Vererbung

Syntax

```
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() {...}
}
```

- *B* erbt *a* und *F()*, fügt *b* und *G()* hinzu
  - Konstruktoren werden nicht vererbt
  - Überschreiben siehe später
- Eine Klasse kann nur von *einer* Klasse erben, aber mehrere Interfaces implementieren.
- Eine Klasse kann nur von einer *Klasse* erben, aber nicht von einem Struct.
- Structs können nicht erben, sie können aber Interfaces implementieren.
- Alle Klassen sind direkt oder indirekt von *object* abgeleitet. Structs sind über Boxing mit *object* kompatibel.

Zuweisungen und Typprüfungen


```
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
```

Typprüfung zur Laufzeit

```
a = new C();
if (a is C) ...     // true, wenn dyn.Typ(a) C oder Unterklasse ist; sonst false
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
```

Geprüfte Typumwandlungen

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
```

Überschreiben von Methoden

Überschreibbare Methoden müssen als virtual deklariert werden

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

Überschreibende Methoden müssen als override deklariert 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
    }
}
```

- Überschreibende Meth. müssen dieselbe Schnittstelle haben wie überschriebene Meth.:
  - 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.

Dynamische Bindung (vereinfacht)


```
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"); }
}
```

Es wird die Methode des dynamischen Typs des Empfängers aufgerufen
(stimmt nicht ganz, siehe später)

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

Zweck: Jede Methode, die mit *A* arbeiten kann, kann auch mit *B* arbeiten

```
void Use (A x) {
    x.WhoAreYou();
}

Use(new A());       // "I am an A"
Use(new B());       // "I am a B"
```

Verdecken von Members

Members können in Unterklasse mit new deklariert werden.

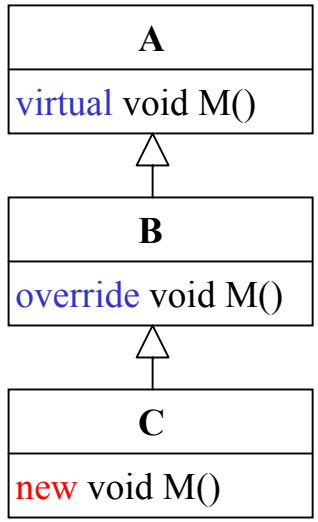
Dadurch verdecken sie gleichnamige geerbte Members.

```
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)b).x = ...;             // spricht A.x an !
((A)b).F(); ... ((A)b).G(); // ruft A.F und A.G auf !
```

Dynamische Bindung (mit Verdecken)

Methodensuche bei obj.M()

```
st = static type of obj;
dt = dynamic type of obj;
m = Method "M" of st;
for (all types t between st (exclusive) and dt (inclusive)) {
    if (t has an override method "M") m = "M" of t;
    else if (t has a non-override method "M") break;
}
call m;
```



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"); }
}

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

Komplexeres Beispiel


```
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"); }
}

Beagle pet = new AmericanBeagle();
pet.WhoAreYou();                    // "I am an american beagle"

Animal pet = new AmericanBeagle();
pet.WhoAreYou();                    // "I am a dog" !!
```

Fragile-Baseclass-Problem

Ausgangssituation

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

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

Später: Hersteller liefert neue Version von LibraryClass

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

- In Java würde Aufruf von *myObj.CleanUp()* nun das Löschen der Platte bedeuten.
- In C# passiert gar nichts, solange *MyClass* nicht übersetzt wird.
  MyClass basiert noch auf alterVersion 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.

Konstruktoren in Ober- und Unterklasse

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)

Sichtbarkeit protected und internal

protected	Sichtbar in deklarierender Klasse und ihren Unterklassen (restriktiver als in Java)
internal	Sichtbar im deklarierenden Assembly (s.später)
protected internal	Sichtbar in deklarierender Klasse, ihren Unterklassen und im deklarierenden Assembly

Beispiel

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

Abstrakte Klassen

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.
- 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 abstrakten Klassen kann man keine Objekte 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 müssen beim Überschreiben hinsichtlich get und set gleich sein

Versiegelte Klassen (sealed)

Beispiel

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

Bemerkungen

- *sealed*-Klassen können nicht erweitert werden (entspricht *final* in Java), können aber selbst Unterklassen sein.
- *override*-Methoden können auch einzeln als *sealed* deklariert werden.
- Zweck:
  - Sicherheit (verhindert versehentliches Erweitern der Klasse)
  - Methoden können u.U. statisch gebunden aufgerufen werden

Klasse object (System.Object)

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

Beispiele *(Verwendung von object)*


```
class Fraction {
    int x, y;
    public Fraction(int x, int y) { this.x = x; this.y = y; }
    ...
    public override string ToString() { return String.Format("{0}/{1}", x, y); }
    public override bool Equals(object o) { Fraction f = (Fraction)o; return f.x == x && f.y == y; }
    public override int GetHashCode() { return x ^ y; }
    public Fraction ShallowCopy() { return (Fraction) MemberwiseClone(); }
}
```

```
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.ToString());        // 1/2
        Console.WriteLine(a);                   // 1/2 (ToString wird automatisch aufgerufen)

        Console.WriteLine(a.Equals(b));         // true
        Console.WriteLine(a == b);              // false

        Console.WriteLine(a.GetHashCode());     // 3

        a = c.ShallowCopy();
        Console.WriteLine(a);                   // 3/4
    }
}
```

Beispiel (Überladen der Operatoren == und !=)


```
class 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); }
}
```

```
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 == überladen wird, muß auch != überladen werden.
- Compiler erzeugt Warnung, wenn == und != überladen werden, aber *Equals* nicht überschrieben wird.