Abschnitt 6: Grundlagen der objektorientierten Programmierung

6. Grundlagen der objektorientierten Programmierung
6.1 Abstrakte Datentypen: von Structures zu Klassen
6.2 Das objektorientierte Paradigma

Überblick

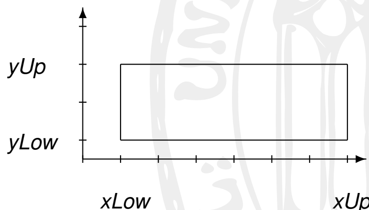
6. Grundlagen der objektorientierten Programmierung
6.1 Abstrakte Datentypen: von Structures zu Klassen
6.2 Das objektorientierte Paradigma

Grenzen der bisherigen Konzepte

- Bei der Darstellung der Daten, die wir verarbeiten wollten, sind wir teilweise auf Grenzen gestoßen
- Bei der Darstellung des Wechselgelds haben wir verschiedene Möglichkeiten der Datendarstellung (Arrays, globale Größen) versucht; keine der Lösungen war frei von massiven Nachteilen
- Woran lag das?
- Das Wechselgeld bestand aus einer Menge von unterschiedlichen Teilen (1 EUR, 2 EUR, 5 EUR Entitäten, 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

Grenzen der bisherigen Konzepte

- Wir führen die Diskussion weiter, mit einem neuen Beispiel
- Wir wollen ein Rechteck im 2D Raum darstellen/verarbeiten (bspw. wollen wir das Rechteck verschieben oder berechnen, ob ein Punkt in diesem Rechteck liegt, etc.)
- Ein Rechteck kann durch vier Zahlen $xLow$, xUp, $yLow$ und yUp vom Typ **`float`** 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 **`float`**, 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 mandazu 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
- Grundsätzlich ist die Handhabung, dem Aufrufer einer Methode diese Komplexität der Eingabe aufzubürden, nicht akzeptabel

Selbst definierte Datentypen

- Lösung dieser gesamten Problematik wäre, wenn man zusätzlich zu den primitiven Datentypen (und den Arrays) noch einen eigenen *selbst definierten Datentypen* bauen und benutzen könnte
- 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 für selbst definierte Datentypen das Sprachkonstrukt der Klasse (**`class`**) oder des Aufzählungstyps (**`enum`**) an

LMU

Klassen

- Eine *Klasse* bildet ein Objekt der realen Welt in ein Schema ab, das der Compiler versteht
- *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
- 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
- Wir 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 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) Typ $T_1, \ldots, T_n$ ein n-Tupel, d.h. ein Element aus $T_1 \times \ldots \times T_n$
- 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`
- 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

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 statische Operationen, die man zusammen mit der Strukturdefinition natürlich in ein gemeinsames Modul packen kann

LMU

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 rundsätzliche 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

```
double get_xLow(struct Rechteck r)
{
    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)

LMU

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;
```

- In C++ repräsentiert diese Variablen 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
- Dazu muss spezifiziert werden, 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 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
- 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.

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
- 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` 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
- Während diese Objekt-Komponenten Eigenschaften konkreter Objekte der Klasse sind (und damit auch nur im Kontext eines konkreten Objekts Sinn machen), sind die statischen Elemente einer Klassendefinition (i.Ü. ganz im Sinne unseres Modulkonzepts) nicht an die Existenz eines Objekts gebunden, sondern unabhängig immer verfügbar

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 aufegrufen werden kann
- Eine Objektmethode `objektMethode(...)` dieser Klasse macht nur für ein konkretes Objekt Sinn, d.h. dazu muss es zunächst eine (lokale) Variable `var` vom Typ `Klasse` geben, die ein konkretes Objekt repräsentiert; die Methode kann dann mit der Punktnotation aufgerufen werden: `var.objektMethode(...)`