Die Sprache C# 3. Teil

School of Engineering

- Properties
- Indexer
- Überladene Operatoren
- Klassen erweiterte Konzepte
- Vererbung, Interfaces
- Einfach I/O Operationen

Properties

Quiz

Gegeben sei folgender Code Ausschnitt

```
public class A {
     public String name;
}
....
a.name = "Hallo";
```

wo liegt das Problem?

```
public class A {
    private String  name;

    public setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
....
a.setName("Hallo");
```

geht es nicht eleganter?

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")

in Java lediglich Konvention:
setter und getter Methoden

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

Properties (Forts.)

Alle Zuweisungsoperatoren funktionieren auch mit Properties

```
class C {
  private int size;

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

```
c.Size = 3;
c.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
```

Properties Kurzform

- Wenn lediglich ein Wert gesetzt und/oder gelesen werden soll
  - Es wird automatisch eine entsprechende Variable generiert

```
class C {

    public int Size {
            get;
            set;
    }
}
```

Properties

School of Engineering

- Erleichtern den Zugriff auf Felder
- Syntaktisch identisch zu Feldzugriff
- read-only und write-only-Properties.
- Validierung beim Zugriff.
- Durch **Inlining** der set/get-Aufrufe Zugriff gleich schnell wie Feldzugriff.
- Aussensicht und Implementierung der Felder können unterschiedlich sein.
- Properties sind syntaktisch klar erkennbar, z.B. bei Zugriff via Reflection

Indexer

Quiz

Gegeben sei folgender Code Ausschnitt

```
String s = "Hallo";
char c = s.charAt(3);
```

geht es nicht eleganter?

Indexer

- **Zugriff auf einzelne Werte mittels Index**

```
class File {
   FileStream s;

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

Typ des indizierten Ausdruck

immer this

Typ und Name des Indexwerts

Verwendung

```
File f = ...;
byte 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 (weiteres 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"]);
```

Indexer

- Erleichtert den Zugriff auf einzelne Werte von sequentiell organisierten Datenstrukturen
  - Arrays
  - Collections
  - Streams
  - …
- Vereinheitlicht den Zugriff auf Werte zwischen eingebauten (arrays) und Bibliotheksdatenstrukturen (siehe z.B. assoziative Arrays)
- Es können beliebig viele Indexer definiert werden, jedoch für jeden Index Typ nur einer
- Viele .NET Datenstrukturen bieten Indexer an
  - Strings
  - Dictionary
  - List

Überladene Operatoren

Quiz

Gegeben sei folgender Code Ausschnitt

$$\frac{1}{2}+\frac{3}{4}=\frac{4+6}{8}$$

```
class Fraction {
  int x, y;
  Fraction add(Fraction b) {
     return new Fraction(x * b.y + b.x * y, y * b.y);
  }
  …
}
Fraction c1 = new Fraction (1,2);
Fraction c2 = new Fraction (3,4);
Fraction c3 = c1.add(c2);
```

geht es nicht eleganter?

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);
   }
}
```

- Verwendung

```
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 Ergebnis liefern

Übung

- a) Definieren und implementieren Sie den Multiplikationsoperator für Fraction.
- b) Definieren und implementieren Sie einen Indexer, so dass der 0-te Index dem Zähler und der 1-te Index dem Nenner entspricht.

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

public int this[int i] {
  get {
        if (i == 0) return  this.x; else return this.y;
    }
   set {
         if (i== 0) this.x = value; else this.y = value;
    }
}
```

Beispiel (Überladen der Operatoren == und !=)

```
struct Fraction {
  int x, y;
  public Fraction(int x, int y) { this.x = x; this.y = y; }
  ...
  public static bool operator == (Fraction a, Fraction b) {
      return a.x == b.x && a.y == b.y; }
  public static bool operator != (Fraction a, Fraction b) { return ! (a == b); }
  public override bool Equals (Object o) {
       return (this == (Fraction)o);
   }
}

class Client {
  static void Main() {
      Fraction a = new Fraction(1, 2);
      Fraction b = new Fraction(1, 2);
      Fraction c = new Fraction(3, 4);
      Console.WriteLine(a == b);       // true
      Console.WriteLine((object)a == (object)b);      // false
      Console.WriteLine(a.Equals(b)); // true, da in Fraction überschrieben
  }
}
```

- Wenn == (<, <=, true) überladen wird, dann sollten auch != und Equals (>=, false) überladen werden.

Wahrheitstabelle 3-wertige (ternäre) Logik

- Neben der Booleschen (zweiwertigen) Logik gibt es auch 3 (oder mehr) wertige Logiken
- Auch für solche Logiken lassen sich Wahrheitstabellen und Operationen definieren

x	y	x&&y	x||y
true	true	true	true
true	false	false	true
true	undecided	undecided	true
false	true	false	true
false	false	false	false
false	undecided	false	undecided
undecided	true	undecided	true
undecided	false	false	undecided
undecided	undecided	undecided	undecided

es geht noch komplizierter

Überladen von && und ||

- Um && und || zu überladen, muss man &, |, true und false überladen

```
struct 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.state == -1 || y.state == -1) return new TriState(-1);
      else if (x.state == 1 && y.state == 1) return new TriState(1);
      else return new TriState(0);
   }
   public static TriState operator | (TriState x, TriState y) {
      if (x.state == 1 || y.state == 1) return new TriState(1);
      if (x.state == -1 && y.state == -1) return new TriState(-1);
      else return new TriState(0);
   }
}
```

- 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)
```

Kurzschlussauswertung

Überladen von Konversionsoperatoren

School of Engineering

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; }
}
```

int --> Fraction

cast nicht notwendig

cast notwendig

Verwendung

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

Übung

- Definieren Sie den impliziten und expliziten Konversationsoperator von TriState zu boolean. Beim Expliziten sollen alle Zustände ausser 1 zu false abgebildet werden.

```
public static implicit operator TriState (bool x) {
        return new TriState((x)?1:-1);
}

public static explicit operator bool (TriState f) {
        return f.state > 0;
}

TriState t = true;
bool b = (bool)t;
```

Klassen 2.
Erweiterte Konzepte

Vererbungsgrundkonzepte wie in Java

School of Engineering

```
class A {           // Oberklasse
  int a;
  public A() {...}
  public void F() {...}
}

class B : A {       // Unterklasse (erbt von A, erweitert A)
  int b;
  public B() {...}
  public void G() {...}
}
```

in Java extends

- *B* erbt *a* und *F()*, fügt *b* und *G()* hinzu
  - aber Konstruktoren werden nicht vererbt
- Klassen
  - können nur von ***einer Klasse*** erben, aber ***mehrere Interfaces*** implementieren.
  - können aber nicht von einem Struct erben
  - sind direkt oder indirekt von *object* abgeleitet
- Structs (sind eingeschränkt)

wie in Java !

Vererbung

Quiz

Gegeben sei folgendes C# Code Fragment

```
class A {
     public void WhoAreYou() { Console.WriteLine("I am an A"); }
}

class B : A {
     public void WhoAreYou() { Console.WriteLine("I am a B"); }
}
… …
A a = new B();
a.WhoAreYou();
```

was wird ausgegeben?

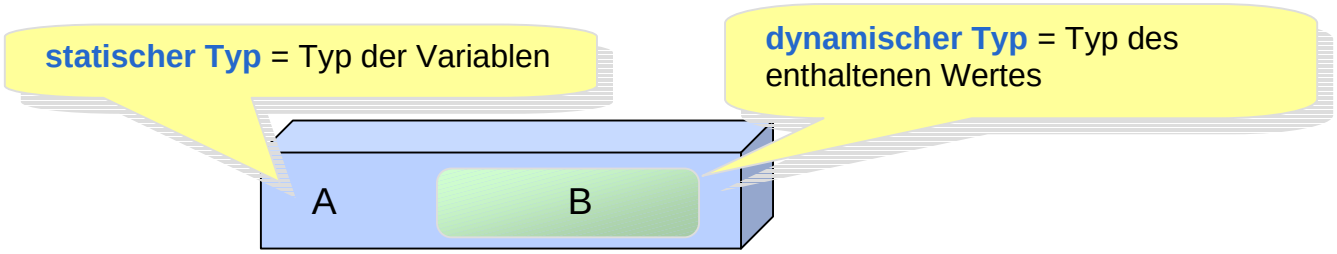
Überschreiben von Methoden

```
class A {
   public virtual void WhoAreYou() { Console.WriteLine("I am an A"); }
}

class B : A {
   public override void WhoAreYou() { Console.WriteLine("I am a B"); }
}
```

Ab Java 5 optional @Override

Es wird die Methode des dynamischen Typs aufgerufen

```
A a = new B();
a.WhoAreYou();      // "I am a B"
```



… Überschreiben von Methoden

School of Engineering

- Zweck: das Verhalten von Klassen soll spezialisiert/geändert werden
  - z.B. eine (abstrakte) Methode soll durch spezifische, neue ersetzt werden
- Ein grundlegendes OO Konzept
  - mittels Überschreiben kann die Funktionalität bestehender Klassen geändert werden ohne das Kapselungsprinzip zu verletzen,d.h. ohne die Implementation der bestehenden Klasse offenlegen zu müssen
- Überschreibende Methoden müssen dieselbe Schnittstelle haben wie überschriebene Methoden:
  - gleiche Parameteranzahl und Parametertypen (einschliesslich Funktionstyp)
  - gleiches Sichtbarkeitsattribut (public, protected, ...).
- Auch Properties und Indexers können überschrieben werden (virtual, override).
- Statische Methoden können nicht überschrieben werden -> auch keine Operatoren

… Überschreiben von Methoden

Überschreibbare Methoden müssen *zwingend* als virtual deklariert werden

```
class A {
   public void F() {...} // nicht überschreibbar
   public virtual void G() {...}   // überschreibbar
}
```

Überschreibende Methoden müssen als override deklariert werden, sonst Warnung

mittels base kann auf die Basisklassen Implementation zugegriffen werden

```
class B : A {
   public void F() {...} // Warnung: verdeckt geerbtes F() => new verwenden
   public void G() {...} // Warnung: verdeckt geerbtes G() => new verwenden
   public override void G() {// korrekt: überschreibt geerbtes G
      ... base.G();          // ruft geerbtes G() auf
   }
}
```

Verdecken von Methoden

Methoden können in Unterklasse mit new deklariert werden. Dadurch verdecken sie gleichnamige geerbte Methoden.

```
class A {
   public int x;
   public void F() {...}
   public virtual void G() {...}
}
class B : A {
   public new int x;
   public new void F() {...}
   public new void G() {...}
}
B b = new B();
b.x = ...;          // spricht B.x an
b.F(); ... b.G();   // ruft B.F und B.G auf
A a = new B();
a.F()               // ruft A.F
```

New ist der Default!; es wird aber je nach Compiler Einstellungen eine Warnung erzeugt

Zweck: Neue Funktionalität, die aber von bestehendem Code nicht berücksichtigt wird

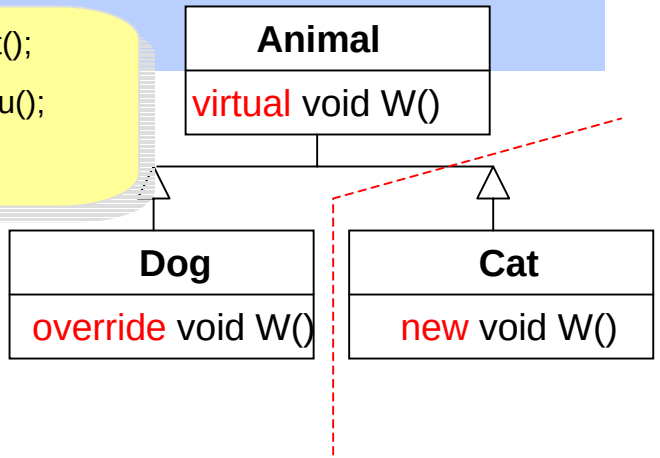
Verdecken vs Überschreiben

Funktioniert in einfachen Fällen wie gewohnt

```
class Animal {
   public virtual void WhoAreYou() { Console.WriteLine("I am an animal"); }
}
class Dog : Animal {
   public override void WhoAreYou() { Console.WriteLine("I am a dog"); }
}

class Cat : Animal {
   public new void WhoAreYou() { Console.WriteLine("I am a cat"); }
}
```

```
Animal pet = new Dog();
pet.WhoAreYou();  // "I am a dog
Animal pet = new Cat();
pet.WhoAreYou();  // "I am an anim
Cat cat = new Cat();
cat.WhoAreYou();  // "I am a cat"
```

Animal pet = new Cat();
((Cat)pet).WhoAreYou();
// I am a cat



Lernkontrolle

School of Engineering

```
class Animal {
   public virtual void WhoAreYou() { Console.WriteLine("I am an animal"); }
}
class Dog : Animal {
   public override void WhoAreYou() { Console.WriteLine("I am a dog"); }
}
class Beagle : Dog {
   public new virtual void WhoAreYou() { Console.WriteLine("I am a beagle"); }
}
class AmericanBeagle : Beagle {
   public override void WhoAreYou() { Console.WriteLine("I am an american beagle"); }
}
```

```
Animal animal = new AmericanBeagle();
Beagle beagle = new AmericanBeagle();
beagle.WhoAreYou();
// "I am an american beagle"
animal.WhoAreYou();
// "I am a dog" !!
((AmericanBeagle)animal).WhoAreYou();
// "I am an american beagle"
((Beagle)animal).WhoAreYou();
// "I am an american beagle"
((Dog)beagle).WhoAreYou()
// "I am a dog"
((Animal)beagle).WhoAreYou();
// "I am a dog";
```

Wieso so kompliziert ?

Ausgangssituation

1
```
class LibraryClass {
   public void CleanUp() { ... }
}
```

2
```
class MyClass : LibraryClass {
   public void Delete() { ... erase the hard disk ... }
}
```

Später: Hersteller liefert neue Version von *LibraryCla*

3
```
class LibraryClass {
   string name;
   public virtual void Delete() { name = null; }
   public void CleanUp() { Delete(); ... }
}
```

```
class MyClass : LibraryClass {
   public void Delete() { ... erase the hard disk
}
```

es wird die überladene Methode aufgerufen

In C# passiert gar nichts, solange MyClass nicht übersetzt wird. MyClass basiert noch auf alter Version von LibraryClass (Versioning) => altes CleanUp() ruft kein LibraryClass.Delete() auf.

Wird MyClass übersetzt, gibt es eine Warnung, dass Delete als new oder override deklariert werden sollte.

In Java würde Aufruf von *myObj.CleanUp()* nun das Löschen der Platte bedeuten.

"Korrekte" Änderung in Basisklasse führt einem Fehler -> **Fragile-Baseclass-Problem**

Konstruktoraufruf in Ober- und Unterklasse

School of Engineering

Impliziter Aufruf des Basisklassenkonstruktors			Expliziter Aufruf

class A {
  ...
}
 	
class B : A {
  public B(int x) {...}
}
	class A {
	  public A() {...}
	}
	
	class B : A {
	  public B(int x) {...}
	}
		class A {
		  public A(int x) {...}
		}
		
		class B : A {
		  public B(int x) {...}
		}
			class A {
			  public A(int x) {...}
			}
			
			class B : A {
			  public B(int x)
			  : base(x) {...}
			}

B b = new B(3);	B b = new B(3);	B b = new B(3);	B b = new B(3);
OK - Default-Konstr. A() - B(int x)	**OK** - A() - B(int x)	**Compilefehler!** - kein expliz. Aufruf des A-Konstruktors - Default-Konstr. A() existiert nicht	**OK** - A(int x) - B(int x)

Abstrakte Klassen

Abstrakte Klassen

zh aw School of Engineering

Beispiel

```
abstract class Stream {
   public abstract void Write(char ch);
   public void WriteString(string s) { foreach (char ch in s) Write(s); }
}

class File : Stream {
   public override void Write(char ch) {... write ch to disk ...}
}
```

Bemerkungen

- Abstrakte Methoden haben keinen Anweisungsteil (Implementation)
- Abstrakte Methoden sind implizit *virtual*.
- Wenn eine Klasse abstrakte Methoden enthält (d.h. deklariert oder erbt und nicht überschreibt), muss sie selbst *abstract* deklariert werden.
- Von abstrakte Klassen lassen sich keine Instanzen erzeugen

Abstrakte Properties und Indexers

Beispiel

```
abstract class Sequence {
   public abstract void Add(object x);  // Methode
   public abstract string Name { get; } // Property
   public abstract object this [int i] { get; set; }      // Indexer
}

class List : Sequence {
   public override void Add(object x) {...}
   public override string Name { get {...} }
   public override object this [int i] { get {...} set {...} }
}
```

Bemerkungen

- Indexers und Properties sind beim Überschreiben hinsichtlich get und set gleich

Versiegelte Klassen (sealed)

Beispiel

in Java final

```
sealed class Account : Asset {
   long val;
   public void Deposit (long x) { ... }
   public sealed override void Withdraw (long x) { ... }
   ...
}
```

Bemerkungen

- *von sealed*-Klassen kann nicht geerbt werden (entspricht *final* in Java), können aber erben.
- Methoden können auch einzeln als *sealed* deklariert werden
- Für nicht geerbte Methoden einfach das *virtual* weglassen (selber Effekt).
- Zweck:
  - Sicherheit: verhindert versehentliches/mutwilliges Erweitern der Klasse

Interfaces

Syntax

Syntaktisch keine Unterscheidung zwischen extends und implements

```
public interface IList : ICollection, IEnumerable {
  int Add (object value);   // Methoden
  bool Contains (object value);
  ...
  bool IsReadOnly { get; }  // Property
  ...
  object                    this [int index] { get; set; }        // Indexer
}
```

in C# beginnen alle Interfaces

nach Konvention mit einem grossen 'I'

- Interface = rein abstrakte Klasse: nur Schnittstelle, kein Code.
- Darf nur Methoden, Properties, Indexers und Events enthalten (keine Felder, Konstanten, Konstruktoren, Destruktoren, Operatoren, innere Typen).
- Interface-Members sind implizit *public abstract* (*virtual*).
- Interface-Members dürfen nicht *static* sein.
- Klassen und Structs können mehrere Interfaces implementieren.
- Interfaces können andere Interfaces erweitern.

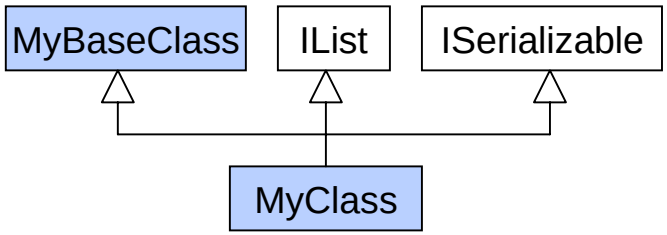
Implementierung eines Interface

School of Engineering

```
class MyClass : MyBaseClass, IList, ISerializable {
   public int Add (object value) {...}
   public bool Contains (object value) {...}
   ...
   public bool IsReadOnly { get {...} }
   ...
   public object this [int index] { get {...} set {...} }
}
```

- Eine Klasse kann von *einer* Klasse erben, aber *beliebig viele* Interfaces implementieren.
- Jede geerbte Interface-Methode (Property, Indexer) *muss implementiert* oder von einer anderen Klasse *geerbt* werden.
- Beim Überschreiben von Interface-Methoden muss man kein *override* angeben, ausser man überschreibt eine von einer Klasse geerbte Methode.

Benutzung von Interfaces



Zuweisungen: MyClass c = new MyClass();

IList list = c;

Methodenaufrufe: list.Add("Tom"); // dyn.Bindung => MyClass.Add

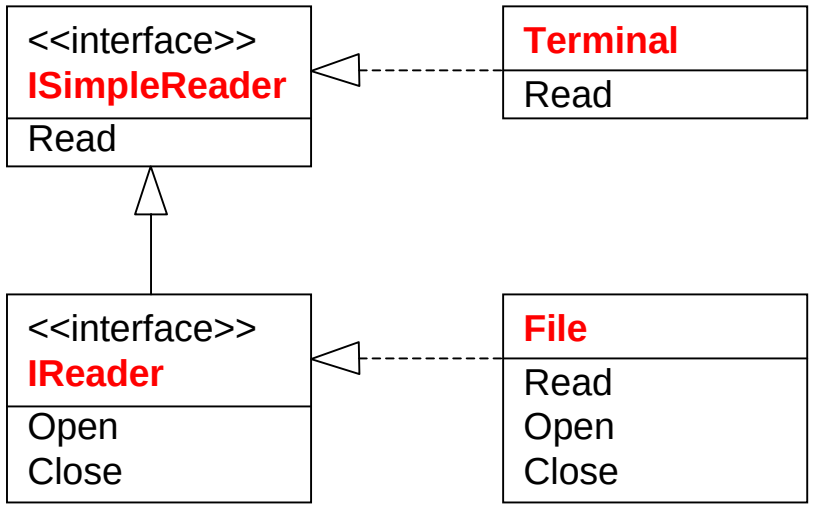
Typumwandlungen: c = (MyClass) list;

ISerializable ser = (ISerializable) list;

Typprüfungen: if (list is MyClass) ... // true

Beispiel

```
interface ISimpleReader {
  int Read();
}

interface IReader : ISimpleReader {
  void Open(string name);
  void Close();
}

class Terminal : ISimpleReader {
  public int Read() { ... }
}

class File : IReader {
  public int Read() { ... }
  public void Open(string name) { ... }
  public void Close() { ... }
}

ISimpleReader sr = null;      // Zuweisung von null erlaubt
sr = new Terminal();
sr = new File();

IReader r = new File();
sr = r;
```



Name Clashes

Wenn zwei geerbte Interfaces eine gleichnamige Methode enthalten

```
interface I1 {
   void F();
}
interface I2 {
   void F();
}
class B : I1, I2 {
   //----- Implementierung durch eine einzige F-Methode
   public void F() { Console.WriteLine("B.F"); }
   //----- Implementierung durch getrennte F-Methoden
   void I1.F() { Console.WriteLine("I1.F"); }  // darf nicht public sein !?!
   void I2.F() { Console.WriteLine("I2.F"); }  // -- " --
}
```

```
B b = new B();
b.F();       // B.F
I1 i1 = b;
i1.F();      // I1.F
I2 i2 = b;
i2.F();      // I2.F
```

Operationen auf Klassen, Typen

Zuweisungen und Typprüfungen

School of Engineering

class A {...}

class B : A {...}

class C: B {...}

Zuweisungen

```
A a = new A(); // statischer Typ von a ist immer A,
               // dynamischer Typ ist hier auch A
a = new B();   // dyn.Typ(a) == B
a = new C();   // dyn.Typ(a) == C
B b = a;            // verboten; Compilefehler
```

Dynamische Typprüfungen zur Laufzeit: is

```
a = new C();
if (a is C) ...     // true, wenn dyn.Typ(a) C oder Unterklasse ist;
if (a is B) ...     // true
if (a is A) ...     // true, aber Warnung, weil sinnloser Test
a = null;
if (a is C) ...     // wenn a == null, liefert a is T immer false
```

Java instanceof

Geprüfte Typumwandlungen

School of Engineering

Typumwandlung mit Cast

```
A a = new C();
B b = (B) a;// if (a is B) stat.Typ(a) wird in diesem Ausdruck zu B;
            // else Laufzeitfehler
C c = (C) a;
a = null;
c = (C) a;  // ok; => null lässt sich in jeden Referenztyp konvertieren
```

Typumwandlung mit as

```
A a = new C();
B b = a as B;       // if (a is B) b = (B)a; else b = null;
C c = a as C;
a = null;
c = a as C; // c == null
```

Der Type-Typ

Einstiegspunkt
für die Reflection

- **typeof**
- liefert *Type*-Objekt zu einer z.B. Klasse

```
Type t = typeof(int);
Console.WriteLine(t.Name);     // liefert Int32
```

- **getType()**
- liefert Type-Objekt zu einem Objekt (i.e. Instanz)

```
int k = 3;
Type t = k.getType();
Console.WriteLine(t.Name);     // liefert Int32
```

Klasse object (System.Object)

School of Engineering

Oberste Wurzelklasse aller Klassen

```
class Object {
   protected object MemberwiseClone() {...}
   public Type GetType() {...}
   public virtual bool Equals (object o) {...}
   public virtual string ToString() {...}
   public virtual int GetHashCode() {...}
}
```

Direkt zu verwenden:

```
Type t = x.GetType();          liefert Typbeschreibung (für Reflection)
object copy = x.MemberwiseClone(); führt Shallow Copy durch (aber protected!)
```

In Unterklassen zu überschreiben:

```
x.Equals(y)                    Wertvergleich durchführen
x.ToString()                   String-Darstellung des Objekts liefern
int code = x.getHashCode();    möglichst eindeutigen Code liefern
```

Verwendung von object

Wird als Standardtyp *object* benutzt

```
object obj;
```

Zuweisungskompatibilität

```
obj = new Rectangle();
obj = new int[3];
```

Parameter Typ in pseudo-generischen Klassen

```
void Push(object x) {...}
Push(new Rectangle());
Push(new int[3]);
```

Boxing und Unboxing

1.5

School of Engineering

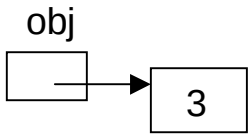
Umwandlung von Wertetypen zu Referenztypen - und zurück

Boxing

Bei der Zuweisung

```
object obj = 3;
```

wird der Wert 3 in ein Heap-Objekt eingepackt




```
class TempInt {
    int val;
    TempInt(int x) {val = x;}
}

obj = new TempInt(3);
```

Unboxing

Bei der Zuweisung

```
int x = (int) obj;
```

wird der eingepackte int-Wert wieder ausgepackt

Boxing/Unboxing

Erleichtert Verwendung pseudo-generischen Klassen

```
class Queue {
    ...
    public void Enqueue(object x) {...}
    public object Dequeue() {...}
    ...
}
```

Diese Queue kann für Referenz- und Werttypen verwendet werden

```
Queue q = new Queue();
q.Enqueue(new Rectangle());
q.Enqueue(3);
Rectangle r = (Rectangle) q.Dequeue();
int x = (int) q.Dequeue();
```

es gibt aber auch echte Generizität (später)

Freigabe von Objekten/Ressourcen

- Muss nicht explizit gemacht werden
- Sog. Garbage Collector entfernt nicht mehr referenzierte Objekte automatisch
- G.C. als niedrig priorisierter Thread: läuft u.U. erst verzögert
- Nicht mehr referenziert: keine versteckten Referenzen

```
class A {
       public A() {...}     // Konstuktor
       public ~A() {...}    // Destructor
}
```

- Heikel wenn Objekt eine Wrapperklasse für z.B. Betriebssystemressource

IDisposable und Using

- Klasse implementier IDisposable somit Dispose Methode

```
public class SomeDisposableType : IDisposable
{
    ...implmentation details...
}
```

- Dispose ist die Aufräummethode

```
SomeDisposableType t = new SomeDisposableType();
try {
    OperateOnType(t);
}
finally {
   t.Dispose();
}
```

- Abgekürzte Schreibweise mit using

```
using (SomeDisposableType u = new SomeDisposableType()) {
    OperateOnType(u);
}
```

Beispiel SqlConnection

- Sicherstellung der Freigabe durch finally Block

```
SqlConnection connection = new SqlConnection(connectionString);
try
{
    connection.Open();
    ret = command.ExecuteScalar();
}
finally
{
    if (connection != null) connection.Close();
}
```

- Automatische Freigabe am Ende des Blocks

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    ...
    connection.Open();
    object ret = command.ExecuteScalar();
}
```

Einfache I/O Operationen

Konsolen-Ausgabe

Beispiele

```
Console.Write(intVal);          // überladen für alle Standardtypen;
Console.WriteLine(intVal);      // bei Objekten wird automatisch ToString()
aufgerufen

Console.Write("Hello {0}", name);  // Platzhalter

Console.WriteLine("{0} = {1}", x, y);
```

Platzhalter-Syntax

"{" n ["," width] [":" format [precision]] "}"

n	Argumentnummer (beginnend bei 0)
width	Feldbreite (wenn zu klein, wird sie überschritten) positiv = rechtsbündig, negativ = linksbündig
format	Formatierungscode (z.B. d, f, e, x, ...)
precision	Anzahl der Nachkommastellen (machmal Anzahl der Ziffern) B

Beispiel: **{0,10:f2}**

Formatierungscode für Zahlen

d, D	Dezimalformat (ganze Zahl mit führenden Nullen)	-xxxxx
	precision = Anz. Ziffern	
f, F	Fixpunktformat	-xxxxx.xx
	precision = Anz. Nachkommastellen (Default = 2)	
n, N	Nummernformat (mit Tausender-Trennstrich)	-xx,xxx.xx
	precision = Anz. Nachkommastellen (Default = 2)	
e, E	Exponentialformat (groß/klein signifikant)	-x.xxxE+xxx
	precision = Anz.Nachkommastellen	
c, C	Currency-Format	$xx,xxx.xx
	precision = Anz. Nachkommastellen (Default = 2)	
	Negative Werte werden in Klammern gesetzt	($xx,xxx.xx)
x, X	Hexadezimalformat (groß/klein signifikant)	xxx
	precision = Anzahl Hex-Ziffern (evtl. führende 0)	
g, G	General (kompaktestes Format für gegebenen Wert; Default)	

Beispiele

```
int x = 17;

Console.WriteLine("{0}", x);       17
Console.WriteLine("{0,5}", x);        17

Console.WriteLine("{0:d}", x);     17
Console.WriteLine("{0:d5}", x);    00017

Console.WriteLine("{0:f}", x);     17.00
Console.WriteLine("{0:f1}", x);    17.0

Console.WriteLine("{0:E}", x);     1.700000E+001
Console.WriteLine("{0:E1}", x);    1.7E+001

Console.WriteLine("{0:x}", x);     11
Console.WriteLine("{0:x4}", x);    0011
```

Allgemeine Stringformatierung

Mittels *ToString*, für numerische Standardtypen (*int*, *long*, *short*, ...):

```
string s;
int i = 12;
s = i.ToString();      // "12"
s = i.ToString("x4");  // "000c"
s = i.ToString("f");   // "12.00"
```

Mittels *String.Format*, für beliebige Typen

```
s = String.Format("{0} = {1,6:x4}", name, i);      // "val =    000c"
```

Länderspezifische Formatierung System.Globalization

```
s = i.ToString("c");         // "$12.00"
s = i.ToString("c", new CultureInfo("en-GB"));    // "£12.00"
s = i.ToString("c", new CultureInfo("de-CH"));    // "sFr. 12.00"
```

Lesen der Kommandozeilenparameter

```
using System;

class Test {

  static void Main(string[] arg) {// Aufruf z.B.: Test value = 3
    for (int i = 0; i < arg.Length; i++)
      Console.WriteLine("{0}: {1}", i, arg[i]); // Ausgabe (Tokentrennung bei
Leerzeichen):
                                    //  0: value
                                    //  1: =
                                    //  2: 3
    foreach (string s in arg)
      Console.WriteLine(s);         // Ähnliche Ausgabe wie oben
  }
}
```

Tastatur-Eingabe

int ch = Console.Read();

Liefert nächstes Zeichen.

Wartet, bis Benutzer Zeile mit Return-Taste abgeschlossen hat.
Z.B. Eingabe: "abc" + Return-Taste.
Read liefert der Reihe nach: 'a', 'b', 'c', '\r', '\n'.
Nach letztem Zeichen (Strg-Z + Return) liefert Read() den Wert -1

string line = Console.ReadLine();

Liefert nächste Zeile (nach Strg-Z+CR+LF liefert es null).

Wartet, bis Benutzer Zeile mit CR,LF abgeschlossen hat.

Liefert dann diese Zeile ohne CR, LF.

Es gibt keinen Tokenizer aber **Split** Methode ist in Strings definiert

Formatierte Ausgabe auf Datei

```
using System;
using System.IO;

class Test {

   static void Main() {
      FileStream s = new FileStream("output.txt", FileMode.Create);
      StreamWriter w = new StreamWriter(s);
      w.WriteLine("Table of sqares:");
      for (int i = 0; i < 10; i++)
         w.WriteLine("{0,3}: {1,5}", i, i*i);
      w.Close();
   }
}
```

- Trennung zwischen Formatierung und Medium
- StreamWriter wird auf einen Stream (File/Netzwerk) aufgesetzt

Lesen von einer Datei

```
using System;
using System.IO;

class Test {
   static void Main() {
      FileStream s = new FileStream("input.txt", FileMode.Open);
      StreamReader r = new StreamReader(s);
      string line = r.ReadLine();
      while (line != null) {
         ...
         line = r.ReadLine();
      }
      r.Close();
   }
}
```

- *Trennung zwischen Formatierung und Medium*
- *StreamReader* wird auf einen Stream (File/Netzwerk) aufgesetzt

Zusammenfassung

- **Properties**
  - ersetzt die set-get Konvention von Properties in Java durch Sprachkonstrukt
- **Operatoren, Indexer und Konverter**
  - können in C# überladen werden (wie in C++)
- **Vererbung expliziter als in Java**
  - **virtual** überschreibare Methoden
  - **override** überschreiben der Methode
  - **new** erzeugen einer neuen Methode
- **Interfaces**
  - wie Java, beginnen mit 'I‘
- **Einfache I/O**
  - System.Console, File I/O

Fragen?