

# Ausnahmen und Schnittstellen

**Programmieren Tutorium Nr.12**

Aleksandr Zakharov | 13. Januar 2026

# Organisatorisches

## Übungsblatt 4

- Abgabe: 15.01.2026 (06:00 Uhr)

## Übungsblatt 5

- Ausgabe: 14. Januar 2026 ca. 12:00 Uhr
- Abgabe: 21.01.2026 (12:00 Uhr) – 29.01.2026 (06:00 Uhr)

## Präsenzübung

- 21. Januar 2026 17:30 Uhr - 19:00 Uhr

Orga  
●

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

If anything can go wrong, it will.  
- Murphy's Law

Orga  
○

Exceptions  
●○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○

# Exceptions

## Beispiel - GOTO-Zeitalter

```
1 w: ...
2   if (...) {
3     ...
4     if (...) { // Fehler
5       goto e
6     }
7     ...
8     if (...) { // Schleifenende
9       goto l
10    }
11  }
12  ...
13  goto w
14 l: ...
15  goto x
16 e: ... // Fehlerbehandlung
17 x:
```

Naja ...

- Schleife durch goto
- Fehlerbehandlung am Ende
- Schleifenbedingung mitten in Schleifen
- Komplizierter Kontrollfluss

# Exceptions

## Beispiel - ohne GOTO

```
1 boolean quit, err;  
2 while(!quit) {  
3     if (...) {  
4         ...  
5         if (...) { // Fehler  
6             quit, err = true;  
7         } else {  
8             ...  
9             if (...) { // Schleifenende  
10                quit = true;  
11            }  
12        }  
13    }  
14    if (!quit) {  
15        ...  
16    }  
17 }  
18 if (!err) {  
19     ...  
20 } else {  
21     ... // Fehlerbehandlung  
22 }
```

Naja ...

- Schleifenbedingung mitten in Schleifen
- Zahllose if-Abfragen
- Komplizierter Kontrollfluss
- kaum besser als vorher
- gotos sind böse und man sollte sie **nie** benutzen (Goto considered harmful)

# Exceptions

## Typische (lokale) Fehlerbehandlung

```
1 f = openFile("input.txt");
2 if (f < 0) {
3     System.out.println("Datei kann nicht geoeffnet werden");
4     System.out.println("Grund: %s", f);
5 } else {
6     // ...
7 }
```

### Probleme:

- Vermischung von Programmlogik und Fehlerbehandlung
- Fehlerausgaben im Programm verstreut
- Keine Trennung von Algorithmus und Benutzerinteraktion

# Exceptions

## Exception

- eine *Ausnahme*
- Zur Laufzeit des Programms
- Zur Unterbrechung des normalen Kontrollflusses

## Verwendung einer Exception

- Ein *Problem* tritt auf
- Normales Fortfahren nicht möglich
- Lokale Reaktion darauf nicht sinnvoll/möglich
- Behandlung des Problems *an anderer Stelle* nötig

Orga  
○

Exceptions  
○○○●○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

## Exceptions in Java

### Ausnahme in Java

- echtes Objekt (Methoden, Attribute, ...)
- Erzeugung mit new
- Auslösen mit throw

## Beispiel

```
1 public void setMonth(int month) throws IllegalArgumentException {  
2     if ((month < 1) || (month > 12)) {  
3         throw new IllegalArgumentException("Wrong month: " + month);  
4     }  
5     this.month = month;  
6 }
```

Orga  
○

Exceptions  
○○○○○●○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

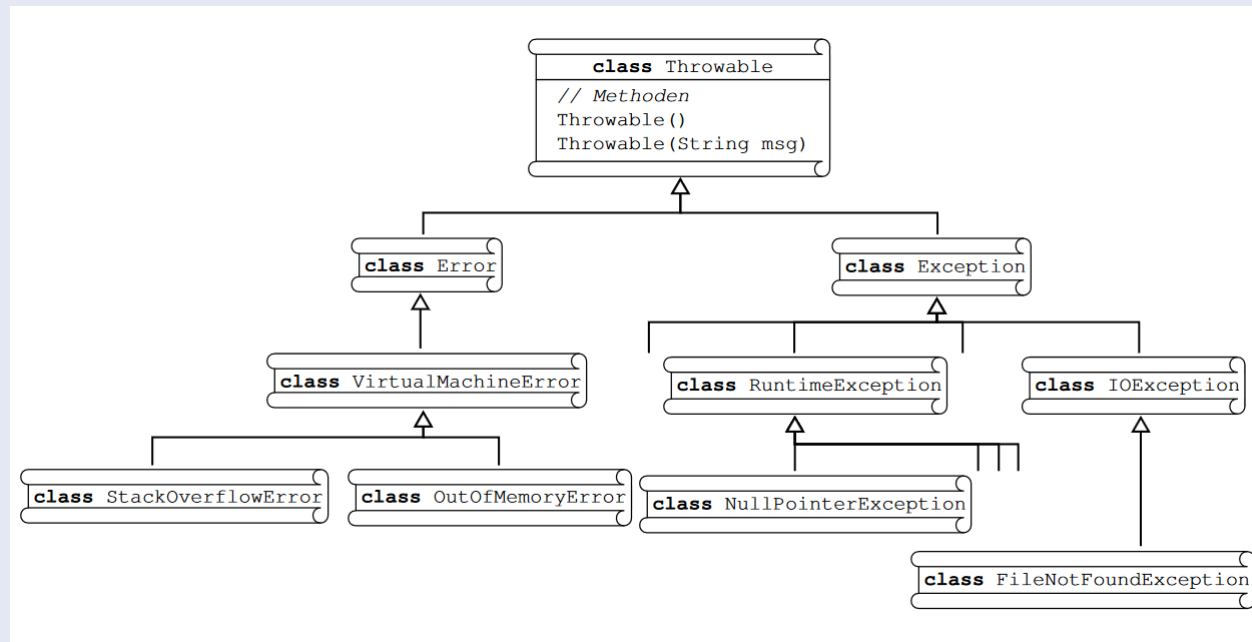
Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○



# Exceptions

## Auszug aus der Exceptionhierarchie



# Exceptions

## Exceptions in Java

- Typ (Klasse) der Ausnahme/Exception
- Von Klasse `Exception` abgeleitet
- Mindestens zwei Konstruktoren: `Default` & mit `String`-Parameter (mit zusätzlichen Informationen)
- Methoden: `getMessage()` & `printStackTrace()`

## Error

- Neben `Exception` existiert `Error`
- Schwerwiegende Fehler
- Keine sinnvolle Behandlung möglich

Orga  
○

Exceptions  
○○○○○○●○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○

# Exceptions

## Ausnahmebehandlung in Java

```
1 try {  
2     // Hier koennte eine Exception auftreten  
3 } catch (ExceptionType1 e) {  
4     // Fehlerbehandlung fuer ExceptionType1  
5 } catch (ExceptionType2 e) {  
6     // Fehlerbehandlung fuer ExceptionType2  
7 }
```

## Fall-through, die Zweite

- Java ruft den **ersten** passenden catch Block auf!
- Alle weiteren werden *ignoriert*

# Exceptions

## Beispiel

```
1 try {  
2     FileReader fr = new FileReader(".test");  
3     int nextChar = fr.read();  
4     while (nextChar != -1) {  
5         nextChar = fr.read();  
6     }  
7 } catch (FileNotFoundException e) {  
8     System.out.println("Nicht gefunden.");  
9 } catch (IOException e) {  
10     System.out.println("Ooops.");  
11 }
```

Orga  
○

Exceptions  
○○○○○○○○○●○○○○○○○○○○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

## Exception Handler

- Behandlung einer Ausnahme
- an einer Stelle
- irgendwo im Aufrufstack
- getrennt von normalen Programmcode

## Catch or specify

Jede ausgelöste geprüfte (checked) Exception muss

- behandelt (Exception Handler) oder
- deklariert (throws)

werden.

Orga  
○

Exceptions  
○○○○○○○○○○●○○○○○○○○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

## Deklaration von Ausnahmen

- Deklaration im Methodenkopf: `private String readFile(String filename) throws IOException, FileNotFoundException {}`
- Aufrufer muss sich um Exception kümmern
- `throws` ist Teil der Signatur (Vorsicht beim Überschreiben)  
Exceptions können in überschriebenen Methoden weggelassen werden, aber nicht hinzukommen.
- Nicht deklarationspflichtig sind `RuntimeException` & `Error` (sowie deren Unterklassen)

*Anmerkung: Da `IOException` Oberklasse von `FileNotFoundException` ist, müsste letzteres nicht extra deklariert werden. Dokumentationszwecke!*

# Exceptions

## Ort der Behandlung

Finden der passenden Ausnahmebehandlung:

- Suche im Aufrufstack nach umgebenden try - catch-Blöcken, gehe zu erstem passenden catch-Block
- Nach der Behandlung: Fortsetzung am Ende des try - catch-Block

Orga  
○

Exceptions  
○○○○○○○○○○○○●○○○○○○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

## Error

(Katastrophale) Probleme, die eigentlich nicht auftreten dürfen. Speicher voll, Illegaler Byte-Code, JVM-Fehler, ...

## RuntimeException

Durch *fremde* Fehler erzeugte Probleme. Falsche Benutzung einer Klasse, Programmierfehler.

## Geprüfte Exception (*checked Exception*)

Vorhersehbare und behandelbare Fehler. Datei nicht vorhanden, Festplatte voll, Fehler beim Parsen, ...

Orga  
○

Exceptions  
○○○○○○○○○○○○○○●○○○○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○



# Exceptions

## Behandlung?

- **Error und Unterklassen:** Nein, nicht sinnvoll behandelbar
- **Exception:** Nein, viel zu allgemein
- **RuntimeException:** Prinzipiell Nein
- **Unterklassen dessen:** Programmierfehler beheben! (Ausnahme: `NumberFormatException`)
- **Andere:** Ja, wenn sinnvoll behandelbar

# Exceptions

## Werfen?

- **Error:** Ja, ggf. mit eigener Unterklasse
- **Exception:** Niemals, nur als eigene Unterklasse
- **RuntimeException:** Ja, eigene (semantisch passende) Unterklasse

## Beispiel

```
1 if ((month < 1) || (month > 12)) {  
2     throw new IllegalArgumentException("Wrong month: %s", month);  
3 }  
4 switch (month) {  
5     case 1: break; // ...  
6     default: throw new Error();  
7 }
```

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○●○○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

## Eigene Exceptions

- Ableiten einer eigenen Unterklasse von `Exception` oder `RuntimeException`
- Implementierung der zwei Standard-Konstruktoren
- Definition einer eigenen, sinnvollen Exception-Hierarchie (bei Bedarf)
- Verwendung von vorhandenen Exceptions nur für dafür vorgesehene Zwecke (Javadoc anschauen)

## Beispiele in der Java-API

- `IllegalArgumentException`
- `IllegalStateException`
- `UnsupportedOperationException`
- `NullPointerException`

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○●○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

## Verwendung

Exceptions sollen ...

- zur Vereinfachung dienen
- die absolute Ausnahme darstellen
- mittels @throws im Javadoc beschrieben werden
- NICHT den normalen Kontrollfluss steuern

## Böse!

```
1 try {  
2     while (character != array[i]) { i++; }  
3 } catch (Exception e) {  
4     System.out.println("Element nicht gefunden!");  
5 }
```

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○●○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

## Verboten!

- try-Block um das ganze Programm
- Leerer catch-Block
- Explizites Fangen des Typs Exception
- Explizites Fangen des Typs Throwable

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○●○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

## Frühe Fehlererkennung

- Kleinerer Suchbereich beim Debugging
- Abbruch/Fehlermeldung bei inkonsistentem Programmzustand
- Häufig: null oder negative Zahlen
- Defensive Programmierung
- Zusätzliche Abfragen im Programmtext

# Exceptions - Zusammenfassung

## Ausnahmen

- werden ausgelöst (throw) und behandelt (try-catch) oder deklariert (throws)
- sollen die Ausnahme bleiben
- trennen sauber Programmlogik und Fehlerbehandlung

## Fehlererkennung

- so früh wie möglich
- defensiv
- mittels if & Exceptions

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○●○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

## Aufgabe

Schreiben Sie eine eigene Exception `InputException`, die bei falschen Benutzereingaben geworfen werden soll. Dazu erbt sie von der Exception `IllegalArgumentException`.

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○●○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○



# Exceptions

## Mögliche Lösung - Teil 1

```
1 package edu.kit.informatik.exception;
2
3 /**
4  * The {@code InputException} is thrown in case of invalid or inappropriate user
5  * input.
6  *
7  * @author upgcv
8  * @version 1.0
9  */
10
11 public class InputException extends IllegalArgumentException {
12
13     /**
14      * Constructs a new {@code InputException} without a detail message.
15      */
16     public InputException() {
17         super();
18     }
19 }
```

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○●○

Interfaces  
○○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Exceptions

## Mögliche Lösung - Teil 2

```
1  /**
2   * Constructs a new {@code InputException} with a specified detail message.
3   *
4   * @param message detailed error message
5   */
6  public InputException(final String message) {
7      super(message);
8  }
9  }
```

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○●

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Interfaces

## Spiele!

```
1 class Movable {  
2     public void moveTo(Location location) {...}  
3 }  
4 class Damageable {  
5     public void damage(int amount) {...}  
6 }
```

## Spiele - Ergibt das so Sinn? Was fällt auf?

- Mit einem Array von `Movables` kann man alle bewegen, mit einem Array von `Damageables` kann man AOE-Effekte auf alle anwenden
- Es macht aber Sinn, die beiden zu trennen! Manche Gegner sind vielleicht unbesiegbar und Gebäude kann man vielleicht zerstören.
- Jede Klasse modelliert hier eine **Eigenschaft** oder ein bestimmtes Verhalten

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
●○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Interfaces

## Ein NPS

```
1 class NPC extends Damageable, Movable {...}
```

Was ist hier falsch?

## Lösung: Interfaces!

```
1 interface Movable {  
2     public void moveTo(Location location) {...}  
3 }  
4 interface Damageable {  
5     public void damage(int amount) {...}  
6 }  
7  
8 class NPC implements Damageable, Movable {...}  
9 class Building implements Damageable {...}
```

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○●○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Interfaces - Ganz kurz

## Die Idee

- Interfaces abstrahieren gewisse Eigenschaften und Verhaltensweisen **über Klassen hinweg!**

## Benennung

- Im Beispiel: -able für Eigenschaften/Verhalten
- Allgemein wie Klassen (sinnvoll :))

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○●○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Interfaces

## Die Idee

Interfaces **abstrahieren von der tatsächlichen Implementierung** einer Klasse.

## Merke

Ein **Interface** ...

- ist eine Sammlung von Methodenköpfen ohne Rümpfe
- legt die Methodennamen und Parametertypen fest
- macht keine Annahme über die Implementierung der Methoden
- kann selbst zur Typisierung verwendet werden (ist damit Datentyp für Objekte)
- wird von einer Klasse implementiert, indem für jede Methode des Interface eine Implementierung vorgenommen wird

# Interfaces - Syntax

## Syntax

### ■ Deklaration

```
1 interface Name { Konstanten und Methodenkoepfe }
```

### ■ Implementierung

```
1 class Klassenname implements Interface1, Interface2, Interface3, ... { ... }
```

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○●○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Interfaces - Zusammenfassung

## Zusammenfassung

- Eine Klasse kann mehrere Interfaces gleichzeitig implementieren
- Alle Methoden aller Interfaces müssen in der implementierenden Klasse implementiert werden
- Syntax für Methodenköpfe im Interface: *Rückgabetyp Methodenname(Parameterliste);*
- Interfaces gehören (normalerweise) wie Klassen in eigene Dateien (Schlüsselwort *interface* statt *class*)
- Interfaces haben keine Objekte (keine Instantiierung mit *new*!)
- Methodenaufruf nutzt jeweils die speziellste implementierte Methode
- Alle Methoden und Konstanten sind automatisch **public**
- Alle Konstanten sind zwingend `public static final`

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○●

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○



# Comparable

Eine Klasse, die Comparable implementiert, vergleicht ein Objekt mit sich selbst.  
Der Rückgabewert ist -1, 0, 1 und nicht boolean.

## Beispiel

```
1 public class Point implements Comparable<Point> {  
2     // ...  
3     public int compareTo(Point p) {  
4         // ...  
5     }  
6 }  
7  
8 // Sort according to Point's compareTo()-method  
9 Collections.sort( list );
```

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
●○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Comparator

Hingegen eine Klasse, die Comparator implementiert, zwei externe Objekte miteinander vergleicht.

## Beispiel

```
1 public class PointCompare implements Comparator<Point> {
2     // ...
3     public int compare(Point p1, Point p2) {
4         // ...
5     }
6 }
7
8 // Sort according to PointCompare's compare()-method
9 PointCompare pointCompare = new PointCompare();
10 Collections.sort( list , pointCompare);
```

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
○●

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○○

# Abstrakte Klassen & Methoden

## Allgemein

Mit dem Schlüsselwort `abstract` kann man eine Klasse als „reine Oberklasse“ deklarieren

- Abstrakte Klassen können nicht direkt instanziiert werden
- Keine oder unvollständige Implementierung
- Methoden können ebenfalls als `abstract` deklariert werden, wenn sie in der abstrakten Klasse (noch) nicht implementiert werden
- Eine Klasse mit abstrakten Methoden muss abstrakt sein
- Unterschied zu Interfaces?

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
○○

Abstrakt  
●○

Aufgaben: Schnittstellen  
○○○○○○○

# Abstrakte Klassen & Methoden

## Beispiel

```
1 public abstract class Fruit {  
2     // Das geht auch in Interfaces!  
3     public abstract void consume();  
4 }  
5  
6 public class Orange extends Fruit {  
7     @Override  
8     public void consume() {  
9         System.out.println("Peel first , then eat.");  
10    }  
11 }
```

- Die Frucht kann nicht direkt instanziiert werden
- consume wird nicht implementiert, nur die Signatur

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
○○

Abstrakt  
●○

Aufgaben: Schnittstellen  
○○○○○○○

# Aufgabe

## Aufgabe 1 - Teil 1

Für einen Nährwertrechner sei folgendes Interface gegeben:

```
1 public interface Essbar {  
2     int brennwert();  
3     double eiweiss();  
4     double kohlenhydrate();  
5     double fett() ;  
6 }
```

Überlegen Sie sich einige Klassen, die dieses Interface implementieren könnten.  
Wie müsste die Implementierung dieser Klassen aussehen?

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
●○○○○○○

# Aufgabe 1 - Teil 1

## Lösungsvorschlag

```
1 public class Milch implements Essbar {
2     public int brennwert() {
3         return 70;
4     }
5     public double eiweiss() {
6         return 3.3;
7     }
8     public double kohlenhydrate() {
9         return 4.7;
10    }
11    public double fett() {
12        return 4.2;
13    }
14 }
15
16 public abstract class Pflanze { ... }
17 public class Paprika extends Pflanze implements Essbar { ... }
```

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○●○○○○○

# Aufgabe 1 - Teil 2

## Aufgabe 1 - Teil 2

Warum ist es für diesen Zweck sinnvoll, ein Interface einzusetzen? Wäre die Umsetzung auch mittels Vererbung möglich?

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○●○○○○

# Aufgabe 1 - Teil 2

## mögliche Erklärung

Die entsprechenden Klassen sollen dieselbe Schnittstelle (die durch das Interface implementierten Methoden) zur Verfügung stellen.

Dies mittels Vererbung zu realisieren ist jedoch nicht sinnvoll bzw. möglich, da sich die essbaren Objekte (z.B. 'Schweinesteak' und 'Paprika') grundlegend unterscheiden können und Mehrfachvererbung in Java nicht möglich ist.

Vergleiche hierzu die Vererbungshierarchie in Abbildung 1:

Schwein und Paprika sind essbar, während dies für Kreuzotter und Efeu nicht der Fall ist (zumindest gemeinhin nicht öfters als einmal). Da Java keine Mehrfachvererbung unterstützt, ist es somit nicht möglich, 'Schwein' und 'Paprika' mittels einer gemeinsamen Oberklasse als essbar zu kennzeichnen.

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○●○○○



# Aufgabe 1 - Teil 2

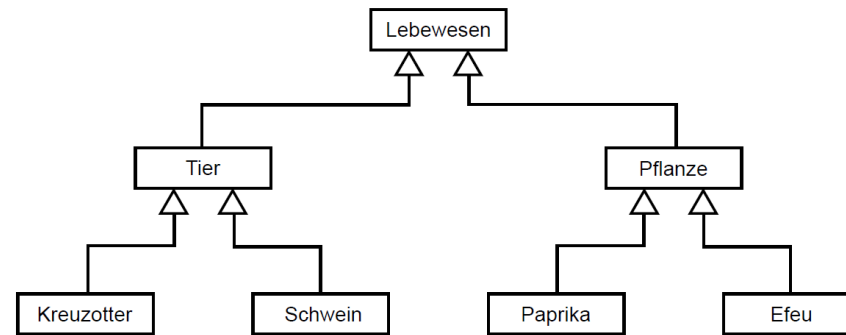


Abbildung 1: Vererbungshierarchie

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○●○○

# Aufgabe 1 - Teil 3

## Aufgabe 1 - Teil 3

Erstelle Instanzen von verschiedenen Ausprägungen des Interfaces, speichere sie in einem gemeinsamen Array und gebe für alle das Fett aus.

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○●○

# Aufgabe 1 - Teil 3

## Mögliche Lösung

```
1 public static void main(String... args) {  
2     Essbar[] essbares = new Essbar[3];  
3     essbares[0] = new Milch();  
4     essbares[1] = new Schweinesteak();  
5     essbares[2] = new Paprika();  
6     for (Essbar e : essbares) {  
7         System.out.println(e.fett());  
8     }  
9 }
```

Orga  
○

Exceptions  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Interfaces  
○○○○○○

Comparable | Comparator  
○○

Abstrakt  
○○

Aufgaben: Schnittstellen  
○○○○○○●