

Abschnitt 8: Grundlagen der objektorientierten Programmierung

8. Grundlagen der objektorientierten Programmierung

- 8.1 Abstrakte Datentypen I: Benutzereigene Strukturen
- 8.2 Abstrakte Datentypen II: von Structures zu Klassen
- 8.3 Das objektorientierte Paradigma
- 8.4 Klassen und Objekte in Java
- 8.5 Vererbung, abstrakte Klassen, Polymorphismus
- 8.6 Interfaces
- 8.7 Speicherverwaltung in Java

Überblick

8. Grundlagen der objektorientierten Programmierung

8.1 Abstrakte Datentypen I: Benutzereigene Strukturen

8.2 Abstrakte Datentypen II: von Structures zu Klassen

8.3 Das objektorientierte Paradigma

8.4 Klassen und Objekte in Java

8.5 Vererbung, abstrakte Klassen, Polymorphismus

8.6 Interfaces

8.7 Speicherverwaltung in Java

Grenzen der bisherigen Konzepte

- ▶ Bei der Darstellung der Daten, die wir verarbeiten wollten, sind wir teilweise auf Grenzen gestoßen, insbesondere dann, wenn die Daten eine komplexe Struktur haben.
- ▶ Bei der Darstellung des Wechselgelds haben wir verschiedene Möglichkeiten der Datendarstellung mit Arrays verwendet.
- ▶ Bei genauerem Hinsehen ist keiner der Lösungen frei von massiven Nachteilen (potentiellen Fehlerquellen):
 - ▶ Bei der Verarbeitung des Ergebnisses durch den Wechselgeldautomat muss der Programmierer genau wissen, wie das Ergebnis-Array aufgebaut ist.
 - ▶ Das Wechselgeld kann später (z.B. unwissentlich) von einem Programmierer verändert werden, obwohl dies vermutlich wenig sinnvoll ist.

Grenzen der bisherigen Konzepte

- ▶ Woran liegt das?
- ▶ Das Wechselgeld besteht aus einer Menge von unterschiedlichen Teilen (1 EUR, 2 EUR, 5 EUR Münzen/Scheinen, je nach Rechnungsbetrag), die zusammen genommen ein *Objekt* (der realen Welt) darstellen, das Teil der Verarbeitung ist.
- ▶ Unsere bisherigen Konzepte geben so einen *Objektbegriff* nicht her.
- ▶ Wir können solche komplex strukturierten, aus mehreren Teilen bestehenden Objekte bisher nur über Arrays modellieren und hoffen, dass alle, die diese Modellierung verwenden, sie richtig verwenden.

Grenzen der bisherigen Konzepte

- ▶ Schade eigentlich.
- ▶ Genau genommen könnte man ein „Wechselgeld-Objekt“ ja (ähnlich wie in der Variante im vorherigen Kapitel) so modellieren, dass es drei Werte besitzt: jeweils einen für die 1er, 2er und 5er Münzen/Scheine.
- ▶ Ein „Wechselgeld-Objekt“ besitzt also drei *Attribute*:
 - ▶ $anzahl1er \in \mathbb{N}$
 - ▶ $anzahl2er \in \mathbb{N}$
 - ▶ $anzahl5er \in \mathbb{N}$

und wäre damit formal ein Element aus $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$.

- ▶ Ein konkretes „Wechselgeld-Objekt“ wäre z.B. (0, 2, 3) für einen Rückbetrag von 19 EUR.



Grenzen der bisherigen Konzepte

- ▶ Wir können natürlich einfach die Möglichkeit einrichten, solche komplex strukturierten „Objekte“ als eigenen Datentyp zu definieren.
- ▶ Dieser Datentyp könnte z.B. *WechselGeld* heißen und wir können dessen „Struktur“ festlegen mit:

$$\textit{WechselGeld} \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N}.$$

- ▶ Unter der Annahme, dass wir diesen *benutzer-definierten Datentypen* irgendwo vereinbart haben, könnten wir unseren Algorithmus zur Wechselgeld-Berechnung abändern, sodass er ein entsprechendes Objekt dieses Typs zurück gibt:

$$\textit{wechselGeld} : \mathbb{N} \rightarrow \textit{WechselGeld}.$$



Grenzen der bisherigen Konzepte

- ▶ Um mit diesen Objekten des Typs *WechselGeld* arbeiten zu können (die Anzahl der einzelnen Komponenten muss ja im Algorithmus *wechselGeld* gesetzt werden und der Geldautomat muss dann darauf zugreifen können) muss auch klar sein, wie man auf die einzelnen Komponenten des kartesischen Produkts zugreifen und sie verändern kann.
- ▶ Wir können z.B. festlegen, dass für Objekte $o \in \textit{WechselGeld}$ gilt:
 - ▶ $o[1]$ bezeichnet die erste Komponente des kartesischen Produkts, hier die Anzahl der 1er Münzen im Wechselgeld.
 - ▶ $o[2]$ bezeichnet die zweite Komponente des kartesischen Produkts, hier die Anzahl der 2er Münzen im Wechselgeld.
 - ▶ $o[3]$ bezeichnet die dritte Komponente des kartesischen Produkts, hier die Anzahl der 5er Scheine im Wechselgeld.

Grenzen der bisherigen Konzepte

- ▶ Damit ist unser Problem aber noch nicht wirklich gelöst, sondern eigentlich nur anders verpackt (schon die Schreibweise des Komponentenzugriffs erinnert fatal an Arrays).
- ▶ Ein Programmierer muss immer noch die Komplexität verstehen, nun die Struktur von *WechselGeld*-Objekten (d.h. ganz allgemein des Datentyps).
- ▶ Ein Programmierer kann auch immer noch ein durch den Algorithmus *wechselGeld* „erzeugtes“ Objekt später jederzeit verändern (ob gewollt oder ungewollt).

Grenzen der bisherigen Konzepte

- ▶ Alternativ, um die Komplexität zu vermindern, könnten wir die Punktnotation mit Attributnamen verwenden, d.h. für Objekte $o \in WechselGeld$ legen wir fest:
 - ▶ $o.anz1er$ bezeichnet die Anzahl der 1er Münzen im Wechselgeld.
 - ▶ $o.anz2er$ bezeichnet die Anzahl der 2er Münzen im Wechselgeld.
 - ▶ $o.anz5er$ bezeichnet die Anzahl der 5er Scheine im Wechselgeld.
- ▶ Zumindest ist nun nicht mehr wichtig, wie das kart. Produkt von *WechselGeld* genau aufgebaut ist, denn wir sprechen die einzelnen Attribute nun über Namen an (statt deren Stelle im kart. Produkt).
- ▶ Warum löst das unsere Probleme von vorher nur äußerst bedingt???



Grenzen der bisherigen Konzepte

- ▶ Um die Probleme zu lösen, müssen wir die Komplexität des Datentyps komplett verbergen und vor ungewolltem Zugriff schützen (man spricht hier von *kapseln*).
- ▶ Der Programmierer muss nicht genau wissen, wie der Datentyp *WechselGeld* aufgebaut ist, solange er weiß, wie er ihn „bedienen“ muss (z.B. wie er auf die Anzahl der 5er zugreifen kann).
- ▶ Dieses „Bedienen“ funktioniert am besten über Methoden (Funktionen/Prozeduren), die mit ihrer Signatur und Funktionalität eine wohl-definierte *Schnittstelle* (*Interface*) bilden (und u.a. ausführlichst kommentiert werden können).
- ▶ D.h. sowohl das Lesen als auch das Verändern einzelner Attribute darf nur noch durch entspr. Methoden möglich sein.

Grenzen der bisherigen Konzepte

- ▶ Wir nennen die Definition eines eigenen Datentyps (hier *WechselGeld*) in Anlehnung an Java eine *Klassendefinition*.
- ▶ Diese enthält:
 - ▶ die Definition der Struktur von Objekten des Typs durch geeignete Attribute (die implizit ein kart. Produkt der Typen dieser Attribute definieren). Entsprechend dieser Struktur können Objekte diesen Typs unterschiedliche Zustände annehmen.
Die Attribute in unserem Beispiel *WechselGeld* sind $anz1er \in \mathbb{N}$, $anz2er \in \mathbb{N}$ und $anz5er \in \mathbb{N}$ und induzieren $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$.
 - ▶ Methoden, die auf wohl-definierte Weise auf den aktuellen Zustand eines Objekts zugreifen können, ihn verändern können, etc.
 - ▶ Methoden, die Objekte neu erzeugen können (sog. *Konstruktoren*), so ähnlich, wie wir es bereits bei Arrays gesehen haben.

Grenzen der bisherigen Konzepte

- Der entsprechende Datentyp *WechselGeld* in unserer „Pseudo“-Notation¹⁷:

```
class WechselGeld
  comment  Stellt den Datentyp WechselGeld bereit.
  attributes    // Die innere Struktur
    anz1er :  $\mathbb{N}$ ;
    anz2er :  $\mathbb{N}$ ;
    anz5er :  $\mathbb{N}$ ;
  constructors    // Die Konstruktoren (hier nur einer))
    WechselGeld(r :  $\mathbb{N}$ )
      comment  Erzeugt ein Wechselgeldobjekt für einen Rechnungsbetrag r.
      body    ...
  methods    // Die Methoden (Funktionen/Prozeduren)
    function  getAnz5er()  $\rightarrow \mathbb{N}_0$ 
      output  gibt die Anzahl der 5er aus.
      body    ...
    function  getAnz2er()  $\rightarrow \mathbb{N}_0$ 
      output  gibt die Anzahl der 2er aus.
      body    ...
    function  getAnz1er()  $\rightarrow \mathbb{N}_0$ 
      output  gibt die Anzahl der 1er aus.
      body    ...
```

¹⁷UML wäre eine akzeptable Variante; wir ersparen uns aber einen weiteren Formalismus.

Grenzen der bisherigen Konzepte

- Dies definiert einen Datentyp aus drei Attributen vom Typ $\mathbb{N}$, also eines 3-Tupels:

$$\textit{WechselGeld} \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

- Die einzelnen Komponenten (Attribute) des 3-Tupels werden intern *anz1er*, *anz2er* und *anz5er* genannt.
- Dabei legen wir fest, dass diese Attribute in **attributes** für andere Programmierer außerhalb dieser Klassendeklaration nicht sichtbar sind (Konstruktoren in **constructors** und Methoden in **methods** aber natürlich schon).
- D.h. Programmierer, die den Datentyp *WechselGeld* benutzen wollen, wissen nicht, dass es sich dabei um ein 3-Tupel handelt.

Grenzen der bisherigen Konzepte

- ▶ Programmierer können nun aber den Typ *WechselGeld* trotzdem wie (von den Arrays) gewohnt verwenden:
- ▶ Um Objekte (Literele) dieses Typs zu erzeugen, gibt es den Konstruktor *WechselGeld(r)* (dessen Rumpf wir nicht angegeben haben; hier sollte natürlich ein konkreter (Wechselgeld-)Algorithmus stehen¹⁸).
- ▶ Offensichtlich ist der Sinn dieses Konstruktors, für einen gegebenen Rechnungsbetrag r , ein entsprechendes Objekt (3-Tupel) zu erzeugen.
- ▶ Der Konstruktor belegt dabei die Attribute mit den entsprechenden Werten (z.B. für $r = 81$ hat das erzeugte Objekt den *Zustand* $\{(anz1er, 0), (anz2er, 2), (anz5er, 3)\}$).

¹⁸Der Konstruktor implementiert damit also den Wechselgeld-Algorithmus und ersetzt eine entsprechende Methode $wechselGeld : \mathbb{N} \rightarrow WechselGeld$



Grenzen der bisherigen Konzepte

- ▶ Um mit diesem Typ zu arbeiten, gibt es die Methoden *getAnz1er()*, *getAnz2er()* und *getAnz5er()* (auch deren Rümpfe sollten konkrete Algorithmen enthalten).
- ▶ Wichtig: auch diese Methoden sind Eigenschaften eines Objekts vom Typ *WechselGeld* und geben, je nach Zustand des Objekts, für das sie aufgerufen werden, ggfs. unterschiedliche Werte zurück.
Für das Objekt mit Zustand $\{(anz1er, 0), (anz2er, 2), (anz5er, 3)\}$ gibt *anz1er* z.B. 0 zurück.
- ▶ Außer diesen Methoden gibt es keine Möglichkeit, den Zustand eines Objekts abzufragen oder zu verändern, was offensichtlich gewünscht ist!

Vom primitiven Typ zur Klasse: Klassen

- Diese Klassendefinition *WechselGeld* würde in Java so aussehen:

```
public class WechselGeld {  
  
    private int anzEiner;  
    private int anzZweier;  
    private int anzFuenfer;  
  
    public WechselGeld(int r) { ... }  
  
    public int getAnzEiner() { ... }  
    public int getAnzZweier() { ... }  
    public int getAnzFuenfer() { ... }  
}
```

- Wie gesagt, um die Methodenrümpfe kümmern wir uns später ...
- Und wie gesagt, alles sollte ausführlichst mit JavaDoc dokumentiert werden ...

Vom primitiven Typ zur Klasse: Klassen

- Der Programmierer eines Wechselgeldautomaten kann diese Klasse nun verwenden um ein oder mehrere Wechselgeld-Objekte zu erzeugen (der Automat könnte vielleicht parallel 3 verschiedene Ein-Ausgabe-Einheiten/Displays haben, jeweils mit Rechnungsbetrag `r1`, `r2` und `r3`):

```
WechselGeld g1 = new WechselGeld(r1);
```

```
WechselGeld g2 = new WechselGeld(r2);
```

```
WechselGeld g3 = new WechselGeld(r3);
```

- Um die Anzahl der 5er in `g2` zu erhalten, kann man nun auf dieses Objekt mit Punktnotation die Methode `getAnzFuenfer` anwenden, die einen Wert aus `int` zurück gibt:

```
int f2 = g2.getAnzFuenfer();
```



Vom primitiven Typ zur Klasse: Klassen

- Beispiel: Nehmen wir an $r1=96$, $r2=81$ und $r3=14$:
Offenbar gilt:

$g1 = (0, 2, 0)$

d.h.

`g1.getAnzEiner()` hat den Wert 0
`g1.getAnzZweier()` hat den Wert 2
`g1.getAnzFuenfer()` hat den Wert 0

$g2 = (0, 2, 3)$

`g2.getAnzEiner()` hat den Wert 0
`g2.getAnzZweier()` hat den Wert 2
`g2.getAnzFuenfer()` hat den Wert 3

$g3 = (1, 0, 17)$

`g3.getAnzEiner()` hat den Wert 1
`g3.getAnzZweier()` hat den Wert 0
`g3.getAnzFuenfer()` hat den Wert 17

Vom primitiven Typ zur Klasse: Klassen

- ▶ Das ist perfekt: der Geldautomat kann nun ohne Wissen, wie das Wechselgeld strukturiert ist, damit hantieren:

- ▶ Er könnte den Wechselgeldbetrag anzeigen:

```
int betrag2 = g2.getAnzEiner() + g2.getAnzZweier() +  
g2.getAnzFuenfer();
```

- ▶ Er könnte ein Array mit 1ern, 2ern und 5ern erzeugen, denn:

```
q1 = g2.getAnzFuenfer();  
q2 = g2.getAnzFuenfer();  
r2 = g2.getAnzFuenfer();
```

- ▶ Er könnte ein Array der Länge 3 erzeugen wie in der Variante im vorherigen Kapitel.
 - ▶ usw.

Vom primitiven Typ zur Klasse: Klassen

Nochmal: was war der Clou?

- ▶ Wir haben einen neuen, komplex-strukturierten Datentypen deklariert.
- ▶ Dabei haben wir dessen Struktur (allgemein ein n -Tupel über typischerweise verschiedene Typen T_i) vor den Benutzern verborgen (*Kapselung*).
- ▶ Wir haben lediglich eine Schnittstelle definiert: Konstruktoren zur Erzeugung neuer Objekte (Literals) dieses Typs und Methoden zum Zugriff und zur Manipulation des Zustands von Objekten (Literalen) dieses Typs.

Der erste Schritt zur objektorientierten Programmierung ist getan ...



Überblick

8. Grundlagen der objektorientierten Programmierung

8.1 Abstrakte Datentypen I: Benutzereigene Strukturen

8.2 Abstrakte Datentypen II: von Structures zu Klassen

8.3 Das objektorientierte Paradigma

8.4 Klassen und Objekte in Java

8.5 Vererbung, abstrakte Klassen, Polymorphismus

8.6 Interfaces

8.7 Speicherverwaltung in Java

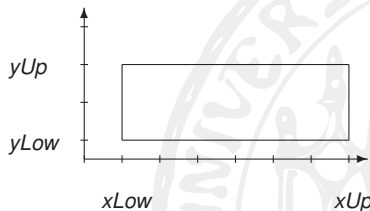
Grenzen der bisherigen Konzepte

- ▶ Wir führen die Diskussion noch einmal, aber mit einem neuen Beispiel.
- ▶ Dabei betrachten wir auch die historische Entwicklung der Programmier-Paradigmen: vom reinen imperativen Paradigma zur Objektorientierung.
- ▶ Dies schauen wir uns daher gleich „in Java“ an.
- ▶ Wir wollen dazu ein Rechteck im 2D Raum darstellen/verarbeiten (bspw. wollen wir das Rechteck verschieben oder berechnen, ob ein Punkt in diesem Rechteck liegt, etc.).



Grenzen der bisherigen Konzepte

- Ein Rechteck kann durch vier Zahlen $xLow$, xUp , $yLow$ und yUp vom Typ `double` repräsentiert werden:



- Die Zahlen definieren den unteren linken und den oberen rechten Eckpunkt.



Grenzen der bisherigen Konzepte

- Eine Methode `punktInRechteck`, die prüft ob ein Punkt, gegeben durch zwei Koordinaten ebenfalls vom Typ `double`, in einem Rechteck enthalten ist, könnte entsprechend so definiert sein:

```
public static boolean punktInRechteck(double xCoord, double yCoord,  
    double xLow, double xUp, double yLow, double yUp)  
{  
    return ( (xLow <= xCoord && xCoord <= xUp) &&  
        (yLow <= yCoord && yCoord <= yUp));  
}
```



Grenzen der bisherigen Konzepte

► Problem?

- Die einzelnen Eingabeparameter der Methode spiegeln die semantische Beziehung nicht wider (vier Werte definieren das Rechteck, zwei den Punkt).
- Wenn die Dokumentation schlecht ist, oder fehlt, ist die Gefahr hoch, dass die Parameter falsch benutzt werden.
- Im schlimmsten Fall ist die Signatur der Methode sogar so implementiert:

```
public static boolean punktInRechteck(double x1, double x2,  
    double x3, double y1, double y2, double y3)
```

- Dann wird es sehr wahrscheinlich, dass diese Methode falsch verwendet wird.



Grenzen der bisherigen Konzepte

- ▶ Eine andere Möglichkeit wäre, wenn man die Koordinaten des Rechtecks in ein Array packt:

```
public static boolean punktInRechteck(double xCoord, double yCoord,  
    double[] xCoords, double[] yCoords)  
{  
    return ( (xCoords[0] <= xCoord && xCoord <= xCoords[1]) &&  
        (yCoords[0] <= yCoord && yCoord <= yCoords[1]));  
}
```

- ▶ Dabei ist der Aufbau der Eingabe- Arrays implizit als korrekt angenommen:
 - ▶ Die beiden Arrays müssen (mind.) die Länge 2 haben.
 - ▶ Die beiden Arrays müssen richtig gefüllt sein, d.h. in `xCoords` muss an Stelle 0 der Wert *xLow* und an Stelle 1 der Wert *xUp* stehen (analog in `yCoords`).

Grenzen der bisherigen Konzepte

- ▶ Nun besteht zwar immerhin (implizit) ein semantischer Zusammenhang zwischen den einzelnen x -Koordinaten des Rechtecks und zwischen den einzelnen y -Koordinaten des Rechtecks.
- ▶ Einen Zusammenhang zwischen allen vier Werten des Rechtecks gibt es aber immer noch nicht.
- ▶ Natürlich könnte man dazu ein 2-dimensionales Array als Eingabe verlangen (statt zwei 1D Arrays), dies würde die Komplexität des Methodenaufrufs aber nur noch vergrößern.
- ▶ Der Aufrufer muss die Struktur des Eingabe-Arrays verstehen und richtig abbilden (insbesondere die Stelligkeit beachten).
- ▶ Grundsätzlich ist die Handhabung, dem Aufrufer einer Methode diese Komplexität und Spezifika der Datendarstellung aufzubürden, nicht akzeptabel.

Selbst definierte Datentypen

- ▶ Lösung dieser gesamten Problematik ist natürlich, dass man zusätzlich zu den primitiven Datentypen (und den Arrays) noch einen eigenen *selbst definierten Datentypen* bauen und benutzen könnte.
- ▶ Wie bereits erwähnt: dieser Typ sollte die Komplexität des Aufbaus vor dem Nutzer verbergen.
- ▶ Ein selbst definierter Datentyp ist dem Compiler standardmäßig *nicht* bekannt.
- ▶ Wenn die Programmiersprache, die man benutzt, entsprechende Sprachmittel zur Verfügung stellt, kann der Programmierer eigene Datentypen (die für die Modellierung einer Anwendung von Bedeutung sind – z.B. einen Datentyp *Wechselgeld*) erfinden und diesen dem Compiler bekannt machen.
- ▶ Java bietet dafür das Sprachkonstrukt der Klasse (`class`) an.

Klassen

- ▶ Eine *Klasse* bildet ein Objekt der realen Welt in ein Schema ab, das der Compiler versteht (d.h. macht einen neuen Datentyp bekannt).
- ▶ *Objekte* können prinzipiell beliebige Gegenstände der realen Welt sein (der für einen Menschen eine Bedeutung hat und der sprachlich beschrieben werden kann).
- ▶ Objekte haben genau die Eigenschaften, die im Schema der Klasse definiert sind, sie sind *Instanzen* der Klasse (die Literale des neuen Typs).
- ▶ In Java gibt es eine große Anzahl vordefinierter Datentypen, die in der Java API als *Bibliotheksklassen* zur Verfügung gestellt werden.
- ▶ Ein selbst definierter Datentyp kann also durch eine Bibliothek zur Verfügung gestellt werden oder von einem Programmierer eingeführt werden.

Vom primitiven Typ zur Klasse

- ▶ Wir diskutieren die Entwicklung vom einfachen Datentyp zur Klasse in den unterschiedlichen Programmiersprachen am Beispiel des *Objekts* Rechteck.
- ▶ Wie wir bereits diskutiert haben, kann ein Rechteck durch vier `double`-Werte, bzw. Variablen beschrieben werden:

```
double xLow;
```

```
double xUp;
```

```
double yLow;
```

```
double yUp;
```

- ▶ Problem ist eben zunächst nur, dass der Benutzer (Programmierer) sich merken muss, dass diese Variablen alle zum selben Rechteck gehören.

Vom primitiven Typ zur Klasse: Tupel/Structures

- ▶ Eine der ersten Programmiersprachen, die darauf eine Antwort hatte, war Nikolaus Wirths *Pascal*.
- ▶ Er führte für die Sprache die Möglichkeit ein, *zusammengesetzte Datentypen* zu vereinbaren.
- ▶ Diese Typen sind aus Komponentenvariablen aufgebaut (die ihrerseits einen beliebigen primitiven Typ haben können).
- ▶ Formal ist dieser zusammengesetzte Typ, bestehend aus n Komponenten von primitiven (nicht notwendigerweise verschiedenen) Typen $T_1, \dots, T_n$ ein n -Tupel, d.h. ein Element aus $T_1 \times \dots \times T_n$.
- ▶ Nikolaus Wirth nannte diesen zusammengesetzten Typ daher auch *Record*, eine englische Bezeichnung für *Tupel*.

Vom primitiven Typ zur Klasse: Tupel/Structures

- ▶ Kernighan und Ritchie wollten den Begriff Record für die Sprache C nicht übernehmen, wohl aber den zusammengesetzten Datentyp, den sie *Struktur* (engl. *Structure*, Schlüsselwort `struct`) nannten.
- ▶ Ein neuer (zusammengesetzter) Datentyp `Rechteck` kann in C wie folgt vereinbart werden:

```
struct Rechteck
{
    double xLow;
    double xUp;
    double yLow;
    double yUp;
}
```

- ▶ Dieser Typ enthält die Komponenten `xLow`, `xUp`, `yLow` und `yUp`, jeweils vom Typ `double`.

Vom primitiven Typ zur Klasse: Tupel/Structures

- Der neue Typ `Rechteck` kann in C nun als Typ für Variablen verwendet werden:

```
struct Rechteck r;
```

- `r` ist eine *Strukturvariable* mit den vier Komponenten `xLow`, `xUp`, `yLow` und `yUp`, deren Referenzierung über Punktnotation erfolgt.
- Eine Festlegung der Werte der Komponenten kann durch Zuweisungen erfolgen:

```
r.xLow = 1.0;
```

```
r.xUp = 7.0;
```

```
r.yLow = 1.0;
```

```
r.yUp = 3.0;
```

Vom primitiven Typ zur Klasse: Tupel/Structures

- ▶ Was war der Fortschritt dieser Neuerung:
Es gab erstmals ein Sprachmittel, ein Objekt, wie z.B. ein Rechteck, über seine *Eigenschaften* explizit als (semantisch zusammenhängendes) Ganzes zu beschreiben.
- ▶ Der Zugriff auf die Komponenten erfolgt im Programm durch die Punktnotation, die ab dann zum Standard in vielen Programmiersprachen wurde.

Vom primitiven Typ zur Klasse: Tupel/Structures

- ▶ Natürlich kann man Variablen vom Typ **struct** Rechteck mit Funktionen bearbeiten.
- ▶ Diese Funktionen spezifizieren gewissermaßen das *Verhalten* dieses Typs.
- ▶ In C (wie in Pascal) werden diese Funktionen allerdings außerhalb der Struktur definiert, also quasi als (ganz normale) statische Operationen, die man zusammen mit der Strukturdefinition natürlich in ein gemeinsames Modul packen kann.

Vom primitiven Typ zur Klasse: Tupel/Structures

► Beispiel: die altbekannte Funktion

```
int punkt_in_rechteck(struct Rechteck r, double xCoord, double yCoord)
{
    return ( (r.xLow <= xCoord && xCoord <= r.xUp) &&
             (r.yLow <= yCoord && yCoord <= r.yUp));
}
```

(Sie sehen nebenbei, dass die grundsätzliche Syntax von C (und C++) der von Java sehr ähnelt, allerdings gibt es keinen Typ `boolean` (Pfui), logische Ausdrücke haben stattdessen den Typ `int` (Nochmehr Pfui!!!))

- Neben der ganzen Sauerei mit dem fehlenden Typ `boolean` ist aber eines ganz **wichtig**:
 - Da diese Funktion außerhalb der Struktur (als Pendant zur statischen Methode in Java) definiert werden muss, kann sie natürlich nicht wissen, auf welchem Rechteck sie arbeitet.
 - Das zu bearbeitende Rechteck muss in der Parameterliste übergeben werden (naja, so kennen wir das ja auch?!?!).

Vom primitiven Typ zur Klasse: Tupel/Structures

- ▶ Wie gesagt, diese Eigenschaft ist zunächst nicht störend.
- ▶ Bei folgender Funktion (Zugriff auf die Komponente `xLow`) ist es aber schon etwas komischer:

```
{  
    return r.xLow;  
}
```

- ▶ Der Sinn dieser Funktion soll zunächst nicht diskutiert werden.
- ▶ Es ist aber sofort zu sehen, dass die Ausdrücke `r.xLow` und `get_xLow(r)` den selben Wert liefern.
- ▶ Offenbar wäre es konsequent, nicht nur die Eigenschaften (Komponenten) der Elemente (Objekte) eines Strukturtyps innerhalb der Struktur zu vereinbaren, sondern auch deren Verhalten (Operationen)

Vom primitiven Typ zur Klasse: Tupel/Structures

- Diesen Fortschritt brachte *C mit Klassen* (später *C++* genannt).
- Eine Struktur durfte nun nicht nur Komponenten (Daten) enthalten, sondern auch Operationen.

```
struct Rechteck
{
    double xLow;
    double xUp;
    double yLow;
    double yUp;

    double get_xLow() {
        return xLow;
    }

    double get_xUp() { return xUp; }

    double get_yLow() { return yLow; }

    double get_yUp() { return yUp; }

    int punkt_in_rechteck(double xCoord, double yCoord) { ... }
}
```



Vom primitiven Typ zur Klasse: Tupel/Structures

- ▶ Die Funktionen der Struktur (z.B. `get_xLow` haben nun keinen Übergabeparameter vom Typ `struct Rechteck` mehr nötig.
- ▶ Die Methoden sind nämlich nun innerhalb der Struktur definiert (das ist neu und typisch für die oo-Programmierung) und haben damit automatisch Zugriff auf die Komponenten.
- ▶ Eine Variable `r` vom Typ `struct Rechteck` kann wie gewohnt vereinbart werden:

```
struct Rechteck r;
```

Vom primitiven Typ zur Klasse: Tupel/Structures

- ▶ In C++ repräsentiert diese Variablen x nun ein *Objekt*.
- ▶ Dieses Objekt hat als Komponenten:
 - ▶ die Komponentenvariablen (*Daten*): `xLow`, `xUp`, `yLow` und `yUp`.
 - ▶ die Funktionen (*Funktionalitäten*, die das *Verhalten* spezifizieren):
`get_xLow`, `get_xUp`, `get_yLow`, `get_yUp` und `punkt_in_rechteck`.

Vom primitiven Typ zur Klasse: Tupel/Structures

- ▶ Was haben wir getan?
- ▶ Wir haben die Möglichkeit eingeführt, einen neuen, selbst definierten Datentyp zu definieren, indem spezifiziert wurde, welche Eigenschaften dieser Datentyp hat:
 - ▶ Einerseits sind das die Daten-Komponenten (auch *Attribute* oder *Objekt-(Member-)Variablen*).
 - ▶ Andererseits sind das die Funktions-Komponenten (auch *Methoden*), die quasi einen Satz an Grundoperationen für diesen Datentyp definieren.
- ▶ Die Funktions-Komponenten haben dabei Zugriff auf die Datenfelder.
- ▶ Alle Komponenten werden über die Punktnotation abgerufen, z.B.

```
struct Rechteck r;  
double value = r.xLow;  
int test = r.punkt_in_rechteck(2.0,3.0);
```



Vom primitiven Typ zur Klasse: Klassen

- ▶ Schreibt man statt `struct` nun `class` ist man bei den *Klassen* angekommen.
- ▶ Der Unterschied zwischen Klassen und Strukturen in C++ soll hier nicht diskutiert werden (er besteht i.W. in unterschiedlichen Default-Sichtbarkeiten der versch. Komponenten von außen).
- ▶ Java kennt nur das Sprachkonstrukt der Klasse.

Vom primitiven Typ zur Klasse: Klassen

► Die Vereinbarung einer Klasse

```
class Rechteck {  
    double xLow;  
    double xUp;  
    double yLow;  
    double yUp;  
  
    double getXLow() { return xLow; }  
    ...  
}
```

macht den Datentyp `Rechteck` dem Compiler bekannt.

- Bemerkung: Der Hauptunterschied zwischen Klassen in C++ und Java besteht bei der Bildung von Variablen (wird hier aber nicht diskutiert).

Klassen

- ▶ Eine Klassendefinition ist also die Vereinbarung eines neuen (eigenen) Typs (den wir zunächst genau wie die primitiven Typen verwenden dürfen, also z.B. eine Variable diesen Typs vereinbaren).
- ▶ Objekte (auch *Instanzen*) dieser Klasse sind so etwas wie die Literale (Elemente) dieses Typs.
- ▶ Die Datentypen bestehen aus Datenkomponenten und Funktions-Komponenten.



Klassen

- ▶ Die Datenkomponenten sind eine Art Blaupause und beschreiben die Struktur der Objekte, sie heißen daher auch *Objekt-* oder *Instanzvariablen*.
- ▶ Die Funktions-Komponenten (*Objekt-* bzw. *Instanz-Methoden*) spezifizieren das Verhalten von Objekten.
- ▶ Jedes Objekt der Klasse besitzt diese Struktur (alle Objektvariablen) und das Verhalten (alle Objektmethoden).

Statische und nicht-statische Elemente

- ▶ In der Klassendefinition `Rechteck` (wie übrigens auch in `WechselGeld`) ist Ihnen vielleicht aufgefallen, dass nun das Schlüsselwort **static** bei den Variablen und Methoden fehlt.
- ▶ Das macht auch nochmal den Unterschied zu unseren bisherigen Konzepten deutlich.
- ▶ Klassen sind in Java hauptsächlich dazu da, neue (eigene) Datentypen zu vereinbaren, deren Eigenschaften durch die Objektvariablen und Objektmethoden definiert werden.

Statische und nicht-statische Elemente

- ▶ Diese Objekt-Komponenten sind Eigenschaften konkreter Objekte der Klasse und machen damit auch nur im Kontext eines konkreten Objekts Sinn.
- ▶ Sie können auch nur für konkrete Instanzen der Klasse aufgerufen werden.
- ▶ Die statischen Elemente einer Klassendefinition sind dagegen (i.Ü. ganz im Sinne unseres Modulkonzepts) nicht an die Existenz eines Objekts gebunden, sondern unabhängig von der Existenz einzelner Instanzen immer verfügbar.
- ▶ Sie werden daher nicht für konkrete Instanzen der Klasse, sondern „für die Klasse selbst“ aufgerufen.

Statische und nicht-statische Elemente

- ▶ Dieser Unterschied zeigt sich auch in der Namensgebung der unterschiedlichen Elemente.
- ▶ Eine statische Methode `statischeMethode(...)` der Klasse `K` hat den (abgesehen vom Packagenamen vollständigen) Namen `K.statischeMethode(...)`, mit dem sie aufgerufen werden kann.
- ▶ Eine Objektmethode `objektMethode(...)` dieser Klasse macht nur für ein konkretes Objekt Sinn, d.h. dazu muss es zunächst eine (lokale) Variable `v` vom Typ `Klasse` geben, die ein konkretes Objekt repräsentiert; die Methode kann dann mit der Punktnotation aufgerufen werden:
`v.objektMethode(...)`.