Überblick

6. Grundlagen der objektorientierten Programmierung
6.1 Abstrakte Datentypen: von Structures zu Klassen
6.2 Das objektorientierte Paradigma
6.3 Klassen und Objekte in Java

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

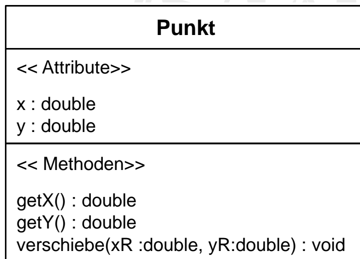
- Also gut, wir hatten schon gesehen, dass wir zwei Klassen *Rechteck* und *Punkt* dafür definieren wollen
- Dazu müssen wir uns überlegen, wie diese Klassen genau aussehen sollen, also welche Attribute und welche Methoden diese Klassen haben sollen
- Das Ergebnis dieser Überlegung ist ein *Modell* der raelen 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) die 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
- Im Rahmen dieser Vorlesung werden wir UML-*Klassendiagramme* verwenden, die den statischen Entwurf (Klassen, Objekte und deren Beziehungen zu einander) 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

- Damit zurück zu unserem Problem
- 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
- 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:

Rechteck
<< Attribute>> origin : Punkt xExt : double yExt : double
<< Methoden>> getXExt() : double getYExt() : double getOrigin() : Punkt verschiebe(xR :double, yR:double) : void innerhalb(p : Punkt) : boolean

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;
    }

    /** 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();`
- Da Arrays auch Objekte sind, wissen Sie nun auch, warum man bei der Erzeugung eines Arrays den `new`-Operator benötigt

Verwendung: Objekterzeugung

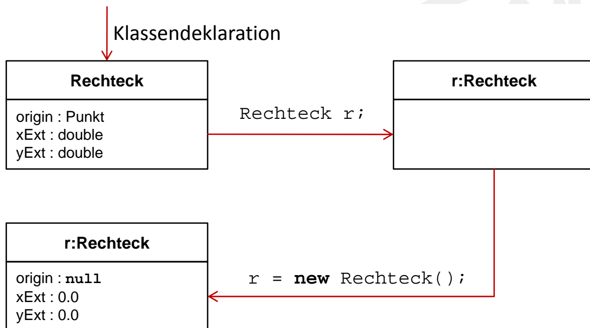
- Nach der Generierung eines Objekts haben alle Attribute mit primitiven Datentypen zunächst ihre entsprechenden Standardwerte (siehe Arrays)
- Attribute mit Objekttypen haben den Standardwert **`null`**, die *leere* Referenz
- Zugriff auf Zustand / Methoden eines Objekts:
  - **`private`**: Auf diese 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?
- Die Antwort hat damit zu tun, dass man bei der Sichtbarkeit nicht zwischen lesend/schreibend unterscheiden kann

Objekterzeugung

- Nochmal der Erzeugungsprozess bildlich:



- Jetzt stellt sich natürlich die Frage, wie man die Attributswerte des Rechteckst `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 könne wir die Attribute nicht explizit setzen
- Dann ist das Prinzip der Kapselung schon wieder Schall und Rauch???
- Mitnichten. Es 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 Wer
  * @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, dannach 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* intialisieren könnten

Konstruktoren

- Diese Möglichkeit bieten sog. *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`

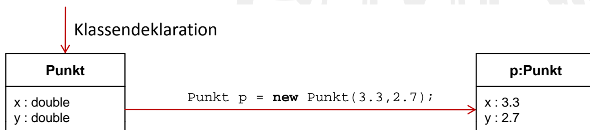
```
public class Punkt {
    // Attribute
    private double x;
    private double y;

    // 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 Audruck 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 *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 garnicht ändert)

Setter-Methoden vs. Konstruktoren

- Nochmal: wann machen Setter-Methoden grundsätzlich Sinn und wann nicht?
- Wenn die Attribute der Objekte gleich bei ihrer Erzeugung intialisiert 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