Modellierung und Programmierung 1

Übung 6

Stefan Preußner

7./ 8. Dezember 2020

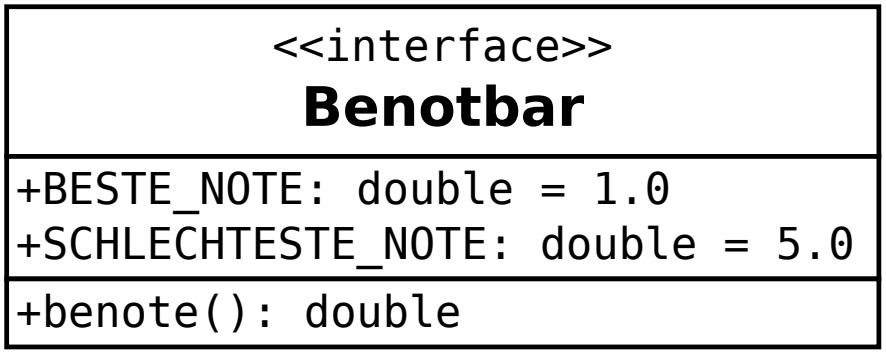
Generics

siehe `Paar.java` und `PaarMain.java`

Schnittstellen (Interfaces)

- In Java gibt es keine Mehrfachvererbung, eine Klasse kann also nicht von mehreren Klassen erben
- Soll eine Klasse mehrere Typen haben, so kann sie **Schnittstellen** - **Interfaces** - implementieren
- Eine Schnittstelle legt fest, welche Methoden eine Klasse besitzen muss, stellt aber selbst keine Implementierung zur Verfügung

Schnittstellen (Interfaces)




```java
public interface Benotbar
{
    double BESTE_NOTE = 1.0;
    double SCHLECHTESTE_NOTE = 5.0;

    double benote();
}
```

Schnittstellen (Interfaces)

- Von einer Schnittstelle können keine Objekte erzeugt werden, nur von den sie implementierenden Klassen
- Eine Schnittstelle darf deshalb keinen Konstruktor haben
- Die Methoden eines Interfaces sind automatisch `public` und `abstract`
  - `abstract` Methoden werden nur deklariert, aber nicht implementiert
  - Die Deklaration einer Schnittstellenmethode enthält nur Modifizierer, Rückgabetyp und Signatur

Variablen in Schnittstellen

- Instanzvariablen sind immer Teil einer Implementierung; da Schnittstellen keine Implementierung enthalten, besitzen sie auch keine Instanzvariablen
- In Schnittstellen können Konstanten festgelegt werden
  - Diese müssen `public`, `static` und `final` sein
  - Erweitert eine Schnittstelle eine andere Schnittstelle, so kann sie deren Konstanten mit eigenen Werten überschreiben

Implementierung

- Schlüsselwort `implements`

```
public class Klausur implements Benotbar
```

- Mehrere Interfaces können durch Kommata getrennt werden:

```
public class Klausur implements Benotbar, Comparable<Klausur>
```

Implementierung

- Eine Klasse **muss** alle Methoden eines Interfaces implementieren
  - Dies gilt nicht für Klassen, welche `abstract` sind (von abstrakten Klassen können keine Instanzen erzeugt werden, daher müssen sie keine Methoden implementieren)
  - Wird ein Interface erweitert, müssen alle Klassen, welche das Interface implementieren, entsprechend angepasst werden
- Implementierte Methoden **müssen** `public` sein

Schnittstellen und instanceof

`instanceof` funktioniert bei Schnittstellen wie bei Klassen. Im obigen Beispiel geben die Tests

```
Klausur mup;
mup instanceof Klausur;
mup instanceof Benotbar;
mup instanceof Comparable;
```

alle `true` zurück.

Array vs. ArrayList

- Arrays und ArrayLists sind beides Strukturen, welche Daten eines vorgegebenen Typs sequenziell speichern
- Arrays haben eine feste Größe; Operationen sind im wesentlichen auf auf das Auslesen und Setzen von Elementen und die Ermittlung der Größe beschränkt
- Die Größe von ArrayLists ist dynamisch, es gibt zahlreiche Funktionen zur Manipulation der Liste

- Die Klasse `ArrayList` muss aus `java.util` importiert werden:
  - `import java.util.ArrayList`

  oder
  - `import java.util.*`
- Eine neue, leere `ArrayList`, welche Objekte vom Typ `Datentyp` speichert, wird wie folgt erzeugt:

```
ArrayList<Datentyp> listenname = new ArrayList<Datentyp>();
```

- `List` ist eine Schnittstelle, welche Basismethoden zur Listenmanipulation zur Verfügung stellt und von zahlreichen Listenklassen implementiert wird (`ArrayList`, `LinkedList`, `Stack`, `Vector`, ...).
- Oft ist ein Deklaration als `List` anstelle einer `ArrayList` besser:

```
List<Datentyp> listenname = new ArrayList<Datentyp>();
```

(Grund: soll später der Datentyp von `listenname` geändert werden, weil eine andere Listenimplementierung von Java besser geeignet ist, muss nur diese eine Zeile im Code verändert werden)

Einige nützliche Funktionen einer ArrayList<Datentyp>:

- `add(Datentyp obj)` - fügt das Objekt `obj` am Ende der Liste ein
- `add(int index, Datentyp obj)` - fügt das Objekt `obj` an der angegebenen Stelle `index` ein
- `addAll(Collection<Datentyp> objs)` - fügt alle Objekte aus einer `Collection` (hierzu zählen auch ArrayLists) am Ende der Liste ein
- `removeAll(Collection<Datentyp> objs)` - entfernt alle Elemente, die in der Collection `objs` enthalten sind, aus der Liste

Einige nützliche Funktionen einer ArrayList<Datentyp>:

- `contains(Object obj)` - testet, ob die Liste das Objekt `obj` enthält
- `get(int index)` - liefert das Element an der Stelle `index` zurück
- `remove(int index)` - entfernt das Element an der Stelle `index` und gibt es zurück
- `remove(Object obj)` - entfernt das erste Element in der Liste, welches `obj` entspricht
- `set(int index, Datentyp element)` - ersetzt das Element an der Stelle `index` mit `element`

Einige nützliche Funktionen einer ArrayList<Datentyp>:

- `size()` - liefert die Länge der Liste
- `toArray()` - wandelt die ArrayList in ein Array `Datentyp[]` um
- `isEmpty()` - testet, ob die Liste leer ist
- `iterator()` - erzeugt einen Listeniterator

Interface `java.lang.Comparable`

- `java.lang.Comparable` ist nützlich, um Objekte leicht sortieren zu können
- Es muss lediglich die Funktion `compareTo` implementiert werden. `x.compareTo(y)` soll folgende Werte zurückgeben:
  - eine Zahl größer 0, wenn `x` in einer sortierten Liste nach `y` stehen soll
  - 0, wenn beide Objekte gleich(wertig) sind
  - eine Zahl kleiner 0, wenn `x` in der sortierten Liste vor `y` stehen soll

```
public class Klausur implements Comparable<Klausur>
{
    private double note;

    public int compareTo(Klausur query)
    {
        if (this.note < query.note)
        {
            return -1;
        }
        if (this.note == query.note)
        {
            return 0;
        }
        return 1;
    }
}
```

Die Sortierung kann dann mit `Collections.sort()` (oder einer der Sortierfunktionen aus den von `Collections` abgeleiteten Klassen) erfolgen:

```
List<Klausur> klausuren = new ArrayList<Klausur>();
klausuren.add(new Klausur( 1.3 ));
klausuren.add(new Klausur( 3.7 ));
klausuren.add(new Klausur( 2.0 ));

Collections.sort(klausuren);
```

Die Liste `klausuren` ist nun so sortiert, dass die Klausur mit der besten Note ganz vorne und die Klausur mit der schlechtesten Note ganz hinten in der Liste steht.

Assoziative Datenfelder / Dictionary / Map

- Interface: `java.util.Map`
- Beispiele für implementierende Klassen: HashMap, TreeMap
- Die Klassen repräsentieren *assoziative Datenfelder*: gespeichert werden Paare bestehend aus einem Schlüssel und einem Wert - mit jedem Schlüssel ist also ein Wert assoziiert
- Jeder Schlüssel darf in einer `Map` nur einmal vorkommen

HashMap

- **HashMap** speichert die Schlüssel-Wert-Paare in einer Hashtabelle, vom Schlüssel wird also der Hash berechnet
  - Für eigene Klassen müssen i.d.R. `equals()` und `hashCode()` überschrieben werden
  - Das Einfügen und Suchen von Schlüsseln erfolgt im Optimalfall in konstanter Zeit

TreeMap

- **TreeMap** speichert die Daten anhand des Schlüssels sortiert in einem balanzierten Binärbaum
  - Die Schlüssel müssen vergleichbar sein - bei eigenen Klassen muss wenigstens eins der Interfaces `Comparable` oder Comparator implementiert sein
  - In der Regel langsamer als HashMap

- Eine neue HashMap wird wie folgt erzeugt:

```
Map<K, V> variablenname = new HashMap<K, V>();

Map<String, Integer> meinemap = new HashMap<String, Integer>();
```

  - `K` ist der Datentyp des Schlüssels
  - `V` ist der Datentyp des gespeicherten Werts
- Ein neuer Wert wird mit `put` gespeichert:

```
Map.put(K, V);

meinemap.put("Hallo", 12345)
```

- Der zu einem Schlüssel gehörende Wert wird mit `get` zurückgegeben:

```
V variablenname = Map.get(K);

Integer meinint = meinemap.get("Hallo");
```

  - Ist ein Schlüssel nicht vorhanden, so wird `null` zurückgegeben
- Mit `containsKey` kann überprüft werden, ob ein Schlüssel in der Map vorkommt:

```
boolean Map.containsKey(K);

boolean meinboolA = meinemap.containsKey("Test");  // false
boolean meinboolB = meinemap.containsKey("Hallo"); // true
```

- Mit `containsValue` kann überprüft werden, ob ein Wert in der Map vorkommt:

```
boolean Map.containsValue(V);

boolean meinboolA = meinemap.containsValue(7890);  // false
boolean meinboolB = meinemap.containsValue(12345);  // true
```

  - Achtung: der Sinn von Maps ist es, anhand eines bestimmten Schlüssels schnell auf Daten zugreifen zu können. `containsKey` ist deshalb sehr schnell, `containsValue` dagegen sehr langsam!

- Die Menge aller Schlüssel lässt sich mit `keySet` zurückgeben:

```
Set<K> variablenname = Map.keySet();

Set<String> meinset = meinemap.keySet();
```

- Die Menge aller Schlüssel-Wert-Paare erhält man mit `entrySet()`:

```
Set<Map.Entry<K,V>> variablenname = Map.entrySet();

Set<Map.Entry<String,Integer>> meinset = meinemap.entrySet();
```

  - Über dieses Set kann iteriert werden, um nacheinander auf alle Schlüssel-Wert-Paare zuzugreifen
  - Map.Entry besitzt die Methoden `getKey()` und `getValue()` für den Zugriff auf den Schlüssel bzw. den Wert

Die Klassen Integer und Double

- `int` und `double` sind primitive Datentypen in Java
  - Primitive Datentypen sind keine Klassen, sie haben daher keine Attribute oder Methoden
- Durch die Klassen Integer und Long bzw. Float und Double können Ganzzahlen bzw. Gleitkommazahlen als Objekte dargestellt werden
- Die Erzeugung eines neuen Integer- bzw. Double-Objekts erfolgt wie bei anderen Klassen:

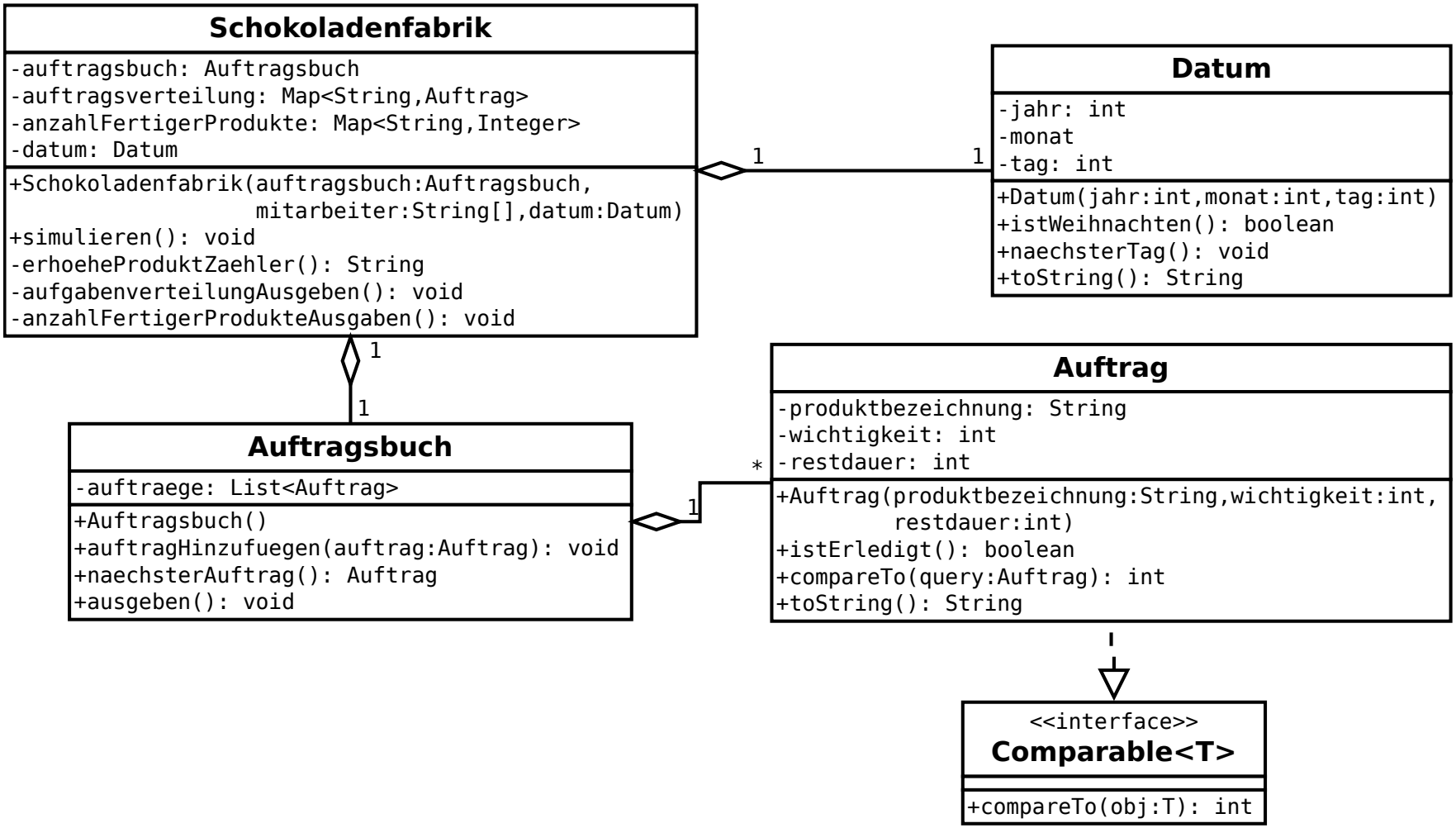
```
Double d = new Double(3.1);
Integer i = new Integer(13);
```

Java-Programmierung - List, Map, Comparable

Programmierübung:

Simulation der Produktion von Süßigkeiten in einer Schokoladenfabrik in den Tagen vor Weihnachten

Java-Programmierung - List, Map, Comparable



Java-Programmierung - List, Map, Comparable

Klasse `Datum`

- `istWeihnachten` gibt `true` zurück, wenn es der 24.12. eines beliebigen Jahres ist
- `naechsterTag` ändert das Datum auf den nächsten Tag und berücksichtigt dabei Monats- bzw. Jahreswechsel (aber keine Schaltjahre)
- `toString` gibt das Datum im Format Tag.Monat.Jahr zurück

Java-Programmierung - List, Map, Comparable

Klasse `Auftrag`

- Die Restdauer muss bei der Erzeugung eines neuen Auftrags mindestens 1 betragen
- Die Wichtigkeit muss einen Wert zwischen 0 und 3 haben
- `istErledigt` gibt `true` zurück, wenn die Restdauer 0 ist
- `compareTo` soll so implementiert werden, dass Aufträge nach absteigender Wichtigkeit sortiert werden. Bei gleicher Wichtigkeit soll nach absteigender Restdauer sortiert werden.
- `toString` gibt die Produktbezeichnung, Wichtigkeit und Restdauer als String zurück

Java-Programmierung - List, Map, Comparable

Klasse `Auftragsbuch`

- `auftragHinzufuegen` fügt den übergebenen Auftrag zur Liste aller Aufträge hinzu und sortiert die Liste anschließend
- `naechsterAuftrag` entfernt den ersten Auftrag aus der Auftragsliste und gibt ihn zurück. Ist die Auftragsliste leer, dann soll `null` zurückgegeben werden.
- `ausgeben` gibt alle Aufträge aus

Java-Programmierung - List, Map, Comparable

Klasse `Schokoladenfabrik`

- `simulieren` simuliert tagesweise den Zeitraum zwischen `datum` und Weihnachten
  - Jedem Mitarbeiter wird, solange das Auftragsbuch noch Aufträge enthält, ein Auftrag zugewiesen
  - Bei jedem Auftrag, der einem Mitarbeiter zugewiesen wurde, verringert sich die Restdauer jeden Tag um einen Tag
  - Ist ein Auftrag erledigt, wird dem Mitarbeiter ein neuer Auftrag zugewiesen und der Zähler für das entsprechende Produkt um 1 erhöht
  - Es werden täglich die Auftragsverteilung und Zahl der insgesamt fertiggestellten Produkte ausgegeben

Java-Programmierung - List, Map, Comparable

Klasse `Schokoladenfabrik`

- `auftragsverteilung` bildet jeden Mitarbeiter (repräsentiert durch seinen Namen) auf einen Auftrag ab
- `anzahlFertigerProdukte` gibt an, wie oft ein Produkt (repräsentiert durch seine Bezeichnung) bereits fertiggestellt wurde
- `erhoeheProduktZaehler` erhöht den Zähler für ein fertiggestelltes Produkt um 1
- `aufgabenverteilungAusgeben` gibt die aktuelle Aufgabenverteilung aus
- `anzahlFertigerProdukteAusgaben` gibt den Produktzähler aus