

Programmierung 2 (Prog2)

Organisatorischer Einstieg



**Hochschule
Bonn-Rhein-Sieg**
University of Applied Sciences

Einführung in die Programmierung



Prof. Dr. Rudolf Berrendorf



M.Sc. André Kless



Prof. Dr. Andreas Priesnitz



Dipl.-Inf. Sigrid Weil

Programmierung 2



Prof. Dr. Karl Jonas
- Ruhestand -



BI + BCSP



BWI

André Kless



- 39 Jahre, verheiratet, 4 Kinder
- M.Sc. Informatik H-BRS
- Entwickler der CCM-Webtechnologie
- Entwickler des Digital Makerspace (DMS)

seit 2015: Wissenschaftlicher Mitarbeiter

2015 - 2020: F&E-Verbundprojekt “work&study”

2021 - 2022: F&E-Verbundprojekt “EILD”

seit 2011

Lehraufträge in

- Einführung in die Programmierung
- Datenstrukturen und Algorithmen
- Software Engineering I
- Einführung in Web Engineering

2020-21

Code-Trainer bei Deutsche Telekom

seit 2022

Lehrkraft für besondere Aufgaben (LfbA)

Dozenten-Teams

BI + BCSP



BWI



Dipl.-Inf. Markus Schneider



M.Sc. Moritz Balg



M.Sc. Javed Razzaq

Markus Schneider



- Dipl.-Inf. Markus Schneider (Uni Bonn)
- 48 Jahre
- Freiberuflicher Software-Entwickler
 - Java-Entwicklung
 - Training von Java-Entwicklern
 - Geodatenverarbeitung

Wissenschaftlicher Mitarbeiter (seit 2021)

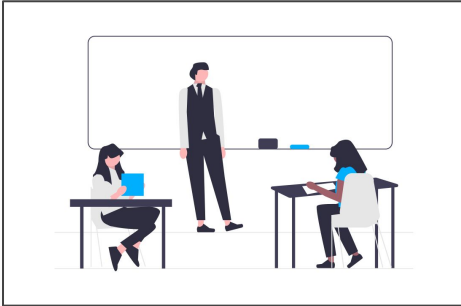
- Datenbanken-Team (Übungsleiter)
- Oracle Academy
- EidP
- KI-Beauftragter

Moritz Balg



- M.Sc. Informatik H-BRS
- 28 Jahre
- Wissenschaftlicher Mitarbeiter (seit 2023)
 - Java-Entwicklung
 - KI-Beauftragter des FB
- Lehre:
 - Einführung in die Programmierung (seit WS20)
 - Programmierung 2
 - Datenbanken
 - Software Engineering 2

Ablauf der Lehrveranstaltung

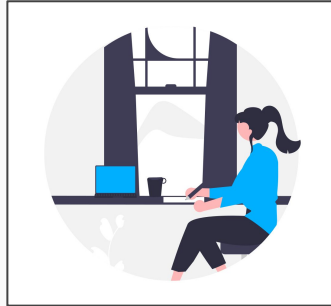


Vorlesung

=

Verstehen

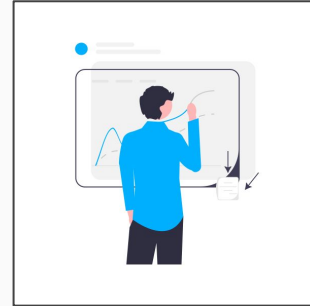
(Wissensvermittlung)



Selbststudium

=

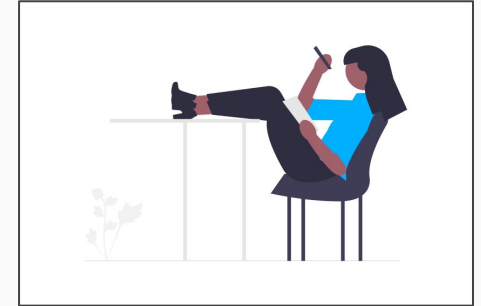
Vertiefen



Übung

=

Anwenden



Prüfung

=

Übertragen

Inhaltliche Orientierung



- Prog2-App + Discord
- Ablauf der Vorlesung
- Ablauf der Übungen
- Gemeinsame Klausur ←

Vorlesungsfolien ←

Übungsblätter ←



Geplante Inhalt

- JUnit
- Generics
- Datenstruktur: Dynamische Arrays
- Datenstruktur: Verkettete Liste
- Iteratoren
- Abstrakte Datentypen (ADT):
Menge, Stapel
- Schlangen und Ringpuffer
- ADT Folge
- Comparable bzw. Comparator
- Typinferenzen
- ADT Suchbaum
- ADT Prioritätswarteschlange, Heap
- ADT Folge, ADT Zuordnung

- Änderungen vorbehalten -


Präsenz-Blöcke im Stundenplan

		Unser Stundenplan																																						
		09:00	09:15	09:30	09:45	10:00	10:15	10:30	10:45	11:00	11:15	11:30	11:45	12:00	12:15	12:30	12:45	13:00	13:15	13:30	13:45	14:00	14:15	14:30	14:45	15:00	15:15	15:30	15:45	16:00	16:15	16:30	16:45	17:00	17:15	17:30				
Mo										Übung C177 • BI Gruppe 2 • Kless										Vorlesung Hörsaal 1/2 • Kless								Übung C177 • BI Gruppe 1 • Kless												
										Übung C175 • BI Gruppe 3 • Balg																														
										Übung C181 • BI Gruppe 4 • Schneider																														
Di										Übung C063 • BCSP Gruppe 1 • Schneider																				Übung C177 • BI Gruppe 5 • Kless										
										Übung C177 • BCSP Gruppe 2 • Kless																														

Zusatzangebot

Online-Übung bei Sigrid Weil → Freitags von 09:00 bis 13:45 Uhr.

LEA-Kurs zur Lehrveranstaltung (Weil)

 2024 SS - Programmierung 2

[Inhalt](#) [Info](#) [Einstellungen](#) [Mitglieder](#) [Lernfortschritt](#) [Metadaten](#) [Export](#) [Rechte](#) [Voransicht als Mitglied aktivieren](#) ➤

[Zeigen](#) [Verwalten](#) [Sortieren](#)

[Neues Objekt hinzufügen](#) [Seite gestalten](#)


Herzlich willkommen im Kurs Programmierung 2

Hier ein paar **wichtige Infos** zur Veranstaltung:

- Die **Vorlesungen** werden von Sigrid Weil oder André Kless gehalten. Die Inhalte sind identisch. Besuchen Sie die Vorlesung, die in Ihren Stundenplan passt.
- Die **Übungsgruppen** sind frei wählbar, die Dozent:innen gestalten die Übungsgruppen aber nach unterschiedlichen Konzepten. Sollte in einer Übungsgruppe kein Platz mehr im Raum frei sein, besuchen Sie bitte eine andere Übungsgruppe. Schauen Sie auch in die Stundenpläne der anderen Bachelor-Studiengänge!
- Die Übung von Sigrid Weil findet freitags 9:00 - 13:45 Uhr **online** statt. Die Zugangsdaten finden Sie weiter unten auf dieser Seite.
- Es wird für alle Studiengänge eine gemeinsame schriftliche **Klausur** geben. Hierzu gibt es **keine Zulassungsvoraussetzung**, es gibt kein Bonuspunktesystem, es werden keine Hilfsmittel zugelassen sein.
- Die **Lernmaterialien** (Vorlesungen, Übungen, Beispiele, Literatur, etc.) finden Sie kapitelweise weiter unten bzw. in der App von André Kless (<https://kaul.inf.h-brs.de/ccm/prog2/ss24/>)



Achtung: Es macht keinen Sinn an "Programmierung 2" teilzunehmen, wenn Sie den Stoff aus "Einführung in die Programmierung" nicht beherrschen bzw. die Klausur nicht bestanden haben. Wenn Sie "Einführung in die Programmierung" nicht bestanden haben, ist es sinnvoll **aktiv** am Kurs "Einführung in die Programmierung" für Wiederholer:innen von Sigrid Weil teilzunehmen.

Die **online-Übung** am Freitag ist als "freie Werkstatt" organisiert: Sie können den (virtuellen) Raum betreten und verlassen, wann Sie wollen. Allerdings gilt: **Sie müssen sich aktiv beteiligen**, indem Sie mit anderen Studierenden (verbal, dh über Mikro) kommunizieren, indem Sie bereit sind Ihren Bildschirm zu teilen und anderen Studierenden so Ihre (Teil-)Lösungen zu zeigen, indem Sie also aktiv an der Erarbeitung der Lösungen mitwirken!

Zugangsdaten zur online-Übung:
[online-Übung](#) 



Meeting-Kennnummer (Zugriffsscode): 2789 007 0147
Meeting Passwort: 8aZ8TM4zhq

Foren

 **Diskussionsforum Programmierung 2 2024ss** 

Hier können Sie über Inhalte der Veranstaltung diskutieren. Bitte halten Sie sich an Regeln der "Netiquette", wenn Sie im Forum diskutieren. Dieses Forum ist anonym, es ist jedoch sinnvoll, wenn Sie unter einem einheitlichen Pseudonym schreiben.

Beiträge (Ungelesen): 0 (0) Forum anonymisiert: Ja

 **Newsforum Programmierung 2 2024ss** 

Hier finden Sie alle wichtigen Ankündigungen.

Beiträge (Ungelesen): 0 (0)

Prog2-App + Discord-Server zur Lehrveranstaltung (Kless)

The screenshot shows the Prog2-App interface. At the top, it says 'Programmierung 2 - Kless' with a user profile for 'André Kless' and a 'Logout' button. Below this is a blue header for 'Kapitel 1: Interfaces, JUnit, Lambda-Ausdrücke'. The main content area lists the following items:

- Vorlesung
- Übung 1
- Beispiel: Roboter
- Beispiel: Lambda
- U1A1 Testklasse
- Literatur

At the bottom, there is a navigation bar with icons and labels for: Discord, Kommentare, Kurs-Infos, LEA-Kurs, and Videos.

The screenshot shows the Discord server interface for 'Programmierung 2 - Kless'. The server name is 'regeln'. The main channel is 'regeln'. The server description says: 'Willkommen bei Programmierung 2 - Kless. Das ist dein funkelnelgeheuer Server. Hier sind einige Schritte für den Anfang. Weitere Infos findest du im [Anfängerleitfaden](#).' Below this are four onboarding steps, each with a green checkmark:

- Freunde einladen
- Server mit Icon personalisieren
- Deine erste Nachricht senden
- Deine erste App hinzufügen

The date '4. April 2024' is shown. Below this is a message from 'M.Sc. André Kless' dated 'gestern um 14:20 Uhr' with the title 'Regelwerk'. The rules listed are:

- Ein Respektvoller Umgang ist allzeit geboten
- Wer sich unbedingt streiten möchte tut dies bitte in seinen DMs
- Rechte Hetze wird unverhandelbar mit einem sofortigen Bann bestraft
- Diskriminierende Inhalte jeglicher Art sind untersagt
- Pornografische Inhalte sind unerwünscht

Below the rules is a 'Disclaimer' section:

Disclaimer
Dieser Discord ist für den fachlichen Austausch im Bezug auf die Vorlesung Programmierung 2 gedacht. Überlegen Sie sich vorher bitte immer, ob Ihre Nachricht angebracht ist.
Wir behalten uns vor, jegliche Inhalte kommentarlos zu löschen, auch wenn diese nicht explizit gegen eine der genannten Regeln verstoßen. (Bearbeitet)

At the bottom, there is a search bar and a message input field with the placeholder 'Nachricht an #regeln'.

Kommunikation

- Fragen zu Folien stellen Sie bitte direkt unter der Folie als Kommentar in der App.
- Für alles andere nutzen wir Discord: <https://discord.gg/n2kqFVEJtK>
- Persönliche Anliegen an andre.kless@h-brs.de
(oder markus.schneider@h-brs.de, oder moritz.balg@h-brs.de)
- Für den persönlichen Austausch in Präsenz nutzen Sie Vorlesung und Übung.
- Auf Anfrage können Sie auch eine persönliche Sprechstunde vereinbaren.

Inhalt

- 1 Vorbereitung
 - Wiederholung: Interfaces
 - Testen mit JUnit
 - Funktionale Interfaces und Lambda-Ausdrücke

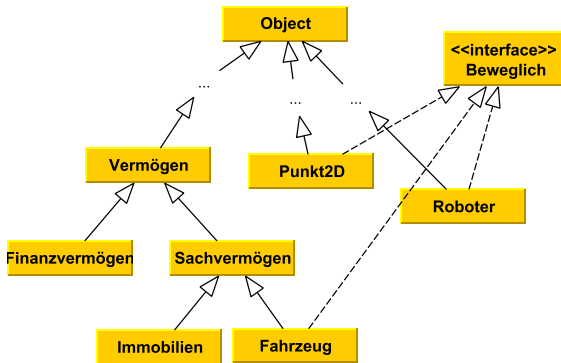
Begriffe und Konzepte aus EidP

Was wir dringend benötigen:

- ▶ Variablenkonzept, Typen: simple types vs. Referenztypen
- ▶ Anweisungen: Verzweigung, Schleifen
- ▶ Arrays
- ▶ statische vs. nicht-statische Komponenten
- ▶ Klassenhierarchie, Vererbung, Abstrakte Klassen
- ▶ Rekursion
- ▶ Exceptions
- ▶ Interfaces

UML-Beispiel: interface Beweglich

„Gestrichelte“ Pfeile für Implementierung von Schnittstellen:



Interfaces - Schnittstellen

- ▶ verhalten sich z.T. wie abstrakte Klassen:
 - Interfaces beschreiben **Typen**
 - enthalten (i.d.R.) nur Methoden**köpfe**, keine Implementierungen
 - müssen durch (konkrete) Klassen implementiert werden
 - können sub-interfaces haben
 - können durch mehrere Klassen implementiert werden
- ▶ aber auch nicht:
 - Interfaces sind **keine Klassen**, deshalb müssen sie sich nicht in den Baum der Klassen-Hierarchie einordnen.
 - eine Klasse kann **mehrere Interfaces** implementieren

Das Liskov'sche Substitutionsprinzip

schon bekannt: „subtyping“, dh Zuweisungen der Art
Obertyp var = (*Wert von Untertyp*);
sind zulässig.
(Entsprechendes gilt zB für Parameterübergaben an Methoden.)

„Interfaces beschreiben **Typen**“

- ▶ Interfaces sind Subtypen von Object
- ▶ Unter-Interfaces sind Subtypen von Ober-Interfaces
- ▶ implementierende Klassen eines Interfaces sind Untertypen des Interface

Code-Beispiel: interface Beweglich

```
public interface Beweglich {  
    Punkt2D position(); // aktuelle Position  
    void moveTo(Punkt2D p); // aendert aktuelle Position  
}  
  
public class Roboter implements Beweglich {  
    // ...  
    private Punkt2D standort;  
    // ...  
    public Punkt2D position() {  
        return this.standort;  
    }  
  
    public void moveTo(Punkt2D p) {  
        this.standort = p;  
    }  
}
```

Interface nutzen

Zur Nutzung von Interfaces braucht man i.d.R mindestens

- ▶ die Definition des Interfaces
- ▶ eine implementierende Klasse
- ▶ eine Methode/Klasse, in der für Objekte (vom Typ des Interfaces) Methoden des Interfaces aufgerufen werden

Guter Programmierstil: „gegen“ das Interface programmieren
also: wo möglich

- ▶ als Typbezeichnung das Interface verwenden
- ▶ nur Methoden nutzen, deren Existenz durch das Interface garantiert ist

Beispiel: Interface nutzen

- ▶ Variable von Interface-Typ deklarieren, mit Wert von Klassentyp belegen

```
Beweglich br = new Roboter();  
br.moveTo(...);
```

- ▶ Definition von Methoden mit Interfacetyp als (Ein- oder Rückgabe-)Typ von Parametern

```
public void meth1(Beweglich b) { ... }  
public Beweglich meth2() { ... }
```

- ▶ Aufruf mittels

```
Beweglich br = new Roboter();  
meth1(br);  
Beweglich r2d2 = meth2();
```

Inhalt

- 1 Vorbereitung
 - Wiederholung: Interfaces
 - **Testen mit JUnit**
 - Funktionale Interfaces und Lambda-Ausdrücke

Testgetriebene Entwicklung - testdriven development

Ziel:

- ▶ Trennung von ApplikationsCode und TestCode
- ▶ **Frühzeitiges** Entdecken von Fehlern
- ▶ Vermeiden von „Verschlimmbesserung“ bei späteren Änderungen des ApplikationsCodes

Idee:

- ▶ Identifiziere (typische) Anwendungsfälle
- ▶ Identifiziere Sonderfälle, Grenzfälle, Ausnahmen
- ▶ Implementiere für **jeden Fall** (use case) eine Testmethode
- ▶ Durchlaufe nach jeder (wesentlichen) Änderung des ApplikationsCodes **alle** Testfälle

JUnit

bei uns: JUnit5

- ▶ Framework zur Automatisierung von Tests in Java
- ▶ in den meisten gängigen IDEs (Eclipse, IntelliJ, ...) integriert

Umgang damit:

- ▶ Definiere eine Testklasse (Namenskonvention beachten)
- ▶ ... mit ggf. mehreren (vielen) Testmethoden.
- ▶ In den Testmethoden werden **zu testende Methoden** der ApplicationKlasse aufgerufen und ihr Ergebnis mit den zu erwartenden Ergebnissen verglichen.

Annotations

Vielleicht schon mal gesehen:

- ▶ zusätzliche „modifier“ im Java-Code
- ▶ werden durch vorangestelltes „@“ (at-Zeichen) gekennzeichnet
- ▶ geben „Meta“-informationen, die zB vom Compiler verarbeitet werden können

Beispiele:

`@Override`

`public String toString() { ... }`

`@SuppressWarnings`

Einige der in JUnit genutzten Annotations

`@Test` // deklariert die Methode als Testmethode

`@BeforeAll` // ("setup")

`@BeforeEach` // ("init")

`@AfterAll`

`@AfterEach`

In der Testklasse erforderlich:

```
import static org.junit.jupiter.api.Assertions.*;
```

```
import org.junit.jupiter.api.BeforeEach;
```

```
import org.junit.jupiter.api.Test;
```

```
// ...
```

Assertions

Assertions sind Methoden der Bibliothek `org.junit.jupiter.api.Assertions`, mit deren Hilfe das Ergebnis der zu testenden Methode mit dem **erwarteten** Ergebnis verglichen werden kann.

Einige häufig genutzte assertions:

```
assertTrue(boolean condition)
```

```
assertNotNull(Object obj)
```

```
assertEquals(Object expected, Object actual)
```

```
assertEquals(int expected, int actual)
```

```
// auch fuer andere simple types
```

```
// Achtung bei double: Rundungsfehler bedenken, deshalb
```

```
assertEquals(double expected, double actual, double delta)
```

```
...
```

Inhalt

- 1 Vorbereitung
 - Wiederholung: Interfaces
 - Testen mit JUnit
 - Funktionale Interfaces und Lambda-Ausdrücke

Motivation

- ▶ Ziel: Klasse mit statischer Methode `printTabelle()`, die eine Wertetabelle für Funktionen druckt.
- ▶ Eingabeparameter: Bereich (double-Werte) von - bis, Schrittweite d, (Methode ?) **Funktion** m

Beispiel:

Für $f(x) = x^2$ bzw $g(x) = 0.5 * x$ sollten solche Aufrufe

```
printTabelle(0.0, 1.0, 0.1, f);
printTabelle(-2.0, 2.0, 1.0, g);
```

einen solchen (oder ähnlichen) output erzeugen:

x	0.0	0.1	0.9	1.0
y	0.0	0.01	0.81	1.0

x	-2.0	-1.0	0.0	1.0	2.0
y	-1.0	-0.5	0.0	0.5	1.0

Kein (!) möglicher Java-Code

```
public class Wertetabelle {  
  
    public static void printTabelle(double von, double bis,  
        double d, Funktion methode) {  
  
        // Ueberschrift der Tabelle  
        for (double x = von; x <= bis; x = x+d) {  
            System.out.print(x + \t);  
        } System.out.println();  
  
        // Werte der Tabelle  
        for (double x = von; x <= bis; x = x+d) {  
            // das geht nicht!  
            System.out.print(methode(x) + \t);  
        } System.out.println();  
    }  
}
```

Lösung mittels „Funktionalem Interface“

```
public interface Funktion{  
    double eval(double x);  
}  
  
public class Parabel implements Funktion {  
    public double eval(double x){  
        return x*x;  
    }  
  
public class Halbe implements Funktion {  
    double eval(double x){  
        return 0.5 * x;  
    }  
}
```

So geht's

```
public class Wertetabelle {  
  
    public static void printTable(double von, double bis,  
        double d, Funktion m) {  
        // Ueberschrift der Tabelle wie oben  
  
        for (double x = von; x <= bis; x = x+d) {  
            // das geht!  
            System.out.print(m.eval(x) + "\t");  
        }  
        System.out.println();  
    }  
}
```

Mögliche Aufrufe:

```
Funktion f = new Parabel();  
Funktion g = new Halbe();  
printTabelle(0.0, 1.0, 0.1, f);  
printTabelle(-2.0, 2.0, 1.0, g);
```

Funktionales Interface

- ▶ Interface, in dem nur **eine einzige** Methode deklariert wird
- ▶ ermöglicht **Übergabe** von Methoden als Parameter an andere Methoden

Erfordert:

- ▶ Definition des Interface
- ▶ Definition (mindestens) einer implementierenden Klasse
- ▶ Erzeugung eines Objektes dieser Klasse
- ▶ Aufruf der Methode für dieses Objekt

Lambda-Ausdrücke

Sogenannte **Lambda**-Ausdrücke ermöglichen es, funktionale Schnittstellen wesentlich kompakter zu nutzen:

Mithilfe von Lambda-Ausdrücken

- ▶ kann man funktionale Schnittstellen implementieren
- ▶ in Form eines **Ausdrucks**!
- ▶ Dadurch lässt sich **ausführbarer** Code an Methoden übergeben.

Syntax:

```
( lambdaParams ) -> { Anweisungen }
```

Beispiel Lambda-Ausdruck

- ▶ Definition der Schnittstelle Funktion wie oben
- ▶ Definition der Methode `printTabelle()` wie oben
- ▶ Mögliche Aufrufe:

```
printTabelle(0.0, 1.0, 0.1, (x) -> {return x*x;});
```

```
printTabelle(-2.0, 2.0, 1.0,  
             (x) -> { return 0.5 * x; } );
```

- ▶ Keine (explizite) Implementierung der Schnittstelle durch Klasse `Parabel` oder `Halbe` nötig
- ▶ Keine Erzeugung von Objekten dieser Klasse(n) nötig

Testen von Exceptions

Wo nutzen wir Lambda-Ausdrücke?

- ▶ innerhalb der JUnit-Tests
- ▶ zum Testen von Exceptions

Beispiel

- ▶ Klasse Roboter
- ▶ besitzt ein `int`-Attribut `richtung`
- ▶ erlaubte Werte: `{0, 1, 2, 3}` (für „Nord“, „Ost“, „Sued“, „West“)
- ▶ Methode `turn(int richtung)` soll bei negativen Eingabewerten eine `IllegalArgumentException` auslösen

JUnit-Test:

@Test

```
void testException() {  
    Roboter r = new Roboter();  
    assertThrows (IllegalArgumentException.class ,  
                  () -> {r.turn(-1);} );  
}
```