- `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.
- 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.

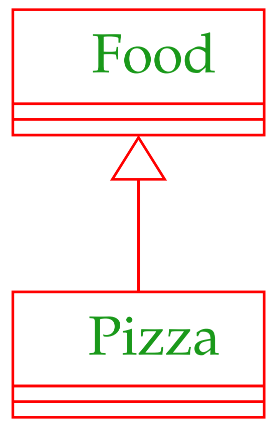
```
public class Words {
    public static void main(String[] args) {
        Dictionary webster = new Dictionary(540,36600);
        webster.message();
    } // end of main
} // end of class Words
```

- Das neue Objekt `webster` enthält die Attribute `pages` wie `defs`.
- Der Aufruf `webster.message()` ruft die Objekt-Methode der Klasse `Dictionary` auf.
- Die Programm-Ausführung liefert:

```
Number of pages:        540
Number of defs:         36600
Defs per page:          67
```

12.2 Private Variablen und Methoden

Beispiel:



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

```
public class Eating {
   public static void main (String[] args) {
      Pizza special = new Pizza(275);
      System.out.print("Calories per serving: " +
         special.calories_per_serving());
   } // end of main
} // end of class Eating
```

```
public class Food {
   private int CALORIES_PER_GRAM = 9;
   private int fat, servings;
   public Food (int num_fat_grams, int num_servings) {
      fat = num_fat_grams;
      servings = num_servings;
   }
   private int calories() {
      return fat * CALORIES_PER_GRAM;
   }
   public int calories_per_serving() {
      return (calories() / servings);
   }
} // end of class Food
```

```
public class Pizza extends Food {
   public Pizza (int amount_fat) {
      super (amount_fat,8);
      }
   } // end of class Pizza
```

- Die Unterklasse `Pizza` verfügt über alle Members der Oberklasse `Food` – wenn auch nicht alle direkt zugänglich sind.
- 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 `calories_per_serving` benutzt werden.

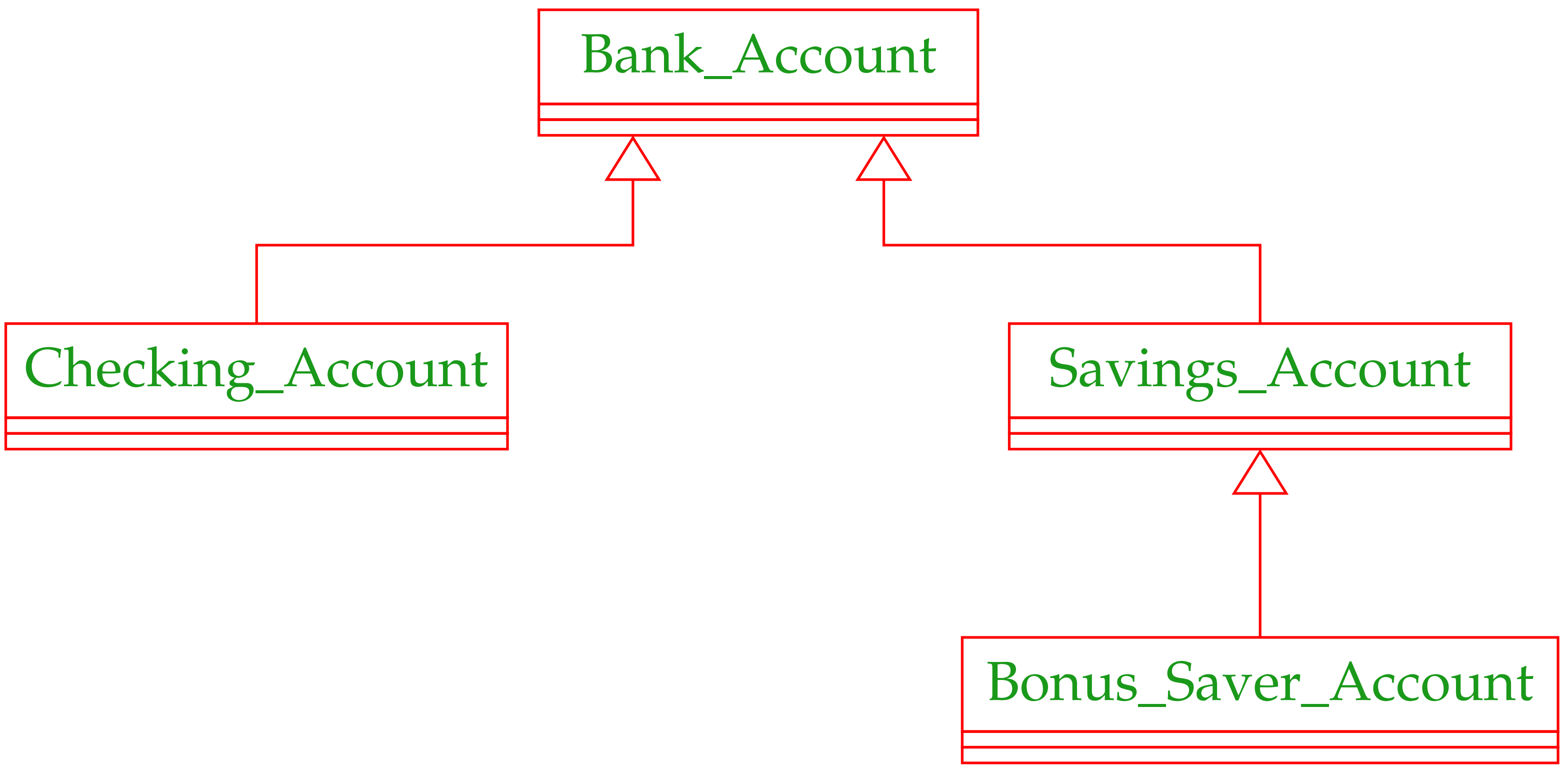
```
public class Pizza extends Food {
   public Pizza (int amount_fat) {
      super (amount_fat,8);
      }
   } // end of class Pizza
```

- Die Unterklasse `Pizza` verfügt über alle Members der Oberklasse `Food` – wenn auch nicht alle direkt zugänglich sind.
- 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 `calories_per_serving` benutzt werden.

... Ausgabe des Programms: `Calories per serving: 309`

12.3 Überschreiben von Methoden

Beispiel:



Aufgabe:

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

Einige Konten:

```
public class Bank {
   public static void main(String[] args) {
      Savings_Account savings =
         new Savings_Account (4321, 5028.45, 0.02);
      Bonus_Saver_Account big_savings =
         new Bonus_Saver_Account (6543, 1475.85, 0.02);
      Checking_Account checking =
         new Checking_Account (9876,269.93, savings);
      ...
```

Einige Konto-Bewegungen:

```
        savings.deposit (148.04);
        big_savings.deposit (41.52);
        savings.withdraw (725.55);
        big_savings.withdraw (120.38);
        checking.withdraw (320.18);
    } // end of main
} // end of class Bank
```

Einige Konto-Bewegungen:

```
        savings.deposit (148.04);
        big_savings.deposit (41.52);
        savings.withdraw (725.55);
        big_savings.withdraw (120.38);
        checking.withdraw (320.18);
    } // end of main
} // end of class Bank
```

Fehlt nur noch die Implementierung der Konten selbst :-)

```
public class Bank_Account {
   // Attribute aller Konten-Klassen:
   protected int account;
   protected double balance;
   // Konstruktor:
   public Bank_Account (int id, double initial) {
      account = id; balance = initial;
      }
   // Objekt-Methoden:
   public void deposit(double amount) {
      balance = balance+amount;
      System.out.print("Deposit into account "+account+"\n"
                     +"Amount:\t\t"+amount+"\n"
     +"New balance:\t"+balance+"\n\n");
      }
      ...
```

- Anlegen eines Kontos `Bank_Account` speichert eine (hoffentlich neue) Konto-Nummer sowie eine Anfangs-Einlage.
- 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 Konto-Bewegung mit.

```
    public boolean withdraw(double amount) {
        System.out.print("Withdrawal from account "+ account +"\n"
                         +"Amount:\t\t"+ amount +"\n");
        if (amount > balance) {
            System.out.print("Sorry, insufficient funds...\n\n");
            return false;
        }
        balance = balance-amount;
        System.out.print("New balance:\t"+ balance +"\n\n");
        return true;
    }
} // end of class Bank_Account
```

- 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 `Checking_Account` verbessert ein normales Konto, indem im Zweifelsfall auf die Rücklage eines Sparkontos zurückgegriffen wird.

Ein Giro-Konto:

```
public class Checking_Account extends Bank_Account {
   private Savings_Account overdraft;
// Konstruktor:
   public Checking_Account(int id, double initial,
                               Savings_Account savings) {
      super (id, initial);
      overdraft = savings;
      }
   ...
```

```
// modifiziertes withdraw():
   public boolean withdraw(double amount) {
      if (!super.withdraw(amount)) {
         System.out.print("Using overdraft...\n");
         if (!overdraft.withdraw(amount-balance)) {
            System.out.print("Overdraft source insufficient.\n\n");
            return false;
         } else {
            balance = 0;
            System.out.print("New balance on account "+ account +": 0\n\n");
      }  }
      return true;
   }
} // end of class Checking_Account
```

- 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:

```
public class Savings_Account extends Bank_Account {
   protected double interest_rate;
// Konstruktor:
   public Savings_Account (int id, double init, double rate) {
      super(id,init); interest_rate = rate;
      }
// zusaetzliche Objekt-Methode:
   public void add_interest() {
      balance = balance * (1+interest_rate);
      System.out.print("Interest added to account: "+ account
                    +"\nNew balance:\t"+ balance +"\n\n");
   }
} // end of class Savings_Account
```

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

Ein Bonus-Sparbuch:

```
public class Bonus_Saver_Account extends Savings_Account {
   private int penalty;
   private double bonus;
// Konstruktor:
   public Bonus_Saver_Account(int id, double init, double rate) {
      super(id, init, rate); penalty = 25; bonus = 0.03;
   }
// Modifizierung der Objekt-Methoden:
   public boolean withdraw(double amount) {
      System.out.print("Penalty incurred:\t"+ penalty +"\n");
      return super.withdraw(amount+penalty);
      }
   ...
```

```
    public void add_interest() {
        balance = balance * (1+interest_rate+bonus);
        System.out.print("Interest added to account: "+ account
                    +"\nNew balance:\t" + balance +"\n\n");
    }
} // end of class Bonus_Safer_Account
```

... als Ausgabe erhalten wir dann:

```
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
```

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

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
```

13 Abstrakte Klassen, finale Klassen und Interfaces

- Eine abstrakte Objekt-Methode ist eine Methode, für die keine Implementierung bereit gestellt wird.
- Eine Klasse, die abstrakte Objekt-Methoden enthält, heißt ebenfalls abstrakt.
- Für eine abstrakte Klasse können offenbar keine Objekte angelegt werden :-)
- Mit abstrakten können wir Unterklassen mit verschiedenen Implementierungen der gleichen Objekt-Methoden zusammenfassen.

Beispiel: Implementierung der JVM

```
public abstract class Instruction {
    protected static IntStack stack = new IntStack();
    protected static int pc = 0;
    public boolean halted() { return false; }
    abstract public int execute();
} // end of class Instruction
```

- Die Unterklassen von `Instruction` repräsentieren die Befehle der JVM.
- Allen Unterklassen gemeinsam ist eine Objekt-Methode `execute()` – immer mit einer anderen Implementierung :-)
- Die statischen Variablen der Oberklasse stehen sämtlichen Unterklassen zur Verfügung.

- Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- Für die abstrakte Methode muss der vollständige Kopf angegeben werden – inklusive den Parameter-Typen und den (möglicherweise) geworfenen Exceptions.
- Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `boolean halted()`.
- Die angebene Implementierung liefert eine Default-Implementierung für `boolean halted()`.
- Klassen, die eine andere Implementierung brauchen, können die Standard-Implementierung ja überschreiben :-)

- Die Methode `execute()` soll die Instruktion ausführen und als Rückgabe-Wert den `pc` des nächsten Befehls ausgeben.

Beispiel für eine Instruktion:

```
public final class Const extends Instruction {
   private int n;
   public Const(int x) { n=x; }
   public int execute() {
      stack.push(n);
      return ++pc;
   }  // end of execute()
}     // end of class Const
```

- Der Befehl CONST benötigt ein Argument. Dieses wird dem Konstruktor mitgegeben und in einer privaten Variable gespeichert.
- Die Klasse ist als `final` deklariert.
- Zu als final deklarierten Klassen dürfen keine Unterklassen deklariert werden !!!
- Aus Sicherheits- wie Effizienz-Gründen sollten so viele Klassen wie möglich als final deklariert werden ...
- Statt ganzer Klassen können auch einzelne Variablen oder Methoden als `final` deklariert werden.
- Finale Members dürfen nicht in Unterklassen umdefiniert werden.
- Finale Variablen dürfen zusätzlich nur initialisiert, aber nicht modifiziert werden $\Longrightarrow$ Konstanten.

... andere Instruktionen:

```
public final class Sub extends Instruction {
   public int execute() {
      final int y = stack.pop();
      final int x = stack.pop();
      stack.push(x-y); return ++pc;
   }  // end of execute()
}     // end of class Sub
public final class Halt extends Instruction {
   public boolean halted() {
      pc=0; stack = new IntStack(); return true;
      }
   public int execute() { return 0; }
}     // end of class Halt
```