Beziehungen zwischen Klassen

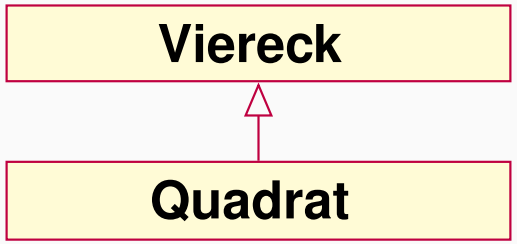
- Grundsätzlich gibt es in der oo Programmierung drei Arten von Beziehungen:
- *Generalisierung* und *Spezialisierung* (“is-a”-Beziehungen)
  - Beispiel: ein *Quadrat* und ein *Rechteck* (Spezialisierungen) sind *Vierecke* (Generalisierung).
- *Aggregation* und *Komposition* (“part-of”-Beziehungen)
  - 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`).
- Das Beispiel von der vorherigen Folie:
  - Ein *Quadrat* “ist ein” *Viereck*.
  - Ein *Quadrat* hat damit alle Eigenschaften eines *Vierecks* (vermutlich v.a. die vier Ecken).
  - Ein *Quadrat* hat aber noch speziellere Eigenschaften als allgemeine (andere) *Vierecke* (z.B. dass alle Kanten gleich lang sind, die Kanten-Winkel alle 90 Grad haben, ...).
- “is-a”-Beziehungen werden in der oo Programmierung durch *Vererbung* modelliert.

Vererbung

- Wir werden Vererbungsbeziehungen graphisch darstellen:



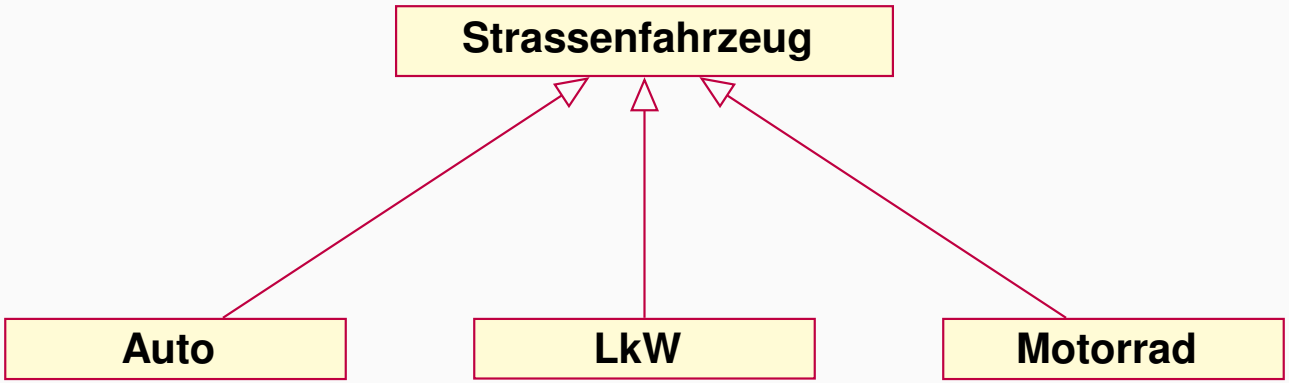
- Vererbung ist eine spezielle Form der Wiederverwendung:
  - Die speziellere Klasse wird nämlich nicht komplett neu definiert, sondern von der allgemeineren Klasse *abgeleitet*.
  - Die speziellere Klasse *erbt* implizit alle Eigenschaften der allgemeineren Klasse (auch: *Vaterklasse* oder *Oberklasse*) ohne, das sie nochmal explizit aufgeführt werden müssen.
  - Eigene Eigenschaften können nach Belieben hinzugefügt werden.

Vererbung

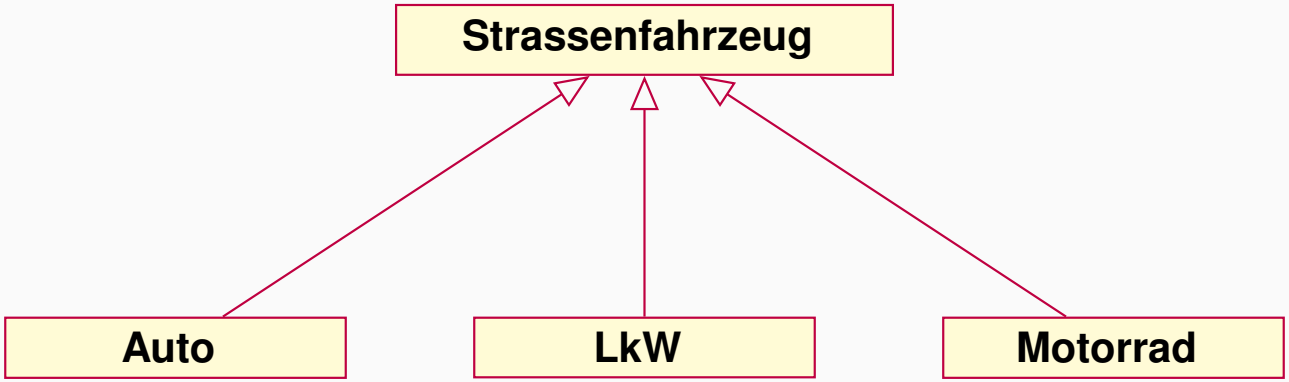
- Vererbungen können mehrstufig sein, d.h. eine abgeleitete Klasse kann wiederum Vaterklasse für andere abgeleitete Klassen sein (dies führt ggfs. zu einer *Vererbungshierarchie*).
- Grundsätzlich kann eine Klasse auch von mehreren Vaterklassen abgeleitet sein (z.B. ist ein Amphibienfahrzeug eine Spezialisierung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

Polymorphismus (bei Objekten)

- Beispiel: *Fährschiffe* können *Autos*, *Lastwägen* und *Motorräder* – oder ganz allgemein *Straßenfahrzeuge* (deren Vaterklasse/Generalisierung) – aufnehmen.
- Um diese Beziehung zu beschreiben, genügt es, zu definieren, dass *Fährschiffe* Objekte vom Typ *Straßenfahrzeuge* aufnehmen (Aggregation).



Polymorphismus (bei Objekten)



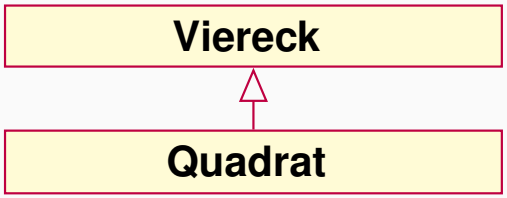
- *Polymorphismus* besagt, dass *Fährschiffe* nicht nur Objekte der Klasse *Straßenfahrzeuge* aufnehmen können, sondern auch Objekte aller abgeleiteten Klassen, also auch von *Autos*, *Lastwägen* und *Motorräder*.

Polymorphismus (bei Objekten)

- Warum? Naja, die Unterklassen von *Straßenfahrzeuge* haben alle Eigenschaften von *Straßenfahrzeuge*, also auch die, die wesentlich sind, um von *Fährschiffe* aufgenommen zu werden.
- Die zusätzlichen Eigenschaften der abgeleiteten Klassen sind für *Fährschiffe* irrelevant.
- Vorsicht: Andersherum funktioniert Polymorphismus nicht: Wenn *Fährschiffe* nur *Autos* aufnehmen könnte, dann nicht automatisch auch *Straßenfahrzeuge*!

Polymorphismus (bei Methoden)

- Ein weiterer Aspekt ist, dass Objekte unterschiedlicher Klassen (in einer Vererbungshierarchie) die gleiche Funktionalität besitzen können (die allerdings in jeder Klasse unterschiedlich realisiert ist).
- Als Beispiel schauen wir uns nochmal Vierecke und Quadrate an:



- In einem Graphik-Editor werden wir möglicherweise auf der Zeichenfläche wiederum Vierecke aufzeichnen können wollen.
- Wegen dem Objekt-Polymorphismus kann die Zeichenfläche dann automatisch auch Quadrate (und ggfls. weitere Unterklassen von Viereck) aufnehmen.

Polymorphismus (bei Methoden)

- Beide Klassen (Viereck und Quadrat) haben die Funktionalität, den Flächeninhalt zu berechnen, z.B. durch eine Methode “flaeche”.
- Sinnvollerweise haben diese Methoden in beiden Klassen denselben Namen, da sie ja die semantisch gleiche Funktionalität anbieten.
- In beiden Klassen steckt aber jeweils ein unterschiedlicher Algorithmus dahinter.
- Diese Funktionalität kann bereits in der Vaterklasse zur Verfügung stehen und in den abgeleiteten Klassen *überschrieben* (*verfeinert*) werden.
- Wird in der Zeichenfläche dann der Flächeninhalt eines Vierecks benötigt (die Methode aufgerufen), wird dynamisch (und automatisch) ausgewählt, welche Methode ausgeführt wird (je nachdem von welchem Typ das Objekt ist, dessen Fläche gerade berechnet werden soll).

Polymorphismus — Zusammengefasst

- Polymorphismus erhöht wiederum die Wiederverwendbarkeit.
- Wenn ich weitere Unterklassen von Viereck definiere, jeweils mit eigenen Varianten der Methode “flaeche” muss ich mich trotzdem um nichts kümmern:
- Wenn die Zeichenfläche des Editors Vierecke verwalten kann, kann es auch Objekte aller Unterklassen von Viereck verwalten.
- Wird für ein solches Viereck eine Methode aufgerufen, wird automatisch entschieden, welche “Version” der Methode (also aus welcher Klasse stammt das konkrete Objekt) ausgeführt werden muss.
- Da dies automatisch geht, muss ich nicht für alle möglichen Klassen extra implementieren, wie die Zeichenfläche diese verwaltet; es reicht einmal für Vierecke.

- Betrachten wir das oo Paradigma noch aus der zweiten Perspektive, der Programmierung auf höherem Abstraktionslevel.
- Oo Programmierung ist ein weiterer Ansatz, Problemstellungen der realen Welt zu modellieren.
- Der oo Ansatz orientiert sich dabei an *Dingen* (Objekte, die gewisse Eigenschaften und ein gewisses Verhalten haben), die modelliert werden müssen.
- Die oo Sichtweise stellt sich die Welt als *System von Objekten* vor, die untereinander *Botschaften* austauschen.
- Ein oo Programm besteht daher aus Objekten, die miteinander interagieren.

Beispiel

- Susi in Rosenheim möchte ihrer Freundin Gabi in Buxtehude einen Blumenstrauß schicken.
- Susi geht zum Blumenhändler Mark und erteilt ihm den Auftrag.
- In der oo Programmierung sind Susi und Mark Objekte: Susi sendet an Mark eine Botschaft: *Sende 7 gelbe Rosen an Gabi, Schneestr. 1 in Buxtehude*.
- Susi hat getan, was sie konnte, nun ist es in Marks Verantwortung, den Auftrag zu bearbeiten.
- Mark versteht die Botschaft und weiß, was zu tun ist, d.h. er kennt einen Algorithmus für das Verschicken von Blumen.
- ...

- ...
- Er versucht, einen Blumenhändler in Buxtehude zu finden: Florist Sascha; Mark schickt Sascha eine entspr. Botschaft (z.B. mit Absender).
- Mark ist damit fertig: er hat die Verantwortung für den Prozess an Sascha weitergegeben.
- Auch Sascha hat einen zur Botschaft passenden Algorithmus parat: er stellt die 7 Rosen zusammen und beauftragt seinen Boten Daniel, sie auszuliefern.
- Daniel muss den Weg zur Zieladresse finden und fragt sein Navi, das ihm entsprechend antwortet.
- Daniel findet den Weg, überreicht Gabi die Blumen und teilt ihr in einer Botschaft den Absender mit.
- Damit ist der von Susi angestoßene Vorgang, an dem mehrere Objekte beteiligt waren, beendet.

- Dieser ganze Prozess ist ein klassisches oo Programm.
- Mark ist ein Objekt vom Typ Blumenhändler, Susi hat seine Methode „Auftrag erteilen“ aufgerufen.
- Diese Methode erkennt, dass es den Auftrag weiterleiten muss und sucht Sascha, ein weiteres Objekt der Klasse Blumenhändler und ruft wieder dessen Methode „Auftrag erteilen“ auf.
- Die Methode erkennt nun, dass Sascha den Auftrag ausführen kann und ruft schließlich die Methode von Daniel (Klasse Bote) „ausliefern“ auf.
- . . .
- Wie schön aufgeräumt (strukturiert) alles ist!!!

- Objekte besitzen Eigenschaften (die Attribute/Instanzvariablen) z.B. Blumenhändler: Ort an dem er sein Geschäft hat, Name, Telefonnummer, Öffnungszeiten, Warenbestand, etc.
- Objekte können bestimmte Operationen (die (Objekt-)Methoden) ausführen, z.B. Blumenhändler: Lieferauftrag für Blumen entgegennehmen, Sträuße binden, Boten schicken, etc.
- Wenn ein Objekt eine geeignete Botschaft empfängt, wird eine zur Botschaft passende Operation gestartet (Methode aufgerufen).
- Der Umwelt (d.h. den anderen Objekten) ist bekannt welche Methoden ein Objekt beherrscht.
- Allerdings weiß die Umwelt von den Methoden nur, was sie bewirken und welche Daten sie als Eingabe benötigen.

- Die Umwelt weiß aber nicht, wie das Objekt funktioniert, d.h. nach welchen Algorithmen die Botschaften verarbeitet werden.
- Das bleibt „privates“ Geheimnis des Objekts
  - Z.B. hat Susi keine Ahnung, wie Mark den Blumentransport bewerkstelligt (es interessiert sie vermutlich auch gar nicht).
  - Susis Aufgabe war einzig und allein, ein für ihr Problem *geeignetes* Objekt zu finden und diesem eine geeignete Botschaft zu senden (ungeeignete Objekte wären z.B. Kathi die Zahnärztin oder Markus der Immobilienmakler gewesen, denn diese Objekte hätten Susis Nachricht nicht verstanden).
  - Für Susi war zudem wichtig, zu wissen, *wie* sie die Botschaft für Mark formulieren muss.
- Eine Methode ist die Implementierung eines Algorithmus, in Java kommt hier bekanntermaßen das imperative Paradigma ins Spiel.

- Die Objekte in diesem Beispiel kann man in Gruppen (Klassen) einteilen
  z.B. sind Sascha und Mark *Blumenhändler*:
  - Sie beherrschen *die selben Methoden* und besitzen *die selben Attribute* (z.B. Ort oder Öffnungszeiten).
  - Sie sind aber unterschiedliche Objekte: die Attribute haben *unterschiedliche Werte*.

  Man sagt: Sascha und Mark sind Objekte (Instanzen) der Klasse *Blumenhändler*.
- Eine Klasse ist eine Definition eines bestimmten Typs von Objekten, ein Bauplan, in dem die Methoden und Attribute beschrieben werden.
- Nach diesem Schema können Objekte (Instanzen) einer Klasse erzeugt werden.
- Ein Objekt ist eine Konkretisierung (Inkarnation) einer Klasse.

- Alle Instanzen einer Klasse sind von der Struktur (Methoden/Attribute) her gleich, sie unterscheiden sich allein in der Belegung ihrer Attribute mit Werten[2].
- Beispiel: Sascha und Mark haben beide das Attribut Ort, aber bei Sascha hat es den Wert „Buxtehude“ und bei Mark den Wert „Rosenheim“.

[2] Natürlich denken Sie dabei an Substitutionen, und haben absolut recht dabei: unser Zustands-Modell gilt genauso für Instanz-Variablen von Objekten wie für lokale Variablen primitiver bzw. Referenz-Typen.

Zusammenfassung

- Eine Klasse definiert die Attribute und Methoden ihrer Objekte.
- Der Zustand eines konkreten Objekts wird durch seine Attributwerte und Verbindungen (Links) zu anderen konkreten Objekten bestimmt.
- Das mögliche Verhalten eines Objekts wird durch die Menge von Methoden beschrieben.
- Die wichtigsten Konzepte der oo Programmierung sind:
  - Abstraktion
  - Kapselung
  - Wiederverwendung
  - Beziehungen
  - Polymorphismus

  die wir uns bereits angeschaut haben.

Kapitel 3: Klassen und Objekte in Java i

Ein Beispiel

- Zurück zu unserem ganz ursprünglichen Beispiel mit der Klasse für Rechtecke.
- Wir nehmen an, dass wir für unsere Anwendung Rechtecke verwalten wollen.
- Konkret wollen wir mit den Rechtecken folgendes tun:
  - Punkt-in-Rechteck-Test
  - Rechteck um jeweils einen spezifischen Wert entlang der x- und y-Richtung verschieben
- Wir hatten bereits diskutiert, dass wir zwei Klassen *Rechteck* und *Punkt* dafür definieren können.
- Aus oo Sicht sollten wir das auch tun.

Ein Beispiel

- Allerdings sollten wir uns überlegen, wie diese Klassen genau aussehen sollen, also welche Attribute und v.a. welche Funktionalitäten (also Methoden) diese Klassen haben sollen.
- Das Ergebnis dieser Überlegung ist ein *Modell* der realen Welt, das in der oo Softwareentwicklung auch *statischer Entwurf* genannt wird.
- Wie schon bei der imperativen oder funktionalen Programmierung ist der Entwurf der Klassen (und damit die Modellierung der Problemstellung) eine entscheidende Herausforderung.
- Hinzu kommt natürlich der Entwurf der Algorithmen, die die Methoden der Klassen implementieren.

UML Klassendiagramme

- Der Entwurf umfasst häufig viele Klassen (die in Beziehung zu einander stehen, daher gibt es für die Darstellung des oo Entwurfs die Konzeptsprache *Unified Modelling Language* (UML).
- UML ist eine Art Pseudo-Code, der allerdings eine wohl-definierte Semantik besitzt und von vielen Programmen verarbeitet werden kann (z.B. können Implementierungsdetails z.B. in Java-Notation angegeben werden, aus denen automatisch Java-Code generiert werden kann).
- UML-Code selbst ist nicht ausführbar, dennoch wird UML von vielen Experten als Prototyp für die nächste Generation von Programmiersprachen betrachtet.

UML Klassendiagramme

- Im Rahmen dieser Vorlesung werden wir ebenfalls UML-*Klassendiagramme* verwenden, die den statischen Entwurf (Klassen, Objekte und deren Beziehungen zueinander) konzeptionell beschreiben.
- Realisierungsdetails werden i.Ü. meist mit anderen Diagrammtypen beschrieben, die wir hier nicht verwenden.
- Einen tieferen Einblick in UML erhalten Sie in den Vorlesungen zur Software-Entwicklung.

Entwurf von Rechteck und Punkt

- Die Klasse *Punkt* ist sehr einfach modelliert:
  Ein (2D-)Punkt besteht aus zwei Koordinaten vom Typ **`double`**.
- Als Methoden definieren wir zunächst den Zugriff auf die Koordinaten und eine Methode, die den Punkt verschiebt.
- In UML schaut das dann so aus:

Punkt
xCoord : double yCoord : double
getXCoord() : double getYCoord() : double verschiebe(xDir: double, yDir: double) : void

Entwurf von Rechteck und Punkt

- Die Klasse *Rechteck* modellieren wir mit einem (Anker-)Punkt (links unten) und zwei Werten vom Typ `double` für die Ausdehnung in x- bzw. y-Richtung (2. Variante von vorhin).
- D.h., Rechteck und Punkt stehen (wie bereits erwähnt) in einer Kompositions-Beziehung zueinander, in UML mit einem speziellen Pfeil dargestellt:

- Die Annotation “origin” ist der Name der “Rolle”, die Punkt in Rechteck spielt (hier der Ursprungs-Punkt = links unten).
- Die Annotation “1” bedeutet, dass ein Rechteck genau einen Punkt verwendet.

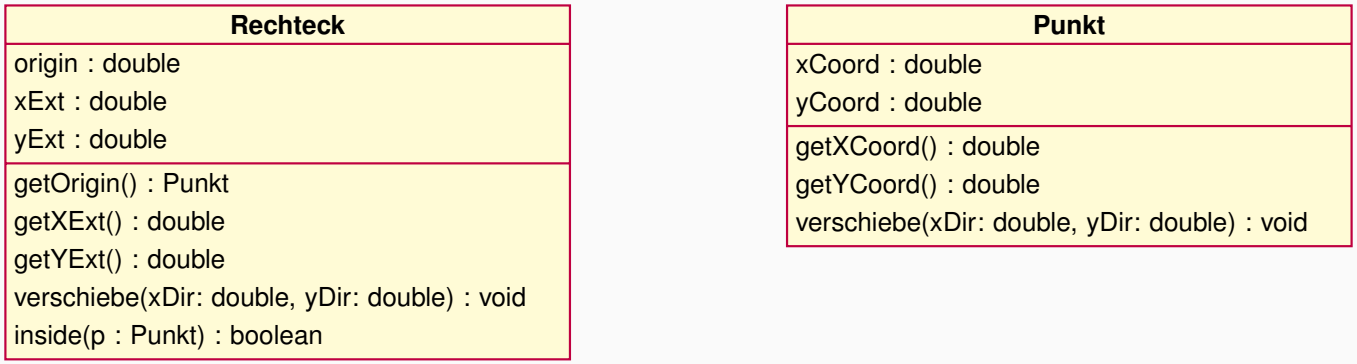
Entwurf von Rechteck und Punkt

- Diese Kompositions-Beziehung ist in diesem Fall sehr einfach umzusetzen:
- Die Klasse Rechteck bekommt ein Attribute (Instanz-Variable) “origin” vom Typ Punkt.
- Im nachfolgenden Klassendiagramm ist das entsprechend berücksichtigt, sodass die Komposition nicht mehr durch einen entsprechenden Pfeil dargestellt werden muss[3].
- Wir gehen hier i.Ü. nicht näher auf die Syntax und Semantik von Assoziationen in UML ein (es gibt auch einen speziellen Pfeil für die Aggregation bzw. für allgemeine Beziehungen und natürlich für die Vererbung, den kennen wir ja schon), auch das lernen Sie in anderen Vorlesungen.

[3] Warum gibt es diese Pfeil-Darstellung für Komposition überhaupt?

Entwurf von Rechteck und Punkt

- Als Methoden definieren wir den Zugriff auf den (Anker-)Punkt und die Ausdehnungen sowie eine Methode, die das Rechteck verschiebt und testet, ob ein Punkt in dem Rechteck enthalten ist
- Dann schaut das ganze (wie gesagt mit bereits umgesetzter Komposition) aus:



Die Klasse `Punkt`

- Implementierung der Klassen in Java:

```
public class Punkt {
     /** Die x-Koordinate des Punkts. */
     private double xCoord;
     /** Die y-Koordinate des Punkts. */
     private double yCoord;

     /** Gibt die x-Koordinate des Punkts zur&uuml;ck. */
     public double getXCoord() {
          return xCoord;
     }

     /** Gibt die y-Koordinate des Punkts zur&uuml;ck. */
     public double getYCoord() {
          return yCoord;
     }

     /** Verschiebt den Punkt um entsprechende Werte in x-Richtung und in yRichtung. */
     public void verschiebe(double xDir, double yDir) {
          xCoord = xCoord + xDir;
          yCoord = yCoord + yDir;
     }
}
```

Die Klasse `Rechteck`

```
public class Rechteck {
     private Punkt origin;
     private double xExt;
     private double yExt;

      /** Gibt den (Anker-) Punkt des Rechtecks zur&uuml;ck. */
     public Punkt getOrigin() {
          return origin;
     }

     /** Gibt die x-Ausdehnung des Rechtecks zur&uuml;ck. */
     public double getXExt() {
          return xExt;
     }

     /** Gibt die y-Ausdehnung des Rechtecks zur&uuml;ck. */
     public double getYExt() {
          return yExt;
     }

     /** Verschiebt das Rechteck um entsprechende Werte in x-Richtung und in yRichtung. */
     public void verschiebe(double xR, double yR) {
          origin.verschiebe(xR, yR);
     }

     /** Testet, ob ein Punkt im Rechteck liegt. */
     public boolean pointInside(Punkt p) {
          return (origin.getX() <= p.getX() && p.getX() <= origin.getX()+xExt)
                    &&
                 (origin.getY() <= p.getY() && p.getY() <= origin.getY()+yExt)
     }
}
```

Verwendung

- Um ein Objekt der Klasse `Rechteck` zu *erzeugen*, muss man an der ensprechenden Stelle im Programm eine Variable vom Typ der Klasse deklarieren und ihr mit Hilfe des **`new`**-Operators ein neu erzeugtes Objekt zuweisen (so wie wir es von Record-Typen kennen):

```
public class Test {
     public static void main(String[] args) {
          ...
          Rechteck r;
          r = new Rechteck();
          Punkt p = new Punkt();
          ...
          r.verschiebe(2.0,3.0);
     }
}
```

Verwendung: Objekterzeugung

- Dei Anweisung `Rechteck r;` ist natürlich eine klassische Deklaration einer Variablen vom Typ der Klasse (Objekttyp).
- Wie bei Record-Typen wird hier anstelle eines primitiven Datentyps der Name einer zuvor definierten Klasse verwendet.
- Die Variable `r` wird angelegt, darauf steht nun eine *Referenz* (*Zeiger*) auf einen speziellen Speicherplatz für das Objekt (das noch nicht existiert).
- Anweisung `r = new Rechteck();`: Generiert das Objekt mittels des **`new`**-Operators.
- Deklaration und Objekterzeugung kann natürlich wieder zusammenfallen, siehe `Punkt p = new Punkt();`.

Verwendung: Objekterzeugung

- Nach der Generierung eines Objekts haben alle Attribute mit primitiven Datentypen zunächst ihre entsprechenden Standardwerte (siehe Arrays).
- Was ist der Standard-Wert von Objekttypen (z.B. `Punkt`)?
- Attribute mit Objekttypen haben den Standardwert **`null`**, die *leere* Referenz.
- Also: Variable `r` ist eine Referenz-Variable mit Verweis auf das "Rechteck", das auf der Halde liegt.
- Auf der Halde liegen die Zettel für die Instanz-Variablen: dort wird auch wieder zwischen "normalen" Variablen (mit primitiven Werten) oder Referenz-Variablen (mit Referenzen auf einen anderen Ort der Halde) unterschieden!
- Bei Objekt-Erzeugung stehen auf den Instanz-Variablen von `r` die Default-Werte.

Verwendung: Objekterzeugung

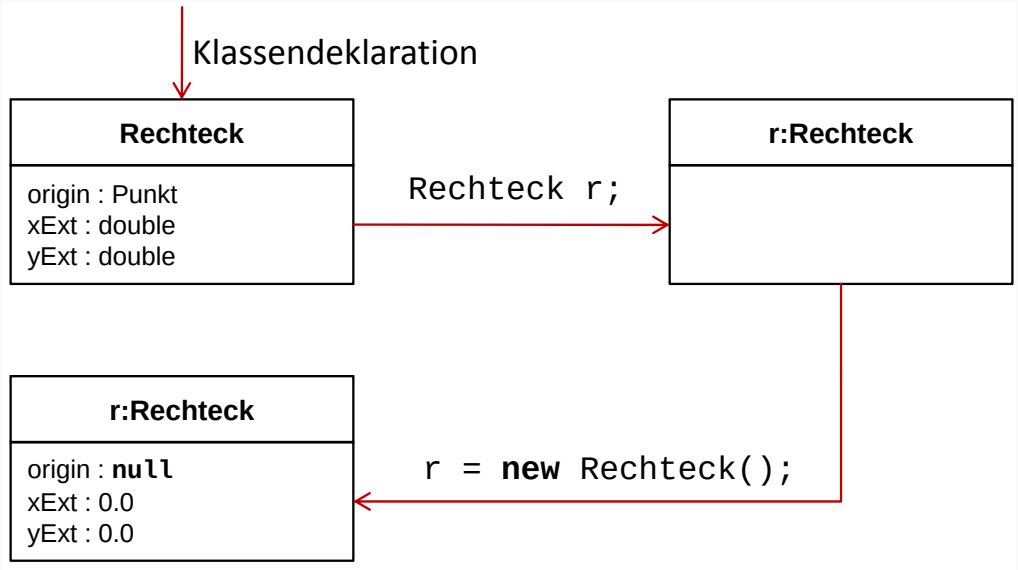
- Zugriff auf Zustand / Verhalten eines Objekts:
  - **`private`**: Auf diese Attribute / Methoden kann außerhalb der Klasse nicht zugegriffen werden, z.B. kann aus der `main`-Methode der Klasse `Test` für das rechteck `r` nicht auf das Attribut `origin` der Klasse `Rechteck` über `r.origin` zugegriffen werden.
  - **`public:`** Auf diese Attribute / Methoden kann außerhalb der Klasse über die Punktnotation zugegriffen werden, z.B. kann aus der `main`-Methode der Klasse `Test` für das Rechteck `r` nicht auf das Attribut `origin` der Klasse `Rechteck` über `r.origin` zugegriffen werden.
- Beispiel:

```
public class Test {
   Rechteck r = new Rechteck();
   Punkt p = new Punkt();
   r.origin = p; // nicht erlaubt!!!
}
```

- Offensichtlich widerspricht der direkte Zugriff auf den Zustand (Attribute) eines Objekts dem Prinzip der Kapselung.
- Daher ist es guter oo Programmierstil, Attribute grundsätzlich als **`private`** zu deklarieren.
- In unserem Fall haben wir es als sinnvoll erachtet, Methoden zu vereinbaren, die den (lesenden) Zugriff auf den Zustand erlauben, z.B. die Methode **`public double`** `getX()` in der Klasse `Punkt`.
- Diese Methoden heißen auch *Getter* (-Methoden).
- Warum ist es besser, Getter-Methoden zu schreiben, als den direkten Zugriff zu erlauben? — OK, das wissen Sie natürlich längst.

Objekterzeugung

- Nochmal der Erzeugungsprozess bildlich:



- Jetzt stellt sich natürlich die Frage, wie man die Attributswerte des Rechtecks `r` verändern bzw. setzten kann (z.B. würde ich gerne ein Rechteck mit dem Ankerpunkt $(1.0, 4.0)$ und mit Ausdehnung 2.0 in x- bzw. 3.2 in y-Richtung erzeugen).

Setter-Methoden

- Momentan keine Chance: alle Attribute sind **`private`** und über die Methoden können wir die Attribute nicht explizit setzen.
- Dann ist das Prinzip der Kapselung ja total sinnlos???
- Mitnichten. Es gibt natürlich Konzepte, diese Möglichkeit einzuräumen.
- Eine Möglichkeit ist, entsprechende Methoden, die öffentlich sichtbar (also **`public`**) sind, bereitzustellen, z.B. eine Methode

```
  * Setzt/ver&auml;ndert die x-Koordintane des Punkts auf den spezifizierten Wert.
  * @param xCoord   die neue x-Koordinate des Punkts
  */
public void setX(double xNew) {
     xCoord = xNew;
}
```

- Solche Methoden werden auch *Setter* (-Methoden) genannt.

Setter-Methoden vs. Konstruktoren

- Machen Setter-Methoden in unseren Klassen Sinn?
- Eher nicht: die Werte der Punkte und Rechtecke sollten einmal gesetzt werden, danach aber nicht mehr änderbar sein (außer durch die Methoden, die von der Anwendung dafür vorgesehen sind, also Verschieben implementieren).
- Was dann?
- Tja, es wäre doch irgendwie schön, wenn wir die Attribute der Objekte gleich bei ihrer Erzeugung *richtig* initialisieren könnten.

Konstruktoren

- Diese Möglichkeit bieten die *Konstruktoren*.
- Konstruktoren sind spezielle Methoden, die zur Erzeugung und Initialisierung von Objekten aufgerufen werden können.
- Konstruktoren sind Methoden ohne Rückgabwert (nicht einmal `void`), die als Methodenname den Namen der Klasse erhalten (case-sensitive!!!).
- Konstruktoren können eine beliebige Anzahl von Eingabe-Parametern besitzen und überladen werden (d.h. sie heißen alle gleich, haben aber eine unterschiedliche Signatur).

Konstruktoren

- Beispiel: ein Konstruktor für die Klasse `Punkt`

```
public class Punkt {
     // Attribute
     private double xCoord;
     private double yCoord;

     // Konstruktor
     /**
       * Konstruktor zur Erzeugung eines Punkts mit zwei Koordinaten.
       * @param xCoord   die x-Koordinate des neuen Punkts.
       * @param yCoord   die y-Koordinate des neuen Punkts.
       */
     public Punkt(double xCoord, double yCoord) {
          this.xCoord = xCoord;
          this.yCoord = yCoord;
     }

     // Methoden
     ...
}
```

- Das Schlüsselwort **`this`** bezeichnet dabei eine “Selbstreferenz” (siehe später) und dient hier zur Unterscheidung der Instanz- und Eingabe-Variablen.

Verwendung

- Der Konstruktor kann nun verwendet werden, um einen neuen Punkt mit Koordinaten $(3.3, 2.7)$ zu erzeugen:

```
public class Test {
     public static void main(String[] args) {
          ...
          Punkt p = new Punkt(3.3,2.7);
          ...
     }
}
```

- Dabei wird eine Variable `p` im Stack angelegt mit einer Referenz auf die Halde.
- Dort liegen die Zettel der Instanzvariablen `xCoord` und `yCoord` jeweils mit den Werten `3.3` und `2.7`.
- Wie gesagt, unser Zustandsmodell gilt weiterhin, letztlich sind alle Variablen “nur” Zettel.

Konstruktoren

- Wie in allen Methodenrümpfen darf man natürlich auch im Rumpf des Konstruktors auf die Attribute (Instanzvariablen) zugreifen.
- Tatsächlich bezieht der Compiler zunächst alle Variablen ohne Punktnotation auf das Objekt selbst.
- Mit *Objekt selbst* ist natürlich eine Referenz gemeint und diese Referenz steht in einer speziellen Referenzvariable: **`this`**.
- Diese Referenz wird implizit an alle nicht-statischen Objektmethoden übergeben.
- Eine beliebige Variable `x` wird daher implizit als **`this`**`.x` interpretiert, außer es handelt sich um eine lokal vereinbarte Variable.
- Mit **`this`** lassen sich insbesondere Namenskonflikte zwischen Instanzvariablen und Eingabevariablen lösen.

Konstruktoren

- Analog ein Konstruktor für die Klasse `Rechteck` mit Verwendung der **`this`**-Referenz:

```
public class Rechteck {
     \\ Attribute
     private Punkt origin;
     private double xExt;
     private double yExt;

     \\ Konstruktor
     public Rechteck(Punkt origin, double xExt, double yExt) {
          this.origin = origin;
          this.xExt = xExt;
          this.yExt = yExt;
     }

     \\ Methoden
     ...
}
```

- Die Verwendung von **`this`** ist sinnvoll: man hebt grundsätzlich hervor, dass es sich um eine Instanzvariable, und nicht um eine lokale Variable (oder Eingabevariable) handelt.

Verwendung

- Mit den beiden Konstruktoren können wir nun Objekte bei ihrer Erzeugung gleich auf die gewünschte Art initialisieren:

```
public class Test {
     public static void main(String[] args) {
          Rechteck r;
          r = new Rechteck(new Punkt(1.0,1.0), 5.2, 3.5);
          r.verschiebe(3.3,2.7);
     }
}
```

- In `r = new Rechteck(new Punkt(1.0,1.0), 5.2, 3.5);` erzeugt `new Punkt(1.0,1.0)` zunächst den Punkt $(1.0, 1.0)$, der direkt in den Konstruktor von `Rechteck` übergeben wird (genauer an die entspr. Eingabevariable des Konstruktors: `origin`).
- Der gesamte Ausdruck erzeugt ein Rechteck mit Ankerpunkt $(1.0, 1.0)$ und Ausdehnung 5.2 bzw. 3.5.

Konstruktoren

- Wird kein expliziter Konstruktor deklariert, gibt es einen *Default-Konstruktor*, der keine Eingabeparameter hat, um überhaupt ein Objekt zu erzeugen (den Default-Konstruktor haben wir übrigens bisher benutzt: `Rechteck()` und `Punkt()` und auch bei den Record-Typen — deshalb haben wir damals schon immer Funktionsklammern dahinter geschrieben, ist ja eine spezielle null-stellige Methode).
- Wird mindestens ein expliziter Konstruktor vereinbart, steht in java *kein* Default-Konstruktor zur Verfügung!!!
- Möchte man trotz anderer Konstruktoren auch einen Konstruktor ohne Eingabeparameter zur Verfügung stellen, muss dieser explizit programmiert werden.

Konstruktoren

- Während die Methoden eine wohldefinierte Schnittstelle zur Veränderung von Objektzuständen zur Verfügung stellen, stellen die Konstruktoren eine wohldefinierte Schnittstelle zur Erzeugung von Objekten dar.
- Neben den Konstruktoren gibt es noch die Möglichkeit, initiale Attributwerte in der Klassendefinition direkt zu vereinbaren (Die Attributsdefinitionen sehen dann so aus wie Variablenvereinbarungen mit Initialisierung):

```
public class Punkt {
     private double xCoord = 1.0;
     private double yCoord = 1.0;
     ...
}
```

- Dies ist hier aber offenbar nicht sinnvoll (sondern nur, wenn ein Standardwert für einzelnen Attribute angegeben werden kann, der sich auch später selten oder gar nicht ändert).

Setter-Methoden vs. Konstruktoren

- Nochmal: wann machen Setter-Methoden grundsätzlich Sinn und wann nicht?
- Wenn die Attribute der Objekte gleich bei ihrer Erzeugung initialisiert werden können und auch sollen und später die Werte nicht mehr direkt verändert werden sollen (außer durch andere Methoden), ist es vernünftig entsprechende Konstruktoren bereitzustellen und auf Setter-Methoden zu verzichten (das wäre in unserer Anwendung bei Punkt und Rechteck vermutlich der Fall, kann aber in anderen Anwendungen ganz anders aussehen).
- Wenn es erwünscht ist, dass sich die einzelnen Attribute nach Erzeugung nochmal ändern können, muss man bei ordentlich gekapselten Klassen natürlich Setter-Methoden bereitstellen, unabhängig von der Bereitstellung eigener Konstruktoren.

Kurze Bemerkung

- Das zeigt uns natürlich etwas die Grenzen der Wiederverwendbarkeit auf:
- Eine sauber gekapselte Klasse müsste so implementiert werden, dass alle möglichen Benutzungen durch die bereitgestellten Methoden ermöglicht werden.
- Das ist natürlich meist etwas schwierig, insbesondere, weil man eine Klasse meist im Kontext einer speziellen Problemstellung schreibt und daher eine ganz spezielle Anwendung im Kopf hat.
- Aber wenigstens für diese Anwendung muss die Klasse natürlich alles notwendige“können”, d.h. entsprechende Methoden bereitstellen.
- Das Design der Klasse richtet sich dementsprechend nach diesen Anforderungen.

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;
  }

  ...
}
```

Nochmal der Sinn von Kapselung

- Wenn sonst nur noch Getter vereinbart werden, ist die Veränderung des Zustands 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.

Zustandsänderungen

- Stellt sich eigentlich nur noch die Frage, wie diese Zustandsänderungen funktionieren, also was z.B. passiert, wenn die Methode `abheben` aufgerufen wird.
- Wir hatten z.B. oben den Fall

```
public class Test {
     public static void main(String[] args) {
          ...
          Rechteck r;
          r = new Rechteck(new Punkt(1.0,1.0), 5.2, 3.5);
          ...
          Punkt p = new Punkt();
          r.verschiebe(3.3,2.7);
     }
}
```

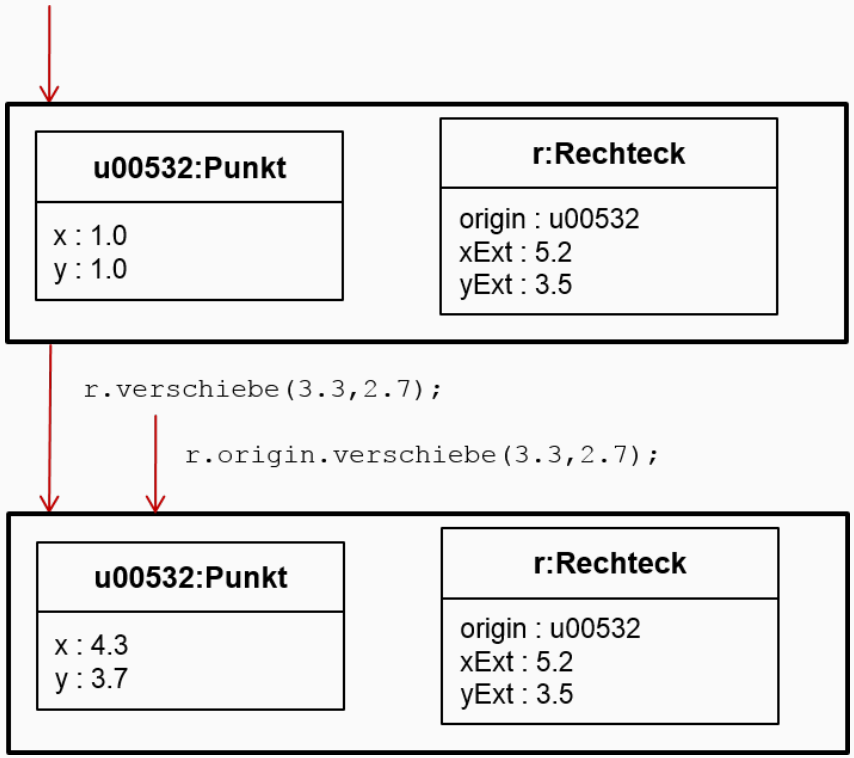
Der Aufruf `r.verschiebe(3.3,2.7);` soll offenbar das Rechteck `r` entsprechend verschieben.

Zustandsänderungen

- Der Aufruf `r.verschiebe(3.3,2.7);` ist tatsächlich ein ganz normaler Methoden-Aufruf, so wie wir ihn früher formalisiert haben.
- Die Parameter werden call-by-value übergeben (in unserem Beispiel nicht relevant, es werden nämlich direkt Literale übergeben).
- Allerdings wird zusätzlich noch der Zettel `r` "übergeben", damit klar ist, von welchem Objekt der Zustand geändert wird.

Zustandsänderungen

- Die Methode `r.verschiebe(3.3,2.7);` ruft wiederum `verschiebe(3.3,2.7);` für das Attribut `origin` vom Typ `Punkt` auf:



(u00532 soll eine Referenz auf die Halde darstellen)

Zustände

- An diesem Beispiel sehen wir noch einmal:
- Letztlich bleibt unser Zustandsbegriff, den wir im vorherigen Kapitel formalisiert haben, 1-zu-1 erhalten.
- Zusätzlich zu den Zuständen der primitiven Variablen, haben wir jetzt zusätzlich die Zustände der Objektvariablen (Variablen mit einem Objekttyp) bzw. allgemein Referenzvariablen (Arrays!).
- Der Zustand eines Objekts ergibt sich aus den Zuständen seiner Instanzvariablen (Attribute).

Zustände

- In unserem Beispiel war also der Zustand vom Objekt, das wir mit der Objektvariable `r` bezeichnen, definiert durch den Wert (Zustand) der Instanzvariablen `origin`, `xExt` und `yExt`.
- Während `xExt` und `yExt` wieder klassische Zettel mit (primitiven) Werten ((`xExt`, 5.2) und (`yExt`, 3.5)) darstellen, ist `origin` wiederum ein Objekt (Typ `Punkt`), dessen Zustand von seinen eigenen Instanzvariablen abhängt.
- Diese sind `x` und `y` und es gilt (`x`, 1.0) und (`y`, 1.0).

Zustände

- D.h. wir könnten unser bisheriges Zustandsmodell einfach weiter verwenden und wie folgt für Variablen mit Referenztypen erweitern:
- Sei T ein Referenztyp mit Struktur $T \subseteq T_1 \times \ldots \times T_n$ mit Instanzvariablen $t_1 : T_1, \ldots, t_n : T_n$.
- Wird eine Variable v vom Typ T vereinbart, dann wird wie gewohnt ein Paar (v, ω) zum bisherigen Zustand $\mathcal{S}$ hinzugefügt.
- Wird die Variable mit einem Objekt $o \in T$ initialisiert, wird aus diesem Zettel $(v, ref(o))$.
- $ref(o)$ bezeichnet die Referenz auf $o = ((t_1, d_1), \ldots, (t_n, d_n))$ wobei die d_i die Werte der Instanzvariablen t_i von o bezeichnen (und selbst wieder Referenzen sein können).

Zustände

- Beachten Sie aber bei dieser Formalisierung, dass auf den Zetteln (lokale Variable/formaler Parameter/Instanzvariable) also entsprechend entweder
  - der tatsächliche Wert (wenn die Variable einen primitiven Typ hat), bzw.
  - eine Referenz (wenn die Variable einen Referenztyp hat)

  steht!!!
- Wie besprochen führt dies zu Effekten bei Methodenaufrufen, Kopieren und Testen auf Gleichheit.

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 jederzeit 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).*

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 Attribut möglicherweise einen anderen Wert.

Zusammenspiel von imperativen und oo Konzepten

- Die Bank wünscht sich nun: für jedes neueröffnete Konto wird eine neue, fortlaufende Kontonummer 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

  /* ** Konstruktor (immer nicht-statisch)** */
  public Konto(String kundenName)
  {
    this.kundenName = kundenName;
    kontoStand = 0.0;
    KONTO_NR = aktuelleKNR;
    aktuelleKNR++;
  }

  /* ** Methoden (statisch und nicht-statisch) ** */
  ...
}
```