""name"": ""Jason"",
        ""address"": {
            ""street"": ""Unknownstreet""
        },
        ""identities"": [ ""Franz"", ""Xaver"", ""Hiasi"" ]
    }
";

dynamic o = JsonConvert.DeserializeObject(rawJson);

Console.WriteLine(o.name.Value);
Console.WriteLine(o.address.street.Value);

foreach (dynamic identity in o.identities)
{
    Console.WriteLine(identity.Value);
}
```

DYNAMIC| Einsatzgebiete

- COM-Interop
- „natürlicherer“ Zugriff auf XML und JSON
- Dynamische Skriptsprachen in .NET
 - IronRuby, IronPython
- Kann oft statt Reflection angewendet werden und führt zu „schönerem“ Code
- Eigene Proxy-Objekte mit DynamicObject

FINALIZERS

- Zum Verständnis von Finalizern muss zuerst noch der Garbage Collector des .NET Frameworks erklärt werden
- Sie entsprechen dem Destructor in C++ und finalize() in Java

USING STATIC

```
public static void UsingNormalMath(int value) => Console.WriteLine(Math.Abs(value) * Math.PI);
```

// Mit using static können Methoden aus einem Typen direkt importiert werden

```
using static System.Math;
```

```
using static System.Console;
```

```
public static void UsingStaticMath(int value) => WriteLine(Abs(value) * PI);
```


NAMEOF

```
public class Nameof
{
    public string FirstName { get; set; }

    public void UseNameof()
    {
        Console.WriteLine(nameof(CSharpAdvancedLanguageFeatures)); // CSharpAdvancedLanguageFeatures
        Console.WriteLine(nameof(FirstName));                       // FirstName
        // es wird nur der letzte Identifier ausgewertet
        Console.WriteLine(nameof(Console.WriteLine));               // WriteLine
    }

    public void WhoIsYourGreatestHero(string heroName)
    {
        // wird nun bei Refactorings auch in der Exception angepasst
        if (heroName != "Chuck Norris")
            throw new ArgumentException("Der Parameter ist ungültig", nameof(heroName));
    }
}
```

EXCEPTION FILTERS

```
public class ExceptionFilters
{
    public void DoStuffThatThrows()
    {
        try
        {
            // Do Stuff
        }
        catch (HttpException ex) when (ex.ErrorCode == (int)HttpStatusCode.InternalServerError)
        {
            // Handle 500 - Internal Server Error
        }
        catch (HttpException ex) when (ex.ErrorCode == (int)HttpStatusCode.NotFound)
        {
            // Handle 404 - Not Found
        }
        catch
        {
            // The Rest
        }
    }
}
```

TUPLES

```
public class Tuples
{
    // Tuple ist als Reference Type implementiert
    public Tuple<decimal, decimal> GetAmountAndDiscountTuple() => Tuple.Create(100m, 20m);

    // Liefern beide ValueTuples welche als Value Type implementiert sind
    // Für .NET Framework < v4.7 muss das NuGet-Package System.ValueTuple vorhanden sein
    public (decimal, decimal) GetAmountAndDiscountValueTupleNoName() => (100m, 20m);

    public (decimal amount, decimal discountInPercent) GetAmountAndDiscountValueTupleWithName()
    {
        return (100m, 20m);
    }
}
```

TUPLES

```
public class Tuples
{
    public void UseTuples()
    {
        var t1 = GetAmountAndDiscountTuple();
        Console.WriteLine($"Amount {t1.Item1} with Discount {t1.Item2}");

        var t2 = GetAmountAndDiscountValueTupleNoName();
        Console.WriteLine($"Amount {t2.Item1} with Discount {t2.Item2}");

        var t3 = GetAmountAndDiscountValueTupleWithName();
        Console.WriteLine($"Amount {t3.amount} with Discount {t3.discountInPercent}");

        // Tuple lassen sich auf direkt in lokale Variablen übertragen
        var (betrag, rabatt) = GetAmountAndDiscountValueTupleNoName();

        var (betrag2, rabatt2) = GetAmountAndDiscountValueTupleWithName();
    }
}
```

PATTERN MATCHING

```
public void IsExpressionWithPatterns(object o)
{
    if (o is null) return;    // constant pattern "null"
    if (o is int i) // type pattern "int i"
    {
        // Der if-Block wird nur betreten wenn es sich um ein int handelt
        // die Variable i ist nur innerhalb des if-Blocks zugewiesen
        Console.WriteLine(i);
    }

    if (!(o is decimal d))
        return;

    // Ab hier weiß der Compiler das o ein decimal sein muss
    Console.WriteLine(d);
}
```

PATTERN MATCHING

```
public void SwitchWithPatterns(Shape shape)
{
    switch (shape)
    {
        case Circle c:
            WriteLine($"circle with radius {c.Radius}");
            break;
        // Es können auch Bedingungen mit when definiert werden
        case Rectangle s when (s.Length == s.Height):
            WriteLine($"{s.Length} x {s.Height} square");
            break;
        case Rectangle r:
            WriteLine($"{r.Length} x {r.Height} rectangle");
            break;
        default:
            WriteLine("<unknown shape>");
            break;
        case null:
            throw new ArgumentNullException(nameof(shape));
    }
}
```

OUT-VARIABLES

```
decimal q;  
if (!map.TryGetValue("AUD", out q))  
{  
    q = 10;  
}  
  
// C# 7: out-Variables  
// amount ist nur innerhalb des if-Blocks zugewiesen  
if (map.TryGetValue("USD", out decimal amount))  
{  
    q = amount;  
}
```

LOCAL FUNCTIONS

```
public class LocalFunctions
{
    public int Fibonacci(int x)
    {
        if (x < 0) throw new ArgumentException("Less negativity please!", nameof(x));
        return Fib(x).current; // Hier endet die eigentliche Funktion

        // Lokale Funktion mit ValueTuple als return Type
        // Kann auch auf lokale Variablen der umgebenden Funktion zugreifen
        (int current, int previous) Fib(int i)
        {
            if (i == 0) return (1, 0);
            var (p, pp) = Fib(i - 1);
            return (p + pp, p);
        }
    }
}
```


Weitere C# Features

- C# Wird ständig weiterentwickelt, einen Überblick über neue Features finden Sie hier:
- <https://docs.microsoft.com/de-de/dotnet/csharp/whats-new/csharp-8>