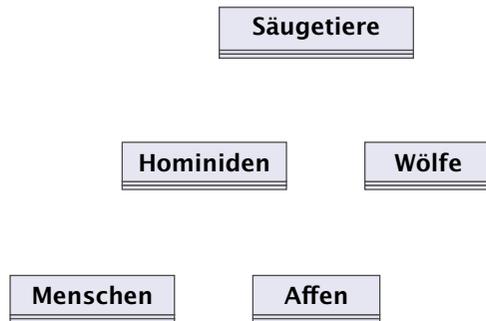


## 12 Vererbung

### Beobachtung

Oft werden mehrere Klassen von Objekten benötigt, die zwar ähnlich, aber doch verschieden sind.



## 12 Vererbung

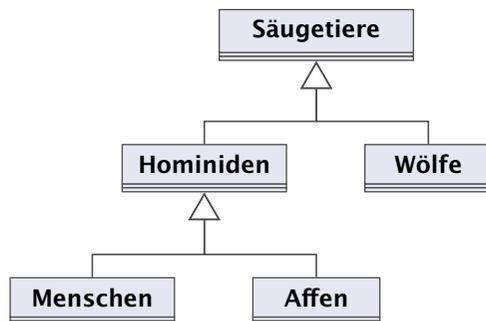
### Idee:

- ▶ Finde Gemeinsamkeiten heraus!
- ▶ Organisiere in einer Hierarchie!
- ▶ Implementiere zuerst was allen gemeinsam ist!
- ▶ Implementiere dann nur noch den Unterschied!

⇒ inkrementelles Programmieren

⇒ Software Reuse

## 12 Vererbung

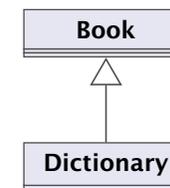


## 12 Vererbung

### Prinzip

- ▶ Die Unterklasse verfügt über all Members der Oberklasse und eventuell noch über weitere.
- ▶ Das Übernehmen von Members der Oberklasse in die Unterklasse nennt man **Vererbung** (oder **inheritance**).

### Beispiel



## Implementierung

```
1 public class Book {
2     protected int pages;
3     public Book() {
4         pages = 150;
5     }
6     public void page_message() {
7         System.out.println("Number of pages: "+pages);
8     }
9 } // end of class Book
10 // continued...
```

## Implementierung

```
1 public class Dictionary extends Book {
2     private int defs;
3     public Dictionary(int x) {
4         pages = 2*pages;
5         defs = x;
6     }
7     public void defs_message() {
8         System.out.println("Number of defs: "+defs);
9         System.out.println("Defs per page: "+defs/pages);
10    }
11 } // end of class Dictionary
```

## Erlauierungen

- ▶ `class A extends B { ... }` deklariert die Klasse **A** als Unterklasse der Klasse **B**.
- ▶ Alle Members von **B** stehen damit automatisch auch der Klasse **A** zur Verfugung.
- ▶ Als **protected** klassifizierte Members sind auch in der Unterklasse **sichtbar**.
- ▶ Als **private** deklarierte Members konnen dagegen in der Unterklasse **nicht** direkt aufgerufen werden, da sie dort nicht sichtbar sind.
- ▶ Wenn ein Konstruktor der Unterklasse **A** aufgerufen wird, wird **implizit** zuerst der Konstruktor `B()` der Oberklasse aufgerufen.

## Beispiel

`Dictionary webster = new Dictionary(12400);`  
liefert



## Methodenaufruf

```
1 public class Words {
2     public static void main(String[] args) {
3         Dictionary webster = new Dictionary(12400);
4         webster.page_message();
5         webster.defs_message();
6     } // end of main
7 } // end of class Words
```

- ▶ Das neue Objekt `webster` enthält die Attribute `pages` und `defs`, sowie die Objekt-Methoden `page_message()` und `defs_message()`.
- ▶ Kommen in der Unterklasse nur weitere Members hinzu, spricht man von einer `is_a`-Beziehung. (Oft müssen aber Objekt-Methoden der Oberklasse in der Unterklasse umdefiniert werden.)

## Methodenaufruf

Die Programmausführung liefert:

Number of pages: 300  
Number of defs: 12400  
Defs per page: 41



## 12.1 Das Schlüsselwort `super`

- ▶ Manchmal ist es erforderlich, in der Unterklasse **explizit** die Konstruktoren oder Objekt-Methoden der Oberklasse aufzurufen. Das ist der Fall, wenn
  - ▶ Konstruktoren der Oberklasse aufgerufen werden sollen, die Parameter besitzen;
  - ▶ Objekt-Methoden oder Attribute der Oberklasse und Unterklasse gleiche Namen haben.
- ▶ Zur Unterscheidung der aktuellen Klasse von der Oberklasse dient das Schlüsselwort `super`.

## Beispiel

```
1 public class Book {
2     protected int pages;
3     public Book(int x) {
4         pages = x;
5     }
6     public void message() {
7         System.out.println("Number of pages: "+pages);
8     }
9 } // end of class Book
```



## Beispiel

```
1 public class Dictionary extends Book {
2     private int defs;
3     public Dictionary(int p, int d) {
4         super(p);
5         defs = d;
6     }
7     public void message() {
8         super.message();
9         System.out.println("Number of defs: "+defs);
10        System.out.println("Defs per page: "+defs/pages);
11    }
12 } // end of class Dictionary
```



## Erläuterungen

### „super“ als Konstruktoraufruf

- ▶ `super(...)`; ruft den entsprechenden Konstruktor der Oberklasse auf.
- ▶ Analog gestattet `this(...)`; den entsprechenden Konstruktor der eigenen Klasse aufzurufen.
- ▶ Ein solcher expliziter Aufruf muss stets ganz am Anfang eines Konstruktors stehen.



## Erläuterungen

Die exakten Sichtbarkeitsregeln werden auf den folgenden Folien präzisiert.

### „super.“ zum Zugriff auf members der Oberklasse

Deklariert eine Klasse `A` einen Member `memb` gleichen Namens wie in einer Oberklasse, so ist nur noch der Member `memb` aus `A` sichtbar.

- ▶ Methoden mit unterschiedlichen Argument-Typen werden als verschieden angesehen.
- ▶ `super.memb` greift für das aktuelle Objekt `this` auf Attribute oder Objekt-Methoden `memb` der Oberklasse zu.
- ▶ Eine andere Verwendung von `super.` ist **nicht gestattet**.

Insbesondere Zugriffe wie `super.super.memb` sind nicht möglich.



## Verschattung von Variablen

### Falls `memb` eine Methode ist:

- ▶ Wenn `memb` eine Methode mit den gleichen Argumenttypen (in der gleichen Reihenfolge), und dem gleichen Rückgabetypen ist, dann ist zunächst nur `memb` aus `A` sichtbar (**Überschreiben**).  
(seit Java 5 darf der Rückgabetyptyp auch ein Untertyp des ursprünglichen Rückgabetyptyps sein)
- ▶ Wenn `memb` eine Methode mit unterschiedlichen Argumenttypen ist, dann sind sowohl `memb` aus `A` als auch die Methode der Oberklasse sichtbar (**Überladen**).
- ▶ Wenn die Argumenttypen übereinstimmen, aber der Rückgabetyptyp kein Untertyp ist, erhält man einen Compilerfehler.

Der kovariante Rückgabetyptyp ist nur für Referenzdatentypen möglich.

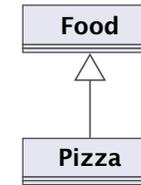


## Verschattung von Variablen

Falls `memb` eine Variable ist:

- ▶ Direkt (d.h. ohne `super.`) ist nur `memb` aus `A` sichtbar. `memb` kann einen anderen Typ als in der Oberklasse haben.

## 12.2 Private Variablen und Methoden



Das Programm `Eating` soll die Anzahl der **Kalorien pro Mahlzeit** ausgeben.

## Implementierung

```
1 public class Eating {
2     public static void main (String[] args) {
3         Pizza special = new Pizza(275);
4         System.out.print("Calories per serving: " +
5             special.caloriesPerServing());
6     } // end of main
7 } // end of class Eating
```

## Implementierung

```
7 public class Food {
8     private int CALORIES_PER_GRAM = 9;
9     private int fat, servings;
10    public Food (int numFatGrams, int numServings) {
11        fat = numFatGrams;
12        servings = numServings;
13    }
14    private int calories() {
15        return fat * CALORIES_PER_GRAM;
16    }
17    public int caloriesPerServing() {
18        return calories() / servings;
19    }
20 } // end of class Food
```

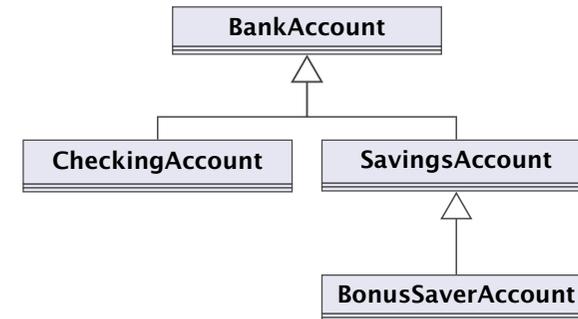
## Implementierung + Erläuterungen

```
21 public class Pizza extends Food {
22     public Pizza (int amountFat) {
23         super(amountFat,8);
24     }
25 } // end of class Pizza
```

- ▶ Die Unterklasse **Pizza** verfügt über alle Members der Oberklasse **Food** — nicht alle **direkt** zugänglich.
- ▶ Die Attribute und die Objekt-Methode **calories()** der Klasse **Food** sind privat, und damit für Objekte der Klasse **Pizza** verborgen.
- ▶ Trotzdem können sie von der **public** Objekt-Methode **caloriesPerServing** benutzt werden.

Ausgabe des Programms:      Calories per serving: 309

## 12.3 Überschreiben von Methoden



## Aufgabe

- ▶ Implementierung von einander abgeleiteter Formen von Bankkonten.
- ▶ Jedes Konto kann eingerichtet werden, erlaubt Einzahlungen und Auszahlungen.
- ▶ Verschiedene Konten verhalten sich unterschiedlich in Bezug auf Zinsen und Kosten von Kontobewegungen.

## Einige Konten

```
1 public class Bank {
2     public static void main(String[] args) {
3         SavingsAccount savings =
4             new SavingsAccount(4321, 5028.45, 0.02);
5         BonusSaverAccount bigSavings =
6             new BonusSaverAccount (6543, 1475.85, 0.02);
7         CheckingAccount checking =
8             new CheckingAccount (9876,269.93, savings);
9         savings.deposit(148.04);    System.out.println();
10        bigSavings.deposit(41.52);  System.out.println();
11        savings.withdraw(725.55);   System.out.println();
12        bigSavings.withdraw(120.38); System.out.println();
13        checking.withdraw(320.18);  System.out.println();
14    } // end of main
15 } // end of class Bank
```

"Bank.java"

Die Parameter beschreiben id, initialer Kontostand, und Zinssatz für die ersten beiden Typen. Für den **CheckingAccount** gibt es keine Zinsen. Der dritte Parameter ist ein zugehöriger **SavingsAccount**.



## Implementierung

```
1 public class BankAccount {
2     // Attribute aller Konten-Klassen:
3     protected int account;
4     protected double balance;
5     // Konstruktor:
6     public BankAccount(int id, double initial) {
7         account = id; balance = initial;
8     }
9     // Objekt-Methoden:
10    public void deposit(double amount) {
11        balance = balance + amount;
12        System.out.println(
13            "Deposit into account " + account + "\n"
14            + "Amount:\t\t" + amount + "\n"
15            + "New balance:\t" + balance);
16    }
```

"BankAccount.java"

## Erläuterungen

- ▶ Anlegen eines Kontos `BankAccount` speichert eine (hoffentlich neue) Kontonummer sowie eine Anfangseinlage.
- ▶ Die zugehörigen Attribute sind `protected`, d.h. können nur von Objekt-Methoden der Klasse bzw. ihrer Unterklassen modifiziert werden.
- ▶ die Objekt-Methode `deposit` legt Geld aufs Konto, d.h. modifiziert den Wert von `balance` und teilt die Kontobewegung mit.



## Implementierung

```
17 public boolean withdraw(double amount) {
18     System.out.println(
19         "Withdrawal from account "+ account + "\n"
20         + "Amount:\t\t" + amount);
21     if (amount > balance) {
22         System.out.println(
23             "Sorry, insufficient funds...");
24         return false;
25     }
26     balance = balance - amount;
27     System.out.println(
28         "New balance:\t"+ balance);
29     return true;
30 }
31 } // end of class BankAccount
```

"BankAccount.java"

## Erläuterungen

- ▶ Die Objekt-Methode `withdraw()` nimmt eine Auszahlung vor.
- ▶ Falls die Auszahlung scheitert, wird eine Mitteilung gemacht.
- ▶ Ob die Auszahlung erfolgreich war, teilt der Rückgabewert mit.
- ▶ Ein `CheckingAccount` verbessert ein normales Konto, indem im Zweifelsfall auf die Rücklage eines Sparkontos zurückgegriffen wird.



## Ein Girokonto

```
1 public class CheckingAccount extends BankAccount {
2     private SavingsAccount overdraft;
3     // Konstruktor:
4     public CheckingAccount(int id, double initial,
5         SavingsAccount savings) {
6         super(id, initial);
7         overdraft = savings;
8     }
9 }
```

"CheckingAccount.java"



## Modifiziertes withdraw()

```
8 // modifiziertes withdraw():
9 public boolean withdraw(double amount) {
10     if (!super.withdraw(amount)) {
11         System.out.println("Using overdraft...");
12         if (!overdraft.withdraw(amount - balance)) {
13             System.out.println(
14                 "Overdraft source insufficient.");
15             return false;
16         } else {
17             balance = 0;
18             System.out.println(
19                 "New balance on account "
20                 + account + ": 0");
21         }
22     }
23     return true;
24 }
25 } // end of class CheckingAccount
```

"CheckingAccount.java"

## Erläuterungen

- ▶ Die Objekt-Methode `withdraw` wird neu definiert, die Objekt-Methode `deposit` wird übernommen.
- ▶ Der Normalfall des Abhebens erfolgt (als Seiteneffekt) beim Testen der ersten `if`-Bedingung.
- ▶ Dazu wird die `withdraw`-Methode der Oberklasse aufgerufen.
- ▶ Scheitert das Abheben mangels Geldes, wird der Fehlbetrag vom Rücklagen-Konto abgehoben.
- ▶ Scheitert auch das, erfolgt keine Konto-Bewegung, dafür eine Fehlermeldung.
- ▶ Andernfalls sinkt der aktuelle Kontostand auf 0 und die Rücklage wird verringert.



## Ein Sparbuch

```
1 public class SavingsAccount extends BankAccount {
2     protected double interestRate;
3     // Konstruktor:
4     public SavingsAccount(int id, double init, double rate) {
5         super(id, init);
6         interestRate = rate;
7     }
8     // zusätzliche Objekt-Methode:
9     public void addInterest() {
10         balance = balance * (1 + interestRate);
11         System.out.println(
12             "Interest added to account: " + account
13             + "\nNew balance:\t" + balance);
14     }
15 } // end of class SavingsAccount
```

"SavingsAccount.java"



## Erläuterungen

- ▶ Die Klasse `SavingsAccount` erweitert die Klasse `BankAccount` um das zusätzliche Attribut `double interestRate` (Zinssatz) und eine Objekt-Methode, die die Zinsen gutschreibt.
- ▶ Alle sonstigen Attribute und Objekt-Methoden werden von der Oberklasse geerbt.
- ▶ Die Klasse `BonusSaverAccount` erhöht zusätzlich den Zinssatz, führt aber Strafkosten fürs Abheben ein.



## Ein Bonus-Sparbuch

```
1 public class BonusSaverAccount extends SavingsAccount {
2     private int penalty;
3     private double bonus;
4     // Konstruktor:
5     public BonusSaverAccount(int id, double init,
6                               double rate) {
7         super(id, init, rate);
8         penalty = 25;
9         bonus = 0.03;
10    }
11    // Modifizierung der Objekt-Methoden:
12    public boolean withdraw(double amount) {
13        boolean res;
14        if (res = super.withdraw(amount + penalty))
15            System.out.println(
16                "Penalty incurred:\t"+ penalty);
17        return res;
18    }
```

"BonusSaverAccount.java"

## Ein Bonus-Sparbuch

```
19 public void addInterest() {
20     balance = balance * (1 + interestRate + bonus);
21     System.out.println(
22         "Interest added to account: " + account
23         + "\nNew balance:\t" + balance);
24 }
25 } // end of class BonusSaverAccount
```

"BonusSaverAccount.java"



## Programmausgabe

Deposit into account 4321  
Amount: 148.04  
New balance: 5176.49

Deposit into account 6543  
Amount: 41.52  
New balance: 1517.37

Withdrawal from account 4321  
Amount: 725.55  
New balance: 4450.94

Withdrawal from account 6543  
Amount: 145.38  
New balance: 1371.9899999999998  
Penalty incurred: 25

Withdrawal from account 9876  
Amount: 320.18  
Sorry, insufficient funds...  
Using overdraft...  
Withdrawal from account 4321  
Amount: 50.25  
New balance: 4400.69  
New balance on account 9876: 0

