

PROGRAMMIEREN 2

Teil 1

2025

Silke Jandl
Bernhard Fuchs

Zu meiner Person

- IT & Wirtschaftsinformatik
- Studium an TU Graz
- Aktuell
 - ▶ Hauptberuflicher Lektor am Campus02
 - ▶ Seit 2003 Software Engineer in Baubranche
 - ▶ DevOps



Seite 2

INHALT (1)

| Lehrveranstaltungsdetails | | |
|--|--|---------------------------------------|
| Informationsportal | Fachbereich Wirtschaftswissenschaften | Fachbereich Wirtschaftswissenschaften |
| Programmieren 2 | PR2 | Integrierte Lehrveranstaltung |
| Lehrveranstaltungstyp (EN) | Lehrveranstaltungstyp | Unterrichtsprüfung (EN) |
| Lehrveranstaltungstyp (DE) | Pflichtveranstaltung | Deutsch |
| Verfasser (EN) | Lehrveranstaltung | |
| 25 | 1 | |
| Lehrveranstaltung (EN) | Grundvorlesung | Grundvorlesung |
| BE, AH, VO, BNE, OOP, PR2, 1 | 9 | 2025 |
| Lehrinhalte und Lernziele | | |
| Lehrinhalte | Gründliche | Fachliche |
| Grundlagen der OOP | Nach positiver Absolvierung der Lehrveranstaltung sind die Studierenden in der Lage... | Grundlagen der OOP verstehen |
| Klassen, Interfaces, Vererbung und Polymorphie | Klassen, Interfaces, Vererbung und Polymorphie verstehen und in Praxisbeispielen anwenden | |
| Arrays und Collections | Arrays und Collections in Verbindung mit Klassen verwenden können | |
| Generics-Klassen | Eigene Generics-Klassen erstellen und verwenden können | |
| Testautomatisierung | Grundlagen der Testautomatisierung zusammenfassen können, um Tests theoretisch beschreiben und anwenden können | |
| Spezielle Interfaces | Spezielle Interfaces zur Sortierung kennen lernen und anwenden | |

INHALT (2)

| Leistungsbeurteilung der Lehrveranstaltungen | |
|---|---|
| Die Leistungsbeurteilung erfolgt auf Lehrveranstaltungsebene | |
| | Notenskala sehr gut (1), gut (2), befriedigend (3), genügend (4), nicht... |
| Teilleistung | Gewichtung |
| Klausurarbeit (schriftlich oder am PC) Präsenzauftritt | 100 % Mindestanforderung Prüfungsgespräch 3. Antritt |
| Total | 100 % |
| Weitere Informationen weitere Informationen zum 1. Antritt: Prüfung (100 Prozent - schriftlich oder am PC) | |

TERMINE

0004VD1004 PROGRAMMIEREN 2 (51UE IL, SS 2024/25)

| Gruppe ▾ | | | | | | | | | |
|----------------|------------|-------|-------|-------|----------------|--------|-----------------------------|---------------|---------------------------------|
| Tag | Datum | von | bis | Ort | Ereignis | Termin | Lehrer | Vortragende*r | Anmerkung |
| Standardgruppe | | | | | | | | | |
| Fr | 21.03.2025 | 08:15 | 12:15 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Do | 27.03.2025 | 08:15 | 15:30 | CZ004 | Abhaltung | fix | Jandl, Silke | | |
| Di | 01.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | Jandl, Silke | | |
| Mo | 07.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Di | 08.04.2025 | 13:30 | 15:30 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Do | 10.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Mo | 28.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Mi | 30.04.2025 | 08:15 | 11:45 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Mi | 30.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Mi | 14.05.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Mo | 26.05.2025 | 08:15 | 12:15 | CZ004 | Prüfungstermin | fix | | | Teilergebnisse: "Klausurarbeit" |

PRÜFUNGSMODALITÄTEN

Für einen positiven Abschluss der Lehrveranstaltung gibt es 3 Antrittsmöglichkeiten:

- 1. ANTRITT:** z.B. (schriftl./mündl Hauptklausur + E-Learning Übung)
- 2. ANTRITT:** z.B. (schriftl./mündl. Nachklausur + E-Learning Übung aus 1. Antritt)
- 3. ANTRITT:** kommissionelle Prüfung (zB schriftlich und mündlich)

Für diese 3 Antritte stehen 4 Termine zur Verfügung.

PROGRAMMIEREN 2 TEIL 1

- Constants
- Enums
- Wrapper Klassen
- Generics
- Collections

KONSTANTE

- Werte die während der Laufzeit nicht geändert werden können
 - Wird an zentraler Stelle deklariert (ganz oben)
 - Konvention: uppercase snakecase / constant case (Großbuchstaben mit Unterstrichen)
- static final int DAYS_IN_A_YEAR = 365;

CODING: KONSTANTEN

AgeCalculator

- ❖ Wir bauen eine Klasse AgeCalculator die uns berechnet wie alt jemand in Monaten, Wochen und Tagen ist.
- ❖ Dafür brauchen wir 3 Konstanten (DAYS_IN_A_YEAR, WEEKS_IN_A_YEAR, MONTHS_IN_A_YEAR)
- ❖ Wir brauchen außerdem 3 Methoden die jeweils den Parameter *int age* nehmen (ageInDays, ageInWeeks, ageInMonths).

CODING: KONSTANTEN

AgeCalculator

- ❖ Wir werden auch eine Klasse mit main Methode brauchen
- ❖ In dieser Klasse fragen wir mittels Scanner nach dem Alter in Jahren (*int age*)
- ❖ Dann schreiben wir das Alter in Monaten, Wochen und Tagen in die Konsole

ENUMS

- ❖ Aufzählungstyp
- ❖ Spezieller Datentyp
- ❖ Jeder Wert ist eine Konstante
- ❖ Jeder Wert hat eindeutigen Namen
- ❖ Logische Gruppierungen: wartbarer & lesbarer

ENUMS

```
1 package Intro;
2
3 5 usages
4 public enum MediaType {
5     1 usage
6     BOOK, FILM, SERIES, OTHER;
7 }
```

- Class (or inner class)
- String values

ENUMS OpeningHours OpeningVariation

- Als erstes brauchen wir die Klasse OpeningVariation
- Sie sollte die folgenden 2 Werte haben: MONTOFRI and TUETOFRI

ENUMS OpeningHours OpeningHours

- Als nächstes brauchen wir eine Klasse OpeningHours mit drei Feldern (String openingTime, String closingTime, OpeningVariations openingVariations) und einen Konstruktor
- Diese Klasse hat die Methode getOpeningHoursToday(DayOfWeek dayOfWeek), die einen String mit der Information ob heute (von wann bis wann) geöffnet ist, oder nicht, zurück gibt

ENUMS OpeningHours Shop

- Als nächstes brauchen wir eine Klasse Shop mit 2 Feldern (name, openingVariation)
- Getter für beide Felder
- Einen Konstruktor

ENUMS OpeningHours Main

- In unserer Main Klasse machen wir uns einen oder zwei Shops
- Wir holen uns die Öffnungszeit für heute und geben sie aus

CODE REVIEW

- ❖ Lest euch nochmal den Code durch
- ❖ Kommentiert und/oder macht Notizen
- ❖ Sammelt alles was noch nicht ganz klar ist

PROGRAMMIEREN 2

Bernhard Fuchs
Silke Jandl

2025

PR2

Erster Halbttag

27.03.2025

silke.jandl@lv.campus02.at
<https://github.com/08jandl>
<https://www.meetup.com/de-DE/women-plus-tech-graz>



2023-

wealthpilot

2020-2023



2022-



Part of Clarivate

2023-

NAME GAME

Mein Name ist Silke...

- ... und ich mag Spinat aber ich esse niemals Selchfleisch
- ... und ich war noch nie in Sizilien oder Sibirien
- ... und meine Mutter ist aus Südamerika

QR-Code

<https://dal.campus02.at/admin/courses>

Programmieren 2 Teil 1

- ▣ Wrapper Klassen
- ▣ Generics
- ▣ Collections

Wrapper Klassen

- ▣ Erlaubt dort primitive Datentypen zu verwenden, wo eigentlich Objekte erwartet werden; z.B.: beim Arbeiten mit Generics
- ▣ Wrapper Klassen können null sein, primitive Datentypen nicht
- ▣ Wrapper Klassen bringen eigene Methoden mit

Jeder primitive Datentyp hat eine entsprechende Wrapper Klasse

boolean -> Boolean
byte -> Byte
short -> Short
int -> Integer
long -> Long

float -> Float
double -> Double
char -> Character

Wrapper Klassen

- Mit Autoboxing und Unboxing kann man zwischen primitive Datentypen und ihren Wrapper Klassen hin- und herkonvertieren

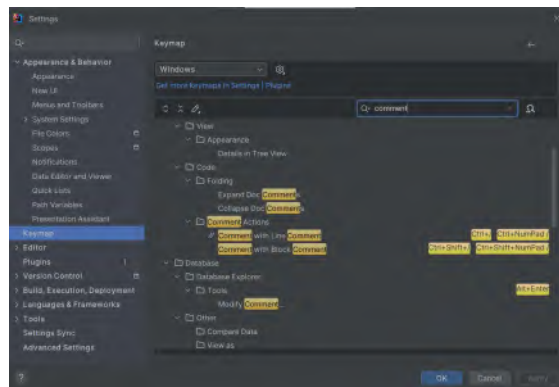
Autoboxing:

Integer autoboxedInt = 123;

Auto-unboxing:

int autounboxedInt = autoboxedInt;

IntelliJ exkurs: Keymap



Data Validator

Es gibt ein Programm für Studierende, bei dem sie unter bestimmten Umständen teilnehmen können. Mit Hilfe eines Scanners wollen wir herausfinden, ob das Alter, der Notendurchschnitt, und das Interessensfeld passen.

Wahr

Falsch

Konstanten speichern Werte die während der Laufzeit noch geändert werden können

Wahr

Falsch

Konstanten speichern Werte die während der Laufzeit noch geändert werden können

Wahr

Falsch

So sieht eine richtig deklarierte Konstante aus:

```
static final int DAYS_IN_A_YEAR = 365;
```

Wahr

Falsch

So sieht eine richtig deklarierte Konstante aus:

```
static final int DAYS_IN_A_YEAR = 365;
```


Wahr

Falsch

Konstanten werden immer ganz unten als letztes in
einer Klasse deklariert

Wahr

Falsch

Konstanten werden immer ganz unten als letztes in
einer Klasse deklariert

Wahr

Falsch

Enums werden dazu verwendet große Datenmengen
zu speichern.

Wahr

Falsch

Enums werden dazu verwendet große Datenmengen
zu speichern.

Wahr

Falsch

Enums sind ein spezieller Datentyp, ein
Aufzählungstyp.

Wahr

Falsch

Enums sind ein spezieller Datentyp, ein
Aufzählungstyp.

Wahr

Falsch

Jeder Enum Wert ist eine Konstante.

Wahr

Falsch

Jeder Enum Wert ist eine Konstante.

Archive

- Wir stellen uns vor, wir sind Sammlerinnen. Wir sammeln Bücher. Für unsere Bücher brauchen wir Boxen um sie vor Staub und Licht zu schützen. In einer Box haben 10 Bücher Platz.

Archive: Books

- Book
 - Klasse die ein Buch darstellt. Hat Titel und Autor (Strings)
- BookBox
 - Klasse die die Box darstellt. Hat Platz für 10 Bücher.
 - Kann ein Buch aufnehmen, falls noch Platz ist
 - Kann die Titel der Bücher in die Konsole schreiben.
- Archive
 - Main Methode, zum Erstellen von Büchern und Boxen

Archive: Fossils

Jetzt wollen wir in unser Archiv eine neue Art von Objekt aufnehmen: Fossilien (Fossil: String fossilOf; double weight, int approxAge)

Auch dafür brauchen wir wieder eine eigene Box, sonst machen wir unsere Bücher und/oder Fossilien kaputt.

Die Box kann wieder 10 Fossilien aufnehmen, und ausgeben um welches Fossil es sich handelt.

Archive: Generic Box

Was wenn wir eine Box bauen könnten, die man mit allen Möglichen Objekten füllen könnte?

GENERICICS

- ermöglichen das erstellen von Klassen und Methoden, die mit verschiedenen Datentypen arbeiten können
- der Compiler überprüft die Typsicherheit
- Konvention: T als Generische Typbezeichnung
- Als Klasse: `public class Box<T>`
- Wird in Collections verwendet

Generics

Wofür sind Generics gut?

Collections

- Ermöglichen das Speichern und Verwalten von Objekten
- Verschiedene Datenstrukturen (Listen: `ArrayList` & `LinkedList`; Maps: `HashMap` & `TreeMap`; Sets: `HashSet` & `TreeSet`)
- Bieten viele Methoden zum Hinzufügen, Löschen und Manipulieren der enthaltenen Objekte

Collections: ArrayList

Collections: Arraylist

- Wir bauen einen ReadingTracker
- Dafür brauchen wir 2 Listen; toBeRead und alreadyRead
- Wir werden ein Book Objekt brauchen und einige davon instanziiieren und sie in die entsprechenden Listen sortieren
- Dann wollen wir und ein Buch aus der toBeRead Liste holen und "lesen" (also von der toBeRead zu alreadyRead Liste schieben)
- Einen Listeneintrag verändern
- Die beiden Listen in die Konsole schreiben

Wahr

Falsch

Enums können jeden Typ von Wert speichern.

Wahr

Falsch

Enums können jeden Typ von Wert speichern.

Wahr

Falsch

Wrapper-Klassen in Java bieten eine Möglichkeit, primitive Datentypen als Objekte darzustellen.

Wahr

Falsch

Wrapper-Klassen in Java bieten eine Möglichkeit, primitive Datentypen als Objekte darzustellen.

Wahr

Falsch

Autoboxing und Unboxing ermöglichen automatische Konvertierung zwischen primitiven Datentypen und ihren entsprechenden Wrapper-Klassen

Wahr

Falsch

Autoboxing und Unboxing ermöglichen automatische Konvertierung zwischen primitiven Datentypen und ihren entsprechenden Wrapper-Klassen

Wahr

Falsch

Konstanten können und sollen während der Laufzeit noch geändert werden können.

Wahr

Falsch

Konstanten können und sollen während der Laufzeit noch geändert werden können.

Wahr

Falsch

Statische Methoden gehören zur Klasse und nicht zu Instanzen der Klasse.

Wahr

Falsch

Statische Methoden gehören zur Klasse und nicht zu Instanzen der Klasse.

Wahr

Falsch

Generics erhöhen die Wahrscheinlichkeit, dass es zu Typsicherheitsfehlern kommt.

Wahr

Falsch

Generics erhöhen die Wahrscheinlichkeit, dass es zu
Typsicherheitsfehlern kommt.

Wahr

Falsch

Generics können nur für Klassen verwendet werden und nicht für
Methoden.

Wahr

Falsch

Generics können nur für Klassen verwendet werden und nicht für
Methoden.

Wahr

Falsch

ArrayLists in Java sind dynamisch veränderbare Listen, die Elemente
beliebigen Datentyps speichern können.

Wahr

Falsch

ArrayLists in Java sind dynamisch veränderbare Listen, die Elemente beliebigen Datentyps speichern können.

Icebreaker

- Person (firstName, lastName, favourites)
 - ▶ Constructor, introduceMe
- Medium(Enum: BOOK, FILM, SERIES, VIDEOGAME, MAGAZINE)
- Favourite (Medium, title, comment)
 - ▶ The comment is optional
- Introduction(main)

PR2

Dritter Halbttag

01.04.2025

Collections: HashMap

- First steps
- Series Tracker

Local Date

- Immutable: Wie String, wenn es instanziiert wird, kann es nicht mehr geändert werden. Operationen wie `plusDays()` machen ein neues Objekt.
- Thread-safe: Weil es immutable (unveränderbar) ist, kann es in gleichzeitig laufenden Applikationen verwendet werden.
- Es hat Methoden wie `plusDays()`, `minusMonths()`, und `isBefore()`.
- Kann vergleichen: `isBefore()`, `isAfter()`, `isEqual()`, `until()`
- ISO-8601 Format: `LocalDate` verwendet das ISO-8601 Standard Format (YYYY-MM-DD), außer es wird händisch bei Instanziierung geändert.

Wohin würdest du am liebsten reisen?

- | | |
|--------|------------|
| A: | B: |
| Asien | Amerikas |
| C: | D: |
| Afrika | Australien |

Welches Getränk hast du am liebsten?

- | | |
|----------|--------|
| A: | B: |
| Wasser | Kaffee |
| C: | D: |
| Limonade | Tee |

Welche Jahreszeit ist dir am liebsten?

- | | |
|----------|--------|
| A: | B: |
| Frühling | Sommer |
| C: | D: |
| Herbst | Winter |

Hast/hättest du gerne ein Haustier?

- | | |
|------------------|---------|
| A: | B: |
| Ja | Nein |
| | |
| C: | D: |
| Hoffentlich bald | Niemals |

Was ist deine Lieblingsfarbe (oder am nächsten dran)?

- | | |
|------|------|
| A: | B: |
| Rot | Grün |
| | |
| C: | D: |
| Blau | Gelb |

In Gruppen

Versucht das Programm zu planen:

- Welche neuen Objekte werdet ihr brauchen?
 - Was müssen diese Objekte haben?
- Welche Methoden werdet ihr brauchen?
 - Wo sollten diese Methoden sein?
- Welche Art von Collection könnte euch hier helfen?

Gift Organizer

- I want to keep track of gifts I could give to friends and family
- I want to be able to know which birthday comes up in a specified time range
- I want to know which gifts could be in my price range (one that I can specify)
- I want to know whether a gift I'm considering has already been given

Gift Organizer

- Gift
 - Label, price, description(optional)
 - Constructor, getter, setter
- Person
 - Name, birthday, alreadyGifted, giftIdeas
 - Constructor, getter, setter, add giftIdea
- OrganizingService
 - Gifts, people
 - Constructor, findGiftsFor, birthdaysBefore, findGiftsInPriceRange, giveGift
- Organizer
 - main

PR2

Vierter Halbttag

01.04.2025

PASCAL Case

- Case: Schreibweise
- Pascal case: Alle Wörter werden ohne Satzzeichen zusammengeschrieben und von jedem Wort wird der erste Buchstabe großgeschrieben
- Bsp: ShoppingCenter, OpeningHourVariation
- Wird in Java verwendet um Klassen zu benennen

Camel Case

- Case: Schreibweise
- Camel case: Alle Wörter werden ohne Satzzeichen zusammengeschrieben und von jedem Wort wird nur der erste Buchstabe großgeschrieben **außer** vom ersten Wort
- Bsp: checkInNewGuest, isOpenOnMondays
- Wird in Java verwendet um Methoden, Variablen, Parameter zu benennen

UPPERCASE_WITH_UNDERSCORES

- UPPERCASE_WITH_UNDERSCORES: Alle Wörter werden mit Unterstrich zusammengeschrieben und alle Buchstaben werden großgeschrieben
- Bsp: BLUE, HIGH_FANTASY
- Wird in Java verwendet um Konstanten und Enum-Konstanten zu benennen

Lowercase with dot separator

- Lowercase mit Punktseparator: Typischerweise nur ein Wort, wenn mit Punkt getrennt, zeigt es die Hierarchie an
- Bsp: campus.practice (campus ist das package in dem practice drin ist)
- Wird in Java verwendet um packages zu benennen

Same Concept, different names

- Klassenvariablen
- Instanzvariablen
- Member variable
- Fields
- Class fields
- Instance fields
- Attributes

Klassenvariablen

- Werden in einer Klasse deklariert (nicht in einer Methode oder Konstruktor)
- Werden von allen Instanzen genutzt
- Werte sind spezifisch für jede Instanz einer Klasse
- Halten Zustand
- Statisch: nur für Klasse gedacht
- Nicht-statisch: für Instanzen der Klasse gedacht

Parameter

- Werden in einer Methode oder im Konstruktor deklariert
- Repräsentiert die Daten die für die Methode oder den Konstruktor unbedingt notwendig sind
- Sind Platzhalter für Daten die von außerhalb der Methode/Konstruktor bereitgestellt werden
- Sind nicht außerhalb der Methode/Konstruktor verfügbar

Naming Conventions

- Namen von Klassen, Methoden, Parametern und Variablen sollten beschreibend sein – auch wenn das die Namen lang macht
 - findGiftInPriceRange()
- Boolean Variablen sollten mit "is-", "has-", oder "can-" beginnen
 - hasGiftAlreadyBeenGiven, isNew, canChange
- Variablennamen die nur aus einem Buchstaben bestehen, sollten nur für temporäre Variablen verwendet werden, wie in loops
 - for (int i = 0, i < 10, i++)

Klassen, Objekte, Datentypen...

- OOP -> Objektorientiertes Programmieren heißt, dass wir alles (außer primitive Datentypen) als Objekt darstellen.
- Objekte sind wie Datentypen, die wir selbst bestimmen können. Wir können die Eigenschaften bestimmen und wir können bestimmen (Klassenvariablen), was diese Objekte machen können (Methoden).
- Weil Objekte wie Datentypen sind, kommt es oft vor, dass wir eine oder mehrere Klasse(n) in einer anderen Klasse verwenden. Je komplexer ein Objekt, desto mehr Verschachtelungen kann es geben.

Hotel



Hotel 1: Keyrack

- Es gibt einen Schlüsselkasten in dem alle Schlüssel für die Räume aufbewahrt werden
- Wenn ein Gast eincheckt, muss der Schlüssel aus dem Kasten genommen werden
- Wenn ein Gast auscheckt, muss der Schlüssel wieder zurück in den Schlüsselkasten

Hotel 1: KEYRACK

- Key
 - ▶ roomNumber
 - ▶ isWithGuest
 - Constructor, Getter, Setter, toString

Hotel 1: KEYRACK

- Guest
 - ▶ Key
 - ▶ roomNumber
 - ▶ Name
 - Constructor, Getter, Setter

Hotel 1: KEYRACK

- KeyRack
 - ▶ Key[]
 - ▶ numberOfRooms
 - giveKeyToGuest
 - takeKeyBackFromGuest
 - showKeyRack

Hotel 1: KEYRACK

- Reception
 - ▶ main

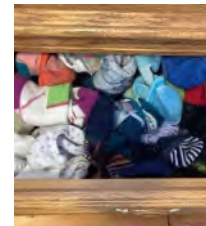
Hotel 2: GUESTBOOK

- Jetzt möchten wir unser Programm um ein Gästebuch erweitern
- Ein Gast kann einen Eintrag ins Gästebuch machen
- Man kann das Gästebuch anschauen

Hotel 2: GUESTBOOK

- Guestbook
 - ▶ Message
 - ▶ guestbookEntries
 - addEntry
 - showGuestbook
 - Constructor

Wofür könnten diese bilder stehen?



Array



Array

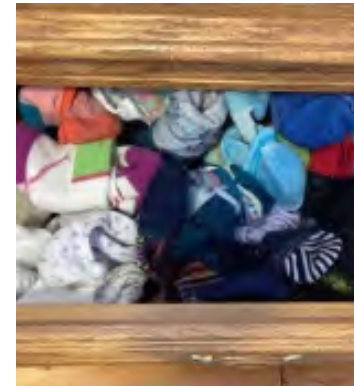


- Es werden nie mehr als 12 Flaschen Platz haben (kann nicht erweitert werden)
- Nur Flaschen können gelagert werden
- Wenn man die Flasche einordnet, hat sie einen fixen Platz (z.B.: ganz oben außen rechts)

ArrayList



ArrayList



- Welches Paar Socken an welchem Platz ist, ist nicht sofort sichtbar
- Die Lade kann man so vollstopfen wie man will – man muss nicht eine neue Lade kaufen, wenn man ein 15. Paar Socken kauft (eine wachsende Lade wäre noch besser)
- Wenn man ein bestimmtes Paar Socken sucht, kann es sein, dass man jedes Paar Socken einmal angreifen muss, bevor man es findet

HashMap



FH CAMPUS 02 // 15/05/25

HashMap



FH CAMPUS 02 // 15/05/25

- Wenn es keine Beschriftung gibt, weiß man nicht in welcher Box man suchen muss
- Wegen der Beschriftung muss man nicht alle Boxen durchsuchen um zu finden was man sucht
- Man kann unendlich viele Boxen dazukaufen und beschriften um mehr Dinge aufzubewahren
- Beschriftung und Inhalt müssen zusammenpassen damit das System verwendbar ist

Git

- <https://courses.cs.washington.edu/courses/cse373/19wi/resources/intellij/git/>
- <https://www.jetbrains.com/help/idea/using-git-integration.html>
- <https://www.jetbrains.com/help/idea/github.html#register-account>

15/05/25

FH CAMPUS 02 //

Kenn mich aus (vorne)



ENUMS

Hab keine Ahnung (hinten)

15/05/25

FH CAMPUS 02 //

Kenn mich aus (vorne)



ARRAYList

Hab keine Ahnung (hinten)

15/05/25

FH CAMPUS 02 //

Kenn mich aus (vorne)



HashMap

Hab keine Ahnung (hinten)

15/05/25

FH CAMPUS 02 //

Kenn mich aus (vorne)



WRAPPER KLASSEN

Hab keine Ahnung (hinten)

15/05/25

FH CAMPUS 02 //

Kenn mich aus (vorne)



Generics

Hab keine Ahnung (hinten)

15/05/25

FH CAMPUS 02 //

Kenn mich aus (vorne)



KONSTANTEN

Hab keine Ahnung (hinten)

15/05/25

FH CAMPUS 02 //

Kenn mich aus (vorne)



Klassen

Hab keine Ahnung (hinten)

15/05/25

FH CAMPUS 02 //

Kenn mich aus (vorne)



ARRAYS

Hab keine Ahnung (hinten)

15/05/25

FH CAMPUS 02 //

Kenn mich aus (vorne)



Loops

Hab keine Ahnung (hinten)

15/05/25

FH CAMPUS 02 //

VERERBUNG

Bernhard Fuchs

Basierend auf Folien von Christian Hofer

TERMINE

0004VD1004 PROGRAMMIEREN 2 (51UE IL, SS 2024/25)

| Gruppe ▼ | | | | | | | | | |
|----------------|------------|-------|-------|-------|----------------|---------|-------------|-----------------------------|---------------------------------|
| Tag | Datum | von | bis | Ort | Ereignis | Termtyp | Lerneinheit | Vortragende*r | Anmerkung |
| Standardgruppe | | | | | | | | | |
| Fr | 21.03.2025 | 08:15 | 12:15 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Do | 27.03.2025 | 08:15 | 15:30 | CZ004 | Abhaltung | fix | | Jandl, Silke | |
| Di | 01.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | Jandl, Silke | |
| Mo | 07.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Di | 08.04.2025 | 13:30 | 15:30 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Do | 10.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mo | 28.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mi | 30.04.2025 | 08:15 | 11:45 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mi | 30.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mi | 14.05.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mo | 26.05.2025 | 08:15 | 12:15 | CZ004 | Prüfungstermin | fix | | | Teilergebnisse: "Klausurarbeit" |

MOTIVATION

- Wiederverwendbarkeit und Erweiterbarkeit von Software als **Qualitätsmerkmale**
- Steigerung der **Stabilität** und **Robustheit** von Software durch Einsatz erprobter Komponenten
- **Zeitersparnis** in der Entwicklung

WIEDERVERWENDUNG VON KLASSEN

- Zwei Varianten
 - **Komposition**
 - **Vererbung**

WIEDERVERWENDEN VON KLASSEN

Komposition

- Zusammensetzen einer Klasse aus bestehenden Klassen

Vererbung

- Erweitern der Eigenschaften einer Klasse und Übernehmen der zugänglichen bestehenden Eigenschaften aus bestehenden Klassen

BEZIEHUNGEN VON KLASSEN

- Besitzt eine Klasse eine Instanz einer anderen Klasse ?
 - HAS-A Beziehung („hat ein“)
 - **Komposition**
- Ist eine Klasse eine Variation einer anderen Klasse ?
 - IS-A Beziehung („ist ein“)
 - **Vererbung**






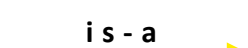
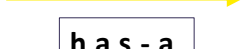
BEISPIEL

Welche Begriffe haben welche Beziehungen ?

| | |
|----------|----------|
| PKW | Fahrzeug |
| Hund | Tier |
| Siedlung | Haus |
| Mann | Person |
| Familie | Kind |
| VW Golf | Auto |
| Auto | Sitze |

BEISPIEL

Welche Begriffe haben welche Beziehungen ?

| | | |
|----------|---|----------|
| PKW |  | Fahrzeug |
| Hund |  | Tier |
| Siedlung |  | Haus |
| Mann |  | Person |
| Familie |  | Kind |
| VW Golf |  | Auto |
| Auto |  | Sitze |

KOMPOSITION - DEFINITION




- Komposition ist eine Möglichkeit zur Wiederverwendung von Klassen
- Eine Klasse **enthält** bzw. **besteht** aus bzw. ist **zusammengesetzt** aus anderen Klassen
- Komposition erzeugt eine **HAS-A** Beziehung
- Die Komposition beschreibt die Beziehung zwischen **einem Ganzen und seinen Teilen**

KOMPOSITION - EIGENSCHAFTEN

- Neue Klasse unabhängig von der Schnittstelle, dem Typ der verwendeten Klassen.
- Die Komposition ist die einzige Möglichkeit Funktionen **mehrerer** (unterschiedlicher) Klassen **zusammenzufügen**.
 - Person mit Address
 - Person mit Dog
 - Usw.

BEISPIELE AUS BISHERIGER LV?

KLASSE IN KLASSENDIAGRAMM

-  Oberes Rechteck:
 - ▶ Klassenname
-  Mittleres Rechteck:
 - ▶ Attribute
-  Unteres Rechteck:
 - ▶ Methoden



KLASSE IN KLASSENDIAGRAMM (2)

Falls nur ein Rechteck:

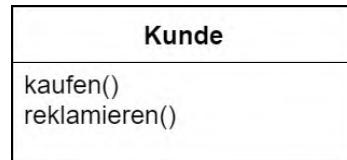
- Klassenname



(Üblicherweise)

Falls nur zwei Rechtecke:

- Klassenname
- Methoden



ATTRIBUTDEKLARATIONEN

```
[sichtbarkeit] [/] name [: Typ] [[Multiplizität]]
[= Vorgabewert] [{eigenschaftswert [, eigenschaftswert]*}]
```

Sichtbarkeit:

- + (public)
- - (private)
- # (protected)
- ~ (package)

```
att1 : int
+att2 : int
+pi : double = 3.1415
-att3 : boolean
#att4 : short
~att5 : String = "Test" {readOnly}
att6 : String [0..*] {ordered}
/ att7
```

Klassenattribut unterstrichen

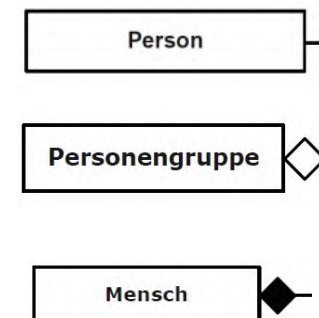
- Alle anderen Möglichkeiten noch nicht wichtig

(MÖGLICHE) DARSTELLUNG KOMPOSITION



ALTERNATIVE DARSTELLUNG KOMPOSITION

- Sind grundsätzlich unterschiedliche Beziehungen (für uns aber nicht relevant)



VERERBUNG - DEFINITION

- Vererbung ist eine weitere Möglichkeit zur Wiederverwendung von Klassen
- Eine Klasse wird von einer anderen Klasse abgeleitet, um **ihre Funktionalität zu erben** und zu erweitern.
- Vererbung erzeugt eine **IS-A** Beziehung
- Vererbung **konkretisiert eine allgemeine** Beschreibung eines Typs

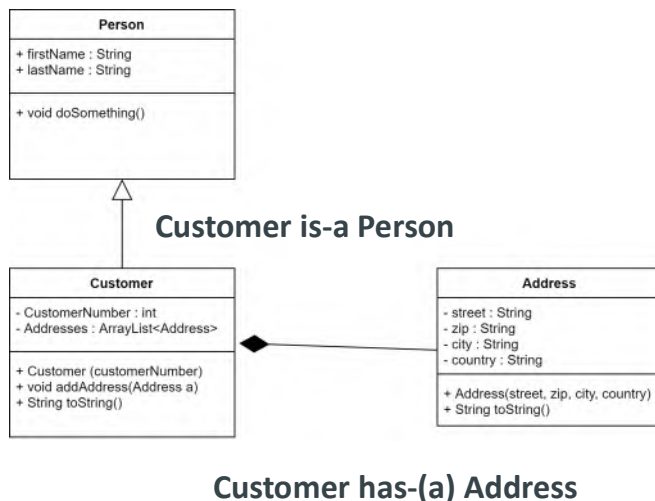
VERERBUNG - EIGENSCHAFTEN

Die neue (konkretere) Klasse hat immer

- zumindest dieselbe Schnittstelle (=alle Methoden der allgemeinen Klassen)
- „denselben Typ“

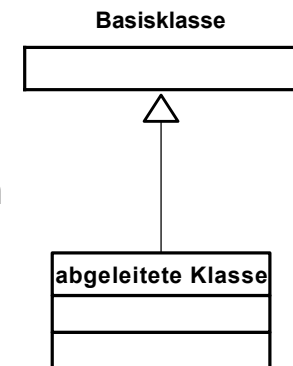
wie die **allgemeine Klasse**.

VERERBUNG VS. KOMPOSITION



VERERBUNG - BEGRIFFE

- Eine Klasse **erbt** von einer **Basisklasse**.
- Die **abgeleitete Klasse** hat Zugriff auf die nicht privaten Eigenschaften und Methoden der Basisklasse.
- Basisklasse wird auch **Super-Klasse** genannt.
- Abgeleitete Klasse wird auch **spezialisierte Klasse** genannt



VERERBUNG IN JAVA

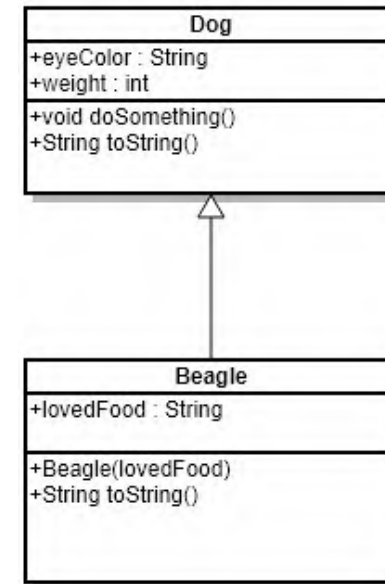
Schlüsselwort: **extends** (erweitert)

Beispiel:

```
class Dog
{
    ...
}

class Beagle extends Dog
{
    ...
}
```

BEISPIEL

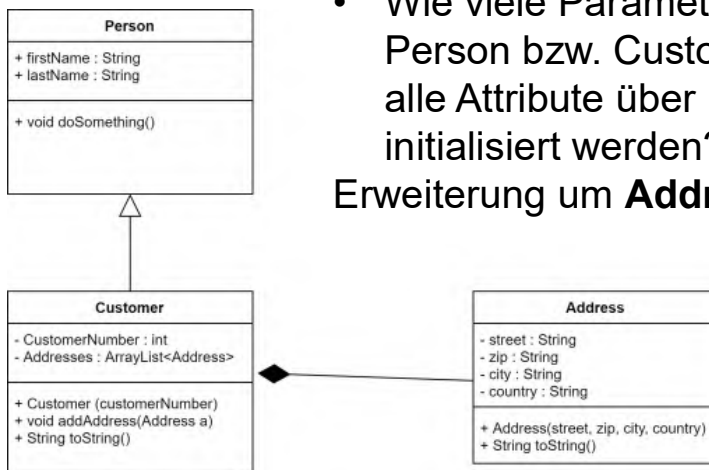


BEISPIEL PERSON

Erstellen Sie die Klassen **Person** und **Customer**

- Vorerst ohne Adressen
- Wie viele Parameter benötigt Person bzw. Customer falls alle Attribute über Konstruktor initialisiert werden?

Erweiterung um **Address** Klasse



VERERBUNG ALTERNATIVE BEGRIFFE

- Vererbung (inheritance)
 - Ableitung (deriving)
 - Spezialisierung
- Basisklasse (base class)
 - Superklasse (super class)
 - Oberklasse
 - Elternklasse (parent class)
- Abgeleitete Klasse
 - Subklasse (sub class)
 - Unterklasse
 - Kindklasse (child class)



VERERBUNG IN JAVA

- Die abgeleitete Klasse hat Zugriff auf die **nicht privaten** Eigenschaften und Methoden der Basisklasse.
- Die abgeleitete Klasse ist **eine Erweiterung** der Basisklasse.
- Es ist nicht möglich die Sichtbarkeit einzelner Methoden / Attribute einzuschränken

VERERBUNG IN JAVA

- Im Konstruktor der abgeleiteten Klasse wird zuerst der Konstruktor der Superklasse aufgerufen.
- Gibt es in der Superklasse **keinen** Default- Konstruktor, so muss ein spezieller Konstruktor explizit aufgerufen werden.
- Schlüsselwort: **super** (*parameters*) ;

SICHTBARKEIT UND VERERBUNG

- Abgeleitete Klassen haben Zugriff auf
 - **public**
 - **protected**
 - **package access** (sofern im selben Package)
 Member (und Methoden) der Superklasse
- Private Member einer Klasse sind für die abgeleitete Klasse **nicht sichtbar** !

Sichtbarkeit und Vererbung

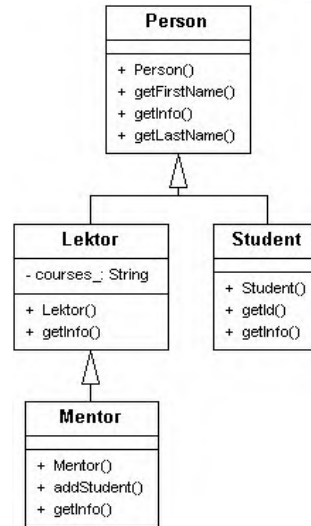
[The official tutorial](#) may be of some use to you.

099



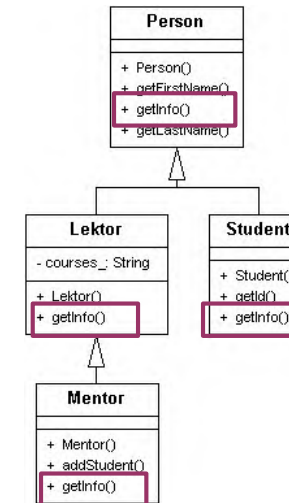
| | Class | Package | Subclass (same pkg) | Subclass (diff pkg) | World |
|--|-------|---------|------------------------|------------------------|-------|
| public | + | + | + | + | + |
| protected | + | + | + | + | |
| no modifier | + | + | + | | |
| private | + | | | | |
| + : accessible blank : not accessible | | | | | |

VERERBUNG KLASSENHIERARCHIE



Seite 29

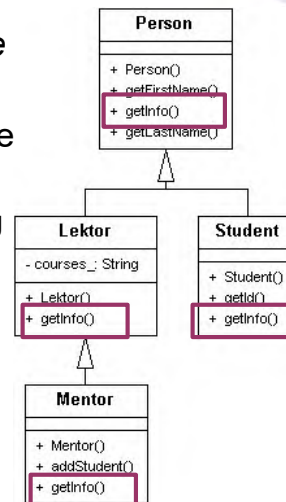
ÜBERSCHREIBEN VON METHODEN



Seite 30

ÜBERSCHREIBEN VON METHODEN

- Abgeleitete Klassen können *zusätzliche* Methoden anbieten
- Abgeleitete Klassen können bestehende Methoden *ersetzen* (= **überschreiben**)
 - Es wird immer die “**neueste**” Implementierung einer Klasse genommen
 - Kennen wir von: toString(), equals(), ...



Seite 31

ÜBERSCHREIBEN VON METHODEN

- Erlaubt in der abgeleiteten Klasse, **eine eigene Implementierung einer vererbten Methode zu erstellen** und somit die Klasse weiter zu spezialisieren.
- Komplette Methodensignatur **bleibt jedoch gleich** (im Unterschied zum Überladen)

Seite 32

TERMINE

0004VD1004 PROGRAMMIEREN 2 (51UE IL, SS 2024/25)

| Gruppe | | | | | | | | | |
|----------------|------------|-------|-------|-------|----------------|--------|-----------------------------|--------|---------------------------------|
| Tag | Datum | von | bis | Ort | Ereignis | Termin | Termin | Termin | Anmerkung |
| Standardgruppe | | | | | | | | | |
| Fr | 21.03.2025 | 08:15 | 12:15 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Do | 27.03.2025 | 08:15 | 15:30 | CZ004 | Abhaltung | fix | Jandl, Silke | | |
| Di | 01.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | Jandl, Silke | | |
| Mo | 07.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Di | 08.04.2025 | 13:30 | 15:30 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Do | 10.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Mo | 28.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Mi | 30.04.2025 | 08:15 | 11:45 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Mi | 30.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Mi | 14.05.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | Fuchs, Bernhard, Dipl.-Ing. | | |
| Mo | 26.05.2025 | 08:15 | 12:15 | CZ004 | Prüfungstermin | fix | | | Teilergebnisse: "Klausurarbeit" |

ÜBERSCHREIBEN VON METHODEN: REGELN

- **Gleicher** Methodenname
- **Gleiche** Parameterliste (und Reihenfolge)
- **Gleicher** Rückgabetyt
- Zugriff darf nicht eingeschränkter sein, wie in Basisklasse

ANNOTATION @OVERRIDE

- Annotations sind Hinweise für den Compiler
- Werden vor Methoden-Deklaration geschrieben
- Mit **@override** definieren wir, dass es sich um eine überschriebene Methode handelt
- Vorteil:
 - Fehlermeldung, wenn dies nicht der Fall ist

ANNOTATION @OVERRIDE

- Müssen nicht verwendet werden
- Stellen jedoch sicher, dass es diese Methode auch in der Basisklasse gibt
 - zB.: Sicherstellen von Schreibfehlern

ANNOTATION @OVERRIDE

- Ausprobieren in Beispiel

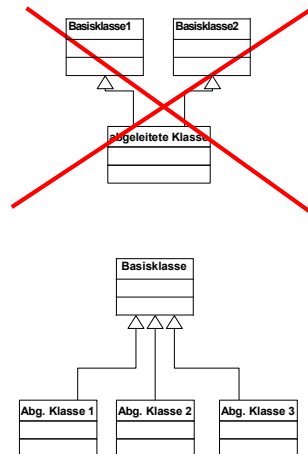
KEYWORD FINAL

- **Bei Attributen:** Wert wird nach Initialisierung nicht mehr verändert
- **Bei Methoden:** Dürfen in abgeleiteten Klassen nicht überschrieben werden
- **Bei Klassen:** Dürfen nicht abgeleitet werden

```
public final class NoInherit
{
    public final static int CONSTANT = 1;
    public final void noOverride() {...}
}
```

KEINE MEHRFACHVERERBUNG

- Man kann in Java höchstens von einer Klasse ableiten !
- Von einer Klasse können aber beliebig viele andere Klassen abgeleitet werden



FALLEN BEIM ABLEITEN

Konstruktoren

- Das Statement **super()** wird im Konstruktor einer abgeleiteten Klasse am Anfang **automatisch aufgerufen**. Gibt es **keinen default-Konstruktor**, so meldet der Compiler **einen Fehler**, falls kein spezieller Konstruktor explizit aufgerufen wird.

FALLEN BEIM ABLEITEN

Designfehler (wie z.B. unpassende Methodennamen werden mit vererbt)

Beispiel Säugetiere:

- Methoden: `getSize()`, `getMaxSpeed()`, `walk()`
- Problem: Wale gehen nicht, sie schwimmen
- Lösung: Allgemeine Beschreibung des Typs in der Basisklasse (`walk()` → `move()`)

FEHLER BEIM ABLEITEN

- Vererbung **dient der Spezialisierung** von Typen, **nicht der Initialisierung** von Werten.
- Das Setzen von Werten ist Aufgabe der Instanzen von Klassen.
- Beim Ableiten ändert sich nur die Funktionalität / „Bedeutung“ der Klasse, nicht zwingend die Werte der Variablen.

UP-CASTING

- Up-Casting heißt, dass eine Instanz eines speziellen Typs einer Variable eines allgemeinen Typs zugeordnet wird.
- Beispiel:

```
Dog myDog = new Beagle ();
```

UP-CASTING

Warum funktioniert das ?

- Ein Beagle **ist** immer noch ein Hund (IS-A). Daher kann er alles was man von einem allgemeinen Hund erwartet.
- Manche allgemeine Methoden können spezielle Ausprägungen haben (ein Beagle bellt anders als ein Schäferhund, aber da beide Hunde sind, können sie bellen)

UP-CASTING

- Da alle Klassen, entweder direkt oder indirekt von der Klasse *Object* abgeleitet sind, gilt folgender Ausdruck immer:
- `Object myObj = new myClass ();`
- Darum hat jede Klasse eine Methode `toString()`

KLASSE OBJECT

Wichtige Methoden von **Object**

- `int hashCode()`
- `String toString()`
- `boolean equals(Object obj)`

<http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

DOWN-CASTING

- Down-Casting heißt, dass eine Instanz eines allgemeinen Typs einer Variable eines speziellen Typs zugeordnet wird.
- Beispiel:

```
Beagle mybeagle = (Beagle) myDog;
```

DOWN-CASTING

- Warum funktioniert das ?
- Eine Variable eines allgemeinen Typs kann durch Up-Casting auf ein Objekt eines speziellen Typs zeigen.
- Durch **explizites** Down-Casting wird (auf Verantwortung der umsetztenden Person) ein spezieller Typ angenommen.
- Wenn Fehler, dann erst zur Laufzeit sichtbar (`ClassCastException`)

AUSPROBIEREN

Up and Downcasting

BEISPIEL (AUCH NACH OSTERN)

- Erstellen Sie eine Klasse „Hase“ mit einem Attribut für den Namen und folgenden Methoden:
 - schlafen, hoppeln, fressen, Konstruktor(String name)
(Funktionalität: `System.out.println(name + „ schläft“;`)
- Leiten Sie davon folgende Klassen mit speziellen Methoden ab:
 - Weihnachtshase (spezielle Meth.: `verteileGeschenke()`)
 - Osterhase (spezielle Meth.: `versteckeOstereier()`)
 - Hoppel Methode überschreiben
- Schreiben Sie eine Applikation (Main Klasse), welche die Klassen verwendet

BEISPIEL (ERWEITERUNG)

Schreiben Sie eine Klasse Hasenstall, die alle Hasen von zuvor in einer ArrayList verwaltet.

Hinweis: ArrayList hat als Typ Hase (damit wir alle Hasen aufnehmen können)

Hasenstall soll Methoden besitzen

- um Hasen hinzuzufügen (add methode)
- und eine Methode die alle Hasen einmal hoppeln lässt.

VERERBUNG UND POLYMORPHISMUS

LERNZIELE

- Welche Arten des Polymorphismus gibt es ?
- Was ist Subtyp - Polymorphismus?

POLYMORPHIE

Griechisch für „Vielgestaltigkeit“

Arten von Polymorphie in OOP:

- Impliziter Polymorphismus
- Überladener Polymorphismus
- Parametrisierter Polymorphismus
- Subtyp-Polymorphismus

IMPLIZITER POLYMORPHISMUS

Implizite Konvertierung von Datentypen

Beispiel: 3.0 + 2

Beispiel: Implizites Upcasten

- `public void doSomething (Object o) {...}`
- `doSomething (new String („abcdef“));`

ÜBERLADENER POLYMORPHISMUS

Überladen von Methoden

- `public void doSomething (int p1) {...}`
- `public void doSomething (int p1, int p2) {...}`

Überladene Operatoren

- 3 + 2
- 4.14 + 6.23
- „abc“ + „def“

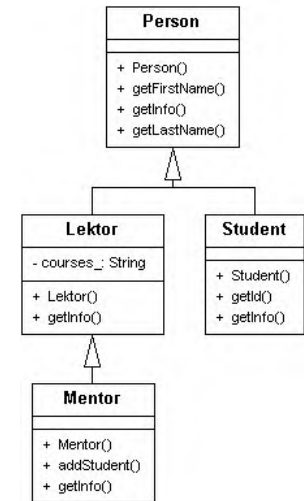
PARAMETRISIERTER POLYMORPHISMUS

- Typen, deren Definitionen Typvariablen enthalten
- Generische Typen (seit Java 5.0)
- Beispiel:
 - ArrayList <String>, ArrayList <Integer>, ...

TYPEN BEI VERERBUNG

Was ist zulässig?

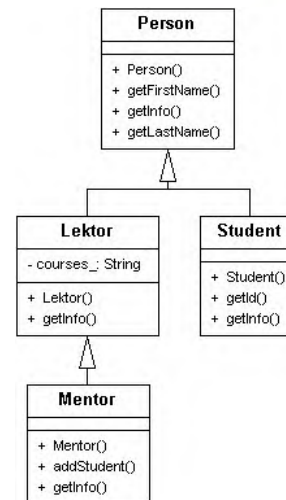
```
Lektor le1 = new Lektor();
Student st1 = new Student();
Lektor me1 = new Mentor();
Object obj = le1;
Student st2 = le1;
Person pel = st1;
pel = le1;
Mentor me2 = st1;
me2 = me1;
le1 = me1;
```



TYPEN BEI VERERBUNG

Was ist zulässig?

```
✓ Lektor le1 = new Lektor();
✓ Student st1 = new Student();
✓ Lektor me1 = new Mentor();
✓ Object obj = le1;
✗ Student st2 = le1;
✓ Person pel = st1;
✓ pel = le1;
✗ Mentor me2 = st1;
✗ me2 = me1;
✓ le1 = me1;
```



SUBTYP - POLYMORPHISMUS

Auch Inklusionspolymorphie

*...bezeichnet die Tatsache, dass eine Instanz einer abgeleiteten Klasse auch über eine Referenz ihrer Basisklasse benutzt werden kann, wobei aber **weiterhin die Implementierung** der abgeleiteten Klasse aufgerufen wird.*

SUBTYP - POLYMORPHISMUS

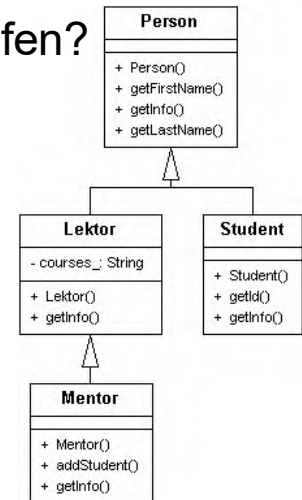
*Das Verhalten einer Methode wird also je nach
benutztem Objekttyp unterschiedlich sein.*

*Bei überschriebenen Methoden wird die
speziellste verfügbare Implementierung
ausgeführt.*

GEÄNDERTES VERHALTEN

Welche Methode wird aufgerufen?

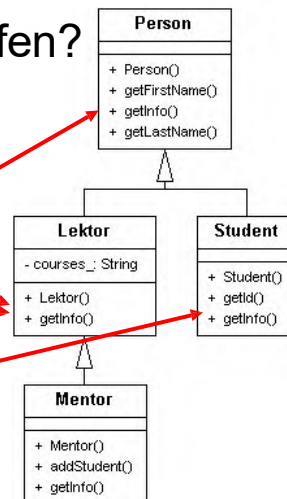
```
Lektor le = new Lektor(..);  
le.getInfo();  
Person p = le;  
p.getInfo();  
p = new Person(..);  
p.getInfo();  
p = new Student(..);  
p.getInfo();
```



GEÄNDERTES VERHALTEN

Welche Methode wird aufgerufen?

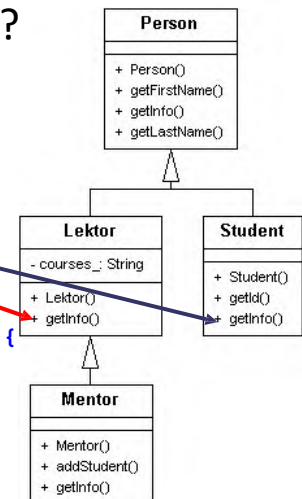
```
Lektor le = new Lektor(..);  
le.getInfo();  
Person p = le;  
p.getInfo();  
p = new Person(..);  
p.getInfo();  
p = new Student(..);  
p.getInfo();
```



GEÄNDERTES VERHALTEN

Welche Methode wird aufgerufen?

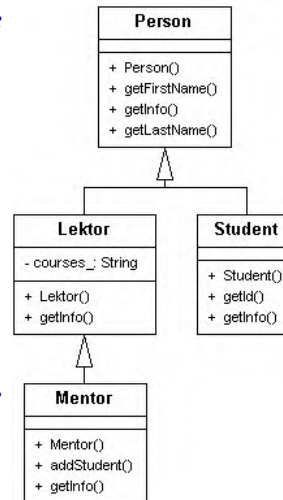
```
Class Campus{  
    public void addPerson(Person p) {  
        p.getInfo();  
    }  
}  
Class Application{  
    public static void main (String[] args){  
        Campus c02 = new Campus();  
        c02.addPerson(new Lektor(..));  
        c02.addPerson(new Student(..));  
    }  
}
```



WIEDERHERSTELLEN DES OBJEKTTYPEES

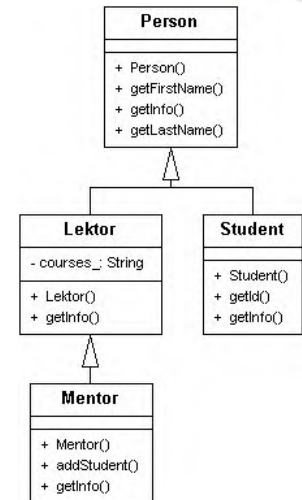
```
Object[] store = new Object[2];  
store[0] = new Lektor(..);  
store[1] = new Student(..);
```

```
✗ Person p = store[0];  
  p = (Person) store[0];  
✗ Lektor l = (Person) store[0];  
  l = (Lektor) store[0];  
  p = (Person) store[1];  
✗ Student s = (Student) store[0];  
  s = (Student) store[1];
```



WIEDERHERSTELLEN DES OBJEKTTYPEES

```
Person p = new Student(..);
```



BEISPIEL ORCHESTER

- Schreiben Sie Klassen für Instrumente (Gitarre, Trompete, Klarinette, Geige...) die sich von einer Basisklasse Instrument (mit dem Attribut lautstaerke vom Typ int) ableiten
- Instrumente sollen die Methode **public int play()** der Basisklasse überschreiben in der sie ihre aktuelle Lautstärke als int zurückliefern und klingen, also ihren Namen und ihre Spielweise auf die Konsole schreiben (z.B.: "Pauke wird geschlagen", "Geige wird gestrichen", ...)
- Implementieren Sie eine Klasse Orchester die mehrere Instrumente beinhalten kann (ArrayList), und eine Methode playAll hat. Diese soll alle Instrumente spielen (play aufrufen) und die aktuelle Lautstärke (Summe aller Instrumente) zurückliefern können.

BEISPIEL HUNDEFRISEUR

- Finden Sie sich in 2-er Gruppen zusammen.
- Sie leiten gemeinsam ein Team in einer kleinen Programmierfirma. Um Arbeit auszulagern schreiben Sie ein Requirements Doc für die Praktikantin. Die P. soll ein kleines Projekt erstellen mit einem Hundefriseur.
- Definieren Sie genau welche Klassen es geben soll, und wie die Funktionalität des Hundefriseurs in der Main() getestet wird (sind die Haare nachher kürzer?).
- Kern: Hundefriseur(Dog d) schneidet von diesem Hund die Haare.

ABSTRAKTE KLASSEN & INTERFACES

Bernhard Fuchs

basierend auf Folien von Christian Hofer

TERMINE

0004VD1004 PROGRAMMIEREN 2 (51UE IL, SS 2024/25)

| Gruppe | | | | | | | | | |
|----------------|------------|-------|-------|-------|----------------|--------|--------|--------|---------------------------------|
| Tag | Datum | von | bis | Ort | Ereignis | Termin | Termin | Lern | Vortragende*r |
| | | | | | | typ | heit | inheit | Anmerkung |
| Standardgruppe | | | | | | | | | |
| Fr | 21.03.2025 | 08:15 | 12:15 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. |
| Do | 27.03.2025 | 08:15 | 15:30 | CZ004 | Abhaltung | fix | | | Jandl, Silke |
| Di | 01.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | | Jandl, Silke |
| Mo | 07.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. |
| Di | 08.04.2025 | 13:30 | 15:30 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. |
| Do | 10.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. |
| Mo | 28.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. |
| Mi | 30.04.2025 | 08:15 | 11:45 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. |
| Mi | 30.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. |
| Mi | 14.05.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. |
| Mo | 26.05.2025 | 08:15 | 12:15 | CZ004 | Prüfungstermin | fix | | | Teilergebnisse: "Klausurarbeit" |

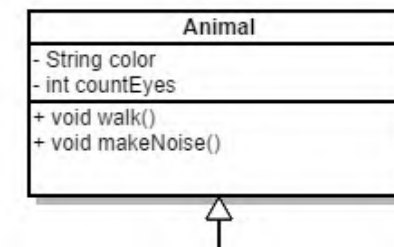
LERNZIELE

- Was sind abstrakte Klassen?
- Was sind abstrakte Methoden?
- Was sind Interfaces?
- Wo liegen die Unterschiede zwischen

Interfaces und

abstrakten Klasse?

BEISPIEL BASISKLASSE ANIMAL




```
public class Animal
{
    private String name;
    private String color;
    private int countEyes;

    public Animal(String name, String color, int countEyes)
    {
        this.name = name;
        this.color = color;
        this.countEyes = countEyes;
    }

    public void walk()
    {

    }

    public void makeNoise()
    {

    }
}
```

Seite 5

FRAGESTELLUNG?

- Wie geht ein „Animal“ / Tier eigentlich?
- Welche Geräusche macht ein „Animal“ / Tier eigentlich?
- Soll man von Animal ein Objekt erstellen können?
Wenn ja, was ist das für ein Tier?

Seite 6

PROBLEMSTELLUNG?

- Es sollte eigentlich **kein** Objekt von *Animal* erstellt werden dürfen
- „Es gibt kein Tier, welches nur ein Tier ist“
- Die leeren Methoden *walk()* und *makeNoise()* bei Tier sind nicht schön (haben eigentlich keine Funktion)

Seite 7

SCHLÜSSELWORT: ABSTRACT

```
public abstract class Animal
{
    private String name;
    private String color;
    private int countEyes;

    public Animal(String name, String color, int countEyes)
    {
        this.name = name;
        this.color = color;
        this.countEyes = countEyes;
    }

    public abstract void walk();

    public abstract void makeNoise();
}
```

Seite 8

ABSTRAKTE METHODEN

```
public abstract void walk();  
public abstract void makeNoise();
```

- Schlüsselwort *abstract* definiert Methoden **ohne konkrete** Implementierung
- Haben **keinen Methodenkörper**
- Abstrakte Methode führt automatisch dazu, dass auch Klasse abstrakt sein muss.
→ Dh. auch Klasse muss mit *abstract* deklariert werden

ABSTRAKTE METHODEN

```
public abstract void walk();  
public abstract void makeNoise();
```

- Schlüsselwort stellt sicher, dass die Methode in einer Subklasse überschrieben werden muss.
 - Ausnahme: Die Subklasse ist auch abstrakt.
- Eine konkrete Implementierung wird erzwungen!

ABSTRAKTE KLASSEN

```
public abstract class Animal  
{
```

- Schlüsselwort stellt sicher, dass von der Klasse kein Objekt erstellt werden kann
Animal a = new Animal(...) → funktioniert **NICHT**
- Klasse kann jedoch als Basisklasse verwendet werden
- Klasse kann abstrakte Methoden beinhalten









ABSTRAKTE KLASSEN

```
public abstract class Animal  
{
```

- Klasse kann jedoch als Typ (Upcast) verwendet werden
Animal a = new Dog(...); → **FUNKTIONIERT**

TERMINE

0004VD1004 PROGRAMMIEREN 2 (51UE IL, SS 2024/25)

| Gruppe  | | | | | | | | | | |
|--|---|---|---|---|--|--------|--------|---|---|---------------------------------|
| Tag | Datum  | von  | bis  | Ort  | Ereignis  | Termin | Termin | Lerneinheit  | Vortragende*r  | Anmerkung |
| Standardgruppe | | | | | | | | | | |
| Fr | 21.03.2025 | 08:15 | 12:15 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. | |
| Do | 27.03.2025 | 08:15 | 15:30 | CZ004 | Abhaltung | fix | | | Jandl, Silke | |
| Di | 01.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | | Jandl, Silke | |
| Mo | 07.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. | |
| Di | 08.04.2025 | 13:30 | 15:30 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. | |
| Do | 10.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mo | 28.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mi | 30.04.2025 | 08:15 | 11:45 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mi | 30.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mi | 14.05.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mo | 26.05.2025 | 08:15 | 12:15 | CZ004 | Prüfungstermin | fix | | | | Teilergebnisse: "Klausurarbeit" |

BEISPIEL

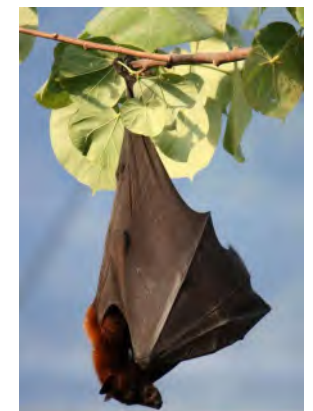
- Führen Sie ein **Refactoring** des bestehenden Instrument-Beispiels durch.
 - Definieren Sie Methode play in der Basisklasse als abstrakte Methode
 - Ändern Sie die Klasse *Instrument* in eine abstrakte Klasse

INTERFACES

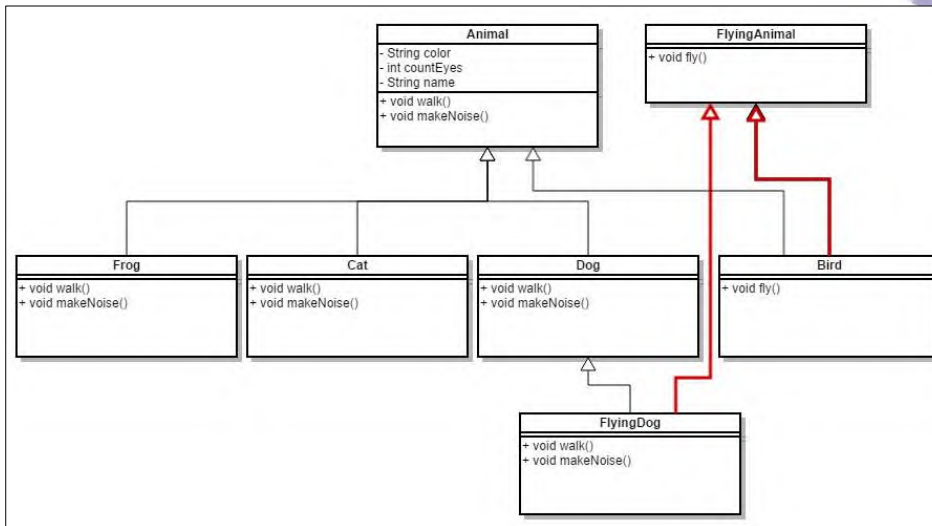
Wir möchten von mehreren Klassen „erben“

WELCHE BASISKLASSE?

- Wir möchten Flughunde hinzufügen
- Wir haben aber zwei Basisklassen:
 - Animal
 - FlyingAnimal
- Wovon sollen wir ableiten?



BEISPIEL



PROBLEMSTELLUNG

- JAVA bietet keine Mehrfachvererbung
- „Hin und wieder“ ist es aber trotzdem notwendig, dass Methoden „geteilt“ (=vorgegeben) werden

public void fly()

PROBLEMSTELLUNG

JAVA bietet keine Mehrfachvererbung

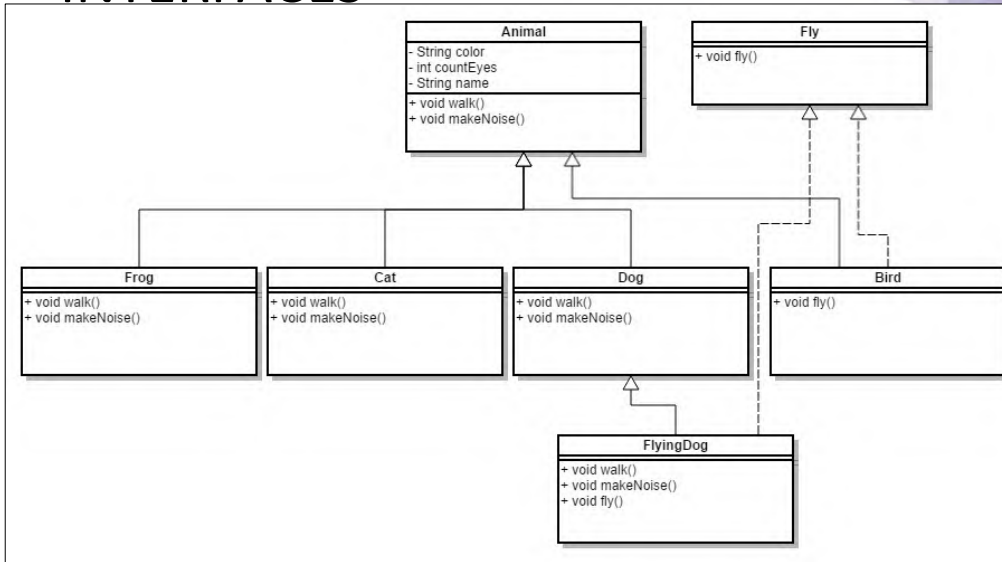
„Hin und wieder“ ist es aber trotzdem notwendig, dass Methoden „geteilt“ werden

public void fly()

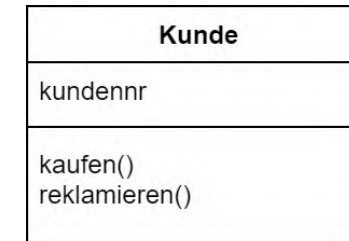
INTERFACES

- Vollständig abstrakte Klassen
 - Keine Methode hat einen Körper
 - Jede Methode ist automatisch **abstract**
 - Jedes Attribut ist automatisch **final**
 - Alle Member sind **public**
- Interface definiert eine abstrakte Funktionalität (=eine Schnittstelle)

INTERFACES



KLASSE



- Abstrakte Klasse
 - **Kunde {abstract}** oder kursiv
- Interface
 - **<<interface>> Kunde**

ABHÄNGIGKEIT (U.A. INTERFACE IMPLEMENTIEREN)



SCHLÜSSELWORT INTERFACE

```

public interface Fly {
    void fly();
}
    
```

- Schlüsselwort *interface* definiert ein Interface
- Auf *abstract* und *access modifier* kann verzichtet werden
 - automatisch abstract
 - automatisch public

SCHLÜSSELWORT IMPLEMENTS

```
public class Bird
    extends Animal
    implements Fly
{
```

- *implements* implementiert Interface
- Eine Klasse kann maximal **von einer anderen** Klasse erben
- Eine Klasse kann **mehrere** Interfaces implementieren
 - Trennung mittels Beistrich

SCHLÜSSELWORT IMPLEMENTS

```
public class Bird
    extends Animal
    implements Fly
{

    public Bird(String name, String color, int countEyes) {
        super(name, color, countEyes);
    }

    public void walk() {
        System.out.println("step by step");
    }

    public void makeNoise() {
        System.out.println("piep piep");
    }

    public void fly() {
        System.out.println("flying in the air");
    }

}
```

VERWENDUNG

- Von Interfaces können **keine** Objekte erzeugt werden
Fly f = new Fly(...) → funktioniert **NICHT**
- Upcast in Interface funktioniert
Fly a = new Bird(...); → **FUNKTIONIERT**

BEISPIEL LOGISTICMANAGER

- Implementieren Sie eine Klasse *Car* mit den Eigenschaften Type, Color und Weight
- Implementieren Sie eine Klasse *Shirt* mit den Eigenschaften Brand, Size und Color
- Definieren Sie ein Interface *Moveable* mit der Methode *void move(String destination)*
- Implementieren Sie dieses Interface bei den Klassen *Car* und *Shirt*. Die Methode *move()* soll beispielsweise ausgeben:
„Blue VW Golf will be moved to Graz“
- Erstellen Sie eine Klasse *LogisticManager*, welcher in einer Liste *Moveable*-Objekte verwaltet. Erstellen Sie im *LogisticManager* eine Methode *public void moveAll(String destination)* und rufen Sie *move()* für alle verwaltete Objekte auf
- Erstellen Sie eine Demo-Anwendung

EXKURS: INSTANCEOF OPERATOR

Objekt instanceof Klasse

- Ist Objekt nicht null
- Objekt ist Instanz der Klasse (oder Subklasse oder implementiert Interface)

KEIN ERSATZ FÜR POLYMORPHISMUS

- Basisklassen mit gemeinsamen Methoden immer sinnvoller so möglich

EXKURS: GETCLASS

Osterhase o = new ...

o.getClass()

- Ausgabe: class at.campus02.pr2.Osterhase

Osterhase.class

- Ausgabe: class at.campus02.pr2.Osterhase

Überprüfung ob Objekt genau einer KLASSE entspricht

- if (o.getClass().equals(Osterhase.class))


SORTIERUNG VON DATEN

Bernhard Fuchs

basierend auf Folien von Christian Hofer

TERMINE

0004VD1004 PROGRAMMIEREN 2 (51UE IL, SS 2024/25)

| Gruppe  | | | | | | | | | |
|--|------------|-------|-------|-------|----------------|--------|--------|-----------------------------|---------------------------------|
| Tag | Datum | von | bis | Ort | Ereignis | Termin | Termin | Lern | Anmerkung |
| | | | | | | typ | heit | einheit | |
| Standardgruppe | | | | | | | | | |
| Fr | 21.03.2025 | 08:15 | 12:15 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Do | 27.03.2025 | 08:15 | 15:30 | CZ004 | Abhaltung | fix | | Jandl, Silke | |
| Di | 01.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | Jandl, Silke | |
| Mo | 07.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Di | 08.04.2025 | 13:30 | 15:30 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Do | 10.04.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mo | 28.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mi | 30.04.2025 | 08:15 | 11:45 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mi | 30.04.2025 | 12:30 | 16:00 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mi | 14.05.2025 | 08:15 | 16:00 | CZ004 | Abhaltung | fix | | Fuchs, Bernhard, Dipl.-Ing. | |
| Mo | 26.05.2025 | 08:15 | 12:15 | CZ004 | Prüfungstermin | fix | | | Teilergebnisse: "Klausurarbeit" |

INHALTE & ZIELE

- Sortieren von Daten
- Daten die in Arrays od. Collections vorliegen nach unterschiedlichen Kriterien sortieren können
 - ▶ Unterschied zw. den Interfaces Comparable und Comparator erklären können
 - ▶ Comparable und Comparator nach diversen Vorgaben eigenständig implementieren und anwenden können

SORTIEREN VON ARRAYS

- Arrays:
 - ▶ um Daten zu sortieren die in Arrays vorliegen bietet Java die "Hilfsklasse" **java.util.Arrays**
 - ▶ ja nach Datentyp gibt es entsprechende Überladungen einer Methode zum Sortieren **static void sort(type[] a)**
 - ▶ type: byte, char, double, float, short, int, long, Object
- **WICHTIG:** kein Rückgabewert d.h. sortiert direkt das übergebene Array

SORTIEREN VON ARRAYS

- **Arrays.sort(type[] a) Algorithmen**
 - ▶ verwendet für primitive Typen eine **spezielle Variante von QuickSort**
 - ▶ verwendet für Object[] **eine spezielle Kombination von MergeSort und InsertionSort**

⇒ Details zu den Algorithmen QuickSort und MergeSort werden in anderen LVs vorgestellt

SORTIEREN VON ARRAYS

- Arrays.sort(type[] a)
 - ▶ die **Reihenfolge** für die Sortierung ergibt sich aus der sog. "**natürlichen Ordnung**" des jeweiligen Typ
 - ▶ bei numerischen Typen in aufsteigender numerischer Reihenfolgen z.B. int[] {3,1,9,5} -> {1,3,5,9}
 - ▶ bei char[] aufsteigend nach Ascii Tabelleneintrag

Was passiert bei Object[] a => Welche natürliche Ordnung macht hier Sinn?

SORTIEREN VON ARRAYS

- Arrays.sort(Object[] a)
 - wollen wir z.B. ein String[] sortieren lassen funktioniert das erwartungsgemäß
 - übergeben wir jedoch einen **selbst erstellten Referenztyp** z.B. ein Person[] wird eine **ClassCastException** ausgelöst – Warum?

=> Java signalisiert damit, dass **keine Möglichkeit besteht “ohne Zusatzinformation” die Sortierung durchzuführen!**

SORTIEREN VON ARRAYS

- aus der Java API Dokumentation zu

```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. **All elements in the array must implement the Comparable interface. ...**

COMPARABLE

- Java API definiert generisches Comparable Interface (= eine „Schnittstelle“)
 - dieses Interface sollte bei Klassen implementiert werden, für deren Instanzen Vergleiche hinsichtlich einer Ordnungsrelation benötigt werden

```
public interface Comparable <T> {
    public int compareTo(T o);
}
```

COMPARABLE INTERFACE

- compareTo Methode von Comparable
 - nimmt Instanz des entsprechenden Typs entgegen und macht zwischen dem **aktuellen (this)** und dem **übergebenen Objekt (other)** den Vergleich
 - Methode **liefert negativen Wert (< 0)** wenn **aktuelles Objekt vorher** in Sortierung sein soll
 - Methode **liefert positiven Wert (> 0)** wenn **aktuelles Objekt nachher** in Sortierung sein soll
 - Methode **liefert 0 zurück**, sofern **beide Objekte gleichwertig bezogen auf Sortierung** sind

COMPARABLE INTERFACE

- **compareTo Methode von Comparable**

```
public class Person implements Comparable<Person> {
    private int id;
    private String firstName;
    private String lastName;

    @Override
    public int compareTo(Person o) {
        //NOTE: ascending according to id
        if(this.id < o.id) return -1;
        if(this.id > o.id) return 1;
        return 0;
    }
}
```

SORTIEREN VON ARRAYS

```
Person[] people =
    {new Person(4321,"Max", „Mustermann"),
      new Person(3456,"Silvia", „Musterfrau");

    //uses compareTo Method of Comparable Interface
    Arrays.sort(people);

    for(Person p : people) {
        System.out.println(p);
    }
```

SORTIEREN VON COLLECTIONS

▣ Collections

- um Daten zu sortieren die in Collections vorliegen bietet Java die "Hilfsklasse" java.util.Collections
- es gibt dazu eine generische Methode zum Sortieren von Listen welche Referenztypen beinhalten

static void sort(List<T> list)

WICHTIG: kein Rückgabewert d.h. sortiert direkt die übergebene Collection

SORTIEREN VON COLLECTIONS

- **Sortieren mit Comparable**

```
List<Person> people = new ArrayList<>();
people.add(new Person(4321,"Max", „Mustermann"));
people.add(new Person(3456,"Silvia", „Musterfrau"));

//uses compareTo Method of Comparable Interface
Collections.sort(people);

for(Person p: people) {
    System.out.println(p);
}
```


COMPARABLE INTERFACE

- equals() und hashCode()
...kein Zwang aber EMPFEHLUNG!
 - die von Object geerbten equals() sowie hashCode() sollten ebenso implementiert bzw. überschrieben werden, falls eine Klasse Comparable<T> implementiert
 - auf Konsistenz mit compareTo Methode achten
`equals==true <==> compareTo==0`
`equals==false <==> compareTo!=0`

BEISPIEL: HASEN

- Hasen sollen Alter halten und nach dem Alter aufsteigend sortiert werden
- Kleines Testprogramm dazu:
 - Ein paar Hasen in einer ArrayList sortieren
 - Lässt sich damit auch ein Osterhase sortieren?

HASEN SORTIEREN ERWEITERN

- Wir möchten nach zwei Kriterien sortieren
 - Zuerst nach Alter und als zweites Kriterium nach Anzahl der Karotten
- Testen
 - In ArrayList sortieren

COMPARABLE INTERFACE

- Problem:** Für eine Klassen werden unterschiedliche Sortierreihenfolgen benötigt
 - Wir möchten Hasen manchmal
 - nach Alter und Karotten sortieren
 - wie viel Urlaubstage sie haben
 - (oder anderen Kriterien die uns letzts eingefallen sind)
- Das Interface Comparable können wir in einer Klasse jedoch nur einmal implementieren und damit ist die gewünschte Sortierreihenfolge vorgegeben...

STATTDESSEN: COMPARATOR

■ Lösung:

- ▶ die "Hilfsklassen" *java.util.Arrays* sowie *java.util.Collections* bieten **Überladungen der sort Methoden** an welche einen **Comparator** zur Sortierung übernehmen

=> durch geeignete Comparator Klassen lassen sich Arrays als auch Collections beliebig sortieren

COMPARATOR INTERFACE

■ Java API definiert generisches Comparator Interface

- ▶ Comparator Klassen implementieren dieses Interface und können dann für benutzerdefinierte Sortierreihenfolgen verwendet werden

```
public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

COMPARATOR INTERFACE

- compare Methode von Comparator
 - nimmt zwei Instanzen des entsprechenden Typs entgegen und macht den Vergleich
 - Methode liefert negativen Wert (< 0) wenn **Objekt 1 vorher** in Sortierung sein soll
 - Methode liefert positiven Wert (> 0) wenn **Objekt 1 nachher** in Sortierung sein soll
 - Methode liefert 0 zurück, sofern beide Objekte gleichwertig bezogen auf Sortierung sind

COMPARATOR PERSON

- Bsp. Personen nach Id sortieren

```
public class IdComparatorASC
    implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        //NOTE: ascending according to id
        if(o1.getId() < o2.getId()) return -1;
        if(o1.getId() > o2.getId()) return 1;
        return 0;
    }
}
```

COMPARATOR PERSON

- Bsp. Personen nach Vornamen sortieren

```
public class FirstNameComparator
    implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        //NOTE: ascending according to id
        return o1.getFirstName()
            .compareTo(o2.getFirstName());
    }
}
```

SORTIEREN VON COLLECTIONS

- Sortieren mit Comparator

```
List<Person> people = new ArrayList<>();
people.add(new Person(4321, "Max", "Mustermann")),
people.add(new Person(3456, "Silvia", "Musterfrau"));

//uses compareTo Method of Comparable Interface
Collections.sort(people, new IdComparatorAsc());

for(Person p: people) {
    System.out.println(p);
}
```

- Analoge Verwendung bei Arrays

HASSEN SORTIEREN ERWEITERN

- Comparator um nach Urlaubstagen (absteigend) zu sortieren

ANONYMOUS COMPARATOR (FALLS NOCH ZEIT IST)

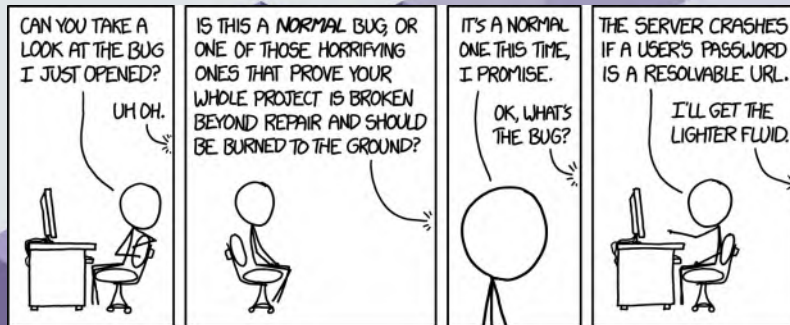
- kürzere Schreibweise durch Comparator als anonyme innere Klasse (falls wir wirklich nur einmal brauchen)

```
//ordering according to comparator
//given as anonymous inner class
//using last name
Collections.sort(lp, new Comparator<Person>() {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getLastName()
            .compareTo(o2.getLastName());
    }
});
```

TESTEN

Bernhard Fuchs

basierend auf Folien von Christian Hofer



INHALTE & ZIELE

- Motivation
- Begriffe
- Debuggen und Testen
- Unit Testing
- JUnit Framework
- Unsere ersten Schritte
- Was gibt es noch



Seite 2

TESTEN SIE IHREN CODE!

- Sonst wird jemand anders sehr schnell Ihren Fehler finden!
- Gleich den Fehler zu finden ist immer besser als auf Tester*in (oder Kund*in) zu warten.



Ein Test ist der erste User Ihres Codes!

Seite 3

WARUM WIR AUTOMATISIERUNG BENÖTIGEN

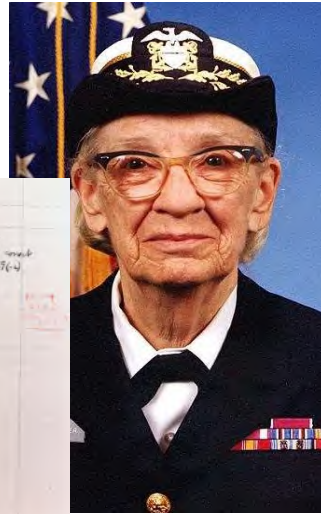
- Am Anfang schreiben wir nur einfache Methoden
 - Mit Ausgaben überprüfen wir ob sie funktionieren
- Durch kontinuierliche Weiterentwicklung an einem Projekt wird Software immer größer und komplexer
 - Wir brauchen mehr als nur Ausgaben die wir manuell interpretieren.
- Wir brauchen mächtige Werkzeuge die uns unterstützen unseren Source Code immer wieder und halb automatisch zu testen!



Seite 4

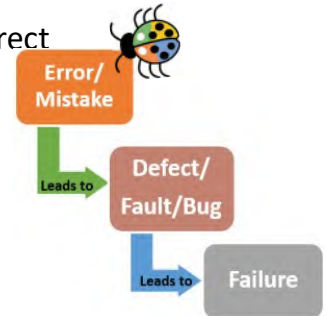
DER ERSTE BUG

- Seit 14 Jhd.
 - „object of terror“
- Motte in Relais
Grace Hopper
(1947)



UNTERSCHIEDLICHE BEDEUTUNG BEGRIFF FEHLER

- Error (oder Mistake):
 - „A human action that produces an incorrect result“ (ISTQB)
- Ein Error kann zu einem Defect führen
- Beispiele
 - Syntax oder logisch
 - Requirementsanalyse
 - Testen
 - ...



UNTERSCHIEDLICHE BEDEUTUNG BEGRIFF FEHLER

- Defect (auch bekannt als Bug oder Fault)
 - „An imperfection or deficiency in a work product where it does not meet requirements or specifications.” (ISTQB)
- Ein Defect kann zu einem Failure führen
 - (unter gewissen Bedingungen)



UNTERSCHIEDLICHE BEDEUTUNG BEGRIFF FEHLER

- Failure
 - „An event in which a component or system does not perform a required function within specified limits.“ (ISTQB)
- (Kann zu unglücklichen Kund*innen bzw. Umsatzverlust führen)



VOM ERROR ZUM FAILURE (INFEKTIONSKETTE)

• Error



- Error made by developer
- Can cause a defect



Defect

- Defect in program state
- Can cause a failure



Failure

- Issue in delivered program

Seite 9

LÖSUNGEN?

Wie haben Sie bis jetzt getestet?

```
System.out.println("it work's");
```

```
//System.out.println("test 1");
```

```
if(something == true)
    System.out.println("it works now");
```



Seite 10

DEBUGGEN UND TESTEN

“Program testing can be a very effective way to show the **presence of bugs**, but is hopelessly **inadequate for showing their absence**”

Dijkstra

Debuggen hilft uns die Infektionskette nachzuvollziehen.



Seite 11

TESTING FOR DEBUGGING

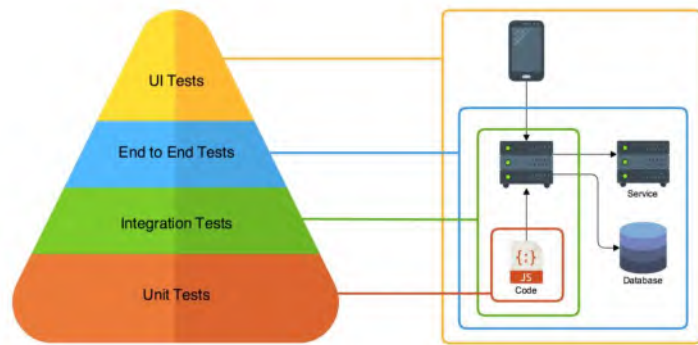
- Was tun **nachdem Fehler bereinigt wurde?**

- **Testen** den Source Code **erneut** damit
 - **Reproduziere** Anwendungsfälle/Fehler
 - **Verifizieren** Fehlerlösung
 - **Ausführen** aller Tests vor nächster Veröffentlichung
 - **Automatisieren** Testfälle

- **Nur funktionsfähigen Source Code** einchecken in das remote repository

Seite 12

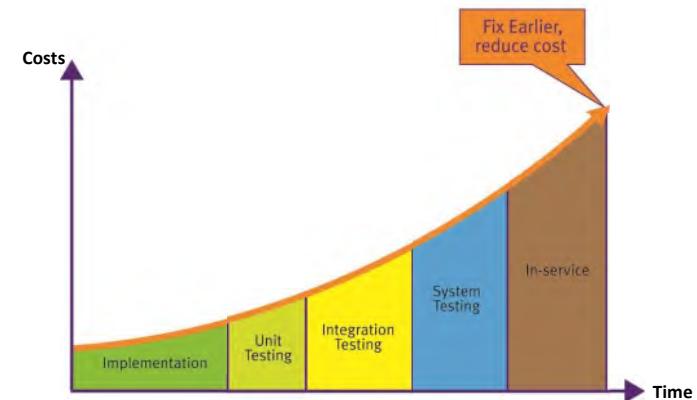
TEST PYRAMIDE



<https://medium.com/@nathankpeck/microservice-testing-introduction-347d2f74095e>

Seite 13

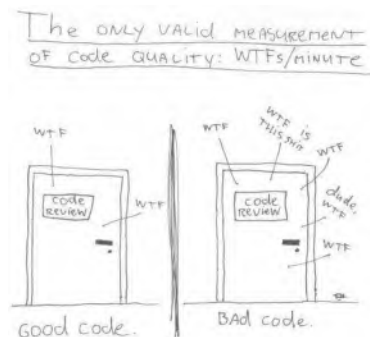
JE FRÜHER DESTO BESSER



<https://edwardthienhoang.wordpress.com/2014/10/29/i-dont-write-unit-tests-because-the-excuses/>

Seite 14

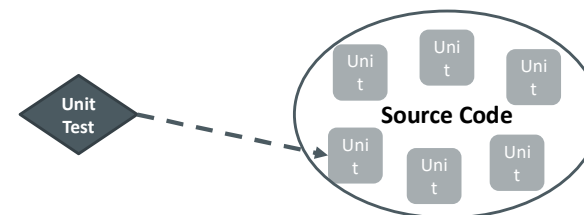
TESTAUTOMATISIERUNG



Seite 15

KONZEPT XUNIT TESTS

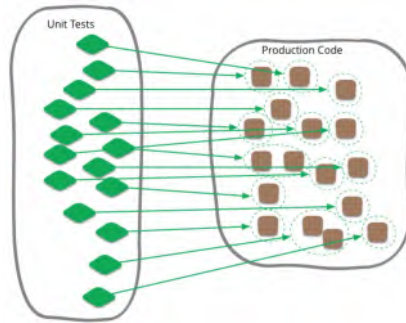
- Testen **kleine Teile** des Source Codes (sogenannte Einheiten units)
- **Verfügbar** in vielen Sprachen



Seite 16

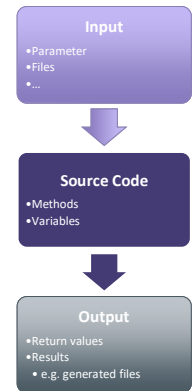
UNIT TESTS UND SOURCE CODE

- Der ganze Code **soll getestet** werden so möglich.
- Jeder Test erhöht die **test coverage**.
- Schreiben kurzen **Test Code** um kleinen Teil des **Source Codes** zu überprüfen

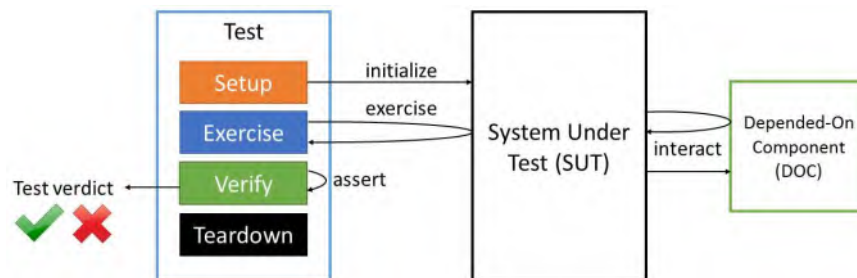


UNIT TEST IN A NUTSHELL

- **Testen** den Anwendungscode
- **Verifizieren** das Verhalten von Methoden
- Unterschiedliche **Inputs**
- Verhalten des **Source Code**
- Überprüfung **Outputs**



TEST STRUKTUR



JUNIT, EIN SIMPLES FRAMEWORK

- Simple Java Framework um **wiederholbare Tests zu schreiben**
- Tests **benötigen keinen Menschen zur Entscheidung ob etwas funktioniert oder nicht**
- **Leicht** viele hintereinander auszuführen



- Nur kleine Unterschiede zwischen **JUnit 4 und 5**

JUNIT 4 BEISPIEL (ALTE VERSION)

```
public class SimpleTest{
    @Before
    public void setUp() { /* ... */ }
    @After
    public void tearDown() { /* ... */ }
    @Test
    public void testMethod() {
        /* ... */
        Assert.assertTrue( /* ... */ );
    }
}
```

Seite 21

JUNIT 5 BEISPIEL

```
public class SimpleTest{
    @BeforeEach
    public void setUp() { /* ... */ }
    @AfterEach
    public void tearDown() { /* ... */ }
    @Test
    public void testMethod() {
        /* ... */
        Assert.assertTrue( /* ... */ );
    }
}
```

<https://junit.org/junit5/docs/current/user-guide/>

Seite 22

JUNIT 4 / 5 ANNOTATIONS

- **@Test** identifiziert eine Testmethode – muss keine Parameter haben bzw. void sein.
- **@Ignore** markiert eine Testmethode die ignoriert werden soll (**@Disabled**)
- **@Before**, **@After** zum Initialisieren und Freigeben vor bzw. nach jeder Testmethode
- **@BeforeEach**, **@AfterEach** zum Initialisieren und Freigeben vor bzw. nach jeder Testmethode (JUnit 5)

Seite 23

ANNOTATIONS: JUNIT 4

- @Test
- @Ignore
- @Before, @After
- @BeforeClass, @AfterClass
- @Test(expected=Exception.class)
- @Test(timeout=1000)

<http://www.java2novice.com/junit-examples/junit-annotations/>

JUNIT 5

- @Test
- @Disabled
- @BeforeEach, @AfterEach
- @BeforeAll, @AfterAll
- ~~@Test(expected=Exception.class)~~
 - In Testmethode mit **assertThrows**

Seite 24

JUNIT ASSERTIONS

- Fehlgeschlagene Assertions werden aufgezeichnet
- Direkt verwenden:
Assert.assertEquals(...)
 - Meistens zwei Parameter
 - Erwarteter Wert
 - Tatsächlicher Wert
 - Falls beide gleich dann Ergebnis positiv!!

JUNIT TEST VIER PHASEN

JUnit Test



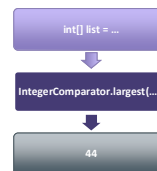
SIMPLER JUNIT 4 / 5 TEST

```
import static org.junit.Assert.*;
import org.junit.Test;

public class TestIntegerComparator {

    @Test
    public void testLargestNumber() {
        // create demo list
        int[] list = new int[]{ 12, 23, 9, 44, 2 };
        assertEquals(44, IntegerComparator.largest(list));
    }
}
```

The code is annotated with JUnit phases: **Setup** (red dashed box around the list creation), **Exercise** (blue dashed box around the `assertEquals` call), and **Verify** (green dashed box around the `assertEquals` call).



ERSTES BEISPIEL

- Wie können wir JUnit5 integrieren?



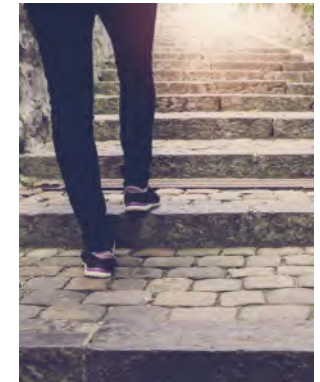
ERSTES BEISPIEL

- Spezifizieren Abhängigkeit in pom.xml (Maven)
 - Siehe Dokumentation

```
<dependencies>
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-api</artifactId>
<version>5.8.2</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-engine</artifactId>
<version>5.8.2</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-params</artifactId>
<version>5.8.2</version>
<scope>test</scope>
</dependency>
</dependencies>
```

(MÖGLICHEE) NÄCHSTE SCHRITTE

- Weitere JUnit Features
- Mock Objekte
- Hamcrest
- Softwaretest Methoden
 - Statisches vs. dynamisches Testen
 - Black-box vs. white box Testen
- Software Entwicklungsprozesse
 - Test-driven development (TDD)
 - Behavior-driven development (BDD)

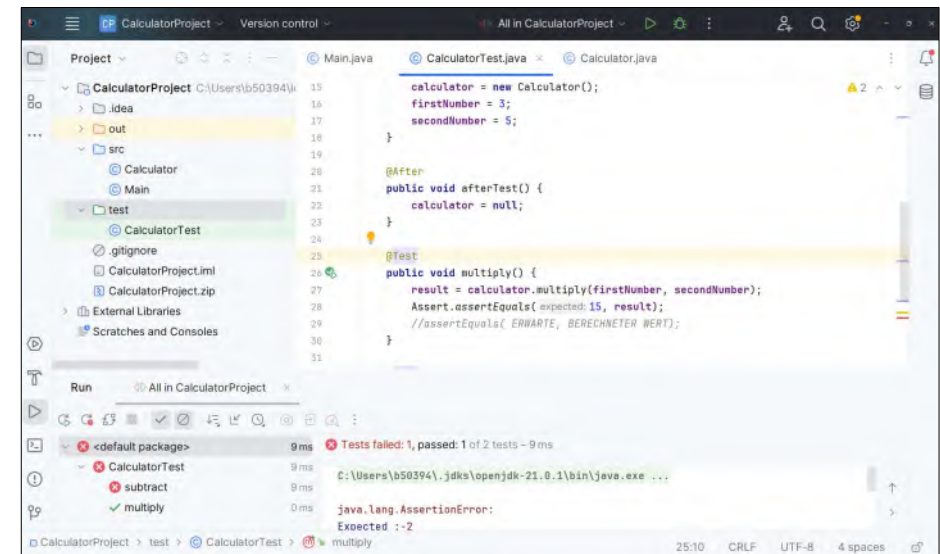


BEISPIEL

1. Create and setup a "tests" folder
 - In the Project sidebar on the left, right-click your project and do New > Directory. Name it "test" or whatever you like.
 - Right-click the folder and choose "Mark Directory As > Test Source Root".
2. Adding JUnit library
 - Right-click your project and choose "Open Module Settings" or hit F4. (Alternatively, File > Project Structure, Ctrl-Alt-Shift-S is probably the "right" way to do this)
 - Go to the "Libraries" group, click the little green plus (look up), and choose "From Maven...".
 - Search for "junit" -- you're looking for something like "junit:junit4.11".
 - Check whichever boxes you want (Sources, JavaDocs) then hit OK.
 - Keep hitting OK until you're back to the code.
3. Write your first unit test
 - Right-click on your test folder, "New > Java Class", call it whatever, e.g. MyFirstTest.
 - Write a JUnit test -- here's mine:

```
import org.junit.Assert;
import org.junit.Test;

public class MyFirstTest {
    @Test
    public void firstTest() {
        Assert.assertTrue(true);
    }
}
```
4. Run your tests
 - Right-click on your test folder and choose "Run 'All Tests'". Presto, test.
 - To run again, you can either hit the green "Play"-style button that appeared in the new section that popped on the bottom of your window, or you can hit the green "Play"-style button in the top bar.



REFERENZEN

- *Andreas Zeller*
Why Programs Fail
A Guide to Systematic Debugging
dpunkt.verlag, 2009
- *Gerard Meszaros*
xUnit Test Patterns, Refactoring Test Code
2007, Addison Wesley
- David Thomas, Andrew Hunt
The Pragmatic Programmer – 20th Anniversary Edition
2019, Pearson Education

LINKS

- *Unit Test*
Martin Fowler
<https://martinfowler.com/bliki/UnitTest.html>
last time visited: 14.03.2021
- *JUnit 4*
JUnit
<https://junit.org/junit4/>
last time visited: 21.04.2020
- *JUnit 5*
JUnit
<https://junit.org/junit5/>
last time visited: 14.03.2021