Klassen und Structs

Inhalt von Klassen und Structs

```
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

```
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

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
  - \- Lebensdauer auf Container eingeschränkt, nicht für dynamische Datenstrukturen
- Können aber müssen nicht mit new erzeugt werden.

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

- Felder 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, aber Interfaces implementieren.

Sichtbarkeitsattribute (Auszug)

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

Beispiel

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

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 = ...;              // falsch! Methode der Klasse C darf nicht auf private Members
                                // einer anderen Klasse zugreifen.
    }
}
```

Felder und Konstanten

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

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

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

Zugriff aus anderen Klassen

```
C c = new C();
... 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 ...

Konstanten dürfen nicht static deklariert werden

Methoden

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

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

Value-Parameter (Eingangsparameter)

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

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

ref-Parameter (Übergangsparameter)

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

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

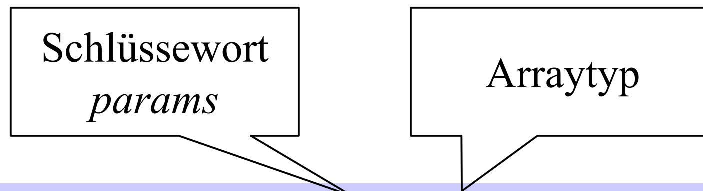
out-Parameter (Ausgangsparameter)

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

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

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
```

Anderes Beispiel

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

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);    // mehrdeutig zwischen F(int x, long y) und F(long x, int y); => Compilefehler
F(i, i);    // mehrdeutig zwischen F(int x, long y) und F(long x, int y); => Compilefehler
```

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 unterscheiden

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

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

- 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

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 <u>kein</u> 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

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 <u>alle</u> Felder des Struct initialisieren.

Statische Konstruktoren

Sowohl bei Klassen als auch bei Structs möglich

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

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

- 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 ...
    }

}
```

- Entspricht Finalizers in Java.
- 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*.
- Ist gefährlich und meist nicht nötig
  (es könnte z.B. ein totes Objekt in eine lebende Datenstruktur eingehängt werden).
- Structs dürfen keinen Destruktor haben (Grund nicht ganz klar).

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 ein Feld benutzt ("smart fields")

```
Data d = new Data();

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

Durch Inlining der get/set-Aufrufe geht Zugriff gleich schnell wie normaler Feldzugriff.

Properties (Forts.)

Alle Zuweisungsoperatoren funktionieren auch mit Properties

```
class C {
    private static int size;

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

Size = 3;
Size += 2;  // Size = Size + 2;
```

Properties (Forts.)

get oder set kann fehlen

```
class Account {
    long balance;

    public long Balance {
        get { return balance; }
    }
}

x = account.Balance;          // ok
account.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 Felder in Interfaces.
- Daten sind über Reflection deutlich als Property erkennbar (wichtig für Komponentenorientierte Programmierung).

Indexer

Programmierbarer Operator zum Indizieren einer Folge (Collection)

Typ des indizierten Ausdrucks

Name (immer *this*)

Typ und Name des Indexwerts

```
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);
            }
    }
}
```

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.

Indexer (anderes Beispiel)

```
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"]);
```

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 == 10, c.y == 8
```

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

Überladen von && und ||

Um && und || zu überladen, muß man &, |, true und false überladen

```
class TriState {
    int state;  // -1 == false, +1 == true, 0 == undecided
    public TriState(int s) { state = s; }

    public static bool operator true (TriState x) { return x.state > 0; }
    public static bool operator false (TriState x) { return x.state < 0; }

    public static TriState operator & (TriState x, TriState y) {
        if (x) return y; else return new TriState(-1);
    }
    public static TriState operator | (TriState x, TriState y) {
        if (x) return new TriState(1); else return y;
    }
}
```

true und false werden nur implizit aufgerufen

```
TriState x, y;
if (x) ...          => if (TriState.true(x)) ...
x = x && y;         => x = TriState.false(x) ?  x : TriState.&(x, y);
x = x || y;         => x = TriState.true(x) ? x : TriState.|(x, y)
```

Überladen von Konversionsoperatoren

Implizite Konversion

- Wenn Konversion immer möglich ist und kein Genauigkeitsverlust stattfindet
- Z.B.: long = int;

Explizite Konversion

- Wenn Laufzeittypprüfung nötig ist oder u.U. abgeschnitten wird
- Z.B. int = (int) long;

Konversions-Operatoren für eigenen Typ

```
class Fraction {
    int x, y;
    ...
    public static implicit operator Fraction (int x) { return new Fraction(x, 1); }
    public static explicit operator int (Fraction f) { return f.x / f.y; }
}
```

Benutzung

```
Fraction f = 3;     // implizite Konversion, f.x == 3, f.y == 1
int i = (int) f;    // explizite Konversion, i == 3
```

Geschachtelte Typen

SSW

```
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.

Unterschiede zu Java und C++

SSW

- Keine anonymen Klassen wie in Java
- (Noch) keine Templates wie in C++
- Andere Standard-Sichtbarkeit für Members
  C#: private
  Java: package
- Andere Standard-Sichtbarkeit für Klassen
  C#: internal
  Java: package