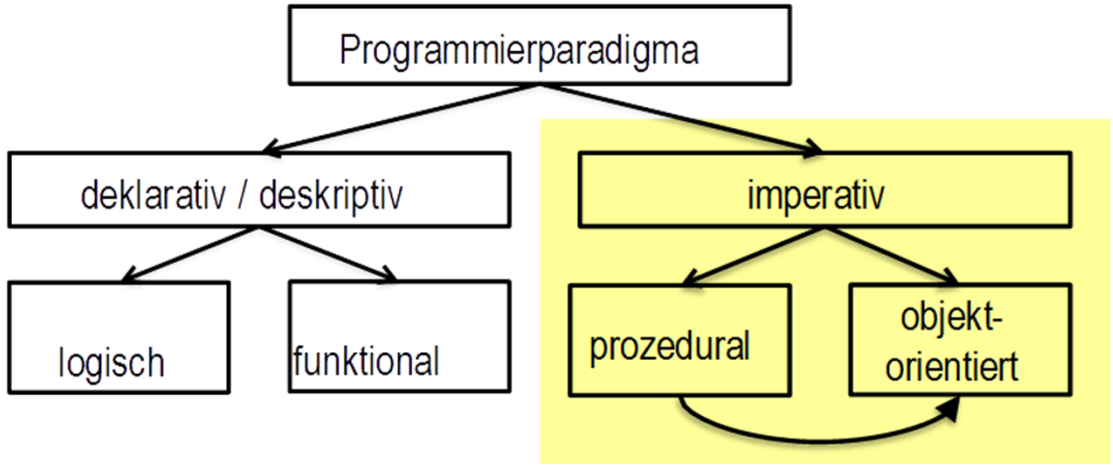
Programmierung 2

Kapitel 4 – Objektorientierung

Vielen Dank an Prof. Erik Buchmann
für die beigesteuerten Folien

HfTL

Programmierparadigmen



- Prozedural
  - Prozeduren (Methoden) zur Aufteilung in Teilaufgaben
  - Programmiersprachen: C, Basic, ...
- Objekt-orientiert
  - Einheit von Daten und Funktion (Methoden)
  - Programmiersprachen: Java, C++, …
    (praktisch alle OO-Sprachen können auch prozedural)

Objektorientierung vs. Prozedural

- Prozedurale Programmierung
  - Daten und Funktionen werden getrennt betrachtet
  - Entscheidend ist die Reihenfolge der Anweisungen bzw. Methodenaufrufe

- Objektorientierte Programmierung
  - Klassen kapseln **Daten und Funktionen als Einheit**
  - Entscheidend ist **Kommunikation der Objekte untereinander**

Objektorientierte Softwareentiwcklung

- Analyse
  - Modell der Problemstellung
- Entwurf
  - Modell der Software
- Implementierung
  - Programmierung
  - Testen
- ...

OO Analyse

- Aufgabe:
  - Untersuchung des Problem bzw. Anwendungsbereiches
- Ziel:
  - Erstellung eines OO-Modells
    - **statischer Aufbau** des Anwendungsgebietes
    - **dynamische Abläufe** im Anwendungsgebiet
- Aktivitäten:
  - Identifikation von **Objekten** und **Klassen**
  - Identifikation von **Beziehungen** zwischen Objekten
  - Identifikation von **Attributen** und **Methoden**
  - Identifikation von (Arbeits-)**Abläufen**

OO Entwurf

- Aufgabe:
  - Abbildung des Modells des Anwendungsgebietes (OOA-Modells) auf ein Modell des zu entwickelnden Systems (OOD-Modell)
- Ziel:
  - Ein implementierbares Modell
- Aktivitäten, u.a.:
  - Ergänzung des OOA-Modells um weitere Klassen, Objekte, Attribute, Methoden, die im Problembereich nicht auftreten, bei der Implementierung aber unerlässlich sind (Speicherung, Darstellung, ...)

OO Implementierung

- Aufgabe:
  - Umsetzung des OOD-Modells in eine konkrete Programmiersprache (z.B. Java)
- Ziel:
  - Erstellung eines Programmsystems
- Aktivitäten:
  - Implementierung der Klassen des OOD-Modells
  - Aufbau von Klassenbibliotheken
- Methoden:
  - Überführung des OOD-Modells in ein Programm
  - Nutzung von Klassenbibliotheken

Anwendungsbeispiel: Bank Terminal

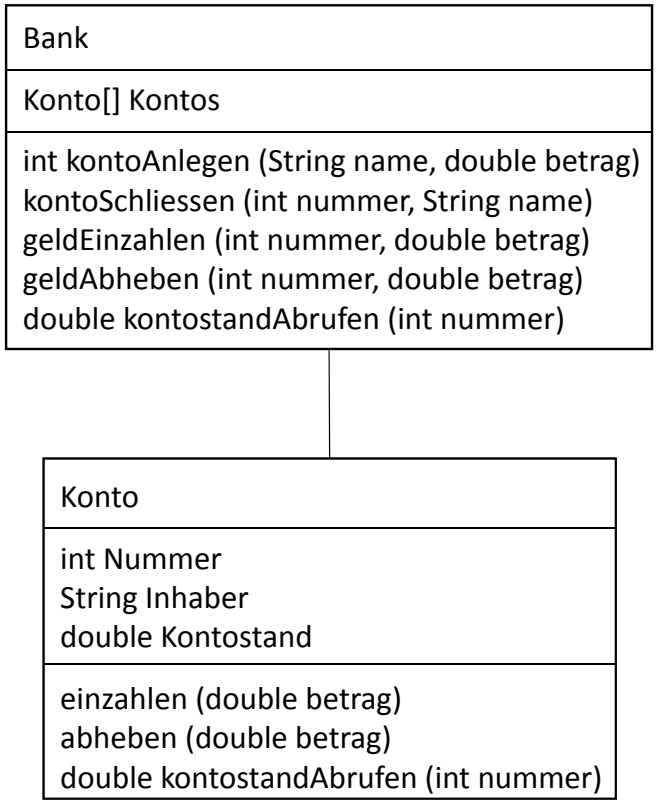
- Ein Kund wünscht eine einfache Terminal-Anwendung, die es Mitarbeitern der Bank erlaubt, Konten für Kunden zu eröffnen, zu schließen, den Kontostand abzufragen und Ein- und Auszahlungen zu verbuchen
- Kontonummern sind eindeutig zu vergeben. Sie können nach der Vergabe nicht mehr verändert werden.
- Beim Anlegen eines neuen Kontos wird eine eindeutige Kontonummer automatisch vergeben. Darüber hinaus wird der Name des Kontoinhabers sowie ein etwaiger Startbetrag eingegeben.
- Auf Wunsch des Kunden, kann der Kontostand eines Kontos abgefragt werden. Hierzu ist die Eingabe der Kontonummer notwendig
- Für Ein- und Auszahlungen muss sichergestellt werden, dass Geldbewegungen auf dem richtigen Konto vorgenommen werden.

Beispiel OO Analyse

- Identifikation von Objekten/Klassen
  - Bank, Konto, Mitarbeiter
- Identifikation von Beziehungen zwischen Objekten
  - Eine Bank verwaltet mehrere Konten
- Identifikation von Eigenschaften und Methoden von Objekten
  - Eigenschaften von Bank
    - Menge an Konten, ...
  - Methoden von Bank
    - Konto eröffnen, Konto schließen, Kontostand abfragen, ...
  - Eigenschaften von Konto
    - Eindeutige Kontonummer
    - Name des Kontoinhabers
    - Geldbetrag (Kontostand)
  - Methoden
    - Kontostand abfragen, Geld einzahlen, Geld abheben

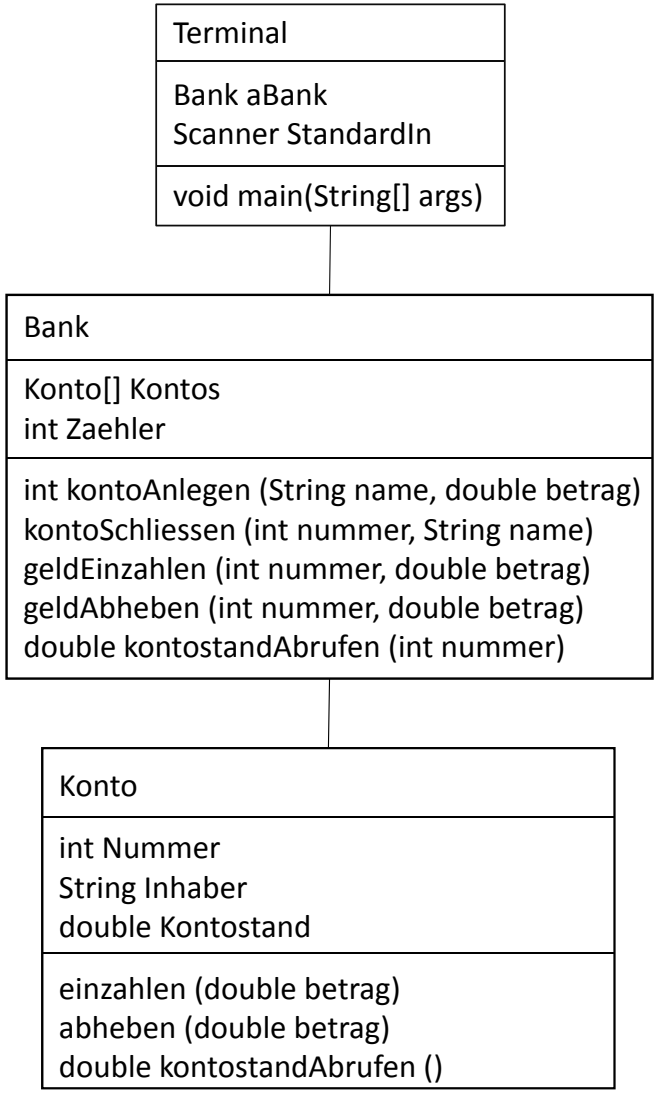
Beispiel OO Analyse

- Erstellung eines ersten OOA Modells (z.B. mit UML)



Beispiel OO Entwurf

- Verfeinerung und Ergänzung des Modells um weitere Klassen, Objekte, Attribute, Methoden
  - Beispiele
    - Weitere Klasse "Terminal" um Mitarbeitern die Interaktion mit der Anwendung zu ermöglichen
    - Hauptmethode "main" um die Ausführung der Anwendung zu ermöglichen
    - Weiteres Attribute "Zaehler" in Klasse Bank für die Generierung eindeutiger Kontonummern
  - Andere mögliche Erweiterungen
    - Datenbankanbindung für die persistente Speicherung von Kontoinformationen
    - Methoden für Mitarbeiterlogin- und Authentifizierung
    - …



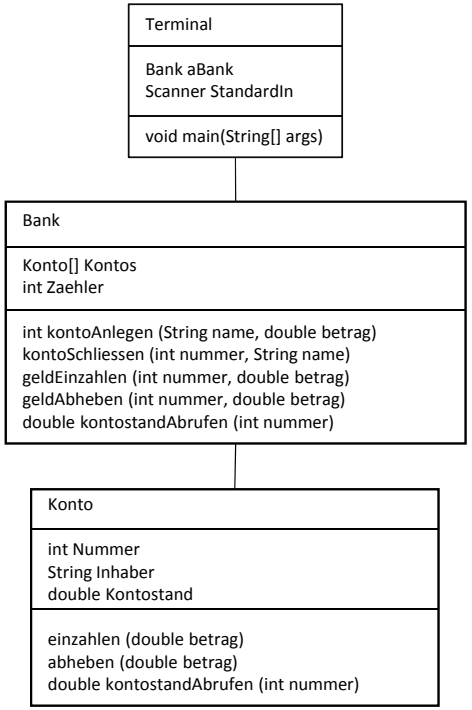
Beispiel OO Implementierung

- Siehe Quellcode im ILIAS

OO: Von der Problemstellung zum Programm

OO Analyse *OO Entwurf* *OO Implementierung*

Problemstellung → Modell (z.B. UML) → Java Programmcode




```
public class Bank {
   Konto[] kontos;

   publi
      kor
   }
}
```

```
public class Konto {
   int nummer;
   String inhaber;
   double kontostand;

   public void einzahlen(double betrag) {
      kontostand += betrag;
   }
}
```

Zur Laufzeit:

```
Konto konto1 = new Konto(1, Eva, 20.00);
Konto konto2 = new Konto(2, John, 5.00);

Konto konto3 = new Konto(100, Steve, 0.0);
```

Klassen und Objekte

- Eine **Klasse** beschreibt
  - Eigenschaften (Attribute, Properties) und
  - Verhalten (Methoden)
- einer Gruppe gleichartiger Objekte (→ Referenzdatentyp!)

- **Objekte** (Instanzen) werden durch Klassen beschrieben
  - Datenelemente (Attribute) → Eigenschaften, Struktur, Zustand
  - ausführbare Operationen (Methoden) → Verhalten

- Alle Objekte einer Klasse
  - haben gleiche Attribute und gleiche Methoden
  - unterscheiden sich "nur" in den Werten ihrer Attribute
  - Klasse ist "Bauplan" bzw. "Schablone"

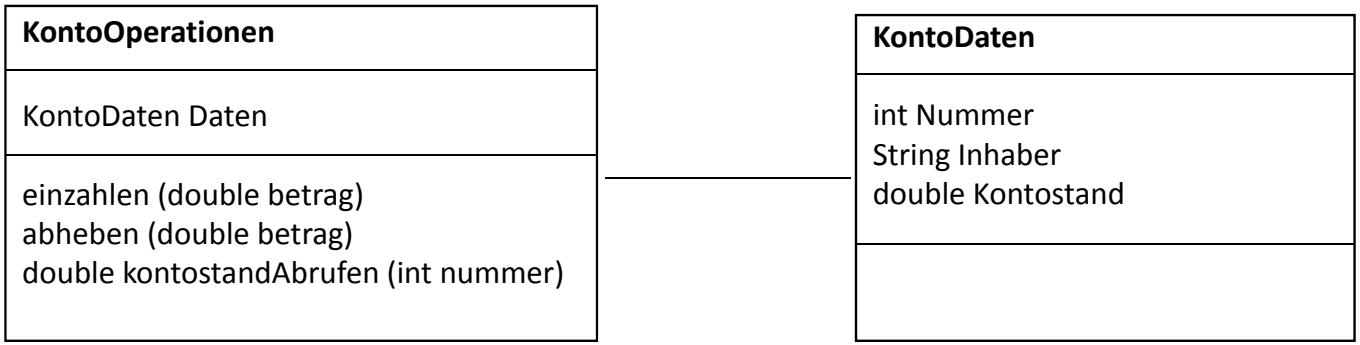
Beispiel Klassen und Objekte

- Klasse Konto
  - Attribute: Inhaber, Nummer, Kontostand
  - Methoden: einzahlen(), abheben(), überweisen()
- Objekte bzw. Instanzen der Klasse Konto
  - „Paul“, DE77...1, 100 EUR
  - „Fred“, DE77...2, 500 EUR
  - „Lisa“, DE77...3, 1000 EUR
  - ...

Gute und Schlechte Objektorientierung

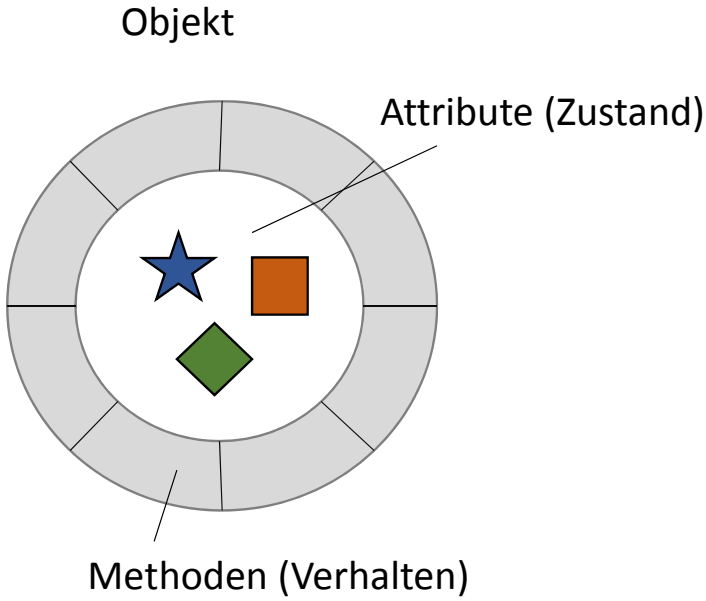
- Modellierung und Entwurf von Systemen interagierender Objekte statt Programmabläufe mit passiven Daten
  - handelnde Objekte mit Eigenschaften, veränderlichem Zustand und eigenen Operationen darauf statt Funktionen auf passiven Daten
- Objekt bestimmt, wie es einen Methodenaufruf ausführt:
  - Aufrufer ruft Methode in Empfänger-Objekt auf
  - Empfänger führt Methode auf eigenem Objektzustand aus
- **Eigenverantwortlichkeit**, Selbständigkeit der Objekte!
- Schnittstellen: von außen sichtbare Methoden des Objektes
  - Schnittstelle beschreibt das **Was** und verbirgt das **Wie**

Beispiel Schlechte Objektorientierung



Kapselung

- Software-Objekte verhalten sich ähnlich Objekten der realen Welt
- Kapselung bedeutet das der interne Zustand eines Objekts nicht direkt von außen manipuliert werden kann, sondern nur durch dafür bereitgestellte Methoden
- Objekt behält volle Kontrolle, wie sein Zustand verändert werden kann



Kapselung

- Objekt von außen durch Operationen ansprechbar
  - interner Zustand über Methoden der Klasse änderbar, **nach den in der Klasse implementierten Regeln**
- Beispiel: Klasse Konto
  - Kontonummer
    - kann beliebig oft gelesen, aber nur einmal geschrieben werden
    - muss vor der ersten Ein- oder Auszahlung festgelegt werden
  - Kontostand
    - kann nur durch Ein- und Auszahlungen geändert werden
    - keine Auszahlung falls Auszahlungsbetrag größer als Kontobetrag
    - kann beliebig oft gelesen werden
  - Kontoinhaber
    - (...)

Warum ist Kapselung wichtig?

- Kontostand, **prozedural**
  - **Geldautomat** implementiert abbuchen
    - lese Kontostand
    - wenn Kontostand > Betrag
      - zahle aus
      - verringere Kontostand um Betrag
  - **Bank** implementiert überweisen
    - lese Kontostand von Senderkonto
    - wenn Kontostand > Betrag und Empfängerkonto existiert
      - verringere Kontostand um Betrag
      - lese Kontostand von Empfängerkonto
      - erhöhe Kontostand auf Empfängerkonto um Betrag

Warum ist Kapselung wichtig?

- Kontostand, **objektorientiert**
  - **Konto** implementiert
    - auszahlen
    - einzahlen
  - **Geldautomat** implementiert Funktion abheben
    - wenn Konto „auszahlen“ erfolgreich
    - dann gib Bargeld aus
  - **Bank** implementiert Funktion überweisen
    - wenn Empfängerkonto existiert und „auszahlen“ erfolgreich
    - dann führe „einzahlen“ auf Empfängerkonto-Objekt aus

Wo werden welche Aufgaben implementiert und ausgeführt?
Wer kann welche Informationen sehen und manipulieren?

Vorteile Kapselung

- Modularität
  - Quellcode für verschiedene Klassen/Objekte kann unabhängig geschrieben, getestet und erweitert werden
- Information-hiding
  - Details der internen Implementierung einer Klasse/Objekts sind von der Außenwelt nicht einsehbar
- Wiederverwendung
  - Code bereits programmierter Klassen/Objekte können in mehreren Programmen verwendet werden
  - Ermöglicht Spezialisten das Implementieren und Testen von vertrauenswürdigen Klassen/Objekten
- Austauschbarkeit
  - Problematische Klassen/Objekte können aus einer Anwendung entfernt und durch andere Klassen/Objekte ersetzt werden

Kapselung: Set- und Get-Methoden

- Code-Konvention: Kapselung aller Attribute, die nach außen sichtbar/änderbar sein sollen, über get- und set-Methoden
  - Sie müssen das nicht tun, SOLLTEN es aber
  - üblich get + Name des Attributs mit großem Anfangsbuchstaben

```
private String name;    // Name des Kontoinhabers

public String getName() {
    return name;
}

public void setName( String name ) {
    // weitere Anweisungen möglich
    this.name = name;
}

public int getNummer() {
    return nummer;
}
// keine Set Methode für nummer (Kontonummer)!
```

Kapselung: Zugriffskontrolle

	Klasse	Package	Unterklasse (gleiches Package)	Unterklasse (anderes Package)	Welt
Public	X	X	X	X	X
Protected	X	X	X	X	
Default *(ohne Schlüsselwort)*	X	X	X		
Private	X				

Pakete und Zugriffsrechte

Motivation

- Problem
  - Programme bestehen aus vielen Klassen (oftmals hunderte, tausende)
  - Programme bestehen aus unterschiedlichen logische Einheiten, z.B. Drei-Schichten-Architektur
  - Kontrollierter Zugriff auf Klassen von „außerhalb“ (anderen Programmen)
- Hilfsmittel gesucht:
  - Übersichtliche Strukturierung und Gruppierung von Klassen und Programmteilen
  - Klare Trennung von internen Klassen und Klassen die auch anderen Programmen zur Verfügung stehen
- Lösung
  - Pakete und Zugriffsrechte

Definition Pakete

- Schlüsselwort: **package**
- Package Anweisung:
  package <Paketname>;
- Ist immer die erste Anweisung in einer Klassendatei

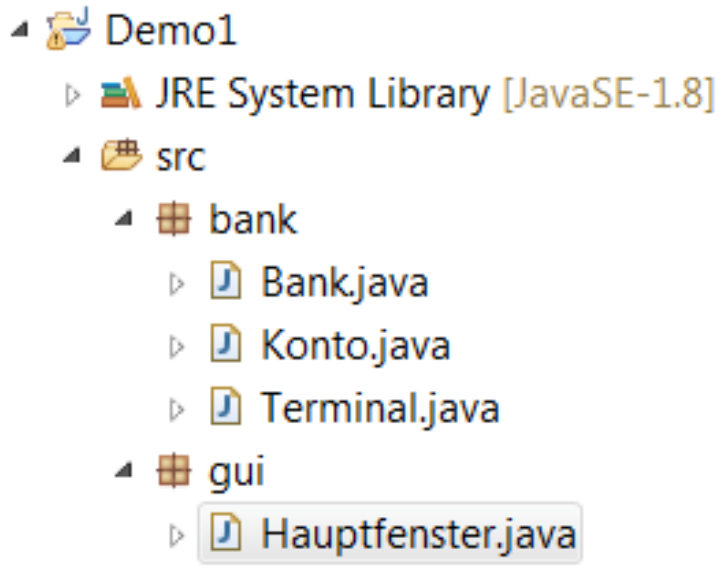
```
package bank;

public class Konto {
    private double kontostand;
    private int nummer;
    private String name;

    public Konto( int nummer, String name, double betrag ) {
        this.nummer = nummer;
        this.name = name;
        kontostand = betrag;
    }
    …
```

Definition Pakete

- Der Paketname ist ein Java Bezeichner
- Alle Klassen eines Pakets befinden sich im gleichen Unterverzeichnis im Quellcode
- Pro Verzeichnis ist genau ein Paket definiert
- Verzeichnisname = Paketname
- Pakete können weiter strukturiert werden (Pakethierarchie, Punkt-Notation)



Nutzung von Paketen

- Schlüsselwort **import**
- Import-Anweisung
  - erlaubt das Einbinden von Klassen aus anderen Paketen
  - folgt direkt auf die Paket-Anweisung in einer Klassendatei
- Mehrere **import** Anweisungen pro Klassendatei erlaubt

```
package gui;

import bank.Konto;

public class Hauptfenster {

    public Hauptfenster()
    {
        Konto konto = new Konto(0, "Eva", 0.0);
        …
}
```

Nutzung von Paketen: Importieren

- Kein expliziter Import

```
java.util.Date date = new java.util.Date();
java.util.Vector vector = new java.util.Vector();
```

- Import einzelner Klassen (empfohlen)

```
import java.util.Date;
Date date = new Date();
Vector vector = new Vector(); // Fehler! Klasse unbekannt
```

- Import aller Klassen eines Pakets (ohne Unterpakete)

```
import java.util.*;
Date date = new Date();
Vector vector = new Vector();
```

- Import eines Pakets

```
//Fehler, nur Klassen können importiert werden
import java.util;
```

Nutzung von Paketen: Namenskonflikte

- Problem
  - Verwendung gleichnamiger Klassen aus unterschiedlichen Paketen
- Lösung
  - Unterscheidung der Klassen im Programmcode durch Angabe des vollständigen Paketpfads
  - Eindeutige Paketnamen

    com.hftl.banking.bank

- Beispiel
  - package java.util enthält Klasse Vector
  - package misc enthält Klasse Vector

```
import java.util.*;
import misc.*;
Vector vector = new Vector();//Fehler, welcher Vector?
```

```
import java.util.Vector;
import misc.Vector;//Fehler, Namenskollision
```

```
java.util.Vector vector1 = new java.util.Vector();
misc.Vector vector2 = new misc.Vector();
```

Nutzung von Paketen

- JDK-Pakete: java.lang
  - Z.B. Klassen System, String, Object
  - Import Anweisungen nicht notwendig (implizites Import)

- Default package/unbenanntes Paket
  - Für sehr kleine Programme kann es ausreichen alle Klassen in einem Paket unterzubringen
  - Speicherung der Klassen im Quellcode-Wurzelverzeichnis
  - Keine Vergabe eines Paketnamen
  - Keine package-Anweisung in Klassendateien

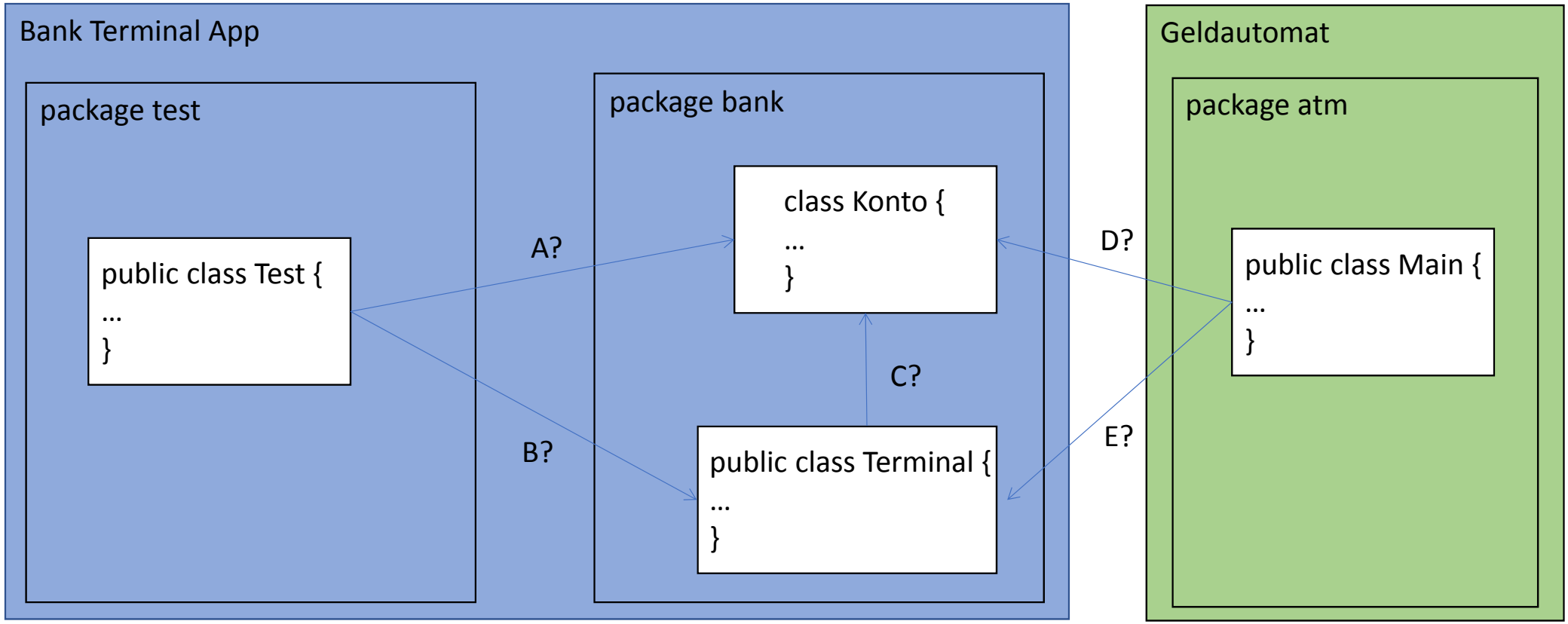
- Zugriff auf Klassen und deren Elemente eines unbenannten Pakets ist auf Klassen innerhalb des Verzeichnisses beschränkt
  - Kein Zugriff von Klassen aus anderen Paketen!

Zugriffsrechte: Klassen

- Möglichkeiten Zugriffsrechte auf Klassen zu definieren
  1. Schlüsselwort **public**:
     - Zugriff von allen Paketen in diesem und anderen Programmen
  2. Default, auch package-private genannt (*ohne Schlüsselwort*)
     - Zugriff von allen Klassen innerhalb desselben Pakets
- In einer Datei können mehrere Klassen definiert sein
  - Nur eine Klasse darf **public** sein
  - Name der public Klasse = Name der Datei
- Richtlinien:
  - Will man öffentlich zugängliche Klassen definieren, muss man sie als **public** deklarieren
  - Hilfsklassen sollten als *default* deklariert werden

Zugriffsrechte: Klassen

Welche Zugriffe sind möglich?

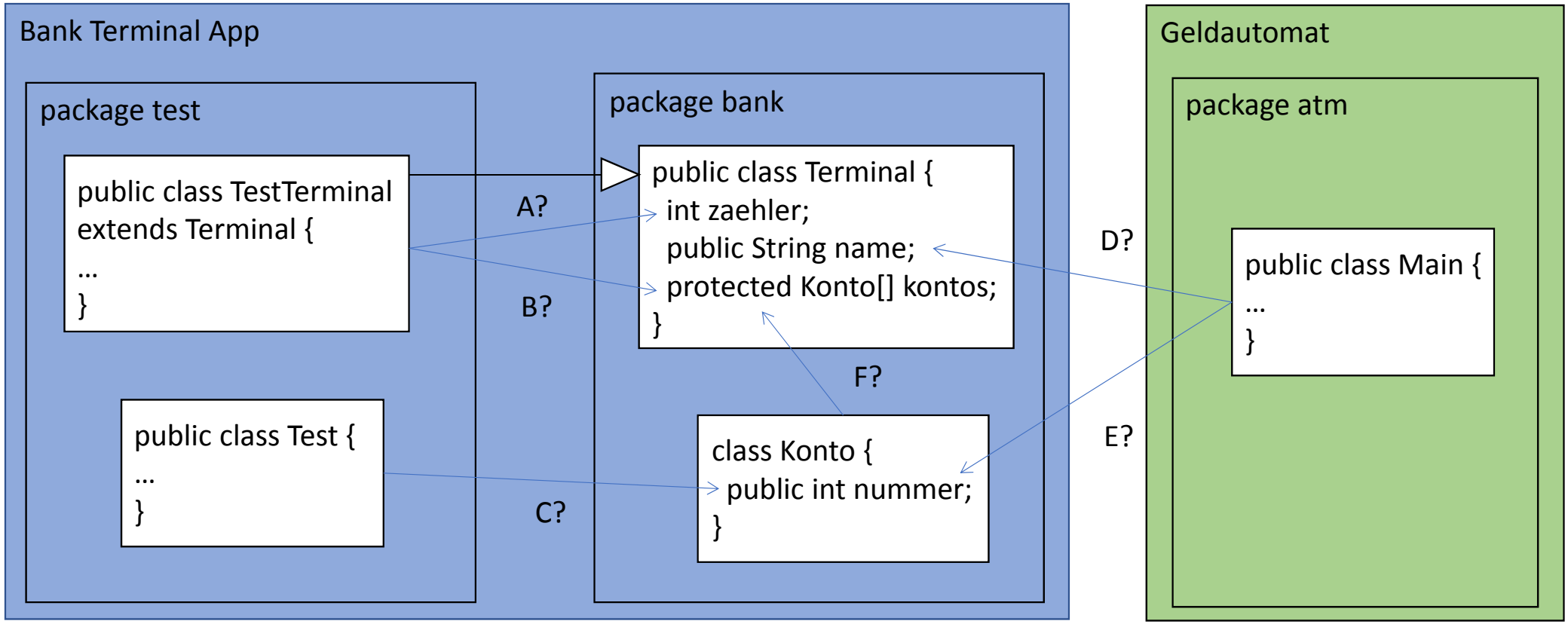


Zugriffsrechte: Attribute und Methoden

- Möglichkeiten Zugriffsrechte auf Elemente einer Klasse zu definieren
  1. Schlüsselwort **public**
     - Zugriff von allen Paketen in diesem und anderen Programmen
  2. Schlüsselwort **protected**
     - Zugriff von allen Klassen innerhalb desselben Pakets und von Unterklassen in anderen Paketen
  3. Default, auch package-private genannt (*ohne Schlüsselwort*)
     - Zugriff von allen Klassen innerhalb desselben Pakets
  4. Schlüsselwort **private**
     - Zugriff nur innerhalb derselben Klasse

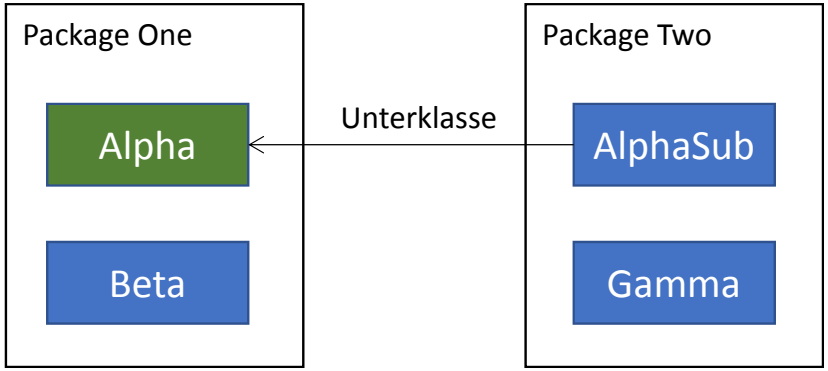
Zugriffsrechte: Attribute und Methoden

Welche Zugriffe sind möglich?



Zugriffsrechte: Attribute und Methoden

	Klasse	Package	Unterklasse (gleiches Package)	Unterklasse (anderes Package)	Welt
Public	X	X	X	X	X
Protected	X	X	X	X	
Default *(ohne Schlüsselwort)*	X	X	X		
Private	X				



Sichtbarkeit von Attributen/Methoden der Alpha Klasse

	Alpha	Beta	AlphaSub	Gamma
Public				
Protected				
Default				
Private				

Siehe https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Zugriffsrechte: Verwendung

- Attribute einer Klasse
  - Wenn immer möglich, als **private** deklarieren (kein direkter externer Zugriff)
  - Falls Zugriff in Unterklassen notwendig ist, als **protected** deklarieren
- Konstanten einer Klasse
  - Normalerweise als **public** deklariert
- Methoden einer Klasse
  - Setter/Getter-Methoden normalerweise als **public** deklariert
  - Zugriffsrechte für andere Methoden je nach Notwendigkeit vergeben

Instanzen von Klassen

```java
package hftl;
public class Konto {
    int iban;
    String inhaber;
    double betrag;

    /**
     * Konstruktor
     * @param nr Iban-Nummer
     * @param name Kontoinhaber
     */
    public Konto(int nr, String name) {
        inhaber = name;
        iban = nr;
        betrag = 0;
    }

    /**
     * Geld auf das Konto einzahlen
     * @param b Einzahlungsbetrag
     */
    public void einzahlen(double b) {
        // Gültigkeit der Parameter testen
        if (b < 0) return;
        betrag += b;
    }
    public void auszahlen(double b) {}
}
```

Drei Instanz-Attribute: Jede Instanz der Klasse speichert diese Variablen.

Konstruktor zur Initialisierung des Objekts

public: Ist von „außen“ sicht- und zugreifbar, → Signatur

Instanz-Methoden

Instanzen von Klassen

- Referenzdatentyp
- Instanziierung mit Schlüsselwort *new*

```
package hftl;
public class MyClass {

    public static void main(String[] args) {

        Konto klaus = new Konto(1, "Klaus");
        Konto heinz = new Konto(2, "Heinz");
        Konto vielekonten[] = new Konto[100];

        heinz.einzahlen(5000.00);

        heinz.auszahlen(100.00);
        klaus.einzahlen(100.00);
    }
}
```

Deklaration ebenso wie bei Referenzdatentypen

Übergabe von Werten an den Konstruktor

Aufruf von Methoden der Instanzen

Konstruktor

- **Konstruktor**: Methode, die aufgerufen wird wenn mittels new eine neue Instanz angelegt wird
- Eine Klasse kann mehrere Konstruktoren haben
- Konstruktor mit passender Signatur muss existieren, sonst schlägt new fehl

```
// ok
… new Konto(3, "Erika");

// Fehler, Konstruktor
// mit der Signatur ()
// existiert nicht
… new Konto();
```

```
/**
 * Konstruktor 1
 * @param nr Iban-Nummer
 * @param name Kontoinhaber
 */
public Konto(int nr, String name) {
    inhaber = name;
    iban = nr;
    betrag = 0;
}

/**
 * Konstruktor 2
 * @param name Kontoinhaber
 * @param b Anfangsguthaben
 */
public Konto(String name, double b) {
    inhaber = name;
    // Hole Zufallswert
    Random r = new Random();
    iban = r.nextInt();
    betrag = b;
}
```

Default-Konstruktor

- Default Konstruktur hat keine Parameter

- Eine Klasse muss keinen Konstruktor definieren
  - Speziell ist es nicht notwendig einen leeren Konstruktur zu schreiben

```
class Konto {
   public Konto() {
      // keine Anweisungen
   }
   …

}
```

- Compiler benutzt automatisch den Default-Konstruktor (der Superklasse)
  - In Vererbungshierarchie sind zusätzliche Anforderungen an die Verwendung eines Default-Konstruktors geknüpft (… spätere Vorlesung)

Copy-Konstruktor

- Bequemes erzeugen einer Objektkopie
  - Konstruktor erhält ein Objekt vom gleichen Typ als Parameter und kopiert die Attribute der übergebenen Instanz in die eigenen Attribute

```
public class Konto {
    int iban;
    String inhaber;
    double betrag;

    public Konto(Konto k) {
        inhaber = k.inhaber;
        iban = k.iban;
        betrag = k.betrag;
    }
(...)
```

- *Vorsicht mit Attributen die selber Referenzdatentypen sind!*

Zugriff auf Attribute mit dem Schlüsselwort this

- Nötig, um sich innerhalb einer Methode auf die Instanz des Objekts zu beziehen, in der man sich gerade befindet

```
public class Konto {
    int iban;
    String inhaber;
    double betrag;

    public Konto(int iban, String inhaber) {
        this.inhaber = inhaber;
        this.iban = iban;
        betrag = 0;
    }
(...)
```

Kann / darf / muss vor betrag auch ein this stehen?

Problem: Übergebene Variablen tragen den gleichen Namen wie Instanz

Unterscheidung mit `this`

`this.inhaber` := Attribut der Instanz

`inhaber` := Übergabeparameter

Konstruktor-Aufruf mit dem Schlüsselwort this

```
public Konto( int nummer, String name, double betrag ) {
    this.nummer = nummer;
    this.name = name;
    kontostand = betrag;
}

public Konto( int nummer, String name )
{
    this(nummer, name, 0.0);

}
```

- Zwei oder mehrere Konstruktoren einer Klasse können denselben Namen haben, wenn sie unterschiedliche Parameter besitzen
- Wiederverwendung eines Konstruktors, z.B. zum setzen von Default-Werten von Attributen
- Konstruktor-Aufruf innerhalb eines Konstruktors muss immer die erste Anweisung sein

Code Review

```
public  class Konto {
   private double kontostand;
   private String name;
   private int nummer;

public Konto( int nummer, String name, double betrag ) {
   this.nummer = nummer;
   name = name;
   kontostand = nummer;
}
public Konto( int nummer, String name, double betrag ) {
   this.nummer = nummer;
   this.name = name;
   return 10;
}
```

```
public class Bank {
   private int zaehler = -1;
   private Konto[] kontos = new Konto[100];

   public Bank( Bank bank ) {
      kontos = bank.kontos;
      zaehler = bank.zaehler;
   }
   public int kontoAnlegen( String name, double betrag ) {
      zaehler++;
      Konto konto = new Konto(zaehler, name);
      kontos[zaehler] = konto;
      return zaehler;
   }
}
```

Destruktoren?

- Warum definieren Java Klassen keine Destruktoren?

Destruktoren?

- Warum definieren Java Klassen keine Destruktoren?

➡ Automatische Speicherverwaltung in der JVM!

Methoden und Attribute der Instanz
Methoden und Attribute der Klasse

Instanz-Attribute vs. Klassen-Attribute

- Java unterscheidet zwischen Attributen
  - die für jedes Objekt unterschiedliche Werte annehmen (**Instanz-Attribute**)

```
public class Konto {
    int nummer;
    ...
}
```

Alle Instanzen der Klasse Konto können eine unterschiedliche Kontonummer haben

  - die für alle Objekte einer Klasse den identischen Wert annehmen (**Klassen-Attribute**, **Schlüsselwort** *static*)

```
public class Konto {
    static double zins;
    ...
}
```

Alle Instanzen der Klasse Konto haben denselben Wert in der Variable „zins“

Wenn der Wert in einer Instanz geändert wird, ändert er sich sofort für alle anderen Instanzen mit

Beispiel

- Klassenattribute (*static*) existieren nur einmal, egal wie viele Instanzen angelegt werden

- *Wofür ist das gut?*

```
public class Konto {
    static double zins;
(…)
    public double getZins() {
        return zins;
    }
    public void setZins(double zins) {
        this.zins = zins;
    }
(…)
public class MyClass {

    public static void main(String[] args) {
        Konto klaus = new Konto(1, "Klaus");
        Konto heinz = new Konto(2, "Heinz");

        klaus.setZins(0.25);
        System.out.println(heinz.getZins());
        // Ausgabe: 0.25
```

Instanz-Attribute vs. Klassen-Attribute

Instanz-Attribute

```
public class Konto {
    int nummer;
     ...
}
```

Konto
Zins = 3.0;
Name = Lisa;
Nummer = 1;
Betrag = 500.0;

Konto
Zins = 3.0;
Name = Steve;
Nummer = 45;
Betrag = 0.0;

Lisa's Kontonummer = 1
gilt nur in Lisa's Instanz

Steve's Kontonummer = 45
gilt nur in Steve's Instanz

Klassen-Attribute

```
public class Konto {
    static double zins = 3.0;
     ...
}
```

Konto
Zins = 3.0;
Name = Mike;
Nummer = 506;
Betrag = 610.23;

Zins = 3.0 gilt für alle Instanzen

Mike's Kontonummer = 506
gilt nur in Mike's Instanz

Zugriff: Instanz-Attribute vs. Klassen-Attribute

Klassen-Attribute

- Zugriff mit Punkt-Notation
- Zugriff über Klasse
  (keine Instanz notwendig)

```
double aktuellerZins = Konto.zins;
```

Instanz-Attribute

- Zugriff mit Punkt-Notation
- Zugriff über Instanz/Objekt

```
Konto konto = new Konto("Lisa", 150.00);
int aktuelleNummer = konto.nummer;

// besser: Zugriff über Getter-Methode
aktuelleNummer = konto.getNummer();
```

Konstanten

- Schlüsselworte **final static**
  - *static*: Klassen-Attribut, d.h. existiert nur einmal für alle Objekte
  - *final*: kann nach der Initialisierung nicht mehr geändert werden

- Code-Konvention: ausschließlich Großbuchstaben

```
public class Konto {

    // Klassen-Attribut (keine Konstante)
    static double zins = 0.25;

    // Klassen-Konstanten
    final static int EURO = 1;
    final static int DOLLAR = 2;
```

Instanz-Methoden vs. Klassen-Methoden

- Java unterscheidet zwischen Methoden,
  - die auf Instanz-Attributen arbeiten (**Instanz-Methoden**)
  - die ausschließlich auf Klassen-Attributen arbeiten (**Klassen-Methoden, kein Zugriff auf die Instanz**)
    - wieder Schlüsselwort *static*

```
public class Konto {
    static double zins;
    int nummer;

    public static double getZins( ) {
        return zins;
    }

    public int getNummer() {
        return nummer;
    }
```

Wofür ist das gut?

Klassen-Methoden

- Kein Zugriff auf Instanz-Attribute und Instanz-Methoden
- **Können aufgerufen werden, ohne eine Instanz des Objekts anzulegen!**

```
public class MyClass {

    public static void main(String[] args) {

        Konto klaus = new Konto(1, "Klaus");

        // Verändere den Zins für alle Konten,
        // ohne eine Instanz anzufassen
        Konto.setZins(0.25);

        System.out.println(klaus.getZins());
        // Ausgabe: 0.25
```

Semantisch sauberer als das Beispiel 5 Folien vorher!

Zugriff: Instanz-Methoden vs. Klassen-Methoden

- Innerhalb einer Instanz-Methode…
  - Zugriff auf jegliche Instanz- und Klassen-Elemente möglich
  - Zugriff auf Instanz-Elemente über Instanz-Bezeichner
  - Empfehlung:
    - Zugriff auf Klassen-Elemente immer über den Klassen-Bezeichner

- Innerhalb einer Klassen-Methode…
  - Zugriff auf Instanz- und Klassen-Elemente anderer Klassen möglich

  - Kein Zugriff auf Instanz-Elemente derselben Klasse

Warum ist das so?

Zusammenfassung

- Objektorientierung ist mehr als eine Programmiertechnik
  - Methode um Problem aus der realen Welt "auf natürliche Art und Weise" in ein abstraktes Modell und dann in Programmcode abzubilden
- Klasse: Bauplan für gleichartige Dinge (Objekte)
  - Eine Klasse definiert einen Referenz-Datentyp
  - Erzeugung von Objektinstanzen mit Hilfe von Konstruktoren
- Instanz-Methoden: Eine Klasse definiert Methoden, die für Objekte der Klasse aufgerufen werden können und die Attribute des Objekts manipulieren
  - *this* zum Zugriff auf Attribute der eigenen Objektinstanz
- Instanz-Attribute: Eine Klasse definiert Attribute, die über Methodenaufrufe mit Hilfe der Punktnotation über Objektvariablen für ein konkretes Objekt manipulieren werden können
- *static-Elemente*: Klassen-spezifische Elemente
  - Klassenvariablen und Klassenmethoden (vs. Instanz-Attribute und Instanz-Methoden)

Kontrollfragen

- Erläutern Sie die Begriffe Objektorientierung
- Erklären Sie den Unterschied zwischen einer Klasse und einem Objekt
- Was verbirgt sich hinter dem Begriff Kapselung?
- Was ist ein Konstruktor?
- Was bewirken die Schlüsselworte **this**, **static**, **private**, **protected** und **public**?
- Was ist der Unterschied zwischen Instanz-Attributen/Methoden und Klassen-Attributen/Methoden?