Einführung in die Programmierung

Teil 6: Grundlagen der oo Programmierung

Prof. Dr. Peer Kröger,
Florian Richter, Michael Fromm

Wintersemester 2018/2019

Übersicht

Kapitel 1: Klassen i

Grenzen der bisherigen Konzepte

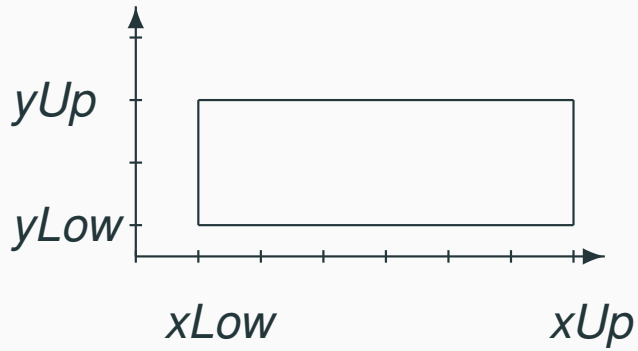
- Im vorgherigen Teil haben wir grundlegende Datenstrukturen kennengelernt, die es uns ermöglicht haben, Mengen von Objekten zu verarbeiten.
- Arrays und Tupel waren jeweils Elemente von Kreuzprodukten über dieselbe Grundmenge bzw. über beliebige Grundmengen.
- Speziell die Tupel erlauben die Definition von beliebig strukturierten Datentypen.

Grenzen der bisherigen Konzepte

- Wie erwähnt sind diese beiden Strukturen in den meisten höheren Programmiersprachen auch vorhanden.
- Dennoch haben diese Konzepte einige entscheidenden Schwächen, die wir uns jetzt genauer an einem Beispiel anschauen (und die u.a. zur Entwicklung der oo Programmierung führten).
- 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.
- Alternativ lässt sich das Rechteck natürlich auch mit anderen vier Zahlen darstellen, z.B. *xLow*, *yLow* (linker unterer Eckpunkt) und $xUp - xLow$, $yUp - yLow$ (Ausdehnungen in x bzw. y Richtung).

Grenzen der bisherigen Konzepte

- Aus diesen Überlegungen kann man sehr einfach einen Record-Datentyp für Rechtecke definieren.
- Wir wollen den Datentyp in einer eigenen Datei definieren und "der Welt" zur Benutzung zur Verfügung stellen:

```
public class Rechteck {
  public double xLow;
  public double xUp;
  public double yLow;
  public double yUp;
}
```

- Dazu könnten wir noch ein paar statische Methoden definieren — müssen wir aber nicht, wir können auch so Variablen vereinbaren und mit diesem Datentyp "arbeiten".

Grenzen der bisherigen Konzepte

- Problem?
- Alle Programmierer, die den Datentyp `Rechteck` verwenden wollen, müssen die *innere Struktur* dieses Datentyps kennen und verstehen (insbesondere. welche Instanzvariablen was bedeuten, um sie korrekt zu verwenden, d.h. zu verändern bzw. abzurufen).
- Wenn Sie ein paar Wochen nicht damit gearbeitet haben, werden Sie sich auch nicht mehr genau daran erinnern.
- Gut ist hier vielleicht nicht so schwer aber im Prinzip: *Wiederverwendbarkeit* gleich null, dann kann ich den Datentyp gleich selbst (nochmal) programmieren ...

Grenzen der bisherigen Konzepte

- Um z.B. eine (statische) Methode `punktInRechteck` zu implementieren, die prüft ob ein Punkt, gegeben durch zwei Koordinaten vom Typ **`double`**, in einem Rechteck enthalten ist, müssen wir die Details der Klasse `Rechteck` verstehen:

```
public static boolean punktInRechteck
     (double x, double y, Rechteck r) {
      return ( (r.xLow <= x && x <= r.xUp) &&
          (r.yLow <= y && y <= r.yUp));
}
```

- Hätten wir einen alternativen Modellierungsansatz für Rechtecke gewählt, wäre diese Methode “falsch”!
- Falls die Details nicht dokumentiert sind, bleibt uns nichts anderes übrig, als den Quell-Code zu lesen ...

Abstrakte Datentypen

- Es wäre natürlich schön, wenn wir die Komplexität des Aufbaus einzelnen Datentypen vor dem Nutzer im Sinne eines *abstrakten Datentyps* verbergen könnten.
- Dieses Prinzip nennt man *information hiding* oder im Deutschen auch *Kapseln* und ist eines der zentralen Ideen des oo Paradigmas.
- Es fördert die Wiederverwendbarkeit von Programmteilen, ohne deren Details genau verstehen zu müssen.
- Wir nähern uns dieser “neuen” Art der Klassen, indem wir uns die historische Entwicklung vom einfachen Record-Datentyp zur Klasse in den unterschiedlichen Programmiersprachen am Beispiel des *Objekts* Rechteck anschauen.

Vom primitiven Typ zur Klasse

- Wie wir bereits diskutiert haben, kann ein Rechteck durch vier **`double`**-Werte, bzw. Variablen beschrieben werden:

```
double xLow;
double xUp;
double yLow;
double yUp;
```

- Bevor es Record-Typen gab, mussten diese Werte durch einzelne Variablen verwaltet werden ohne den semantischen Zusammenhang explizit darzustellen (Sie erinnern sich an unseren aller ersten Kalender ...).

Vom primitiven Typ zur Klasse: Tupel/Structures

- Eine der ersten Programmiersprachen, die Record-Typen erlaubte, war Nikolaus Wirths *Pascal*.
- Er führte für die Sprache die Möglichkeit ein, *zusammengesetzte Datentypen* zu vereinbaren.
- Diese Typen sind entsprechend aus Komponentenvariablen aufgebaut, die allerdings auf primitive Typen eingeschränkt sind (diese Einschränkung besteht in Java nicht, wir haben bereits Arrays über Strings kennengelernt und es gibt keinen Grund, nicht auch für Instanzvariablen von Records Referenztypen zuzulassen).
- Nikolaus Wirth nannte diesen zusammengesetzten Typ *Record*, eine englische Bezeichnung für *Tupel* (macht Sinn).

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 Struktur-(/Record-)Variable mit den vier Komponenten `xLow`, `xUp`, `yLow` und `yUp`, deren Referenzierung wie in Java über die Punktnotation (die zum Standard in vielen Sprachen wurde) erfolgt.
- Eine Festlegung der Werte der Komponenten kann ebenfalls wie in Java 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

- Natürlich kann man Variablen vom Typ `struct Rechteck` mit Funktionen bearbeiten.
- Diese Funktionen spezifizieren gewissermaßen das *Verhalten* dieses Typs.
- In C (wie in Pascal) und auch bisher bei uns werden diese Funktionen allerdings “außerhalb” der Struktur definiert, als statische Operationen, die man zusammen natürlich in ein gemeinsames Modul packen kann.
- Einzige Neuerung in Java bisher: wir dürfen auch Instanzvariablen verwenden, die nicht einen primitiven Typ sondern einen Referenz-Typ (also z.B. einen anderen Record-Typ) haben.

Vom primitiven Typ zur Klasse: Tupel/Structures

- Die altbekannte Funktion in C:

```
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` (**PFUI!!!**))

Vom primitiven Typ zur Klasse: Tupel/Structures

- 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?!?!).
  - Die Problematik ist natürlich auch hier, dass wir, um diese Methode zu schreiben, die Details der Struktur kennen und verstehen müssen.

Vom primitiven Typ zur Klasse: Tupel/Structures

- Nehmen wir zunächst an, dass dies nicht extrem störend ist, d.h. auch folgende Funktion (Zugriff auf die Komponente `xLow`) ist für uns natürlich einfach:

```
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).

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 (Ähnliches gilt für `punkt_in_rechteck`, auch hier wird kein Rechteck mehr übergeben).
- 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.
- Und fast noch wichtiger: sie sind nun Teil der Eigenschaften der konkreten Objekte (Literale).

Vom primitiven Typ zur Klasse: Tupel/Structures

- Eine Variable `r` vom Typ **`struct`** `Rechteck` kann wie gewohnt vereinbart werden: **`struct`** `Rechteck r;`
- In C++ repräsentiert diese Variablen `r` nun ein *Objekt*.
- Dieses Objekt hat als Komponenten:
  - die *Komponentenvariablen* (auch *Instanzvarialen*, *Objektvariablen*, also die *Daten* bzw. die Werte, die den *Zustand* des gesamten Objekts ausmachen): `xLow`, `xUp`, `yLow` und `yUp`.
  - die *Funktionen* (auch (*Objekt-*) *Methoden*, also die *Funktionalitäten*, die das *Verhalten* des gesamten Objekts spezifizieren):
    `get_xLow`, `get_xUp`, `get_yLow`, `get_yUp` und `punkt_in_rechteck`.

Vom primitiven Typ zur Klasse: Tupel/Structures

- Mit Objekt-Funktionen ist der Schritt zum abstrakten Datentyp nicht weit:
- Wenn wir die Objekt-Variablen und die Objekt-Funktionen so definieren, dass die Funktionen die einzige Möglichkeit bilden, auf die Objektvariablen zuzugreifen bzw. sie zu verändern, haben wir unser Ziel erreicht:
  - Wir müssen dann nicht mehr die Struktur der Objekte verstehen, um ihren Zustand abzulesen bzw. zu verändern
  - Wir müssen nur wissen, wie wir die Funktionen(Methoden) anwenden (aufrufen) müssen.
  - Die Funktionen/Methoden bilden damit eine Abstraktionsebene, indem sie eine Schnittstelle für die Verwendung der Objekte darstellen.

Vom primitiven Typ zur Klasse: Tupel/Structures

- Das ist übrigens im Prinzip sehr ähnlich zu den primitiven Typen [1].
- Es interessiert uns nicht, wie der Computer die Werte des Typs `int` darstellt.
- Es interessiert uns auch nicht, wie Grundoperationen (z.B. die Addition) implementiert sind.
- Die Grundoperationen funktionieren (hoffentlich) und wir wissen, wie wir sie verwenden können und was sie bewirken, das ist das einzige was zählt.

[1] Allerdings wirklich nur vom Prinzip, also Vorsicht!

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

Vom primitiven Typ zur Klasse: Klassen

- Die Vereinbarung einer Klasse

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

      double getXLow() { return xLow; }
      ... // weitere Objekt-Methoden
}
```

macht den Datentyp `Rechteck` wie bisher dem Compiler bekannt.

Klassen

- Was wir im Prinzip schon wissen:
  - Eine *Klasse* bildet ein Objekt der realen Welt in ein Schema (definiert durch die Instanzvariablen) ab, das der Compiler versteht (d.h. macht einen neuen Datentyp bekannt).
  - Objekte haben genau die Eigenschaften, die im Schema der Klasse definiert sind, sie sind *Instanzen* der Klasse (die Literale des neuen Typs).
- Neu ist: Objekte haben nun auch Methoden, die ihr Verhalten spezifizieren und die Objekte “benutzbar” machen.
- In Java gibt es eine große Anzahl vordefinierter Datentypen, die in der Java API als *Bibliotheksklassen* zur Verfügung gestellt werden.

Klassen

- Eine Klassendefinition kann also mehr als ein simpler Record-Typ sein: Die Datentypen können nun aus Datenkomponenten (Instanzvariablen) und Funktions-Komponenten (instanz-Methoden) bestehen.
- Die Datenkomponenten sind eine Art Blaupause und beschreiben die Struktur der Objekte (“Literale”) dieses Typs.
- Die Funktions-Komponenten spezifizieren das Verhalten von Objekten.
- Jedes Objekt der Klasse besitzt diese Struktur (alle Objektvariablen) und das Verhalten (alle Objektmethoden).
- Was wir noch nicht gemacht haben: die Objektvariablen sauber zu kapseln, aber das kommt bald.

Statische und nicht-statische Elemente

- In der Klassendefinition `Rechteck` ist Ihnen vielleicht aufgefallen, dass nun das Schlüsselwort **`static`** bei den Variablen und insbesondere den Methoden komplett fehlt.
- Das macht auch nochmal den Unterschied zu unseren ursprünglichen rein imperativen 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 (mit Punktnotation).
- Die statischen Elemente einer Klassendefinition sind dagegen (i.Ü. ganz im Sinne unseres Modulkonzepts) nicht an die Existenz eines konkreten 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(...)`.

Klassen, Klassen, Klassen

- Die erste bedeutende Errungenschaft der oo Programmierung ist also die Möglichkeit, eigene Datentypen durch Klassen zu spezifizieren.
- Dies ist mit einigen weiteren Aspekten und Feinheiten verbunden, von denen wir nachfolgend einige diskutieren wollen.
- Neben der neuen Möglichkeit zur Datendarstellung bzw. Modellierung von Daten zu deren Verarbeitung gilt:
  Die eigentliche Verarbeitung geschieht in den (Klassen- und Objekt-)Methoden: hier werden die eigentlichen Algorithmen implementiert, bei Java mit Hilfe des imperativen Paradigmas.
- Aus dieser Sicht ist oo also v.a. ein Modellierungs-Paradigma.

Objekte, Objekte, Objekte

- Die zweite bedeutende Errungenschaft der oo Programmierung ist, dass man das Programmieren nun auf ein neues Abstraktions-Level hievt.
- Statt komplexe Algorithmen zu schreiben, die einzelne Teilaufgaben ggfs. in Funktionen/Prozeduren erledigen (die aufgerufen werden), besteht ein oo Programm v.a. aus Objekten (unterschiedlicher Klassen), die miteinander kommunizieren.
- Dies geschieht hauptsächlich über gegenseitige Methodenaufrufe.
- Vereinfacht: die Kunst ist „nur noch“, die richtigen Methoden in der richtigen Reihenfolge aufzurufen, anstatt die richtigen Anweisungen.

Objekte, Objekte, Objekte

- Vorsicht: das suggeriert, dass ich nur noch Knöpfe richtig zusammenstecken/drücken muss, ohne zu wissen, wie die Knöpfe genau funktionieren (nur was sie tun).
- Das ist zwar richtig, dazu gehört aber leider auch die Fähigkeit, einzelne Knöpfe (Methoden) selbst zu programmieren (dazu brauchen wir funktionales und imperatives Programmieren).
- Und dazu gehört aber auch, zu verarbeitende Daten als Objekte zu abstrahieren und in Klassen zu strukturieren/modellieren.
- Im Folgenden, diese zwei Aspekte (oo Modellierung und oo Programmierung) im Detail . . .

Klassen als Mittel der Abstraktion

- Klassen bieten eine weitere Möglichkeit der *Abstraktion*, da es dem Programmierer nun möglich ist, ähnliche Objekte (mit ähnlichen Eigenschaften) zusammen zu fassen.
- Statt für alle möglichen Objekte gewisse Eigenschaften (insbes. Methoden) zu programmieren, werden die Unterschiede wegabstrahiert und die Methode einmal zur Verfügung gestellt (implementiert).
- Das spart nicht nur Arbeit sondern auch potentielle Fehlerquellen wie z.B. copy-paste von vermeintlich redundantem Programmcode.

Klassen als Mittel der Abstraktion

- Ein anschauliches Beispiel:
  - Sie lernen einmal wie ein Auto zu fahren ist und können dann alle möglichen Autos (untersch. Hersteller etc.) fahren.
  - Warum? Die Bedienung abstrahiert von den speziellen (für das Fahren irrelevanten) Eigenschaften der unterschiedlichen Fabrikate.
  - Sie müssen z.B. nur wissen, wie man das Lenkrad dreht, um zu lenken (wie diese Lenk-Bewegung tatsächlich auf die Straße gebracht wird, müssen Sie nicht wissen).
  - Analoges gilt fürs Gasgeben, Bremsen, Blinken, etc.

Klassen als Mittel der Abstraktion

- Noch ein Beispiel: Lichtschalter in einem Haus
  - Alle Lichtschalter sind gleich zu bedienen.
  - Alle Lichtschalter sind gleich konstruiert.
  - Dennoch unterscheidet sich der Lichtschalter für die Flurbeleuchtung vom Lichtschalter fürs Badezimmer: Es ist nicht derselbe Lichtschalter.
  - Statt jeden einzelnen Lichtschalter neu zu modellieren/programmieren, abstrahiert man eine Klasse Lichtschalter, der alle Funktionalitäten nur einmal (für alle möglichen Lichtschalter) implementiert.

Klassen als Mittel der Abstraktion

- In unserem konkreten Beispiel der Klasse `Rechteck`:
  - Sie können z.B. testen, ob ein Punkt in einem beliebigen Rechteck ist, ohne andere (für diesen Test irrelevante) Besonderheiten unterschiedlicher Rechtecke (wie z.B. der Flächeninhalt, die Farbe des Rechtecks (wenn es überhaupt eine hat)) zu kennen.
  - D.h. Sie müssen nicht alle Details eines Rechtecks kennen, nur die für die Aufgabe wesentlichen.
- Fazit: Abstraktion hilft, Details zu ignorieren, und reduziert damit die Komplexität des Problems; dadurch werden komplexe Apparate und Techniken beherrschbar.

Klassen als Mittel der Datenkapselung

- Eine weitere wichtige Errungenschaft von Klassen ist die Möglichkeit, die innere Struktur der Objekte einer Klasse (also die Daten-Komponenten/Instanzvariablen) vor dem Benutzer/Programmierer zu *verstecken*.
- Wie erwähnt spricht man hier von *Datenkapselung* (engl. *information hiding*).
- Konkret geschieht das in Java mit dem schon kurz angesprochenen Sichtbarkeitsattribut **`private`** für “nur innerhalb dieser Klasse sichtbar”.

Klassen als Mittel der Datenkapselung

- Unsere Klasse `Rechteck` nun sauber gekapselt:

```
class Rechteck {
    private double xLow;
    private double xUp;
    private double yLow;
    private double yUp;

    public double getXLow() { return xLow; }
     ...
     // weitere Objekt-Methoden
     // typischweise als public deklariert
}
```

Klassen als Mittel der Datenkapselung

- Die Instanz-Variablen repräsentieren den *Zustand* der Objekte, sind sie nun als **`private`** deklariert, sind sie außerhalb der Klassendefinition nicht mehr sichtbar!!!
- Die Methoden sind nur einmal realisiert / definiert und operieren bei jedem Aufruf auf den Daten eines bestimmten konkreten Objekts (sie repräsentieren das Verhalten der Objekte).
- Die nicht als **`private`** deklarierten Methoden bilden die öffentliche Schnittstelle der Klasse.

Klassen als Mittel der Datenkapselung

- Kapselung bedeutet, dass (von gewollten Ausnahmen abgesehen) die nicht als **`private`** deklarierten Methoden die einzige Möglichkeit darstellen, mit einem konkreten Objekt zu kommunizieren und so Informationen über dessen Zustand (die aktuellen Werte der Instanz-Variablen) zu gewinnen oder diesen zu verändern.
- Meist sind diese Methoden als **`public`** deklariert. Fehlt die Sichtbarkeit, sind sie wie erwähnt zumindest im eigenen Package “von außerhalb der Klasse” sicht- und aufrufbar (package-scoped). Darüberhinaus gibt es noch die Sichtbarkeit **`protected`**, die wir auch sehr bald kennen lernen werden.

Klassen als Mittel der Datenkapselung

- Die Sichtbarkeitsattribute sind entscheidend für die Umsetzung der Kapselung.
- Kapselung hilft, die Komplexität der Bedienung eines Objekts zu reduzieren.
- Durch Kapselung werden die Implementierungsdetails von Objekten verborgen.
- Dadurch können Daten nicht bewusst oder versehentlich verändert werden.
- Kapselung ist daher auch ein wichtiger Sicherheitsaspekt: Ein direkter Zugriff auf die Daten wird unterbunden, der Zugriff erfolgt nur über definierte *Schnittstellen*, nämlich über die bereitgestellten Methoden.

Klassen als Mittel der Datenkapselung

Beispiel:

- Welche Attribute den Zustand eines Lichtschalters definieren kann uns egal sein, solange wir wissen, wie wir den Lichtschalter bedienen müssen.
- Tatsächlich können wir den Zustand eines Lichtschalters nur über dessen Schnittstelle verändern (indem wir den Schalter drücken oder drehen).
- Und das ist vermutlich gut so, andernfalls könnten wir möglicherweise einen Stromschlag bekommen . . .

Klassen als Mittel der Datenkapselung

- Nochmal die Klasse `Rechteck`:

```
public class Rechteck {
      // Attribute: alle private
      private double xLow;
      private double xUp;
      private double yLow;
      private double yUp;

      // Methoden: alle public
      public double getXLow() { return xLow; }
      public double getYLow() { return yLow; }
      public double getXUp() { return xUp; }
      public double getYUp() { return yUp; }
      public void verschiebe(double xDir, double yDir) { ... }
}
```

Klassen als Mittel der Datenkapselung

- Zugriff auf den Zustand eines Objekts erfolgt hier nur z.B. über die **`public`** Methode `getXLow`, die als **`private`** deklarierten Instanz-Variablen sind *nach außen* nicht sichtbar.
- Die Methode `verschiebe(`**`double`**` xDir, `**`double`**` yDir)` ist in dieser Implementierung die einzige Möglichkeit, um den Zustand eines konkreten Rechtecks zu ändern.
- Will man dies ändern, muss main weitere Methoden hinzufügen, z.B.

```
public void setXLow(double newXLow) { xLow = newXLow; }
```

- All diese Methoden kann man benutzen ohne Implementierungsdetails zu kennen.

Klassen als Mittel der Datenkapselung

- Wie erwähnt, wäre eine (gleichwertige) Alternative, das Rechteck durch vier Doubles zu modellieren, die nun den linken unteren Punkt und eine Ausdehnung in x- und eine Ausdehnung in y-Richtung definieren.
- Die entsprechende Klasse `RechteckAlternativ`:

```
public class RechteckAlternativ {
     private double xCoord; // x-Koordinate des Punktes
     private double yCoord; // y-Koordinate des Punktes
     private double xRange; // x-Ausdehnung
     private double yRange; // y-Ausdehnung
     ...
}
```

Klassen als Mittel der Datenkapselung

- Die Methode `verschiebe` ist hier vermtlich einfacher und (ein klitziklein wenig) effizienter zu implementieren (wir müssen nur zwei Werte ändern).
- An diesem Beispiel sieht man auch die Stärken der Kapselung: dem Benutzer der Methode `verschiebe` ist letztlich herzlich egal, wie die Klasse nun intern strukturiert ist.
- D.h. vielleicht hat der Programmierer der Klasse `Rechteck` sogar zunächst Variante 1 gewählt und später die effizientere Variante 2 gefunden.
- Er kann das problemlos ändern, solange die Signatur der Methoden (Schnittstelle) sich nicht ändert kriegen die Benutzer das garnicht mit.

Klassen als Mittel der Datenkapselung

- Übrigens: auch der Typ `String` ist ein Beispiel für Daten-Kapselung.
- Die Implementierungsdetails von Zeichenketten (letztlich als `char[]` bleiben vor dem Benutzer verborgen.
- Schnittstelle dieser Klasse ist aber relativ speziell und ist für die allgemeine Veranschaulichung aller weiteren Aspekte der oo Programmierung nicht ganz perfekt geeignet.
- Durch saubere Datenkapselung sind Klassen die perfekte und konsequente Erweiterung der Idee der *abstrakten Datentypen*.

Klassen unterstützen die Wiederverwendbarkeit

- Durch Abstraktion und Kapselung wird die *Wiederverwendbarkeit* von Programmteilen gefördert.
- Wiederverwendbarkeit hilft, effizienter und fehlerfreier zu programmieren.
- Beispiel: Sog. Collections sind Objekte, die Sammlungen anderer Objekte aufnehmen und verarbeiten können.

Klassen unterstützen die Wiederverwendbarkeit

- Collections sind meist sehr kompliziert aufgebaut, haben dafür aber typischerweise eine einfache Art der Bedienung (Schnittstelle).
- Werden Collections als Klasse realisiert, werden durch die Kapselung die komplexen Details des Aufbaus wegabstrahiert.
- Dies erleichtert die Wiederverwendung: Wenn ein Programm eine spezielle Collection benötigt, muss ein Objekt der passenden Klasse erzeugt werden, auf das das Programm über die einfache Schnittstelle (Methoden der Klasse) zugreifen kann.

Beziehungen zwischen Klassen

- Betrachten wir noch einmal unsere Klasse mit einer weiteren Methode, die prüft, ob ein Punkt innerhalb des Rechtecks liegt:

```
public class Rechteck {
   private double xLow;
   private double xUp;
   private double yLow;
   private double yUp;
     ...
  public boolean pointInside(double xCoord, double yCoord) { ... }
}
```

- Ist das Argument der Methode `pointInside` nicht ein Objekt *Punkt*, bestehend aus zwei Koordinaten ($xCoord, yCoord$)?
- Das Rechteck selbst wird ebenfalls von zwei solchen *Punkten* ($xLow, yLow$) und (xUp, yUp) aufgespannt.

Beziehungen zwischen Klassen

- Das riecht nach einer Klasse `Punkt`! Und hier ist sie:

```
public class Punkt {
   private double x;
   private double y;

   public double getX() { return x; }
   public double getY() { return y; }
}
```

- Damit können wir auch `Rechteck` neu definieren

```
public class Rechteck {
     private Punkt lowLeft;
     private Punkt upRight;

     public boolean pointInside(Punkt p) { ... }
     ...
}
```

- Klassen (Objekte) existieren i.d.R. nicht isoliert, sondern stehen in *Beziehung* zueinander.

Beziehungen zwischen Klassen

- Grundsätzlich gibt es in der oo Programmierung drei Arten von Beziehungen:
- *Generalisierung* und *Spezialisierung* (“is-a”-Beziehungen)
  - Beispiel: ein *Quadrat* und ein *Rechteck* (Spezialisierungen) sind *Vierecke* (Generalisierung).
- *Aggregation* und *Komposition* (“part-of”-Beziehungen)
  - Beschreibt die Zusammensetzung eines Objekts aus anderen Objekten (Aggregation: nicht essentiell, Komposition: essentiell).
  - Beispiel: ein *Punkt* ist Teil eines *Rechtecks* (Komposition, das Rechteck existiert sonst nicht)
- *Verwendungs-* und *Aufrufbeziehungen* (typw. in Methoden)
  - Beispiel: ein *Rechteck* verwendet in unserem Beispiel einen *Punkt* als Eingabeparameter in einer Methode.

Vererbung

- Eine “is-a”-Beziehung zwischen zwei Klassen `A` und `B` sagt aus, dass `B` alle Eigenschaften von `A` besitzt (und darüberhinaus typischerweise noch ein paar mehr — `B` ist ja eine Spezialisierung von `A`).
- “is-a”-Beziehungen werden in der oo Programmierung durch *Vererbung* modelliert.
- Die speziellere Klasse wird dabei nicht komplett neu definiert, sondern von der allgemeineren Klasse *abgeleitet*.
- Die speziellere Klasse *erbt* implizit alle Eigenschaften der allgemeineren Klasse (auch: *Vaterklasse* oder *Oberklasse*) ohne, das sie nochmal explizit aufgeführt werden müssen.
- Eigene Eigenschaften können nach Belieben hinzugefügt werden.

Vererbung

- Vererbungen können mehrstufig sein, d.h. eine abgeleitete Klasse kann wiederum Vaterklasse für andere abgeleitete Klassen sein (dies führt ggfs. zu einer *Vererbungshierarchie*).
- Grundsätzlich kann eine Klasse auch von mehreren Vaterklassen abgeleitet sein (z.B. ist ein Amphibienfahrzeug eine Spezialisierung sowohl eines Wasserfahrzeug als auch eines Landfahrzeug).
- In diesem Fall spricht man von *Mehrfachvererbung*.