Noch ein Beispiel

- Wir wollen in einer Bank die Konten der Kunden modellieren (um sie dann entspr. zu verarbeiten

```
public class Konto
{
  private String kundenName;
  private double kontoStand;
  private double dispoLimit;
  private final int KONTO_NR;

  public Konto(String kundenName, int kontoNR)
  {
    this.kundenName = kundenName;
    this.kontoStand = 0.0;
    this.KONTO_NR = kontoNR;
  }

  public void abheben(double betrag) {
    if(this.kontostand - betrag > this.dispolimit) { kontostand = kontostand - berag; }
    else { System.out.println("Abheben nicht moeglich!!!"); }
  }

  public void setDispoLimit(double dispoLimit) {
    this.dispoLimit = dispoLimit;
  }

  // Getter-Methoden
  ...
}
```

Nochmal der Sinn von Kapselung

- Die Veränderung des Zustands durch das Geldabheben ist nur über die Methode `abheben` möglich und damit wohldefiniert
- Ist die Methode einmal richtig implementiert, kann man sie überall wiederverwenden
- Dies ist offenbar ein Schutz vor fehlerhaftem Verhalten

Zusammenspiel von imperativen und oo Aspekten in Java

- Noch einmal der Unterschied zwischen *statischen* und *nicht-statischen* Elemente einer Klasse:
- Die statischen Elemente (Methoden oder Attribute) Elemente existieren unabhängig von Objekten, wogegen nicht-statische Elemente an die Existenz von Objekten gebunden sind
- Werden statische Variablen (Klassenvariablen) von einem Objekt verändert, ist diese Veränderung auch in allen anderen Objekten der gleichen Klasse sichtbar
- Dies ist bei nicht-statischen Variablen (Attributen) natürlich nicht so
- Beide Arten von Elementen kann man nebeneinander verwenden und manchmal auch sehr sinnvoll kombinieren

Zusammenspiel von imperativen und oo Konzepten

- Ein imperatives Programm `Programm` in Java besteht aus einer Klassendefinition für die Klasse `Programm` mit einer statischen `main`-Methode
- Die Klasse `Programm` ist dabei (typischerweise) keine Vereinbarung eines neuen Datentyps, sondern ein Modul
- Die Erzeugung einzelner Objekte der Klasse `Programm` ist daher vermutlich gar nicht vorgesehen (auch wenn dies mit dem Default-Konstruktor möglich wäre)
- Die `main`-Methode existiert trotzdem unabhängig von Objekten der Klasse `Programm`

Zusammenspiel von imperativen und oo Konzepten

- Wie bereits erwähnt, gibt es auch in der Java Klassenbibliothek einige Module, die ausschließlich statische Elemente zur Verfügung stellen, d.h. es ist vom entsprechenden Programmierer nicht vorgesehen, Objekte dieser Klassen zu erzeugen
- (Teils schon bekannte) Beispiele:
  - Die Klasse `java.lang.Math`
  - Die Klasse `java.util.Arrays`
    This class contains various methods for manipulating arrays (such as sorting and searching)
  - Die Klasse `java.lang.System`
    mit der statischen Variable `System.out` vom Typ `java.io.PrintStream` stellt es u.a. einen Verweis auf den Standard-Outputstream zur Verfügung (Default ist das der Bildschirm); die Klasse `java.io.PrintStream` stellt wiederum statische Methoden `PrintStream.print` und `PrintStream.println` zur Verfügung, deren Wirken wir ja schon kennen

Zusammenspiel von imperativen und oo Konzepten

- Das Attribut `kontoStand` der Klasse `Konto` ist dagegen ein Objekt-Attribut
- Es existiert nur dann, wenn es mindestens ein Objekt der Klasse `Konto` gibt
- Für jedes existierende Objekt der Klasse `Konto` existiert dieses Attribut
- Für jedes unterschiedliche Objekt der Klasse `Konto` hat dieses Attribute möglicherweise einen anderen Wert

Zusammenspiel von imperativen und oo Konzepten

- Die Bank wünscht sich nun, für jedes neueröffnete Konto wird eine neue, fortlaufende Kontonummer zu vergeben, d.h. der Mitarbeiter vergibt die Nummer nicht mehr beim Anlegen des Kontos
- Das kann z.B. mit einer statischen Variablen `aktuelleKNR` realisiert werden, die bei jeder neuen Kontoeröffnung gelesen wird (um die neue Kontonummer zu bestimmen) und anschließend inkrementiert wird
- Dieses Attribut sollte **`private`** sein, damit nur die Objekte der Klasse darauf zugreifen können
- Das heißt die Klasse `Konto` wird neben den nicht-statischen Attributen, die für jedes neue Objekt neu angelegt werden (z.B. für den Namen des Kontoinhabers) auch das statische Attribut `aktuelleKNR` haben, das unabhängig von den existierenden Objekten der Klasse `Konto` existiert und verwendet werden kann.

Zusammenspiel von imperativen und oo Konzepten

```
public class Konto
{
  /* ** Statische (objekt-unabhaengige) Attribute ** */
  private static int aktuelleKNR = 1;

  /* ** Nicht-statische (objekt-abhaengige) Attribute ** */
  private String kundenName;
  private double kontoStand;
  private double dispoLimit;
  private final int KONTO_NR;
  ... // weitere Attribute

  /* ** Nicht-statische (objekt-abhaengige) Konstruktor ** */
  public Konto(String kundenName)
  {
    this.kundenName = kundenName;
    kontoStand = 0.0;
    KONTO_NR = aktuelleKNR;
    aktuelleKNR++;
  }

  /* ** Nicht-statische (objekt-abhaengige) Methoden ** */
  ...
}
```

Beispiel: Wrapper-Klassen für primitive Typen

- Zu jedem primitiven Datentyp in Java gibt es eine korrespondierende (sog. *Wrapper-*) Klasse, die den primitiven Typ in einer oo Hülle kapselt.
- Es gibt Situationen, bei denen man diese Wrapper-Klassen anstelle der primitiven Typen benötigt. Z.B. werden in Java einige Klassen zur Verfügung gestellt, die eine (dynamische) Menge von beliebigen Objekttypen speichern können. Um darin auch primitive Typen ablegen zu können, benötigt man die Wrapper-Klassen.
- Zu allen numerischen Typen und zu den Typen **`char`** und **`boolean`** existieren Wrapper-Klassen.

Beispiel: Wrapper-Klassen für primitive Typen

Wrapper-Klasse	Primitiver Typ
`Byte`	**`byte`**
`Short`	**`short`**
`Integer`	**`int`**
`Long`	**`long`**
`Double`	**`double`**
`Float`	**`float`**
`Boolean`	**`boolean`**
`Character`	**`char`**
`Void`	**`void`**

Beispiel: Wrapper-Klassen für primitive Typen

- Zur Objekterzeugung stellen die Wrapper-Klassen haupsächlich zwei Konstruktoren zur Verfügung:
  - Für jeden primitiven Typ `type` stellt die Wrapperklasse `Type` einen Konstruktor zur Verfügung, der einen primitiven Wert des Typs `type` als Argument fordert:
    z.B. `public Integer(int i)`
  - Zusätzlich gibt es bei den meisten Wrapper-Klassen die Möglichkeit, einen String zu übergeben:
    z.B. `public Integer(String s)` wandelt die Zeichenkette `s` in einen Integer um, z.B. `"123"` in den `int`-Wert `123`.

Beispiel: Wrapper-Klassen für primitive Typen

- Kapselung:
  - Der Zugriff auf den Wert des Objekts erfolgt ausschließlich lesend über entsprechende Methoden:
    z.B. `public int intValue()`
  - Die interne Realisierung (wie wird der Wert gespeichert) ist dem Benutzer verborgen.
  - Insbesondere kann der Wert des Objekts nicht verändert werden.

Beispiel: Wrapper-Klassen für primitive Typen

- Statische Elemente (u.a.):
  - Wichtige Literale aus dem enstrechenden Wertebereich, z.B. Konstanten
    `public static int MAX_VALUE` bzw.
    `public static int MIN_VALUE`
    für den maximal/minimal darstellbaren **`int`**-Wert
    oder z.B. Konstanten
    `public static double NEGATIVE_INFINITY` bzw.
    `public static double POSITIVE_INFINITY`
    für $-\infty$ und $+\infty$
  - Hilfsmethoden wie z.B.
    `static double parseDouble(String s)` der Klasse `Double` wandelt die Zeichenkette `s` in ein primitiven **`double`**-Wert um und gibt den **`double`**-Wert aus.