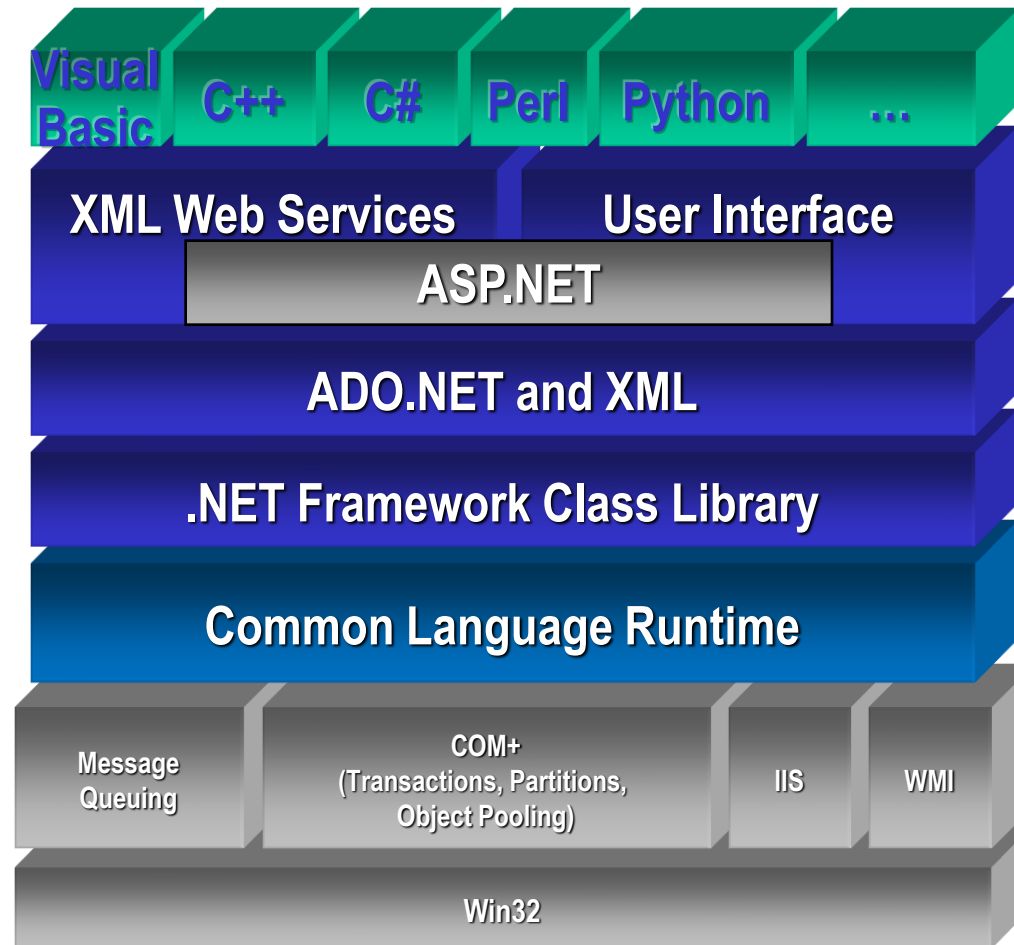


# **C#-Grundlagen**

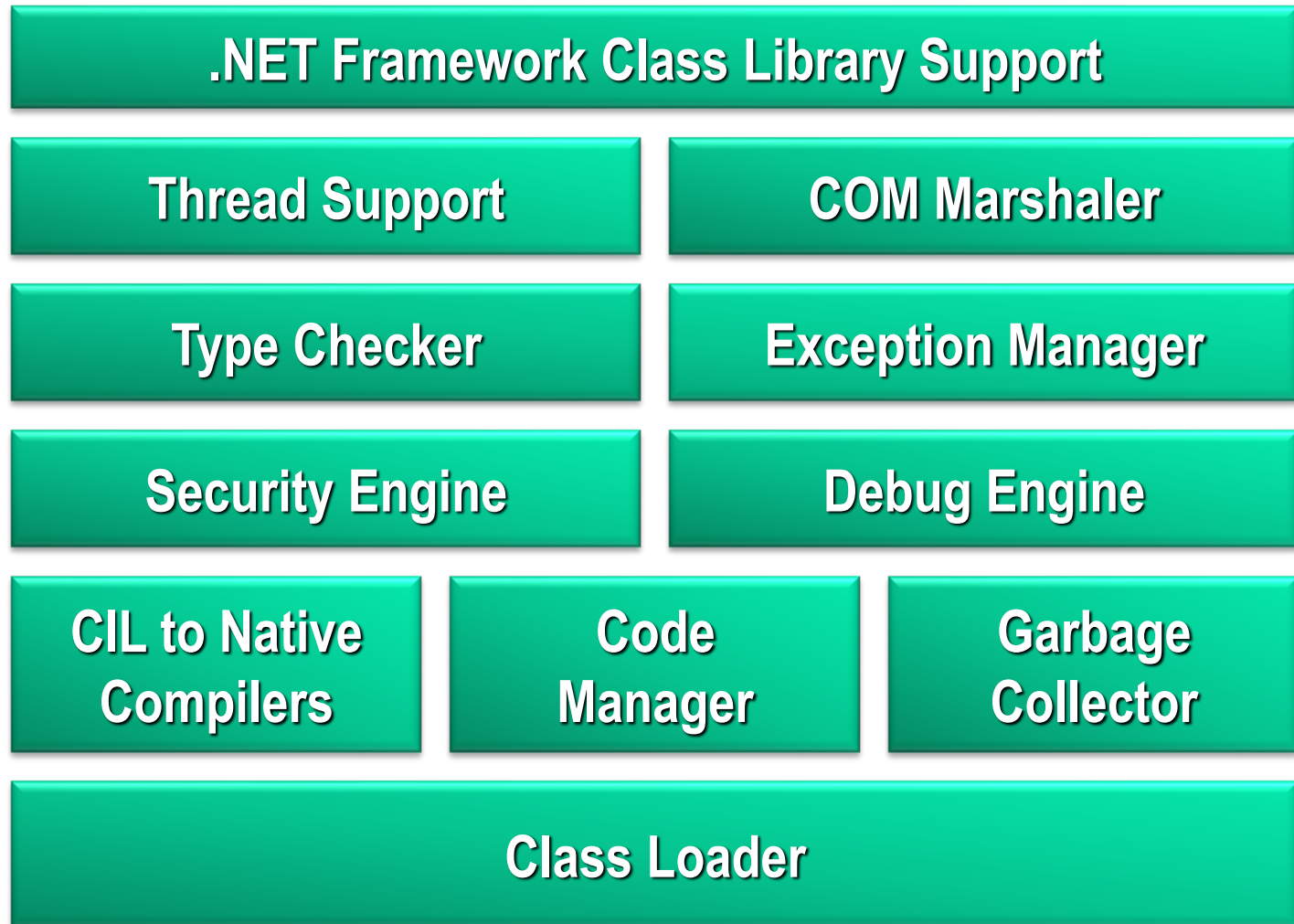
**Dr. Joachim Fuchs**

Berater, Softwareentwickler, Trainer

# The .NET Framework Components 2001



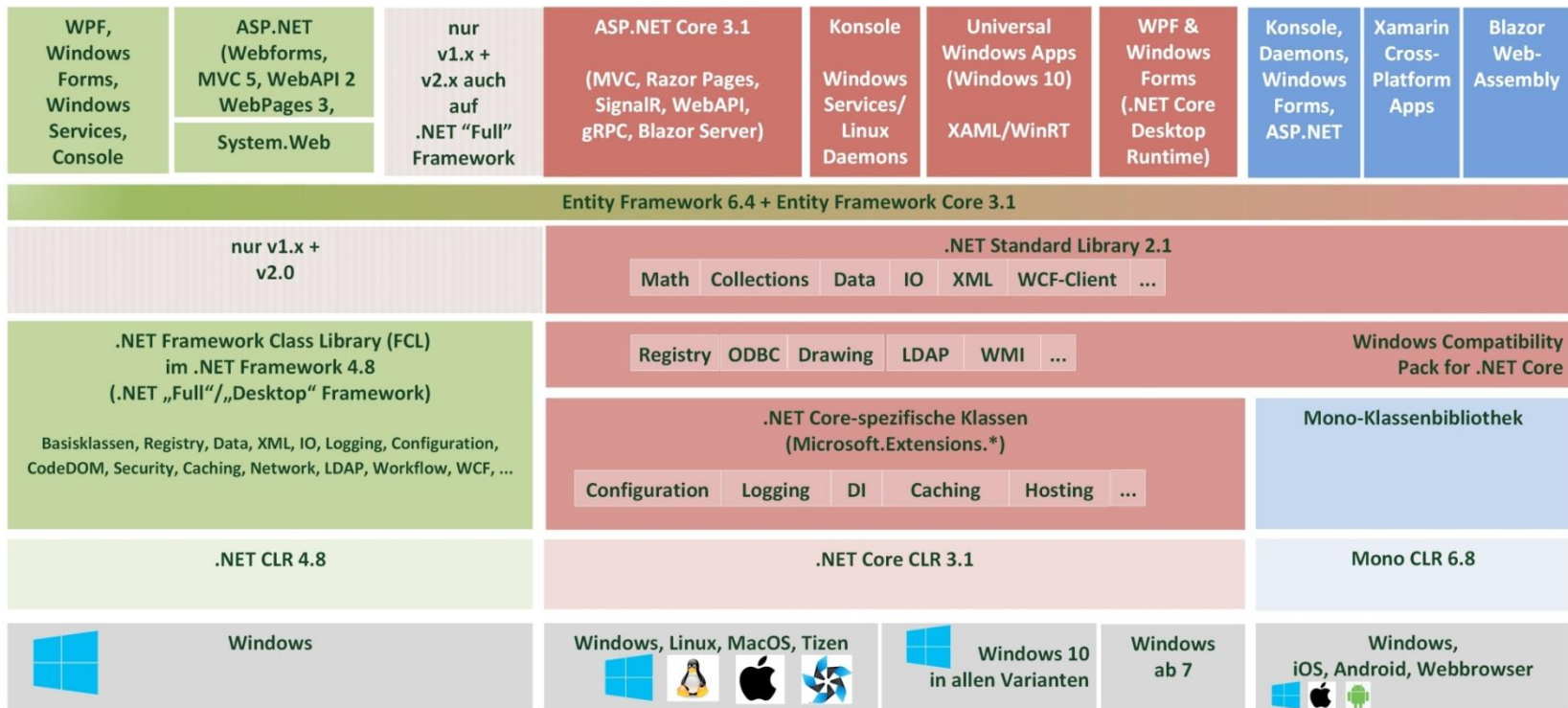
# The Common Language Runtime Components



# Anwendungsmodelle in .NET und .NET Core

## Die .NET-Familie 2020

© Dr. Holger Schwichtenberg, [www.IT-Visions.de](http://www.IT-Visions.de), Stand 10.02.2020



Quelle: H. Schwichtenberg, <https://www.heise.de/developer/artikel/Der-Stand-der-NET-Familie-zum-Jahresbeginn-2020-4656530.html>

# Versionen

- 2001.NET 1.0 CLR 1.0 C# 1.0 ab NT4
- 2003.NET 1.1 CLR 1.1
- 2005.NET 2.0 CLR 2.0 C# 2.0 ab Windows 2000
- 2006/7 .NET 3.0 ab Windows XP
- =.NET 2.0
- + WPF + WCF
- + WF + Cardspace
- 2007/8 .NET 3.5 CLR 2.0 C# 3.0 LINQ
- 2008.NET 3.5 SP1
- 2010.NET 4.0 CLR 4.0 C# 4.0
- 2012 .NET 4.5 CLR 4.0 C# 5.0 ab Vista
- 2015 .NET 4.6 / Core CLR 4.0 C# 6.0
- 2017 .NET 4.7 / Core 2.0 C# 7.x
- 2019 .NET 4.8 / Core 3.1 C# 8.0

# .NET Core Versionen

| Version       | Original Release Date | Latest Patch Version | Patch Release Date | Support Level | End of Support    |
|---------------|-----------------------|----------------------|--------------------|---------------|-------------------|
| .NET Core 3.1 | December 3, 2019      | 3.1.0                | 3.1.0              | LTS           |                   |
| .NET Core 3.0 | September 23, 2019    | 3.0.1                | November 19, 2019  | Maintenance   | March 3, 2020     |
| .NET Core 2.2 | December 4, 2018      | 2.2.8                | November 19, 2019  | Maintenance   | December 23, 2019 |
| .NET Core 2.1 | May 30, 2018          | 2.1.14               | November 19, 2019  | LTS           | August 21, 2021   |
| .NET Core 2.0 | August 14, 2017       | 2.0.9                | July 10, 2018      | EOL           | October 1, 2018   |
| .NET Core 1.1 | November 16, 2016     | 1.1.13               | May 14, 2019       | EOL           | June 27 2019      |
| .NET Core 1.0 | June 27, 2016         | 1.0.16               | May 14, 2019       | EOL           | June 27 2019      |

Quelle: H. Schwichtenberg, <https://www.heise.de/hintergrund/Umstieg-auf-.NET-Core-Migrieren-oder-nicht-migrieren-4628946.html?seite=2>

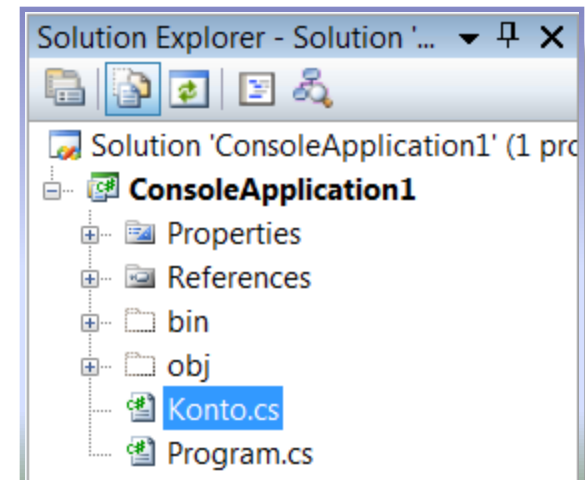
aktuell (2021): .NET 5

ab November 2021: .NET 6

siehe [https://www.dotnetframework.de/dotnet/DOTNET\\_Geschichte\\_Zukunft.aspx](https://www.dotnetframework.de/dotnet/DOTNET_Geschichte_Zukunft.aspx)

# Projektaufbau

- Projektmappe
  - enthält ein oder mehrere Projekt(e)
  - in der Praxis ist ein Projekt eine .NET-Assembly
  - Projekt
    - unterschiedliche Typen (Konsolenanwendung, Bibliothek, Windows-Forms-Anwendung, WPF-Anwendung, Web-Anwendung usw.)
    - definiert Datentypen (Klassen, Strukturen)
    - kann auf andere Assemblies verweisen (Knoten References)



# Grundlegende Syntax

- **Kommentare**

- **// und /\* \*/**

`int i = 0; // Kommentiertes Statement`

`// Kommentar zur Lage der Nation`

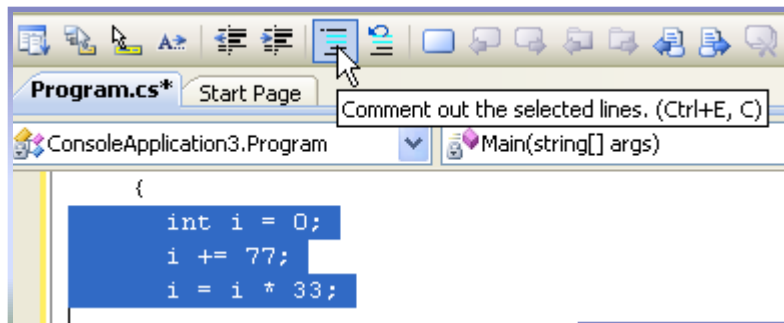
`// Es geht weiter...`

`/* Auch mehrzeilige Kommentare sind möglich,  
* werden aber nur noch selten benutzt,  
* da sie durch einzeilige Kommentare in Verbindung  
* mit Visual-Studio-Unterstützung ersetzt werden können  
*/`



# Kommentare

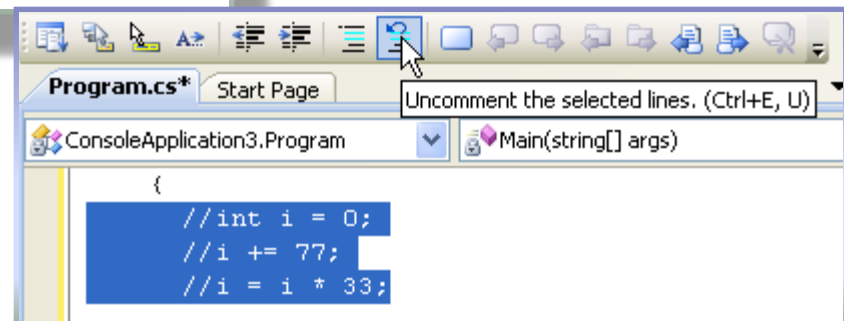
- Markierte Blöcke aus- und einkommentieren



Program.cs\* Start Page

ConsoleApplication3.Program Main(string[] args)

```
{  
    int i = 0;  
    i += 77;  
    i = i * 33;  
}
```



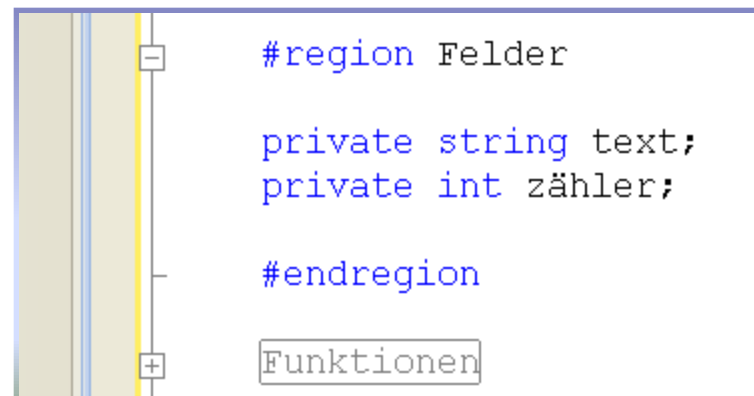
Program.cs\* Start Page

ConsoleApplication3.Program Main(string[] args)

```
{  
    //int i = 0;  
    //i += 77;  
    //i = i * 33;  
}
```

# Kommentare

- `#region` - `#endregion`
  - Blöcke zusammenfassen
  - Können im Editor auf- und zugeklappt werden



# Kommentare

- **Documentation Comments (XML-Kommentare)**
  - Öffentliche Klassen und deren Member dokumentieren
  - Intellisense-Unterstützung
  - HTML-Dokumentation mit zusätzlichen Tools wie Sandcastle oder NDOC

# Kommentare

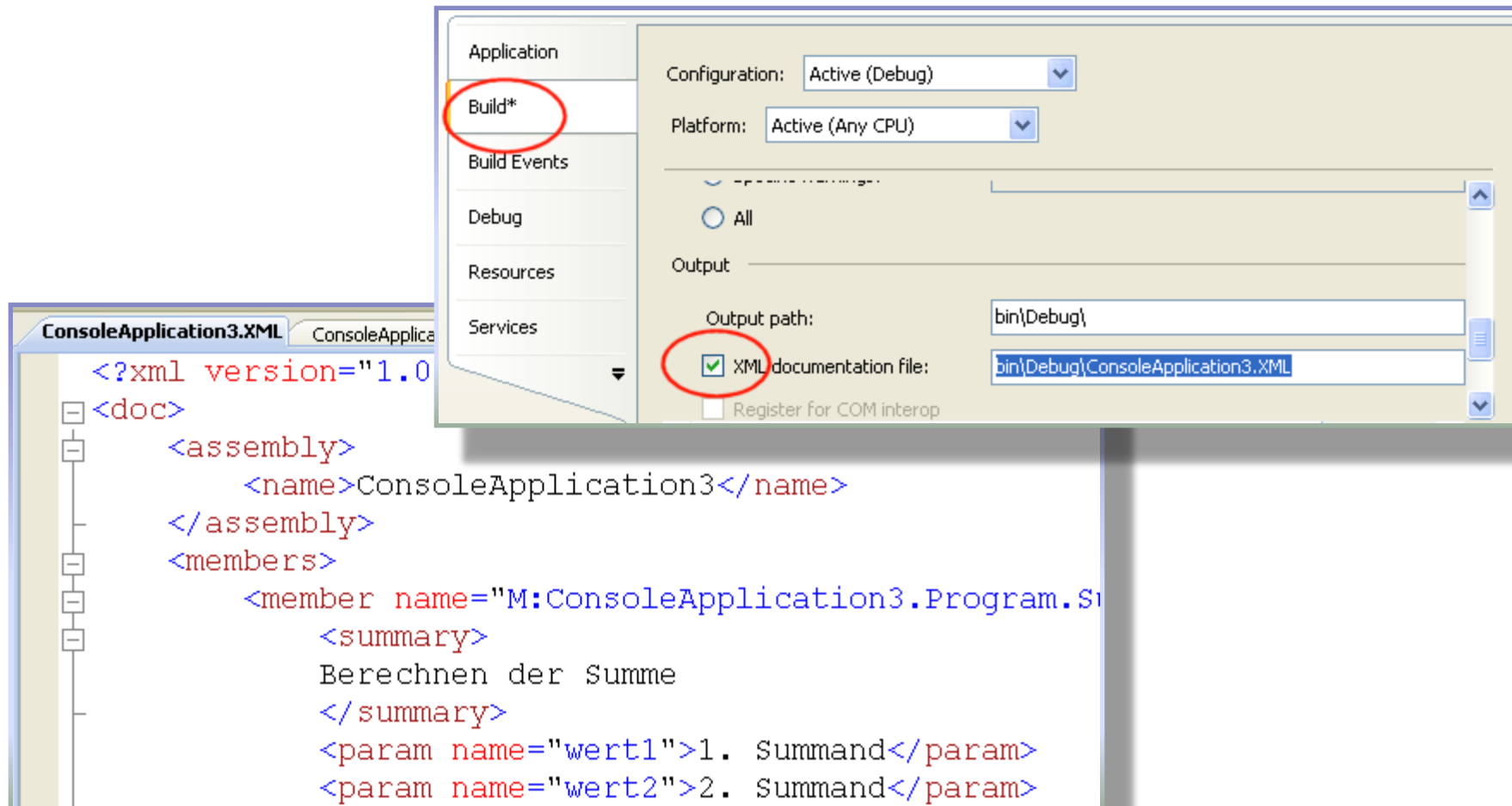
- **Documentation Comments (XML-Kommentare)**

```
/// <summary>  
/// Berechnen der Summe  
/// </summary>  
/// <param name="wert1">1. Summand</param>  
/// <param name="wert2">2. Summand</param>  
/// <returns>Ergebnis laut Summenformel</returns>  
public int Summe(int wert1, int wert2)  
{  
    return wert1 + wert2;  
}
```

Summe (|  
int Program.Summe (int wert1, int wert2)  
wert1:↓  
1. Summand

# Kommentare

- **Documentation Comments (XML-Kommentare)**



# Grundlegende Syntax

- **Lokale Variablen**
  - **Definition**
  - **Wertzuweisung, Verwendung**
  - **Variablen müssen vor der ersten Verwendung initialisiert werden**

```
string text;  
double x;  
int i = 0;  
  
Console.WriteLine(x);
```

(local variable) double x

Error:

Use of unassigned local variable 'x'

# Grundlegende Syntax

- **Operatoren**

- **Arithmetik:** +, -, \*, /, %
- **Vergleiche:** ==, !=, <, >, <=, >=
- **Zuweisung, Kombination:** =, +=, -=, \*= ...
- **Bit-Verknüpfungen:** &, |, ~, ^, <<, >>
- **Boolesche Verknüpfungen:** &&, ||, !
- **trinärer Operator:** ?

# Grundlegende Syntax

- **Kontrollanweisungen**
  - **if / else**

```
int n = 7;
bool b = n > 4;

if (n == 7)
    Console.WriteLine("n hat den Wert Sieben");
else
    Console.WriteLine("n hat einen anderen Wert");

if (b)
{
    Console.WriteLine("Die Bedingung ist erfüllt");
    Console.WriteLine("Mehrere Anweisungen müssen als Block " +
        "zusammengefasst werden");
}
```



# Grundlegende Syntax

- **Kontrollanweisungen**
  - **while, do ... while**
    - **Kopf- und fußgesteuerte Schleifen**

```
while (n < 100)
{
    Console.WriteLine("n: " + n);
    n = n + 12;
}

do
{
    Console.WriteLine("n: " + n);
    n = n + 12;
} while (n < 100);
```

# Grundlegende Syntax

- **Kontrollanweisungen**

- **for**

```
for (int i = 0; i < 100; i++)  
{  
    Console.WriteLine(i);  
}
```

- **foreach**

```
foreach (char zeichen in "Hallo Welt")  
{  
    Console.WriteLine(zeichen);  
}  
  
List<Konto> konten = new List<Konto>();  
  
foreach (Konto konto in konten)  
{  
    konto.Einzahlen(12345);  
}
```

# Grundlegende Syntax

- **Kontrollanweisungen**

- **switch**

```
switch (n)
{
    case 1:
        Console.WriteLine("Eins");
        break;

    case 2:
    case 3:
    case 4:
        Console.WriteLine("zwei oder drei oder vier");
        break;

    default:
        Console.WriteLine("Was anderes...");
        break;
}
```

# Grundlegende Syntax

- **Kontrollanweisungen**
  - **break**
    - **Beenden einer Schleife**
    - **Verlassen eines switch-case-Blocks**
  - **return**
    - **Verlassen einer Methode**

```
foreach (Konto konto in konten)
{
    if(konto.Saldo > 999999)
        break;

    if (konto.Saldo < 0)
        return;
}
```

# Grundlegende Syntax

- **Methoden**
  - **Definition, Aufruf**

```
int z = Summe(2, 33);  
TuWas("C# lernen");  
  
}  
  
static int Summe(int wert1, int wert2)  
{  
    return wert1 + wert2;  
}  
  
static void TuWas(string wasDenn)  
{  
    Console.WriteLine("Tut was: " + wasDenn);  
}
```

# Grundlegende Syntax

- **Methoden**
  - **Überladungen**
    - Derselbe Name
    - Unterschiedliche Parameterliste (Typen, Anzahl)
    - Rückgabotyp spielt keine Rolle
    - Compiler muss beim Aufruf erkennen können, welche Methode ausgeführt werden soll

# Grundlegende Syntax

- **Methoden**
  - **Überladungen**

```
static void Rechnen()  
{  
    int n1 = Summe(1, 2);  
    int n2 = Summe(3, 4, 5);  
    double d = Summe(1.4, 5.0);  
}
```

```
static int Summe(int wert1, int wert2) ...  
static double Summe(double wert1, double wert2) ...  
static int Summe(int wert1, int wert2, int wert3) ...
```

# Grundlegende Syntax

- **Methoden**
  - **params**-Schlüsselwort
    - **Variable Anzahl von Parametern**
    - **Nur der letzte Parameter kann mit params gekennzeichnet werden**



# Grundlegende Syntax

- **Methoden**
  - **params-Schlüsselwort**

```
static int Summe(params int[] zahlen)
{
    int summe = 0;

    foreach (int zahl in zahlen)
        summe += zahl;

    return summe;
}
```

```
static void Rechnen()
{
    int n;
    n = Summe();
    n = Summe(1);
    n = Summe(2, 3);
    n = Summe(3, 4, 6, 25, 77, 2, 44, 646);
}
```

# Grundlegende Syntax

- **Methoden**
  - Parameterübergabe **byValue** / **byRef**
    - **Standard: byValue**
    - **ref**: Übergabe einer initialisierten Variablen
    - **out**: Übergabe einer nicht initialisierten Variablen
    - **ref** / **out** muss immer paarweise angegeben werden (Funktionsdeklaration / -aufruf)

# Grundlegende Syntax

- **Methoden**

- **Parameterübergabe** **byValue** / **byRef**

- **ref**

```
static void Vertauschen(ref int a, ref int b)
{
    int hilf = a;
    a = b;
    b = hilf;
}
```

```
static void Test1()
{
    int x = 1;
    int y = 2;

    Vertauschen(ref x, ref y);
}
```

# Grundlegende Syntax

- **Methoden**

- **Parameterübergabe** **byValue** / **byRef**

- **out**

```
static void Initialisieren(out string bezeichnung, out int zahl)
{
    bezeichnung = "irgendwas";
    zahl = 42;
}
```

```
static void Test2()
{
    string text;
    int wert;
    Initialisieren(out text, out wert);
}
```

# Strukturen (Wertetypen)

- Standard-Wertetypen im Framework (Auswahl)

| C#      | Bytes | System. |
|---------|-------|---------|
| bool    |       | Boolean |
| int     | 4     | Int32   |
| long    | 8     | Int64   |
| float   | 4     | Single  |
| double  | 8     | Double  |
| decimal | 12    | Decimal |
| char    | 2     | Char    |

# Strukturen (Wertetypen)

- Definition

- Inhalt sollte nur aus Wertetypen bestehen
- klein halten (Daumenwert:  $\leq 16$  Byte)
- Programmierung eines parameterlosen Konstruktors nicht möglich
- keine Vererbung möglich
- Kopie bei der Zuweisung
- Speicherung abhängig vom Kontext

```
struct Punkt
{
    public int X;
    public int Y;
}
```

```
Punkt p;
p.X = 7;
p.Y = 8;

// Kopie anlegen
Punkt p2 = p;
```

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - Namensräume
  - Felder
  - Instanziierung (**new**)
  - Konstruktoren
  - **this**-Referenz
  - Sichtbarkeit: **private**, **public**, **internal**
  - Eigenschaften (**get**, **set**)
  - Methoden und Konstruktoren überladen
  - Instanz-Member vs. Klassen-Member (**static**)
  - Statische Konstruktoren
  - Lebensdauer von Objekten, Garbage Collection

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - Namensräume

```
namespace Firma
{
    public class Klasse1[...]

    namespace Bereich
    {
        namespace Produkt1
        {
            public class Klasse2 [...]
        }
    }

    namespace Abteilung.Unterabteilung.Basics
    {
        public class Klasse3[...]
    }
}
```



# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**

- **Namensräume**

- für jeden Datentyp muss der Namensraum entweder vollständig angegeben werden
    - oder per `using` am Anfang der Datei aufgeführt werden
    - `using` bindet nur genau einen Namensraum ein
    - bei Mehrdeutigkeiten muss die vollständige Qualifizierung erfolgen
    - `using` bindet keine Bibliotheken ein, sondern stellt nur eine syntaktische Vereinfachung dar!

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**

- **Felder (Fields)**

- Beschreiben den Zustand eines Objekts
    - Werden meist mit `private` deklariert, um sie vor Änderungen von außen zu schützen
    - Werden grundsätzlich initialisiert

```
class Konto
{
    // Felder (Fields)
    private int kontonummer;
    private string inhaber = "unbekannt";
    private double saldo;
}
```

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - Konstruktoren
  - Instanziierung (**new**)

```
public Konto() : this(0, "n.n.", 0)
{
}

public Konto(int kontonummer, string inhaber, double saldo)
{
    this.kontonummer = kontonummer;
    this.inhaber=inhaber;
    this.saldo = saldo;
}
```

```
Konto k1 = new Konto();
Konto k2 = new Konto(1000, "Dagobert", 9999999);
```

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - **Objekte sind Instanzen von Klassen**
  - **für den Zugriff auf ein Objekt wird eine Referenz benötigt**
  - **eine Referenz verweist entweder auf ein gültiges Objekt oder besitzt den Wert `null`**
  - **sind zwei Referenzen identisch, verweisen sie auf dasselbe Objekt**
  - **die Operatoren `==` und `!=` vergleichen im Normalfall die Referenzen und nicht die Objektzustände, können aber überladen werden**

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - **Objekte besitzen**
    - einen Zustand (beschrieben durch die Felder)
    - ein Verhalten (beschrieben durch Methoden und Eigenschaften)
    - eine Identität (beschrieben durch die Referenz)

# Klassen (Referenztypen) Teil 1

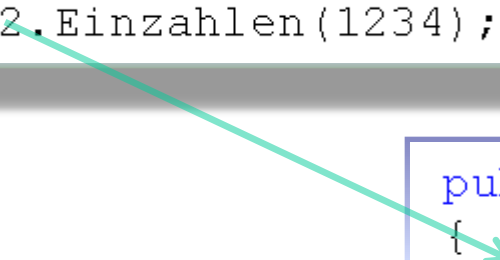
- **Klasse – Objekt – Referenz**

- **this-Referenz**

- verweist in Instanz-Methoden auf das Objekt, für das die Methode aufgerufen wurde
    - kann vom Compiler automatisch eingefügt werden, sofern der Member-Name im jeweiligen Kontext eindeutig ist

```
k2.Einzahlen(1234);
```

```
public void Einzahlen(double betrag)
{
    this.saldo += betrag;
}
```



# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - **Sichtbarkeit**

| C#                              | ist bekannt  |
|---------------------------------|--|
| private                         | nur in der Klasse  |
| public                          | überall  |
| protected (anders als in Java!) | nur in der Vererbungshierarchie                            |
| internal                        | innerhalb der Assembly wie public, außerhalb wie private   |
| protected internal              | innerhalb der Assembly wie public, außerhalb wie protected |

**Achtung! private etc. ist KEIN Datenschutz, sondern nur ein Strukturierungshilfsmittel!**

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - **Eigenschaften (get, set) (Properties)**
    - sehen nach außen wie Felder aus
    - kapseln intern zwei Methoden
    - set {} kann entfallen -> ReadOnly-Eigenschaft
    - get {} kann entfallen -> WriteOnly Eigenschaft
    - vielfältige Unterstützung in Visual Studio

```
public string Inhaber
{
    get { return inhaber; }
    set { inhaber = value; }
}

public double Saldo
{
    get { return saldo; }
}
```



# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - **Eigenschaften (get, set) (Properties)**
    - viele Mechanismen des Frameworks greifen auf öffentliche Eigenschaften zurück und ignorieren Felder
    - vereinfachte Syntax kapselt verdecktes privates Feld

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

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - **Eigenschaften (get, set) (Properties)**
    - die Sichtbarkeit des get- oder set- Accessors lässt sich zusätzlich einschränken

```
public int Kontonummer
{
    get { return kontonummer; }
    private set { kontonummer = value; }
}

public int BLZ { get; private set; }
```

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - **Methoden und Konstruktoren**
    - **können auf derselben Ebene überladen werden**
  - **Methoden können auch über die Vererbungshierarchie hinweg überladen werden**

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**

- **Instanz-Member vs. Klassen-Member (static)**

- gehören zur Klasse, nicht zur Instanz
    - können von außen nur über den Klassennamen referenziert werden

- **Statische Felder können initialisiert werden**

- **Statische Methoden und Eigenschaften besitzen keine this-Referenz**

```
private static int nächsteKontonummer = 1000;  
public static int NächsteKontonummer  
{  
    get { return nächsteKontonummer; }  
}
```

```
Console.WriteLine(Konto.NächsteKontonummer);
```

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**
  - **Statische Konstruktoren**
    - ein einziger pro Klasse möglich
    - besitzt keine Modifizierer und keine Parameter
    - wird ausgeführt, bevor ein Zugriff auf die Klasse erfolgt
    - wird nach der Initialisierung statischer Felder ausgeführt

```
static Konto()  
{  
    // hier vielleicht eine DB-Abfrage  
    nächsteKontonummer = 12345;  
}
```

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**

- **Lebensdauer von Objekten, Garbage Collection**

- Objekte werden im Programm nur instanziiert (new)
    - es gibt keine Möglichkeit, angelegte Objekte zu löschen
    - wenn das Programm keine Referenz eines Objektes besitzt, kann (aber muss nicht) die Speicherverwaltung das Objekt vom Heap entfernen
    - die Speicherverwaltung läuft asynchron (eigener Thread), ihr Eingreifen kann nicht vorhergesagt werden
    - Eingriffe in die Speicherverwaltung sind zwar theoretisch möglich, sollten aber unbedingt vermieden werden

# Klassen (Referenztypen) Teil 1

- **Klasse – Objekt – Referenz**

- **Lebensdauer von Objekten, Garbage Collection**

- Destruktoren (Finalizer) können programmiert werden, ihr Aufruf ist jedoch nicht vorhersagbar
    - die Speicherverwaltung bereinigt ausschließlich den managed Heap. Externe Ressourcen werden NICHT berücksichtigt (siehe Dispose-Pattern)

# Typumwandlungen und –prüfungen

- **Typecasts**
- **is-Operator**
- **as-Operator**



# Typumwandlungen und –prüfungen

- **Typecasts**

- implizit oder explizit
- werden immer zur Compile- und zur Laufzeit geprüft
- führen zu einer Exception, wenn zur Laufzeit der Typ nicht passt

```
object obj = new Konto();  
Konto k3 = (Konto)obj;  
  
double wert = 123.456;  
int ganzzahl = (int)wert;
```

# Typumwandlungen und –prüfungen

- **is-Operator**
  - ermöglicht die Typprüfung zur Laufzeit

```
if (obj is Konto)
{
    Konto dasKonto = (Konto)obj;
}
```

- **as-Operator**
  - kombiniert Typecast und is-Operator
  - löst keine Exception aus

```
Konto einKonto = obj as Konto;
if(einKonto !=null)
    Console.WriteLine(einKonto.Inhaber);
```

# ??-Operator (null-coalescing)

- prüft 1. Operanden auf null
- Ergebnis ist
  - 1. Operand, falls dieser ungleich null ist
  - sonst 2. Operand

Konto gefunden = ...

Konto konto = gefunden ?? new Konto { Inhaber = "niemand" };

# Boxing

- Tritt auf, wenn einer Object-Referenz ein Wertetyp zugewiesen wird
- Geboxte Werte werden auf dem Heap abgelegt und sind schreibgeschützt
- Mehrmaliges Boxing desselben Wertes führt zu mehreren Objekten unterschiedlicher Identität
- Unboxing erfordert einen Typecast mit dem Typ des geboxten Wertes

# Boxing

```
// Boxing eines Integer-Wertes  
object einObjekt = 123;  
int eineZahl = (int)einObjekt;  
  
// Boxing eines Double-Wertes  
einObjekt = 1.5;  
double eineAndereZahl = (double)einObjekt;  
  
// doppelter Typecast  
eineZahl = (int)(double)einObjekt;
```

```
object obj1 = 7;  
object obj2 = 7;  
Console.WriteLine(obj1 == obj2);
```

false

# Generische Datentypen

- Spracherweiterung in C# 2.0
- Entsprechen Templates aus C++
- Erlauben typneutrale Programmierung
- Eine oder mehrere Typvariablen möglich
- Für die Typvariablen können Bedingungen (Constraints) festgelegt werden
- Generische Klassen sind Metaklassen und somit nicht instanzierbar

```
class GenerischeBeispielklasse<T>  
{  
    public void TuWas(T womit) { }  
    private T daten;  
}
```

# Generische Datentypen

- Constraints (Einschränkungen)
  - Basisklasse
  - Schnittstelle(n)
  - class (Typ muss ein Referenztyp sein)
  - struct (Typ muss ein Wertetyp sein)
  - new() (Typ muss einen öffentlichen, parameterlosen Konstruktor besitzen)

```
class GenerischeBeispielklasse<T> where T : class, new() ...
```

```
GenerischeBeispielklasse<Konto> gbk =  
    new GenerischeBeispielklasse<Konto>();  
  
gbk.TuWas(new Konto());
```

# Generische Datentypen

- **Nullable<T>**
  - ergänzt null-Referenz für Wertetypen
  - ermöglicht Abfrage ob Wert vorhanden oder nicht

```
Nullable<int> zahl = null;  
if(zahl.HasValue)  
    Console.WriteLine("Zahl vorhanden: " + zahl.Value);  
  
zahl = 123;  
Console.WriteLine(zahl.GetValueOrDefault(42));
```

```
// Kurzform  
int? zahl = null;  
DateTime? datum = DateTime.Now;
```



# Auflistungen

- **Arrays**
  - **Speicherung von Arrays**
  - **Deklaration und Verwendung**
  - **Eigenschaften und Methoden von Arrays**
  - **Statische Methoden der Klasse Array**
- **Object-basierte Auflistungen**
  - **ArrayList**
  - **Hashtable**
  - **Queue**
  - **SortedList**

# Auflistungen

- **Arrays**

- **sind Referenztypen, liegen also immer auf dem Heap**
- **beschreiben einen zusammenhängenden, linear adressierbaren Speicherbereich von Werten gleichen Typs**
- **können ein- oder mehrdimensional sein**
- **Arrays müssen instanziiert werden**
- **nach der Instanzierung ist die Größe unveränderlich**
- **Array-Elemente werden immer initialisiert (Default-Werte)**

# Auflistungen

- **Arrays**
  - **Deklaration und Verwendung (eindimensional)**

```
int[] zahlen1;  
zahlen1 = new int[5];  
  
int[] zahlen2 = new int[5];  
int[] zahlen3 = new int[5] { 3, 6, 11, 35, 66 };  
int[] zahlen4 = new int[] { 5, 77, 262 };  
  
// Verkürzte Initialisierung nur in Verbindung mit Deklaration  
int[] zahlen5 = { 4, 9, 23 };
```

# Auflistungen

- **Arrays**
  - **Deklaration und Verwendung (mehrdimensional)**

```
int[,] lottoschein = new int[3, 6];  
string[,] texte = {  
    {"1", "eins"},  
    {"2", "zwei"},  
    {"100", "Hundert"}  
};
```

# Auflistungen

- **Arrays**
  - **Eigenschaften und Methoden von Arrays**

| Name                     | Bedeutung                              |
|--------------------------|--|
| Length                   | Anzahl aller Elemente                  |
| Rank                     | Anzahl der Dimensionen                 |
| GetLength(dimension)     | Anzahl der Elemente für eine Dimension |
| [] – Indexer             | Zugriff auf einzelnes Element          |
| GetUpperBound(dimension) | höchstzulässiger Index                 |
| CopyTo (überladen)       | Bereiche in anderes Array kopieren     |

# Auflistungen

- **Arrays**
  - **Statische Methoden der Klasse `Array`**

| Name             | Bedeutung                                    |
|------------------|--|
| Sort()           | Sortieren                                    |
| IndexOf()        | Lineare Suche                                |
| BinarySearch()   | Binäre Suche, erfordert vorherige Sortierung |
| Clear()          | Elemente auf Standardwert setzen             |
| Reverse()        | Reihenfolge umkehren                         |
| Copy (überladen) | Bereiche in anderes Array kopieren           |

# Auflistungen

- **Object-basierte Auflistungen**
  - ArrayList
  - Hashtable
  - Queue
  - SortedList
- **einsetzbar für alle Datentypen**
- **keine Typsicherheit zur Compile-Zeit**
- **alle Datenzugriffe arbeiten mit Object-Referenzen!**
- **beim Umgang mit Wertetypen erfolgt Boxing!**

# Auflistungen

- **Object-basierte Auflistungen**

- **ArrayList**

- speichert intern ein Array mit Object-Referenzen
    - verwaltet das interne Array automatisch
    - Hinzufügen über Add() oder Insert()
    - Methoden zum Suchen und Sortieren

```
System.Collections.ArrayList liste =  
    new System.Collections.ArrayList();  
liste.Add(123);  
liste.Add("Hallo");  
liste.Add(new Konto());
```



# Auflistungen

- **Object-basierte Auflistungen**

- **ArrayList**

- implementiert IEnumerable -> foreach...
    - Zugriff über Indexer (Typecast erforderlich)

```
foreach (object obj in liste)  
    Console.WriteLine(obj);  
  
string text = (string)liste[1];
```

# Auflistungen

- **Object-basierte Auflistungen**
  - andere Auflistungstypen

| Name       | Funktion         |
|------------|------------------|
| Hashtable  | Key-/Value-Liste |
| Queue      | FIFO-Liste       |
| Stack      | LIFO-Liste       |
| SortedList | Sortierte Liste  |

# Auflistungen

- **Generische Auflistungsklassen (C# 2.0)**
  - List
  - Dictionary
  - Weitere
- **werden typisiert definiert**
- **arbeiten intern mit den festgelegten Datentypen**
- **kein Boxing**
- **Typsicherheit zur Compile-Zeit**

# Auflistungen

- **Generische Auflistungsklassen (C# 2.0)**
  - Konzept (Beispiel ArrayList-Variante)

```
class GenerischeArrayList<TValue>
{
    private TValue[] daten=new TValue[5];
    public void Add(TValue data)...
    public TValue GetDataAt(int index)...
}
```

# Auflistungen

- **Generische Auflistungsklassen (C# 2.0)**
  - **List<T>**
    - generische Variante von ArrayList

```
List<int> zahlen = new List<int>();  
zahlen.Add(|
```

```
void List<int>.Add (int item)
```

**item:**

The object to be added to the end of the System.Collections.Generic.List<T>. The value can be null for reference types.

```
int zahl = zahlen[0];
```

# Auflistungen

- **Generische Auflistungsklassen (C# 2.0)**
  - **Dictionary<TKey, TValue>**
    - generische Variante von Hashtable

```
Dictionary<string, int> ziffern1 =  
    new Dictionary<string, int>();  
ziffern1.Add("eins", 1);  
ziffern1.Add("zwei", 2);  
ziffern1.Add("drei", 3);  
  
int z = ziffern1["zwei"];
```

# Klassen Teil 2

- **Schnittstellen**
  - Definition
  - Implizite Implementierung
  - Explizite Implementierung
  - Beispiele im Framework
- **Delegates**
  - Definition, Einsatz
  - Anonyme Methoden
  - Lambda-Ausdrücke

# Klassen Teil 2

- **Schnittstellen**

- **Definition**

- nur Deklarationen von
      - Methoden
      - Eigenschaften
      - Events
    - keine Implementierungen
    - keine Öffentlichkeits-Modifizierer
    - Schnittstellen können voneinander erben



# Klassen Teil 2

- **Schnittstellen**
  - **Definition**

```
public interface ISteuerobjekt
{
    string Steuernummer { get; set; }
    void SteuerAbführen();
    event EventHandler VorauszahlungFällig;
}
```

# Klassen Teil 2

- **Schnittstellen**
  - **Implizite Implementierung**
    - als öffentliche Member

```
class KFZ : ISteuerobjekt
{
    public string Steuernummer
    {get;set;}

    public void SteuerAbführen()
    {
    }

    public event EventHandler VorauszahlungFällig;
}
```

# Klassen Teil 2

- **Schnittstellen**

- **Explizite Implementierung**

- ohne Öffentlichkeits-Modifizierer
    - Zugriff nur über Referenz vom Typ des Interfaces

```
public class Hund : ISteuerobjekt
{
    string ISteuerobjekt.Steuernummer {get;set;}
    void ISteuerobjekt.SteuerAbführen() {}

    void TuWas()
    {
        ISteuerobjekt so = (ISteuerobjekt)this;
        so.SteuerAbführen();
    }
}
```

# Klassen Teil 2

- **Schnittstellen**
  - **Beispiele im Framework**
    - Comparable
    - Comparer
    - Enumerable
    - IList
    - Disposable

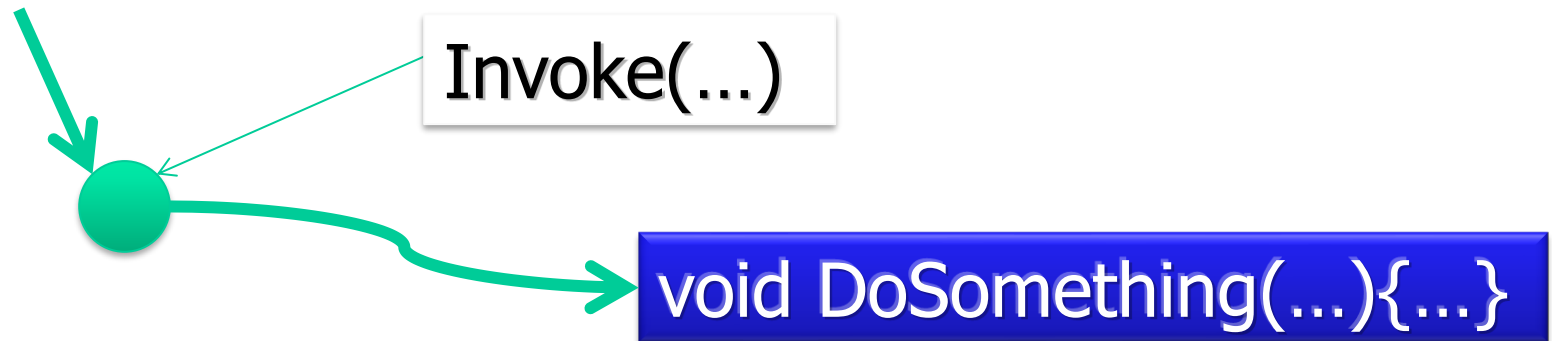
# Klassen Teil 2

- **Delegates**

- **Definition, Einsatz**

- Festlegen einer Methodensignatur für indirekte Funktionsaufrufe
    - Delegate-Definitionen sind Klassendefinitionen
    - eine Delegate-Instanz verweist auf
      - eine Methode
      - ein Objekt, falls es sich um eine Instanzmethode handelt
      - eventuell auf ein weiteres Delegate-Objekt (Multicast)
    - über die Methode Invoke() eines Delegate-Objektes werden alle angehängten Methoden ausgeführt

# Delegates



# Klassen Teil 2

- **Delegates**
  - **Definition, Einsatz**

```
public delegate void Rückruf(string info);  
public delegate int Berechnen(int wert1, int wert2);
```

```
static int Summe(int wert1, int wert2)...
```

```
// Instanzieren  
Berechnen methode = new Berechnen(Summe);  
  
// Ausführen  
methode.Invoke(3, 4);
```

```
// Vereinfachte Syntax  
Berechnen methode = Summe;  
methode(5, 6);
```

# Klassen Teil 2

- **Delegates**

- **Anonyme Methoden**

- „Inline“-Methodendeklaration
    - Referenz kann weitergegeben werden
    - Rückgabetyp ergibt sich auf return-Statement
    - darf auf lokale Variablen der umschließenden Methode zugreifen

```
Berechnen methode =  
    delegate(double a, double b)  
    {  
        return a - b;  
    };
```



# Klassen Teil 2

- **Delegates**

- **Lambda-Ausdrücke (C# 3.0)**

- neue Syntax für anonyme Funktionen
    - nutzt die Type-Inference des Compilers
    - benötigt für LINQ (Language Integrated Query)

```
Berechnen methode =  
    (wert1, wert2) =>  
    {  
        return wert1 * wert2;  
    };  
  
// oder noch kürzer  
  
methode = (wert1, wert2) => wert1 * wert2;
```

# Klassen Teil 3 und 4

- **Indexer**
- **Überladen von Operatoren**
- **Vererbung**
  - **Überschreiben von Methoden / Eigenschaften (virtual, override)**
  - **Sichtbarkeiten protected und protected internal**
  - **Verdecken (new)**
  - **Abstrakte Methoden, abstrakte Klassen**
  - **Besonderheiten bei der Implementierung von Schnittstellen**

# Klassen Teil 3

- **Indexer**

- ermöglichen Zugriffe über []-Signatur
- sind derzeit die einzige Möglichkeit, eine (die Standard-) Eigenschaft indizierbar zu machen

```
class Längenmaß
{
    private double längeInMeter;
    public double this[string einheit]
    {
        get {
            switch (einheit)
            {
                case "m": return längeInMeter;
                case "cm": return 100 * längeInMeter;
            }
        }
        set { ... }
    }
}
```

# yield return

## Enumeration mittels yield return (Generator Pattern)

```
static IEnumerable<string> GetStrings()
{
    yield return "Hallo";
    yield return "Welt";
    for (int i = 0; i < 10; i++)
    {
        yield return i.ToString();
    }
}
```

# Klassen Teil 3

- **Überladen von Operatoren**

- definiert Operatoren für bestimmte Datentypen neu
- gelegentlich sinnvoll für Vergleichsoperatoren
- nur als statische Methoden möglich

```
public static Längenmaß operator +(Längenmaß m1, Längenmaß m2)
{
    Längenmaß maß = new Längenmaß();
    maß.längeInMeter = m1.längeInMeter + m2.längeInMeter;
    return maß;
}
```

# Klassen Teil 4

- **Vererbung**

- Jede Klasse besitzt genau eine Basisklasse
- C# unterstützt keine Mehrfachvererbung
- Basisklasse ist implizit System.Object
- Methoden sind NICHT automatisch virtuell
- Nicht-virtuelle Methoden können verdeckt werden (Zauberwort new)
- Virtuelle Methoden können überschrieben werden (Zauberwort override)
- Mittels sealed können weitere Ableitungen verhindert werden

# Klassen Teil 4

- **Vererbung**

- **Überschreiben von Methoden / Eigenschaften (virtual, override)**
- **Auswahl der Methode zur Laufzeit, abhängig vom Objekttyp**

```
class A
{
    public virtual void TuWas ()
    {
    }
}

class B : A
{
    public override void TuWas ()
    {
        Console.WriteLine("Tut was...");
        base.TuWas ();
    }
}
```

# Klassen Teil 4

- **Vererbung**

- **Verdecken (new)**
- Auswahl der Methode zur Compile-Zeit, abhängig vom Typ der Referenzen

```
class A
{
    public void TuWas ()
    {
    }
}

class B : A
{
    public new void TuWas ()
    {
        Console.WriteLine("Tut was...");
        base.TuWas ();
    }
}
```



# Klassen Teil 4

- **Vererbung**

- **Abstrakte Klassen**

- Müssen mit `abstract` gekennzeichnet werden
    - Können nicht instanziiert werden
    - Abgeleitete Klassen müssen alle abstrakten Member implementieren oder selbst mit `abstract` gekennzeichnet werden

- **Abstrakte Methoden und Eigenschaften**

- Werden ebenfalls mit `abstract` gekennzeichnet
    - Besitzen keine Implementierung

# Klassen Teil 4

- **Vererbung**
  - **Abstrakte Klassen Methoden und Eigenschaften**

```
public abstract class Fahrzeug
{
    public abstract void Fahren(string wohin);
    protected abstract string Kenzeichen { get; set; }
}
```

```
public class Schiff : Fahrzeug
{
    ...
}
```



Implement abstract class 'Fahrzeug'

```
public class Schiff : Fahrzeug
{
    public override void Fahren(string wohin) ...
    protected override string Kenzeichen
    {
        get { return "K-1234"; }
        set { }
    }
}
```

# Klassen 4

- **Vererbung**

- **Abstrakte Klassen und Schnittstellen**

- Alle Member einer zu implementierenden Schnittstelle müssen implementiert oder deklariert werden, können aber mit **abstract** gekennzeichnet werden

```
public abstract class Fahrzeug : IComparable<Fahrzeug>
{
    public abstract int CompareTo(Fahrzeug other);
    ...
}
```

# Exceptions

- **Sinn und Zweck**
- **try – catch – finally**
- **Auslösen von Exceptions**
- **InnerException, Stacktrace**

# Exceptions

- **Sinn und Zweck**

- Abfangen nicht anderweitig abfragbarer Fehlersituationen
- Sollte, wie der Name schon sagt, nur die Ausnahme sein
- Weiterreichen von Fehlerzuständen, die auf der aktuellen Ebene nicht behoben werden können

# Exceptions

- **try – catch – finally**

```
private static int Berechnen(int a, int b)
{
    try
    {
        return a / b;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Fehler: " + ex.Message);
    }
    finally
    {
        Console.WriteLine("Wird immer ausgeführt");
    }
}
```

# Exceptions

- **Auslösen von Exceptions**

```
catch (Exception ex)
{
    throw; // aktuelle Exception weiterleiten

    // oder neue Exception auslösen
    throw new ApplicationException("Geht nicht...", ex);
}
```

oder Varianten davon

- **InnerException, Stacktrace**

```
catch (Exception ex)
{
    Console.WriteLine(ex.Message);

    if (ex.InnerException != null)
        Console.WriteLine(ex.InnerException.Message);

    Console.WriteLine(ex.StackTrace);
}
```

# Exceptions

- **Was kommt denn hierbei raus?**

```
public static int WelchesErgebnis()  
{  
    int x = 1;  
  
    try  
    {  
        x = 2;  
        return x;  
    }  
    finally  
    {  
        x = 3;  
    }  
}
```



# Strings

- **Speicherung**
- **Methoden für die String-Verarbeitung**
- **Formatierungen**
- **StringBuilder**

# Strings

- **Speicherung**

- **Strings sind Referenztypen!**
- **Strings sind immutable (unveränderbar)**
- **Operatoren ==, != vergleichen die Inhalte!**
- **Intern wird immer mit Unicode (2 Byte pro Zeichen) gearbeitet**
- **Unterscheidung **null**-Referenz und Leerstring**
- **Funktionen und Operatoren erzeugen neue Objekte**

```
string t1 = "Hallo";  
string t2 = new string('*', 30);  
string t3 = null;
```

# Strings

- **Instanzmethoden für die String-Verarbeitung**

| Methode / Eigenschaft    | Bedeutung          |
|--------------------------|--------------------|
| Length                   | Anzahl der Zeichen |
| IndexOf, IndexOfAny      | Suche              |
| StartsWith, EndsWith     | Anfang/Ende prüfen |
| Substring                | Teilstring         |
| ToLower, ToUpper         | Groß-/Kleinschrift |
| Trim, TrimEnd, TrimStart | Zeichen entfernen  |
| Insert, Remove, Replace  |                    |

# Strings

- **Statische Methoden und Operatoren der Klasse String**

| Methode / Eigenschaft | Bedeutung                        |
|-----------------------|----------------------------------|
| +, Concat             | Strings verknüpfen               |
| ==, !=                | Vergleiche                       |
| Format                | Formatierung                     |
| Join                  | Aufzählung                       |
| IsNullOrEmpty         | Prüfung auf Null- und Leerstring |
| Literale:             |                                  |
| @ "c:\A\B\x.txt"      |                                  |
|                       |                                  |

# Strings

- **StringBuilder**
  - Arbeitet mit vordefiniertem Puffer
  - Vermeidet häufige Objektinstanzierung bei String-Operationen

```
StringBuilder sb = new StringBuilder(1000);  
sb.Append("Hallo");  
sb.Append(" Welt");  
sb.Insert(6, "liebe ");  
string t4 = sb.ToString();
```

# C# 3.0 Spracherweiterungen

- Anonyme Typen
- Object initializer (`new Klasse(...){Eigenschaft = Wert}`)
- Vereinfachte Property-Syntax
- Extension Methods
- Lambda-Ausdrücke

# C# 4.0 Spracherweiterungen

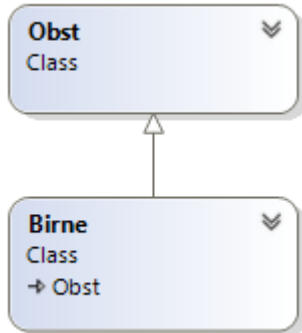
- Optionale und benannte Parameter
- dynamic
- Ko- und Kontravarianz

# Optionale Parameter

- Seit C# 4.0 möglich
- Definition der Parameter mit Standardwert
- Aufrufmöglichkeiten
  - klassisch (alle Werte in der richtigen Reihenfolge)
  - Angabe des Parameternamens gefolgt von einem Doppelpunkt und dem Parameterwert (Parametername : derWert)
  - Bei Angabe des Parameternamens muss beim Aufruf die Reihenfolge nicht mit der der Definition übereinstimmen.
  - Parameter können weggelassen werden und haben dann den angegebenen Standardwert
  - nicht-optionale Parameter müssen vor optionalen Parametern definiert werden



# Kovarianz




```
delegate T Anbauen<T>();

static Obst ObstAnbauen() { return new Obst(); }
static Birne BirneAnbauen() { return new Birne(); }
```

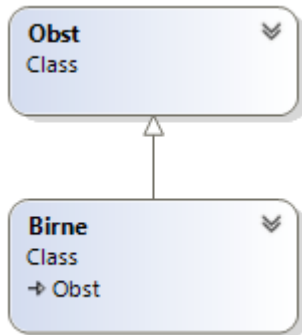
```
Anbauen<Obst> m4 = ObstAnbauen;
Anbauen<Birne> m5 = BirneAnbauen;
Anbauen<Obst> m6 = m5; // Fehler
```

Lösung:

```
delegate T Anbauen<out T>();
```



# Kontravarianz



```
delegate void Verarbeiten<T>(T obst);

static void VerarbeiteObst(Obst obst) { }
static void VerarbeiteBirne(Birne birne) { }
```

```
Verarbeiten<Obst> m1 = VerarbeiteObst;
Verarbeiten<Birne> m2 = VerarbeiteBirne;
Verarbeiten<Birne> m3 = m1; // Fehler
```

Lösung:

```
delegate void Verarbeiten<in T>(T obst);
```

A red arrow points to the **in** keyword in the delegate signature `Verarbeiten<in T>`.

# C# 5.0

- `async / await`
- `CallerMemberAttribute`

# Neuerungen in C# 6

- Auto-property-Verbesserungen
- Expression-bodied function members
- nameof
- Using static
- Null-conditional operators
- String interpolation
- Index initializers
- Exception filters
- Await in catch- und finally- Blöcken

<https://github.com/dotnet/roslyn/wiki/New-Language-Features-in-C%23-6>

<https://msdn.microsoft.com/en-us/magazine/dn879355.aspx>

# Initializer für Auto-Properties, get-only

```
class Person
{
    // Initialisierte Properties
    public string Name { get; set; } = "unbekannt";

    // Auch ohne Setter möglich
    public string Wohnort { get; } = "nirgends";

    public Person(string wohnort)
    {
        // oder Initialisierung im Konstruktor
        this.Wohnort = wohnort;
    }
}
```

# Methoden als Lambda-Ausdruck

## Definition von Methoden und Operatoren

```
public Längenmaß Mult(double faktor) => new Längenmaß  
{ LängeInMeter = this.längeInMeter * faktor };
```

```
public static Längenmaß operator *  
(Längenmaß maß, double faktor) => maß.Mult(faktor);
```

```
public static implicit operator string (Längenmaß maß)  
=> maß.längeInMeter.ToString("0.00") + " m";
```

# Methoden als Lambda-Ausdruck

## Verwendung wie Methoden / Operatoren

```
Längenmaß m1 =  
    new Längenmaß { LängeInMeter = 100 };  
Console.WriteLine(m1);  
Längenmaß m2 = m1.Mult(5);  
Console.WriteLine(m2);  
Längenmaß m3 = m1 * 7;  
Console.WriteLine(m3);
```

```
100,00m  
500,00m  
700,00m
```

# Readonly-Property als Lambda-Ausdruck

```
private int alter;  
public bool IstErfahren => alter >= 50;
```

z. B. Bindung hieran in WPF:

```
<DataTrigger Binding="{Binding IstErfahren}" Value="True">
```



# nameof

```
public int Alter
{
    get { return alter; }
    set
    {
        alter = value;
        OnPropertyChanged();
        OnPropertyChanged(nameof(IstErfahren));
    }
}
```

```
protected void OnPropertyChanged
([CallerMemberName] string propertyName = null)...
```

# using static

- Erlaubt den Zugriff auf statische Member einer Klasse, ohne Angabe des Klassennamens

statt

```
Console.WriteLine(Math.Sin(1) * Math.Cos(2));
```

geht jetzt auch:

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

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

...

```
WriteLine(Sin(1) * Cos(2));
```

**Achtung!**

**Nicht für Erweiterungsmethoden  
gedacht!**

# null-conditional operators

- Vereinfachen die Prüfung auf null

```
int? anzahl = firma?.Mitarbeiter?.Count;  
ergibt tatsächliche Anzahl oder null
```

```
Person ersterMitarbeiter = firma?.Mitarbeiter?[0];  
ergibt erstes Listenelement oder null
```

```
int anzahl = firma?.Mitarbeiter?.Count ?? 0;  
ergibt tatsächliche Anzahl oder 0
```

# null-conditional operators

- Verkettung erlaubt

```
int? länge = firma?.Mitarbeiter?[0]?.Name?.Length;
```

ergibt tatsächlichen Wert, wenn alle überprüften Referenzen ungleich null sind, sonst **null**

# null-conditional operators

- Einsatz bei Events / Delegates
  - nur über expliziten Aufruf von *Invoke*

```
protected void OnPropertyChanged(... string propertyName...)
{
    PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyName));
}
```

```
public event PropertyChangedEventHandler PropertyChanged;
```

# String interpolation

- Vereinfachung von String.Format
  - Parameter inline definieren, statt Indizierung
  - Escape-Sequenz für Klammern: {{ bzw. }}

```
string s1 = $"Länge: {m1.LängeInMeter:0.00}m";
```

-> Länge: 100,00m

```
string s2 = $"Mitarbeiter: {ersterMitarbeiter.Name,25}, Alter:  
                {ersterMitarbeiter.Alter:000} Jahre";
```

-> Mitarbeiter: Peter Schmidt, Alter: 045 Jahre

```
string json = $"{{alter:\"{ersterMitarbeiter.Alter}\",  
                name:\"{ersterMitarbeiter.Name}\"}}";
```

-> {alter:"45", name:"Peter Schmidt"}

# Index initializers

- alte Syntax (verwendet Methode Add):

```
Dictionary<string, int> dict1 = new Dictionary<string, int>  
{ { "eins", 1 }, ...{ "drei", 3 } };
```

- doppelte Schlüssel nicht erlaubt bei Dictionary.Add

- neue Syntax (verwendet Indexer):

```
Dictionary<string, int> dict2 = new Dictionary<string, int>  
{ ["eins"] = 1, ["zwei"] = 2, ["eins"] = 42 };
```

- doppelte Schlüssel sind hier erlaubt bei Dictionary und überschreiben den vorherigen Wert

# Exception filters

- Neue Syntax

```
try {...}  
catch (AbcException ex) when (Ausdruck)  
{ ... }
```

- Ausdruck muss Booleschen Wert zurückgeben
  - true: Catchblock wird ausgeführt
  - false: Exception-Verarbeitung wird fortgeführt
- Mehrere Catchblöcke mit selbem Exception-Typ erlaubt
- Allgemein auch zum Loggen von Exceptions geduldet, obwohl es nicht dem Filtergedanken entspricht



# await in catch- und finally- Blöcken

- Das C#-Team hat nun doch eine Lösung gefunden, async in catch- und finally-Blöcken zu implementieren.
- Ab C# 6 ist nun der Einsatz von await auch hier erlaubt

# Reflection

- Typ-Objekte
- Felder, Eigenschaften, Methoden usw. abrufen
- Öffentliche und private Member
- Einsatzbereiche

# Interop

- [System.Runtime.InteropServices Namespace](#)
- [DllImportAttribute](#)
- [StructLayoutAttribute](#)
- [MarshalAsAttribute](#)
- [SafeHandle](#)
- [Marshal](#)
- [RuntimeEnvironment](#)

# Literatur – kostenlose Bücher

- Visual C# 2012
  - [http://openbook.rheinwerk-verlag.de/visual\\_csharp\\_2012/index.html](http://openbook.rheinwerk-verlag.de/visual_csharp_2012/index.html)
- C# Language Specification 5.0
  - <http://www.c-sharpcorner.com/ebooks/free/83/>
- LINQ Quick Reference with C#
  - <http://www.c-sharpcorner.com/ebooks/free/78/>
- The 68 things the CLR does before executing a single line of your code
  - <http://mattwarren.org/2017/02/07/The-68-things-the-CLR-does-before-executing-a-single-line-of-your-code/>
- Syntaxgegenüberstellung VB – C#  
[http://www.harding.edu/fmccown/vbnet\\_csharp\\_comparison.html](http://www.harding.edu/fmccown/vbnet_csharp_comparison.html)

# Am Ziel

Vielen Dank für Ihre  
Aufmerksamkeit

Fragen?



Jetzt oder später: (EMAIL)



Brauchen Sie Unterstützung  
bei .NET, SilverLight, SQL Server, SharePoint, Windows  
Server, BizTalk, CRM,u.v.a. Microsoft-Produkten?

- Beratung bei Einführung, Migration und Betrieb
- (Vor-Ort-)Schulungen, Workshops
- Coaching (Vor-Ort | Telefon | E-Mail | Online-Meeting)
- Support (Vor-Ort | Telefon | E-Mail | Online-Meeting)
- Entwicklung von Prototypen und Lösung

<http://www.IT-Visions.de>

Telefon 0201/7490-700

[hs@IT-Visions.de](mailto:hs@IT-Visions.de)