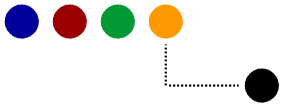
Vertiefung Objektorientierung Vererbung



- Begriffliche Grundlagen aus Realität & Denken
- Implementierungsvererbung
- Zugriff durch Unterklasse
- Kapselung
- Indirekter Zugriff

Vererbung Inheritance

Reale Welt : **Objekte** sind begrifflich **hierarchisch klassifizierbar**

Hierarchisch tieferstehende **Begriffe** bezeichnen **speziellere** Varianten von **Oberbegriffen**

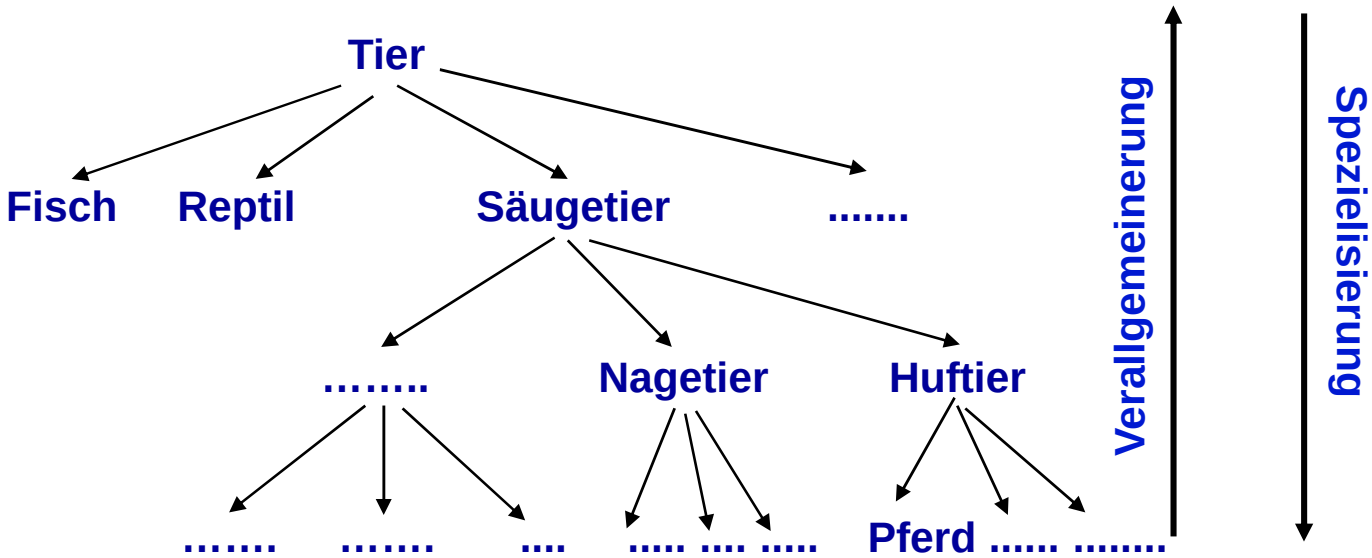
→ **Spezialisierungshierarchie**

⇒ Besitzen **deren** Eigenschaften **+** noch **weitere** **spezifischere** Eigenschaften

Vererbung : Klassen **übernehmen** Methoden + Attribute anderer Klassen …

… können aber **zusätzliche** Methoden + Attribute enthalten

Klasse erbt von übergeordneter Klasse + enthält optional weitere Methoden / Attribute



Begriffshierarchie

Test-Relation :

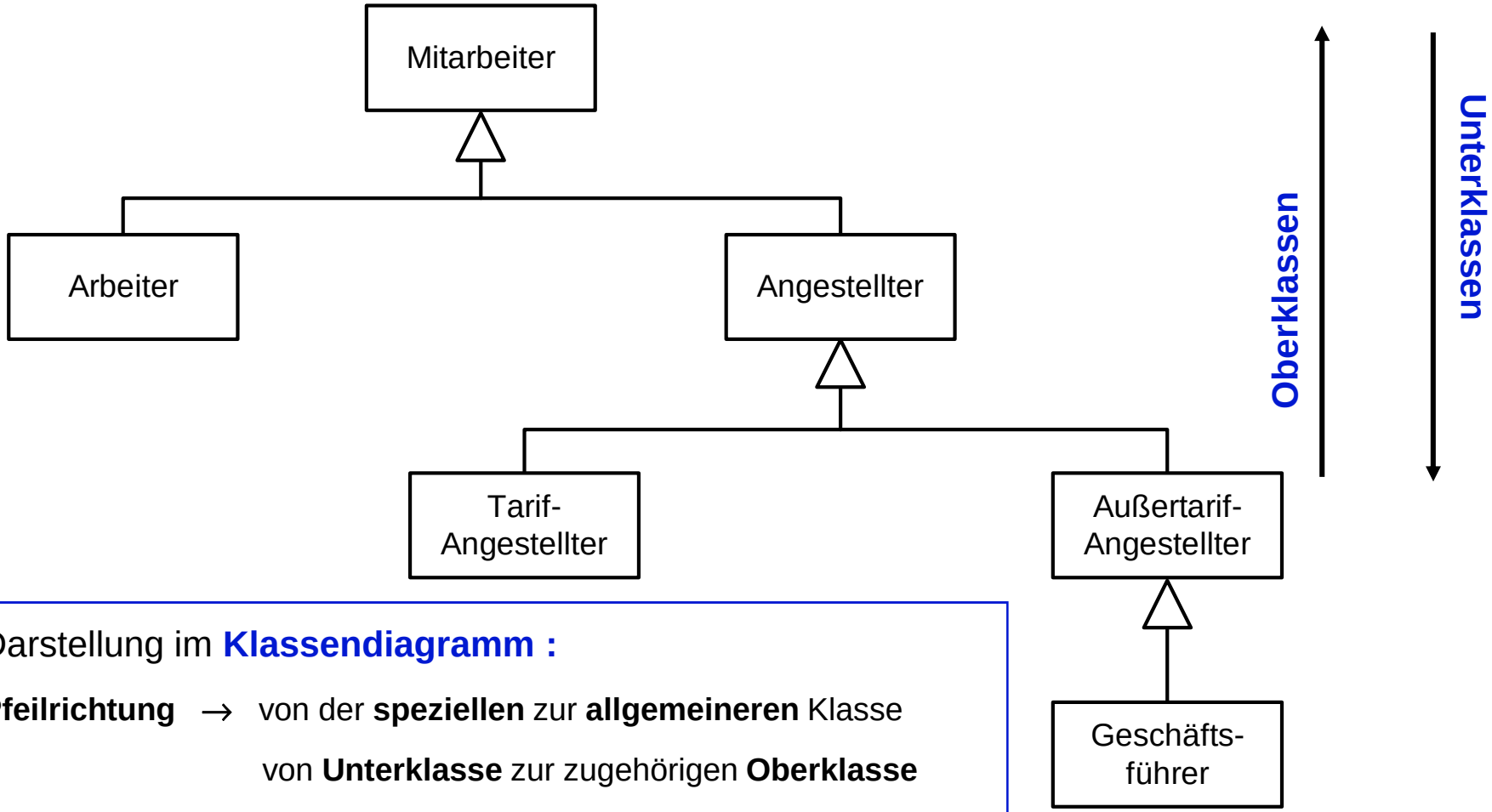
" Ist ein ... "

Jedes Säugetier ist ein Tier

Nur wenn eine durchgängige "Ist-Ein"-Relation besteht, ist die Begriffshierarchie semantisch sinnvoll !

Vererbung

Kann sich über **zahlreiche** Ebenen erstrecken ⇒ **Hierarchie**



Darstellung im **Klassendiagramm :**

Pfeilrichtung → von der **speziellen** zur **allgemeineren** Klasse

von **Unterklasse** zur zugehörigen **Oberklasse**

Java : Eine Unterklasse kann nur **eine direkte** Oberklasse haben =

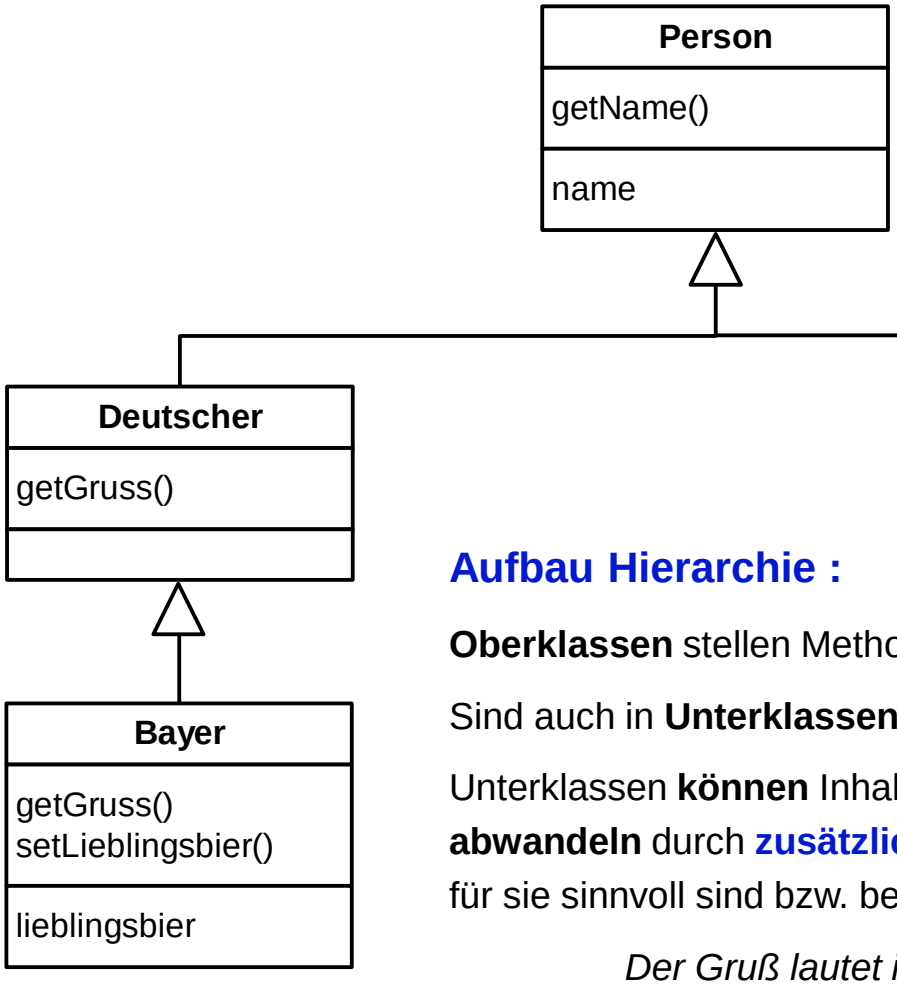
Einfachvererbung **keine** Mehrfachvererbung !

Implementierungsvererbung - *Syntax :*

Schlüsselwort **extends** in Klassendeklaration

Implementierungsvererbung

Schlüsselwort extends




```
class Deutscher extends Person {
    // …
}
```

Spezialisierungshierarchie

Aufbau Hierarchie :

Oberklassen stellen Methoden + Attribute bereit

Sind auch in **Unterklassen** verwendbar

Unterklassen **können** Inhalt der Oberklassen **spezifisch erweitern + abwandeln** durch **zusätzliche** eigene Methoden + Attribute, die erst für sie sinnvoll sind bzw. besondere Ausprägung annehmen

Der Gruß lautet in jeder Sprache anders …

Unterklasse (subtype) ist spezielle Abart der Oberklasse (supertype)

$$\text{public Attribute}^{\text{Ober}} \subseteq \text{public Attribute}^{\text{Unter}}$$

$$\text{public Methoden}^{\text{Ober}} \subseteq \text{public Methoden}^{\text{Unter}}$$

Vererbung - extends

In Unterklassen **alle public** Methoden **+** Attribute der Oberklasse **verfügbar / ansprechbar**

< Aber : Konstruktoren werden **nicht** vererbt (s.u.) >

⇓

Objekte der Klassen **Franzose** und **Deutscher** haben **geerbte** public Methode **getName( )** und **geerbtes** public Attribut **name**

Oberklasse Person liefert **public Elemente**, die auch in Unterklassen direkt **verwendbar** sind

Erweiterung in Unterklassen durch spezifische Methode **getGruss( )**

Konzentration von Code in Oberklasse
Vermeidet Duplizieren von Code in Unterklassen

Wiederverwendbarkeit vereinfacht Testen, Fehlersuchen, Wartung

```
class Person  {
   public String name ;
   public Person( ) {
         name = "Anonym" ;
   }
   public String getName( )  {
          return name ;
   }
}

class Franzose  extends Person  {
   public Franzose( String nn ) {
         name = nn ;
   }
   public String getGruss( ) {
          return "Bonjour " + getName( ) ;
   }
}

class Deutscher  extends Person  {
   public Deutscher( String nn ) {
         name = nn ;
   }
   public String getGruss( ) {
          return "Mahlzeit "  + getName( ) ;
   }
}
```

Vererbung - Zugriff


```
class Person {
   public String name ;
   public int alter ;
   public Person( ) {
         name = "Anonym" ;
   }
   public String getName( ) {
          return name ;
   }
}

class Franzose extends Person {
   public Franzose( String nn ) {
         name = nn ;
   }
   public String getGruss() {
        String s = "Bonjour " + getName( ) ;
        return s ;
   }
}
```

```
class Test {
    public static void main( String[] args ) {

          Franzose f = new Franzose( "Jean" ) ;

          IO.writeln( f.getGruss( ) ) ;

          IO.writeln( f.getName( ) ) ;

          f.alter = 25 ;

          IO.writeln( f.name + " : " + f.alter ) ;
     }
}
```

Public Attribute + Methoden von *Person* stehen in Implementierung der Klasse *Franzose* und bei Objekten vom Typ *Franzose* zur Verfügung

Auch **statische** public Attribute und Methoden werden vererbt und stehen für Unterklasse als **statische** Elemente zur Verfügung

Öffentliche **statische** Attribute der **Oberklasse** sind durch alle Unterklassen + ihre Objekte manipulierbar – dabei wird für **alle** ein ***einheitlicher*** Wert gehalten.

Vererbung - Kapselung

In Unterklassen alle **public** Methoden + Attribute der Oberklasse ansprechbar

Aber :

Auf private Attribute + Methoden der Oberklasse hat auch Unterklasse keinen Zugriff !

Ebenso wenig wie alle anderen Klassen !

Kein Unterlaufen der Kapselung durch Vererbung !!

Weitere Zugriffsspezifikationen :

protected ***package***

→ Vererbung + Paketkonzept

```
class Person {
   public String name ;
   public int alter ;
   private String ort ;
   public Person( ) {
         name = "Anonym" ;
   }
   public String getName( ) {
          return name ;
   }
    private void intern( ) {  /* … */  }
}

class Franzose  extends Person {
   public Franzose( String nn ) {
         name = nn ;
   }
   public String getGruss( ) {
      return "Bonjour" + getName( ) ;
   }
   public void test( ) {
          ort = "Mosbach" ;     //  Fehler!
          intern( )             //  Fehler!
   }
}
```

Vererbung - Indirekter Zugriff

In Unterklassen ist ein *indirekter* Zugriff auf private Elemente der Oberklasse möglich :

Via öffentlicher Methoden der Oberklasse !

⇓

Mit den **Unterklassen-Objekten** werden auch entsprechende **Oberklassen-Objekte** im Speicher vorgehalten !

⇓

Werden "automatisch" mit ihren **Unterklassen-Objekten** bei **deren** Erzeugung angelegt

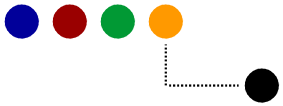
⇓

Objekterzeugung + **Konstruktoren** in **Klassenhierarchien** sind noch zu untersuchen !

```
class Person {
    public Person( ) {
        // …
    }
    public String getName( ) {
        return name ;
    }
    public void setName( String n ) {
        name = n ;
    }
    private String name = "Anonym" ;
}

class Franzose  extends Person {
    public Franzose( String nn ) {
        setName( "Müller" ) ;
    }
    public String getGruss( ) {
        return "Bonjour" + getName( ) ;
    }
    // …
}
```

Vertiefung Objektorientierung Vererbung



Spezielle Techniken & Regeln :

- Überschreiben von Methoden
- "Überschreiben" von Attributen
- super. - Zugriffe

Überschreiben von Methoden

Spezialisieren geerbter Methoden in Unterklasse :

Klasse **Bayer** hat **eigene Methode** **getGruss( )**

⇓

Für **Bayer-Objekte** wird bei getGruss()-Aufruf die **eigene** getGruss()-Methode aufgerufen - **nicht** geerbte getGruss()-Methode von **Deutscher** !

Überschreiben (Overriding) :

Deklaration Methode in Unterklasse mit **gleicher Schnittstelle** (Name, Parameter, Rückgabewerte) wie in Oberklasse …

… jedoch **spezieller Implementierung**

```
class Deutscher extends Person  {
   public Deutscher( String nn ) {
         // …
   }
   public String getGruss( ) {
         return "Mahlzeit" ;
   }
}

class Bayer extends Deutscher {
   public Bayer( String nn )  {
         // …
   }
   public String getGruss( )  {
         return "Grüß Gott!" ;
   }
}
```

Sinn : **Anpassen** geerbter Methoden an **spezifische Bedürfnisse** der Unterklassen

Einbau zusätzlicher Prüfungen, Verwenden anderer interner Datenformate …

Überschreiben von Methoden

Regeln :

1. **Kein Einschränken** der **Sichtbarkeit** beim Überschreiben,

   d.h. public nicht durch protected oder private überschreibbar !

2. **Nicht-statische** Methode nur durch **nicht-statische** Methode überschreibbar,

   statische Methode nur durch **statische** Methode überschreibbar !

Sinn der Regeln :

Beim Überschreiben soll **Vertrag der Oberklasse** auch in Unterklassen-Implementierung **eingehalten** werden.

Nicht verwechseln :

Überschreiben ≈ "Verbergen" geerbter Methoden in Unterklasse durch deren eigene "Version"

Überladen ≈ "Nebeneinander" unterschiedlich parametrisierter Methoden in derselben Klasse

Überschreiben von Methoden – Regeln


```
class Deutscher extends Person  {
   public Deutscher( String nn ) {   /* … */  }
   public String getGruss( ) {  return "Mahlzeit" ;  }
}

class Bayer extends Deutscher {
   public Bayer( String nn )  {   /* … */  }
   @Override  public String getGruss( )  {  return "Servus!" ;  }     // OK !
   // …
   private String getGruss( )  {  return "Servus" ;  }    //  Fehler  –  Zugriffsrechte !
   public int getGruss( )  {  return 11111 ;  }             //  Fehler  –  return-Typ !

   public String getGruss( int n )  {  return "Servus!" + n ;  }
   // OK – aber kein Überschreiben sondern Überladen
   //  Geerbte Methode  getGruss( )  wurde überladen
   //  Klasse Bayer hat nun beide Varianten
   //                              ⇓
   // Kombination von überladener und geerbter Variante :
   public String getGruss( int n )  {  return getGruss( ) + n ;  }
}
```

Annotation @Override

Compiler- Prüfung, ob **wirklich** überschrieben oder (unabsichtlich) überladen wird !

```
@Override
public String getGruss( int n ){  … }
```

Compilerfehler !

Super-Aufrufe - nicht-statische Methoden

Schlüsselwort **super.*methodenname*(…)**

Aufruf einer geerbten Methode der direkten Oberklasse in Unterklassen-Objekt

Unterklasse überschreibt geerbte Methode der Oberklasse.

Dennoch kann auch weiterhin die **geerbte Methode** der Oberklasse **gerufen** werden

⇓

Auch nach Überschreiben geerbter Methoden sind geerbte "Orginal"-Oberklassenmethoden in Unterklasse **zugreifbar**

Einschränkungen :

In **statischen Methoden** steht **super.** ***nicht*** zVfg.

Zugriff auf statische Methoden der Oberklasse via **Oberklassenname** jedoch erlaubt

Kein ~~**return super**~~ möglich !

Kein super auf Objektinstanzen: ~~**f.super.getName() ;**~~ !

```
class Person {
   public Person( )  {
         name = "Anonym" ;
   }
   public String getName( ) {
          return name ;
   }
   public String name ;
}


class Franzose extends Person  {
   public Franzose( String nn ) {
         name = nn ;
   }
   public String getName( ) {
        String n = super.getName( ) ;
        n = "Name ist: " + n ;
        return n ;
   }
}
```

"Überschreiben" von Attributen

Verbergen / Hiding

Vererbte Attribute **können** überschrieben werden :

Primitive Typen ebenso wie *Referenz*typen

Nicht-statische ebenso wie statische Attribute

Finale ebenso wie nicht-finale Attribute

Deklaration gleichnamiger Attribute in Unterklasse *verbirgt* geerbte Attribute der Oberklasse –

für tiefere Unterklassen ebenso wie für Objekte der überschreibenden Klasse.

⇓

Bei Wertzuweisung wird **eigenes** Attribut der Unterklasse belegt.

Zugriff auf gleichnamige geerbte Attribute der Oberklasse **weiterhin** mittels :

super.*variablenname*

Sichtbarkeit *darf* eingeschränkt werden !

Typ *kann* sich ändern ! **Finalität *kann*** sich ändern!

```
class Ober {
   public final boolean testvar ;
   public Ober ( ) {
        testvar = true ;
   }
}

class Unter extends Ober {
    // Eigenes Attribut
    // verbirgt von Oberklasse geerbtes Attribut
    private long testvar ;

    public Unter ( long a ) {
       testvar = a ;
       super.testvar = false ;
    }
}
```

"Überschreiben" von Attributen

Verbergen / Hiding

Sinn :

Verbergen geerbter öffentlicher Attribute durch **private Überschreibung** in Unterklasse

Sind somit **nicht mehr** Teil der öffentlichen **Schnittstelle** der Unterklasse

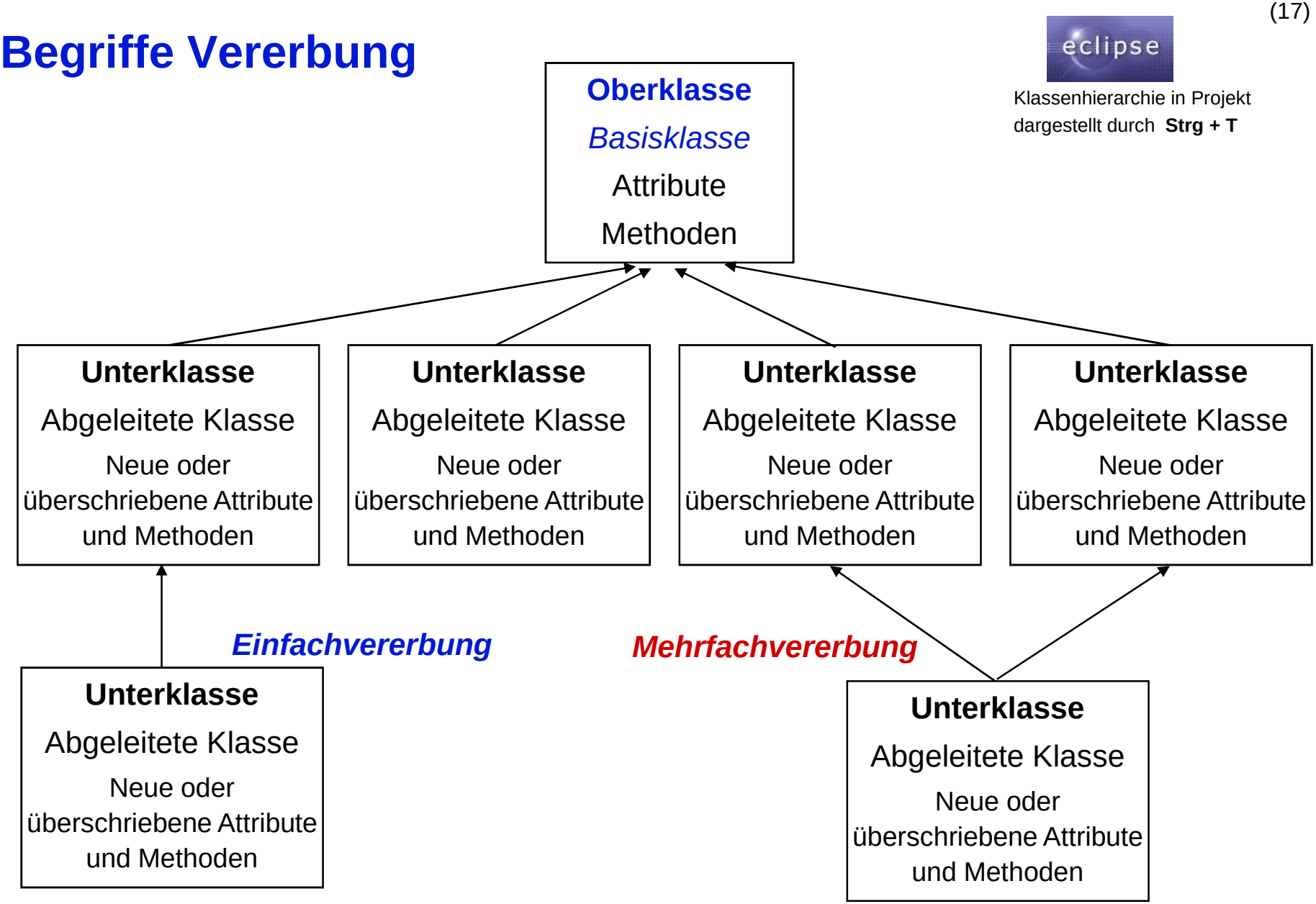
Werden **nicht mehr** an tiefere Unterklassen **vererbt**

Dadurch **"heilt"** die Unterklasse eine **zu offene Schnittstelle** ihrer zu offenherzigen Oberklasse

```
class BadUpper {
   public boolean b ;
   public double d ;
   public Konto k ;
   public BadUpper ( )  { /* … */ }
}

class GoodUnder extends BadUpper {
   //  Überschreibende Attribute verbergen
   //  von Oberklasse geerbte Attribute
   private boolean b ;
   private double d ;
   private Konto k ;
   // …
}
```

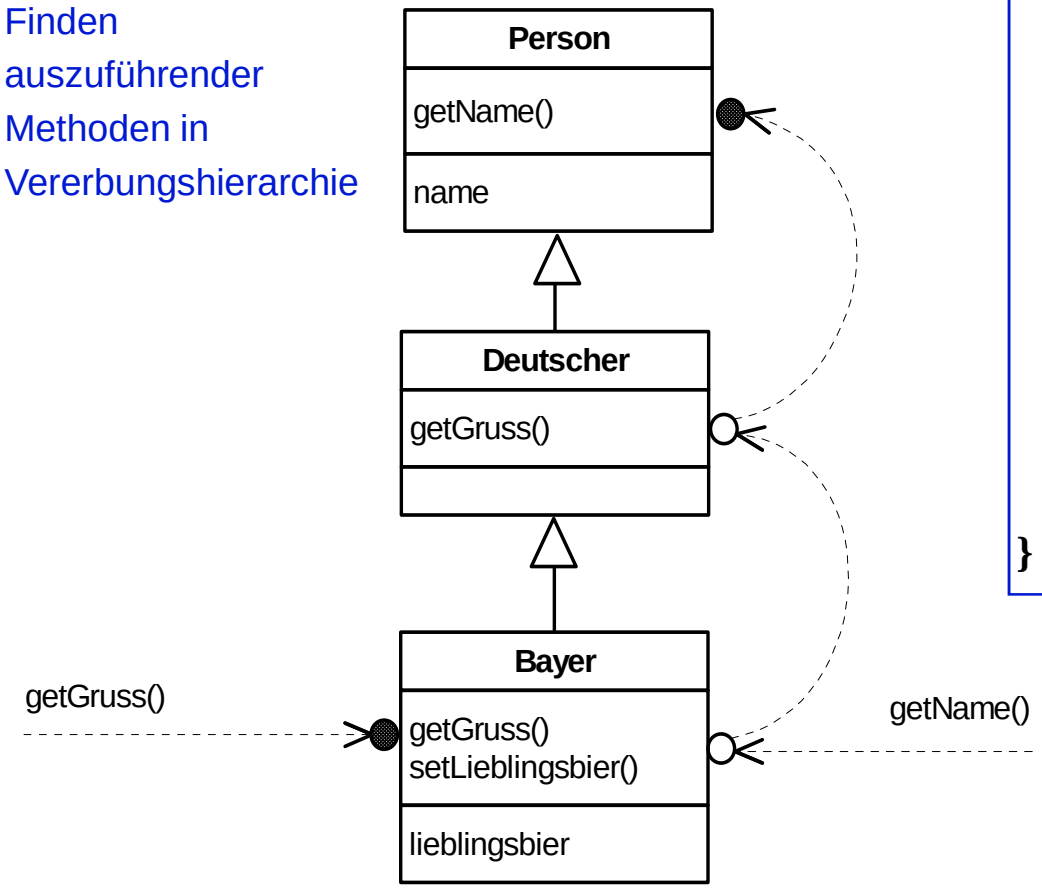
Begriffe Vererbung



Klassenhierarchie in Projekt dargestellt durch **Strg + T**

Methodenauswahl durch JVM

Finden auszuführender Methoden in Vererbungshierarchie




```
class GrussAusgabe  {
   public static void main( String[ ] args )  {

      Bayer sepp = new  Bayer( "Sepp" ) ;

      IO.writeln(  sepp.getName( )  ) ;
      IO.writeln(  sepp.getGruss( )  ) ;
   }
}
```

Ausführung von Methoden aus verschiedenen Ebenen der Hierarchie

getName( ) : aus Klasse Person

getGruss( ) : aus Klasse des Objekts

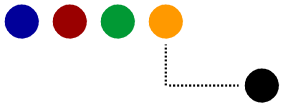
Prinzip :

Methode in Klasse des beauftragten Objekts vorhanden ?

Ja ⇒ **dort** deklarierte Methode ausführen !

Nein ⇒ **Suche** in nächst höherer Klasse der Hierarchie fortsetzen !

Vertiefung Objektorientierung Vererbung

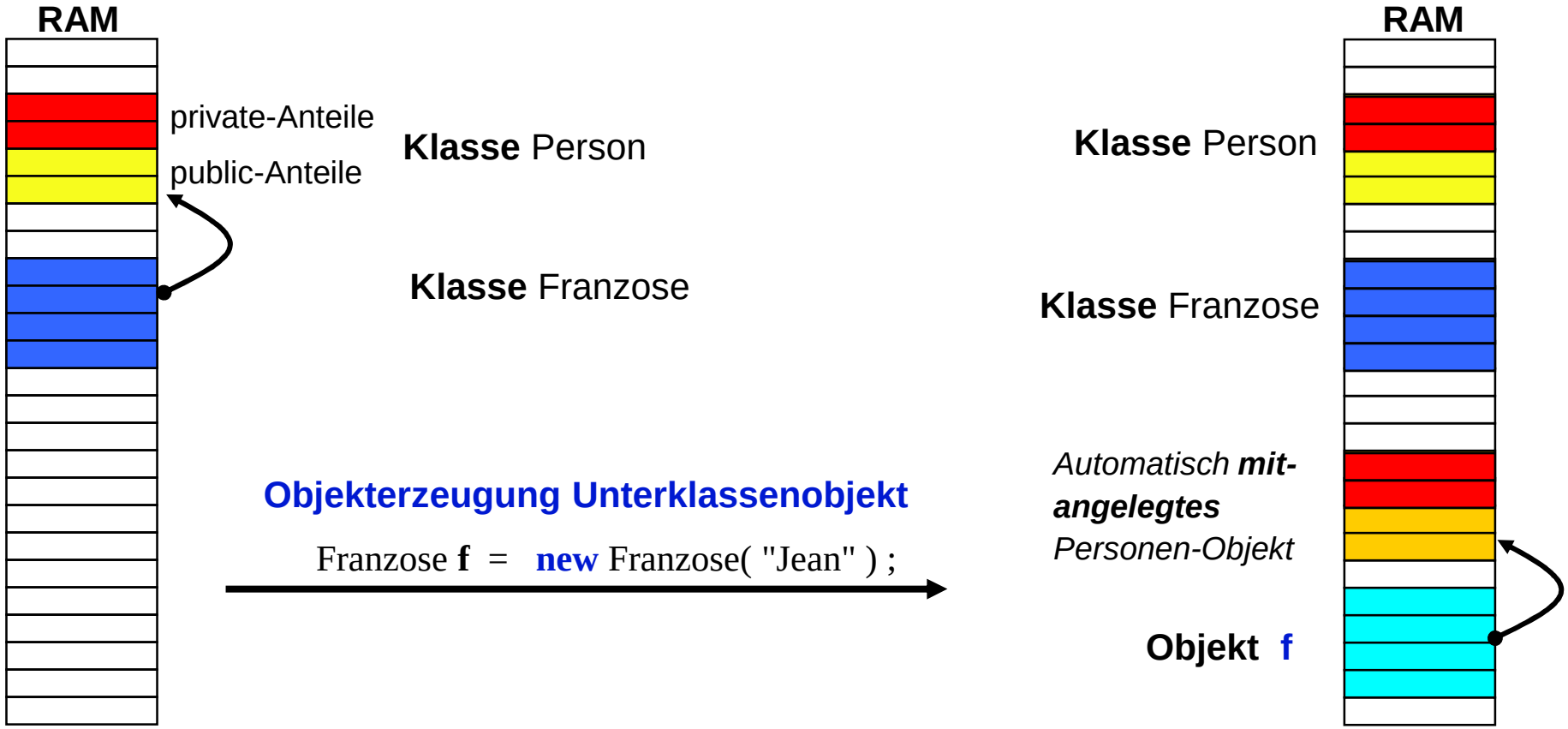


Objekterzeugung in Vererbungshierarchien

- Konstruktoren in Vererbungshierarchien
- super( ) - Regeln
- Beispiele der Techniken
- Probleme der Implementierungsverebung

Vorgang bei Erzeugen von Unterklassen-Objekten

Mit **Unterklassen-Objekt auch** *automatisch* ***alle*** **zugehörigen** Oberklassen-Objekte angelegt



Franzosen-Objekt **f** hat Zugriff auf **öffentliche** Attribute + Methoden der Klasse Person

Kann geerbte **public** Attribute **direkt** mit Werten füllen

Aber auch **private** Attribute **indirekt** mittels geerbter **public** set-/get-Methoden manipulierbar

Konstruktoren in Klassenhierarchien

Aufruf Oberklassenkonstruktor im Konstruktor der **Unterklasse** möglich – oft auch erforderlich :

super(*parameter* **) ;**

Regeln :

1. Bei Anlegen Unterklassen-Objekt wird **immer** auch **Konstruktor der Oberklasse aufgerufen** - da auch Oberklassen-Objekt erzeugt wird
2. Konstruktoren werden **nicht** vererbt ⇒ Unterklassen benötigen **eigene** Konstruktoren
3. **super( ) -** Aufruf **muss** erste Anweisung im Konstruktor der Unterklasse sein - **wenn explizit** verwendet !
4. Compiler setzt **automatisch** parameterlosen **super( )- Aufruf** ein - **wenn kein expliziter** super( ) - Aufruf in Unterklasse !
   ⇒ Aufruf parameterloser Oberklassen-Konstruktor !
5. Oberklassen-Konstruktoren ***sollten*** keine überschreibbaren Methoden aufrufen !

```
class Person  {
   private String name ;
   public Person( String nn )  {
         name = nn ;
   }
   public String getName( )  {
         return name ;
   }
}

class Franzose  extends Person  {
   public Franzose( String nn ) {
         super( nn ) ;
   }
   public String getGruss( ) {
         return "Bonjour" ;
   }
}
```

this(…) muss erste Anweisung sein ⇒
Kann **nicht** mit **super(…)** auftreten !

Konstruktoren in Klassenhierarchien

Wenn expliziter super( ) - Aufruf in Unterklasse unterbleibt wird **automatisch parameterloser Oberklassen-Konstruktor** gerufen.

Fälle :

- **Oberklasse** hat gar **keinen** Konstruktor :

Parameterloser Oberklassen-Standardkonstruktor aufgerufen – **ok** !

- **Oberklasse hat parametrisierten** Konstruktor :

Es **muss parametrisierter** Oberklassen-Konstruktor mit **super(…)** im Unterklassen-Konstruktor gerufen werden **!!**

Denn es ist **kein** parameterlosen Standardkonstruktor verfügbar !

Die Intention des Oberklassen-Entwicklers darf nicht unterlaufen werden.

Konstruktoren der Oberklasse sind zu bedienen !

Detail: Es werden bei Unterklassen-Objekt-Vernichtung **keine finalize( )-Methoden der mitangelegten Oberklassen-Objekte** aufgerufen !

```
class Ober {
   public Ober ( int val )  {
         value = val ;
   }
   private int value ;
}

class Unter extends Ober {
  public Unter ( int a , int b )  {
        super( a ) ;     // !!!!!!
        number = b ;
  }
  private int number ;
}
```

super.	**this.**
super( )	**this( )**

Überschreibpattern → **Spezialisierung : Proxy / Adapter**

```
class Tank {
   private int stand ;
   private int kapazitaet ;
   public Tank( int kap )  {
        kapazitaet = kap ;
   }
   public  int fuelle( int menge ) {
        stand = stand + menge;
        return stand ;
   }
   public int getStand( ) {
        return stand ;
   }
   public int getKapazitaet( )  {
        return kapazitaet ;
   }
}
```

```
class SicherheitsTank  extends Tank  {
   public SicherheitsTank( int kap ) {
        super( kap ) ;
   }
   @Override
   public  int fuelle( int menge ) {
        if(  getStand( ) + menge > getKapazitaet( )  )  {
             return -1 ;     //  bzw. Exception werfen !
        }
        else {
             return   super.fuelle( menge ) ;
        }
   }
}
```

Überschreiben geerbter Methoden zur **Einführung** zusätzlicher semantischer / technischer **Checks**, **Konsistenzprüfungen, Umformatierungen** etc. in Unterklasse

Vererbung **nicht nur** zur bloßen **Wiederverwendung** sondern **auch** zur **Anpassung** → **Proxy- / Adapter-Pattern**

Überschreibmechanik → **Inversion of Control**

```
class Oben {
   public void teil1( )  {
      IO.writeln( "oben 1" ) ;
   }
   public void teil2( )  {
      IO.writeln( "oben 2" ) ;
   }
   public void tusOben( ) {
      this.teil1( ) ;
      this.teil2( ) ;
   }
}
```

```
class Unten extends Oben {
   public void teil1( )  {
      IO.writeln( "unten 1" ) ;
   }
   public void teil2( )  {
      IO.writeln( "unten 2" ) ;
   }
   // …
}
```

```
class Test {
   public static void main( String[] args )  {
      Unten myU = new Unten( ) ;
      myU.tusOben( ) ;
   }
}
```

Ausgabe :
unten 1
unten 2

Möglichst **nicht** bei **Konstruktoren** !

Intendierte **Oberklassen** sollten **keine** Aufrufe öffentlicher = überschreibbarer Methoden enthalten !

Statt dessen sollten in den öffentlichen Methoden nur private = nicht überschreibbare eigene Methoden aufgerufen werden

⇓

Keine Verhaltensänderungen in der Klasse durch Überschreiben ihrer öffentlicher Methoden

Abfolge der Operationen von **tusOben( )** erhalten, aber **Details** von Unterklasse **angepasst** =

Grundidee des **Template-Patterns**

Unterklasse überschreibt geerbte Methoden, die in **anderer Methode** der Oberklasse verwendet werden. Wird ***diese*** geerbte Methode auf Unterklassenobjekt aufgerufen, so werden darin die **überschriebenen Methoden-Varianten der Unterklasse verwendet** !

Identität in Hierarchien - oder : Wer ist *this* ?

```
class Oben {
   public void test( ) {
      IO.writeln( this ) ;
   }
}
class Unten extends Oben {
   public void test( ) {
      IO.writeln( this ) ;
   }
   public void test2( ) {
      super.test( ) ;
   }
}
```

```
Unten u = new Unten( ) ;
u.test( ) ;
u.test2( ) ;
```

Aufrufe auf **Unterklassenobjekt** liefert stets Referenz aufs **Unterklassenobjekt** !

z.B. : Unten@187aeca

Auch im **Debugger** innerhalb der super-Methodenaufrufe im Oberklassenbereich sichtbar

Auch im Kontext der Oberklasse bewahren sich die Unterklassenobjekte ihre Identität und somit ihr Klassenbewußtsein !

Anmerkung : Satz stammt nicht von Marx & Engels …

Klassenbewußtsein in Hierarchien : getClass( ) ?

```
class Oben {
   public void test( ) {
      IO.writeln( this.getClass( ) ) ;
      IO.writeln( super.getClass( ) ) ;
   }
}
class Unten extends Oben {
   public void test( ) {
      IO.writeln( this.getClass( ) ) ;
      IO.writeln( super.getClass( ) ) ;
   }
   public void test2( ) {
      super.test( ) ;
   }
}
```

```
Unten u = new Unten( ) ;
u.test( ) ;
u.test2( ) ;
```

Aufrufe auf **Unterklassenobjekt** liefert **stets** Namen der **Unterklasse** :

class Unten

Na also :

Die Unterklassenobjekte bewahren somit ihr Klassenbewußtsein … !

Probleme Vererbung

Implementierungsvererbung *schwächt im Grunde das* ***Prinzip der Kapselung***

1. Unterklassen an Oberklassen gekoppelt :

Vorsicht bei Weiterentwickeln Oberklasse !

Unterklasse erbt Schnittstelle + Implementierung der Oberklasse ⇒ Invalidierungsgefahr :

Änderung von public + protected-Elementen der Oberklasse wirkt sich auf Unterklassen aus !

Nur Unterklasse kennt ihre Oberklassen + ist auf diese angewiesen - nicht umgekehrt ⇒

Oberklassen-Entwickler merkt nicht sofort, wenn invalidierende Änderung erfolgt !

Bsp : Oberklasse wird neue public-Methode hinzugefügt, die sich nur im return-Typ von Methode unterscheidet, die bereits in Unterklasse existiert …

2. Unterklasse erbt von Oberklasse deren Fehler + Schwächen :

Fehler + Schwächen der public-Schnittstelle der Oberklasse wandern auch in Unterklassen …

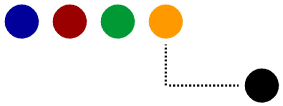
… und werden dort zu Fehlern + Schwächen der public-Schnittstelle der Unterklassen !

Dagegen lässt sich bei Assoziation (Attribute vom Typ der Oberklasse) eine Schnittstelle entwerfen, hinter der Fehler und Schwächen verborgen werden können

⇒ Eine Klasse als potenzielle Oberklasse zu entwerfen zwingt zu sauberem Design !

⇒ Wenn Klasse nicht dafür geeignet, dann Vererbung von ihr verhindern = **finale** Klasse, s.u.

Vertiefung Objektorientierung Polymorphie



Typkompatibilität in Vererbungshierarchien

- "Ist-ein"-Prinzip :

  Unterklassen als Vertreter ihrer Oberklassen
- Upcast versus Downcast
- Zugreifbarkeitsregeln
- Operator instanceof
- Generik durch Polymorphie
- Semantik : Lishkov-Substitution-Principle

Wir haben ein massives SWE-Problem : Jede Klasse ist ein eigener Typ ⇒

Wir ertrinken gerade in Klassen-Typen. Jeder Typ ist nicht-kompatibel zu jedem anderen Typ. Jeder erfordert seine eigenen Methoden + Datenstrukturen.

Wie können wir gemeinsame Strukturen / Behälter / Frameworks zur Vfg stellen ???

Polymorphie *Vielgestaltigkeit*

Gleichnamige Methodenaufrufe für Objekte **verschiedener** Klassen einer Vererbungs-hierarchie liefern **verschiedenes Verhalten**

Aufruf der Methode **getGruss( )** liefert je nach Objekttyp **unterschiedliches** Verhalten !

JVM stellt bei Aufruf fest, zu **welchem Typ** die Methode gehört + führt **dessen** Coding aus.

```
class Person {
  // …
  public String getGruss( )   {  return "Hallo" ;  }
}
class Franzose  extends Person {
   // …
   public String getGruss( ) {  return "Bonjour" ;  }
}
class Deutscher  extends Person {
   // …
   public String getGruss( )   {  return "Mahlzeit" ;  }
}
class Bayer  extends Deutscher {
   // …
   public String getGruss( )   {  return "Grüß Gott" ;  }
}
```

Unterklassen **überschreiben** geerbte **getGruss()-Methode** durch unterklassenspezifische Variante

… jedoch noch viel tiefgreifendere Bedeutung …

Polymorphie + Typbeziehung

Fundamentale **Typkompatbilität :**

Semantisch :

Ausnutzen ***Ist-ein*** Beziehung in Vererbungshierarchie

Jedes **Unterklassen**-Objekt ***ist auch ein spezieller*** Vertreter der **Oberklasse** $\Rightarrow$

Kann diese überall ***vertreten***, wo ein Oberklassenobjekt gefordert wird

Technisch :

Unterklassentyp ist ***typkompatibel*** mit **Oberklassentyp** !

```
Person p ;
Bayer b  =  new Bayer( "Sepp" ) ;
p = b ;      // zulässige Zuweisung !  →  Upcast
```

Jeder Bayer ist eine Person … jedoch nicht umgekehrt !

Vererbung : Typkompatibilität Ober- und Unterklasse

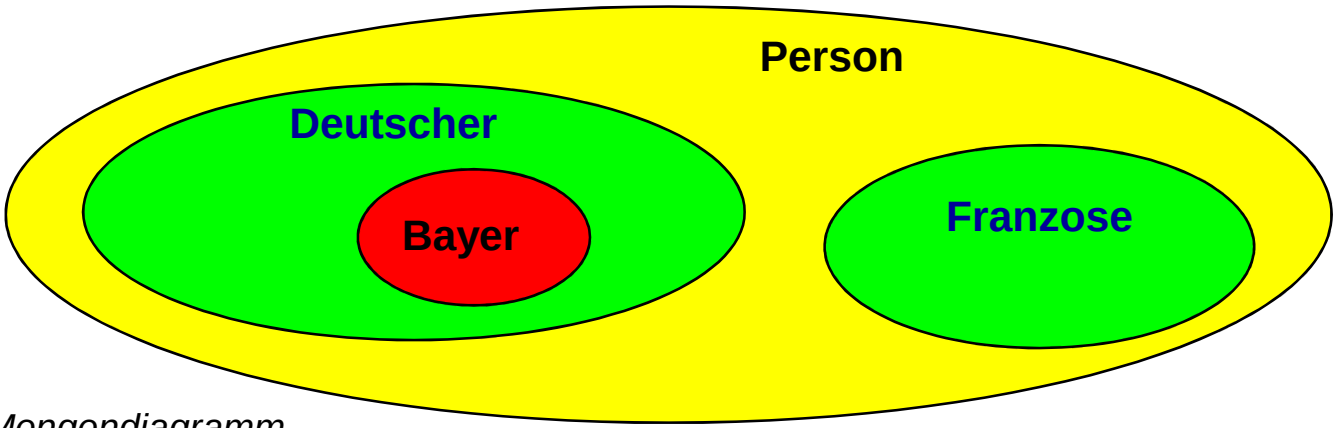
Nicht nur Wiederverwendung von Code durch **Implementierungsvererbung** - sondern auch :

Nutzung **Typkompatibilität Unterklasse zur Oberklasse** : Ist-Beziehung - **Typvererbung**

Jedes "Programm", das in der Lage ist, mit Objekten der Oberklasse zu arbeiten, kann auch mit Objekten aller Unterklassen arbeiten !

Konsequenz : "Alles*) läuft" auch mit Objekten spezialisierter Unterklassen ohne Anpassung !

***)** Programme, Datenbehälter, Frameworks, Systeme …



Mengendiagramm

"Ist ein"-Beziehung :

is_a

is_kind_of

Typkompatibilität

Strenge Typprüfung !

```
double x = 3.3 ;    int y = x ;    //  Fehler!
```

Zuweisung erlaubt, wenn Typ-Anforderungen erfüllbar - automatische, implizite Typumwandlung :

```
int x = 15 ;     double y = x ;    //  erlaubt!
```

Objektreferenzen analog !

Automatische Typumwandlung bei Zuweisung an Objektreferenz aus höherer Stufe der Klassenhierarchie = **UPCAST** :

```
Person p1 = new Franzose( "Jean" ) ;
```

Explizite Typumwandlung aber erforderlich bei Zuweisung an Objektreferenz tieferer Stufe der Klassenhierarchie = **DOWNCAST** :

```
Bayer b = (Bayer) p2 ;
```

```
class GrussAusgabe2  {

   public static void main( String[] args) {

       Franzose f  = new Franzose( "Jean" ) ;
       Person p1 = f ;     // Upcast !

       Person p2 = new  Bayer( "Sepp" ) ;

       Bayer b = (Bayer)  p2 ;      // Downcast !
        b.setLieblingsbier( "Paulaner" ) ;
   }
}
```

Upcast ist immer **unproblematisch** :

Ein Franzose **ist immer** auch eine Person !

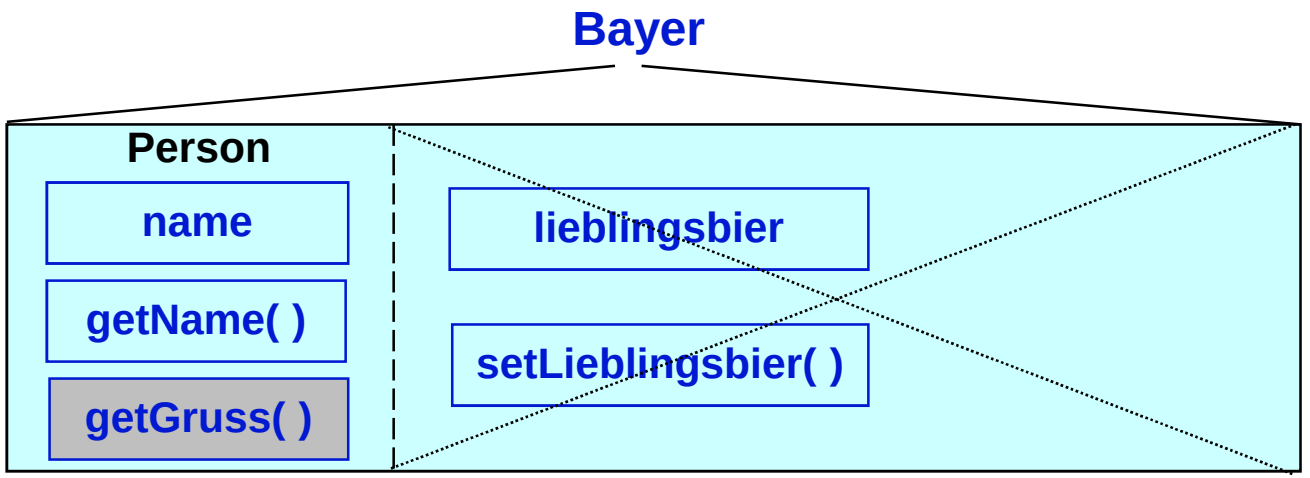
Vom Compiler **ohne** expliziten Cast **akzeptiert** !

Downcast immer **problematisch** + fehleranfällig :

Eine Person **ist nicht immer** ein Bayer !

Vom Compiler **nur** durch **expliziten** Cast akzeptiert !

Typkompatibilität : **Upcast** ***(safe cast)***




```
Bayer b = new Bayer( "Sepp" ) ;

Person p = b ;     // Upcast
```

Korrekt ! ✓

Verlangt : **Person-Objekt** Geliefert : **Bayer-Objekt**

⇒ **"Mehr"** als "verlangt" - aber jedenfalls "**nicht zuwenig**" !

⇒ JVM nimmt, was benötigt wird + "ignoriert" / "verbirgt" das "Überflüssige" **?????**

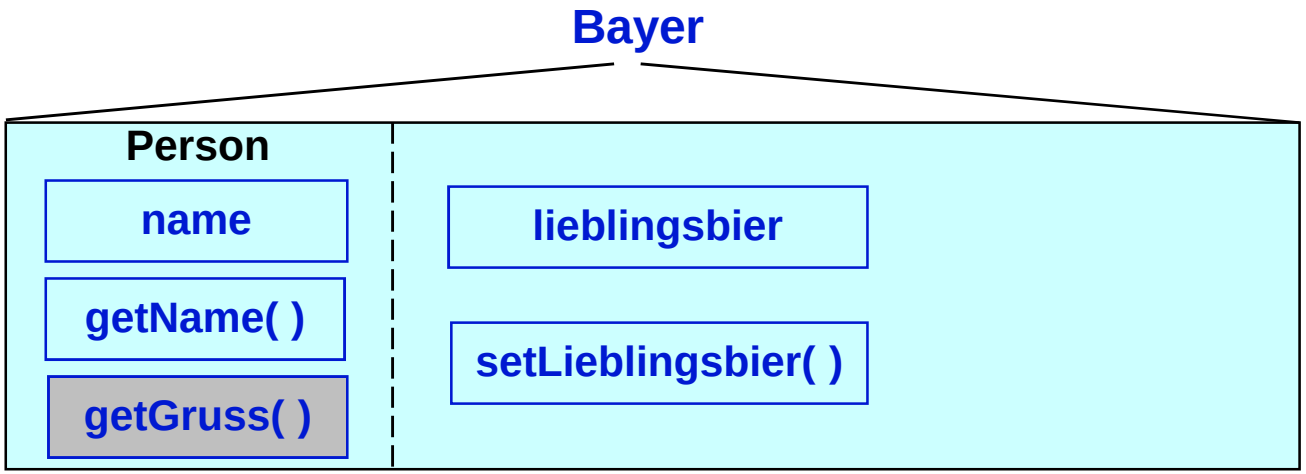
⇒ Macht aus Bayer-Objekt eine Person, "reduziert" Bayer-Objekt auf Person-Objekt

Genug vorhanden, um Person-Objekt korrekt zu instanziieren

Durch "Weglassen" kann man aus Bayern eine bloße Person machen …

⇒ Zuweisung ist **korrekt** und passiert die Typprüfung !

Typkompatibilität : Downcast (*unsafe cast*)




```
Person p = new Person( "Paula" ) ;

Bayer b = (Bayer) p ;   // Downcast
```

Fehler ! ≠

Verlangt : **Bayer-Objekt** Geliefert : **Person-Objekt**

⇒ **"Weniger"** als "verlangt" – Compiler wird durch **cast** gezwungen !

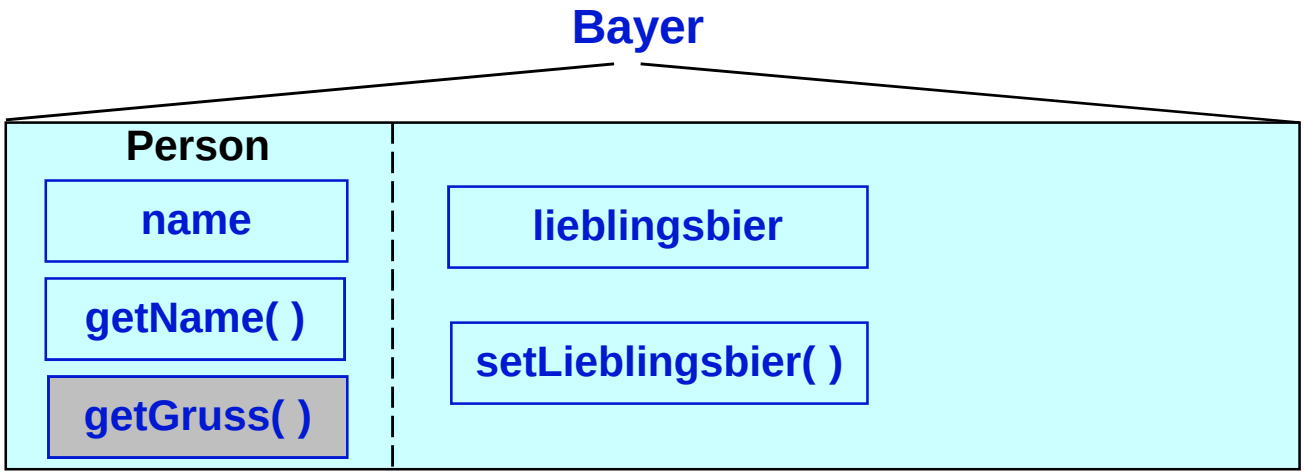
⇒ Das zugewiesene Person-Objekt enthält "zu wenig", um Bayer-Objekt zu "füllen"

⇒ **Nicht "genug" im Person-Objekt, um Bayer-Referenz korrekt zu instanziieren**

Aus bloßer Person läßt sich kein Bayer machen - es "fehlt" etwas ! ...

⇒ Zuweisung **nicht korrekt** ⇒ **ClassCastException** zur **Laufzeit** !

Typkompatibilität : Downcast




```
Bayer b1 = new Bayer( "Sepp" ) ;

Person p = b1 ;   // Upcast

Bayer b2 = (Bayer) p ;   // Downcast
```

Hier : Korrekt ! ✓

Verlangt : **Bayer-Objekt** Geliefert : **Personen-Referenz auf Bayer-Objekt**

⇒ ***Aus dieser speziellen*** Personen-Referenz kann Bayer-Objekt wieder "hervorgeholt" werden

⇒ **Genug "Inhalt" *hinter dieser speziellen* Person-Referenz, um Bayer zu instanziieren**

⇒ Zuweisung **korrekt** - Typkonform zur Laufzeit !

Upcast: unkritisch, direkte Zuweisung

Downcast: kritisch, explizite cast-Operation

Zugreifbarkeits-Regeln

UPCAST :

Person **p2** = new Bayer("Sepp") ;

Auf **p2** **nur** Methoden + Attribute angesprechbar, die **schon in Klasse Person bekannt** sind !

Werden **Methoden** in Klasse Bayer überschrieben, dann wird jedoch **deren** Version gerufen !

Denn : **p2** ist Referenz vom Typ Person, nicht Referenz vom Typ Bayer !

Bayer-**spezifische** Methoden + Attribute **sind nicht** zugänglich !

DOWNCAST :

(Bayer) p2 *oder* Bayer b = (Bayer) p2 ;

Cast von **p2** zu Typ Bayer ⇒

Somit ***nun*** Methoden + Attribute zugreifbar, die **spezifisch** für Klasse Bayer sind - dh in Oberklasse Person nicht existieren.

```
class GrussAusgabe2 {

   public static void main( String[ ] args) {

   Person p1 = new Franzose( "Jean" ) ;
   Person p2 = new Bayer( "Sepp" ) ;

    IO.writeln( p2.getGruss( ) ) ;   // ok !

    p2.setLieblingsbier( "Paulaner" ) ;   // Fehler !!

    ( (Bayer) p2 ).setLieblingsbier( "Paulaner" ) ;


     Bayer b1 = (Bayer) p2 ;     // Downcast!
     b1.setLieblingsbier( "Erdinger" ) ;      // ok !
     }
}
```

Operator instanceof

Kritischer Downcast :

Nur per **expliziter** cast-Operation

Bayer b = **(Bayer)** p2 ; // korrekt **??**

Wenn **p2** nicht auf Bayer-Objekt zeigt, wird **Laufzeitfehler** ausgelöst !

Laufzeit-Typ-Prüfung mit Operator :

p2 instanceof Bayer

⇓

Prüfung, ob **p2** zur **Laufzeit** wirklich Objekt vom Typ Bayer referenziert

Ein Programm ist **typsicher**, wenn schon zur Compilezeit feststeht, dass kein Aufruf an ein Objekt erfolgt, für das das Objekt keine entsprechende Methode besitzt.

Eine Programmiersprache ist typsicher, wenn alle erzeugbaren Programme typsicher sind – so dass keine LZ-Fehler durch falsche Datentyp-Operationen möglich sind – **es sei denn**, dies wird durch Casts erzwungen.

```
class Zuweisung {
   public static void main( String[] args) {

      Franzose f1 = new Franzose( "Jean" ) ;
      Bayer b1 = new Bayer( "Sepp" ) ;

      Person p1, p2 ;

      p1 = f1 ;      // Upcast - ok!
      p2 = b1 ;      // Upcast - ok!

      if ( p2 instanceof Bayer ) {
            // Downcast möglich - ok!
            Bayer b = (Bayer) p2 ;
            b.setLieblingsbier( "Paulaner" ) ;
      }
      else // …
   }
}
```

Compiliert aber nur, wenn beteiligte Typen zur **selben Klassenhierarchie** gehören :

if(**p2 instanceof String**) { /* … */ }

⇒ Compilerfehler

Unvermeidbarkeit des Downcast

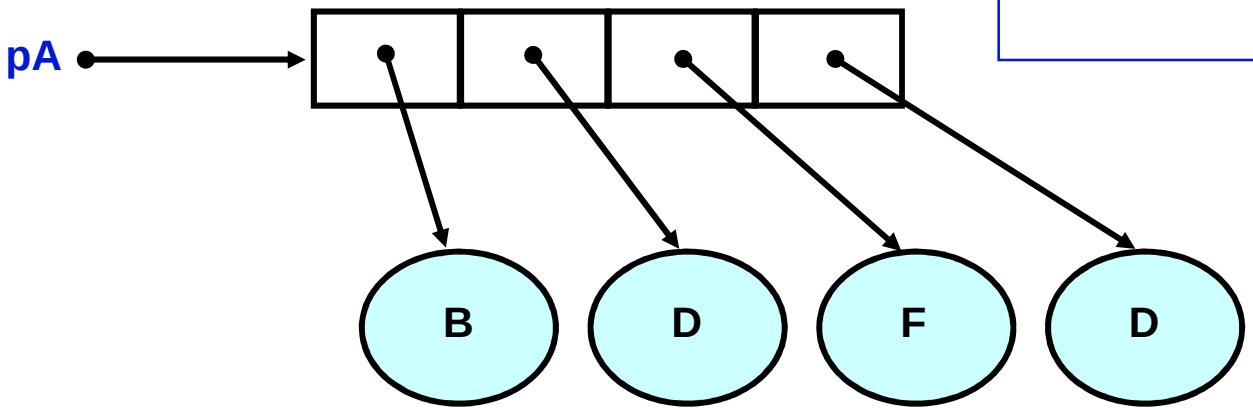
Das **Einfügen** von Unterklassen-Objekten in Datenbehälter ist ein **Upcast**

Jedoch erfordert ihre **"Wiedergewinnung"** als Unterklassen-Objekt einen **Downcast**

Ziel dabei :

Zugriff auf **unterklassen-spezifische** Anteile

```
Person[ ] pA = new Person[ 4 ] ;
// Reinstellen = Upcast
pA[ 0 ] = new Bayer( "Sepp" ) ;
pA[ 1 ] = new Deutscher( "Hans" ) ;
// …
// Rausholen als spezifischer Typ = Downcast :
Bayer b = (Bayer) pA[ 0 ] ;
b.setLieblingsbier( "Paulaner" ) ;
```



Wie erhält man **nur** die **Deutschen**-Objekte ?

Was sagt ein **Bayer** zu **instanceof Deutscher** ?

```
if(  pA[i]  instanceof  Deutscher  &
     !pA[i]  instanceof  Bayer  ) {  /* … */  }
```

Generik durch Polymorphie

Verwendung von Unterklassen-Referenzen anstelle von Oberklassen-Referenzen :

- Bei **Zuweisungen (s.o.) - auch an Datenbehälter :**

```
Person[ ] pArr = new Person[3] ;
pArr[1] = new Bayer("Sepp") ;
```

- Bei **Parameterübergaben :**

Wenn **Methode** als Parameter eine Referenz vom Typ Person erwartet ...

... dann kann auch eine Referenz vom Typ Deutscher, Franzose, Bayer … d.h. Referenz auf **Unterklassen-Objekte** übergeben werden !

Unterklassen-Objekte / -Referenzen sind typkompatibel zu Oberklassen-Objekten / -Referenzen !!

```
class Zuweisung {
   public static void main( String[ ] args ) {

      Bayer b1 = new Bayer( "Sepp" ) ;
       ausgabe1( b1 ) ;   // Korrekt !!!!


      Person p1 = new Person( "N.N." ) ;
       ausgabe2( p1 ) ;    // Fehler!

   }

   public static void ausgabe1( Person p ) {
      IO.writeln(  "Hallo" + p.getName( )  ) ;
    }

    public static void ausgabe2( Bayer b ) {
      IO.writeln( "Hallo" + b.getName( ) ) ;
    }
}
```

Polymorphie

Verwendung :

Methoden schreiben, die mit **Typ** der **Oberklasse** arbeiten, zB :

```
void ausgabe( Person p ) {
    IO.writeln(  p.getGruss()  ) ;
}
```

+

Methoden**aufrufe** mit Objektreferenzen **typkompatibler Unterklassen**

Beispiel :

Klasse Person enthält Methode **getGruss( ) .**

Diese wird in jeder der Unterklassen *spezifisch* **überschrieben**.

```
class Person {
   public Person( String nn)    {  name = nn ; }
   public String getName( )     {  return name ;  }
   public String getGruss( )    {  return "Hallo" ;  }
   private String name ;
}

class Franzose  extends Person {
   public Franzose( String nn ) {   super( nn ) ;  }
   public String getGruss( ) {  return "Bonjour" ;  }
}

class Deutscher  extends Person {
   public Deutscher( String nn )    {  super( nn  ) ;  }
   public String getGruss( )    {  return "Mahlzeit" ;  }
}

class Bayer  extends Deutscher {
   public Bayer( String nn )    {  super( nn ) ;  }
   public String getGruss( )    {  return "Grüß Gott" ;  }
}
```

Polymorphie

Klasse Person enthält Methode **getGruss( )** - **wird** in Unterklassen **überschrieben**.

Dadurch kann **polymorph** auf Objekten der **Unterklassen** die Methode getGruss( ) aufgerufen werden :

Methode **ausgabe( )** ist für Objekte vom Typ **Person** definiert. Da deren Unterklassen typkompatibel sind, ist diese Methode auch mit allen **Unterklassen-Referenzen** aufrufbar!

JVM führt dabei Methode getGruss( ) der übergebenen Objekte aus ⇒

Es wird die spezifische getGruss( )- Methode der übergebenen Unterklassen-Objekte ausgeführt !

Fall bei nicht-statischen Methoden !

```
class GrussAusgabe2 {
   public static void main( String[] args ) {

      Franzose f = new Franzose( "Jean" ) ;
      Deutscher d = new Deutscher( "Hans" ) ;
      Bayer b = new Bayer("Sepp" );

      // polymorphe Aufrufe:
      ausgabe( f ) ;
      ausgabe( d ) ;
      ausgabe( b ) ;
   }

   public static void ausgabe( Person p ) {
         IO.writeln( p.getGruss( ) ) ;
    }
}
```

Voraussetzung : Klasse Person muss eine überschreibbare Methode getGruss( ) besitzen

Polymorphie

Typkompatible Unterklassenobjekte
instanziiert

Ausführung erfolgt auf
Unterklassenobjekten :

*zur **Laufzeit***

Deren klassenspezifische **Methode**
getGruss( ) wird ausgeführt !

Methodenaufruf für Objekte
vom **Typ Person** formuliert

Compilezeit

```
class GrussAusgabe2  {

   public static void main( String[] args) {

      Franzose f = new  Franzose( "Jean" ) ;
      Deutscher d = new  Deutscher( "Hans" ) ;
      Bayer b = new  Bayer( "Sepp" ) ;

       // polymorphe Aufrufe:
       ausgabe( f ) ;
       ausgabe( d ) ;
       ausgabe( b ) ;
    }

    public static void ausgabe( Person p ) {
            IO.writeln( p.getGruss( )  ) ;
     }
}
```

Vorsicht : Wird geerbte Methode in Unterklasse **überladen statt überschrieben**, dann wird die geerbte **Methode der Oberklassen** gerufen !

z.B. : String getGruss(**String s**) { … }

Polymorphie

Methoden mit **Ober**klassenreferenzen als Parameter - werden mit **Unter**klassen-Objektreferenzen aufgerufen.

Resultat :

Je nachdem, welche Art von speziellem Unterklassenobjekt man beim Aufruf übergibt, **verhält sich ein und dieselbe Methode anders** dh : **vielgestaltig = polymorph** :

Hier : Methode **ausgabe(Person p)**

Für Objekte vom Typ der Klasse **Person** geschrieben

Aufgerufen mit Objekten vom Typ der Unterklassen von Person : ⇒

Ergebnis der Aufrufe 1, 2, 3 :

Jedesmal anderes Verhalten - dank Überschreiben !

Bonjour, Mahlzeit, Grüß Gott

```
class GrussAusgabe2 {
   public static void main( String[] args) {

      Franzose f = new  Franzose( "Jean" ) ;
      Deutscher d = new  Deutscher( "Hans" ) ;
      Bayer b = new  Bayer( "Sepp" ) ;

      // Drei polymorphe Aufrufe
      // derselben Methode ausgabe :
      ausgabe( f ) ;    // 1. liefert:  Bonjour
      ausgabe( d ) ;    // 2. liefert:  Mahlzeit
      ausgabe( b ) ;    // 3. liefert:  Grüß Gott
   }

   public static void ausgabe( Person p ) {
         IO.writeln( p.getGruss( ) ) ;
    }
}
```

Polymorphie : Generik + Semantik

Jedes "Programm" , das mit Objekten der Oberklasse arbeitet,

kann auch mit Objekten der Unterklasse arbeiten !

Deshalb auch keine Einschränkung der Sichtbarkeit beim Überschreiben !

Liskov Substitution Principle : ***Einhalten des "Kontrakts" der Oberklassen***

"Subclasses must be usable through the superclass interface without the need for the user to know the difference" ***(Einhalten der semantischen Integrität)***

Unterklassen sollen **Semantik** = **inhaltlichen Sinn bewahren** !

Unterklasse ist **Spezialisierung** ihrer Oberklasse ⇒ soll ihre Oberklasse ***sinnvoll*** **vertreten** können !

Unterklassen sollen sich in **überschriebenen** Methoden **semantisch** **verhalten** wie ursprüngliche Oberklassen-Methoden. Änderung der Implementierung soll **Vertrag nicht verletzen** !

Objekte vom Typ Deutscher, Franzose **verhalten** sich ***prinzipiell*** wie Objekte vom Typ Person :

⇒ Methode **getGruss( )** liefert auch bei Ihnen einen **speziellen** Gruß zurück

Pacta sunt servanda **: Das durch die Basisklasse zugesicherte Objektverhalten soll auch in ihren Unterklassen-Spezialisierungen eingehalten werden !**

Vertiefung Objektorientierung Polymorphie

Umgang mit polymorphem Code

- Statischer und Dynamischer Typ
- Späte Bindung
- Standardfall Überschreiben nicht-statischer Methoden
- Sonderfälle Verbergen :

  statische Methoden + Attribute
- Regeln für Überladene Methoden
- Überschreiben mit Variation Rückgabetyp

Statischer + Dynamischer Typ

Objektvariablen referenzieren Objekte verschiedenen Typs

Statischer Typ :

- Fest **Deklarierter Typ** der Objektvariablen
- bestimmt, welche Methoden + Attribute via Variable

  überhaupt ansprechbar sind

Dynamischer Typ :

- Typ des zur **Laufzeit** referenzierten Objekts
- durch Zuweisungen zur LZ **jederzeit änderbar !**
- bestimmt, **welche Methoden** aufgerufen werden

p1 und **p2** haben statischen Typ **Person** ⇒
Über **p1**, **p2** **nur** Methoden + Attribute ansprechbar, die in Klasse Person deklariert sind

p1 hat nach Zuweisung dynamischen Typ **Franzose**
p2 hat nach Zuweisung dynamischen Typ **Bayer**

⇒ **p1.getGruss( )** → getGruss() von Franzose

p2.getGruss( ) → getGruss() von Bayer

```
class Zuweisung  {
  public static void main( String[] args ) {

        Person  p1, p2 ;

        p1  =  new  Franzose( "Jean" ) ;
        p2  =  new Bayer( "Sepp"  ) ;

        p1.getGruss( ) ;
          // ruft Methode  getGruss( )
          // der Klasse Franzose

        p2.getGruss( ) ;
          // ruft Methode  getGruss( )
          // der Klasse Bayer
   }
}
```

Dynamische Bindung von Methodenaufrufen:

*Aufruf **obj.m()** führt zum Aufruf der **m()**-Methode, die zum **dynamischen** Typ von **obj** gehört !*

*Bei **nicht**-statischen Methoden !*

Späte Bindung nicht-statische Methoden

Statischer Typ : **Deklarierter** **Objekttyp** der Variablen **p**

Dynamischer Typ : **Objekttyp**, den Variable zur **LZ** referenziert - **zur LZ** änderbar !

⇓

Erst zur **LZ** steht **aktueller dynamischer Typ** fest.

Auswahl auszuführender nicht-statischer *Methoden* erst zur **LZ dynamisch** anhand des **aktuellen dynamischen Typs** der Objektvariable …

… und nicht schon statisch zur Compilezeit statisch anhand des statischen Typs der Objektvariablen

⇓

Auszuführendes *Methoden-Coding* nicht schon zur Compilezeit festgelegt - sondern erst zur Laufzeit :

Dynamische, späte Bindung (late binding) ist technische Voraussetzung für Polymorphie

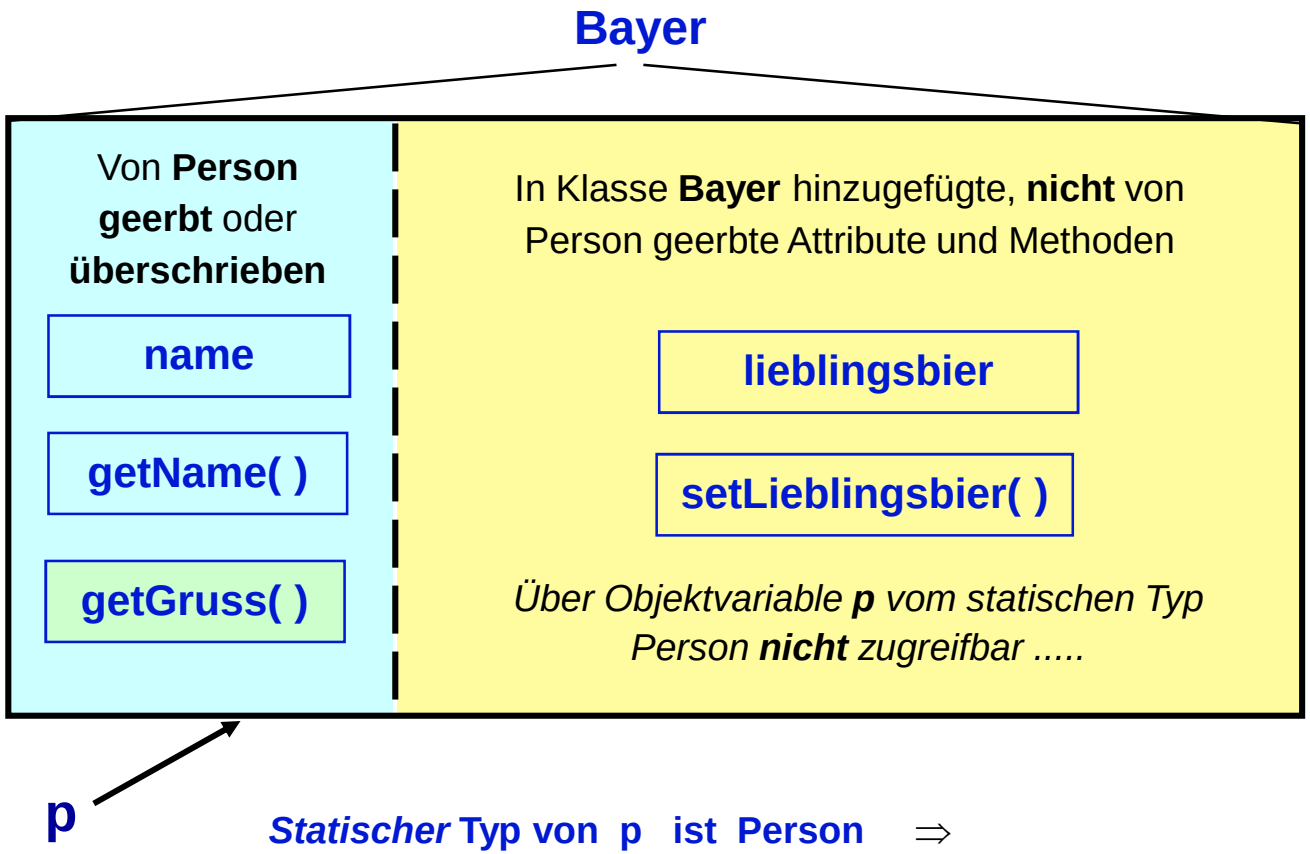
```
class Zuweisung  {
  public static void main( String[ ] args ) {

     Person p ;   // statischer Typ Person

     char c = IO.promptAndReadChar( "f/b?" ) ;
     // Festlegung dynamischer Typ gemäß
     // User-Eingabe erst zur Laufzeit !!
     if( c == ' f ' ) {
        p  = new  Franzose( "Jean" ) ;
      }
      else {
        p  = new Bayer( "Sepp"  ) ;
       }

       p.getGruss( ) ;
       // welche Methode  getGruss( )  gerufen wird
       // steht erst zur Laufzeit fest !
  }
}
```

Statischer / Dynamischer Typ nicht-statische Methoden




```
Person p  = new Bayer( "Sepp" ) ;
IO.writeln(  p.getGruss( )  ) ;
```

Statischer Typ :

Deklarierter Variablen-Typ - bestimmt, **welche** Methoden + Attribute **ansprechbar** sind

Dynamischer Typ :

Typ des Objekts, auf den Variable zur **Laufzeit** zeigt - bestimmt, welche Methoden tatsächlich aufgerufen werden

***Statischer* Typ von p ist Person** ⇒

Nur schon in Klasse Person deklarierte Attribute und Methoden sind ansprechbar !

***Dynamischer* Typ von p ist Bayer** ⇒

Die an Klasse Bayer **vererbten** und dort evtl. **überschriebenen** Methoden werden über **p** angesprochen !

Statischer / Dynamischer Typ

Statische Methoden

*Wenn Unterklasse **statische Methoden "überschreibt"** (genauer : **verbirgt = hides**), entscheidet der deklarierte **statische Typ** der Objektvariable über Methodenaufruf* ⇒

*Entscheidung über **statischen Methoden-Zugriff** fällt **statisch** zur Compile-Zeit mittels Referenz-Typ* ***!!***

Methode use( ) für Objekte vom Oberklassentyp Auto.

Verlässt sich in Implementierung auf Methode **m( )** .

Aufrufbar auch mit Unterklasse-Objekten Typ PKW.

Jedoch wird stets statische Methode der Oberklasse gerufen, da statischer Referenz-Typ des Methodenparameters vom Typ Auto ist !

⇓

Keine Methoden statisch deklarieren, für die man Polymorphie nutzen möchte !

```
class Auto {
  public static void m( ) {
     IO.writeln( "Auto" ) ;
  }
}

class PKW extends Auto {
  public static void m ( ) {      //  "verbirgt"
     IO.writeln( "PKW" ) ;
  }
}

class AttributTest  {
  public static void main( String[] args ) {
    PKW  p = new PKW( ) ;
    use(  p  ) ;     // Ausgabe :  "Auto"   !!
  }

  public static void use( Auto a ) {
    a.m( ) ;       //  Typ Referenz entscheidend !
    //  denn: statisches Verhalten hängt nicht
    //  vom individuellen Objekt ab.
    //  besser also :   Auto.m( ) ;
  }
}
```

Statischer / Dynamischer Typ

Attribute

*Wenn Unterklasse geerbte **Attribute** **"überschreibt"** (**verbirgt = hides**), entscheidet deklarierter **statischer Typ** der Objektvariable, welches **Attribut** verwendet wird* ⇒

*Entscheidung über **Attribut-Zugriff** fällt zur Compile-Zeit mittels Referenz-Typ* ***!!***

Methode use( ) für Oberklassentyp Auto.

Verlässt sich in Implementierung auf Attribut **wert** vom Typ **int**.

Aufrufbar auch mit U.kl.-Objekten vom Typ PKW.

Soll aber nicht invalidiert werden durch in Unterklasse PKW überschriebenes Attribut vom inkompatiblen Typ boolean !!

⇓

Durch o.a. Regel kann Unterklasse PKW doch den Kontrakt der Oberklasse Auto erfüllen !

```
class Auto {
  public int wert = 10 ;
  public void m( ) {
     IO.writeln( "Auto: " ) ;
  }
}

class PKW extends Auto {
  public boolean wert = false ;    //    "verbirgt" !
  public void m ( ) {
     IO.writeln( "PKW " ) ;
  }
}

class AttributTest  {
  public static void main( String[] args ) {
    PKW  p = new PKW( ) ;
    use(  p  ) ;       // Upcast
  }

  public static void use( Auto a ) {
    a.m( ) ;                      //  Typ Objekt entscheidend !
    int x = 100 + a.wert ;        //  Typ Referenz entscheidend !
  }
}
```

*Auch bei **Verbergen** mit **gleichem Typ** oder mit private Attribut wird Oberklassenwert verwendet !*

Zugriff Methoden + Attribute

Oberklasse				
	Nicht-statische Methode	***Statische Methode*** (verbergen)	***Nicht-statisches Attribut***	***Statisches Attribut***
	Überschreiben / Verbergen ohne Einschränkung Sichtbarkeit		**Verbergen** auch mit Einschränkung Sichtbarkeit	
Unterklasse	super-Zugriff	Kein super-Zugriff in statischen Methoden *)	super-Zugriff	Kein super-Zugriff *)
	Typ referenziertes Objekt (dynamischer Typ) entscheidet über Methodenauswahl	**Typ der Referenz / Objektvariablen (= statischer Typ)** entscheidet über Methodenauswahl und Attributzugriff		

relevantester Fall !

***)** Compiler :

Cannot use super in a static context !

Überladene Methoden

Überladen bei **Methoden mit Objektparametern**

Regel : Typ der *Referenz* entscheidet ! *)

⇓

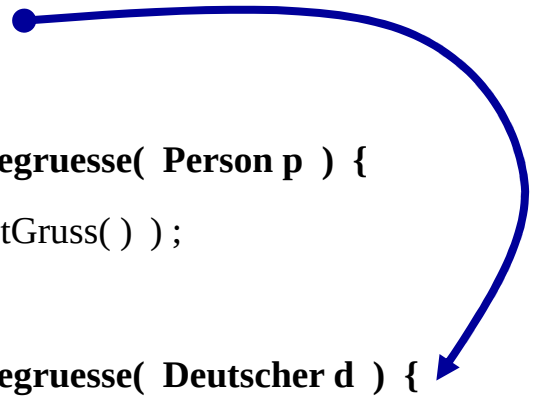
1. Exakt *passende* Methode wird aufgerufen
2. Wenn keine Methode exakt passt, dann **möglichst spezifische Methodenauswahl**

⇓

Aufruf mit **Bayer-Referenz** führt zum Aufruf der Methode mit **Parameter vom Typ Deutscher**, nicht zum Aufruf der "allgemeineren" Methode mit Parameter vom Typ Person

Bei Methoden mit mehreren Objektparametern sind Zweideutigkeiten möglich - die Compiler moniert

```
class GrussAusgabe  {
  public static void main( String[] args ) {
    Person p = new Person( "Jemand" ) ;
    Deutscher d = new Deutscher( "Hans" ) ;
    Bayer b = new Bayer( "Sepp" ) ;
    begruesse( p ) ;
    begruesse( d ) ;
    begruesse( b ) ;
  }
  public static void begruesse(  Person p  ) {
      IO.writeln(  p.getGruss( )  ) ;
  }
  public static void begruesse(  Deutscher d  ) {
      IO.writeln(  "Hier: " + d.getGruss( )  ) ;
  }
}
```



) Wenn Variable **p zB auf **Deutscher**-Objekt zeigt, wird doch Variante für Typ **Person** gerufen*

Überschreiben

Flexibilität bei Rückgabetyp

```
class Person  { /*…*/ }
class Deutscher extends Person { /* … */ }
```

Beim **Überschreiben** darf Typ der **Parameter nicht** verändert werden – sonst würde gererbte Methode *überladen* statt überschrieben !

⇓

Auch **kein** typkompatiber **Unterklassentyp bei Parametern** verwendbar !

```
class Amt  {
  void verwalte( Person p ) { /* … */ }
}
class EinwohnerMeldeAmt extends Amt {
  @Override    // Fehler – kein Überschreiben !
  void verwalte( Deutscher d ) {  /* … */  }
}
```

Beim **Überschreiben darf** der **Rückgabetyp** ein typkompatiber **Unterklassentyp** sein !

⇓

Überschreibende Methode muss mit exakt **gleichen Parametern** wie geerbte Oberklassen-Methode arbeiten …

… aber sie **darf** anstelle Oberklassen-Referenzen auch **typkompatible** Unterklassen-Referenzen **zurückgeben**.

```
class Amt  {
  Person verwalte( String name, … ) { /* … */ }
}
class EinwohnerMeldeAmt extends Amt {
  @Override    // OK – zulässiges Überschreiben!
  Deutscher verwalte( String name, … ) { /* … */ }
}
```

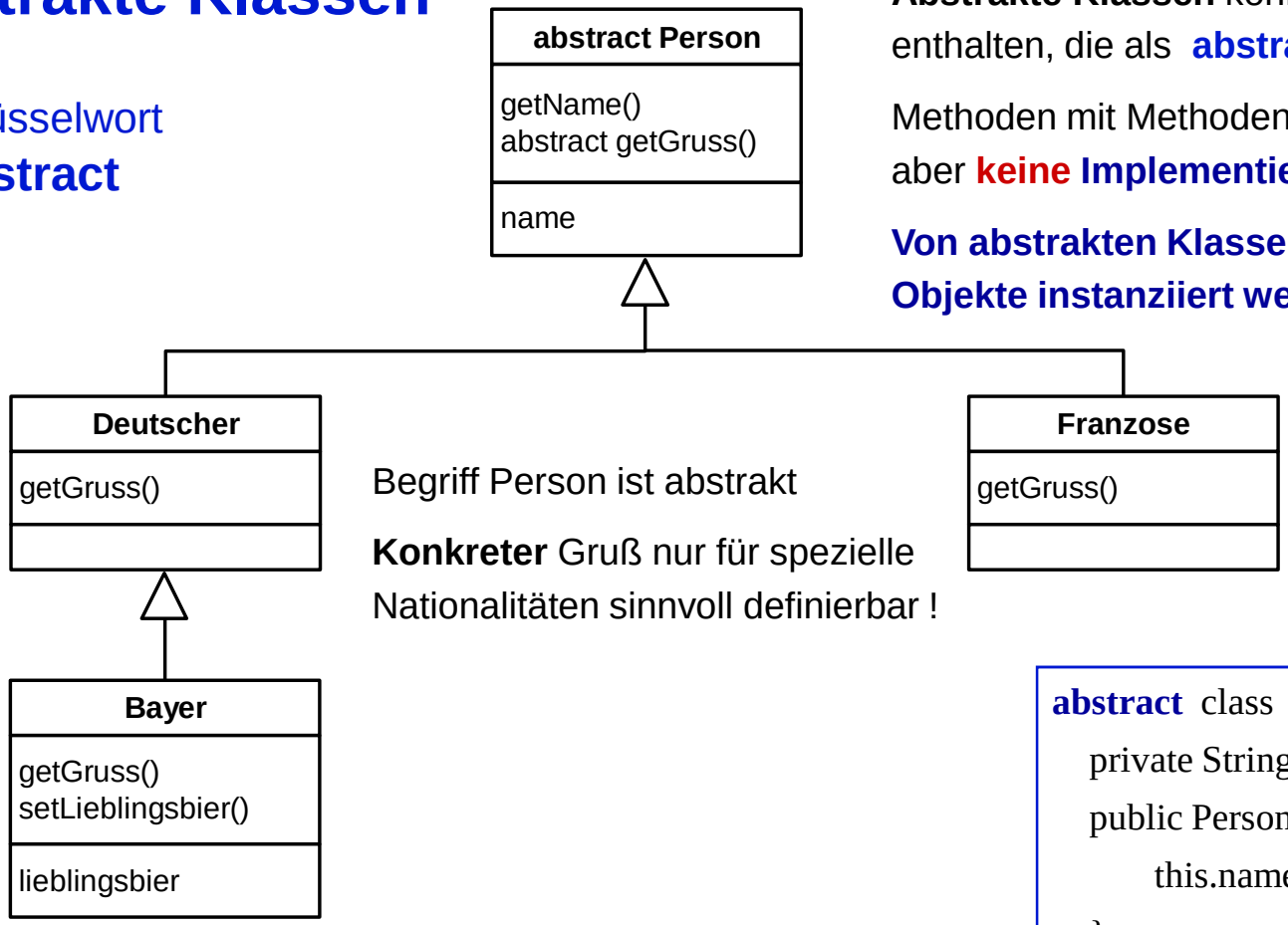
Nicht bei primitiven Datentypen …..

Vertiefung Objektorientierung
Ergänzung zur Implementierungsvererbung

- Abstrakte Klassen + Methoden
- Finale Klassen + Methoden
- Ausnutzen von Typkompatibilität
- Klasse Object als total-generischer Typ
- Anwendungsbeispiele / Template-Pattern
- Vererbung versus Assoziation

Abstrakte Klassen

Schlüsselwort
abstract



Abstrakte Klassen können Methoden enthalten, die als **abstract deklariert** sind

Methoden mit Methodenkopf **deklariert** - aber **keine Implementierung**

Von abstrakten Klassen können keine Objekte instanziiert werden !

Begriff Person ist abstrakt

Konkreter Gruß nur für spezielle Nationalitäten sinnvoll definierbar !

Nicht bei Konstruktoren

Abstrakte Klasse stellt Methoden **zum Überschreiben** in Unterklassen zur Verfügung

Deren konkrete Ausformulierung macht **erst dort Sinn,** weil erst für spezialisierte Unterklassen **semantisch** klar ist, wie die Wirkung einer solchen Methode aussieht !

Implementierung quasi "aufgeschoben"

```
abstract class Person {
    private String name ;
    public Person( String name) {
        this.name = name ;
    }
    public String getName( ) {
        return name ;
    }

    abstract public String getGruss( ) ;
}
```

Abstrakte Klassen

Von abstrakten Klassen können **keine Objekte instanziiert** werden - da nicht klar ist, wie sich Objekte bei Aufruf der abstrakten Methode verhalten sollten

Können jedoch :

- **implementierte** Methoden enthalten
- **statische implementierte Methoden** enthalten – auf abstrakter Klasse aufrufbar !

Bsp: **IO** könnte abstract sein.

Abstracte Klassen könnten nicht-abstrakte statische Factory-Methoden enthalten

Unterklassen *müssen* alle **geerbten abstrakten Methoden** überschreiben **+** implementieren ...

... sonst sind **sie *auch* abstrakt** und müssen als **abstract** gekennzeichnet werden !

Polymorphie funktioniert auch mit **abstrakten** Oberklassen **!**

Abstrakte Methoden können nicht **static** oder **private** sein !

- **Unterklassen** können **konkrete** Methoden **abstrakt überschreiben**
- In **Unterklassen** können *zusätzliche* abstrakte Methoden auftreten ⇒
- Es darf weitere **abstrakte** Unterklassen in einer Klassenhierarchie geben
- Abstrakte Methoden der Oberklasse können **nicht** mit **super.** aufgerufen werden

Abstrakte Klassen und Methoden

Unterklassen können auch weniger Varianten als Oberklasse haben.

Dafür hier **zusätzliche Bedingung** beim Quadrat : **breite == laenge** !

```
abstract class Figur {
   abstract public double flaeche( ) ;
}
```

```
class Kreis extends Figur {
  private double radius ;
  public Kreis( double r ) { radius = r ; }
  public double flaeche( ) {
    return Math.PI * radius * radius ;
  }
}
```

```
class Rechteck extends Figur {
  private double laenge ;
  private double breite ;
  public Rechteck( double l, double b ) {
    laenge = l ;   breite = b ;
  }
  public double flaeche( ) {
    return laenge * breite ;
  }
}

class Quadrat extends Rechteck {
  public Quadrat( double kante ) {
    super( kante, kante ) ;
  }
  // Flächenberechnung funktioniert mit
  // geerbter Methode !
}
```

Beispiel für **abstrakte Oberklasse** als **gemeinsamer Typ** :

Kreise und Rechtecke haben sonst nichts Gemeinsames

Vorsicht :

Wenn man abstrakter Oberklasse weitere abstrakte Methode hinzufügt **invalidiert** man alle bisherigen Verwender

Hinzufügen konkreter Methode mit Default-Implementierung jedoch meist problemlos

Ist der Fall wirklich so eindeutig? Ließe sich Rechteck auch so definieren, dass Quadrat keine sinnvolle Unterklasse wäre?

Die Verhaltensanforderungen des Clients sind entscheidend ….

Abstrakte Klassen und Methoden – Design-Möglichkeiten

```
abstract  class Figur  {
   abstract public double flaeche(  ) ;
}
```

```
abstract  class FCreator  {
   public static Figur createKreis( double r ) {
        return  new Kreis( r ) ;
   }
   public static Figur createRechteck( double l, double b ) {
        return  new Rechteck( l, b ) ;
   }
   public static Figur createQuadrat( double k ) {
        return  new Quadrat( k ) ;
   }
}
```

```
//  im Verwender-Coding :

Figur f   =  FCreator.createKreis( 5.0 ) ;

double flaeche = f.flaeche( ) ;
```

Vorteil :

Konkrete Unterklassen bleiben dem Verwender **verborgen** – ihr Name kann sich ändern.

Verwender arbeitet nur mit **Oberklassen-Typ** – bezieht sich nur auf ihren **Typ**.

Finale Klassen und Methoden

Nicht bei Konstruktoren

Finale Methoden - sind "endgültig" :

Dürfen in Unterklassen **nicht überschrieben** werden !

Dürfen **nicht abstrakt** sein - sonst blieben sie ohne Inhalt !

Finale Klassen - sind "endgültig" :

Ende der Vererbungshierarchie

Von ihnen **können keine Unterklassen abgeleitet werden**

Dürfen **keine abstrakten Methoden** enthalten !

```
final class Muenchner extends Bayer {

   public Muenchner( String name ) {
         super( name ) ;
   }

   final public String getGruss( ) {
         return "Servus !!" ;
   }
}
```

Finale Methoden :

Dürfen in Unterklassen **nicht** überschrieben werden !

↑↓

Abstrakte Methoden :

Sollen in Unterklassen überschrieben werden !

Vorteile :

Geschwindigkeit - Compiler kann **kompakteren** Bytecode generieren, wenn es **keine** weiteren polymorphen Unterklassen gibt

Sicherheit - Unterlaufen von **Regeln** durch Überschreiben von Methoden in Unterklassen wird verhindert.

Vererbung verhindern - bei dafür **ungeeigneten** Klassen !

Wenn finale Methoden intern nicht-finale, nicht-private Methoden verwenden, so kann sich ihr Verhalten doch indirekt in überschreibenden Unterklassen ändern

→ **Template-Pattern**

Anwendung : **Template - Pattern**


```
abstract class KoffeinhaltigesGetraenk  {
  public final void zubereitungsRezept( ) {
    kocheWasser( ) ;
    aufschütten( ) ;
    inTasseSchütten( ) ;
    if ( mitZutaten( )  ) {
      zutatenHinzu( ) ;
    }
  }

  public abstract void aufschütten( ) ;
  public abstract void zutatenHinzu( ) ;

  public void kocheWasser( ) {
    IO.writeln( "Koche Wasser" ) ;
  }
  public void inTasseSchütten( ) {
    IO.writeln( "Schütte in Tasse" ) ;
  }

  public boolean mitZutaten( ) { return true ; }
}
```

Zubereitung ist **final** ⇒

Am Rezept (**Algorithmus** / Reihenfolge der Schritte) können Unterklassen nichts ändern

Der Algorithmus enthält abstrakte Methoden ⇒

Durch **Überschreiben** können Unterklassen doch ***indirekt*** eine **Anpassung** vornehmen

Diese Methode ist ein " **Hook** " :=

Eine Methode mit Default-Implementierung ⇒

Die Unterklassen können sich durch Überschreiben der Methode hier "einhaken" – müssen es aber nicht ...

Anwendung : Template - Pattern


```
class Kaffee extends KoffeinhaltigesGetraenk  {
  public void aufschütten( ) {   IO.writeln( "Kaffee durch Filter"  ) ;  }
  public void zutatenHinzufügen( ) {   IO.writeln( "Zucker + Milch"  ) ;  }
  public boolean mitZutaten( )  {
    char z = IO.promptAndReadChar( "Zutaten?" ) ;
    if ( z == 'j' ) {  return true ;  }
    else {  return false ;  }
  }
}
```

```
class Tee extends KoffeinhaltigesGetraenk  {
  public void aufschütten( ) {   IO.writeln( "Tee in Kanne" ) ;  }
  public void zutatenHinzufügen () {   IO.writeln( "Zitrone" ) ;  }
  public boolean mitZutaten( )  {
    char z = IO.promptAndReadChar( "Zutaten?" ) ;
    if ( z == 'j' ) {  return true ;  }
    else {  return false ;  }
  }
}
```

Die Unterklassen **passen Algorithmus** (das ***Template***) durch Überschreiben an ihre Bedürfnisse **an** !

Viele weitere Getränke können implementiert werden – ohne an abstrakter Basisklasse und am Algorithmus selbst Änderungen vornehmen zu müssen

⇓

Gemeinsame + unveränderliche Anteile in Basisklasse erfasst

Template für Unterklassen

Design-Patterns !

Joshua Bloch, ***Effective Java*** S.234

Don't sacrifice sound architectural principles for performance.

Strive to ***write good programs rather than fast ones****.*

If a good program is not fast enough, its architecture will allow it to be optimized.

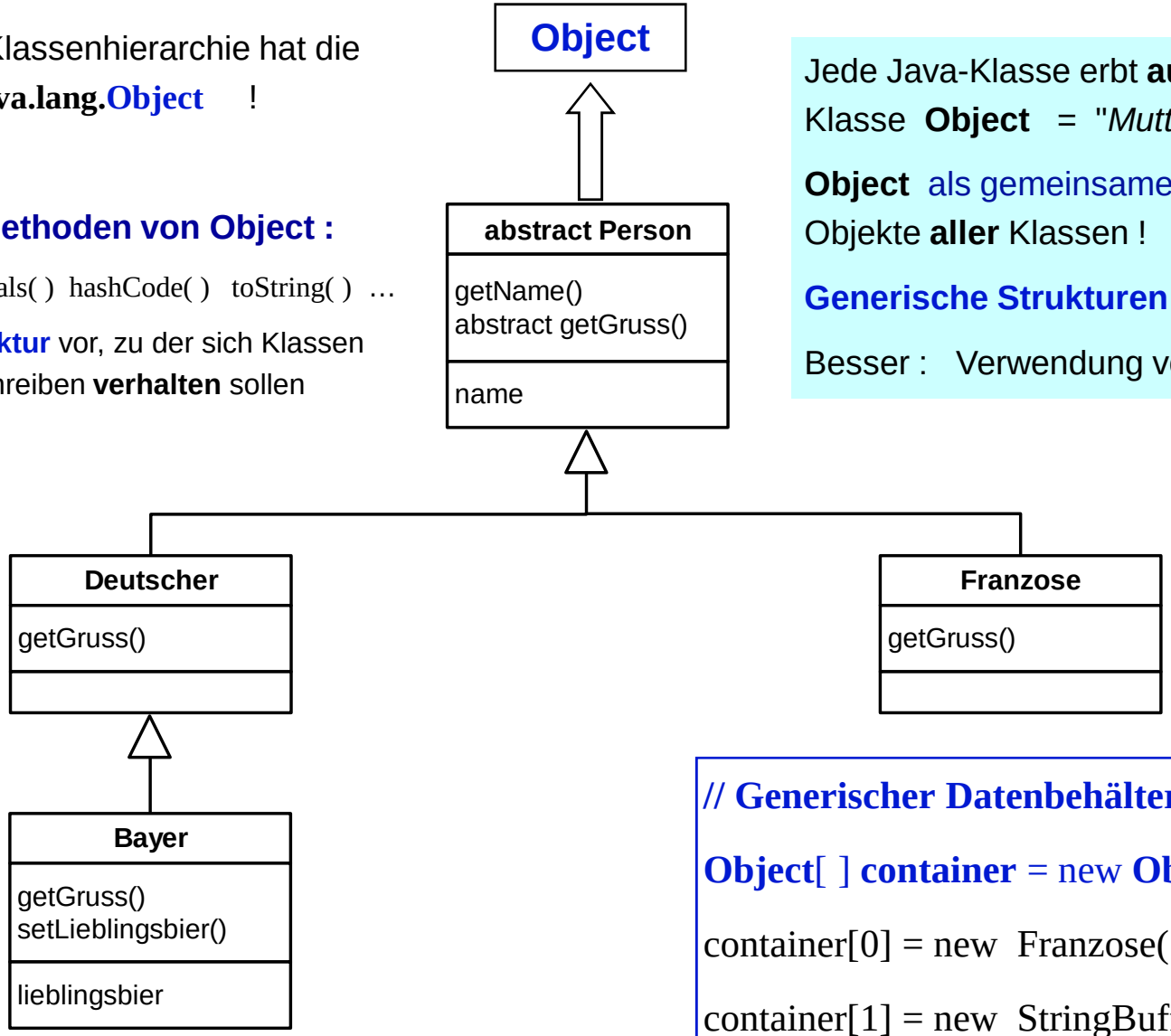
Klasse Object - Wurzel Java-Klassenhierarchie

Jede Java-Klassenhierarchie hat die **Wurzel** **java.lang.Object** !

Konkrete Methoden von Object :

clone( ) equals( ) hashCode( ) toString( ) …

Gibt **Infrastruktur** vor, zu der sich Klassen durch Überschreiben **verhalten** sollen



Jede Java-Klasse erbt **automatisch** von Klasse **Object** = "*Mutter aller Klassen*"

Object als gemeinsamer **supertype** der Objekte **aller** Klassen !

Generische Strukturen mittels Upcast

Besser : Verwendung von Generics

```
// Generischer Datenbehälter - gut !??
Object[ ] container = new Object[3] ;
container[0] = new  Franzose( "Jean" ) ;
container[1] = new  StringBuffer( "Wort" ) ;
container[2] = new  Integer( 156 ) ;
```

Design : Beziehungen zwischen Objekten

Zwei grundsätzlich verschiedene
***semantische* Beziehungen** prägen die **OO**

1. **Ist_Ein – Beziehung :**

Ein Franzose **ist eine** Person ⇒

Darstellung durch **Vererbung** - als Ausdruck von **Spezialisierung, Erweiterung :**

Alles, was sich wesentlich in der Oberklasse findet, ist **auch** in der Unterklasse vorhanden.

Vererbung stellt diese Verhältnisse **semantisch korrekt** dar!

Unterklassen-Objekt kann Oberklassen-Objekt voll und ganz **vertreten** !

2. **Hat_Ein – Beziehung ...**

```
class Person  {
   private String name ;
   public Person( String nn ) {
         name = nn ;
   }
   public String getName( ) {
          return name ;
   }
   private String name ;
}

class Franzose extends Person  {
   public Franzose( String name) {
         super( name ) ;
   }
   public String getName( ) {
        String n = super.getName( ) ;
        n = "Name ist: " + n ;
        return n ;
   }
}
```

Beziehungen zwischen Objekten

2. Hat_Ein-Beziehung :

Eine Linie **hat** Anfangs- und Endpunkt ⇒

Darstellung als **Assoziation** - **nicht** durch Vererbung !

Zusammenwirken gleichberechtigter Objekte

Häufig auch : **Ganzes-Teil-Beziehung**

Darstellung durch Vererbung wäre **semantisch falsch**

Eine Linie **ist keine** spezielle Punkt-Version !

Keine Spezialisierung - sondern **Verwendung einer Objektreferenz als Attribut in einer anderen Klasse**

```
class Point  {
   public Point( double xk, double yk )  {
          x = xk ;
          y = yk ;
    }

    private double x ;
    private double y ;
}


class Linie  {
   public Linie( Point  pA,  Point pB )  {
       pAnfang = pA ;
       pEnde = pB ;
    }

    private Point pAnfang ;
    private Point pEnde ;
    // …
}
```

Vertiefung Objektorientierung
Spezielle Methoden - Klasseninfrastruktur

- equals( )
- hashCode( )
- equals-hashCode-Kontrakt

… wird durch weitere Methoden und Infrastruktur-Interfaces noch vertieft …

Spezielle Methoden : equals( ) + hashCode( )

public boolean equals(Object o) { /* … */ } // **von Object geerbt – dort Identitäts-Check**

Drückt aus, ob **Instanzen äquivalente Entität** repräsentieren

Überschreiben, wenn **verschiedene Instanzen äquivalente Entity repräsentieren** können

Voraussetzung :

Klärung der **semantische Bedingungen** - was sind die **relevanten (unveränderlichen!) Attribute** ?

Asymmetrische Aufrufsyntax : **boolean b = m1.equals(m2) ;**

Reflexivität	a.equals(a) == true
Symmetrie	a.equals(b) == b.equals(a)
Transitivität	a.equals(b), b.equals(c) ⇒ a.equals(c)
Konsistenz	a.equals(b) liefert **immer** selben Wert

Nur nicht-veränderliche Attribute als Kriterium heranziehen !

Implementierungsregeln :

- Instanzen unterschiedlicher Klassen sind **stets** verschieden !
- Bivalenz : Rückgabe stets true oder false - **keine** Exceptions !
- Signatur einhalten – **nicht** überladen statt überschreiben !

```
class Mitarbeiter {
  // ...
  public boolean equals( Object o ) {  }
}
```

```
class Mitarbeiter {
  // ...
  public boolean equals( Mitarbeiter m ) {  }
}
```

Verletzt Transitivität !

Spezielle Methoden : **equals( ) - Implementierung**

```
class Mitarbeiter {
  private int persNr ;
  private String name ;
  Mitarbeiter( int nr, String nn ) {  /* ... */  }

  public boolean equals( Object o ) {    // 1. korrekte Signatur wählen
    // 2. Identitätstest – leichter Performanzgewinn bei Identität - nicht unbedingt nötig :
    if( this== o )  return true ;

    // 3. NullpointerException / unterschiedliche Klassentypen / ClassCastException ausschließen :
    if(  o==null  ||  this.getClass( )  !=  o.getClass( )  )  return false ;

    // 4. Cast auf Typ der relevanten Klasse :
    Mitarbeiter m = (Mitarbeiter) o ;

    // 5. relevanter Attributvergleich - liefert true oder false :
    return  this.persNr == m.persNr ;
  }
}
```

Relevantes Attribut sei eindeutige + unveränderliche Personalnummer

getClass( ) wird von **Object** vererbt. Liefert identisches Class-Objekt bei identischem Klassentyp.

Korrekte Signatur !

Keine Exceptions erzeugen !

Bivalenz !

Spezielle Methoden : **equals( ) -** Mehrere Attribute

```
import java.util.Date ;
class Person {   // relevant seien:   name, gebOrt, gebDatum
   private String name ;      private String vorname ;
   private String gebOrt ;
   private Date gebDatum ;
   public Person( … ) {   /* ... */   }

   public boolean equals( Object o ) {
      if( this == o )  return true ;
      if( o == null   ||   this.getClass( ) != o.getClass( )  )  return false ;
      Person p = (Person) o ;
      if(  ! this.gebName.equals( p.gebName )  ) return false ;
      if(  ! this.gebOrt.equals( p.gebOrt )  )  return false ;
      if(  ! this.gebDatum.equals(  p.gebDatum )  )   return false ;
      return true ;
   }
}
```

Schema :

Teste jedes relevante Attribut

Stimmt nicht überein ⇒ **false**

Am Ende ⇒ **true**

Für jede Klasse
identischer Code-Teil

! equals( ) bei Objekttypen
!= bei primitiven Typen

Spezielle Methoden : **equals( ) -** in Vererbungshierarchie

```
class FarbPunkt extends Punkt {    //  x,y sind relevant
  private String farbe ;  // zusätzliches relevantes Attribut
  public FarbPunkt( int x, int y, String farbe ) {
      super( x, y ) ;       this.farbe = farbe ;
  }

  public boolean equals( Object o ) {
    if( this == o ) return true ;
    if(  o == null   ||   o.getClass( ) != getClass( )  )  return false ;
    if( ! super.equals( o )  )  return false ;    // als Punkt gleich ?
    FarbPunkt that = (FarbPunkt) o ;
    if( ! farbe.equals( that.farbe )  )   return false ;
    return true ;
  }
}
```

Schema :

Delegation der Prüfung von **Superklasse-Attributen** an die Superklasse

Voraussetzung :

Auch diese hat **equals**() korrekt implementiert

Delegation an Superklasse

Diese prüft auf Gleichheit der Koordinaten **(x,y)**

Voraussetzung für jede **weitere** sinnvolle Prüfung

Kontrakt :

*Wer **equals( )** überscheibt, der muss auch **hashcode( )** überschreiben …*

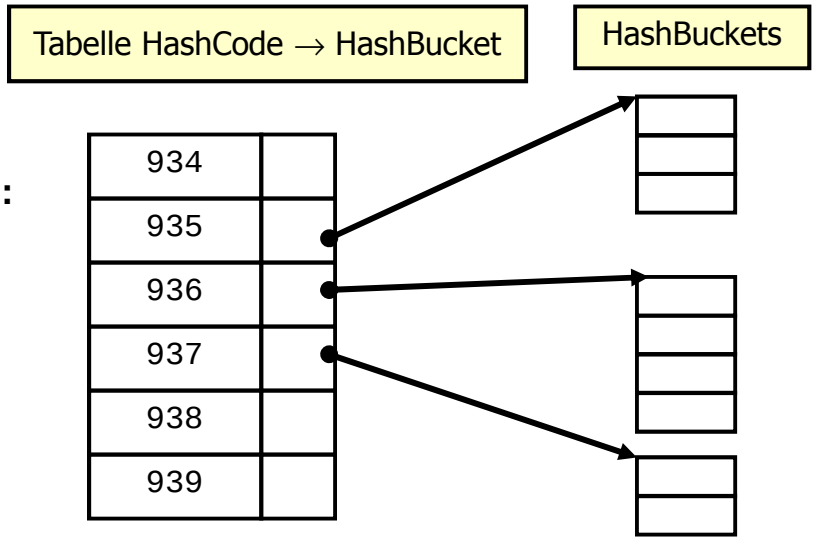
Spezielle Methoden : **equals( )-hashCode( ) - Kontrakt**

```
public int hashCode( ) {   /* … */   }        // von Object geerbt  -  liefert  identityHashcode
```

Einhaltung **Kontrakt** :

Wer equals( ) überschreibt, muss auch hashCode( ) überschreiben !

Sonst funktionieren **Hash-basierte Container nicht** !



Hashcode-basierte Schlüsselindizierte Objekt-Verwaltung :

- **Hashfunktion** liefert für jedes Objekt int-Wert = **Hashcode**
- Objektreferenzen in HashBuckets abgelegt
- Objekte mit gleichem HashCode im gleichen HashBucket
- HashBuckets über Tabelle verwaltet
- Zugriff auf HashBucket mit HashCode
- Größe der HashBuckets variiert dynamisch

equal objects must have **equal hash codes** !

API-Dokumentation zu ***Object.equals( )***

Spezielle Methoden : **hashCode( ) - Implementierung**

```
class Mitarbeiter {
  int persNr ;   // relevantes eindeutiges Attribut
  String name ;
  Mitarbeiter( int nr, String nn ) { /* ... */ }

  public boolean equals( Object o )  {
    if( this== o )  return true ;
    if(  o==null  ||  this.getClass( )  !=  o.getClass()  )
          return false ;
    Mitarbeiter m = (Mitarbeiter) o ;
    return  this.persNr == m.persNr ;
  }

  public int hashCode( ) {
    return persNr ;   // oder Funktion davon ...
  }
}
```

```
import java.util.HashSet ;
class Container {
 public static void main( String[ ] args ) {
  Mitarbeiter m1 = new Mitarbeiter( 2, "Hans" ) ;
  Mitarbeiter m2 = new Mitarbeiter( 2, "Hans" ) ;
  HashSet h = new HashSet( ) ;
  h.add( m2 ) ;
  boolean test = h.contains( m1 ) ;
  // liefert nur bei Überschreiben
  // von hashCode( )  true !
  // ansonsten nicht im Container gefunden !
 }
}
```

Methoden **equals( )** und **hashCode( )** müssen **aneinander angepasst** sein !

Idealfall :

Nur auf nicht-veränderliche Attribute beziehen !

Spezielle Methoden : **hashCode( ) - bei Vererbung**

```
class Punkt {
  private int x ;   private int y ;
  public Punkt( int x, int y ) {
    this.x = x ;  this.y = y ;
  }
  public boolean equals( Object o ){  /* ... */  }
  public int hashCode( ) {
    return 11 * (3 + x) + y ;
    // alternativ :  return x + y ;
  }
}
```

```
class FarbPunkt extends Punkt {
  private String farbe ;
  public FarbPunkt( int x, int y, String farbe ) {
      super( x, y ) ;       this.farbe = farbe ;
  }
  public boolean equals( Object o ) {  /* ... */  }
  public int hashCode( ) {
      return  super.hashCode( ) + farbe.hashCode( ) ;
  }
}
```

Delegation des entsprechenden Teils der **hashCode-Berechnung** an die **Oberklasse**

Anmerkung : **Eclipse** generiert **equals( )** und **hashCode( )** - Methoden

Source → Generate hashCode() and equals() …