Überblick

Ein Beispiel

- Zurück zu unserem Beispiel.
- 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

Ein Beispiel

- Also gut, wir hatten schon gesehen, dass wir zwei Klassen *Rechteck* und *Punkt* dafür definieren wollen.
- Dazu müssen wir uns überlegen, wie diese Klassen genau aussehen sollen, also welche Attribute und welche 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

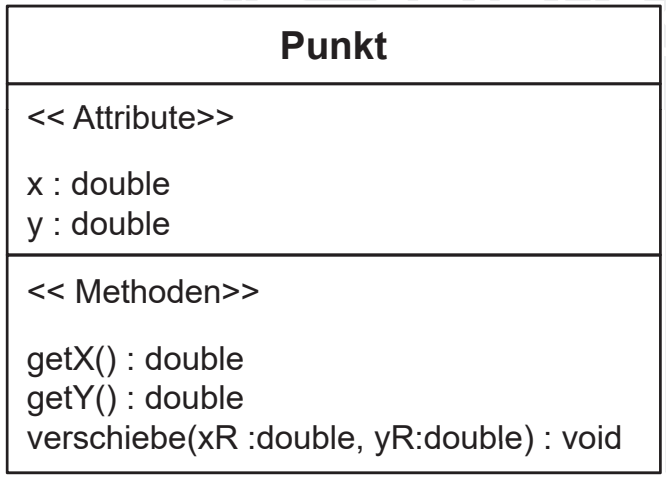
- Für die Darstellung des oo Entwurfs gibt es die Konzeptsprache *Unified Modelling Language* (UML).
- UML ist eine Art Pseudo-Code, der allerdings eine wohl-definierte Semantik besitzt und von vielen Programmen verarbeitet werden kann (z.B. können Implementierungsdetails z.B. in Java-Notation angegeben 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 UML-*Klassendiagramme* verwenden, die den statischen Entwurf (Klassen, Objekte und deren Beziehungen zueinander) konzeptionell beschreiben.
- Realisierungsdetails werden meist mit anderen Diagrammtypen beschrieben.
- Einen tieferen Einblick in UML erhalten Sie in den Vorlesungen zur Software-Entwicklung.

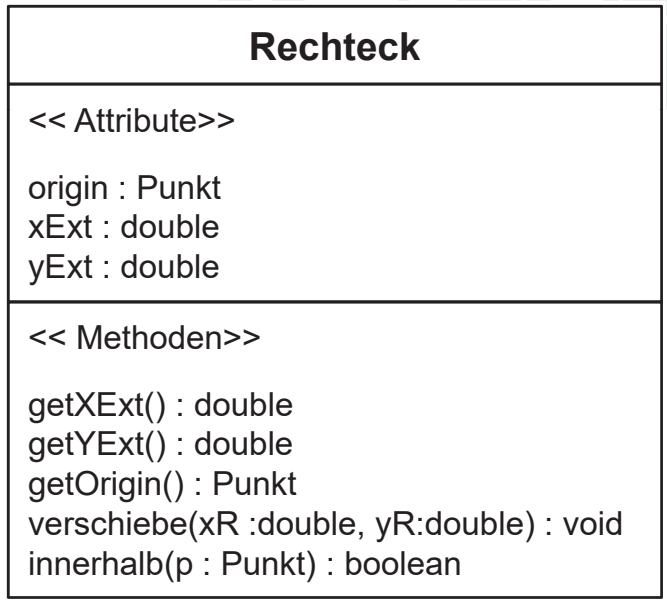
Entwurf von Rechteck und Punkt

- 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:



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).
- 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:



Die Klasse `Punkt`

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

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

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

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

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;
     }

     ...
```

Die Klasse Rechteck

```
    ...

    /** 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:

```
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

- Anweisung `Rechteck r;`: Klassische Deklaration einer Variablen vom Typ der Klasse (Objekttyp).
- Anstelle eines primitiven Datentyps wird hier 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

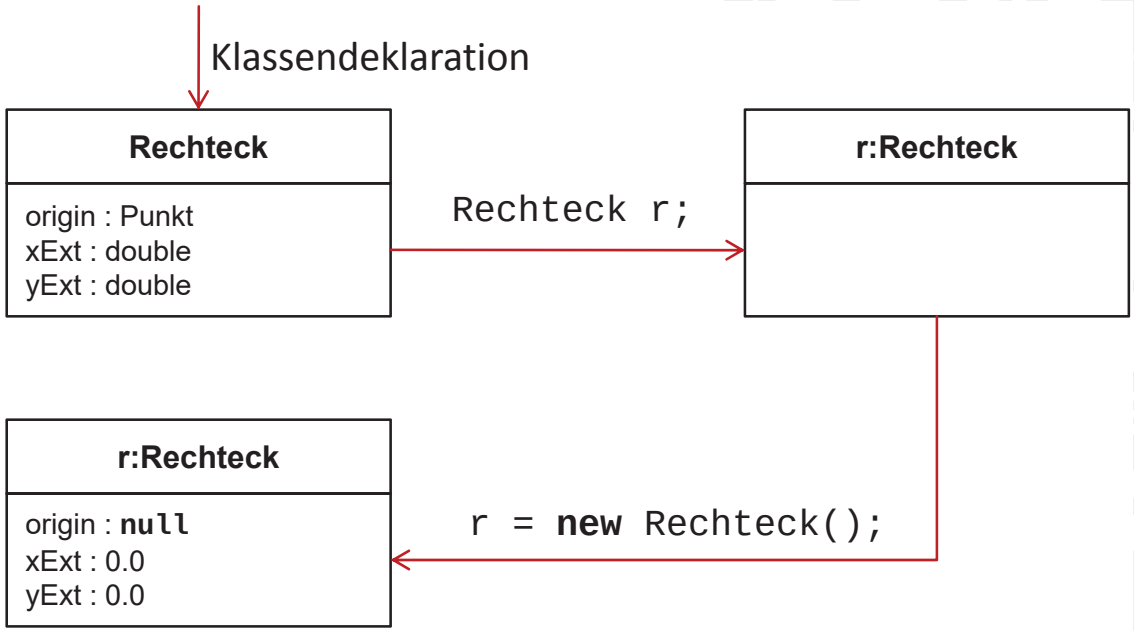
- Nach der Generierung eines Objekts haben alle Attribute mit primitiven Datentypen zunächst ihre entsprechenden Standardwerte (siehe Arrays).
- Attribute mit Objekttypen haben den Standardwert **`null`**, die *leere* Referenz.
- Zugriff auf Zustand / Methoden eines Objekts:
  - **`private`**: Auf diese 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.

oo Programmieren mit Stil

- 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 schon wieder Schall und Rauch???
- Mitnichten. Es gibt natürlich Konzepte, diese Möglichkeit einzuräumen, sonst wäre ja die gesamte oo Idee *ad absurdum* geführt.
- 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 xCoord) {
     x = xCoord;
}
```

- 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.
- 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`

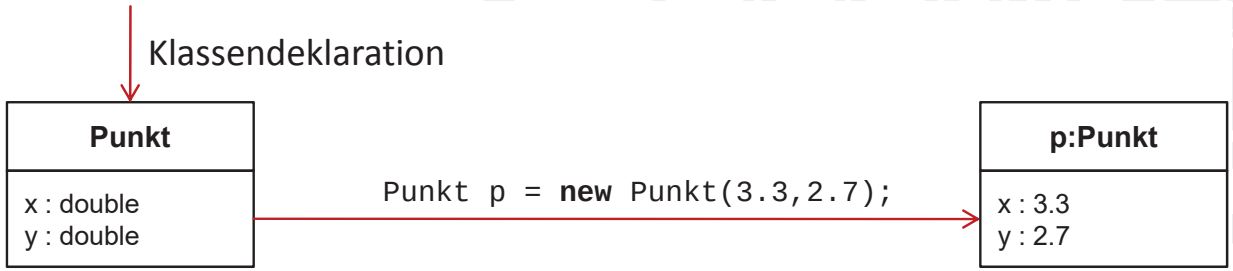
```
public class Punkt {
    // Attribute
    private double x;
    private double y;

    // 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) {
        x = xCoord;
        y = yCoord;
    }

    // Methoden
    ...
}
```

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);
          ...
     }
}
```

- Bildlich:



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 den Zugriff auf eine Instanzvariable, und nicht eine lokale Variable (oder Eingabevariable) handelt.

Verwendung

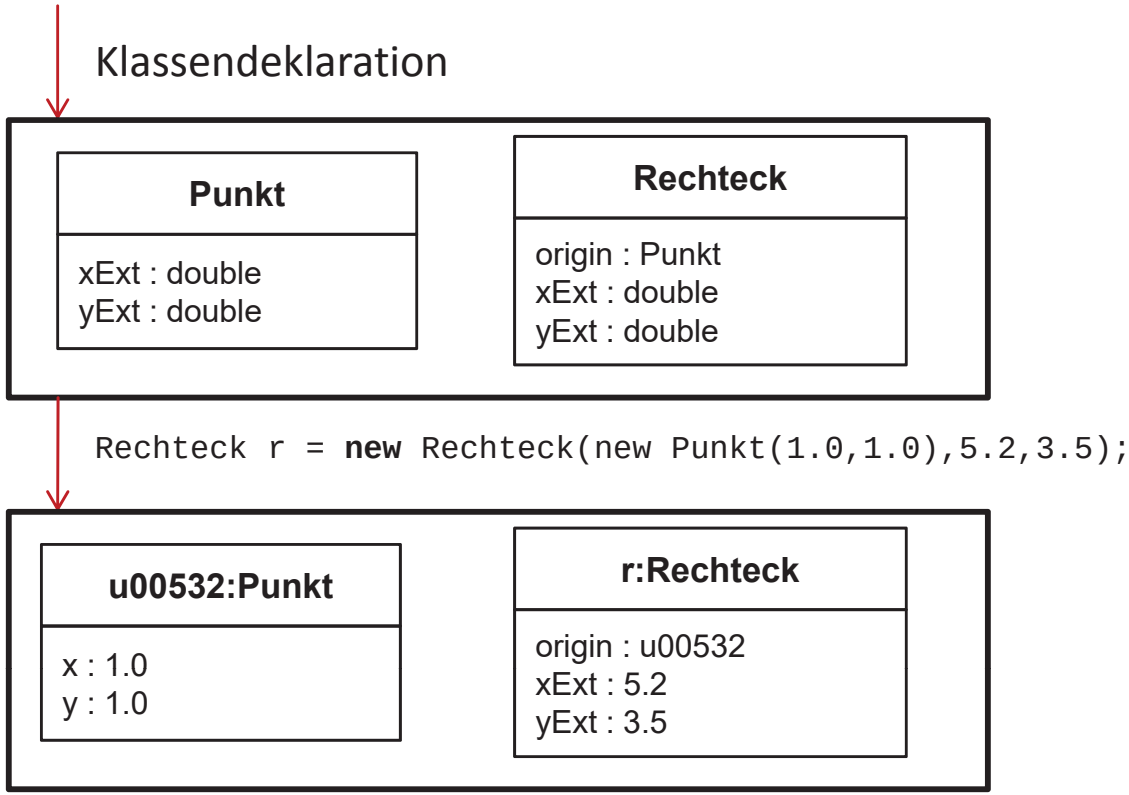
- Mit den beiden Konstruktoren können wir nun Objekte bei ihrer Erzeugung gleich auf die gewünschte Art initialisieren:

```
public class Test {
     public static void main(String[] args) {
          ...
          Rechteck r;
          r = new Rechteck(new Punkt(1.0,1.0), 5.2, 3.5);
          ...
          Punkt p = new Punkt();
          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 einen Punkt $(1.0, 1.0)$, der direkt in den Konstruktor von `Rechteck` übergeben wird.
- Der gesamte Ausdruck erzeugt ein Rechteck mit Ankerpunkt $(1.0, 1.0)$ und Ausdehnung 5.2 bzw. 3.5.

Verwendung

- Bildlich:



(u00532 symbolisiert eine interne ID, z.B. eine Speicheradresse)

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()` in unserer ersten `main`-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 x = 1.0;
     private double y = 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 zu verzichten.
- Wenn es erwünscht ist, dass sich die einzelnen Attribute nach Erzeugung nochmal ändern können, muss man bei ordentlich gekapselten Klassen natürlich Setter bereitstellen.
- Letzter Fall ist generell unabhängig von der Bereitstellung eigener Konstruktoren.

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);
     }
}
```

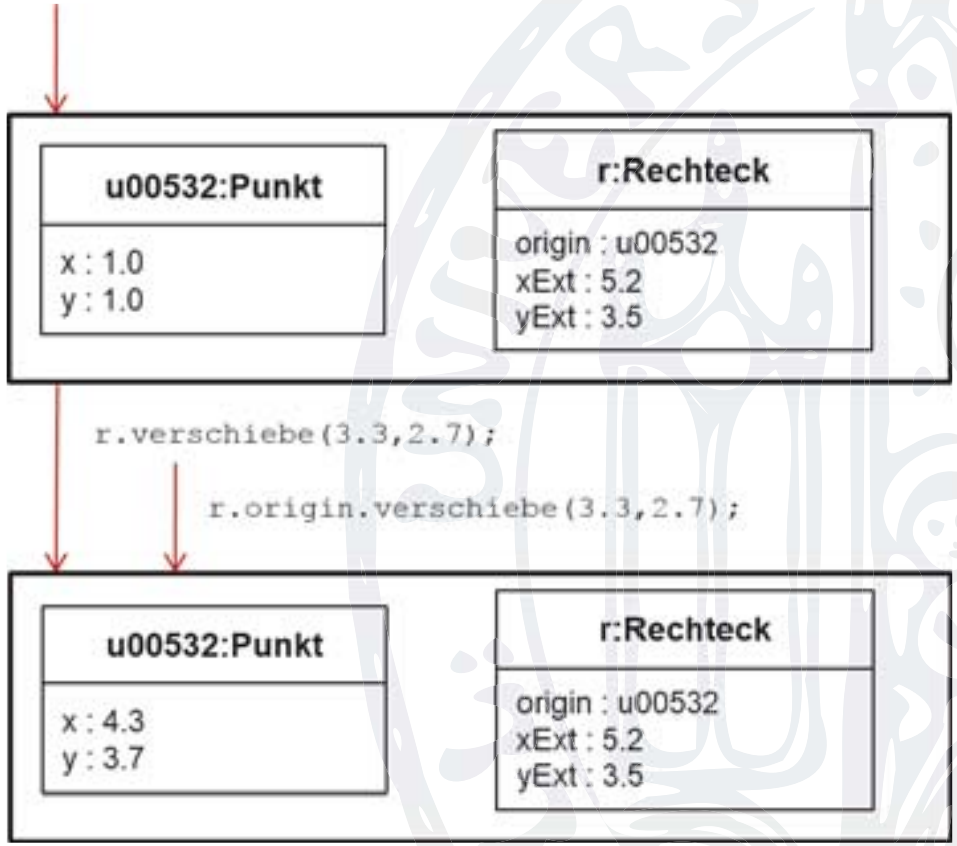
Der Aufruf `r.verschiebe(3.3,2.7);` soll offenbar das Rechteck `r` entsprechend verschieben.

Zustandsänderungen

- Der Aufruf `r.verschiebe(3.3,2.7);` ist tatsächlich ein ganz normaler Methoden-Aufru, 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 `verschiebe(3.3,2.7);` für das Attribut `origin` vom Typ `Punkt` auf:



Zustände

- Was wir an diesem Beispiel gesehen haben:
- 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 (und später nochmal genauer beleuchtet) führt dies ggfs. zu noch unbekannten Effekten.

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); die Klasse `java.io.PrintStream` stellt wiederum statische Methoden `PrintStream.print` und `PrintStream.println` zur Verfügung, deren Wirken wir ja schon kennen.

Zusammenspiel von imperativen und oo Konzepten

- Das Attribut `kontoStand` der Klasse `Konto` ist dagegen ein Objekt-Attribut.
- Es existiert nur dann, wenn es mindestens ein Objekt der Klasse `Konto` gibt.
- Für jedes existierende Objekt der Klasse `Konto` existiert dieses Attribut.
- Für jedes unterschiedliche Objekt der Klasse `Konto` hat dieses 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) ** */
  ...
}
```

Beispiel: Wrapper-Klassen für primitive Typen

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

Beispiel: Wrapper-Klassen für primitive Typen

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

Beispiel: Wrapper-Klassen für primitive Typen

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

```
Integer i1 = new Integer(123);
Integer i2 = new Integer("123");
```

Beispiel: Wrapper-Klassen für primitive Typen

- Kapselung:
  - Der Zugriff auf den Wert des Objekts erfolgt ausschließlich lesend über entsprechende Methoden:

    z.B. `public int intValue()`
  - Die interne Realisierung (wie wird der Wert gespeichert) ist dem Benutzer verborgen.
  - Insbesondere kann der Wert des Objekts nicht verändert werden.
  - Beispiele:

```
int i3 = i1.getIntValue();
```

Beispiel: Wrapper-Klassen für primitive Typen

- Statische Elemente (u.a.):
  - Wichtige Literale aus dem enstrechenden Wertebereich, z.B. Konstanten:

    `public static int` `MAX_VALUE` bzw.

    `public static int` `MIN_VALUE`

    für den maximal/minimal darstellbaren **`int`**-Wert

    oder z.B. Konstanten:

    `public static double` `NEGATIVE_INFINITY` bzw.

    `public static double` `POSITIVE_INFINITY`

    für $-\infty$ und $+\infty$
  - Hilfsmethoden wie z.B.

    `static double` `parseDouble(String s)` der Klasse `Double` wandelt die Zeichenkette `s` in ein primitiven **`double`**-Wert um und gibt den **`double`**-Wert aus.