

# .NET: Fortschrittene Konzepte von C#

© J. Heinzelreiter  
Version 5.2

# Abgrenzung C# – Java/C++

- Merkmale von Java
  - OOP: Vererbung, dynamische Bindung, Interfaces
  - Metainformation,
  - Ausnahmebehandlung,
  - statische und starke Typisierung,
  - Garbage Collection.
- Merkmale von C++
  - Überladen von Operatoren,
  - Möglichkeit, Pointer zu verwenden (*unsafe code*).  
Standardargumente sind hinzugekommen, gibt es in Java nicht
- Neue Eigenschaften
  - Attribute: Benutzerdefinierte Metainformation,
  - Aufruf per Referenz (Übergangs- und Ausgangsparameter)
  - Wertetypen (Strukturen). gibt es in java nach wie vor nicht

# Bezeichner und Namenskonventionen

- Bezeichner
  - Kombination aus Zeichen, Ziffern, \_, @
  - Unicode-Zeichen: `class Téléphone { ... }`
  - Groß-/Kleinschreibung ist relevant.
- Namenskonventionen
  - Pascal-Notation für
    - Methoden: `CopyTo`
    - Properties: `ToString`
    - Typnamen: `TimeZone`
    - Öffentl. Felder: `Empty`
    - Interfaces: `ICloneable`
    - Enums: `Sat, Sun, Mon`
  - Camel-Notation für
    - Variablen: `myVar, i`
    - private Felder: `wordCount`

# Deklarationen

- Gültigkeitsbereiche
  - Namenräume
  - Klasse/Struktur, Interface
  - Enumerationen
  - Blöcke
- Deklarationsreihenfolge ist nicht relevant.
- Lokale Variablen müssen vor Verwendung deklariert werden.

## ■ Beispiel

```
namespace A {  
    public class C {  
        public class D {  
            public static E e;  
        }  
        public enum E { e, f };  
        public static int e;  
    }  
}
```

```
namespace B {  
    class C {  
        static void F() {  
            int e = A.C.e;  
            A.C.E f = A.C.D.e;  
        }  
    }  
}
```

# Anweisungen

- *if-, while-, do-while*-Anweisung: wie in C++
- *switch*-Anweisung
  - Muss mit `break` abgeschlossen werden.
  - *switch*-Ausdruck kann numerischer Typ, Enumeration oder String sein.
  - Mit `goto` kann zu anderem Label gesprungen werden.
- *foreach*-Anweisung
  - Iteration durch Collections, die *IEnumerable* implementieren.
  - Beispiel:

```
string[] names = {"Joe", "Bill", "James"};
foreach (string n in names)
    Console.WriteLine(n);
```

# Operatoren

- Ähnlich wie in C++ bzw. Java

Kategorie	Operatoren
Primär	(x), x.y, x?.y, a[x], a?[x], f(x), x++, x--, new, <b>typeof</b> , <b>nameof</b> , sizeof
Unär	+, -, ++x, --x, (T)x
Punktr.	*, /, %
Strichr.	+, -
Shift	<<, >>
Relational	<, >, <=, >=, <b>is</b> , <b>as</b>

?. null  
 conditional  
 Operator  
 as = instance  
 of ??  
 is = downcast  
 durchführen,  
 wenn nicht mgl  
 dann keine  
 exception  
 sonder  
 nullpointer ==  
 dynamic cast  
 in c++ ??  
 <<<  
 berücksichtigu  
 ng von  
 vorzeichen ja/  
 - - -

Kategorie	Operatoren
Gleichheit	==, !=
Bin. UND	&
Bin. XOR	^
Bin. ODER	
Log. UND	&&
Log. ODER	
Null-Coel.	??
Bed. Ausdr.	?:
Zuweisung, Lambda	=, *=, /=, %=, +=, -=, &=,  =, ^=, =>

# Präprozessor

- Unterstützte Präprozessor-Direktiven:
  - `#define`, `#undef`: Symbole können nur definiert werden, ihnen kann aber kein Wert zugewiesen werden.
  - `#if`, `#elif`, `#else`, `#endif`

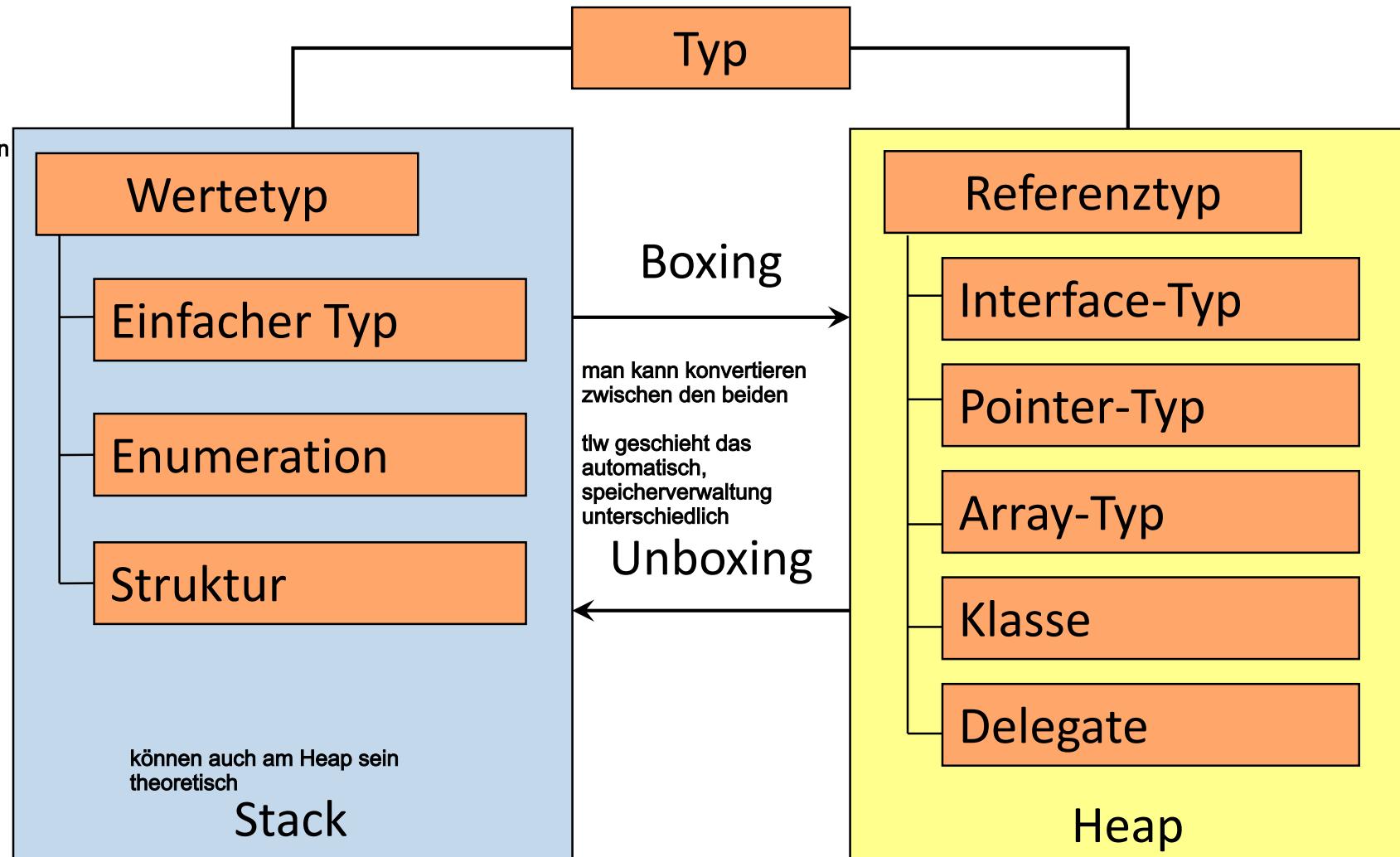
```
#define DEBUG
#if DEBUG
    // Code für Debug-Version
#else
    // Code für Release-Version
#endif
```
  - `#region`, `#endregion`: Kennzeichnung von Code-Blöcken für Editoren, z. B. automatisch generierter Code.
- Benutzerdefinierte Makros sind nicht möglich.

# Typen

Wertetypen in Java können nicht selbst definiert werden in C# schon

Enumeration in java ein Referenzdatentyp

Delegates sind spezielle Klassen



# Werte- und Referenztypen

- Wertetypen (*value types*)
    - Werden am *Stack* bzw. im umgebenden Objekt allokiert.
    - Defaultwert ist 0, '\0' bzw. false.
    - Bei Zuweisung wird Wert kopiert.
    - Im Gegensatz zu Java können Wertetypen auch selbst definiert werden (*enum* und *struct*).
  - Referenztypen (*reference types*)
    - Werden am *Heap* allokiert.
    - Defaultwert ist *null*.
    - Bei Zuweisung wird Referenz, aber nicht das referenzierte Objekt kopiert.  
Änderungen bringen also nichts, bei Referenz schon
- Wertedaten direkt gespeichert  
Referenz nur referenziert  
*struct color*
- gibt beim Kopieren Unterschiede  
Achtung by value Übergabe, bei Wertes wird kopiert übergeben
- 

# Einfache Typen

*Common Type System*

CLS-kompatibel

C#	CTS	Range	Java
sbyte	System.SByte	$-2^7 - 2^7-1$	byte
byte	System.Byte	$0 - 2^{8-1}$	-
short	System.Int16	$-2^{15} - 2^{15}-1$	short
ushort	System.UInt16	$0 - 2^{16}-1$	-
int	System.Int32	$-2^{31} - 2^{31}-1$	int
uint	System.UInt32	$0 - 2^{32}-1$	-
long	System.Int64	$-2^{63} - 2^{63}-1$	long
ulong	System.UInt64	$0 - 2^{64}-1$	-
float	System.Single	7 Stellen/4 Byte	float
double	System.Double	15 Stellen/8 Byte	double
decimal	System.Decimal	28 Stellen/16 Byte	-
bool	System.Boolean	true, false	bool
char	System.Char	Unicode-Zeichen	char

hier  
come  
grauhinter

# Enumerationen

- Deklaration und Verwendung wie in C++.
- Syntax: [modifiers] **enum identifier** [:base-type]  
    {enumerator-list};
- Enumerationskonstanten müssen qualifiziert werden.
- Beispiel:

```
enum Day:byte { Sun=1, Mon, Tue,  
                Wed, Thu, Fri, Sat };  
  
Day d = Day.Sat;  
Console.WriteLine("day = {0}", d);
```

// Output: day = Sat

# Strukturen

- Benutzer-definierbarer Typ, der sich wie einfacher Typ verhält:
  - Werte werden am Stack/im umgebenden Objekt angelegt,
  - Lebensdauer ist auf Lebensdauer des umgebenden Blocks beschränkt,
  - bei Zuweisung wird Wert kopiert (nicht Referenz).
- Strukturen können nicht erben oder vererben.
- Strukturen können Interfaces implementieren.
- Felder dürfen bei Deklaration nicht initialisiert werden.
- Default-Konstruktor darf nicht definiert werden.
- Vorteile
  - speichersparend,
  - müssen nicht vom GC verwaltet werden.

# Strukturen – Beispiel

- Definition

```
struct Color {  
    public byte r, g, b;  
    public Color(byte r, byte g, byte b) {  
        this.r = r; this.g = g; this.b = b;  
    }  
}
```

dann Interface impl.  
nicht Referenzdatentyp ableitbar  
dann davon auch kein Referenzdatentyp  
ableiten

- Verwendung

```
// Initialisierung mit Default-Werten (0)  
Color black = new Color();  
Color yellow = new Color(255, 255, 0);  
yellow.r = 200;
```

mit new nicht automatisch Speicherallokation

→ Speicherung im umgesenen Objekt wie bei Wertedatentypen

neu in Funktion, dann Referenzspeicherung

# Boxing und Unboxing

- *Boxing: Wertetyp → Referenztyp*
  - Boxing wird *implizit* durchgeführt, wenn ein Objekt benötigt wird, aber ein Wert vorhanden ist.
  - Beispiel:

```
Console.WriteLine("i={0} ", i); ist viel Speicher overhead
string s = 123.ToString();
```
- *Unboxing: Referenztyp → Wertetyp*
  - Unboxing muss explizit mit Cast durchgeführt werden.
  - Beispiel:

```
int i = 123;
object o = i;    // boxing
int j = (int)o; // unboxing
```

# Nullable Types

funktioniert auch mit Strukturen

- *Nullable Types* sind Wertetypen, die als Wert auch null annehmen können.
- Syntax: **T?** oder **Nullable<T>** (T beliebiger Wertetyp)

```
struct Nullable<T> {  
    public Nullable(T value);  
    public bool HasValue { get; } Ja/Nein Wert vorhanden?  
    public T Value { get; }  
}
```

- Beispiel:

```
int? i1 = 10;  
int? i2 = null; // implicit conversion  
int j1 = i1.Value; // j1 == 10  
int j2 = (int)i1; // j2 == 10  
int j3 = i2 ?? 0; // j3 == 0 Wert ist null wenn null kommt
```

# Klassen

- Inhalt einer Klasse:
  - Konstanten
  - Felder
  - Konstruktoren/Destruktor
  - Methoden
  - Operatoren
  - Properties
  - Indexer
  - Events
  - statischer Konstruktor
  - (innere) Typen

implizit auch static

```
class Rational {  
    const double Eps = 0.001;  
    int a, b;  
    public Rational(  
        int a, int b) { ... }  
    public void Add(  
        Rational c) { ... }  
    public static Rational  
        operator+(Rational r1,  
                    Rational r2) {...}  
    public int Denom {  
        get { return b; }  
        set { b = value; }  
    }  
}
```

# Sichtbarkeitsattribute

## ■ Sichtbarkeit

<i>public</i>	überall
<i>protected</i>	deklarierende/abgeleitete Klasse(n) <i>gilt in Java auch für selles Paket</i>
<i>internal</i>	selbes Assembly
<i>protected internal</i>	= protected ODER internal <i>abgeleitet und innerhalb des selben Assemblies</i>
<i>private protected</i>	= protected UND internal (C# 7.2) <i>nur im selben Assembly</i>
<i>private</i>	deklarierende Klasse

## ■ Default

Methoden/Felder	<i>private</i> <i>) in Java package visibility</i>
äußere Typdefinition	<i>internal</i>
innere Typdefinition	<i>private</i>
Interfacemethoden	<i>public</i>
Enumerationskonstanten	<i>public</i>

# Konstruktoren

- Konstruktoren dürfen *überladen* werden.
- Konstruktor der *Basisklasse* wird im Kopf mit `base` aufgerufen.
- *Anderer Konstruktor* kann im Kopf mit `this` aufgerufen werden.
- Generierter *Default-Konstruktor* initialisiert alle Felder mit Standardwerten.

```
public class Ellipse {  
    int a,b;  
    public Ellipse(int a, int b) {  
        this.a = a; this.b = b;  
    }  
    public Ellipse(int r) :  
        this(r, r) {}  
}  
  
public class ColoredEllipse :  
    Ellipse {  
    Color color = Color.Black;  
    public ColoredEllipse(  
        int a, int b, Color c) :  
        base(a, b) { color = c; }  
}
```

# Destruktor

- Destruktor/Finalizer wird aufgerufen, unmittelbar bevor GC ein Objekt freigibt.
- Dient zur Ressourcenfreigabe (Schließen von Files, ...)
- Es ist nicht definiert, wann Speicherbereinigung durchgeführt und damit der Destruktor/Finalizer aufgerufen wird.
- Destruktor der Basisklasse wird automatisch aufgerufen (im Gegensatz zu Java).

```
public class A {  
    ~A() { // Sichtbarkeitsattribut darf  
           // nicht angegeben werden.  
        // Ressourcenfreigabe  
    }  
}
```

nichts anderes als  
ein Finalizer  
wenn Garbage Collector  
Objekt freigeben will  
wird Finalizer aufgerufen

Finalizer der Basisklasse muss  
explizit aufgerufen werden

# Verwendung von IDisposable

- Wenn explizite Ressourcenfreigabe möglich sein soll, kann Klasse *IDisposable* implementieren.
- Verwender kann *Dispose()* explizit aufrufen.

```
public class A : IDisposable {  
    ~A() {  
        Dispose();  
    }  
    public void Dispose() {  
        // Ressourcenfreigabe  
        GC.SuppressFinalize(this);  
    } um doppeltes Löschen zu vermeiden  
}
```

achte auf die Lebensdauer der Methode

solche Klassen sollten nicht zu lange

leben → Disposable muss freigegeben werden

```
A a = null;  
try {  
    a = new A();  
    ...  
} finally {  
    if (a!=null) a.Dispose();  
} hier wird auf jeden Fall eine Ressourcenfreigabe gemacht
```

oder (in C#)

```
using (A a = new A()) {  
    ...  
} // Aufruf von a.Dispose()
```

egal wie Block verlassen wird, Dispose wird aufgerufen

# Felder und Konstanten

- Objekt-Felder
  - Beispiel: `int size = 0;`
- Statische Felder
  - Beispiel: `static Color Red = new Color(255,0,0);`
- Konstanten
  - Wert muss von Compiler berechnet werden können.
  - Beispiel: `const int arrLen = byte.MaxValue/2 + 1;`
- Schreibgeschützte Felder (`readonly`)
  - Darf nur in Deklaration oder Konstruktor initialisiert werden.
  - Beispiel: `readonly Pen defaultPen;`

*im Konstruktor  
einmal initialisieren  
dann nicht mehr}*

```
public Drawing(Color c) {  
    defaultPen = new Pen(c);  
}
```

# Methoden

- Objektmethoden
  - Aufruf: *object.Method()*
  - Überladen wie in Java möglich.
- Klassenmethoden (statische Methoden)
  - Aufruf: *Class.Method()*

```
class Date {  
    enum Day { Sun, Mon, Thu, ... }  
    static Day FirstDayInWeek() {  
        return Day.Mon;  
    }  
    public static void Main() {  
        Day d = Date.FirstDayInWeek();  
        Date date = new Date();  
        date.FirstDayInWeek(); // Syntaxfehler!  
    }  
}
```

# Arten von Parametern

- Eingangsparameter (*„call by value“*)
  - Definition: `int Twice(int m) { return 2*m; }`
  - Aufruf: `m=5; n = Twice(m); // m==5, n==10`
- Übergangsparameter (*ref-Parameter*)
  - ref-Parameter muss initialisiert sein.
  - Definition: `void Twice(ref int n) { n *= 2; }`
  - Aufruf: `n=5; Twice(ref n); // n==10`
- Ausgangsparameter (*out-Parameter*)
  - out-Parameter muss **nicht** initialisiert sein.
  - Definition: `void Twice(int m, out int n) { n = 2*m; }`
  - Aufruf: `m=5; Twice(m, out n); // m==5, n==10`

# Variable Anzahl von Parametern

- Definition einer Methode, die eine variable Anzahl von Parametern verarbeiten kann:

```
double Sum(params double[] values) {  
    double sum = 0;  
    foreach (double val in values) sum += val;  
    return sum;  
}
```

- Aufruf
  - Variable Anzahl von Parametern ↗ *automatische Konvertierung als Array*

```
double sum = Sum(1,2,3,4); // sum==10
```

- Parameterübergabe in Form eines Arrays

```
double[] arr = {1,2,3,4};  
double sum = Sum(arr); // sum==10
```

# Properties

- Zusammenfassung einer Getter- und einer Setter-Methode zu einer Einheit.
- Definition einer Property:

```
class Circle {  
    private double rad = 0.0;  
    public double Radius { // Property Radius  
        get { return rad; } // Getter-Methode  
        set { rad = value; } // Setter-Methode  
    }  
}
```

- Verwendung einer Property:

```
Circle c = new Circle();  
c.Radius = 5.0;          // Ausführung der Setter-Methode  
double r = c.Radius;    // Ausführung der Getter-Methode
```

- Uniform Access Principle
  - Zugriff auf Datenkomponente oder auf eine Property unterscheiden sich nicht.

# Indexers

- Zugriff auf ein Element einer Collection mit dem []-Operator.
- Definition eines Indexers:

```
class BirthdayList {  
    public DateTime this[string name] {  
        get { return GetBirthDay(name); }  
        set { SetBirthDay(name, value); }  
    }  
}
```

- Verwendung eines Indexers:

```
BirthdayList bList = new BirthdayList();  
bList["Huber"] = new DateTime(1970, 12, 24);  
Console.WriteLine("Geb. Tag: {0}", bList["Huber"]);
```

# Überladen von Operatoren

- Überladen von arithmetischen, Vergleichs- und Bitoperatoren

```
class Rational {  
    public static Rational operator+(Rational r1, Rational r2){  
        return new Rational(r1.a*r2.b + r2.a*r1.b, r1.b*r2.b);  
    }  
}
```

- Überladen von Konversionsoperatoren

```
class Rational {  
    public static implicit operator double(Rational r) {  
        return (double)r.a/r.b;  
    }  
    public static explicit operator long(Rational r) {  
        return r.a/r.b;  
    }  
}
```

- Verwendung überladener Operatoren

```
Rational r = new Rational(1,2) + new Rational(1,4);  
double d = r;      // implizite Konvertierung  
long l = (long)r; // explizite Konvertierung
```

# Vererbung



- Nur *Einfachvererbung* ist möglich.
- Wird keine Basisklasse angegeben, wird automatisch von *Object* abgeleitet.
- Öffentliche und geschützte Methoden werden vererbt:

```
class A {  
    private int b;  
    public A(int b) {  
        this.b = b;  
    }  
    public void F() { ... }  
}  
  
class B : A {  
    public B(int a) : base(a) {}  
    public void G() { ... }  
}
```

```
public static void Main() {  
    A a = new A();  
    B b = new B();  
    a.F();  
    b.F();  
    b.G();  
}
```

F() wird von A geerbt

# Überschreiben von Methoden: override

- Dynamisch zu bindende Methoden müssen als **virtual** deklariert werden.
- Sollen virtuelle Methoden überschrieben werden, müssen sie als **override** oder **new** deklariert werden.

```
class A {  
    public virtual void F() {  
        ...  
    }  
}  
  
class B : A {  
    public override void F() {  
        ...  
    }  
}
```

nur **virtual** Methoden können  
überschrieben werden.

```
public static void Main() {  
    A a1 = new A(); → statischer Typ von A  
    A a2 = new B();  
    B b = new B();  
    a1.F(); // A.F(); → dynamischer Typ  
    a2.F(); // B.F();  
    b.F(); // B.F();  
}
```



dynamische Bindung

statischer Typ von Compiler  
dynamischer Typ von der Laufzeit <sup>26</sup>

# Überschreiben von Methoden: new



- Eine mit **new** deklarierte Methode ist unabhängig von der gleichnamigen Methode (mit gleicher Signatur) der Basisklasse.
- Referenzen mit dem statischen Typ der Basisklasse haben keinen Zugang mehr zur mit **new** deklarierten Methode.

```
class A {  
    public virtual void F() {}  
    public      void G() {}  
    public virtual void H() {}  
}  
class B : A {  
    public new      void F() {}  
    public new virtual void G() {}  
    public override void H() {}  
}  
class C : B {  
    public override void G() {}  
    public new virtual void H() {}  
}
```

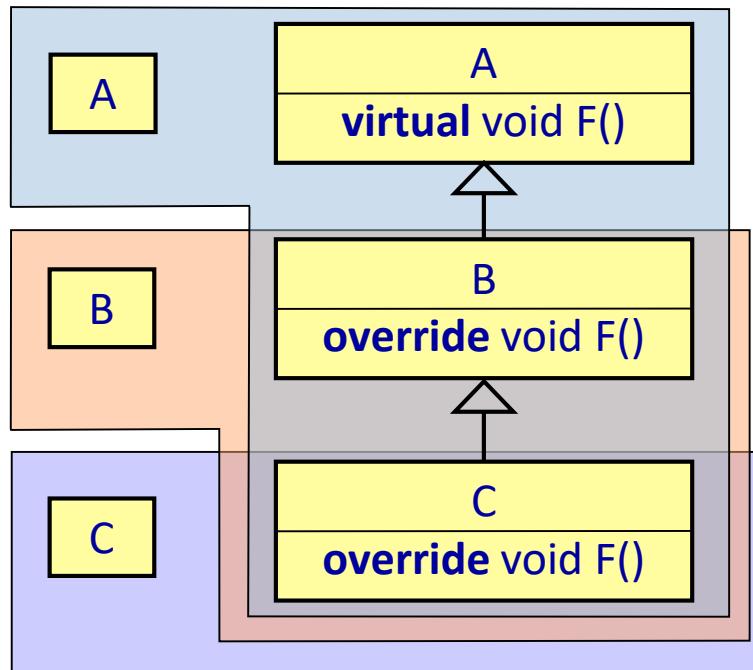
```
static void Main() {  
    A a1 = new B(); statischer Typ: A  
    a1.F(); // A.F()  
    a1.G(); // A.G(); → um Zugriff zu haben müssen  
    a1.H(); // B.H(); statischer Typ: B  
    A a2 = new C();  
    a2.F(); // A.F()  
    a2.G(); // A.G();  
    a2.H(); // B.H(); haben  
    B b = new C();  
    b.F(); // B.F()  
    b.G(); // C.G();  
    b.H(); // B.H();  
}
```

# Gegenüberstellung von *new* und *override*

```
X x = new Y();  
x.F();
```

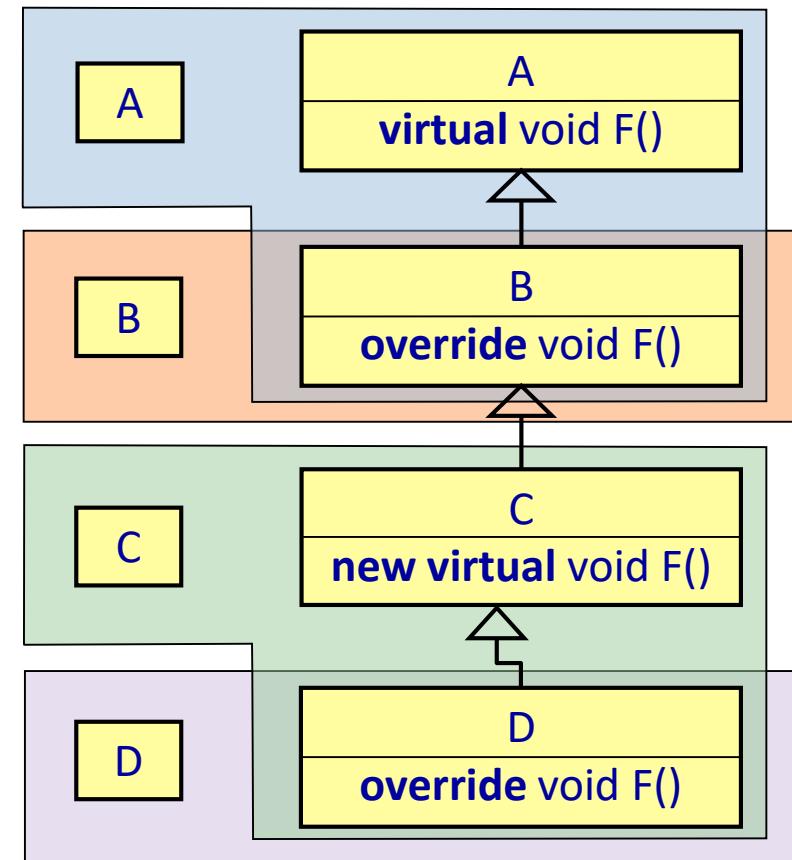
## *override*

stat. Typ (X)    dyn. Typ (Y)



## *new*

stat. Typ (X)    dyn. Typ (Y)



# Anwendung von new

- Austausch von Basisklassen soll sich nicht auf Funktionalität von abgeleiteten Klassen auswirken (*Fragile Base Class Problem*).  
wur überschreiben unab:  
si htigt Methode

alte Version

```
class A {  
    public void F() {  
        ...  
    }  
}
```

neue Version

```
class A {  
    public void G() ←  
    public void F() {  
        G(); ——————  
    }  
}
```

```
class B : A {  
    public new void G() {}  
}
```

— Java →  
— C# →

Im Java kann  
alles überschrieben  
werden

# Abstrakte Klassen und Interfaces

```
abstract class Shape {  
    public abstract void Draw();  
    public void Move() {...}  
}  
  
class Line : Shape {  
    public override void Draw() {  
        ...  
    }  
}
```

- Klassen mit abstrakten Methoden müssen abstrakt sein.
- Abstrakte Methoden sind implizit *virtual*.
- Abstrakte Klassen können *nicht instanziert* werden.

```
public interface I {  
    void F(int i);  
    object P { get; set; }  
}  
  
class A : I, ICloneable {  
    public void F(int i) {...}  
    public object P {  
        get {...} set {...} }  
    public object Clone() {...}  
}
```

- Methoden eines Interfaces sind *public abstract*.
- Klassen und Strukturen dürfen mehrere *Interfaces* implementieren.

# Explizite Implementierung von Interface-Methoden

- Interface-Methoden können durch Qualifikation mit dem Interface-Namen explizit implementiert werden.
- Methode darf weder *public* noch *private* deklariert werden.
- Statischer Typ einer Referenz bestimmt, welche Methode sichtbar ist.

```
interface I {  
    void F();  
    void G();  
}
```

```
interface J {  
    void F();  
    void G();  
}
```

```
class A : I, J {  
    void I.F() { ... } // expl. impl.  
    void J.F() { ... } // expl. impl.  
    public void F() { ... }  
    public void G() { ... }  
}
```

```
static void Main() {  
    A a = new A();  
    I i = a;  
    J j = a;  
  
    a.F(); // A.F();  
    i.F(); // I.F();  
    j.F(); // J.F();  
  
    a.G(); // A.G();  
    i.G(); // A.G();  
    j.G(); // A.G();  
}
```

# Ausnahmen (Exceptions)

- Ausnahmen werden nicht im Methodenkopf deklariert.
- Ausnahmen müssen nicht behandelt werden.

```
StreamReader sr = null;
try {
    sr = new StreamReader(
        new FileStream("data.txt", FileMode.Open));
    sr.ReadLine();
}
catch(FileNotFoundException e) { ... }
catch(IOException e) // Ausnahmenfilter
    when (e.InnerException is ArgumentException) { ... }
catch(IOException e) { ... }
catch { // Behandlung aller anderen Exceptions }
finally { // wird immer durchlaufen
    if (sr != null) sr.Close();
}
```

# Arrays

- Eindimensionale Arrays
  - Deklaration: `int[] arr;`
  - Initialisierung: `arr = new int[] {1,2,3};` oder  
`arr = {1,2,3};`
- Mehrdimensionale Arrays
  - „Jagged“ Arrays

```
int[][] m = new int[2][];
m[0] = new int[]{1,2};
m[1] = new int[]{4,5,6};
m[1][2] = 9;
```
  - Rechteckige Arrays

```
int[,] m = {{1,2,3},{4,5,6}};
m[1,2] = 9;
```

