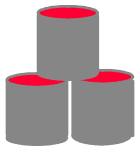
Grundlagen der Informatik
– Objektorientierte Software-Entwicklung –

Prof. Dr. Bernhard Schiefer

(basierend auf Unterlagen von Prof. Dr. Duque-Antón)

bernhard.schiefer@fh-kl.de
http://www.fh-kl.de/~schiefer



Inhalt

- Grundlagen
- Klassen und Objekte
- Methoden
- Konstruktoren
- Statische Attribute und Methoden
- Vererbung
- Abstrakte Klassen
- Modifizierer
- Interfaces

Grundlagen

- Entscheidend für den objektorientieren Ansatz ist nicht das objektorientierte Programmieren, sondern das *Denken in Objekten*
  - ⇨ Es wird dazu in Konzepten und Begriffen der realen Welt anstatt in rechnernahen Konstrukten wie Haupt- und Unterprogrammen gedacht.
  - ⇨ Vor der Programmierung wird analysiert, welche Objekte von Bedeutung sind, um die Aufgaben des Zielsystems zu erfüllen.
- Beispiel: Für die Entwicklung einer Bankanwendung müssen zunächst die relevanten Objekte gefunden werden:
  - ⇨ Konto
  - ⇨ Kunde
  - ⇨ …

Basismechanismen

- Die Grundlagen der objektorientierten Software-Entwicklung sind:
  - ⇨ Das Geheimnisprinzip (Information Hiding)
    - Kapselung
  - ⇨ Spezialisierung bzw. Vererbung

Objekte

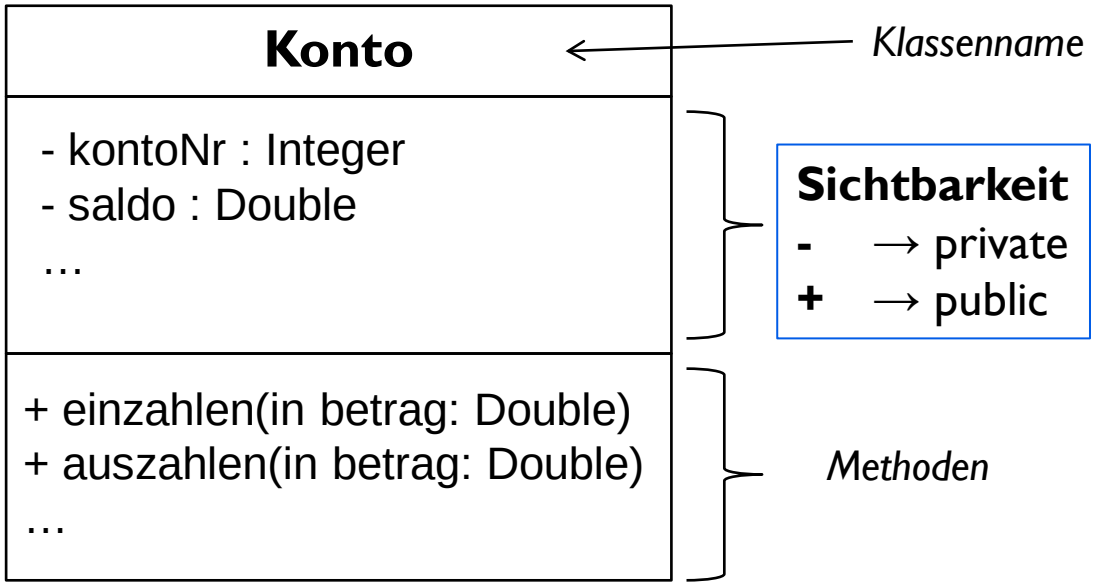
- Jedes Objekt hat gewisse *Attribute/Eigenschaften* und ein *Verhalten*
  - ⇨ Die Eigenschaften werden durch Daten beschrieben
  - ⇨ Das Verhalten durch Operationen, die auf einem Objekt ausgeführt werden können
- Eigenschaften von Objekten
  - ⇨ sind Datenfelder, die Werte von Attributen oder Statusinformationen enthalten.
  - ⇨ Beispiel Konto: KontoNr, Saldo, …
- Operationen auf Objekten werden als „Methoden“ des Objektes bezeichnet
  - ⇨ Methoden legen fest, was man mit dem Objekt alles tun kann.
  - ⇨ Beispiel Konto: abheben, einzahlen, …

Klassen

- Klassen stellen die Baupläne für Objekte dar
  - ⇨ Objekte werden immer gemäß dem in einer Klassen abgelegten Bauplan erzeugt (instanziiert).
  - ⇨ Klassen entsprechen damit Datentypen.
  - ⇨ Die Objekte stellen Variablen (Instanzen) dieser Datentypen dar.
- Aufbau von Klassen
  - ⇨ Jede Klasse hat einen Klassennamen
  - ⇨ Die Klasse legt die Datenfelder und die Methoden der zugehörigen Instanzen fest.
  - ⇨ Klassen stellen einen Namensraum bereit: Die gleichen Datenfelder und Methodensignaturen können in verschiedenen Klassen existieren.
  - ⇨ Eine grafische Darstellung ist unter Nutzung der Konzepte der UML-Klassendiagramme möglich

Erstes Beispiel: Konto

- Modellierung einer Klasse Konto
  - ⇨ Benötigte Attribute: Kontonummer, Saldo, …
  - ⇨ Welche Interaktionen mit Konto sind nötig? (Methoden)
- Darstellung als UML-Klassendiagramm



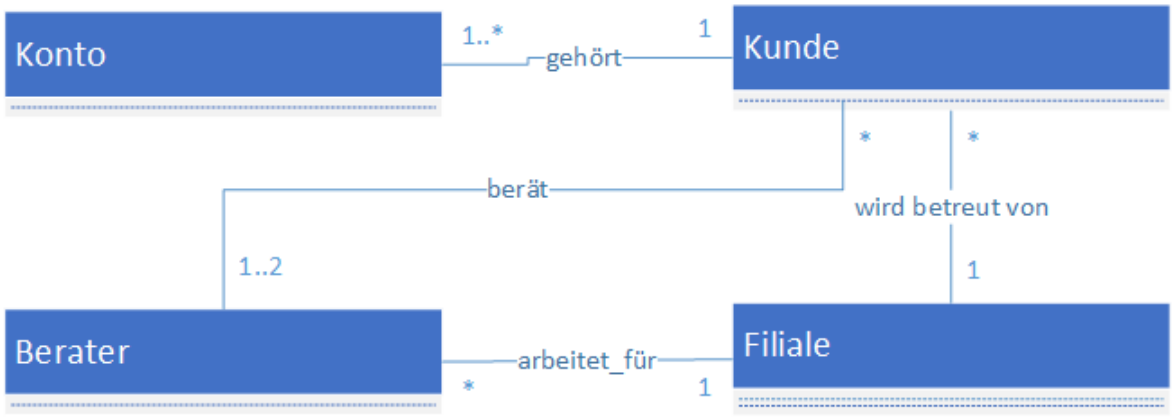
UML (Unified Modeling Language)

- eine grafische Modellierungssprache zur Spezifikation, Konstruktion und Dokumentation von (Software-)Systemen
  - ⇨ Visualisieren, Spezifizieren, Konstruieren, Dokumentieren
- aktuell die dominierende Sprache für die Softwaresystem-Modellierung
- UML umfasst zahlreiche unterschiedliche Diagrammarten
  - ⇨ Verhaltensdiagramme
    - Aktivitätsdiagramm, Use-Case-Diagramm, Sequenzdiagramm, …
  - ⇨ Strukturdiagramme
    - Verteilungsdiagramm, Komponentendiagramm, **Klassendiagramm**, …

UML - Assoziationen

- **Assoziationen** repräsentieren Beziehungen zwischen Instanzen von Klassen.
  - ⇨ Jede Assoziation kann einen Namen besitzen
  - ⇨ Die Multiplizität kann angegeben werden

- Beispiel:

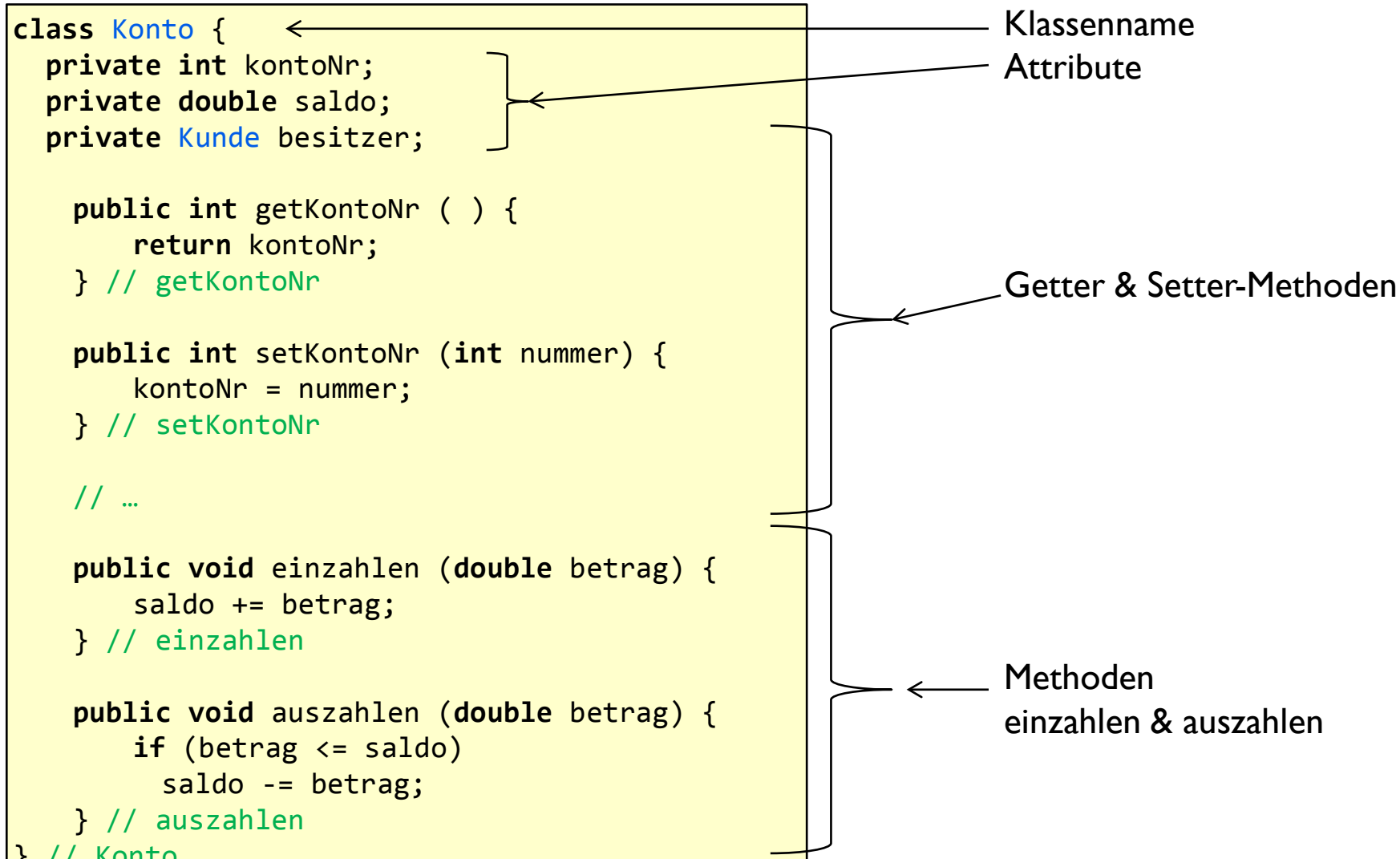


Angaben zu Multiplizität

1	→ genau 1
*****	→ 0 oder mehr
2..4	→ zwischen 2 und 4

Erstes Beispiel: Konto in Java

```
class Konto {
  private int kontoNr;
  private double saldo;
  private Kunde besitzer;

    public int getKontoNr ( ) {
        return kontoNr;
    } // getKontoNr

    public int setKontoNr (int nummer) {
        kontoNr = nummer;
    } // setKontoNr

    // …

    public void einzahlen (double betrag) {
        saldo += betrag;
    } // einzahlen

    public void auszahlen (double betrag) {
        if (betrag <= saldo)
          saldo -= betrag;
    } // auszahlen
} // Konto
```



Erzeugen von Instanzen in Java

- Um von einer Klasse eine Instanz zu erzeugen:
  1. eine Variable vom Typ der Klasse deklarieren
  2. mit Hilfe des **new**-Operators ein neues Objekt erzeugen und der Variablen zuweisen
- Beispiel:
  - ⇨ Konto meinKonto = **new** Konto ();
- Erinnerung:
  - ⇨ Der new-Operator liefert die Adresse eines Speicherbereiches auf dem Heap zurück!
- Die Variable meinKonto enthält also nur eine Referenz auf den entsprechenden Speicherplatz.

Beispiel: Referenz-Variablen

```
Konto meinKonto = new Konto();

Konto deinKonto;

deinKonto  =  meinKonto;
```

- Die Variablen `meinKonto` und `deinKonto` verweisen beide auf das selbe Objekt.
- Für zwei Referenzvariablen x und y hat der Vergleich (x == y) den Wert true, wenn beide auf dasselbe Objekt verweisen

Auswirkung auf Parameter

- Erinnerung: Die Parameterübergabe erfolgt bei Java nach dem Prinzip call-by-value:
  - ⇨ Für jeden Parameter wird innerhalb der Methode ein eigener Speicherplatz erzeugt, in den der Parameterwert kopiert wird.
  - ⇨ Dadurch bleiben die Variablen, die beim Aufruf übergeben werden, unverändert.
- Was geschieht, wenn ein Argument eine Klasse als Typ hat?
  - ⇨ Ist das Argument eine Referenzvariable, so wird die Referenz kopiert !
  - ⇨ Die Methode kann daher die Eigenschaften dieses Objektes verändern.

Beispiel: Parameterübergabe

```
// Datei: ParamTest.java
public class ParamTest {

   static void setVar (int n) {
      n = 4711;
   } // setVar

   static void setKonto (Konto k) {
      k.setKontoNr (4711);
   } // setKonto

   public static void main (String [ ] args) {
      Konto meinKonto = new Konto ( );
      meinKonto.setKontoNr (1000);
      int m = 10;
      System.out.println ("Vorher: " + m);
      setVar (m);
      System.out.println ("Nachher: " + m);

      System.out.println ("Vorher: " + meinKonto.getKontoNr());
      setKonto (meinKonto);
      System.out.println ("Nachher: " + meinKonto.getKontoNr());
   } // main
} // ParamTest
```

Vorbelegungen

- Der Datentyp eines Attributs kann ein primitiver Datentyp oder ein Referenztyp sein.
- Nicht initialisierte Attribute erhalten bei der Objekterzeugung einen Standardwert:
  - ⇨ Für alle primitiven Typen sind dies die gleichen Werte wie beim Anlegen eines Arrays
  - ⇨ Referenzvariablen erhalten den speziellen Wert *null*.
- null-Referenzen
  - ⇨ Die vordefinierte Konstante *null* bezeichnet eine leere Referenz.
  - ⇨ Referenzvariablen mit Wert *null* verweisen nirgendwohin.
- Erinnerung: Vorbelegung erfolgt nur bei komplexen Datentypen
  - ⇨ einfache Variablen müssen immer selbst sinnvoll initialisiert werden

Zugriff auf Attribute/Methoden

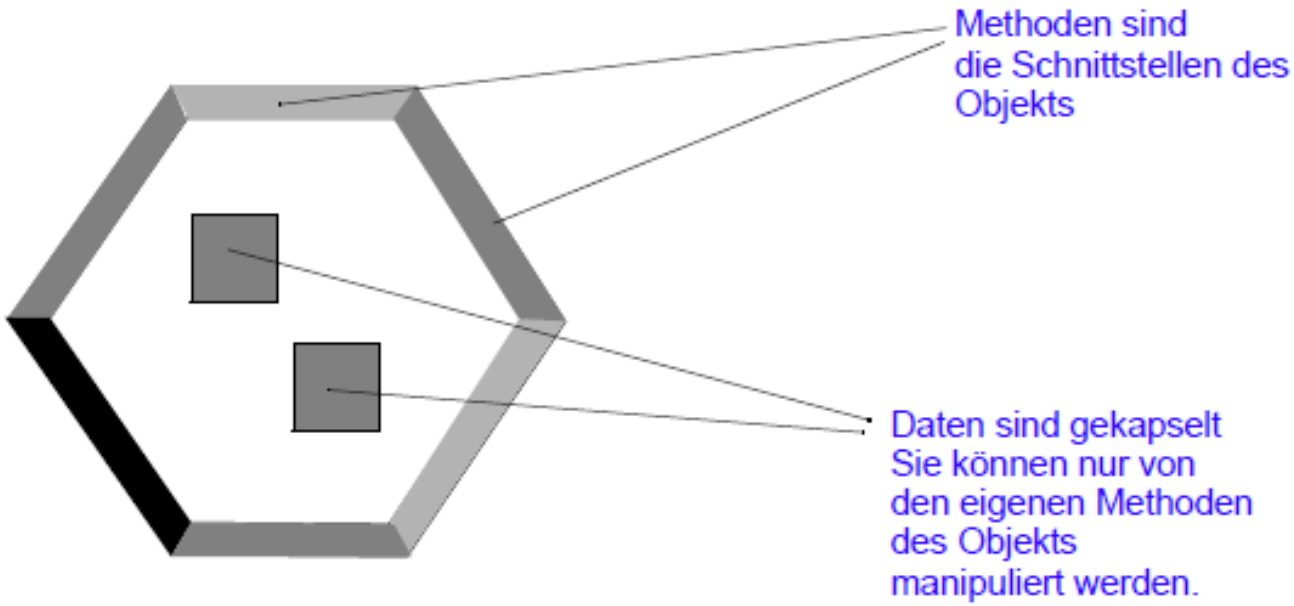
- Um auf Attribute oder Methoden eines Objektes zugreifen zu können, wird die Punktnotation verwendet:
  - ⇨ ObjektReferenz . Attribut
  - ⇨ ObjektReferenz . Methode ( )

- Beispiel:

```
// Datei: KontoTest.java
public class KontoTest {
   public static void main (String [ ] args) {

      Konto meinKonto = new Konto ( );

      meinKonto.setKontoNr(4711);
      meinKonto.einzahlen(500.0);
      meinKonto.einzahlen(10000.0);

      System.out.println(meinKonto.getSaldo());

   } // main
} // KontoTest
```

Information Hiding und Kapselung

- Ein Objekt kann sinnbildlich mit einer Burg verglichen werden:
  - ⇨ Die Daten (Attribute) stellen den Schatz dar, welcher
  - ⇨ durch die Wächter (Methoden) bewacht und verwaltet werden.
  - ⇨ Eine Zugriff auf die Daten kann nur durch einen Auftrag an die Wächter (Methoden) erfolgen.



Spezialisierung

- Elementares Konzept der oo Software-Entwicklung
  - ⇨ Eine Klasse kann explizit als Spezialisierung einer allgemeineren Klasse gekennzeichnet werden
  - ⇨ Beispiel:

Sichtbarkeit

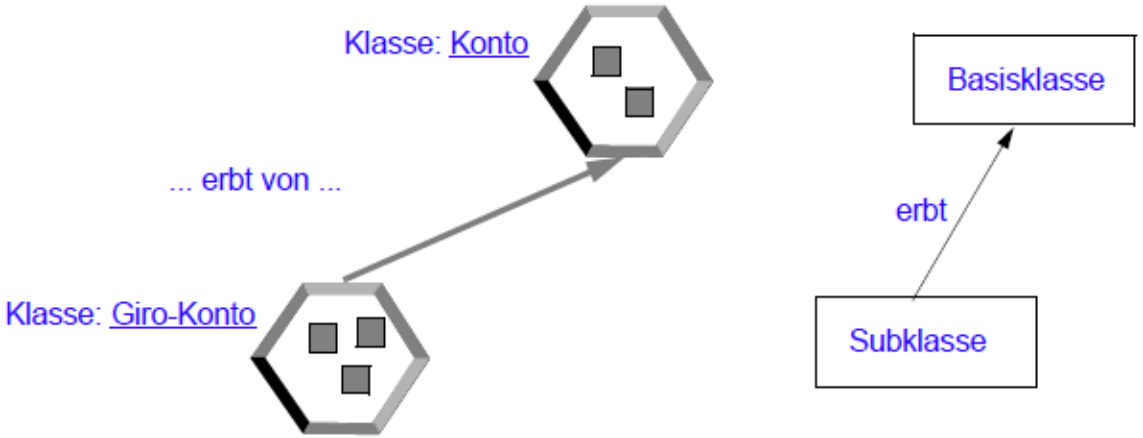
-	→ private
+	→ public
#	→ protected
~	→ package

Sprechweise:
Girokonto *ist ein* Konto

- Bedeutung:
  - ⇨ Mit Instanzen der Spezialisierung kann man alles tun, was man auch mit Instanzen der allgemeineren Klasse tun kann
  - ⇨ Instanzen einer Spezialisierung verfügen über alle Eigenschaften der allgemeineren Klasse – sie können aber zusätzliche Eigenschaften haben
- Eine Spezialisierung „erbt“ alle Eigenschaften der allgemeineren Klasse

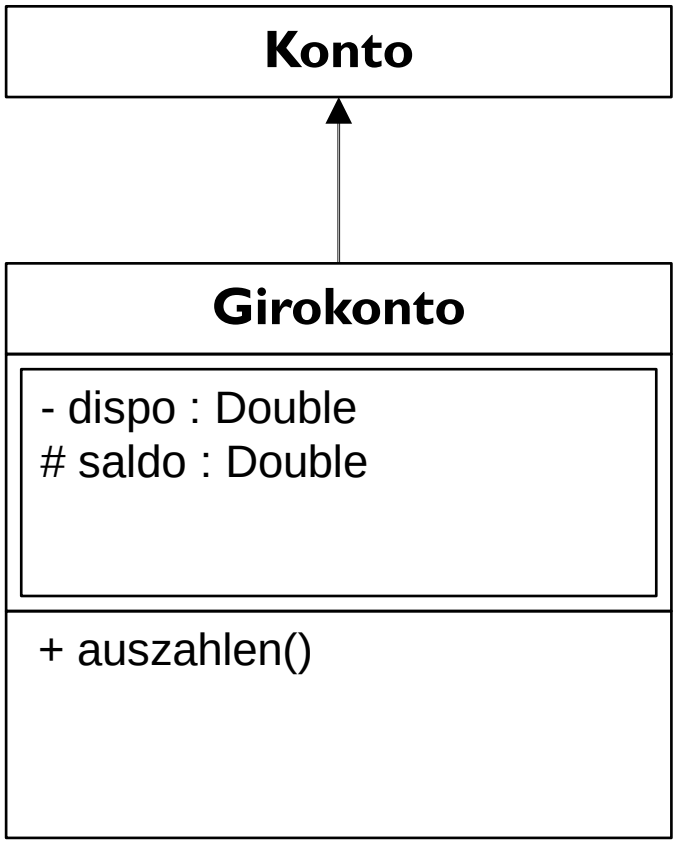
Vererbung

- Vererbung ist Folge einer Spezialisierung
- Wenn Klassen Spezialisierungen (Subklassen) haben, so erben diese alles, was ihre Basisklasse (Elternklasse/Oberklasse/Superklasse) besitzt:
  - ⇨ Eigenschaften und das Verhalten (Attribute und Methoden).
  - ⇨ Subklassen können eigene/zusätzliche Attribute und/oder Methoden hinzubekommen oder auch einiges besser machen – Methoden überschreiben



Zweites Beispiel: Vererbung mit Girokonto

- In UML-Notation wird eine Vererbung durch einen Pfeil gekennzeichnet, der auf das vererbende Element (die Oberklasse) gerichtet ist.



Zu den vererbten Attributen der Elternklasse zusätzliche Attribute

Sichtbarkeit

-	→ private
+	→ public
#	→ protected
~	→ package

Zweites Beispiel: Subklasse Girokonto in Java

```
class Girokonto extends Konto {

   private double dispo;

   void setDispo (double maxUeberziehung) {
      dispo = maxUeberziehung;
   } // setDispo

   @Override
   void auszahlen (double betrag) {
      if (betrag <= saldo + dispo)
         saldo -= betrag;
   } // auszahlen

} // Girokonto
```

extends:
Zeigt an, dass die Klasse von Konto erbt

Methode aus Superklasse wird überschrieben

Vererbung

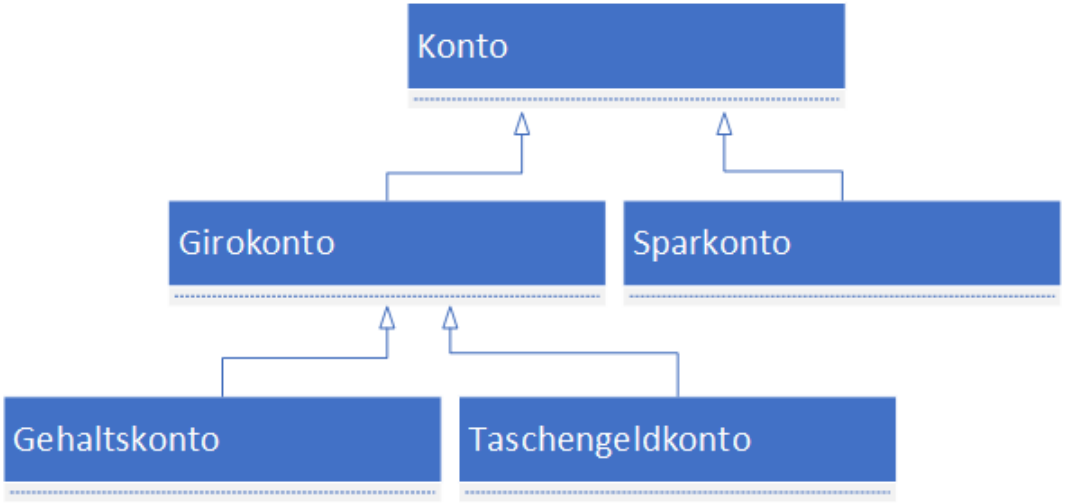
- Neue Klassen können auf Basis bereits vorhandener Klassen mit Hilfe der Vererbung definiert werden.
  - ⇨ Um zu erben wird das Schlüsselwort **extends** zusammen mit dem Namen der Klasse, von der abgeleitet wird, im Kopf der Klasse angegeben.
  - ⇨ Die abgeleitete Klasse wird als *Subklasse* oder *Kindklasse* bezeichnet, die Klasse, die abgeleitet wurde, als *Superklasse* bzw. *Basisklasse* oder *Elternklasse*.
  - ⇨ Die Subklasse übernimmt automatisch alle Attribute und Methoden ihrer Superklasse.
  - ⇨ Zugriff ist jedoch nur auf die nicht als *private* gekennzeichneten möglich.
  - ⇨ Die Subklasse kann natürlich auch eigene Attribute und Methoden besitzen und sogar übernommene Methoden der Superklasse überschreiben.
  - ⇨ Beim Überschreiben einer Methode bleiben ihre Signatur und ihr Rückgabetyp unverändert.
    - Die Anweisungen im Methodenrumpf implementieren die Besonderheiten der Subklasse als Spezialisierung der Superklasse

Überladen von Methoden

- Es können in einer Klasse mehrere Methoden mit gleichem Namen aber unterschiedlichen Signaturen deklariert werden:
  - ⇨ Dieser Vorgang wird als Überladen (Overloading) bezeichnet.
  - ⇨ Die Eigenschaft des Überladens wird verwendet, um je nach Kontext der Argumente, auf welche die Methode angewendet wird, eine andere passende Bedeutung anzubieten.
- Innerhalb einer Klasse dürfen aber nicht zwei Methoden mit genau der selben Signatur deklariert werden.
- Der Typ des Rückgabewertes zählt nicht zur Signatur. Zwei Methoden, die sich nur im Rückgabetyp unterschieden (ansonsten dieselbe Signatur aufweisen) sind nicht möglich und liefern einen Compiler-Fehler, z.B.:
  - ⇨ int berechne (int n) { . . . }
  - ⇨ float berechne (int n) { . . . }
- Nicht verwechseln mit „überschreiben“ !

Vererbungshierchie

- Spezialisierungen können durchaus selbst wieder Spezialfälle haben
  - ⇨ führt zu einer Vererbungshierarchie
- Beispiel:



- Die Spezialisierung – und damit auch die Vererbung – kann über beliebig viele Stufen erfolgen.

Wurzel der Klassenhierarchie: Object

- Eine Klasse, die nicht explizit von einer anderen Klasse erbt, ist automatisch von der in Java definierten Klasse *Object* abgeleitet.
  - ⇨ *Object* selbst besitzt keine Superklasse und ist damit die Wurzel der Klassenhierarchie von Java.

- Durch das Erben von Object stehen viele nützliche Methoden für alle Objekte zur Verfügung:
  - ⇨ String toString()
  - ⇨ boolean equals(Object obj)
  - ⇨ int hashCode()
  - ⇨ Object clone()

Mehrfachvererbung

- Von Mehrfachvererbung spricht man, wenn eine Klasse die Attribute und das Verhalten mehrerer anderer Klassen erbt.

- Java unterstützt keine Mehrfachvererbung (im Ggs. zu z.B. C++)!
  - ⇨ viele Aspekte können jedoch mit Hilfe des Konstruktes *interface* (siehe später) simuliert werden.

Polymorphie

- Polymorphie ist neben Kapselung und Vererbung ein weiteres essentielles Konzept des objektorientierten Ansatzes.
- Polymorphie bedeutet *Vielgestaltigkeit*
  - ⇨ Es bezeichnet die Eigenschaft, dass ein konkretes Objekt mehrere Gestalten annehemen kann – es kann konkret in die Haut jeder Oberklassen der eigenen Klasse schlüpfen.
  - ⇨ Der Aufruf einer Methode kann so unterschiedliche Reaktionen auslösen, je nachdem, auf was für einem Objekt der Aufruf erfolgte
- Polymorphie von Operationen bedeutet, dass eine Operation in verschiedenen Klassen durch eine jeweils eigene Methode, welche den Namen der Operation trägt, implementiert wird.
  - ⇨ Gleiche Methodenköpfe in verschiedenen Klassen stellen kein Problem dar, da jede Klasse einen eigenen Namensraum bildet.

Polymorphie

- Eine Polymorphie von Objekten gibt es nur im Kontext von Vererbungshierarchien.
  - ⇨ Instanz einer abgeleiteten Subklasse ist nicht nur vom Typ dieser Subklasse, sondern auch vom Typ jeder Basisklasse.
  - ⇨ Tritt eine Instanz einer Unterklasse an die Stelle eines Objektes einer Basisklasse, so wird der spezialisierte Anteil der Unterklasse ausgeblendet.
  - ⇨ Auf diese Weise kann ein Objekt einer abgeleiteten Klasse in der Gestalt eines Objekts einer Basisklasse auftreten und sich damit vielgestaltig also *polymorph* verhalten
- Beispiel:
  - ⇨ Eine Instanz von Girokonto darf überall verwendet werden, wo ein Konto erwartete wird
  - ⇨ Konto k = new GiroKonto();
  - ⇨ Hinweis: Einer Variablen von Typ *Object* darf jede beliebige Instanz zugewiesen werden.

Komplexitätsreduktion durch Abstraktion

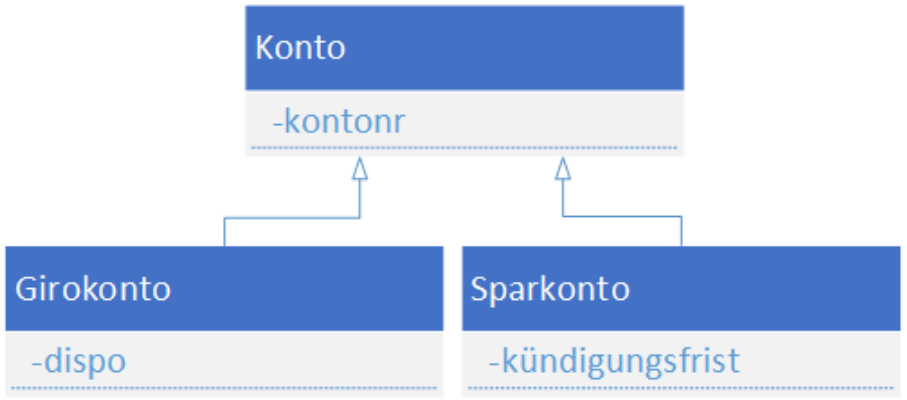
- Das Prinzip der Abstraktion ist ein wichtiger Schlüssel für die Entwicklung einer guten Software.
  - ⇨ Die Kunst der Abstraktion besteht darin, das Wesentliche zu erkennen und das Unwesentliche wegzulassen.
- Abstraktion ist in allen Software-Entwicklungsphasen ein effizientes Mittel, um die Komplexität eines Systems überschaubar zu machen.
  - ⇨ Neben dem Geheimnisprinzip kann dazu auch die Bildung von Hierarchien als Mittel eingesetzt werden.
  - ⇨ Dabei unterscheidet man in der Objektorientierung zwei wichtige Hierarchien:
    - Die Vererbungs-Hierarchie („child of“-Beziehung) und
    - die Zerlegungs-Hierarchie („part of“-Beziehung).

Abstraktion und Zerlegung

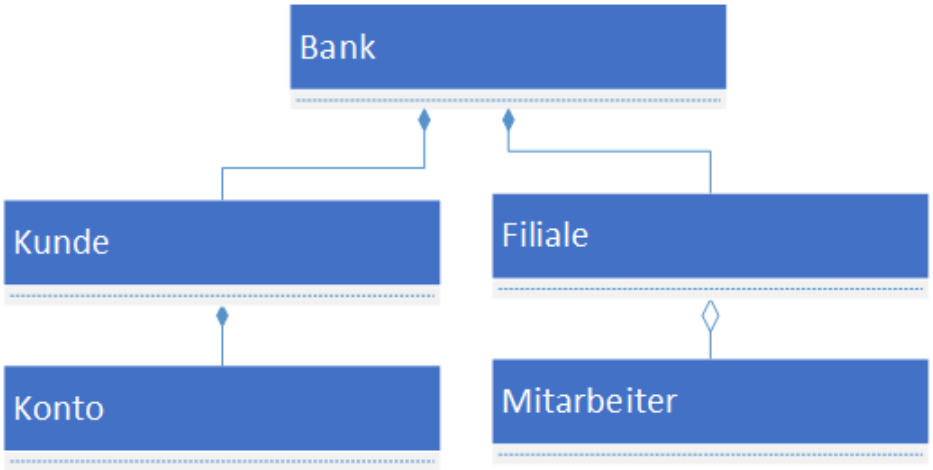
- Komplexität kann durch Abstraktion oder Aggregation reduziert werden
- Durch Anordung von Klassen in Abstraktionsebenen entstehenVererbungshierarchien
  - ⇨ Die jeweils tiefer stehende Klasse ist eine Spezialisierung der Klasse, von der sie abgeleitet ist.
  - ⇨ In umgekehrten Richtung (von unten nach oben) spricht man von Generalisierung.
- Zerlegungshierarchien entstehen durch den Aufbau komplexer Objekte aus einfacheren
  - ⇨ Die Basis sind Aggregationsbeziehungen zwischen den beteiligten Objekten
  - ⇨ Ein Objekt kann dabei als Datenfelder andere Objekte in Form einer Komposition oder einer Aggregation enthalten.

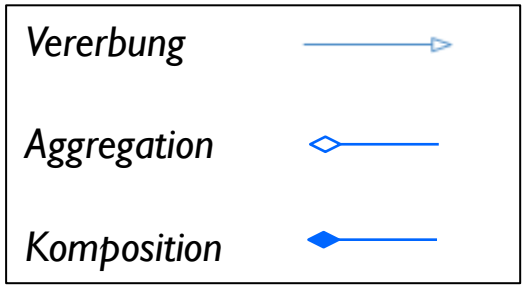
Grafik: Vererbungs- und Zerlegungshierarchie

- Vererbungshierarchie



- Zerlegungshierarchie





Lokale Variablen, this

- Innerhalb von Methoden ist es manchmal notwendig, auf das Objekt auf dem die Methode aufgerufen wurde zuzugreifen.
  - ⇨ objektorientierte Sprachen benötigen dazu eine spezielle Variable, die eine Referenz auf das aktuelle Objekt enthält
  - ⇨ unterschiedliche Bezeichnungen:
    - *self* (Smalltalk, Python), *me* (ABAP, VisualBasic), …
  - ⇨ In Java heisst diese Variable: *this* (auch in C++)
- **this** ist eine Referenzvariable, die auf das aktuelle Objekt selbst zeigt
  - ⇨ kann genutzt werden, um in Methoden auf Attribute und Methoden der eigenen Klasse zuzugreifen oder
  - ⇨ die eigene Instanz als Wert zurückzugeben oder
  - ⇨ sie als Argument beim Aufruf einer Methode einer andere Klasse zu verwenden

Konstruktoren / Destruktoren

- Konstruktoren: spezielle Methoden, die bei der Erzeugung eines Objektes aufgerufen werden und Initialisierungen sowie besondere Aufgaben ausführen können.
- Destruktoren: spezielle Methoden, die bei der Löschung eines Objektes aufgerufen werden
- Im Gegensatz zu den meisten anderen objektorientierten Programmiersprachen gibt es in Java keine Destruktoren, die bei der Zerstörung eines Objektes garantiert aufgerufen werden
  - ⇨ z.B. um entsprechende komplementäre Aufräumarbeiten (z.B. Speicherfreigabe) vorzunehmen.
  - ⇨ ähnliches Konzept: finalize() – aber Aufruf wird nicht garantiert

Konstruktoren in Java

- Ein Konstruktor trägt den **Namen** der zugehörigen **Klasse** und **hat keinen Rückgabetyp**.
  - ⇨ Ansonsten wird er wie eine Methode deklariert und kann damit auch überladen werden.
- Wenn für eine Klasse kein Konstruktor explizit deklariert ist,
  - ⇨ wird ein Standardkonstruktor ohne Parameter vom Compiler eingebaut.
  - ⇨ Achtung: Ist ein Konstruktor mit oder ohne Parameter explizit deklariert, so erzeugt der Compiler keinen Standardkonstruktor mehr.

Beispiel Konstruktoren: Konto

```
class Konto {
   int kontoNr;
   double saldo;

   Konto ( ) { // Standardkonstruktor
   } // Konto

   Konto (int kontonummer) {
      this.kontoNr  = kontonummer;
   } // Konto

   Konto (int kontonummer, double saldo) {
      this.kontoNr = kontonummer;
      this.saldo = saldo;
   } // Konto
   . . .

   int getKontoNr ( ) {
      return kontoNr;
   } // getKontoNummer
} // Konto
```

Test-Beispiel

```
public class KonstruktorTest {

   public static void main (String [ ] args) {
      Konto k1 =  new Konto ( );
      Konto k2 =  new Konto (4177);
      Konto k3 =  new Konto (1234, 1000.);

      System.out.println (k1.getKontonummer ());
      System.out.println (k1.getSaldo ());

      System.out.println (k2.getKontonummer ());
      System.out.println (k2.getSaldo ());

      System.out.println (k3.getKontonummer ());
      System.out.println (k3.getSaldo ());
   } // main

} // KonstruktorTest
```

Überschriebene Methoden und Konstruktoren

- Eine überschriebene Methode *method* der Subklasse verdeckt die ursprüngliche Methode *method* der Superklasse.
  - ⇨ Mit Hilfe des Schlüsselworts *super* kann die Superklassen-Methode *method* aufgerufen werden: *super.method ()*
- Innerhalb eins Konstruktors der Subklasse kann ein Konstruktor der Oberklasse mittels super (...) als erste Anweisung aufgerufen werden.
- Das folgende Beispiel veranschaulicht den Sachverhalt:
  - ⇨
```
Girokonto (int kontonr, double saldo, double dispo) {
    super (kontonr, saldo);
    this.dispo = dispo;
}
```

Überschriebene Methoden und Konstruktoren

- Fehlt der Aufruf von super(...) in einem Konstruktor der Subklasse, so setzt der Compiler automatisch den Aufruf super() ein.
  - ⇨ Fehlt der parameterlose Konstruktor der Superklasse, erzeugt der Compiler einen Fehler.
- Besitzt die Subklasse keinen expliziten Konstruktor, so erzeugt der Compiler automatisch einen parameterlosen Konstruktor.
  - ⇨ der lediglich den Aufruf des parameterlosen Konstruktors der Superklasse enthält

Wrapper-Klassen

- Die primitiven Typen wie int, char, float, double, boolean, … sind keine Objekte und können daher nicht Variablen von Typ `Object` zugewiesen werden.
  - ⇨ Lösung des Problems in Java durch sogenannte *Wrapper-Klassen*
- Eine Wrapper-Klasse erlaubt die Nutzung aller Vorteile der Objektorientierung auf elementaren Werten
  - ⇨ die elementaren Werte werden in Objekte, die den Wert repräsentieren eingepackt.
- Wrapper-Klassen:
  - ⇨ Boolean, Character, Byte, Short, Integer, Long, Float, Double
  - ⇨ Diese sind – so wie alle anderen Java-Klassen – von Object abgeleitet.

Wrapper Klassen: Verwendung

- Einhüllen kann jeweils über einen Konstruktor erfolgen
  - ⇨ Double d_obj = new Double(42.0);
  - ⇨ Integer i_obj = new Integer(42);
- Auspacken über spezielle Methode: *primitiveType*Value()
  - ⇨ double d = d_obj.doubleValue();
  - ⇨ int i = i_obj. intValue();
- Auspacken über einfache Zuweisung (Outounboxing)
  - ⇨ double d = d_obj;
  - ⇨ int i = i_obj;
- Zusätzlich zahlreiche nützliche Funktionen
  - ⇨ Z.B. zur Konvertierung in/aus Strings, …

Autoboxing, Autounboxing

- Die Umwandlung von Wrapper-Objekten in einfache Werte wurde in Java besonders komfortabel gestaltet.
  - ⇨ Bei Bedarf erfolgt Umwandlung automatisch: Autoboxing
  - ⇨ Das Autoboxing, Autounboxing erfolgt z.B. bei der Verwendung von Operatoren automatisch
  - ⇨ Achtung: Beim Vergleich mit == und != werden weiterhin nur Referenzen verglichen
- Beispiele:
  - ⇨
```
Integer i_obj1 = 42;          // Autoboxing
Integer i_obj2 = ++i_obj1;    // Autounboxing, Addition, Autoboxing
```

Statische Attribute und Methoden

- In Java können Attribute und Methoden deklariert werden, deren Nutzung nicht an die Existenz von Objekten gebunden ist.
  - ⇨ Diese kennen wir schon aus dem nicht-oo Teil
- Ein Attribut einer Klasse, dessen Deklaration mit dem Schlüsselwort **static** versehen ist, nennt man **Klassenvariable.**
- Nicht statische Attribute werden zur Unterscheidung auch **Instanzvariablen** genannt.

Statische Attribute und Methoden

- Klassenvariablen sind nicht ein Teil von Objekten, sondern werden bei der Klasse geführt und sind deshalb für alle Objekte einer Klasse nur einmal vorhanden.
  - ⇨ Es existiert nur ein Exemplar dieser Variablen, unabhängig von der Anzahl der Instanzen (Objekte) der Klasse.
  - ⇨ Ihre Lebensdauer erstreckt sich über das ganze Programm.
  - ⇨ Der Zugriff von Außen auf Klassenvariablen ist ohne die Existenz einer Instanz einer Klasse möglich und kann über den Klassennamen in der Form ***Klassenname.Variablename*** erfolgen oder
  - ⇨ wahlweise auch über eine (beliebige) Referenz auf ein Objekt der entsprechenden Klasse, da Objekt weiß, zu welcher Klasse es gehört.

Beispiel: Klassenvariable

```
public class Konto {
  private int kontoNr;
  public static int maxKontoNr = 10000; // Startkontonummer

  public Konto () { // Standardkonstruktor
    kontoNr = maxKontoNr ++;
  }
  public String toString() {
    return "Konto mit kontoNr=" + kontoNr;
  }
} // Konto
```

```
public class KontoTest {
  public static void main (String [] args);
    System.out.println ("maxKontoNr vorher: " + Konto.maxKontoNr);
    Konto[] konten = new Konto[5]; // Array für 5 Konten

    for (int i=0; i < konten.length; i++)
      konten[i] = new Konto();

    for (Konto k : konten)
      System.out.println(k.toString());
    System.out.println ("maxKontoNr nachher: " + Konto.maxKontoNr);
  } // main
} // KontoTest
```

Statische Initialisierung

- Klassenvariablen können neben der üblichen (Werte-) Zuweisung bei der Deklaration durch einen Block, der mit dem Schlüsselwort static eingeleitet ist, initialisiert werden.
  - ⇨ Eine Klasse kann mehrere Blöcke dieser Art haben.
  - ⇨ Sie werden in der aufgeführten Reihenfolge ausgeführt
- Beispiel:

```
static int n;
static int m;

static {
   n = 2;
   m = 5 * n;
} // static
```

Klassenmethode

- Im Gegensatz zu **Klassenmethoden** sind **Instanzmethoden** immer an ein bestimmtes Objekt gebunden.
  - ⇨ Mit Hilfe von **Klassenmethoden** ist daher ein Zugriff auf Instanzvariablen der Klasse nicht möglich.
  - ⇨ Ein Zugriff auf Instanzvariablen und damit die vollständige Bearbeitung von Objekten ist nur mit **Instanzmethoden** möglich.
- Eine Methode einer Klasse, deren Deklaration in Java mit dem Schlüsselwort ***static*** versehen ist, nennt man **Klassenmethode**.
  - ⇨ Klassenmethoden können von außen über den Klassennamen oder eine Instanz aufgerufen werden.
  - ⇨ Man braucht keine Instanz der Klasse, um sie aufrufen zu können.
  - ⇨ Es ist ein guter Programmstil, wenn Klassenmethoden nur über den Klassennamen angesprochen werden.

Beispiel: Klassenmethode I

- Zum externen Zugriff auf private static Attribute werden public static Methoden benötigt.
- Beispiel:

```
public class Konto {
  private int kontoNr;
  private static int maxKontoNr = 10000;

  public Konto () { // Standardkonstruktor
    kontoNr = maxKontoNr ++;
  }
  public String toString() {
    return "Konto mit kontoNr=" + kontoNr;
  }

  public static int getMaxKontoNr() {
    return maxKontoNr;
  }
} // Konto
```

Kann nun von außen nicht mehr verändert werden

Beispiel: Klassenmethode II

- Aufruf der public static Methode getMaxKontoNr

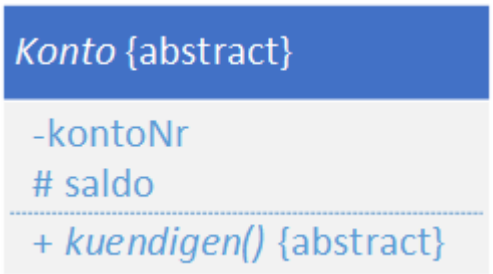
```
public class KontoTest {
  public static void main (String [] args);
    System.out.println ("maxKontoNr vorher: " + Konto.getMaxKontoNr());
    Konto[] konten = new Konto[5]; // Array für 5 Konten

    for (int i=0; i < konten.length; i++)
      konten[i] = new Konto();

    for (Konto k : konten)
      System.out.println(k.toString());

    System.out.println ("maxKontoNr nachher: " + Konto.getMaxKontoNr());
  } // main
} // KontoTest
```

GirokontoTest.java

```
public class GirokontoTest  {

   public static void main (String [ ] args) {

      Girokonto giro = new Girokonto (4711, 500., 2000.);
      System.out.println (giro.getSaldo () );

      giro.auszahlen (3000.);
      System.out.println (giro.getSaldo () );

      giro.setDispo (2500.);
      giro.auszahlen (3000.);
      System.out.println (giro.getSaldo () );

   } // main

} // GirokontoTest
```

Abstrakte Klassen

- *Abstrakte Klassen* enthalten Definition von Datentypen und Deklarationen von Methoden ohne Implementierungen.
- Methoden ohne Implementierung werden als *abstrakte Methoden* bezeichnet.
- Abstrakte Klassen können nicht instanziiert werden
  - ⇨ Instanziierbare Subklassen müssen alle abstrakten Methoden implementieren
- In UML werden abstrakte Klassen kursiv geschrieben und mit der Eigenschaftsangabe {abstract} gekennzeichnet
  - ⇨ Beispiel:



Abstrakte Klassen in Java

- In Java werden abstrakte Klassen mit dem Schlüsselwort *abstract* gekennzeichnet
  - ⇨ Von abstrakten Klassen kann kein Objekt erzeugt werden; sie dienen nur als Basisklasse bei der Vererbung
- In abstrakten Klassen können abstrakte Methoden auftreten
- Abstrakte Methoden:
  - ⇨ werden mit dem Schlüsselwort *abstract* gekennzeichnet
  - ⇨ Sie besitzen keinen Anweisungsblock.
  - ⇨ Sie müssen in Subklassen überschrieben werden,
  - ⇨ Können nur in abstrakten Klassen auftreten!

Beispiel: Figur.java

```
// Datei Figur.java
abstract class Figur  {

   int x, y;

   Figur (int x, int y) {
      this.x = x;
      this.y = y;
   } // Konstruktor

   abstract void zeichne ( );
   abstract double getFlaeche ();

} // Figur
```

- Idee:
  Abstrakte Figur zur Darstellung eines grafischen Objektes;
  Beschreibung nur durch einen Referenzpunkt

Beispiel: Rechteck.java

```
// Datei Rechteck.java
class Rechteck extends Figur  {
   int breite, hoehe;

   Rechteck (int x, int y, int breite, int hoehe) {
      super(x,y);
      this.breite = breite;
      this.hoehe = hoehe;
   } // Konstruktor

   @Override
   void zeichne ( ) {
      System.out.println ("Rechteck mit Bezugspunkt: (" + x + ", " + y
                          + "), Breite: " + breite + ", Hoehe " + hoehe);
   }

   @Override
   double getFlaeche ( ) {
      return breite * hoehe;
   }
} // Rechteck
```

Beispiel: Kreis.java

```
// Datei Kreis.java
class Kreis extends Figur  {
   int radius;

   Kreis (int x, int y, int radius) {
      super (x,y);
      this.radius = radius;
   } // Konstruktor

   @Override
   void zeichne ( ) {
      System.out.println ("Kreis mit Bezugspunkt: (" + x + ", " + y
                                 + "), Radius: " + radius);
   }

   @Override
   double getFlaeche ( ) {
      return radius * radius * 3.14159;
   }
} // Kreis
```

Beispiel: FigurTest.java

```
// Datei: FigurTest.java
public class FigurTest  {

   public static void main (String [ ] args) {
      Figur [ ] f = new Figur [3];
      f[0] = new Kreis (10, 10, 5);
      f[1] = new Rechteck (0, 0, 20, 6);
      f[2] = new Kreis (100, 50, 12);

      for (int i = 0; i <3; ++i) {
         f[i].zeichne ();
         System.out.println ("Flaeche: " + f[i].getFlaeche () );
      } // for

   } // main

} // FigurTest
```

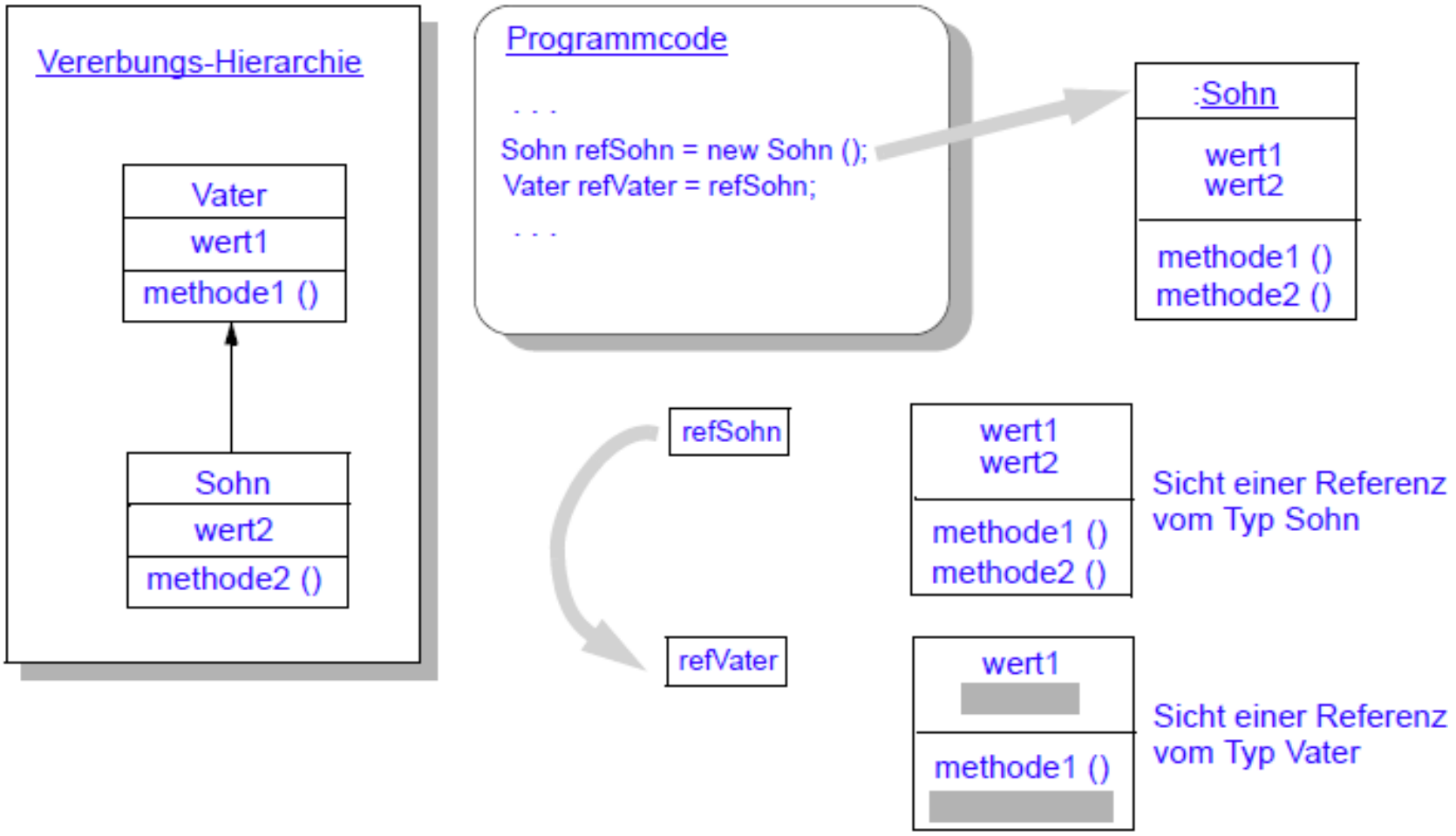
Kompatibilität von Zuweisungen

- In Java ist es nicht unbedingt erforderlich, dass der Typ einer Referenzvariablen identisch mit dem Typ des Objektes ist, auf das die Referenzvariable zeigt.
- Genausowenig muss bei einer Zuweisung der Typ der an der Zuweisung beteiligten Referenzen identisch sein.
- Es gibt wie bei der Typkonvertierung von einfachen Datentypen auch bei Referenztypen eine **implizite** (automatische) **Typkonvertierung** und eine **explizite Typkonvertierung** mit Hilfe des **cast-Operators**

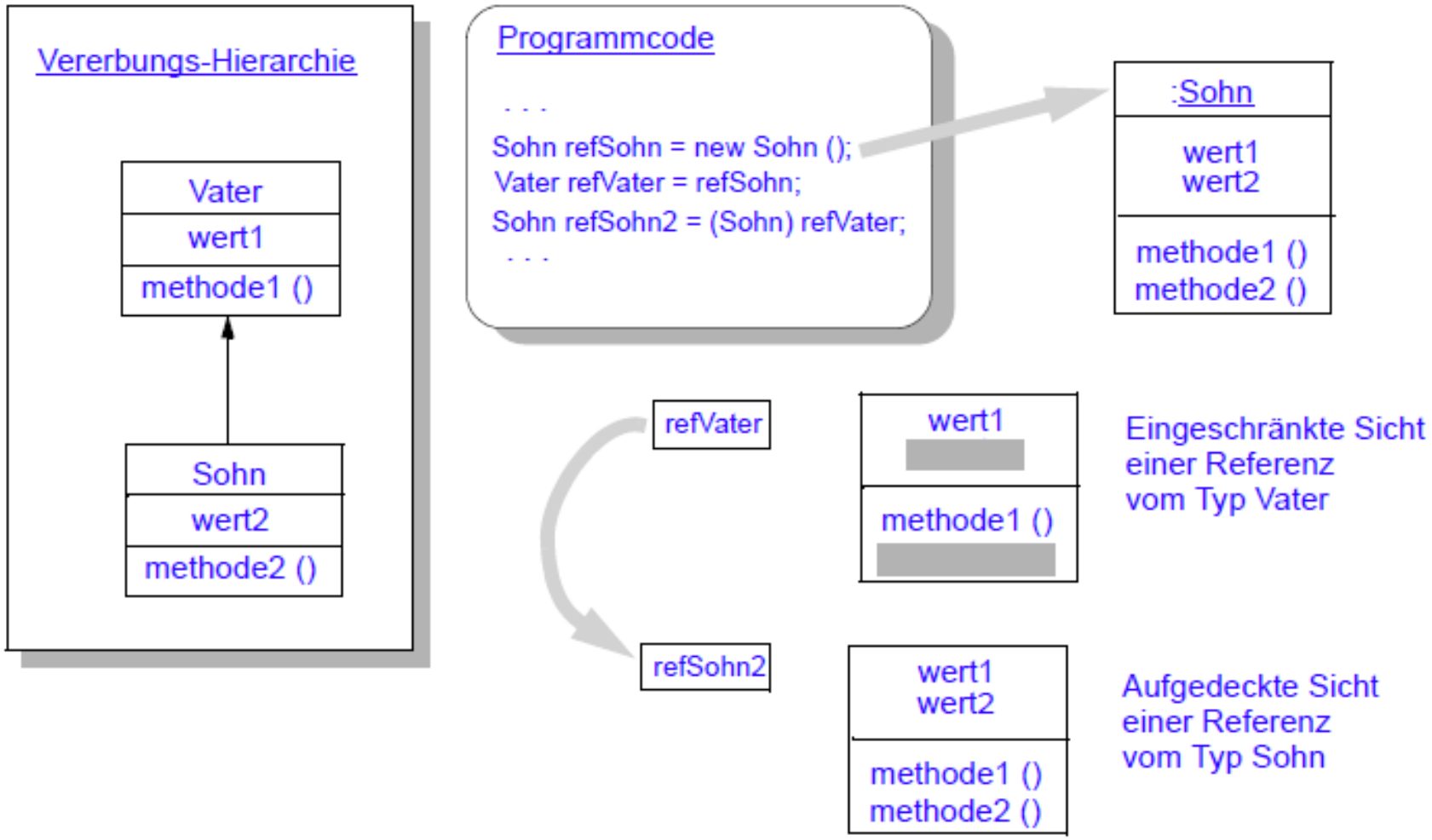
Kompatibilität von Zuweisungen

- Up-Cast:
  - ⇨ Cast in einen Typ, der in der Vererbungshierarchie weiter oben liegt
- Down-Cast:
  - ⇨ Cast in einen Typ, der in der Vererbungshierarchie weiter unten liegt.
- Bei einem gültigen Up-Cast ist es nie erforderlich, den cast-Operator anzugeben.
  - ⇨ Dies erfolgt also implizit (automatisch) durch den Compiler.
- Dagegen erfordert ein Down-Cast immer die explizite Angabe des cast-Operators

Implizierter Up-Cast



Expliziter Down-Cast



KompaTest.java

```
public class KompaTest  {

   public static void main (String [ ] args) {

      Konto k = new Konto (5000.); // mit saldo
      k.auszahlen (1000.);
      System.out.println (k.getSaldo () );     // Zustand (1)

      k = new Girokonto (800., 2000.); // mit saldo und dispo
      k.auszahlen (3000.);
      System.out.println (k.getSaldo () );     // Zustand (2)

      ((Girokonto) k).setDispo (4000.);        // Zustand (3)

      k.auszahlen (3000.);
      System.out.println (k.getSaldo () );

   } // main

} // KompaTest
```

instanceof

- Operator *instanceof :* Verwendung zur Feststellung, zu welcher Klasse ein bestimmtes Objekt gehört
  - ⇨ *rf* instanceof *clx*
  - ⇨ Ausdruck liefert den Wert true, wenn die Variable *rf* auf eine Instanz der Klasse *clx* **oder** auf eine Instanz einer von *clx* abgeleiteten Klasse zeigt.
- Beispiel:
  - ⇨ Was wird ausgegeben?

```
k = new Girokonto (800., 2000.);

System.out.println("k ist ein Konto: " + (k instanceof Konto));
System.out.println("k ist ein Girokonto: " + (k instanceof Girokonto));
System.out.println("k ist ein Sparkonto: " + (k instanceof Sparkonto));
```

Modifizierer (Modifier)

- Mit Hilfe von Modifizierern (bestimmten Schlüsselwörtern in Java) können die Eigenschaften von Klassen, Attributen und Methoden verändert werden.
- Folgende Tabelle führt alle Modifizierer auf und gibt an, wo sie verwendet werden können:

Modifier	Klasse	Attribut	Methode	Konstruktor
public	X	X	X	X
protected		X	X	X
private		X	X	X
static		X	X	
final	X	X	X	
abstract	X		X	
native			X	
synchronized			X	

Zugriffschutz für Attribute und Methoden: Private Modifier

```
class Konto {
   private int kontonummer;
   private double saldo;

   Konto(int nummer, double betrag) {
      kontonummer = nummer; saldo = betrag;
   } // Konto-Konstruktor

   getKontonummer( ) {
      return kontonummer;
   } // getKontonummer
} // Konto

class TestZugriff  {
  public static void main (String [] args) {
    SparGiroKonto  s = new Konto (4711, 1000.);
    System.out.println ("Die Kontonummer : " + s.getKontonummer());
    //System.out.println ("Die Kontonummer : " + s.kontonummer); // Fehler
    //System.out.println ("Der aktuelle Sald : " + s.saldo); // Fehler
   } // main
} // TestZugriff
```

Zugriffschutz für Attribute und Methoden: Default Package

kein Modifier
⇒
uneingeschränkt innerhalb des Packages zugreifbar!

```
class Konto {
    int kontonummer; double saldo;
    Konto (int nummer, double betrag) {
        kontonummer = nummer; saldo = betrag;
    } // Konto-Konstruktor
} // Konto

class GiroKonto extends Konto {
    double dispo;
    GiroKonto (int nummer, double betrag, double grenze) {
        super (nummer, betrag);
        dispo = grenze;
    } // Girokonto-Konstruktor
} // GiroKonto

class TestZugriff  {
    public static void main (String [] args) {
        SparGiroKonto s = new GiroKonto (4711, 1000., 500.);
        System.out.println ("Die Kontonummer : " + s.kontonummer);
        System.out.println  ("Der aktuelle Sald : " + s.saldo);
        System.out.println ("Der Dispo : " + s.dispo);
    } // main
} // TestZugriff
```

Zugriffschutz für Attribute und Methoden: Beliebiges Package

nicht public
⇒
nicht sichtbar
außerhalb
des Packages!

```
package Package1;
public class Konto {
   int kontonummer; double saldo;
   Konto (int nummer, double betrag) {
       kontonummer = nummer; saldo = betrag;
   } // Konto-Konstruktor
} // Konto
public class GiroKonto extends Konto {
   double dispo;
   GiroKonto (int nummer, double betrag, double grenze) {
       super (nummer, betrag);
       dispo = grenze;
   } // Girokonto-Konstruktor
} // GiroKonto
```

```
import Package1.*;
class TestZugriff {
  public static void main (String [] args) {
    // SparGiroKonto s = new GiroKonto (4711, 1000., 500.); Fehler!
    // System.out.println ("Die Kontonummer: " + s.kontonummer); Fehler !
    // System.out.println ("Der aktuelle Sald : " + s.saldo); Fehler!
    // System.out.println ("Der Dispo : " + s.dispo);     Fehler!
  } // main
} // TestZugriff
```

Zugriffschutz für Attribute und Methoden: Mehrere Packages

```
package Package1;
public class Konto {
   public int kontonummer;
   public double saldo;
   protected Konto (int nummer,
                    double betrag) {
      kontonummer = nummer;
      saldo = betrag;
   } // Konto-Konstruktor
} // Konto
```

```
package Package2;
import Package1.Konto;
public class GiroKonto extends Konto {
  public double dispo;
  public GiroKonto (int nummer,
                    double betrag,
                    double grenze) {
    super (nummer, betrag);
    dispo = grenze;
  } // GiroKonto-Konstruktor
} // GiroKonto
```

```
import Package1.Konto;
import Package2.GiroKonto;
class TestZugriff  {
  public static void main (String [] args) {
    SparGiroKonto s = new GiroKonto (4711, 1000, 500);
    System.out.println ("Die Kontonummer : " + s.kontonummer);
    System.out.println ("Der aktuelle Saldo : " + s.saldo);
    System.out.println ("Der Dispo : " + s.dispo);
  } // main
} // TestZugriff
```

Zugriffschutz für Attribute und Methoden: Privater Konstruktor

- Werden alle Konstruktoren einer Klasse für private erklärt, so kann von keiner anderen Klasse aus ein Objekt dieser Klasse erzeugt werden
- Nur innerhalb der Klasse selbst ist es noch möglich, Objekte dieser Klasse zu erzeugen
- Wozu kann dieses Verhalten sinnvoll eingesetzt werden?
  - ⇨ z.B. wenn man die Anzahl der lebenden Objekte einer bestimmten Klasse kontrollieren bzw. regulieren will.
- In diesem Fall wird typischerweise ein Singleton-Muster verwendet

Verhindern der Instantiierung einer Klasse: Singleton

```
class Singleton  { // es darf nur eine Instanz existieren
   private static Singleton instance;

   private Singleton( ) { // Aufruf nur in Methoden der eigenen Klasse!
      System.out.println ("Bin im Konstruktor");
   } // Singleton-Konstruktor

   public static Singleton getInstance( ) {
      if (instance == null)
        instance = new Singleton( );
      return instance;
   } // getInstance
} // Singleton

class TestSingleton {
   public static void main (String [] args) {
   // Singleton = new Singleton( );  // würde Fehler liefern
   Singleton = Singleton.getInstance( );  // new wird aufgerufen
   Singleton = Singleton.getInstance( );  // new wird nicht mehr aufgerufen
   } // main
} // TestSingleton
```

Interfaces

- Interfaces sind Schnittstellenbeschreibungen, die festlegen, was man mit der Schnittstelle machen kann.
  - ⇨ Interfaces realisieren keine Funktionalität.
- Allgemeiner Aufbau:
  - ⇨
```
[public] interface Interfacename [extends Interface-Liste]  {
     Konstanten
     Abstrakte Methoden
 } // Interfacename
```
- Regeln
  - ⇨ Für Interfacenamen gelten die gleichen Namenskonventionen wie bei Klassennamen.
  - ⇨ Die Konstanten haben implizit die Modifizierer: **public**, **static** und **final**.
  - ⇨ Die Methoden sind alle implizit immer: **public** und **abstract**

implements

- Interfaces erzwingen, dass ansonsten unterschiedliche Klassen die gleichen Methoden bereitstellen müssen
  - ⇨ Ohne dass eine abstrakte Superklasse vereinbart werden muss!
- Interfaces fördern damit die Klarheit und Lesbarkeit des Programms.
- Verwendung in Klassen:
  - ⇨

```
[Modifizierer] class Klassenname [extends Basisklasse]
                                 [implements Schnittstellen] {
    Attributen-Deklarationen
    Methoden-Deklaration
} // Klassenname
```

implements

- Klassen können ein oder mehrere Interfaces mit Hilfe des Schlüsselworts *implements* implementieren.
  - ⇨ Mehrere Interfacenamen werden durch Kommata voneinander getrennt.
- Jede konkrete Klasse, die ein Interface implementiert, muss alle Methoden des Interface implementieren
  - ⇨ ansonsten ist die Klasse abstrakt
- Ein Interface ist ein Referenztyp, d.h. Variablen können vom Typ eines Interfaces sein.
  - ⇨ Eine Klasse, die ein Interface implementiert, ist vom Typ des Interface.
- Damit lässt sich mit Hilfe von Interfaces das polymorphe Verhalten einer Klasse mit **Mehrfachvererbung** nachbauen !

Beispiel: Kinderkonto.java

- Für verschiedene Kontoarten (Giro- und Sparkonto) sollen Varianten für Minderjährige anlegbar sein
  - ⇨ Falls ein Konto für Minderjährige eröffnet wird, so müssen noch Daten zu den Erziehungsberechtigten erfasst werden können.
  - ⇨ Für alle Minderjährigenkonten wird ein maximaler freier Abhebebetrag festgelegt

```
public interface Kinderkonto {

   double MAX_AUSGABE = 1000;
   void   setErziehungsberechtigt(String ezb);
   String getErziehungsberechtigt();
}
```

Beispiel: KinderGirokonto.java

```
public class KinderGirokonto implements Kinderkonto
                              extends Girokonto {
  private String erziehungsberechtigt;

  public KinderGirokonto(double saldo) { // Konstruktor
    super(saldo, 0); // kein Dispo
   }
   @Override
   public String toString()  {
     return super.toString() +
            " erziehungsberechtigt="+this.erziehungsberechtigt;
   }
   @Override
   public void setErziehungsberechtigt(String ezb)
     this.erziehungsberechtigt = ezb;
   }
   @Override
   public String getErziehungsberechtigt()
     return this.erziehungsberechtigt;
   }
}
```

Beispiel: InterfaceTest.java

```
public class InterfaceTest {

   public static void main (String [] args) {

      KinderGirokonto k = new KinderGirokonto (200.);

      k.setErziehungsberechtigt ("Hugo Klein");
      System.out.println  (k);

      Girokonto gk = k;
      System.out.println (gk.getSaldo());


      Kinderkonto kiko = k;
      kiko.setErziehungsberechtigt ("Hugo Klein");
      System.out.println  (kiko );
   } // main

} // InterfaceTest
```

Garbage Collector

- Durch new reservierter Speicherplatz wird vom Garbage Collector des Java-Laufzeitsystems freigegeben, wenn er nicht mehr gebraucht wird.
- Erkannt wird das daran, dass ein Objekt nicht mehr erreichbar ist
  - ⇨ Wenn z.B. die Methode, in der ein Objekt erzeugt wurde, endet oder
  - ⇨ die Referenzvariable auf den Wert null gesetzt wurde
- Wann der Garbage Collector startet, ist nicht festgelegt
  - ⇨ spätestens jedoch, wenn Speicherplatz zur Erzeugung neuer Objekte benötigt wird oder explizit wenn System.gc () aufgerufen wird.
- Die Methode void finalize () wird automatisch für ein nicht mehr benötigtes Objekt vom Garbage Collector aufgerufen, bevor der durch das Objekt belegte Speicherplatz freigegeben wird.
- Diese Methode kann durch eine Klasse implementiert werden und spezielle Aufräumarbeiten erledigen.