gds2 Gottlieb-Daimler-Schule 2
Technisches Schulzentrum Sindelfingen
mit Abteilung Akademie für Datenverarbeitung

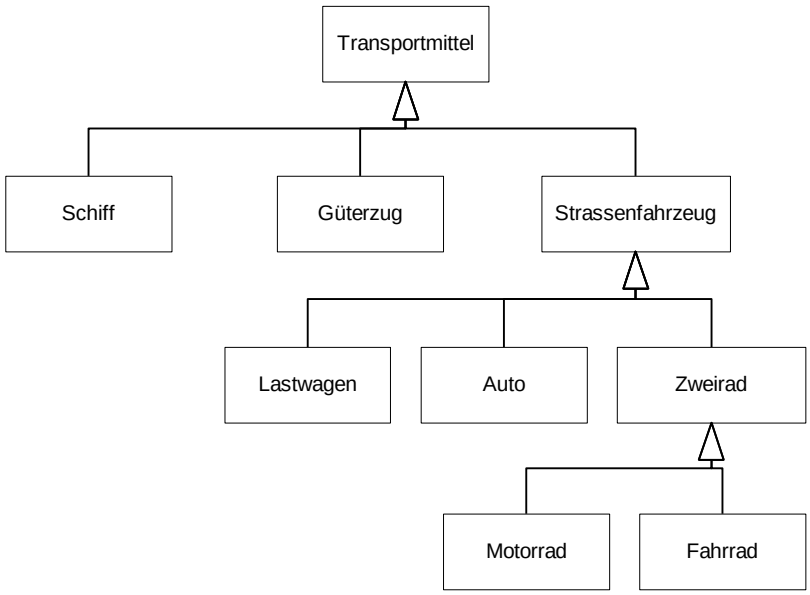
Inhaltsverzeichnis

1 Begriffsbestimmung 2
2 Umsetzung in Java 3
3 Überlagern (überschreiben) von geerbten Methoden 5
3.1 Allgemeines 5
3.2 Die Referenzvariable `super` 5
3.3 Late binding - Polymorphismus 5
3.4 Das Attribut "final" 7
3.5 Abstrakte Klassen 7
3.6 Abstrakte Methoden 8
3.7 Vererbung von Konstruktoren 8

1 Begriffsbestimmung

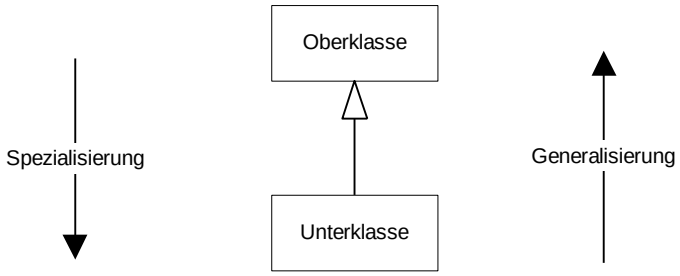
Unter Vererbung versteht man in der objektorientierten Programmierung die Möglichkeit, auf der Grundlage einer bereits vorhandenen Klasse (= Oberklasse/Basisklasse) eine andere Klasse abzuleiten (= Unterklasse, abgeleitete Klasse).
Die Unterklasse *erbt* dabei alle Attribute und Methoden der Oberklasse. Sie sind damit dort automatisch bekannt und brauchen nicht erneut programmiert zu werden.
Ausnahmen: Methoden, die als private deklariert wurden, werden nicht vererbt.
Die Unterklasse kann noch um zusätzliche Attribute und Methoden erweitert werden. Sie ist somit spezialisierter als die Oberklasse.
Die Attribute und Operationen der Unterklasse sind dagegen in der Oberklasse unsichtbar.

Beispiel:



Die Vererbung kann also auch mehrstufig sein, es entsteht eine Vererbungshierarchie.

Allgemeine Darstellung in UML :



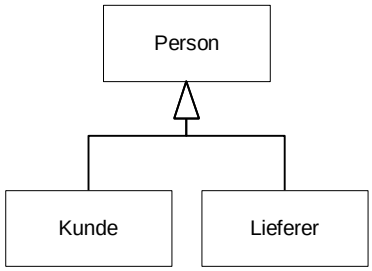
Durch Vererbung ergibt sich eine "ist-ein-Beziehung" ("is-a Relationship"), im Unterschied zur "hat-ein-Beziehung" bei Aggregationen.

Ein Güterzug "ist ein" Transportmittel.
Ein Fahrrad "ist ein" Zweirad, ein Zweirad "ist ein" Straßenfahrzeug, ein Straßenfahrzeug "ist ein" Transportmittel.

Ein Güterzug besitzt alle Attribute und Methoden eines Transportmittels.
Ein Fahrrad besitzt alle Attribute und Methoden eines Zweirads, eines Straßenfahrzeugs und eines Transportmittels.

2 Umsetzung in Java

Die Tatsache, dass eine Klasse von einer Oberklasse abgeleitet ist, wird mit dem Schlüsselwort **extends** angegeben.




```
public class Person     // Datei Person.java
{
  private String szName;
  private String szVorName;

  public void setName(String neuerName)
  {...
  }
  public void setVorName(String neuerVorName)
  {...
  }
  public String getName()
  {...
  }
  public String getVorName()
  {...
  }
}

public class Kunde extends Person
{                            //Datei Kunde.java
  private double dRabatt;

  public void setRabatt (double dRabatt)
  {...
  }
  public double  getRabatt ()
  {...
  }
}

public class Lieferer extends Person
{                          //Datei Lieferer.java
  private int iBestellsumme;

  public void setBestellsumme (int iAnzahl)
  {...
  }
  public int  getBestellsumme ()
  {...
  }
}
```

Ein Kunde "ist eine" Person, erbt damit auch alle Attribute und Methoden der Klasse Person. Somit ist beispielsweise folgendes möglich:

```
public class KundenverwaltungGUI
{
  public static void main(String [] arg)
  {
    Kunde einKunde = new Kunde();
    einKunde.setRabatt (1.5);

    /*
    * Obwohl setName() nicht in der Klasse Kunde programmiert ist:
    */
    einKunde.setName("Maier");
  }
}
```

Einige Anmerkungen

- die Vererbungshierarchie kann ***beliebig lang*** sein
- in Java hat jede Klasse genau eine direkte Oberklasse (=***Einfachvererbung***). Mehrfachvererbung wird in Java nicht unterstützt (im Gegensatz zu C++)
- hat eine Klasse keine direkte Oberklasse, so erbt sie automatisch von der Klasse Object. Object ist also die "Mutter aller Klassen".
- Von einer Klasse mit dem Attribut ***final*** können ***keine Unterklassen gebildet werden***
- Objekte einer Unterklassen sind immer ***zuweisungskompatibel*** zu Verweisvariablen einer Oberklassen, da sie alle Eigenschaften der Oberklasse enthalten:

  ```
  Person einePerson = new Kunde(); oder
  Person[] personen =  new Person[100];
  personen[0] = new Kunde();
  ```

  ist zulässig.

3 Überlagern (überschreiben) von geerbten Methoden

3.1 Allgemeines

Methoden, die von der Oberklasse geerbt werden, können in der erbenden Klasse neu programmiert werden. Dazu muss die Signatur genau übereinstimmen, also Name, und Parameter. Es wird dann immer die überlagernde Methode verwendet.

Die überlagerte Methode ist jedoch nur verdeckt. Sie kann nach wie vor aufgerufen werden, indem dem Methodennamen `super.` vorangestellt wird: `super.ueberlagerteMethode().`

3.2 Die Referenzvariable `super`

Analog zur Referenzvariable `this` ist auch die Referenzvariable `super` eine automatisch erzeugte Referenzvariable, die jedoch nicht auf das eigene, sondern auf das Elternobjekt verweist.

3.3 Late binding - Polymorphismus

Java unterstützt das Prinzip des "Späten Bindens" ("late binding"). Das bedeutet, dass erst zur Laufzeit des Programms entschieden wird, welche Methode aufgerufen wird. Da durch die Vererbung Objekte einer abgeleiteten Klasse immer auch Objekte der Oberklassen sind, können geerbte Methoden überschrieben werden und somit ist erst zur Laufzeit klar, welche Methode aufzurufen ist.

Erklärung zum Bsp. auf der nächsten Seite:

3/7	Die Methode `zeigeTyp()` wird in der abgeleiteten Klasse überlagert
8	Die Referenzvariable `einePerson` kann sowohl auf Objekte der Klasse `Person` als auch `Kunde` verweisen. Diese Tatsache zusammen mit dem „späten Binden“ macht den **Polymorphismus** (Vielgestaltigkeit) der OOP aus.
9	Abhängig von der Benutzereingabe wird ein Objekt der Klasse `Person` oder `Kunde` erzeugt.
10	Erst zur Laufzeit des Programms („at-run-time“) kann entschieden werden, auf welchen Objekttyp `einePerson` verweist und somit welche Methode `zeigeTyp()` aufgerufen werden soll
1/2	Die Klasse `Person` besitzt zwei überladene Konstruktoren, die durch den Aufruf von `this()` miteinander verkettet sind.
4/5	Die abgeleitete Klasse `Kunde` besitzt ebenfalls zwei überladene Konstruktoren, die durch den Aufruf von `this()` miteinander verkettet sind und außerdem durch den Aufruf von `super()` mit den Konstruktoren der Basisklasse verkettet sind. Die Verkettung von Konstruktoren der Unterklasse mit der Basisklasse ermöglicht die Initialisierung der Attribute der Basisklasse, ohne dass die Unterklasse Zugriff auf die Attribute benötigt. **Hinweis**: Der Aufruf von `this()` oder `super()` in einem Konstruktor muss immer die erste Anweisung im Konstruktor sein. Es kann nur entweder `this()` oder `super()` aufgerufen werden, nicht beides. Es wird immer zuerst der Konstruktor der Basisklasse aufgerufen, entweder explizit oder implizit.

Beispiel:

```
        public class Person
        {
            private String sName;

1           public Person()
            {
               this("NoName");
            }

2           public Person(String sName)
            {
               this.sName = sName;
            }

3           public String zeigeTyp()
            {
              return  ("Person: " + this.sName);
            }
        }
        public class Kunde extends Person
        {
            private double dRabatt;

4           public Kunde()
            {
             this("NoBody", 1.5);
            }

5           public Kunde(String sName, double dRabatt)
            {
6            super(sName);
             this.dRabatt = dRabatt;
            }

7           public String zeigeTyp()  //überschreibt geerbte Methode
            {
              return  "Kunde: " + this.getName() + " Rabatt = "
                       + this.dRabatt;
            }
          }
        public class PersonStartUI
        {
          public static void main(String[] args)
          {
8           Person einePerson;
            int i = Eingabe.getInt("Bitte Zahl eingeben:");

9             einePerson = (i > 0) ? new Person() : new Kunde();

10          System.out.println(einePerson.zeigeTyp()); // late binding
          }
        }
```

3.4 Das Attribut "final"

Das Attribut final kann bei Variablen, Methoden und Klassen verwendet werden.

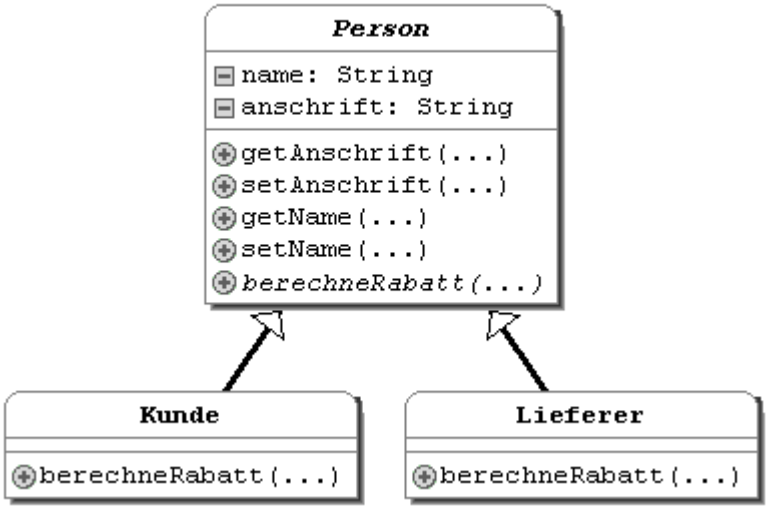
- ***Variablen*** mit dem Attribut final sind symbolische Konstanten. (Siehe Informationsblatt "Variablen ").
  final int MAX = 50;

- ***Methoden*** mit dem Attribut final dürfen nicht überschrieben werden.
  public final void setName(String neuerName) {...}

- Von ***Klassen*** mit dem Attribut final dürfen keine Unterklassen gebildet werden.
  public final class Kunde { ... }

3.5 Abstrakte Klassen

Eine abstrakte Klasse dient nur als Vorlage für davon abgeleitete Klassen. Von einer abstrakten Klasse muss eine Unterklasse gebildet werden. Von einer abstrakten Klasse kann kein Objekt erzeugt werden.
Dies ist dann sinnvoll, wenn es sich um eine allgemeine Oberklasse handelt, von der eine Instanziierung keinen Sinn macht.
Eine abstrakte Klasse wird mit dem Schlüsselwort abstract gekennzeichnet. In der UML wird ihr Name kursiv dargestellt.
So könnte im Beispiel von oben die Klasse Person abstrakt sein:




```
public abstract class Person
{ ...
  public abstract double berechneRabatt();  /* abstrakte Methode,
  * keine {} Klammern!
  */
}
```

3.6 Abstrakte Methoden

Eine abstrakte Methode enthält im Gegensatz zu einer konkreten Methode keine Implementierung, sondern nur die Deklaration. Wenn eine Klasse mindestens eine abstrakte Methode enthält, so muss die Klasse auch abstrakt sein.
Die konkrete Implementierung der Methode muss dann in einer abgeleiteten Klasse erfolgen. So ist gewährleistet, dass alle abgeleiteten Klassen eine bestimmte Methode besitzen.

Beispiel:

```
public abstract class Person
{ ...
  public abstract double berechneRabatt();  /* abstrakte Methode,
  * keine {} Klammern!
  */
}
public class Kunde extends Person
{ ...
  public double berechneRabatt()
  {
    // konkrete Implementierung
  }
}
```

3.7 Vererbung von Konstruktoren

Konstruktoren werden nicht vererbt. Jede Klasse muss ihre(n) eigenen Konstruktor bereitstellen. Allerdings garantiert der Compiler, dass ***immer als erste Anweisung*** in einem Konstruktor der ***Konstruktor der Oberklasse aufgerufen*** wird.

Dies kann explizit durch Aufruf der Methode **super()** geschehen. super() kann mit oder ohne Parametern aufgerufen werden. Es muss also in der Oberklasse ein passender Konstruktor vorhanden sein.
Wird super() nicht explizit aufgerufen, so generiert der Compiler automatisch als erste Anweisung einen Aufruf super().

Durch diese ***Verkettung*** der Konstruktoren wird immer als erstes der Konstruktor der Klasse Object aufgerufen und anschließend alle anderen in der Vererbungshierarchie abwärts. Als letztes also der Konstruktor der aktuellen Klasse.