

C# Inhalt von Klassen und Strukturen

```
class C
{ ... Variablen, Konstanten           ... // für Objektorientierte Programmierung
  ... Methoden ...
  ... Konstruktoren, Destruktoren ...

  ... Properties ...                 // für Komponentenorientierte Progr.
  ... Events ...

  ... Indexers ...                   // Annehmlichkeit
  ... überladene Operatoren ...

  ... geschachtelte Typen (Klassen, Interfaces, Structs, Enums, Delegates) ...
}
```

C# Klassen

```
class Stack
{ int[] values;
  int top=0;
  public Stack(int size) { ... }
  public void Push (int x) {...}
  public int Pop() {...}
}
```

- Objekte werden am Heap angelegt (sind Referenztypen)
- Objekte müssen mit *new* erzeugt werden
Stack s = new Stack(100);
- Können erben, vererben und Interfaces implementieren

C# Structs

```
struct Point {  
    int x, y;  
    public Point(int x, int y) { this.x = x; this.y = y; }  
    public MoveTo (int x, int y) {...}  
}
```

- Objekte werden am Stack statt am Heap angelegt (sind Werttypen).
 - + speichersparend, effizient, belasten GC nicht
 - nicht für dynamische Datenstrukturen
- Können aber müssen nicht mit *new* erzeugt werden.

```
Point p;           // Variablen von p sind so allerdings nicht initialisiert  
Point q = new Point();
```

- Variablen dürfen bei der Deklaration nicht initialisiert werden.

```
struct Point {  
    int x = 0;      // Compilefehler  
}
```

- Deklarierte Konstruktoren müssen mindestens 1 Parameter haben.
- Können weder erben noch vererben.

C# Sichtbarkeitsattribute

public überall bekannt, wo deklarierender Namespace bekannt ist
Standard für:

- Interface-Members
- Enum-Members

Typen auf äußerster Ebene (Klassen, Structs, Interfaces, Enums, Delegates) haben per default Sichtbarkeit *internal* (ähnlich *public*, siehe später)

private Nur in deklarierender Klasse/Struct bekannt
Standard für:

- Klassen-Members (Variablen, Methoden, ..., geschachtelte Typen)
- Struct-Members (Variablen, Methoden, ..., geschachtelte Typen)

Beispiel

```
public class Stack {  
    private int[] val;           // wäre Default  
    private int top;           // wäre Default  
    public Stack() {...}  
    public void Push(int x) {...}  
    public int Pop() {...}  
}
```

C# Zugriff auf private Members

```
class B
{ private int x;
  ...
}

class C
{ private int x;

  public void f (C c)
  { x = ...; // Methode darf auf private Members von this zugreifen.

    c.x = ...; // Methode der Klasse C darf auf private Members
               // eines anderen C-Objekts zugreifen.

    B b = ...;
    b.x = ...; // Fehler! Methode der Klasse C darf nicht auf private Members
               // einer anderen Klasse zugreifen.
  }
}
```

C# Variablen und Konstanten

```
class Test
```

```
{
```

```
int value = 0;
```

Variable

- Initialisierung in Deklaration optional
- Initialisierung darf nicht auf Variablen und Methoden zugreifen
- Struct-Variablen dürfen nicht initialisiert werden

```
const long size = ((long)int.MaxValue + 1) / 4;
```

Konstante

- Muss Initialisierungswert haben
- Wert muss zur Compilezeit berechenbar sein

```
readonly DateTime date;
```

ReadOnly-Variable

- Muss in Deklaration oder Konstruktor initialisiert werden
- Wert muss nicht zur Compilezeit berechenbar sein
- Wert darf später nicht mehr geändert werden
- Wert belegt Speicherplatz (wie Variable)

```
}
```

Zugriff innerhalb Test

```
... value ... size ... date ...
```

Zugriff aus anderen Klassen

```
Test c = new Test();
... c.value ... c.size ... c.date ...
```

C# Statische Variablen und Konstanten

Daten der Klasse und nicht des Objekts

```
class Rectangle
{
    static Color defaultColor;    // existiert einmal pro Klasse
    static readonly int scale;    // -- " --
    int x, y, width,height;      // in jedem Objekt gespeichert
    ...
}
```

Zugriff innerhalb Klasse

... defaultColor ... scale ...

Zugriff aus anderen Klassen

... Rectangle.defaultColor ... Rectangle.scale ...

Konstanten dürfen nicht static deklariert werden

C# Methoden

Beispiele

```
class C
{ int sum = 0, n = 0;

    public void Add (int x) {           // Prozedur
        sum = sum + x; n++;
    }

    public float Mw() {                 // Funktion (muss Wert mit return zurückgeben)
        return (float) sum / n;
    }
}
```

Aufruf aus Klasse C

```
Add(3);
float x = Mw();
```

Aufruf aus anderen Klassen

```
C c = new C();
c.Add(3);
float x = c.Mw();
```

C# Statische Methoden

Operationen auf Klassendaten (statische Daten)

```
class Rectangle
{
    static Color defaultColor;

    public static void ResetColor() {
        defaultColor = Color.white;
    }
}
```

Aufruf aus Rectangle

```
ResetColor();
```

Aufruf aus anderen Klassen

```
Rectangle.ResetColor();
```

C# Arten von Parametern

Value-Parameter(Eingangsparmeter)

```
void Inc(int x) {x = x + 1;}
void f() {
    int val = 3;
    Inc(val); // val bleibt 3
}
```

- "call by value"
- Formaler Parameter (hier x) ist Kopie des aktuellen Parameters (hier val)
- akt.Parameter = beliebiger Ausdruck

ref-Parameter(Übergangsparmeter)

```
void Inc(ref int x) { x = x + 1; }
void f() {
    int val = 3;
    Inc(ref val); // val wird 4
}
```

- "call by reference"
- Formaler Parameter ist anderer Name für den aktuellen Parameter (Alias) (Adresse d. akt. Parameters wird überg.)
- Aktueller Parameter muss Variable sein

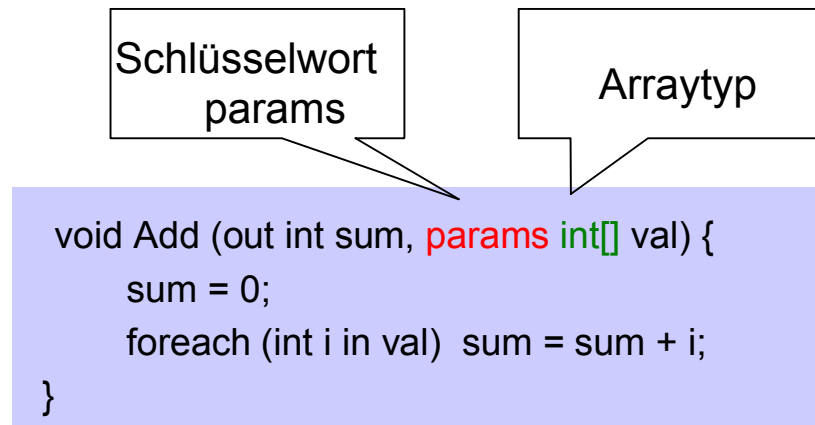
out-Parameter(Ausgangsparmeter)

```
void Read (out int first, out int next ) {
    first = Console.Read(); next = Console.Read();
}
void f() {
    int first, next;
    Read(out first, out next);
}
```

- Wie ref-Parameter, aber wird zur **Rückgabe** von Werten verwendet.
- Darf in der Methode nicht verwendet werden, bevor ihm ein Wert zugewiesen wurde.

C# Variable Anzahl von Parametern

Letzten Parameter dürfen beliebig viele Werte eines bestimmten Typs sein.



params geht nicht für *ref* und *out*

Aufruf

```
Add(out sum, 3, 5, 2, 9); // sum wird 19
```

Beispiel

```
void Console.WriteLine (string format, params object[] arg) {...}
```

C# Überladen von Methoden

Methoden einer Klasse dürfen gleich heißen, wenn sie

- unterschiedliche Anzahl von Parametern haben oder
- unterschiedliche Parametertypen haben oder
- unterschiedliche Parameterarten (*value*, *ref/out*) haben

Beispiele

```
void F (int x) {...}  
void F (char x) {...}  
void F (int x, long y) {...}  
void F (long x, int y) {...}  
void F (ref int x) {...}
```

Aufrufe

```
int i; long n; short s;  
F(i);           // F(int x)  
F('a');        // F(char x)  
F(i, n);       // F(int x, long y)  
F(n, s);       // F(long x, int y);  
F(i, s);       // => Compilefehler  
F(i, i);       // => Compilefehler
```

C# Überladen von Methoden

nicht erlaubt:

Überladene Methoden dürfen sich nicht nur im Funktionstyp unterscheiden

```
int F() {...}  
string F() {...}
```

Folgende Überladung ist ebenfalls nicht erlaubt

```
void P(int[] a) {...}  
void P(params int[] a) {...}
```

```
int[] a = {1, 2, 3};  
P(a);           // sollte eigentlich P(int[] a) aufrufen  
P(1, 2, 3);    // sollte eigentlich P(params int[] a) aufrufen  
➔ Compilefehler
```

C# Konstruktoren in Klassen

Beispiel

```
class Rectangle
{
    int x, y, width, height;
    public Rectangle (int x, int y, int w, int h){this.x = x; this.y = y; width = w; height = h; }
    public Rectangle (int w, int h) : this(0, 0, w, h) {}
    public Rectangle () : this(0, 0, 0, 0) {}
    ...
}
```

```
Rectangle r1 = new Rectangle();
Rectangle r2 = new Rectangle(2, 5);
Rectangle r3 = new Rectangle(2, 2, 10, 5);
```

- Konstruktoren dürfen überladen werden.
- Ein Konstruktor kann einen anderen mittels **this** aufrufen (im Kopf des Konstruktors, wie in C++, nicht im Rumpf wie in Java).
- Zuerst werden die bei der Variablendeklaration angegebenen Initialisierungen ausgeführt dann erst wird der Konstruktor aufgerufen.

C# Default-Konstruktor

hat eine Klasse keinen Konstruktor, wird ein parameterloser Default-Konstruktor angelegt:

```
class C { int x; }  
C c = new C(); // ok
```

Default-Konstruktor initialisiert alle Felder wie folgt:

numerisch	0
enum	0
bool	false
char	'\0'
reference	null

Hat eine Klasse einen Konstruktor, wird kein Default-Konstruktor angelegt:

```
class C  
{ int x;  
  public C(int y) { x = y; }  
}  
  
C c1 = new C(); // Compilefehler  
C c2 = new C(3); // ok
```

C# Konstruktoren in Structs

Beispiel

```
struct Complex
{
    double re, im;
    public Complex(double re, double im) { this.re = re; this.im = im; }
    public Complex(double re) : this(re, 0) {}
    ...
}
```

```
Complex c0; // c0.re und c0.im nicht initialisiert
Complex c1 = new Complex(); // c1.re wird 0, c1.im wird 0
Complex c2 = new Complex(5); // c2.re wird 5, c2.im wird 0
Complex c3 = new Complex(10, 3); // c3.re wird 10, c3.im wird 3
```

- Jeder Struct hat einen parameterlosen Default-Konstruktor, der alle Variablen initialisiert (auch wenn es andere Konstruktoren gibt).
- Structs dürfen daher keinen expliziten parameterlosen Konstruktor haben.
- Ein struct-Konstruktor muß alle Variablen des Struct initialisieren.

C# Statische Konstruktoren

Sowohl bei Klassen als auch bei Structs möglich

```
class Rectangle
{
    ...
    static Rectangle() {
        Console.WriteLine("Rectangle initialisiert");
    }
}
```

```
struct Point
{
    ...
    static Point() {
        Console.WriteLine("Point initialisiert");
    }
}
```

- Es darf nur einen statischen Konstruktor pro Klasse/Struct geben.
- müssen parameterlos sein und haben kein *public* oder *private*
- Wird genau einmal ausgeführt, bevor das erste Objekt der Klasse erzeugt oder das erste Mal auf eine statische Variable der Klasse zugegriffen wird.
- wird verwendet für Initialisierungsarbeiten, z.B. Initialisierung statischer Variablen

C# Destruktoren

```
class Test
{
    ~Test () {
        ... Abschlußarbeiten ...
    }
}
```

- Hat ein Objekt einen Destruktor, wird dieser aufgerufen, bevor der Garbage Collector das Objekt freigibt.
- Kann verwendet werden, um z.B. offene Dateien zu schließen.
- Destruktor der Basisklasse wird anschließend automatisch aufgerufen.
- Kein *public* oder *private* .
- Structs dürfen keinen Destruktor haben

C# Properties

Syntaktische Kurzform für get/set-Methoden

```

class Data
{
    FileStream s;

    public string FileName
    {
        set
        {
            s = new FileStream(value, FileMode.Create);
        }
        get
        {
            return s.Name;
        }
    }
}

```

Typ des Properties

Name des Properties

"Eingangsparameter" von set

Wird wie eine Variable benutzt

```

Data d = new Data();

d.FileName = "myFile.txt"; // ruft set("myFile.txt") auf
string s = d.FileName;    // ruft get() auf

```

C# Properties

Alle Zuweisungsoperatoren funktionieren auch mit Properties

```
class C
{ private static int size;

    public static int Size
    { get { return size; }
      set { size = value; }
    }
}

Size = x;
Size += 2; // Size = Size + 2;
```

C# Properties

get oder set kann auch fehlen

```
class Account
{ long balance;

  public long Balance ← read-only
  { get { return balance; }
  }
}

Account acc = new Account();
x = acc.Balance;    // ok
acc.Balance= ...; // verboten
```

Nutzen von Properties

- read-only und write-only-Daten möglich.
- Validierung beim Zugriff möglich.
- Benutzersicht und Implementierung der Daten können verschieden sein.
- Ersatz für Variablen in Interfaces.

C# Indexer

Programmierbarer Operator zum Indizieren einer Folge (Collection)

```

class File {
    FileStream s;
    public int this [int index] {
        get { s.Seek(index, SeekOrigin.Begin);
              return s.ReadByte();
            }
        set { s.Seek(index, SeekOrigin.Begin);
              s.WriteByte((byte)value);
            }
        }
    }
}

```

Typ des indizierten Ausdrucks (blue box pointing to `int`)
 Name (immerthis) (red box pointing to `this`)
 Typ und Name des Indexwerts (green box pointing to `[int index]`)

Benutzung

```

File f = ...;
int x = f[10];      // ruft f.get(10)
f[10] = 'A';       // ruft f.set(10, 'A')

```

- get oder set-Operation kann fehlen (write-only bzw. read-only)
- Überladene Indexer mit unterschiedlichem Indextyp möglich
- .NET-Bibliothek enthält Indexer für *string(s[i])*, *ArrayList(a[i])*, usw.

C# Indexer

```

class MonthlySales
{
    int[] apples = new int[12];
    int[] bananas = new int[12];
    ...
    public int this[int i] {           // set-Methode fehlt → read-only
        get { return apples[i-1] + bananas[i-1]; }
    }

    public int this[string month] {    // überladener read-only-Indexer
        get {
            switch (month) {
                case "Jan": return apples[0] + bananas[0];
                case "Feb": return apples[1] + bananas[1];
                ...
            }
        }
    }
}

```

```
MonthlySales sales = new MonthlySales();
```

```
...
Console.WriteLine(sales[1] + sales["Feb"]);
```

C# Überladene Operatoren

Statische Methode, die wie ein Operator verwendet werden kann

```
struct Fraction
{
    int x, y;
    public Fraction (int x, int y) {this.x = x; this.y = y; }

    public static Fraction operator + (Fraction a, Fraction b) {
        return new Fraction(a.x * b.y + b.x * a.y, a.y * b.y);
    }
}
```

Benutzung

```
Fraction a = new Fraction(1, 2);
Fraction b = new Fraction(3, 4);
Fraction c = a + b; // c.x wird 10, c.y wird 8
```

- Überladbare Operatoren:
 - arithmetische: +, - (unär und binär), *, /, %, ++, --
 - Vergleichsoperatoren: ==, !=, <, >, <=, >=
 - Bitoperatoren: &, |, ^
 - Sonstige: !, ~, x>, <<, true, false
- Müssen immer ein Funktionsergebnis liefern
- Wenn == (<, <=, true) überladen wird, muss auch != (>=, >, false) überladen werden.

C# Geschachtelte Typen

```
class A
{ int x;
  B b = new B(this);
  public void f() { b.f(); }

  public class B {
    A a;
    public B(A a) { this.a = a; }
    public void f() { a.x = ...; ... a.f(); }
  }
}

class C {
  A a = new A();
  A.B b = new A.B(a);
}
```

Für Hilfsklassen, die versteckt bleiben sollten

- Innere Klasse sieht alle Members der äußeren Klasse (auch private).
- Äußere Klasse sieht nur public-Members der inneren Klasse.
- Andere Klassen sehen innere Klasse nur, wenn sie public ist.

Geschachtelte Typen können auch Structs, Enums, Interfaces und Delegates sein.