hhu.

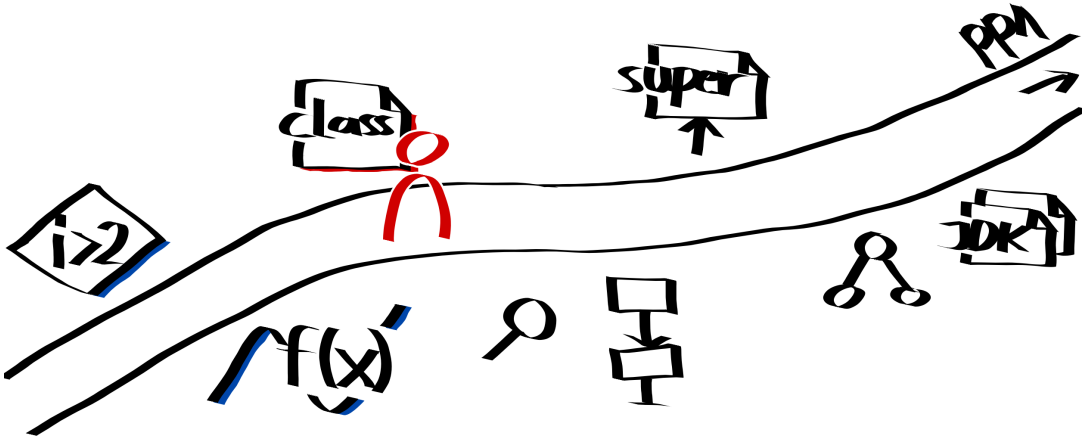
Kapitel 3: Objekte, Speicher & Klassen

VL 12: Interfaces & Polymorphie

Wintersemester 2023/24

Stand 20. November 2023, 07:32 Uhr

Wo stehen wir gerade?



Vorlesungsfeedback

Heinrich Heine Universität Düsseldorf

- Tempo: genau richtig–zu hoch
- letztes Mal viel Neues auf Einmal ⇒ heute und Mittwoch vermehrt Wiederholung/Beispiele; zu `this` bebildertes Beispiel auf Kursseite (W7)
- Scanner auch nochmal in VL-Beispielen benutzen ⇒ in den nächsten zwei Kapiteln
- Classroom hat bei <0,5% der Abgaben Schluckauf (keine Punktzahl/Testausgabe) ⇒ bei uns melden

? Dürfen in den Abgaben JDK-Methoden benutzt werden, die wir nicht in der Vorlesung hatten? Ja, aber nicht notwendig.

? Übungsblätter früher hochladen

Eine befreundete Physikerin . . .

Die Klasse Point ist sehr praktisch für meine Berechnungen mit Positionen von Punktladungen, aber ich finde es unerwartet, dass sich bei der Subtraktion immer der erste Punkt ändert.
Kannst du das nicht so machen, dass ich einen neuen Punkt zurückbekomme?

Die Kundin:

Super, danke.
Meine Kolleg:innen bekommen das mit der Übergabe von Point-Objekten nicht richtig hin und denken immer, sie arbeiten mit einer Kopie, wodurch es zu versehentlichen Änderungen kommt.
Könntest du vllt. ein einmal angelegtes Punkt-Objekt grundsätzlich vor Änderungen schützen?

```java
private static Point moveX(Point point, double xOffset) {
    Point newPoint = point;
    newPoint.setX(point.getX() + xOffset);
    return newPoint;
}
```

Unveränderliche Objekte[1]

→ Objekte, deren Instanzvariablen sich nach Konstruktoraufruf nicht mehr ändern können

Vorteile

- Vermeidung von versehentlichen Änderungen
- Schnelleres Finden von Fehlern, da nur an wenigen Stellen Änderungen möglich
- Weniger Verwirrung durch Stack vs. Heap
- Instanzen mit denselben Werten können denselben Bereich im Heap benutzen
- unproblematisch bei Nebenläufigkeit

Nachteile

- Anlegen von neuem Objekt bei Wertänderung

[1] Immutable Objects

Finale Klassen- und Instanzvariablen

Heinrich Heine Universität Düsseldorf

- Definition von Konstanten, geschützt vor (versehentlichen) Veränderungen
  - Wert in Variable geschützt (keine erneute Zuweisung möglich)
  - ! bei Objekt-Typen referenzierter Objekt-Inhalt (im Heap) *nicht* geschützt
- Klassenvariablen: Festlegung beim Programmieren
- Instanzvariablen: Festlegung beim Programmieren oder einmalig im Konstruktor

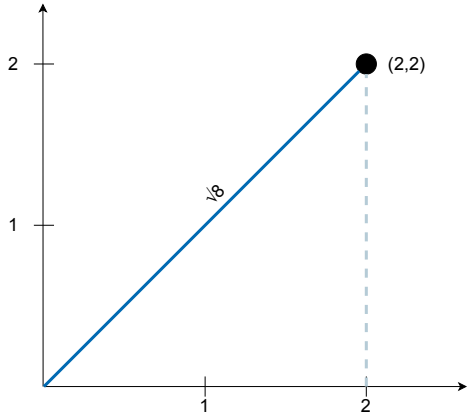
```
1  public class Product {
2      private static final String CURRENCY = "€";
3      private final String name;
4      private final double price;
5
6      public Product(String name, double price) {
7          this.name = name;
8          this.price = price;
9      }
10
11     public String toString() {
12         return name + ", " + price + " " + CURRENCY;
13     }
14 }
```

Die Kundin:

Nice.
Ich brauche auch öfter mal, wenn ich mit Punktladungen rechne, den Abstand eines Punktes zum Ursprung und den Winkel zur x-Achse.
Könntest du da bitte entsprechende Methoden hinzufügen?

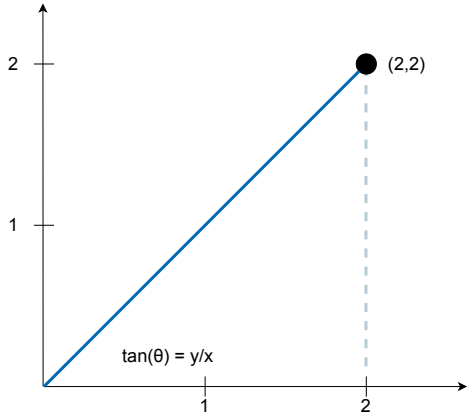
Radius

$$r = \sqrt{x^2 + y^2}$$



Winkel

$$\theta = \arctan2(y,x)$$



Die Kundin:

Cool, danke.
Da wir sehr oft Winkel und Abstand brauchen, hab ich die Klasse umgeschrieben, sodass sie nicht mehr x und y speichert, sondern r und θ. Du weißt schon, Polarkoordinaten und so.
Willst du dir das einmal anschaun?

Polarwas?

- zwei Möglichkeiten, einen Punkt in der Ebene anzugeben:
  - Kartesische Koordinaten: (x,y)
  - Polarkoordinaten: (r,θ)

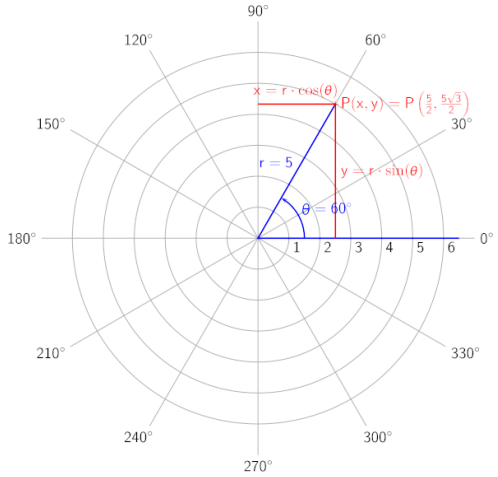


Bild: CC BY-SA 4.0 Wikimedia-User Carstenvogel

Die Kundin:

Okay, das mit den beiden Klassen ist zwar praktisch, aber jetzt kann ich gar nicht mehr von einem kartesischen Punkt einen Polarpunkt abziehen. Kannst du das nicht so machen, dass die kompatibel miteinander sind? Sind ja schließlich beides einfach Punkte.

Wohldefinierte Schnittstellen

„New Chair“ (https://www.flickr.com/photos/57447594@N00/4528611257), Brad.K, CC BY 2.0 (https://creativecommons.org/licenses/by/2.0/), Hintergrund entfernt

„LG-TV-Remote-Control_52738-480x360“ (https://www.flickr.com/photos/28958738@N06/4791222463), Public Domain Photos, CC BY 2.0 (https://creativecommons.org/licenses/by/2.0/), nachbearbeitet

„AUC65 (8049814095)“ (https://commons.wikimedia.org/wiki/File:AUC65_(8049814095).jpg), US Embassy Sweden, CC BY 2.0 (https://creativecommons.org/licenses/by/2.0/), Hintergrund entfernt

Was passiert, wenn Implementierung geändert?

Interfaces

hhu
Heinrich Heine
Universität Düsseldorf

Definition

Ein **Interface** definiert einen Datentyp und Signaturen öffentlicher Methoden. Eine Klasse kann Interfaces **implementieren**.

Interfaces selbst haben

- (meistens) keine Methoden-Implementierungen
- keine Instanz-Variablen
- keinen Konstruktor
  - → keine Instanzen von Interfaces anlegbar

Definition eines Interfaces

Heinrich Heine Universität Düsseldorf

Point.java

```java
public interface Point {

    double getX();
    double getY();

    void setX(int x);
    void setY(int y);

}
```

- im Wesentlichen nur Signaturen
- Methoden implizit public

Implementierung eines Interfaces

Heinrich Heine Universität Düsseldorf

```java
public class CartesianPoint implements Point {
    private double x;
    private double y;

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public void setX(int newX) {
        this.x = newX;
    }

    public void setY(int newY) {
        this.y = newY;
    }
}
```

Anmerkungen zur Implementierung von Interfaces

- Compiler prüft, ob alle Interface-Methoden implementiert
- Klasse darf mehr Methoden implementieren, als von Interface vorgegeben
- Implementierung mehrerer Interfaces möglich
  - Syntax: `implements Interface1, Interface2`

Polymorphie

- Objekt nicht nur Variablen mit Klassen-Typ zuweisbar, sondern auch Variablen mit Typ eines implementierten Interfaces
- Java ruft automatisch passende Methode in der Klasse auf, die zu den gespeicherten Objekt-Daten gehört

```java
public class Calculation {

    public static void main(String[] args) {
        Point center = new CartesianPoint(0, 0);
        Point position = new PolarPoint(Math.sqrt(2), Math.PI / 4);

        System.out.println(center.getX());
        System.out.println(position.getX());
    }

}
```

Gibt es hier ein Problem?

Heinrich Heine Universität Düsseldorf

```java
Point position = new PolarPoint(Math.sqrt(2), Math.PI / 4);

System.out.println(position.getTheta());
```

(Zur Erinnerung: `PolarPoint` definiert eine Methode `getTheta`)

Point.java

```java
public interface Point {

    double getX();
    double getY();

    void setX(int x);
    void setY(int y);

}
```

Gibt es hier ein Problem?

Heinrich Heine Universität Düsseldorf

```
4 Point position = new PolarPoint(Math.sqrt(2), Math.PI / 4);
5
6 System.out.println(position.getTheta());
```

(Zur Erinnerung: `PolarPoint` definiert eine Methode `getTheta`)

```
CalculationWrong.java:6: error: cannot find symbol
        System.out.println(position.getTheta());
                                   ^
  symbol:   method getTheta()
  location: variable position of type Point
1 error
```

→ Nur noch Methoden des Typs `Point` nutzbar (deklarierter Typ zählt, nicht Laufzeit-Typ)

Konstanten in Interfaces

Heinrich Heine Universität Düsseldorf

```java
public interface Produkt {

    public final static double STEUERSATZ = .19;

    public double getNettopreis();

}
```

- Nutzen: Teilen von Konstanten, die unabhängig von Implementierung sind
- Gehören zum Interface, nicht zu einem Objekt

Statische Methoden in Interfaces

Heinrich Heine Universität Düsseldorf

```java
public interface Produkt {

    public final static double STEUERSATZ = .19;

    public double getNettopreis();

    public static double getSteuersatz() {
        return STEUERSATZ;
    }

}
```

- Nutzen: Methoden, die unabhängig von konkretem Objekt sind
- Gehören zum Interface, nicht zu einem Objekt
- Kein Überschreiben (Ändern) durch Klasse möglich

Default-Methoden

```java
public interface Produkt {
    public final static double STEUERSATZ = .19;

    public double getNettopreis();

    public static double getSteuersatz() {
        return STEUERSATZ;
    }

    default public double getBruttopreis() {
        return getNettopreis() * (1 + getSteuersatz());
    }
}
```

- Nutzen: Erweiterung bestehender Interfaces, ohne alle implementierenden Klassen anzupassen
- Werden von Klasse automatisch übernommen oder können überschrieben (durch eigene Implementierung ersetzt) werden

Anwendungsbeispiel: Ereignisverarbeitung

Heinrich Heine Universität Düsseldorf

DragListener.java

```java
public interface DragListener {
    /**
    The listener interface for receiving mouse drag events.

    The class that is interested in processing a mouse being dragged can
    implement this interface. You can add a DragListener l to a Draw
    instance d by calling d.addDragListener(l).

    @param x current x position of the mouse
    @param x current y position of the mouse
     */
    public void mouseDragged(double x, double y);
}
```

benötigt Draw.java, DragListener.java und KeyboardListener.java aus dem Ilias

Lernziele

Heinrich Heine
Universität Düsseldorf

Sie können am Ende der Woche ...

- Interfaces nach vorgegebenen Spezifikationen **definieren**.
- ein vorgegebenes Interface in einer Klasse **implementieren**.
- Variablen mit Interface-Typ **verwenden**.

Wichtige Begriffe

Unveränderlichkeit Interface implements

Polymorphie

Übungsblatt

- Referenzen
- Pascalsches Dreieck
- Rechteck