
03 – OO-Programmierung

Agenda

- Grundlagen der Objektorientierung
- Namensräume
- Klassenelemente
- Statische Klassenelemente
- Vererbung
- Konvertierung
- Abstrakte Klassen
- Interfaces

4 Konzepte

- **Abstraktion**
 - Trennung von Konzept (Klasse) und Umsetzung (Objekt)
- **Kapselung**
 - Zusammenfassung von Daten und dazugehöriger Funktionalität
- **Vererbung**
 - Die Möglichkeit der Spezialisierung und zur Erstellung einer Klassenhierarchie
- **Polymorphie**
 - Fähigkeit eines Objektes, eine Instanz einer von seiner Klasse abgeleiteten Klasse aufzunehmen

Abstraktion – Was ist eine Klasse?

- Eine Klasse ist gewissermassen ein „Bauplan für Objekte“
- Eine Klasse ist Vorlage (Prägestempel) für Objekte und definiert:
 - Informationen (Felder/Attribute und Eigenschaften/Properties)
 - Konstruktor(en)
 - Methoden (Verhalten)
 - Ereignisse

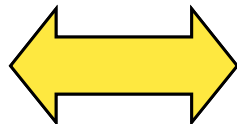
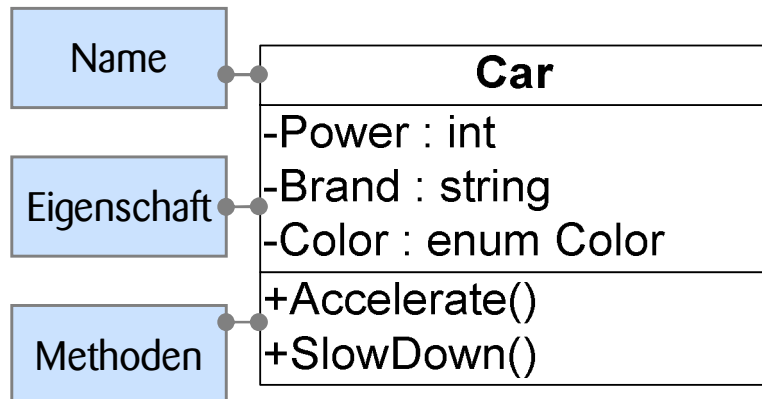
Abstraktion – Was ist ein Objekt?

- Sammlung von zusammengehörenden Informationen und deren Verhalten
- Meist Repräsentation einer Entität aus:
 - der realen Welt (Mitarbeiter, Flugzeug, Produkt)
 - einer virtuellen Welt (Window, Port, HttpResponse)
 - einer Abstraktion (Liste von offenen Tasks/Aufgaben)
- Ist mit Daten versehen

Grundlagen der Objektorientierung

Abstraktion - Verschiedene Sichten auf ein „Ding“

Programmierer-Sicht



Real-World-Sicht



C# Coding Guide

Klassenname beginnt mit Grossbuchstabe

Kapselung

- Zusammenfassen, was zusammengehört
 - Daten und Funktionalität
- Verhindern, dass interne Daten allgemeingültig sind
 - Interne Daten und Funktionalität schützen
 - Nur öffentliche Daten und Funktionalität publizieren
- Kapselung wird in .NET realisiert durch
 - Felder/Instanzvariablen
 - Eigenschaften (Properties)
 - Methoden
 - Ereignisse
- Wiederverwendbarkeit
 - Durch einfache Interfaces kann die Wiederverwendung der Klasse besser erreicht werden

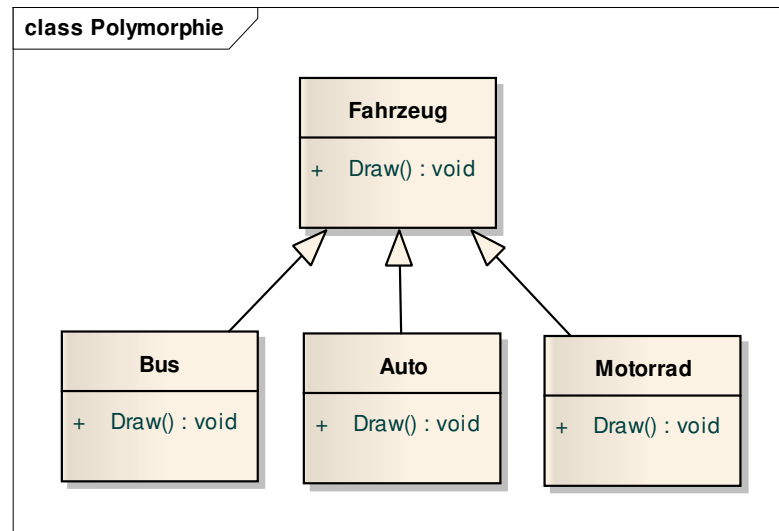
Grundlagen der Objektorientierung

Vererbung

- Eine Klasse kann von einer anderen Klasse abgeleitet werden
- Durch Vererbung kann eine Spezialisierung erreicht werden
- Die abgeleitete Klasse erbt die privaten Daten und Methoden der Vaterklasse nicht

Grundlagen der Objektorientierung

Polymorphie



Namensräume definieren

```
namespace AutoZubehoer
{
    class Spoiler
    {
        ...
    }
    class Felge
    {
        ...
    }
}
```

Namensräume

Warum braucht es Namensräume?

```
namespace AutoZubehoerIPS
{
    public class Spoiler
    { ... }
}
```

```
namespace AutoZubehoerRemus
{
    public class Spoiler
    { ... }
}
```

Unterscheidung bei gleichen Klassennamen

```
class AutoTuning
{
    AutoZubehoerIPS.Spoiler spoilerA;
    AutoZubehoerRemus.Spoiler spoilerB;
}
```

Namensräume anwenden – using

■ Ohne using: lange Schreibweise

```
class AutoTuning
{
    AutoZubehoerIPS.Spoiler spoilerA;
}
```

■ Mit using: kurze Schreibweise

```
using AutoZubehoerIPS;
class AutoTuning
{
    Spoiler spoilerA;
}
```

Verschachtelte Namensräume definieren

■ Übliche Definitionsform

```
namespace Microsoft
{
    namespace Office
    {
        ...
    }
}
```

■ Kurze Definitionsform

```
namespace Microsoft.Office
{
    ...
}
```

Attribute (Felder)

- Die Informationen eines Objekts werden in Attribute gespeichert
- Attribute sind Variablen, welche zu einem bestimmten Objekt “gehören”
- Jedes Attribut hat einen Typ und einen Namen

```
class Mitarbeiter
{
    public string Name;
    private Abteilung abteilung;

    private int batchNr;
    ...
}
```



C# Coding Guide

Public Attributnamen
beginnen mit
Grossbuchstabe,
private mit
Kleinbuchstabe
(nicht mit Unterstrich
«_»)

Schreibgeschützte Felder (Attribute)

- Felder können mit dem Modifikator `readonly` als nur-les Feld definiert werden
- Beispiel:

```
public class Mitarbeiter
{
    public readonly string Firma = „Software Factory“;
    ...
}
```

- Unterschiede zu Konstanten:
 - Feldzugriff via Objektreferenz (Konstantenzugriff via Klassenname)
 - Kann beim Instanziiieren im Konstruktor mit Wert initialisiert werden, nicht aber in Methoden



C# Coding Guide

Public Feldnamen
beginnen mit
Grossbuchstabe,
private mit
Kleinbuchstabe

Eigenschaften (Properties)

- Eigenschaften sind wie öffentliche Felder
- Die Werte der Eigenschaften werden in Feldern gespeichert
- Eigenschaften besitzen Zugriffsmethoden (get / set)

```
class Mitarbeiter {  
    private string name;  
    public string Name  
    {  
        get { return name; }  
        set { name = value; }  
    }  
}
```

- Automatisch implementierte Eigenschaften (ab .NET 3.5):

```
public string Name {get; set;}
```



C# Coding Guide

Public Eigenschaften
beginnen mit
Grossbuchstabe,
private mit
Kleinbuchstabe

Schreibgeschützte Eigenschaften (Properties)

- Nur die `get`-Methode wird implementiert
- Beispiel:

```
class Mitarbeiter
{
    private string name;
    public string Name
    {
        get // Get Methode
        {
            return name;
        }
    }
    ...
}
```

Lesegeschützte Eigenschaften (Properties)

- Nur die `set`-Methode wird implementiert
- Beispiel:

```
class Mitarbeiter
{
    private string name;
    public string Name
    {
        set // Set Methode
        {
            name = value;
        }
    }
    ...
}
```

Konstruktor (1/3)

- Der Konstruktor erzeugt eine Instanz (Objekt) der Klasse
- Ein Konstruktor dient zur Initialisierung des Objekts
- Der erforderliche Speicher für die Daten des Objektes wird auf dem Heap-Speicher reserviert
- Beispiel eines Konstruktors:

```
class Mitarbeiter
{
    public Mitarbeiter()
    {
        // Konstruktor-Implementierung
    }
}
```

Konstruktor (2/3)

- Beispiel eines Konstruktors mit Parameterübergabe:

```
class Mitarbeiter
{
    private string name, vorname; // Private Variablen

    public Mitarbeiter(string name, string vorname)
    {
        // Via Konstruktor initialisieren
        this.name = name;
        this.vorname = vorname;
    }
}
```

Konstruktor (3/3)

- `public` deklarierte Konstruktoren
 - Der Konstruktor kann von allen Benutzern der Klasse aufgerufen werden.
- `internal` deklarierte Konstruktoren
 - Die Verwendung des Konstruktors der Klasse ist auf die aktuelle Assembly eingeschränkt.
- `private` deklarierte Konstruktoren
 - Wird verwendet, wenn eine Klasse nicht instanziiert werden darf. Z.B. Klassen, welche nur statische Methoden enthalten wie eine Klasse `MathLibrary`.

```
class MathLibrary
{
    private MathLibrary() { }
}
```

Objekte/Instanzen

- Objekte werden auch Instanzen einer Klasse genannt
- Ein Objekt erzeugen nennt man «instanziiieren»
- Von einer Klasse können beliebig viele Instanzen erzeugt werden
- Jedes Objekt hat einen eigenen reservierten Speicherbereich
- Die Methoden sind nur einmal für alle Instanzen vorhanden
- Die Methoden sind für alle Instanzen einer Klasse gleich
- Mit einer Referenz wird auf die Objekte zugegriffen

■ Syntax:

```
<Klasse> <Bezeichner> = new <Klasse>();
```

■ Beispiele:

```
Messpunkt anfangswert = new Messpunkt();  
Time goodTime         = new Time();
```

Klassendefinition aufteilen

- Ab .NET Framework 2.0 ist es möglich eine Klassendefinition mit `partial` in mehrere Dateien aufzuteilen.
- Wird beim Erstellen einer «Windows Form»-Applikation vom Form Designer verwendet.
- Beispiel:

```
partial class Auto {  
    ...  
}
```

Datei ClassAuto1.cs

```
partial class Auto {  
    ...  
}
```

Datei ClassAuto2.cs

Klassenelemente

Methoden deklarieren

```
class Rechteck
{
    private double a, b;
    public Rechteck(double s1, double s2) {
        a = s1;
        b = s2;
    }
    public double Flaeche() {
        return a * b;
    }
}
```

Methodendeklaration



Anwendungsbeispiel:

```
Rechteck r = new Rechteck(10, 12.3);
double flaeche = r.Flache();
```


Sichtbarkeit von Methoden und Attributen mit Sichtbarkeitsmodifikatoren setzen

- **private** Nur innerhalb der Klasse sichtbar (default)
- **protected** Innerhalb der Klasse und abgeleiteter Klassen sichtbar
- **internal** Innerhalb des Assemblies sichtbar
- **internal protected** Innerhalb des Assemblies und abgeleiteter Klassen sichtbar
- **public** Von ausserhalb sichtbar, keine Sichtbarkeitsbeschränkung

Beispielcode: Klassendefinition

```
class Kunde
{
    public int Id {get; set;}
    public string Vorname {get; set;};
    public string Nachname {get; set;};

    public void PrintVorname()
    {
        Console.WriteLine("Vorname: {0}", Vorname);
    }

    public void ReadVorname()
    {
        Vorname = Console.ReadLine();
    }
}
```

Übung 3.1



Erste Klasse in C# implementieren

- Entwickle eine Klasse “Mitarbeiter”, welche folgende Daten speichern kann:
 - Vorname
 - Nachname
 - Personalnummer
 - Eintrittsdatum
 - Geburtsdatum
 - Salär
 - Private Adresse
 - Telefonnummer
- Die Daten sollen mit einer Methode von der Tastatur eingelesen werden und in einer zweiten Methode ausgegeben werden können.

Destruktor

- Ein Destruktor beendet die Lebenszeit eines Objektes
- Der Destruktor wird nur implementiert, wenn er tatsächlich benötigt wird, um vor dem Zerstören des Objektes weitere Aktionen auszuführen, z.B. Protokollierung oder Ressourcenfreigabe

- Syntax:

```
~Klassenname () { ... }
```

- Ein Objekt wird zerstört, wenn
 - der Gültigkeitsbereich einer Objektreferenz verlassen wird
 - einer Objektreferenz der Wert `null` zugewiesen wird
- Ein Destruktor kann nicht explizit aufgerufen werden; der Garbage Collector ruft ihn auf

Garbage Collector (GC)

- Der Garbage Collector sucht im Speicher nach Objekten, welche nicht mehr referenziert werden und löscht diese durch Aufruf des Destruktors. Dadurch wird belegter Speicherplatz wieder freigegeben.
- Es kann nicht vorhergesagt werden, wann der Garbage Collector seine Aufgabe ausführt, jedoch spätestens, wenn die Speicherressourcen knapp werden.
- Der Programmierer kann den Garbage Collector anstossen mit:

```
GC.Collect();
```

Die `Dispose()`-Methode (1/3)

- Motivation für `Dispose()` :
 - Es kann nicht exakt vorbestimmt werden, wann ein Destruktor vom Garbage Collector ausgeführt wird.
 - Der Destruktor kann nicht explizit aufgerufen werden, auch nicht von der Klasse, in der er implementiert ist.
 - Beansprucht ein Objekt kostspielige oder begrenzte Ressourcen, muss sichergestellt sein, dass diese so rasch wie möglich wieder freigegeben werden.

Die Dispose () -Methode (2/3)

■ Problemlösung:

- Zusätzlich zum Destruktor, kann die öffentliche Methode `Dispose()` implementiert werden, welche der Benutzer der Klasse aufrufen kann.
- Die Klasse muss zusätzlich um das Interface `IDisposable` erweitert werden.

```
class MyClass : IDisposable
{
    public void Dispose()
    {
        // Anweisungen, um Ressourcen freizugeben
    }
}
```

Die Dispose()-Methode (3/3)

■ Aufruf von Dispose():

- Dispose() muss explizit vom Benutzer aufgerufen werden.

```
MyClass myObject = new MyClass();  
...  
myObject.Dispose();
```

■ Destruktor und Dispose() implementieren

- Da nicht garantiert werden kann, dass der Benutzer die Methode Dispose() aufruft, ist es ratsam, den Destruktor und die Methode Dispose() zu implementieren

```
class MyClass : IDisposable {  
    ~MyClass() { Dispose(); }  
    public void Dispose() {  
        // Anweisungen, um Ressourcen freizugeben  
    }  
}
```


Statische Klasselemente

Statische Variablen (Klassenvariablen)

■ Eigenschaften:

- Statische Variablen werden von allen Instanzen der Klasse geteilt; die Variable gibt es nur einmal und ist die identische

```
class ConnectionPool
{
    public static int ConnectionCount;
    ...
}
```

■ Zugriff auf Klassenvariablen mit Klassenname:

```
int con = ConnectionPool.ConnectionCount;
```

Statische Klassenelemente

Statische Methoden (Klassenmethoden)

■ Verwendung:

- Statische Methoden werden verwendet, wenn keine Objekte notwendig sind
- Typisch sind Methoden einer Bibliothek, wie z.B. die .NET Klasse `System.Math`, welche mathematische Methoden (`Pow`, `Sin`, `Cos`, `Tan`, ...) enthält

■ Beispiel: Elektrische Formeln

```
class Elektro
{
    public static double Power(double u, double i)
    {
        return u * i;
    }
    ...
}
```

Statischer Konstruktor

■ Eigenschaften:

- Der statische Konstruktor hat nur Zugriff auf die statischen Elemente einer Klasse
- Der statische Konstruktor wird automatisch aufgerufen, bevor die erste Instanz einer Klasse erstellt wird, oder der erste Zugriff auf ein statisches Klassenelement erfolgt
- Er wird zur Initialisierung der statischen Variablen verwendet

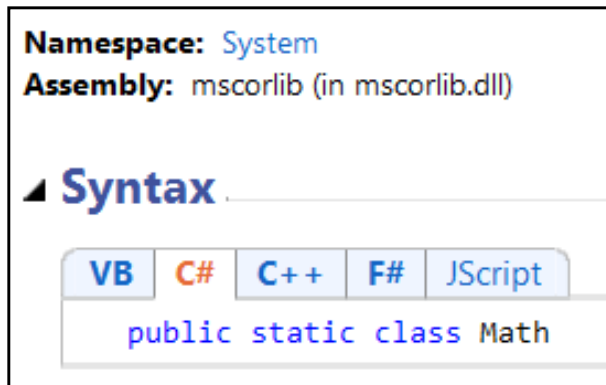
■ Beispiel:

```
class MyClass {  
    public static string ClassName;  
    // statischer Konstruktor  
    static MyClass() {  
        ClassName = "MyClass";  
    }  
}
```

Statische Klassenelemente

Statische Klassen – Beispiel `System.Math`

■ Beschreibung aus MSDN



■ Codebeispiel

```
double angle    = Math.PI * degrees / 180.0;  
double sinAngle = Math.Sin(angle);  
double cosAngle = Math.Cos(angle);
```

Vererbung

Wie definiert sich eine Vererbung?

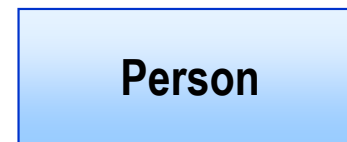
- Eine Vererbung wird durch eine „Ist ein“-Beziehung (engl. „is a“) identifiziert
- Vererbt werden alle Attribute und Methoden, die nicht als `private` gekennzeichnet sind.

```
class Person
{
    ...
}
class Lehrer: Person
{
    ...
}
```

Abgeleitete Klasse

Basisklasse

Doppelpunkt

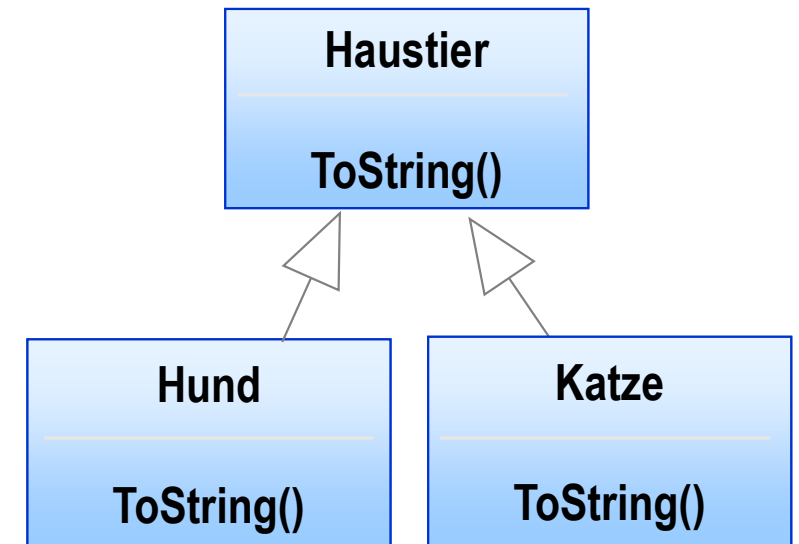


Spezialisierung



Polymorphismus (1/4)

- In einer Hierarchie angeordnete Klassen enthalten Methoden mit identischer Schnittstelle, welche jedoch für jede Klasse spezifisch implementiert ist
- Beispiel: `ToString()`-Methode
 - Die Methode `ToString()` wird in der Klasse `Object` implementiert und gibt für jede Klasse einen „vernünftigen“ String zurück.
 - Jede Klasse kann (sollte!) die Methode überschreiben und einen ausdrucksvollen String, der das Objekt kennzeichnet, zurückgeben.



Polymorphismus (2/4)

- Virtuelle Methoden der Basisklasse werden mit dem Schlüsselwort `virtual` gekennzeichnet.
- Beispiel:

```
class Person
{
    ...
    public virtual string Name( ) { ... }
}
```

Polymorphismus (3/4)

- Die virtuelle Methoden der Basisklasse müssen in der abgeleiteten Klasse mit dem Schlüsselwort `override` überschrieben werden
- Beispiel:

```
class Person
{
    ...
    public virtual string Name( ) { ... }
}

class Lehrer: Person
{
    ...
    public override string Name( ) { ... }
}
```


Polymorphismus (4/4)

- Es können nur als `virtual` markierte Methoden mit identischen benannten `override`-Methoden ersetzt werden
- `override`-Methoden können wieder überschrieben werden; `virtual` braucht dazu nicht explizit angegeben werden
- `override`-Methoden können nicht `static` oder `private` sein

Übung 3.2



Translator (Polymorphie)

- 1) Programmiere eine Basisklasse BaseTranslator mit folgenden virtuellen Methoden:
 - TranslateWord(string word) // Gibt das übersetzte Wort als String zurück
 - TranslateSentence(string sentence) // Gibt den übersetzten Satz als String zurück
- 2) Programmiere die Klassen EnglishTranslator und ItalianTranslator, welche von der Klasse BaseTranslator erben und die obigen Methoden überschreiben
- 3) Programmiere ein kleines Testprogramm, um den Polymorphismus zu testen:
 - Entwickle ein kleines Testprogramm, um ein Wort oder einen einfachen deutschen Satz einzugeben und diesen in die gewünschte Sprache zu übersetzen. Instanziiere den spezifischen Translator erst nachdem der Benutzer die Fremdsprache ausgewählt hat.

Methoden der Basisklasse mit `new` überschreiben

- Das Überschreiben einer Methode der Basisklasse, welche nicht mit `virtual` definiert ist, erfordert das Schlüsselwort `new` in der abgeleiteten Klasse.
- Ansonsten tritt beim Kompilieren eine Warnung auf. Mit dieser Warnung soll eine nicht beabsichtigte Überschreibung verhindert werden. Diese Situation kann auftreten, wenn die Methode zuerst in der abgeleiteten Klasse und erst später (neue Version) in der Basisklasse implementiert wurde.

```
public new string Methode( ) { ... }
```

Methoden der Basisklasse mit `new` überschreiben

■ Beispiel:

```
class Parent
{
    public void WriteHello()
    { Console.WriteLine("Parent->Hello"); }

    public void WriteByeBye()
    { Console.WriteLine("Parent->Bye Bye"); }

    public virtual void WriteSeeYou()
    { Console.WriteLine("Parent->See You"); }
}

class Child : Parent
{
    public new void WriteHello()
    { Console.WriteLine("Child->Hello"); }
    public void WriteByeBye()
    { Console.WriteLine("Child->Bye Bye"); }
    public override void WriteSeeYou()
    { Console.WriteLine("Child->See You"); }
}
```

Bewusste Überschreibung der Methode der Basisklasse mit `new`. Die Methode der Basisklasse wird versteckt.

Warnung beim Kompilieren:
Bewusste Überschreibung der Methode der Basisklasse

⚠ 1 'TestOverrideMethodWithNew.Child.WriteByeBye()' hides inherited member 'TestOverrideMethodWithNew.Parent.WriteByeBye()'. Use the new keyword if hiding was intended.

Elemente der Basisklasse aufrufen mit `base`

- Beispiel:
Konstruktor der Basisklasse mit Parameterübergabe aufrufen

```
class Person
{
    protected Person(string personName) { ... }
}
class Lehrer: Person
{
    public Lehrer(string lName) : base(lName) {...}
}
```

Eine Klasse vor Vererbung schützen

- Mit dem Schlüsselwort `sealed` kann die Klasse versiegelt und somit nicht mehr von ihr geerbt werden.
- Beispiel:

```
namespace SystemClasses
{
    public sealed class String
    {
        ...
    }
}

namespace MyClasses
{
    public class MeinString: String { ... }
}
```

**Vererbung
nicht möglich!**

Verschachtelte Klassen

- Werden innerhalb einer Klasse deklariert
- Die Sichtbarkeit wird über die bekannten Schlüsselworte definiert

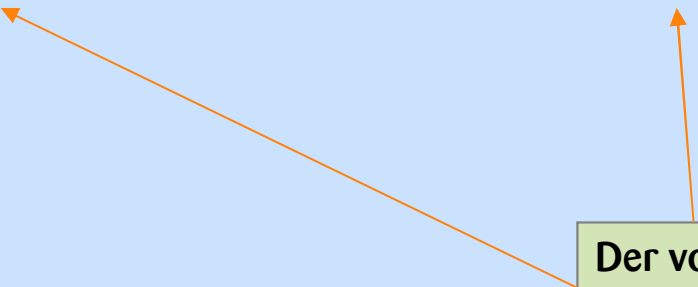
```
class Bank
{
    public class Account { ... }
    private class AccountNumberGenerator { ... }
}
class Program
{
    static void Main( )
    {
        Bank.Account; // zugreifbar
        Bank.AccountNumberGenerator; // gesperrt
    }
}
```



Verschachtelte Klassen

```
class Program
{
    static void Main( )
    {
        Bank.Account yours = new Bank.Account( );
    }
}

class Bank
{
    ... class Account { ... }
}
```



Der volle Name der verschachtelten Klasse enthält den Namen der äusseren kapselnden Klasse

Objekte vergleichen

```
public class Truck
{
    public string Brand { get; set; }

    public Truck(string brand)
    {
        Brand = brand;
    }
}
```

Wann sind Objekte gleich?

- Zwei Referenzen referenzieren das gleiche Objekt
 - Methode `Equals()` liefert `true`, wenn das gleiche Objekt referenziert wird
- Daten zweier Objekte sind identisch
 - Im Speicher können es zwei unterschiedliche Objekte sein
 - Alle Werte der Objekte müssen auf Gleichheit überprüft werden

```
Truck truck1 = new Truck("Volvo");
Truck truck2 = new Truck("Mercedes");
Truck truck3 = new Truck("Volvo");
Truck truck4 = truck1;

Console.WriteLine($"Truck 1: {truck1.Brand}");
Console.WriteLine($"Truck 2: {truck2.Brand}");
Console.WriteLine($"Truck 3: {truck3.Brand}");
Console.WriteLine($"Truck 4: {truck4.Brand}");

Console.WriteLine($"Is Truck 1 == Truck 2: {truck1.Equals(truck2)}");
Console.WriteLine($"Is Truck 1 == Truck 3: {truck1.Equals(truck3)}");
Console.WriteLine($"Is Truck 1 == Truck 4: {truck1.Equals(truck4)}");
```

Ausgabe

```
Objekte vergleichen
-----
Truck 1: Volvo
Truck 2: Mercedes
Truck 3: Volvo
Truck 4: Volvo
Is Truck 1 == Truck 2: False
Is Truck 1 == Truck 3: False
Is Truck 1 == Truck 4: True
```

Konvertierung

Der `is`-Operator

- Der Rückgabewert ist `true`, falls eine Konvertierung möglich ist
- Beispiel: Ist das von `objectRef` referenzierte Objekt von der Klasse `Car`?

```
public class Car
{
    public string Brand { get; set; }
    public string Model { get; set; }
}
```

```
Car car1 = new Car() { Brand = "FIAT", Model = "500 Abarth" };
object objectRef = car1;
Car myCar;

// Operator is
if (objectRef is Car) // Überprüfung
{
    myCar = (Car) objectRef; // Casting
    Console.WriteLine($"myCar = {myCar.Brand}, {myCar.Model}");
}
else
    Console.WriteLine("objectRef ist nicht vom Typ Car.");
```

Konvertierung

Der `as`-Operator

- Konvertiert zwischen zwei Referenztypen
- Im Fehlerfall:
 - Liefert `null`-Referenz zurück und wirft keine Exception
- Beispiel:

```
public class Car
{
    public string Brand { get; set; }
    public string Model { get; set; }
}
```

```
Car car1 = new Car() { Brand = "FIAT", Model = "500 Abarth" };
object objectRef = car1;
Car myCar;

// Operator as
myCar = objectRef as Car; // Casting

if (myCar != null)
{
    Console.WriteLine($"myCar = {myCar.Brand}, {myCar.Model}");
}
else
    Console.WriteLine("objectRef ist nicht vom Typ Car.");
```

Konvertierung von und zum Typ `object`

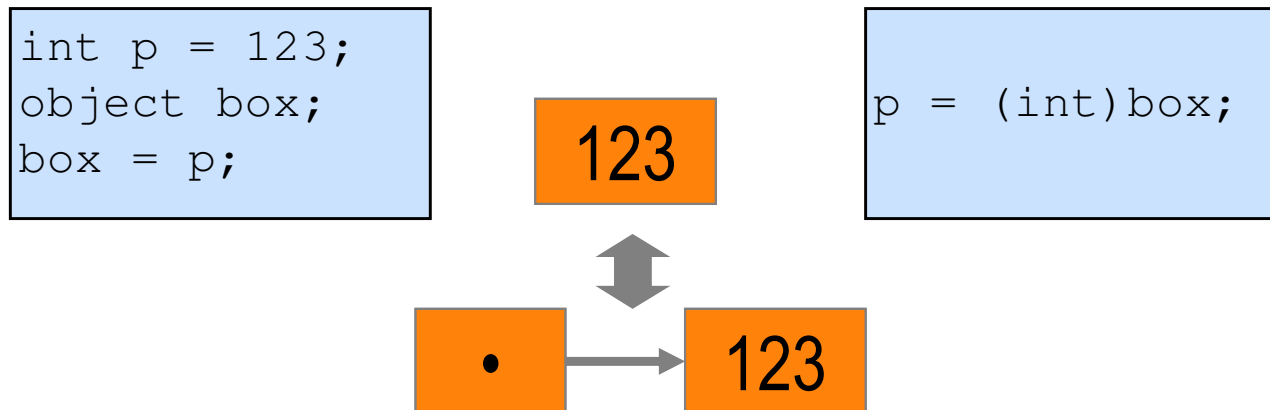
- Jede beliebige Referenz kann einer `object`-Referenz zugewiesen werden
- Umgekehrt muss dafür gesorgt werden, dass die links-seitige Referenz vom Typ der rechts-seitigen Referenz ist oder in der Vererbungshierarchie oberhalb dieser steht.

```
object oRef;  
Person pRef = new Person();  
// Person Ref nach Object Ref  
oRef = pRef;  
oRef = (object) pRef;  
oRef = pRef as object;
```

```
// Object Ref nach Person Ref  
pRef = (Person) oRef;  
pRef = oRef as Person;
```

Boxing und Unboxing

- Wird im Unified Type System verwendet, um
 - Aus einem Werttyp einen Referenztypen zu erzeugen (Boxing)
 - Aus einem Referenztyp einen Werttyp zu erzeugen (Unboxing)



- Beispiel:
 - Aufruf einer Methode auf einem Werttyp: `Int32.ToString()`

Charakteristiken von abstrakten Klassen

- Teilweise implementierte Klasse
- Mindestens eine Methode der Klasse ist nicht implementiert
- Signatur der nicht implementierten Methode wird vorgegeben
- Gemeinsamer Teil einer Gruppe von Klassen
- Gemeinsame Struktur und Logik werden in Basisklasse zusammengefasst
- Basisklasse selbst kann/soll nicht erzeugt werden

Abstrakte Klassen

Deklaration der Klasse mit dem Schlüsselwort `abstract`

```
abstract class Token
{
    ...
}

class Test
{
    static void Main( )
    {
        Token myToken = new Token( ); ❌
    }
}
```

Eine abstrakte Klasse kann
nicht instanziiert werden

Abstrakte Methoden

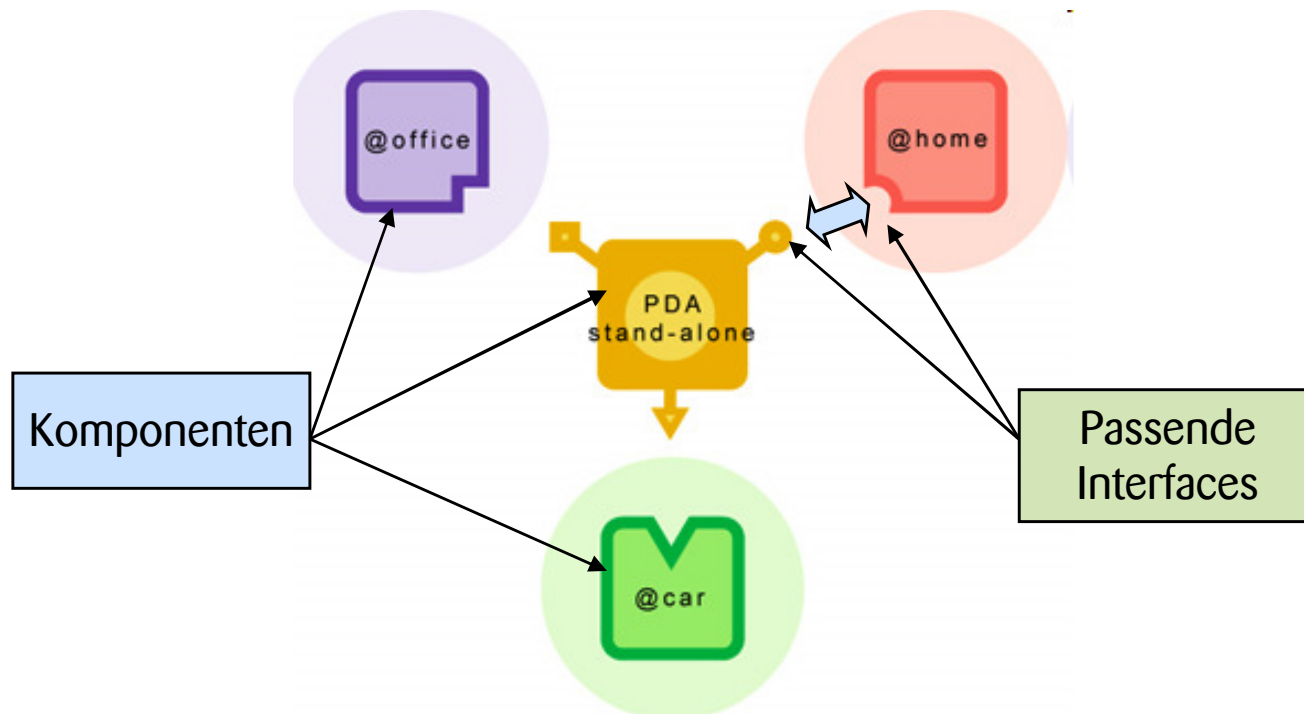
- Syntax: Schlüsselwort `abstract` verwenden

```
abstract class Token
{
    public virtual string Name( ) { ... }
    public abstract int Length( );
}
class CommentToken: Token
{
    public override string Name( ) { ... }
    public override int Length( ) { ... }
}
```

- Nur abstrakte Klassen können abstrakte Methoden enthalten
- Abstrakte Methoden können keine Methodenimplementierung enthalten

Charakteristiken von Interfaces (Schnittstellen)

- Definieren die Schnittstellen von Klassen oder Komponenten
- Beispiel: PDA



Charakteristiken von Interfaces

- Ein Interface kann folgende Elemente vorschreiben:
 - Methoden
 - Eigenschaften
 - Indexer
- Wird oft mit Präfix „I“ als Interface gekennzeichnet
- Enthält keine Implementation der Methoden
- Enthält keine Zugriffsoperatoren; alle Methoden sind automatisch `public`

- Beispiel:

```
interface IComparable
{
    int CompareTo(object obj);
}
```

Interfaces

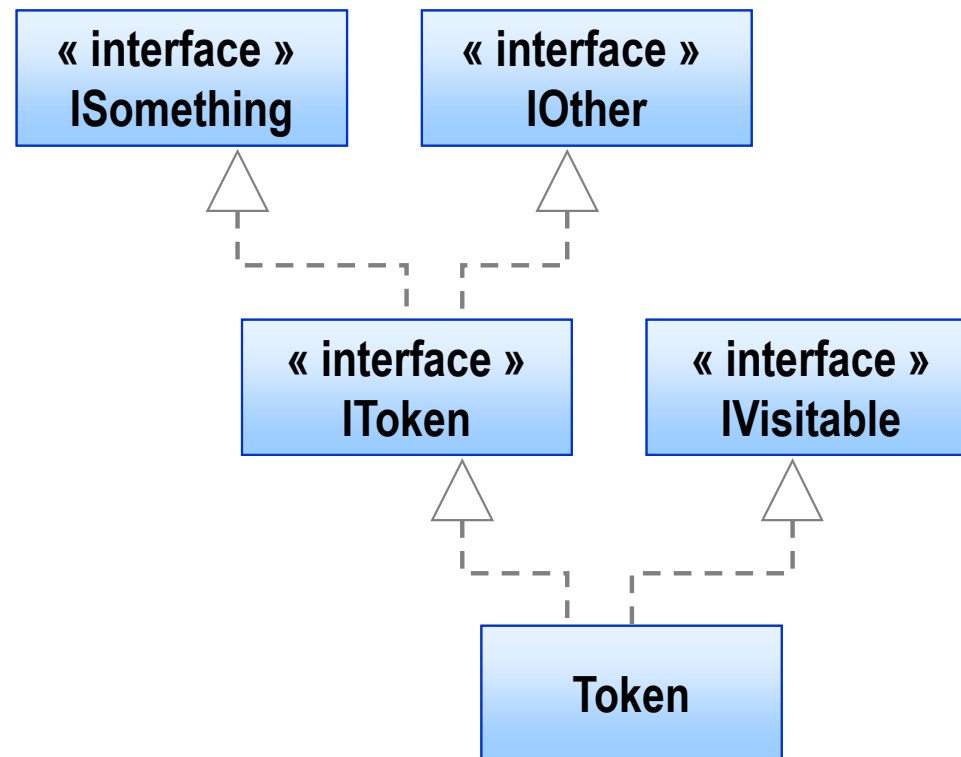
- Eine Klasse kann mehrere Interfaces implementieren
- Die Klasse «erbt» sozusagen die Deklaration der Schnittstelle
- Die Methoden müssen in der abgeleiteten Klasse definiert werden

```
class Person : IComparable
{
    private string Nachname;
    private string Vorname;

    public int CompareTo(object obj)
    {
        if (obj is Person)
        {
            Person p = (Person)obj;
            return Nachname.CompareTo(p.Nachname);
        }
    }
}
```

Interfaces

Mehrere Interfaces



Methoden implementieren

- Alle Methoden müssen implementiert werden
- Die implementierten Methoden können `virtual` sein

```
class Token: IToken, IVisitable
{
    public virtual string Name( )
    { ...
    }
    public void Accept( )
    { ...
    }
}
```

Interfaces mit gleichen Methodennamen

- Es muss der voll qualifizierte Name der Interface-Methode verwendet werden
- Beispiel: Die Interfaces `IToken` und `IVisitable` deklarieren beide die Methode `Accept()`

```
class Token: IToken, IVisitable
{
    string IToken.Accept( )
    { ...
    }

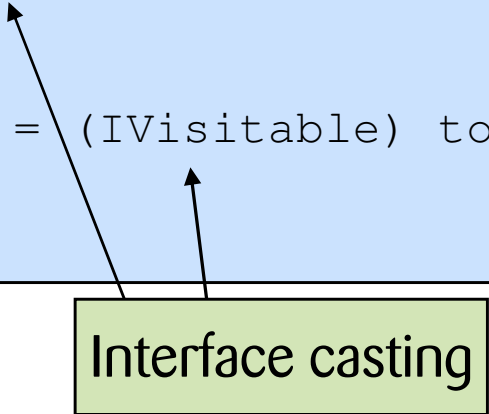
    void IVisitable.Accept()
    { ...
    }
}
```

Interfaces

Interfaces mit gleichen Methodennamen

- Zur Verwendung muss auf das passende Interface konvertiert (to cast) werden
- Beispiel:

```
Token token = new Token();  
  
IToken iToken = (IToken) token;  
iToken.Accept();  
  
IVisitable iVisitable = (IVisitable) token;  
iVisitable.Accept();
```



Abstrakte Klasse vs. Interfaces

- Keine Mehrfachvererbung in C#
- Abstrakte Klasse erlaubt nur genau eine Basisklasse
- Abstrakte Klasse erlaubt teilweise implementierte Basisklassen
- Es können mehrere Interfaces geerbt werden
- Keine Interface-Versionierung (ab C# 8 schon)
- Da Interfaces keine Implementation enthalten, müssen bei einer Interface-Änderung alle erbenden Klassen an das neue Interface angepasst werden («breaking change») (ab C# 8 nicht)

Übung 3.3



Geometrische Formen (Interfaces)

- 1) Erstelle ein Interface `IForm` für geometrische Formen mit folgenden Elementen:
 - Methode: `BerechneFläche()`
 - Eigenschaft: `Umfang`
- 2) Implementiere die beiden Klassen `Quadrat` und `Kreis`, welche das Interface `IForm` umsetzen.
- 3) Erstelle eine TestTreiber-Klasse, um das Interface zu testen. Die Klasse soll eine Testmethode `Drucke()` mit folgender Signatur haben:
 - `static void Drucke (IForm myForm)`
 - Die Methode `Drucke` soll Fläche und Umfang der Form auf die Console schreiben.
- 4) Teste die Implementierung

Einige .NET-Standard-Interfaces

Interface:	Beschreibung
Comparable	Dieses Interface wird von Typen/Klassen implementiert, deren Werte sortiert werden können. Es definiert Methoden zum Sortieren.
Disposable	Dieses Interface wird primär verwendet, um nicht verwaltete (unmanaged) Ressourcen frei zu geben.
Convertible	Dieses Interface definiert Methoden, um den Wert der Instanz in Werte der CLR Typen zu konvertieren.
Cloneable	Definiert ein Standard Interface, um eine exakte Kopie einer Instanz zu erstellen.
Equatable	Definiert ein Standard Interface, um Objekte zu vergleichen .
Formatable	Definiert Methoden, um Daten formatiert auszugeben. Dadurch wird die Ausgabe flexibler als mit <code>ToString()</code> .

Interfaces

Beispiel Interface Comparable

Methoden:	Beschreibung
<code>int compareTo(Object obj)</code>	Vergleicht die aktuelle Instanz mit einem anderen Objekt desselben Typs. Wird zur Bestimmung der Reihenfolge zur Sortierung verwendet.

```
...public interface Comparable
{
    ...int compareTo(object obj);
}
```

Parameter:	Bedeutung
<code>obj</code>	Ein Objekt, das mit dieser Instanz verglichen werden soll.

Rückgabewerte	Bedeutung
Kleiner als 0	Diese Instanz ist kleiner als obj.
0	Diese Instanz ist gleich obj.
Grösser als 0	Diese Instanz ist grösser als obj.

Ausnahmen:	Bedingung
<code>ArgumentException</code>	obj hat nicht denselben Typ wie diese Instanz.

Interfaces

Beispiel Interface ICloneable

Methoden:	Beschreibung
object Clone()	Erstellt ein neues Objekt, das eine Kopie der aktuellen Instanz darstellt.

```
...public interface ICloneable
{
    ...object Clone();
}
```

Parameter:	Bedeutung
keine	

Rückgabewerte	Bedeutung
object	Ein neues Objekt, das eine Kopie dieser Instanz ist.

Ausnahmen:	Bedingung
keine	

Übung 3.4



Umrechnungsformeln

$$C = K - 273.15$$

$$K = C + 273.15$$

$$F = C \times 1.8 + 32$$

$$C = (F - 32) \div 1.8$$

Temperaturen (Comparable)

- 1) Erstelle eine Klasse `Temperatur` mit folgenden Elementen:
 - Eigenschaft: `Celsius`
 - Eigenschaft: `Kelvin`
 - Eigenschaft: `Fahrenheit`
 - Interface: `Comparable`
- 2) Erstelle eine Konsolenapplikation zum Testen der Klasse
 - Instanziiere zwei Objekte `t1` und `t2` der Klasse `Temperatur` und vergleiche sie untereinander mit der Methode `compareTo()`.
- 3) Erstelle eine Messreihe von 10 Temperaturen
 - Benutze ein Array von Temperaturen, um die Messreihe zu speichern
 - Benutze die Methode `Array.Sort()`, um die Temperaturwerte nach deren Grösse zu sortieren