Praktische Informatik für Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler I (PIWIN I, 3 V + 1 Ü) WS 2002/03

8. Vorlesungswoche

Objektorientierte Programmierung: Vererbung

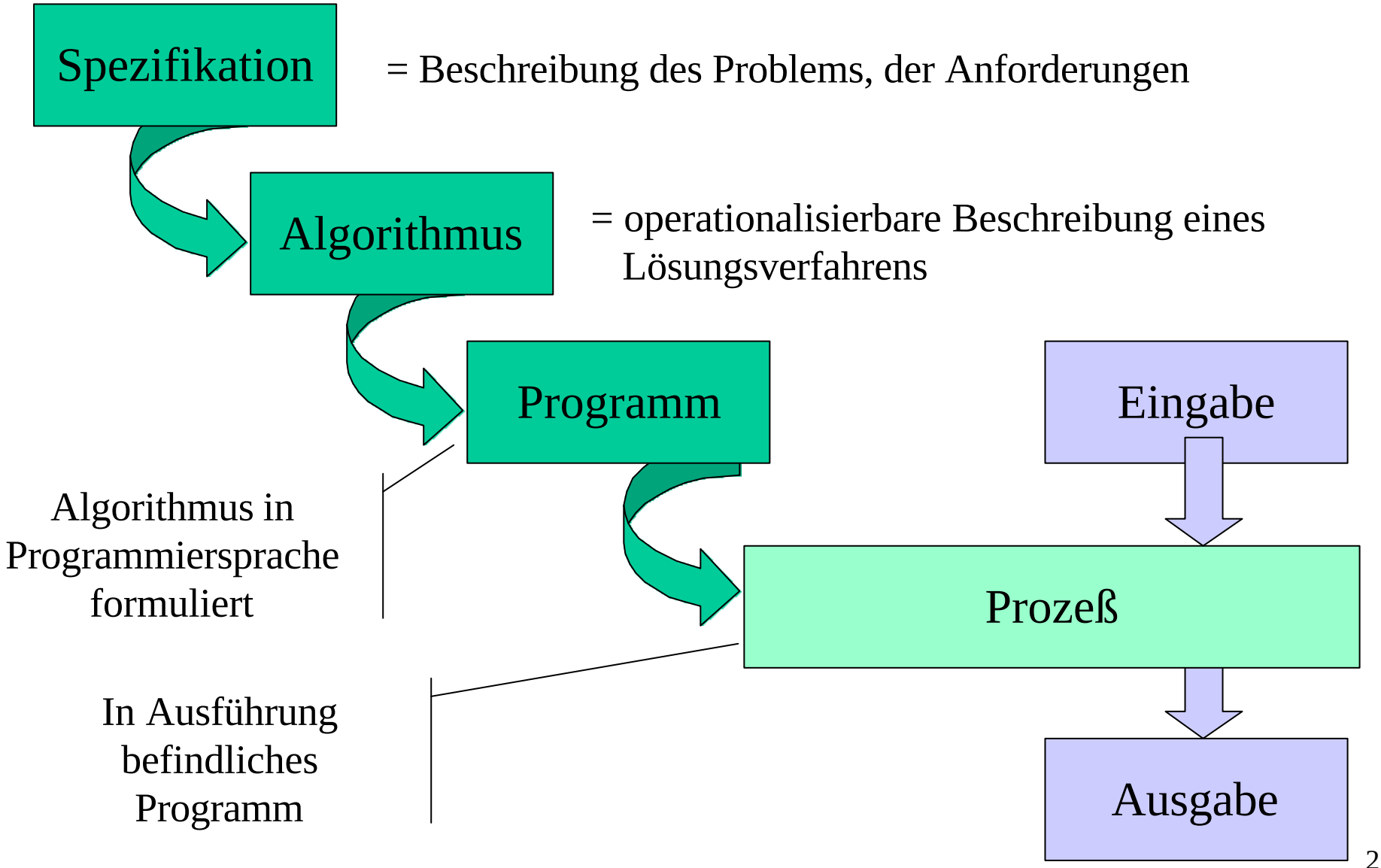
Unterlagen:

Echtle, Goedicke; Einführung in die objektorientierte Programmierung mit Java, dpunkt-Verlag.

Doberkat, Dissmann; Einführung in die objektorientierte Programmierung mit Java, Oldenbourg-Verlag, 2. Auflage.

Folien nach V.Gruhn, Vorlesung Programmierung WS 99/00

Stationen im Entwurf von Algorithmen und Programmen



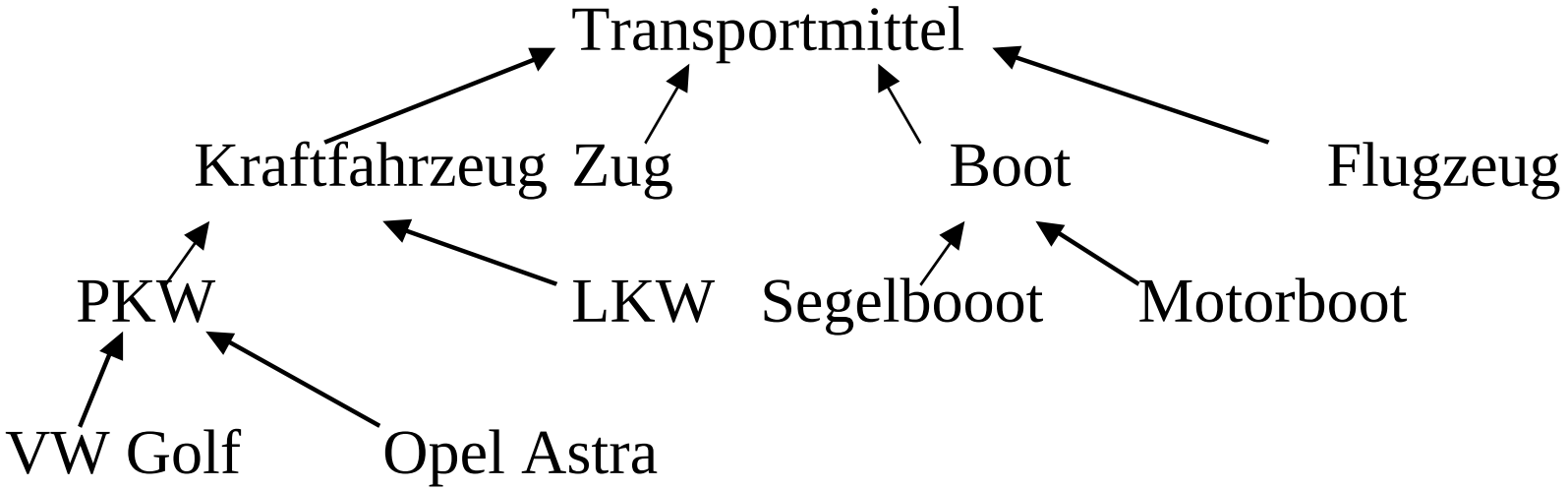
Übersicht

- Begriffe
  - Spezifikationen, Algorithmen, formale Sprachen, Grammatik
- Programmiersprachenkonzepte
  - Syntax und Semantik
  - imperative, objektorientierte, funktionale und logische Programmierung
  - formale Sprachen und Grammatik
- Grundlagen der Programmierung
  - imperative Programmierung:
    - Verfeinerung, elementare Operationen, Sequenz, Selektion, Iteration, funktionale Algorithmen und Rekursion, Variablen und Wertzuweisungen, Prozeduren, Funktionen und Modularität, Zuweisung, Sequenz
  - objektorientierte Programmierung
- Algorithmen und Datenstrukturen
- Berechenbarkeit und Entscheidbarkeit von Problemen
- Effizienz und Komplexität von Algorithmen
- Programmentwurf, Softwareentwurf

Vererbung

- Klassen können zueinander in einer "ist ein"-Beziehung stehen

- Beispiel: Jeder PKW ist ein Kraftfahrzeug,
  jedes Kraftfahrzeug ist ein Transportmittel
  aber: auch jeder LKW ist ein Kraftfahrzeug, jeder Zug,
  jedes Schiff und jedes Flugzeug ist ein Transport-
  mittel



Vererbung (anschaulich)

- Sowohl PKWs, als auch LKWs besitzen *Fahrersitze* und *Fahrertüren*, es handelt sich also um Attribute der Oberklasse Kraftfahrzeug
- Sowohl PKWs als auch LKWs haben die Methoden *Sitz verstellen*, *Tür schließen* und *fahren*, es sind Methoden der Oberklasse Kraftfahrzeug
- PKWs haben jedoch mit der *Rückbank* und dem *Kofferraum* eigene Attribute und mit "*hinten einsteigen*" eigene Methoden
- LKWs haben mit der *Ladefläche* und dem *Anhänger* auch eigene Attribute und "*beladen*" ist eine eigene Methode
  => Unterklassen PKWs und LKWs besitzen alle Attribute und Methoden der Oberklasse, und einige weitere …

Ähnlichkeiten bei Objekten

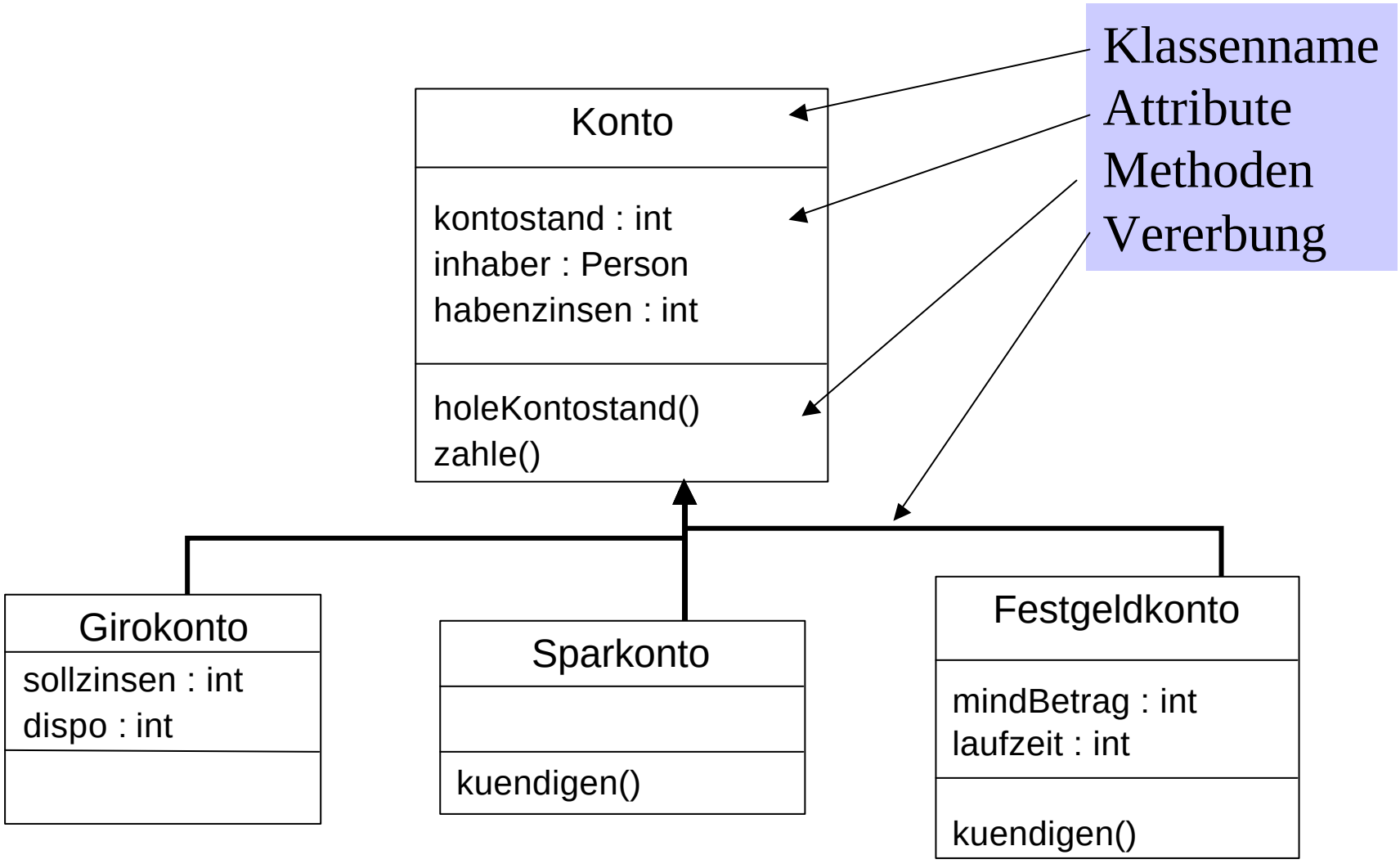
- Und bei Konten tritt so was ähnliches natürlich auch auf :-)
- Zusammenfassen von gleichen Attributen und Methoden

Girokonto
kontostand : int **inhaber : Person** **habenzinsen : int** sollzinsen : int dispo : int
holeKontostand() **zahle()**

Sparkonto
kontostand : int **inhaber : Person** **habenzinsen : int**
holeKontostand() **zahle()** kuendigen()

Festgeld
kontostand : int **inhaber : Person** **habenzinsen : int** mindBetrag : int laufzeit : int
holeKontostand() **zahle()** **kuendigen()**

Graphische Darstellung von Klassen und Vererbung



Begrifflichkeiten

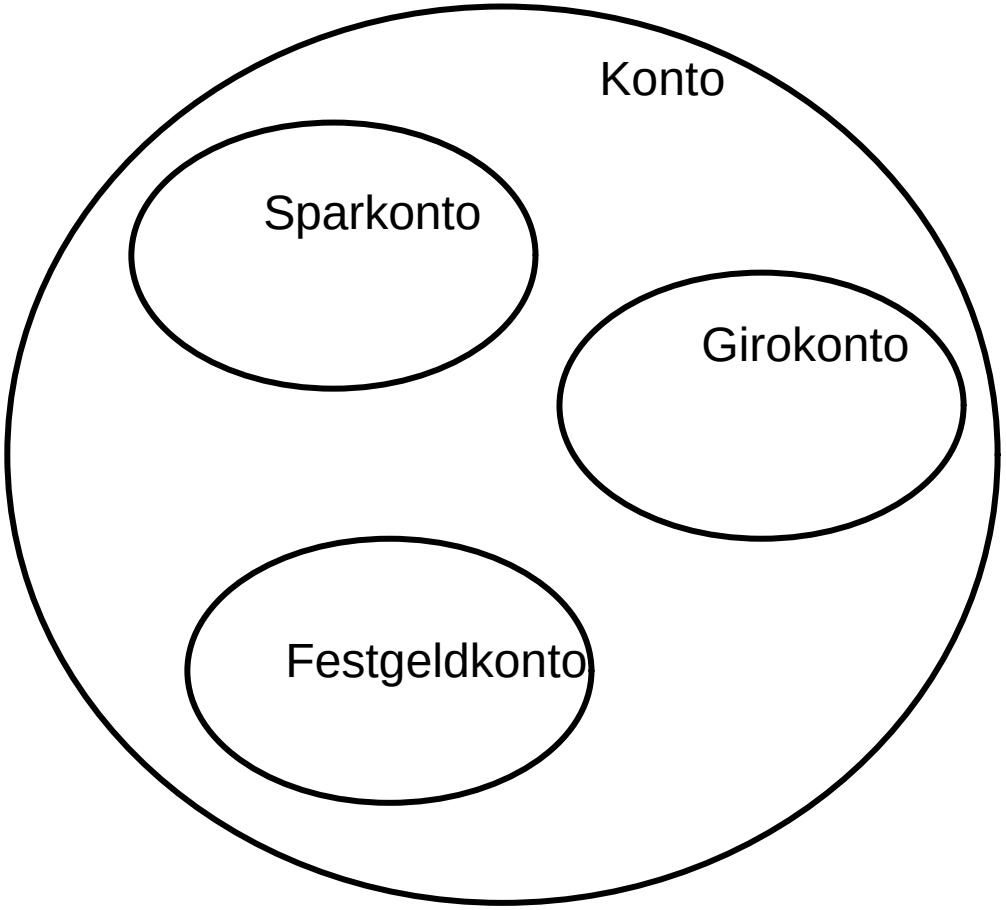
- Die vererbende Klasse heißt Superklasse.
- Die erbenden Klassen sind Unter- oder Subklassen.
- Konto ist also die Superklasse der Klassen Girokonto, Festgeldkonto, Sparkonto. Diese sind die Subklassen der Klasse Konto.

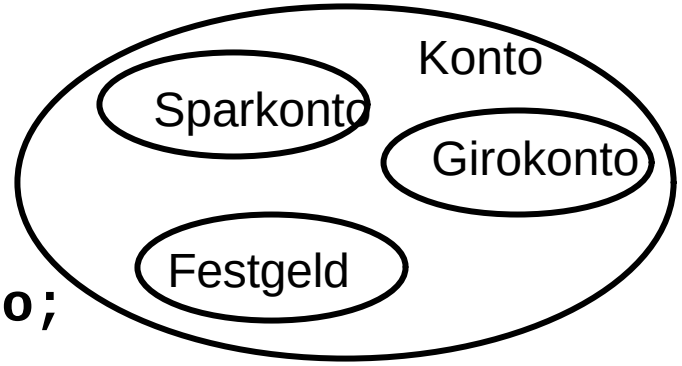
Welche Möglichkeiten entstehen durch diese Konstruktion ?

- Abstraktion & Spezialisierung:
  - Attribute & Methoden werden möglichst problemadäquat zugeordnet
  - allgemeine Lösungen sind von allgemeinen Nutzen
    - verstärkt Interesse an möglichst allgemeinen Lösungen von abstrakten Aufgabenstellungen

Darstellung aus mengentheoretischer Sicht

- Alle Objekte sind Konten!
- Einige sind besondere Arten von Konten.
- Die Menge der Sparkonten, Girokonen, Festgeldkonten ist jeweils eine Teilmenge der Menge der Konten
- Die Teilmengen sind disjunkt.





- Wir nehmen folgende Deklarationen an:
  - **`Girokonto einGirokonto;`**
  - **`Sparkonto einSparkonto;`**
  - **`Konto einKonto, einAnderesKonto;`**
- Legale Zuweisungen:
  - **`einGirokonto = new Girokonto();`**
  - **`einSparkonto = new Sparkonto();`**
  - **`einGirokonto.sollzinsen = 12;`**
  - **`einKonto = einGirokonto;`**
  - **`einAnderesKonto = new Sparkonto();`**
- Nicht legale Zuweisungen:
  - **`einSparkonto = einGirokonto;`**
  - **`einGirokonto = new Sparkonto();`**

Erläuterungen

Jedes Sparkonto / Girokonto ist auch ein Konto, deshalb ist **`einKonto = einSparkonto`** legal.

Ein Objekt der einer Klasse kann also mehrere Erscheinungsformen haben, es kann ein Objekt der Klasse selbst sein oder es kann ein Objekt einer der Unterklassen dieser Klasse sein. Es ist also **polymorph**.

Nicht jedes Konto ist ein Sparkonto.

Ist dann **`einSparkonto = einKonto`** legal?

Ja, denn Objekte der Klasse Sparkonto sind wandelbar zu Objekten der Klasse Konto. Allerdings ist der Zugriff auf alle Attribute nicht möglich, denn einKonto hat ja nicht die Sparkonto-Attribute.

Erläuterungen

Was passiert bei folgender Anweisung?

```
if (x%2 == 0)
    einKonto = einSparkonto;
else
    einKonto = einGirokonto;
```

Der Compiler ist nicht in der Lage, die Klasse von einKonto zu ermitteln.

Die Klasse von einKonto nach dieser Zuweisung ist nicht vorhersehbar.

einKonto kann also nach der Anweisung eine von mehreren Klassen haben, es ist halt polymorph.

Beispiel: Die Klasse Konto

```
public class Konto {
   private int kontostand = 0;
   private Waehrungsrechner wr;
   private Person inhaber;
   private int habenzinsen;

   public Konto (Person inhaber, Waehrungsrechner wr) {
       this.inhaber = inhaber;
       this.wr = wr;
   }
```

→

Beispiel (2): Die Klasse Konto

```
  public void zahle (int pfennige) {
     int euroCent = wr.wandeleInEuroCent(pfennige);
     kontostand += euroCent;
  }

  public int holeKontostand() {
     int pfennige = wr.wandeleInPfennige(kontostand);
     return (pfennige);
  }

} // Ende der Klasse Konto
```

Beispiel (3): Die Klasse Girokonto

```
public class Girokonto extends Konto {

   private int sollzinsen;
   private int dispo;

} // Ende der Klasse Girokonto
```

Schlüsselwort: extends

und natürlich weitere Methoden oder Überlagerung bestehender Methoden, z.B. zur Ergänzung um die Sollzinsenberechnung und die Kontrolle des Dispokredits.

Vererbung in JAVA, (technische Details)

- Vererbung wird über Schüsselwort `extends` realisiert:

```
class Unterklasse extends Oberklasse {
    ... // Hier zusätzliche Attribute und Methoden
}
```

- Die neu definierte Unterklasse erweitert also die anderswo definierte Oberklasse um neue Attribute und Methoden.
- Alle Methoden und Attribute der Oberklasse werden übernommen.
- Ist keine Oberklasse definiert (kein `extends`), so ist die Systemklasse `Object` die Oberklasse
  => `Object` ist eine Oberklasse für alle Klassen (bis auf `Object` selbst)
- Aus wievielen Oberklassen wird geerbt ?
  - Java: jede Klasse hat genau eine Oberklasse
  - C++: eine Klasse kann aus mehreren Oberklassen erben (Mehrfachvererbung)

Vererbung in JAVA, (technische Details)

- Konstruktoren werden nicht vererbt, Konstruktoren der abgeleiteten Klasse müssen neu definiert werden!
- Über Schlüsselwort `super` kann am Anfang eines Konstruktors der abgeleiteten Klasse ein Konstruktor der Oberklasse aufgerufen werden.

  Beispiel:

```
class A {
    A(String name) { ...
    }
}
class B extends A {
    B(String name, int a) {
      super(name);                  // Aufruf des Oberkl.-
      …                             //   Konstruktors
    }
}
```

Vererbung in JAVA, (technische Details)

- Wenn in der ersten Anweisung des Subklassen-Konstruktors nicht einer der Konstruktoren der Superklasse aufgerufen wird, dann wird der parameterlose Superklassen-Konstruktor (Standard-Konstruktor) automatisch aufgerufen, bevor irgendeine andere Anweisung des Subklassen-Konstruktors aufgerufen wird.

Weitere Fragestellungen:

- Wie lässt sich die Variationen von Attributen & Methoden innerhalb der Hierarchie kontrollieren ?

Nutzungsmöglichkeiten von Attributen & Methoden

aufgrund der Teilmengenbeziehung in der Vererbung sind Attribute & Methoden von Oberklassen noch sinnvoll nutzbar

Folgefragen:

- Wie lassen sich bestehende Methoden variieren / anpassen ?
  - Lässt sich diese Möglichkeit auch von der Oberklasse aus verhindern ?
- Zugriffsrechte: bisher:
  - private: nur innerhalb der Klasse
  - public: auch von ausserhalb der Klasse

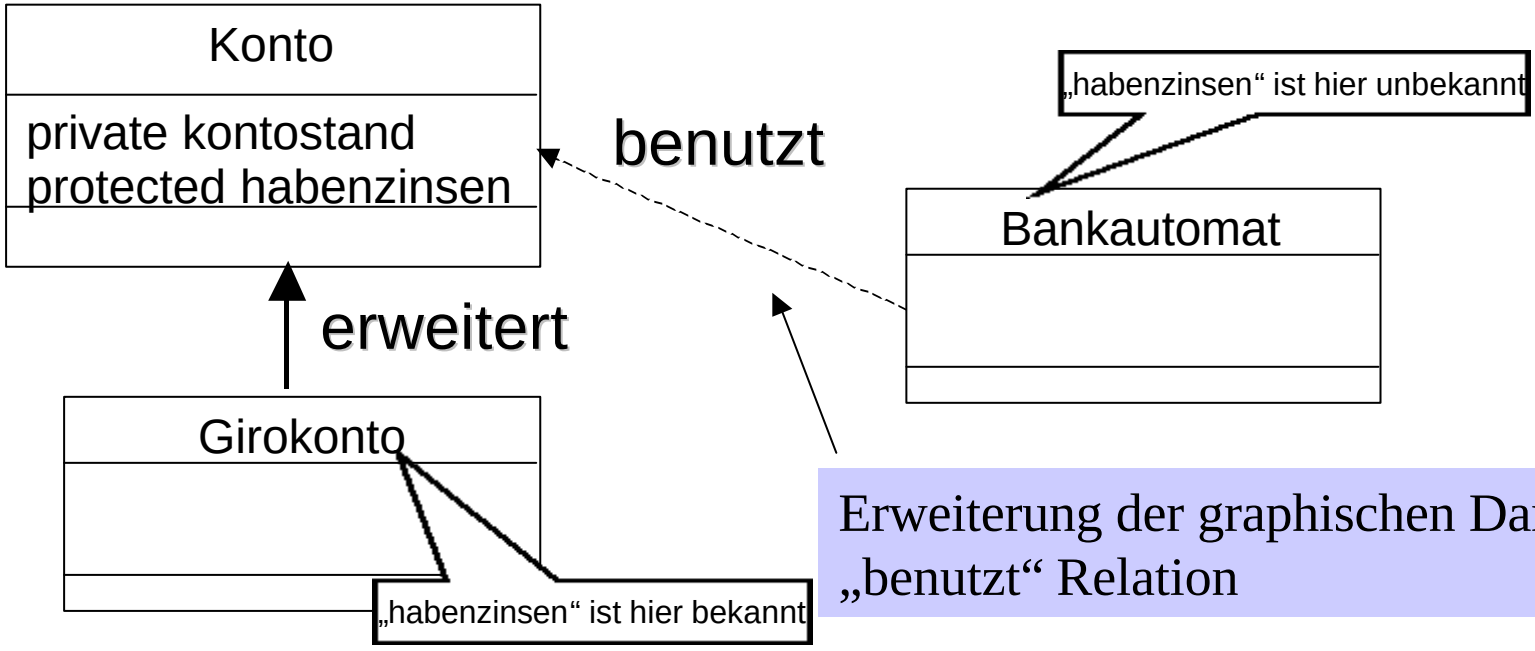
  Gibt es Regelungen auch für die Zugriffsrechte innerhalb der Vererbungshierarchie ?

Wir betrachten im folgenden die Regelung in Java,

Grundprobleme in allen Sprachen mit Vererbung gleich, aber Lösungen und Schlüsselworte können unterschiedlich sein...

Zugriffsrecht: protected (in Java)

- **`private`** Methoden und Attribute sind nur in der Klasse zugreifbar, in der sie definiert sind. Sie sind nicht in den erbenden Klassen zugreifbar.
- Oft ist es so, dass Methoden und Attribute nicht von außen zugreifbar sein sollen, dass sie aber schon vererbt werden sollen. Genau dies wird durch das Schlüsselwort **`protected`** vereinbart.
- **`protected`** Methoden und Attribute sind in der Klasse selbst und in allen Subklassen sichtbar und zugreifbar.



Erweiterung der graphischen Darstellung: „benutzt“ Relation

Überschreiben von Methoden in Vererbungshierarchien

Aufgabenstellung: Berechnung von Zinsen

- Methode: **`berechneZinsen (int tage)`**
- gleiche Implementierung in Sparkonto und Festgeld
- aber: in Girokonto Berechnung aus Sollzinsen und Habenzinsen
- Lösung unter Nutzung der Vererbungshierarchie:
  - Standard-Implementierung in Konto
  - **Überschreiben** der Methode in Girokonto für den Spezialfall

Überschreiben von Methoden

```
public class Konto {
   ...

   /**
   berechnet für die Anzahl Tage die angefallenen Zinsen
   */
   protected int berechneZinsen(int tage) {
     Float zinsen = new Float( kontostand*(
                          habenzinsen/100)*(tage/360));
     return (zinsen.intValue());
   }
   ...
}
```

Wrapper-Klasse: `Float`

Überschreiben von Methoden

```
public class Girokonto extends Konto {
   ...
   protected int berechneZinsen(int tage) {
       int guthaben = holeKontostand();
       if (guthaben > 0) {
          Float zinsen = new Float( guthaben *
                              (habenzinsen/100) *(tage/360));
          return (zinsen.intValue());
       } else {
          Float zinsen = new Float(guthaben*
                              (sollzinsen/100)*(tage/360));
          return (zinsen.intValue());
       }
   }
}
```

Zugriff auf überschriebene Attribute / Methoden

- In einem Objekt einer abgeleiteten Klasse ist **`super`** eine Referenz auf das Teilobjekt der Oberklasse
- Attribute und Methoden der Oberklasse lassen sich so ansprechen (auch überschriebene Attribute und Methoden)
- Beispiel:

```
class A {
int variable;
void methode() {
     ...
}
}
```

```
class B extends A {
   int variable;
   void methode() {
     ...
   }
   void methode2() {       // Zugriff auf
       super.variable = 3; // überschriebene
       super.methode();    // Attribute und
   }                       // Methoden der
                           // Oberklasse
}
```

Schlüsselwort: final

- Verhindert, dass eine Methode überschrieben wird

```
public final int holeKontostand() {...}
```

- Erben von einer Klasse verbieten

```
public final class Girokonto extends Konto {...}
```

  Alle Methoden und Attribute einer finalen Klasse sind implizit auch final.

- Finale Klassen und Methoden sind aus Sicherheitsgründen zuweilen erforderlich. Sie tun das, was sie tun sollen und können nicht manipuliert werden. Typische Anweisung: eine Methode zur Passwort-Prüfung.
- `final` - Attribute sind Konstanten, sie dürfen nicht verändert werden

```
public final int mwst;
```

Einschub: Klasse Datum (wird gleich benötigt)

```
public class Datum {

   public Datum(int tag, int monat, int jahr) { ...}

   public void setzeDatum(int tag, int monat, int jahr)  {
   ... }

   public static Datum aktuellesDatum()  { ... }

   public boolean istSpaeterAls(Datum dat) { ...}

   public boolean istAbgelaufen() { ...}

   public boolean istGleich (Datum dat)  { ...}

   public int gibDifferenz(Datum dat)  { ...}
}
```

Abstrakte Methoden/Klassen

- Situation: Jede Subklasse hat die gleiche Methode aber unterschiedliche Implementierung

- Beispiel: `auszahlen(int betrag)`
  - Girokonto: beliebige Auszahlung bis Limit
  - Sparkonto: Restguthaben von DM 5,- nötig (außer nach Kündigung)
  - Festgeld: Auszahlung erst nach Ende der Laufzeit

- Lösung: **abstrakte Methode** in der Superklasse. Eine abstrakte Methode ist eine Methode, die nicht realisiert ist.
- Die abstrakte Methode der Superklasse gibt dann nur die Signatur der Methode an, nicht aber ihre Realisierung.

Abstrakte Methoden/Klassen

Schlüsselwort: abstract vor Methode und Klasse

```
public abstract class Konto {

   protected Datum letzteTransaktion;
   ...

   public void einzahlen(int betrag) {
        Datum heute  = Datum.aktuellesDatum();
        int zinstage = heute.gibDifferenz(letzteTransaktion);
        int zinsen   = berechneZinsen(zinstage);
        zahle(betrag+zinsen);
   }

   public abstract int auszahlen(int betrag);

}
```

Beispiel: Klasse Girokonto

```
class Girokonto extends Konto {
   ...
   public int auszahlen(int betrag) {
        Datum heute = Datum.aktuellesDatum();
        int zinstage =
   heute.gibDifferenz(letzteTransaktion);
        int zinsen = berechneZinsen(zinstage);
        zahle(zinsen);
        if (kontostand-betrag > dispo) {
               zahle(-betrag);
               return (betrag);
        } else Bildschirm.gibAus("Kein Auszahlen möglich");
        return (0);
   }
}
```

Abstrakte Methoden/Klassen

- Enthält eine Klasse eine abstrakte Methode, so ist die ganze Klasse **abstract**

- Eine abstrakte Klasse kann nicht instanziiert werden. D.h., es können keine Objekte zu dieser Klasse erzeugt werden. Es kann nur Objekte zu den nicht abstrakten Unterklassen geben.

- Abstrakte Methoden **müssen** in den Subklassen implementiert werden (oder die Subklassen sind wieder abstract)

Polymorphie

- **Wunsch:**
  Alle Objekte aus der Oberklasse “Konto” sollen in der gleichen Weise behandelt werden können.

- **Lösung: Polymorphie**
  Eine Oberklassen-Referenz kann auch auf Objekte der Subklassen verweisen.

- Methoden der Oberklasse können so aufgerufen werden. Wurde eine Methode von einer Subklasse **überschrieben**, so wird nicht die Methodenimplementierung der Oberklasse aufgerufen, sondern die Implementierung der Subklasse.

Polymorphie

- Methoden können so mit allen möglichen Konten arbeiten

```
public int berechneVermoegen(Konto[] konten) {
    int vermoegen = 0;
    for (int i=0; i<konten.length; i++) {
          Konto k = konten[i];
          vermoegen += k.holeKontostand();
    }
    return (vermoegen);
}
```

- Methodenaufruf wird an die entsprechende Subklasse weitergeleitet

Polymorphie (technisch)

- Polymorphie wird bei Vererbung durch Überschreiben der Methoden der Oberklasse erreicht, dabei muß die Signatur (also Parameterliste und Rückgabetyp) mit der Methode der Oberklasse übereinstimmen.
- Beim Überschreiben werden die allgemeineren Methoden (der Oberklasse) durch die konkreteren der Unterklasse überschrieben.
- Auch wenn ein Objekt durch eine Variable eines allgemeineren Typs referenziert wird, so werden immer die zum Objekt gehörenden Methoden aufgerufen.
- Überschreiben darf nicht mit Überladen verwechselt werden, bei überladenen Methoden hat man unterschiedliche Signaturen und nur der Methodenname ist der gleiche.

Instanceof

- Da jedes Objekt auch über Referenzen der Oberklasse angesprochen werden kann, ist nicht immer klar, zu welcher Klasse ein Objekt gehört. Daher gibt es das Schlüsselwort `instanceof`, um die Klassenzugehörigkeit zu bestimmen:

  `if(`*Objektname* `instanceof` *Klassenname*`)`
  Anweisung

- Die Abfrage liefert auch dann `true`, wenn durch *Klassenname* eine Oberklasse für das zu *Objektname* zugehörige Objekt angegeben wird

Beispielanwendung: Vererbung

- Motivation
- **Vererben von Attributen und Methoden**
- **Überschreiben von Methoden**
- Aufgabenstellung
- **Die Gemeinsamkeiten von Girokonto, Sparkonto und Festgeldkonto in der gemeinsamen Oberklasse Konto sollen für eine die Implementierung der. Klasse Festgeldkonto als Erweiterung der Klasse Konto genutzt werden.**
  - **neue Attribute: laufzeit und mindBetrag**
  - **neue Methoden:**

    erweiterter Konstruktor : Übergeben der Laufzeit

    `auszahlen(int betrag)` : Auszahlung der Summe nach Ende der Laufzeit

    `kuendigen()` : identisch mit auszahlen
- Entwicklung einer Testklasse
  - **benutzt eine Methode setzeDatum der Klasse Datum, um die letzteTransaktion zurückzusetzen**

Beispiel: Vererbung

```
public class Festgeld extends Konto {
    private int laufzeit = 1080;          // Laufzeit: 3
  Jahre
    public static int mindBetrag = 250000;

public Festgeld (Person inhaber, Waehrungsrechner wr,
     int initialkontostand) {
      super(inhaber, wr, initialkontostand);
      if (initialkontostand < mindBetrag) {
          Bildschirm.gibAus("Mindest-Betrag wird nicht
                                    erreicht");
      }
  }
```

Beispiel: Vererbung

```
public Festgeld (Person inhaber, Waehrungsrechner wr) {
      super(inhaber, wr);
  }

public Festgeld (Person inhaber, Waehrungsrechner wr,
            int initialkontostand, int laufzeit) {
  super(inhaber, wr, initialkontostand);
  this.laufzeit = laufzeit;
  if (initialkontostand < mindBetrag) {
      Bildschirm.gibAus("Mindest-Betrag wird nicht
                         erreicht");
      }
  }
```

Beispiel: Vererbung

```
public void setzeLaufzeit (int neueLaufzeit) {
       this.laufzeit = neueLaufzeit;
   }

public int kuendigen () {
       return (auszahlen(holeKontostand()));
   }
```

```
public int auszahlen (int betrag) {
        if (betrag == holeKontostand()) {
            Datum heute = Datum.aktuellesDatum();
            int abgelaufeneZeit =
                    heute.gibDifferenz(letzteTransaktion);
            if (abgelaufeneZeit >= this.laufzeit) {
                int zinsen =
                    holeWaehrungsrechner().wandeleInPfennig
                            (berechneZinsen(laufzeit));
                int pfennige = holeKontostand()+zinsen;
                zahle(-betrag);
                return (pfennige);
            } else {
                Bildschirm.gibAus("Laufzeit noch nicht
  errreicht");
            }
        } else {
            Bildschirm.gibAus(“Auszahlung der ges. Summe
  nötig");
        }
     }
```

Zusammenfassung

- Vererbung:
  - Klassen können als Unterklasse von einer Klasse definiert werden
  - Java: Vererbungshierarchie mit 1 Superklasse je Klasse
  - C++: Vererbungshierarchie mit mehreren Superklassen für 1 Klasse
- Folgen:
  - Behandlung namens/signaturgleicher Methoden in Super/Subklassen, Zugriffsmöglichkeiten auf verdeckte Attribute & Methoden
  - Erweiterung der Definition von Zugriffsrechten: (private, public, protected)
  - Behandlung von abstrakten („noch zu implementierenden“) Methoden
  - Begrenzung der Möglichkeit des Überschreibens: final
- Nutzen:
  - erlaubt allgemeine Lösungen in Spezialfällen ohne redundanten Code zu nutzen
  - erlaubt Anforderungen zu spezifizieren: abstrakte Klassen
  - erlaubt Abwandlung von Methoden: Überschreiben (bei gleicher Signatur)
    - Achtung: nicht mit Überladen verwechseln (ungleiche Signatur)