C# – 2

by

Dr. Günter Kolousek

Allgemeines zu Klassen

- direkt oder indirekt von `System.Object`
- Zugriffsrechte
  - Member: `public`, `protected`, `private` (default), `internal` (gleiches assembly), `protected internal` (abgeleitete Klasse *oder* gleiches assembly), `private protected` ((gleiche oder abgeleitete Klasse) *und* gleiches assembly!)
  - Klasse: `public`, `internal` (default)
- Geschachtelte Klassen
  - kein Zugriff auf Members der umschließenden Klasse (wie in Java)
  - keine anonymen Klassen

Zugriff auf umschließende Klasse

```
using System;
public class Outer {
    int i=1;

    class Inner {  // -> default: private to Outer!
        Outer o;
        int j=2;
        public int result() { return o.i + j; }
        public Inner(Outer o) { this.o = o; }
    }

    public static void Main() {
        Outer o=new Outer();
        Inner n=new Inner(o);
        Console.WriteLine(n.result());
    } }
```

Allgemeines zu Strukturen

- können nicht von anderen Typen abgeleitet werden
- Zugriffsrechte der Members: `public`, `internal`, `private` (default)
- Attribute können nicht initialisiert werden
- Attribut muss Wert haben bevor zugegriffen werden kann
- benutzerdefinierter Konstruktor *muss* Parameter aufweisen
- keine Properties!

Strukturen

```
struct X {  // nicht: struct X : Y {
  public int x;  // nicht: int x=1;
  public X(int p) { x = p; }  // nicht X() {...}

}
X x1;
// nicht: Console.WriteLine(x1.x);
x1.x = 1;
Console.WriteLine(x1.x);
X x2=new X(2);  // -> constructor!
Console.WriteLine(x2.x);
X x3=new X{ x = 3 };  // object initializer
X x4=new X();  // -> default constructor!
Console.WriteLine(x4.x);  // -> 0
```

Strukturen – 2

ab C# 7.2

```
using System;
readonly struct Point {
    readonly public int x;
    readonly public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public double distance() {
        return Math.Sqrt(x * x + y * y);
    }
    override public string ToString() {return $"({x}, {y})"; }
}
class Program {
    static void Main() {
        Point p1=new Point(1, 2);
        Console.WriteLine(p1); // -> (1, 2)
        // p1.x = 3;  // does not compile
    } };
```

Konstanten, Read-only

- Konstanten mittels `const` (in Java `static final`)
  - z.B. `const int max=3;`
  - kann nur über Klassenname zugegriffen werden (nicht über Instanz)
    - haben aber kein `static`!
  - zur Übersetzungszeit!
    - d.h. Compiler ersetzt mit Wert!
- Read-only mittels `readonly` (in Java `final`)
  - Zuweisung nur bei Initialisierung oder in Konstruktor
  - d.h. zur Laufzeit

Konstruktoren und Destruktoren

- Konstruktoren wie in Java
- statische Konstruktoren (anstatt Java ”static initializer”)
- Destruktoren (wird vom GC automatisch aufgerufen)

```
public class X {
    public X() : this(0) {}  // Konstruktor
    public X(int a) {}
    public ~X() {}  // eher nicht verwenden!
}
```

  - Destruktor ruft implizit `Finalize()` auf, d.h. Destruktor wird vom Compiler ersetzt durch:

```
protected override void Finalize() {
    try {
        // Cleanup statements...
    } finally {
        base.Finalize();
    } }
```

Destruktoren

- kein Destruktor für `struct`
- max. ein Destruktor je Klasse
- Destruktor wird nicht vererbt
- Destruktor hat keine Parameter
- Destruktor kann nicht manuell aufgerufen werden
  - wenn manuelle Verwaltung, dann `IDisposable` implementieren!
- es ist nicht deterministisch *wann* Destruktor aufgerufen wird ($\rightarrow$ GC)
- Destruktor hat einen großen Overhead in C#!

Methoden und Parameter

- ”wie in Java”
- Referenzparameter mittel `ref`

```
void swap(ref int x, ref int y) {
    int tmp=x;
    x = y;
    y = tmp;
}
...
int i=1; int j=2;
swap(ref i, ref j);  // auch hier ref!
```

- Detto: Out-Parameter mittels: `out`
- Detto: In-Parameter mittels: `in` (ab C# 7.2)
  - Parameter dann nicht veränderbar!

Methoden und Parameter – 2

- Variable Anzahl an Parameter
  - an sich wie in Java, nur syntaktische Unterschiede
  - keine Referenzparameter

```
using System;

class Program {
    static int sum(params int[] nums) {
        int acc=0;
        foreach (var n in nums) {
            acc += n;
        }
        return acc;
    }
    static void Main() {
        Console.WriteLine(sum(1, 2, 3));  // -> 6
    }
}
```

Methoden und Parameter – 3

```
using System;

public class Shape {
    private int x;
    private int y;
    // default values
    public void move(int dx=1, int dy=1) {
        x += dx;
        y += dy;
    }
    public static void Main() {
        var shape=new Shape();
        shape.move();
        // named parameters
        shape.move(dy:2, dx:1);
        Console.WriteLine(shape.x + ", " + shape.y)
    } }
```

Statische Klassen

- `static` Klassen können nicht instanziert werden

```
public static class Funktions {
    public static double x2(double x) => x * x;
}
```

- werden als Utility-Klassen verwendet
- sind ”notwendig”, da in C# keine freien Funktionen möglich sind
  - vgl. Java und C# zu z.B. Python und C++

Vererbung

```
using System;

public class Moveable {
    protected int x; protected int y;
    public Moveable(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public virtual void move(int dx, int dy) {
        x += dx;
        y += dy;
        Console.WriteLine($"moved to {x},{y}");
    }
}
```

Vererbung – 2

```
public class Car : Moveable {
    const int max_v=150;
    public int v;
    public Car(int x, int y, int v) : base(x, y) {
        this.v = v;
    }
    public Car(int x, int y) : this(x, y, max_v) {}
    public override void move(int dx, int dy) {
        Console.WriteLine(”Car”);
        base.move(dx, dy);
    } }

public class Program {
    public static void Main() {
        Moveable m=new Car(0, 0);
        m.move(10, 20);
    } }
```

Vererbung – 3

- Achtung: Polymorphismus **nicht** automatisch
- Schlüsselwörter
  - `virtual`: dynamisches Binden (ansonsten statisch!)
  - `override`: Überschreiben einer geerbten Methode
  - new: Ausschalten von `virtual`
- Regeln
  - Überschreiben: `virtual` in Basisklasse und `override` in Kindklasse
  - Neu implementieren: `virtual` in Basisklasse und new in Kindklasse (`virtual` wird ”ausgeschalten”)
  - Weder `override` noch new, dann defaultmäßig: new und Warnung
  - `override` ohne `virtual` $\rightarrow$ Fehler

Vererbung – 4

- `abstract` (wie in Java)
- `sealed` (wie `final` in Java)
  - auf Klassen: kann nicht abgeleitet werden
  - auf Methoden: kann nicht überschrieben werden
- Beispiel

```
class sealed Point {}

class Rectangle {
    public sealed move(int dx, int dy);
}
```

Interface

- `interface` (automatisch `public`)
- nur Methoden-Prototypen (automatisch `public` und `abstract`)
  - d.h. auch keine Konstanten (wie in Java)
  - aber auch Properties, Events, Indexers (ohne Implementierung)
- Namenskonvention: beginnen mit ”I” (z.B. `System.IComparable`)
- ”Implements” mittels `:` wie bei Klassenvererbung
  - durch Beistriche getrennt (Basisklasse am Anfang!)

Interface – 2

```
interface Moveable {
    void move(int x, int y);
}

public class Vehicle {
}

public class Car : Vehicle, Moveable {
  int x; int y; int z;

  public void move(int x, int y) {
    this.x = x;
    this.y = y;
    Console.WriteLine("Move to ({0}, {1})", x, y);
  }
}
```

Interface – 3

```
using static System.Console;
interface Moveable { void move(int x, int y); }
interface Moveable3D { void move(int x, int y); }
public class Car : Moveable, Moveable3D {
  int x; int y; int z;
  public void move(int x, int y) {
      this.x = x; this.y = y;
      WriteLine("Move to ({0}, {1})", x, y);
  }
  // explicit interface implementation:
  void Moveable3D.move(int x, int y) {  // no public!
      move(x, y);  // das "normale" move
      this.z += 1;
      WriteLine("Move to ({0},{1},{2})", x, y, z);
  }
  public static void Main() {
      Car c=new Car();
      c.move(1,2);
      ((Moveable3D)c).move(1,2);
  } }
```

Interface – 4

```
// schnipp-schnapp:
House h=new House();
Moveable3D h3d=(Moveable3D)h;  //-> InvalidCastExc.

// better:
Car c=new Car(); c.move(3, 4);
Moveable3D c3d=c as Moveable3D;
// or:
// c3d = (c is Moveable3D) ? (Moveable3D)c : null;
if (c3d != null) c3d.move(5, 6);
```

Exceptions

- generelles `catch`: `try { ... } catch { ... }`
- keine `throws` Klausel wie in Java
- Weiterreichen einer Exception mittels `throw` ohne Parameter

```
public class FullException : Exception {
  public FullException() {
  }

  public FullException(string msg) : base(msg) {
  }

  public FullException(string msg, Exception inner)
    : base(msg, inner) {
  }
}
```

Exceptions – 2

- Basisklasse `System.Exception`
  - `SystemException` … meist von Runtime geworfen
    - `System.IO.IOException`
  - `ApplicationException` … war für eigene Exceptions vorgesehen; jetzt nicht mehr verwenden
- Weitgehend alle Exceptions in `System` von `SystemException` abgeleitet
  - Wichtige Ausnahme: `IOException` und Subklassen in `System.IO`

Exceptions – 3

- Wichtige Subklassen von `SystemException`:
  - `ArithmeticException`
    - `DivideByZeroException` ... bei integralen oder `decimal`
    - `NotFiniteNumberException` ... bei Gleitkommazahlen
    - `OverflowException` ... wenn in einem ”checked” Kontext
  - `ArgumentException` ... Argument ungültig, $\rightarrow$ Subklassen!
    - `ArgumentNullException`
    - `ArgumentOutOfRangeException`
    - `InvalidEnumArgumentException`
  - `IndexOutOfRangeException`
  - `InvalidCastException`
  - `InvalidOperationException` ... Objekt: ungültiger Zustand
  - `FormatException` ... z.B. bei Verwendung von `Parse()`
  - `KeyNotFoundException` ... $\rightarrow$ Collections
  - `NotSupportedException` ... angeforderte Operation wird nicht unterstützt
  - `NullReferenceException`
  - `RankException` ... Array: ”falsche” Anzahl an Dimensionen

Exceptions – 3

- Wichtige Subklassen von `IOException`:
  - `DirectoryNotFoundException`
  - `DriveNotFoundException`
  - `EndOfStreamException`
  - `FileNotFoundException`
  - `PathToLongException`
- Schreiben eigener Exceptions
  - von `Exception` ableiten
  - überschreiben wenn notwendig
  - wenn serialisierbar
    $\rightarrow$ Attribute `Serializable()` und `NonSerialized()`
  - Konstruktor(en)

Exceptions – 4

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

public class Program {
    public static void Main() {
        int i=0;
        int j=0;
        try {
            WriteLine(i / j);
        } catch (DivideByZeroException e) {
            // property Message
            WriteLine($"{e.Message}");
        } catch {
            WriteLine("catch all");
        } finally {
            WriteLine("finally");
        } } }
```

Exceptions – 5

- Rethrowing
  - gleiche nochmals werfen: `throw;`
    - nicht: `throw e;` → Stacktrace!
  - `throw new DivideByZeroException("throw again", e);`
- (wichtige) `System.Exception` properties
  - `Data` ... key/value für zusätzliche Informationen
  - `InnerException` ... Referenz zu "vorhergehender" Exception
  - `Message`
  - `StackTrace`

Exceptions – 6

```
using System;
public class Program {
  public static void Main() {
    try {
        InvalidOperationException e=
          new InvalidOperationException();
        e.Data["reason"] = "full";
        throw e;
    // exception filter!
    } catch (InvalidOperationException e)
      when ((string)e.Data["reason"] == "empty") {
        Console.WriteLine("Tank leer");
    } catch (InvalidOperationException e)
      when ((string)e.Data["reason"] == "full") {
        Console.WriteLine("Tank voll");
    } catch {
        Console.WriteLine("ungültiger Zustand");
    } } }
```

Exceptions – 7

```
using System;
public class Program {
    // default is ”checked is off”
    // change it using compiler option: -checked+
    public static void Main() {
        float a=3.4e38f;
        float b=3.4e38f;
        Console.WriteLine(a + b);  // -> +unendlich

        // uint c=4294967295 + 1;  // -> comp error
        uint c=unchecked(4294967295 + 1);
        Console.WriteLine(c);  // -> 0

        uint d=4294967295;
        uint e=1;
        Console.WriteLine(d + e);  // -> 0
        checked {
            Console.WriteLine(d + e);
        }
        // -> System.OverflowException: Arithmetic\n
        //     operation resulted in an overflow.
    } }
```

Properties

```
using System;
public class Test {
    public static void Main() {
        Car c = new Car(); c.speed = 125;
        System.Console.WriteLine(c.speed);
        c.speed = 250;
        System.Console.WriteLine(c.speed);
    }
}

public class Car {
   double _speed;
   public double speed {
       // nur get -> read-only
       get { return _speed; }
       // nur set -> write-only
       set { _speed = (value > 150) ? 150 : value; }
   }
}
```

Properties – 2

```
using System;
public class Test {
    public static void Main() {
        Car c = new Car(); c.incr(125);
        //c.speed = 123;  // -> error
        System.Console.WriteLine(c.speed);
        c.incr(125);
        System.Console.WriteLine(c.speed);
    }
}

public class Car {
    public int speed {  // auto-implemented
        get;
        private set;  // -> visibility!
    }
    public void incr(int delta) {
        speed = speed + delta;
    }
}
```

Properties – 3

- Unterschiedliche visibitity modifier für get und set!
  - z.B. nicht für Instanzvariable möglich
- Property ist keine Variable $\rightarrow$ keine Übergabe an `ref` oder `out` formale Parameter!
- `static` Property möglich!
- `virtual`, `override`, `new`, `sealed` und `abstract` möglich
  - wenn nicht `static`

Operatoren

- die üblichen Kandidaten...
- cast operator: `(int)123L`
- `nameof`: `nameof(calc)` $\rightarrow$ `"calc"`
- `sizeof`: `sizeof(int)` $\rightarrow$ `4`
- `typeof`: `typeof(int)` $\rightarrow$ `System.Int32`
- null-conditional Operator
  - `string first_name = p?.first_name;`
    - äquiv. zu `(p != null) ? p.first_name : null;`
  - `int? len = nums?.Length;`
    - äquiv. zu
```
int? len = (nums != null) ?
           (int?)nums.Length : null;
```
- `??`, `is`, `as`

Operator overloading

```
using System;
public struct Complex {
    public double real;  public double imag;
    public Complex(double real, double imag) {
        this.real = real; this.imag = imag;
    }
    public static Complex operator+(Complex op1, Complex op2)
       return new Complex(op1.real + op2.real,
                          op1.imag + op2.imag);
    }
}

public class Test {
    public static void Main() {
        Complex c1 = new Complex(3, 0);
        Complex c2 = new Complex(1, 1);
        Complex c = c1 + c2;
        Console.WriteLine(c.real + " " + c.imag);
    }
}
```

Operator overloading & Equality

```
using System;
//    ref type!     type-safe Equals(o)!
public class Point : IEquatable<Point> {
  public Point(double _x, double _y) { x = _x; y = _y; }
  public double x { get; }
  public double y { get; }
  public override string ToString() => $"Point({x},{y})";
  // Override equality op. (==) if your type is a base
  // type such as a Point, String,...
  public static bool operator==(Point lhs, Point rhs) {
    if (Object.ReferenceEquals(lhs, null)) {
      if (Object.ReferenceEquals(rhs, null)) return true;
      return false;
    }
    // Equals handles case of null on right side.
    return lhs.Equals(rhs); }
  // also to implement if implementing operator==
  public static bool operator!=(Point lhs, Point rhs)
    => !(lhs == rhs);
```

Operator overloading & Equality – 2

```
    // also to implement if implementing Equals
    public override int GetHashCode() =>
      x.GetHashCode() ^ y.GetHashCode();

    // also to implement if implementing operator==
    public override bool Equals(object o) {
        return this.Equals(o as Point);
    }

    public virtual bool Equals(Point o) {
    // Check for null values and compare run-time types.
        if (o == null || GetType() != o.GetType())
            return false;
        return (x == o.x) && (y == o.y);
    }
}
```

Operator overloading & Equality – 3

```
public class Program {
  static public void Main() {
      Point p1=new Point(1, 1);
      Point p2=new Point(1, 1);
      Point p3=new Point(1, 2);
      Console.WriteLine($"{p1 == p2} | {p1 == p3}");
  }
}
```

Operator overloading & Equality – 4

```
//              inheritance!
class Point3D : Point {
  int z;
  public Point3D(int x_, int y_, int z_):base(x_, y_) {
      z = z_; }
  public override bool Equals(Object o)
      => this.Equals(o as Point3D);
  public override bool Equals(Point o)
      => base.Equals(o) && z == ((Point3D)o).z;
  public override int GetHashCode()
      => base.GetHashCode() ^ z.GetHashCode();
```

Operator overloading & Equality – 5

```
// and now for something completely different...
// implementing a value type like a struct:
public struct Point : IEquatable<Point> {
  public Point(double _x, double _y) { x = _x; y = _y; }
  public double x { get; }
  public double y { get; }
  public override string ToString() => $"Point({x},{y})";
  // Override the equality op. (==) if your type is a base
  // type such as a Point, String,...
  public static bool operator==(Point lhs, Point rhs)
    => lhs.Equals(rhs);
  public static bool operator!=(Point lhs, Point rhs)
    => !(lhs == rhs);
  public override int GetHashCode()
    => x.GetHashCode() ^ y.GetHashCode();
  public override bool Equals(object o)
    => this.Equals((Point)o); }
  public bool Equals(Point o) {
      if (GetType() != o.GetType()) return false;
      return (x == o.x) && (y == o.y);  } }
```

Operator overloading & casts

```
using System;
class Currency {
    private decimal v;
    public Currency(decimal v_)
      => v = v_;
    public static implicit operator double(Currency c)
      => (double)c.v;
    public static explicit operator Currency(double v)
      => new Currency((decimal)v);
    public override string ToString()
      => v.ToString();
}

public class Program {
    public static void Main() {
        Currency c=new Currency(121.45m);
        Console.WriteLine(c + 2.006);
        Console.WriteLine((Currency)123.456);
    }
}
```

Indexer

```
using System;
public class Test {
    public static void Main() {
        StrBox b = new StrBox(); b[0] = ”abc”;
        Console.WriteLine(b[0]);
    }
}
public class StrBox {
    private string[] box = new string[5];
    public string this[int idx] {
        get {
            return (0 <= idx && idx <= box.Length) ?
              box[idx] : null;
        }
        set {
            box[idx] = (0 <= idx && idx <= box.Length) ?
              value : null;
        }
    }
}
```