Die Sprache C# 2. Teil

School of Engineering

- Ausdrücke
- Anweisungen
- Deklarationsräume
- Objektorientierte Konzepte
- Klassen und Strukturen

Ausdrücke

Operatoren und Vorrangregeln

School of Engineering

Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
Unary	+ - ~ ! ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is as
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	c?x:y
Assignment	= += -= *= /= %= <<= >>= &= ^= |=

Operatoren der gleichen Zeile werden von links nach rechts ausgewertet.

Die unären Operatoren +, -, ~, ! und Casts werden von rechts nach links ausgewertet.

Arithmetische Ausdrücke

Operandentypen

- numerisch oder *char*
- bei ++ und -- numerisch oder *enum*-Typ (funktioniert auch bei *float* und *double*!)

Ergebnistyp

- Kleinster numerischer Typ, der beide Operandentypen einschliesst, aber zumindest *int*.

Ausnahmen

uint • Vorzeichen behafteter Typ => long

bsp: uint • (sbyte | short | int) => long

Vergleichsausdrücke

School of Engineering

Operandentypen

- bei <, >, <=, >=: numerisch, *char*, *enum*
- bei ==, !=: numerisch, *char*, *enum*, *bool*, Referenzen
- bei x is T: *x*: Ausdruck mit beliebigem Typ, *T*: Referenztyp

  z.B.: obj is Rectangle
  objOfValueType is IComparable
  3 is object
  arr is int[]

Ergebnistyp

bool

Boolesche Ausdrücke (&&, ||, !)

School of Engineering

Operandentypen

bool

Ergebnistyp

bool

Kurzschlussauswertung (bedingte Auswertung)

a && b => if (a) b else false

a || b => if (a) true else b

Nützlich bei z.B.

```
if (p != null && p.val > 0) ...
if (x == 0 || y / x > 2) ...
```

Bit-Ausdrücke (&, |, ^, ~)

School of Engineering

Operandentypen

- bei & | ^ : numerisch, *char*, *enum*, *bool*
- bei ~ : numerisch, *char*, *enum*
- bei unterschiedlich grossen Operandentypen, werden beide Operanden in den grösseren Typ konvertiert

Ergebnistyp

- grösserer der beiden Operandentypen
- bei numerischen Typen und *char* zumindest *int*

Shift-Ausdrücke

zh aw School of Engineering

Operandentypen für x << y und x >> y

- *x*: ganzzahlig oder *char*
- *y*: *int*

Ergebnistyp

Typ von *x*, aber zumindest *int*

Bemerkung

>> führt bei vorzeichenlosen Typen ein **logisches Shift** durch,
sonst ein **arithmetisches Shift** (schiebt Vorzeichen-Bit nach)

Überlaufprüfungen

Normalerweise wird Überlauf nicht erkannt

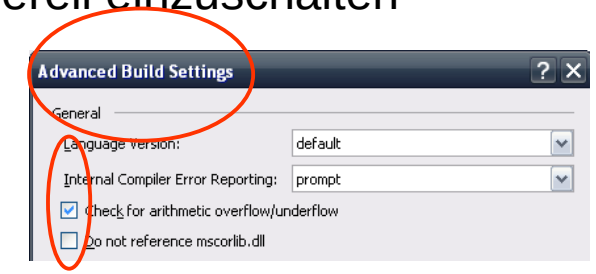
```
int x = 1000000;
x = x * x;   // -727379968, kein Fehler
```

Überlaufprüfung

```
x = checked(x * x);  // liefert System.OverflowException
checked {
   ...
   x = x * x;        // liefert System.OverflowException
}
```

Es gibt auch Compiler-Option, um Überlaufprüfung generell einzuschalten

```
csc /checked Test.cs
```



Anweisungen

Einfache Anweisungen

School of Engineering

Leeranweisung

```
;                                   // ; ist Anweisungs-Terminator, nicht Separator
```

Zuweisung

```
x = 3 * y + 1;
```

Methodenaufruf

```
string s = "a,b,c";
string[] parts = s.Split(',');   // Aufruf einer Objektmethode (nicht static)

s = String.Join(" + ", parts);   // Aufruf einer Klassenmethode (static)
```

if-Anweisung

School of Engineering

```
if ('0' <= ch && ch <= '9')
   val = ch - '0';
else if ('A' <= ch && ch <= 'Z')
   val = 10 + ch - 'A';
else {
   val = 0;
   Console.WriteLine("invalid character " + ch);
}
```

switch-Anweisung

zh aw School of Engineering

```
switch (country) {
   case "Germany": case "Austria": case "Switzerland":
        language = "German";
        break;
   case "England": case "USA":
        language = "English";
        break;
   case null:
        Console.WriteLine("no country specified");
        break;
   default:
        Console.WriteLine("don't know language of " + country);
        break;
}
```

Typ des Switch-Ausdrucks

- ganzzahlig, *char*, *enum* oder *string* (*null* auch als Case-Marke erlaubt).

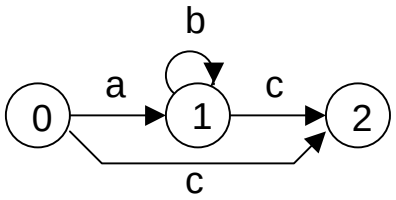
Fall-Through nur bei leerem Case erlaubt!

- andere Case-Anweisungsfolge muss mit break (oder return, goto, throw) enden.
- Wenn keine Marke passt ➔ default.
- Wenn default fehlt ➔ Fortsetzung nach switch-Anweisung

C#

switch mit Sprüngen

Z.B. zur Implementierung endlicher Automaten




```
int state = 0;
int ch = Console.Read();
switch (state) {
   case 0: if (ch == 'a') { ch = Console.Read(); goto case 1; }
           else if (ch == 'c') goto case 2;
           else goto default;
   case 1: if (ch == 'b') { ch = Console.Read(); goto case 1; }
           else if (ch == 'c') goto case 2;
           else goto default;
   case 2: Console.WriteLine("input valid");
           break;
   default:    Console.WriteLine("illegal character " + ch);
           break;
}
```

Schleifen

School of Engineering

while

```
while (i < n) {
    sum += i;
    i++;
}
```

do while

```
do {
    sum += a[i];
    i--;
} while (i > 0);
```

for

```
for (int i = 0; i < n; i++)
    sum += i;
```

Kurzform für

```
int i = 0;
while (i < n) {
   sum += i;
   i++;
}
```

foreach-Anweisung

Zum Iterieren über beliebige Collections und Arrays

```
int[] a = {3, 17, 4, 8, 2, 29};
foreach (int x in a) sum += x;
```

```
string s = "Hello";
foreach (char ch in s) Console.WriteLine(ch);
```

```
Queue q = new Queue(); // Elemente sind vom Typ object
q.Enqueue("John"); q.Enqueue("Alice"); ...
foreach (string s in q) Console.WriteLine(s);
```

Sprünge (Vorsicht!)

zh aw School of Engineering

break; Zum Aussprung aus Schleifen und switch-Anweisungen.

Kein break mit Marke wie in Java (goto anstelle).

continue; Kann in Schleifen verwendet werden, um an den Schleifenanfangzurückzuspringen.

goto case 3: Kann in einer switch-Anweisung zum Ansprung einer ihrer case-Marken verwendet werden.

C#

```
myLab:
...
goto myLab;
```

Springt zur Marke myLab.

Restriktionen:

- kein Einsprung in einen Block
- kein Aussprung aus einem finally-Block

return-Anweisung

School of Engineering

Aussprung aus Methoden

```
void Foo (int x) {
   if (x == 0) return;
   ...
}
```

Aussprung aus Funktionsmethoden - mit Rückgabewert

```
int Max (int a, int b) {
   if (a > b) return a; else return b;
}
```

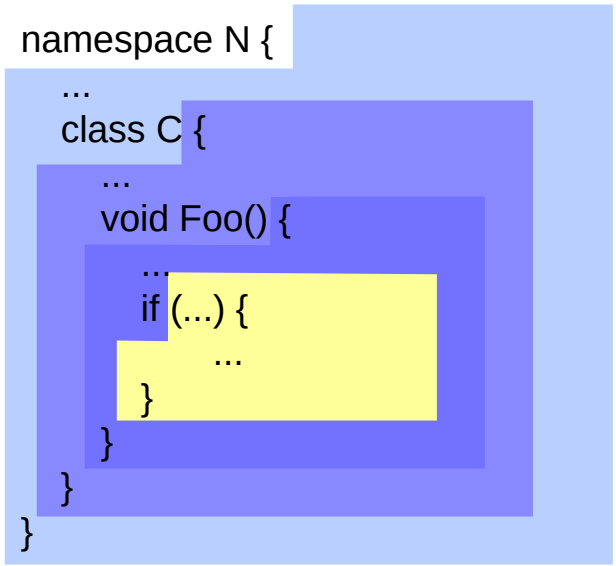
```
class C {
   static int Main() {
      ...
      return errorCode;// Rückgabe eines Fehlercodes
                       // kann in Dos-Box mittels Variable
                       // errorlevel abgeprüft werden).
   }
}
```

Deklarationen

Deklarationsraum

- Sichtbarkeitsbereich von Deklaration

- Arten von Deklarationsräumen
  - Namespace: Deklarationen von Klassen, Interfaces, Structs, Enums, Delegates
  - Klasse, Interface, Struct: Deklarationen von Feldern, Methoden, ...
  - Block: Deklarationen lokaler Variablen
  - geschachelter Block: Deklarationen lokaler Variablen
  - Enum: Deklarationen von Enumerationskonstanten



Regeln

Deklarationsregeln

- **Kein** Name darf in einem Deklarationsraum **mehrfach deklariert** werden.
- Er darf aber in **inneren Deklarationsbereichen neu deklariert** werden
  - Ausnahme: In einem geschachtelten Block dürfen Variablen NICHT neu deklariert werden

Sichtbarkeitsregeln

- Ein Name ist in seinem **ganzen Deklarationsraum sichtbar** (vor- und nach der Deklaration)
  - Ausnahme: lokale Variablen (in Block) erst sichtbar nach Deklaration
- Kein Name ist **ausserhalb seines Deklarationsraums** sichtbar.
  - Deklarationen in **inneren Deklarationsräumen verdecken (engl. shadow)** gleichnamige Deklarationen aus äusseren Deklarationsräumen.
  - Namen von **Enumerationskonstanten** sind nur sichtbar, wenn sie mit dem Namen des Enumerationstyps qualifiziert werden.
- Sichtbarkeit kann mittels **Sichtbarkeitsattribute** weiter gesteuert werden
  - **private, public, protected, internal,..**

Deklarationsraum Klasse, Interface, Struct

School of Engineering

```
class C {  // gilt auch für Structs
   ... Felder, Konstanten ...
   ... Methoden ...
   ... Konstruktoren, Destruktoren ...
   ... Properties ...
   ... Indexers ...
   ... Events ...
   ... überladene Operatoren ...
   ... geschachtelte Typen (Klassen, Interfaces, Structs, Enums, Delegates) ...
}
```

```
interface IX {
   ... Methoden ...
   ... Properties ...
   ... Indexers ...
   ... Events ...
}
```

Deklarationsraum der Oberklasse gehört nicht zum Deklarationsraum der Unterklasse => gleichnamige Deklarationen in der Unterklasse erlaubt : überschreiben, überladen, überschatten.

Deklarationsraum Enum

```
enum E {
   ... Enumerationskonstanten ...
}
```

```
enum Color {red,green,blue}
Color c = Color.red;

Color c2 = c+1;
int i = (int)c;
c = (Color)i;
```

Deklarationattribut public,private

School of Engineering

public überall bekannt, wo deklarierender Namespace bekannt ist

- Standard für:
  - Interface-Members
  - Enum-Members

Java: Default ist "package"

- 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 (Felder, Methoden, ..., geschachtelte Typen)
  - Struct-Members (Felder, Methoden, ..., geschachtelte Typen)
- Beispiel

werden klein geschrieben

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

Deklarationattribut protected

zh aw School of Engineering

protected Sichtbar in deklarierender Klasse und ihren Unterklassen

Beispiel

```
public class Stack {
   protected int[] values = new int[32];
   protected int top = -1;
   public void Push(int x) {...}
   public int Pop() {...}
}
public class BetterStack : Stack {
   public bool Contains(int x) {
      foreach (int y in values) if (x == y) return true;
      return false;
   }
}
public class Client {
   Stack s = new Stack();
   ... s.values[0] ... // verboten: Compile-Fehler
}
```

Deklarationattribut internal

internal Sichtbarkeit bezieht sich darauf, was bei einer Übersetzung sichtbar ist, d.h. der Assembly

csc A.cs B.cs C.cs — Alle internal-Members von A, B, C sehen sich gegenseitig.

```
namespace N {
   internal class A {...}
}
```

```
namespace N {
   internal class B {...}
}
```

```
namespace N {
   internal class C {...}
}
```

Lernkontrolle: Zugriff auf private Members

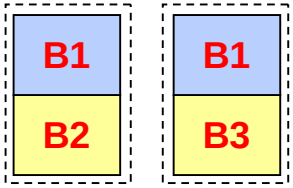
```
class B {
   private int x;
   ...
}


class C {
   private int x;

   public void f (C c) {
     x = ...;

     c.x = ...;



     B b = ...;
     b.x = ...;
   }
}
```

Deklarationsraum Block

- Verschiedene Arten von Blöcken

```
void foo (int x) {     // Methodenblock                    B1
    ... lokale Variablen ...
    if (...) {         // geschachtelter Block      B2
        ... lokale Variablen ...
    }
    for (int i = 0; ...) {    // geschachtelter Block B3
        ... lokale Variablen ...
    }
}
```

- Deklarationsraum eines Blocks schliesst Deklarationsräume geschachtelter Blöcke ein.



- Formale Parameter gehören zum Deklarationsraum des Methodenblocks.
- Deklaration einer Laufvariable gehört zum Deklarationsraum des for-Blocks.
- Deklaration einer lokalen Variablen muss ihrer Verwendung **vorausgehen**.

Lernkontrolle: Deklaration lokaler Variablen

School of Engineering

```
void foo(int a) {   // Parameter a gehört zu innerem Block
   int b;
   if (...) {
       int b;       // ?
       int c;
       int d;
       ...
   } else {
       int a;       // ?
       int d;       // ?
   }
   for (int i = 0; ...) {...} // ?
   for (int i = 0; ...) {...}  // ?

   int c;           // ?
}
```

Deklarationsraum Namespaces

Namespaces

File: XXX.cs

```
namespace A {
   ...
   namespace B {  // voller Name: A.B
      ...
   }
}
```

File: YYY.cs

```
namespace A {
   ...
   namespace B {...}
}
```

```
namespace C {...}
```

File: ZZZ.cs

```
namespace A.B {
   ...
}
```

- Eine Datei kann mehrere Namespaces enthalten.
- Ein Namespace kann über mehrere Dateien verteilt sein. Gleichnamige Namespaces bilden einen gemeinsamen Deklarationsraum.
- Typen die in keinem Namespace enthalten sind, kommen in Default-Namespace (Global Namespace).

Benutzung fremder Namespaces

Color.cs

```
namespace Util {
    public enum Color {...}
}
```

Figures.cs

```
namespace Util.Figures {
    public class Rect {...}
    public class Circle {...}
}
```

Triangle.cs

```
namespace Util.Figures {
    public class Triangle {...}
}
```

```
using Util.Figures;

class Test {
    Rect r;          // Benutzung ohne Qualifikation (weil using Util.Figures)
    Triangle t;
    Util.Color c;    // Benutzung mit Qualifikation
}
```

- Fremde Namespaces müssen entweder
- mit *using* importiert oder
- als Qualifikation vor verwendeten Namen geschrieben werden
- Fast jedes Programm benötigt Namespace System => using System;

C#-Namespaces vs. Java-Pakete

C#

Datei kann mehrere Namespaces enthalten

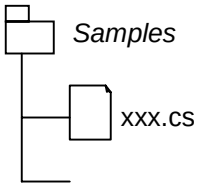
xxx.cs

```
namespace A {...}
namespace B {...}
namespace C {...}
```

Namespaces werden nicht auf Verzeichnisse abgebildet

xxx.cs

```
namespace A {
   class C {...}
}
```



Java

Datei kann nur 1 Paketangabe enthalten

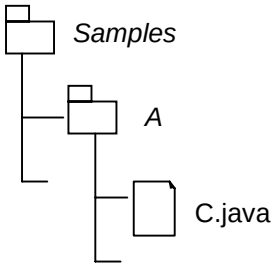
xxx.java

```
package A;
...
...
```

Pakete werden auf Verzeichnisse abgebildet

C.java

```
package A;
class C {...}
```



… Namespaces vs. Pakete

School of Engineering

C#

Es werden *Namespaces* importiert

```
using System;
```

NS werden in andere NS importiert

```
namespace A {
    using C; // gilt nur in dieser Datei
}            // für A
namespace B {
    using D;
}
```

Alias-Namen möglich

```
using F = System.Windows.Forms;
...
F.Button b;
```

Wenn man explizite Qualifikation aber kurze Namen haben will.

Es werden *Klassen* importiert

```
import java.util.LinkedList;
import java.awt.*;
```

Klassen werden in Dateien importiert

```
import java.util.LinkedList;
```

Java kennt Sichtbarkeit innerhalb Paket

```
package A;
class C {
    void f() {...}  // package
}
```

C# kennt Sichtbarkeit in Assembly (!= Namespace)

Klassen und Structs
1.Teil (Einführung)

Klassen und Structs

- In C# gibt es zwei strukturierte Datentypen
- **Klassen**
  - Mittels *new* werden Instanzen erstellt
  - Werte werden auf dem Heap abgelegt
  - Sind Referenztypen
  - Haben "volle" OO Funktionalität (Vererbung, etc.)
- **Structs**
  - Mittels *new* werden Werte initialisiert
  - Werte sind auf dem Stack abgelegt
  - Sind Wertetypen
  - Haben "eingeschränkte" OO Funktionalität
- Vorteile
  - Performance: der Stack kann effizienter verwaltet werden als der Heap
  - Modellierung von eigenen Wertetypen möglich, z.B. Complex,Vector

Aufbau von Klassen (oder Structs)

School of Engineering

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

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

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

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

Klassen

School of Engineering

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

- Objekte d.h. Instanzen von Klassen werden am Heap angelegt -> sind **Referenztypen**
- Objekte müssen mit *new* erzeugt werden

```
Stack s = new Stack(100);
```

- Können erben, vererben und Interfaces implementieren

Structs

Deklaration

```
struct Point {
    public int x, y;    // Felder
    public Point (int x, int y) { this.x = x; this.y = y; }// Konstruktor
    public void MoveTo (int a, int b) { x = a; y = b; }    // Methoden
}
```

Verwendung

```
Point p;                // noch uninitialisiert
Point p = new Point(3, 4);// Initialisierung mit Konstruktor-Aufruf
p.x = 1; p.y = 2;       // Feldzugriff
p.MoveTo(10, 20);       // Methodenaufruf
Point q = p;            // Objektzuweisung (alle Felder)
```

Bemerkungen

- **Structs sind Werttypen!**
  Bei einer Zuweisung werden die Werte kopiert

Structs

```
struct Point {
  public int x, y;                                    // Felder
  public Point (int x, int y) { this.x = x; this.y = y; } // Konstruktor
  public void MoveTo (int a, int b) { x = a; y = b; }     // Methoden
}
```

- Objekte werden am Stack statt am Heap angelegt.
  - \+ speichersparend, effizient, belasten GC nicht
  - \- Lebensdauer auf Container (Block, Methode) eingeschränkt ->nicht für dynamische Datenstrukturen
- Werden mit *new* initalisiert

```
Point p;    // Felder von p sind nicht initalisiert
Point q = new Point(5,4);  // Konstruktor-Aufruf inialisiert Werte
```

- Felder dürfen hingegen bei der Deklaration nicht initialisiert werden.

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

- Structs können weder erben noch vererben, aber Interfaces implementieren.

Vergleich Klassen und Structs

School of Engineering

Klassen

- Referenztypen
  - Objekte am Heap angelegt
- unterstützen Vererbung
- alle Klassen von *object* abgeleitet
- können Interfaces implementieren
- Parameterloser Konstruktor erlaubt
- können Destruktoren haben

Structs

- Werttypen
  - Objekte am Stack angelegt
- keine Vererbung
- zu *object* kompatibel
- können Interfaces implementieren
- keine parameterlosen Konstruktoren
- keine Destruktoren

Felder und Konstanten

- **Feld**

```
class C {
   public int value = 0;
```

  - Initalisierung in Deklaration optional
  - Felder werden in Deklarationsreihenfolge initialisiert
  - Initialisierung darf nicht auf Felder und Methoden zugreifen
  - Struct-Felder dürfen nicht initialisiert werden

C++ const

- **Konstante**

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

C#

  - Muss Initialisierungswert bei der **Deklaration** haben
  - Wert muss zur Compilezeit berechenbar sein

- **ReadOnly-Feld**

Java: final

```
   public readonly DateTime date;
```

  - 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 Feld)

Zugriff innerhalb Klasse

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

Zugriff aus anderen Klassen

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

Statische Felder und Konstanten

Daten der Klasse und nicht des Objekts

```
class Rectangle {
    static Color defaultColor;     // einmal pro Klasse vorhanden
    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
```

Klassenname

Konstanten (const) sind automatisch static

Methoden

School of Engineering

Beispiele

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

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

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

Aufruf aus Klasse C

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

Aufruf aus anderen Klassen

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

Statische Methoden

School of Engineering

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();
```

Arten von Parametern (wie Java)

School of Engineering

Operationen auf Klassendaten (statische Daten)

Value-Parameter (Eingangsparameter)

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

- "call by value" bei Wertetypen
- Formaler Parameter ist Kopie des aktuellen Parameters
- akt.Parameter = beliebiger Ausdruck

ref-Parameter (Übergangsparameter)

```
void Inc(Hastable h) {
   h["hallo"] = "world";
}
void f() {
    Hashtable g;
    Inc(g); //
}
```

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

Arten von Parametern (neu)

ref-Parameter (Übergangsparameter)

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

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

out-Parameter (Ausgangsparameter)

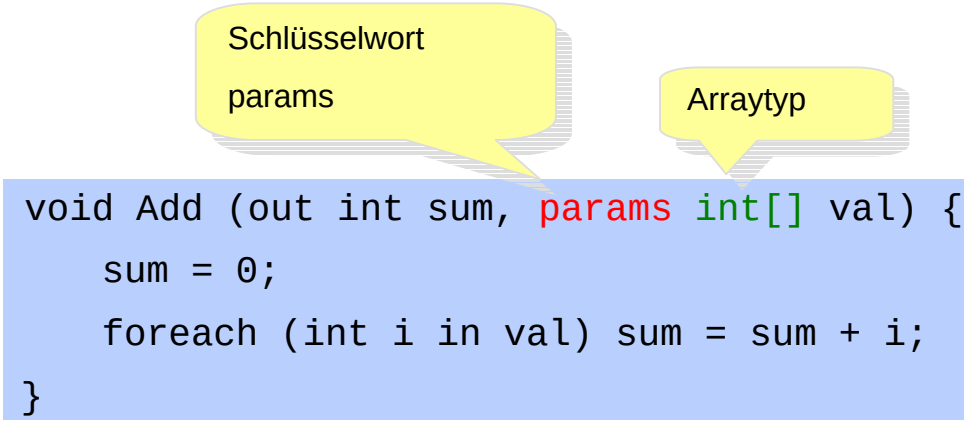
```
void Read (out int first, out int next) {
   first = Console.Read();
   next = Console.Read();
}
void f() {
   int first, next;
   Read(out first, out next);
}
```

- "out" von **Wertetypen**
  Wie ref-Parameter, aber wird zur Rückgabe von Werten verwendet.
- Muss bei **Aufruf nicht initialisiert** sein.
- Darf in der Methode nicht verwendet werden, bevor ihm ein Wert zugewiesen wurde.

Auch ref und out Parameter von Referenztypen sind möglich

Variable Anzahl von Parametern

Letzte *n* Parameter dürfen beliebig viele Werte eines bestimmten Typs sein.




```
void Add (out int sum, params int[] val) {
    sum = 0;
    foreach (int i in val) sum = sum + i;
}
```

params geht nicht für *ref* und *out*

Aufruf

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

Weiteres Beispiel

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

Überladen von Methoden

School of Engineering

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, s); // mehrdeutig zwischen F(int x, long y) und F(long x, int y);
F(i, i); // mehrdeutig zwischen F(int x, long y) und F(long x, int y);
```

Methoden dürfen sich nicht nur im Funktionstyp, durch *params*-Parameter oder durch *ref* vs. *out* unterscheiden!

Überladen von Methoden

Überladene Methoden dürfen sich nicht bloss im Funktionstyp /Rückgabewert unterscheiden

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

```
F();// Aufruf, bei dem der Funktionswert verworfen wird, nicht auflösbar
```

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
```

Grund liegt in der CLR-Implementierung: An der Aufrufstelle wird nicht die Adresse sondern eine Beschreibung der aufzurufenden Methode angegeben. Diese Beschreibung ist in beiden Fällen gleich.

Konstruktoren bei 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 = x; 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);
```

andere Syntax
als Java (C++ angelehnt)

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

Default-Konstruktor

School of Engineering

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
```

Konstruktoren bei Structs

School of Engineering

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 uninitialisiert
Complex c1 = new Complex();  // c1.re == 0, c1.im == 0
Complex c2 = new Complex(5); // c2.re == 5, c2.im == 0
Complex c3 = new Complex(10, 3);   // c3.re == 10, c3.im == 3
```

- Jeder Struct hat einen **parameterlosen Default-Konstruktor**, der alle Felder initialisiert (auch wenn es andere Konstruktoren gibt).
- Structs dürfen daher keinen expliziten parameterlosen Konstruktor haben. **Grund liegt in der Implementierung des CLR**
- Ein struct-Konstruktor muss **alle Felder** des Struct initialisieren.

Statische Konstruktoren

School of Engineering

Sowohl bei Klassen als auch bei Structs möglich

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

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

auch in Java möglich

```
static {
   ....
}
```

- Müssen parameterlos sein (auch bei Structs) und haben kein public oder private.
- Es darf nur einen statischen Konstruktor pro Klasse/Struct geben.
- Wird genau einmal ausgeführt, bevor das erste Objekt der Klasse erzeugt oder das erste Mal auf eine statische Variable der Klasse zugegriffen wird.
- Verwendet für Initialisierungsarbeiten, z.B. Initialisierung statischer Felder.

Destruktoren

```
class Test {

   ~Test() {
      ... Abschlussarbeiten ...
   }
}
```

in Java finalize()

- 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*.
- Achtung: Es wird nicht gesagt, **wann** und **ob** ein der Destruktor aufgerufen wird
- Structs dürfen keinen Destruktor haben (Grund nicht ganz klar).

Geschachtelte Klassen

School of Engineering

```
public 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);
}
```

Zweck

- Für Hilfsklassen, die versteckt bleiben sollen
  - Andere Klassen sehen innere Klasse nur, wenn sie *public* ist
  - Innere Klasse sieht alle Members der äusseren Klasse (auch private).
  - Äussere Klasse sieht nur public-Members der inneren Klasse.
  - Geschachtelte Typen können auch Structs, Enums, Interfaces und Delegates sein.

Klasse aus mehreren Teilen: partial

```
public partial class C {
  int x;
  public void M1(...) {...}
  public int M2(...) {...}
}
```

Datei Part1.cs

```
public partial class C {
  string y;
  public void M3(...) {...}
  public void M4(...) {...}
  public void M5(...) {...}
}
```

Datei Part2.cs

Zweck

- Teile können nach Funktionalitäten gruppiert werden
- erster Teil kann maschinengeneriert sein, zweiter Teil handgeschrieben

Zusammenfassung

- Ausdrücke
  - im Wesentlichen wie Java
- Anweisungen
  - im Wesentlichen wie Java
- Deklarationsräume
  - im Wesentlichen wie Java

- Objektorientierte Konzepte
  - Ähnlich zu Java
- Klassen und Strukturen
  - Klassen wie Java, Strukturen neu

Fragen?

Lösung: Deklaration lokaler Variablen

School of Engineering

```
void foo(int a) {          // Parameter a gehört zu innerem Block
    int b;
    if (...) {
         int b;?           // Fehler: b bereits in äusserem Block deklariert
         int c;
         int d;
         ...
    } else {
         int a;            // Fehler: a bereits im äusseren Block deklariert
         int d;            // ok: keine Überschneidung mit d im then-Block
    }
    for (int i = 0; ...) {...}
    for (int i = 0; ...) {...}  // ok: keine Überschneidung mit i aus letzter
    int c;                 // Fehler: c bereits in einem inneren Block deklariert
}
```