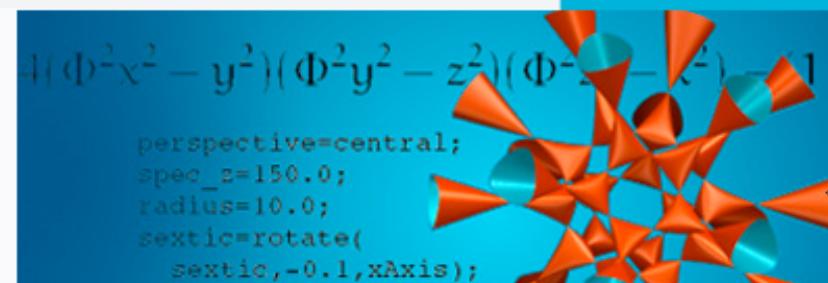


Objektorientierte Programmierung

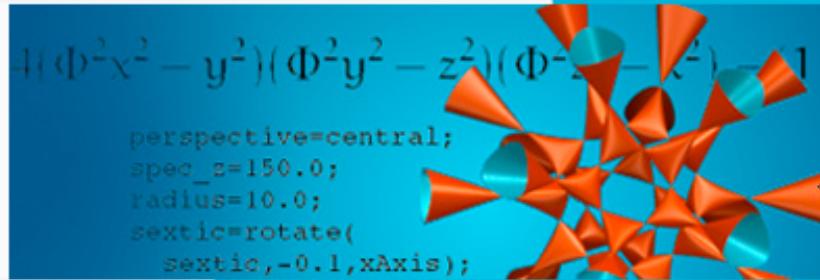
Wintersemester 2019/20

Prof. Christoph Bockisch, Stefan Schulz
(Programmiersprachen und –werkzeuge)





Organisation



4(Φ²x² - y²)(Φ²y² - z²)(Φ²z² - x²) - 1
perspective=central;
spec_z=150.0;
radius=10.0;
sextic=rotate(
sextic,-0.1,xAxis);

Kontakt

- Prof. Christoph Bockisch
 - Raum: Mehrzweckgebäude, 05D15
 - Email: bockisch@mathematik.uni-marburg.de
 - Telefon: 06421/28 - 21515
- Übungsorganisation: Stefan Schulz
 - Raum: Mehrzweckgebäude, 05D04
 - Email: schulzs@mathematik.uni-marburg.de
 - Telefon: 06421/28 - 25419
- Tutoren
 - Entweder über Ilias oder nach Absprache in der Übungsgruppe

Organisation des Moduls

- **Vorlesung**

- Vorlesung: 4 SWS

- Mo 12:00–14:00 in HG +1/0020
- Mi 12:00–14:00 in HG +1/0020

genaue Zeiten → später

- **Übungen**

- Besuch eines Tutoriums (2 SWS)

- **Klausur:** 12.02. 12:00 – 15:00 im HG +1/0020

- **Zweitklausur:** 18.03. 13:00 – 16:00 im HC +5/0010 (**Lahnberge**)

Lernplattform Ilias

- Jeder Teilnehmer des Moduls muss sich bei der Lernplattform Ilias für die Vorlesung anmelden.
 - Magazin > ILIAS: Kurse aller Semester > Fb. 12: Mathematik und Informatik > Informatik > Softwaretechnik > WiSe 2019/20 > Bockisch: VL Objektorientierte Programmierung
 - <https://uni-marburg.de/ejLLm>
- Service über Ilias
 - Folienskript
 - Übungsaufgaben
 - Codebeispiele
 - Tutorien
 - Forum
 - ...



Übungsbetrieb

- Zuständiger Mitarbeiter: Stefan Schulz
- 12 Übungszettel + 1 Bonuszettel
 - Zettelausgabe im ILIAS im Ordner "Aufgabenblätter"
 - Bearbeitungszeit Mittwoch 14 Uhr bis Mittwoch 12 Uhr
 - Abgabe in Gruppen à 3 Personen über ILIAS
 - **Der erste Übungszettel ist ausnahmsweise eine Einzelabgabe**
 - Forum zur Gruppensuche im ILIAS
- Präsenzaufgaben in den Tutorien begleitend zu den Hausübungen
- **Start der Übung in der 2. Semesterwoche**

Übungsbetrieb

- Termine für Tutorien siehe ILIAS
- Anmeldung zu den Tutorien über ILIAS ab dem 16.10. um 18 Uhr
 - Begrenzt auf 30 Plätze pro Tutorium
 - Tauschbörse für Tutorienplätze im ILIAS (Diskussionsforum)
 - Bis zum 30.10., 14 Uhr können Sie noch selbst die Gruppe wechseln. Danach muss der Wechsel durch die Tutoren vorgenommen werden.
- Tutorien 13 & 14 finden in der Stadtmitte statt
 - Freitag 14:00 – 16:00 und 16:00 – 18:00

Übungsbetrieb

- Inhalt vor allem: technische Vorbereitung (Installation der benötigten Software, etc.)
 - Probieren Sie, Installationen bereits vor dem Tutorium durchzuführen
 - Nutzen Sie das Tutorium für technische Unterstützung
- Einige Übungsaufgaben in den Tutorien erfordern Programmierung: bringen Sie – wenn möglich – Ihr Notebook mit

Prüfungs- & Studienleistung

- Voraussetzung für Klausurzulassung:
 - Maximal 2 Übungsblätter unbearbeitet
 - Ein abgegebener Zettel mit 0 Punkten zählt als unbearbeitet
 - Mindestens 50% der Gesamtpunktzahl in den Übungen
- Bonus:
 - 1 Notenpunkt
 - Voraussetzung:
 - Mindestens 80% der Übungspunkte bei bestandener Klausur
 - Vorstellung einer Präsenzaufgabe im Tutorium

Literaturliste

- Einführung in die Informatik
 - H.-P. Gumm, M. Sommer: „Einführung in die Informatik“, 10. Auflage, Oldenbourg, 2012
 - **Zugreifbare im Uninetz:**
<https://www.degruyter.com/viewbooktoc/product/216866>
 - H.-P. Gumm, M. Sommer : Grundlagen der Informatik: Programmierung, Algorithmen und Datenstrukturen (De Gruyter Studium) Taschenbuch – 26. September 2016
 - **Zugreifbar im Uninetz:**
<https://www.degruyter.com/viewbooktoc/product/460906>
- Zu Java gibt es empfehlenswerte Lehrbücher als pdf, die frei zur Verfügung stehen:
 - Christian Ullensboom: „Java ist auch eine Insel“, Galileo Computing, 11. Auflage, 2016. <http://openbook.galileocomputing.de/javainsel//>
 - Guido Krüger, Heiko Hansen:
Handbuch der Java-Programmierung, O'Reilly. <http://www.javabuch.de/>



Hinweis zu meinen Folien

- Der Foliensatz basiert auf einem früheren Skript, erarbeitet von Herrn Prof. Bernhard Seeger zusammen mit Herrn Prof. Andreas Henrich (Uni Bamberg)
- Einige Kapitel sind den Foliensätzen von meinen Kollegen Prof. Gumm und Prof. Sommer entnommen worden.
- Alle Fehler gehen aber natürlich trotzdem zu meinen Lasten.
 - Ich bitte um Ihre Rückmeldung!

Hinweis zum Modul

- Ziel
 - Interessante Gestaltung des Moduls sowohl für Studierende ohne Informatikerfahrung als auch diejenigen mit Erfahrung.
 - Vermittlung von Schlüsselkompetenzen der Informatik
- Verpflichtung des Dozenten und sein Team
 - Optimale Vorbereitung der Vorlesung und des Übungsbetriebs
- Verpflichtung der Studierenden
 - Vorlesungen mitverfolgen und nacharbeiten
 - Übungsaufgaben **selbstständig** lösen

Hinweis zum Modul

- Ziel
 - Interessante Gestaltung des Moduls sowohl für Studierende ohne Informatikerfahrung als auch diejenigen mit Erfahrung.
 - Vermittlung von Schwerpunktkompetenzen der Informatik
- Verpflichtungen
 - Optimaler Betrieb
- Verpflichtungen
 - Vorlesungen mitverfolgen und nacharbeiten
 - Übungsaufgaben **selbstständig** lösen

Vorlesung/Übungsaufgaben zu
einfach oder zu schwer?

→ Statt Resignation und Frust
frühzeitige Rückmeldung an uns!

Ziele der Vorlesung

- Einführung in die wichtigsten Konzepte der (objektorientierten) Programmierung
 - Leichtes Erlernen einer anderen objektorientierten Sprache
- Vertiefende Kenntnisse einer objektorientierten Programmiersprache (Java)
 - aber ohne den Anspruch der Vollständigkeit
- Typische Programmiermuster
- Dieser Kurs ist **nicht** nur ein Java-Programmierkurs!
 - Querbezüge zu anderen Programmiersprachen wie z. B. C und Scala
 - Interne Implementierungsaspekte
 - Theoretische Grundlagen

Themenübersicht

- Einführung
- Algorithmen
- Datentypen
- Klassenkonzept in Java
- Schnittstellen
- Klassenerweiterung
- Parametrisierte Klassen
- Ausnahmebehandlung
- Standardbibliothek
- Fortgeschrittene Konzepte: generische Datentypen, Enumerationen, Stream I/O, Lambdas

Terminplan

- Jede Woche
 - 2 Vorlesungen
 - 1 Präsenzübung
 - 1 Hausübung
- Ausnahmen
 - Woche vor Weihnachten
 - Probeklausur am Mittwoch (18.12.) während der Vorlesungszeit
 - Vorletzte Woche
 - Frage & Antwortstunde am Mittwoch (05.02.)
 - Bonus-Hausübungszettel mit der Hälfte der Punkte
(Achtung: Abgabe bereits am Sonntag, den 09.02.)
 - Letzte Woche
 - Bei Bedarf Frage & Antwortstunde am Montag (10.02.)
 - Klausur am Mittwoch (12.02.)

Vorlesungsstil

- Materialien
 - Folienskript
 - Soweit möglich Literaturverweise
 - Achtung: evtl. ist die Vorlesung manchmal ausführlicher als die Folien!
- Demonstrationen
 - Live-Programming
- Formative Tests
 - Neben den Übungen können Sie in Live-Votes während der Vorlesung Ihr Wissen und Verständnis überprüfen



Live-Vote

PIN: KC2M

x

<https://ilias.uni-marburg.de/vote/KC2M>



1. Einführung

Informatik

- „ist die **Wissenschaft**, **Technik** und **Anwendung** der maschinellen **Verarbeitung**, **Speicherung** und **Übertragung** von **Information**.“
(Broy, 1992)
- Der Name **Informatik** ist ein Kunstwort aus **Information** und **Mathematik** (geprägt in den 60er Jahren).

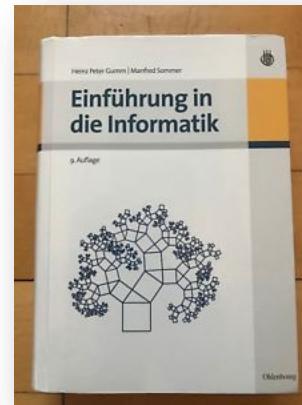
Wichtige Aspekte in der Informatik

- Repräsentation von Information
- Informationsverarbeitung
- Verarbeitungsvorschriften
- Informationsverarbeitende Maschinen

Repräsentation von Information

- Viele Abstraktionsebenen

Interpretation



ISBN 978-3-486-70641-3

01001001 01010011 01000010 01001110
00100000 00111001 00110111 00111000
00101101 00110011 00101101 00110100
00111000 00110110 00101101 00110111
00110000 00110110 00110100 00110001
00101101 00110011



Repräsentation

Typische Aufgabe in der Informatik

- Auswahl der geeigneten Hilfsmittel



- Verwendung der Hilfsmittel um reale Probleme zu lösen



Die zwei Gesichter der Informatik

- Informatik als Mittel zum Zweck
 - Lösung konkreter **Probleme** aus der realen Welt
- Informatik als Methodenwissenschaft
 - Was sind gute **Werkzeuge** zur Lösung konkreter Probleme?

Informatik zur Problemlösung

- Anwender hat ein konkretes Problem, das von einem Informatiker unter Verwendung eines Computers gelöst werden soll.
- Vorgehensweise des Informatikers
 1. Analysieren des Problems
 2. Erstellung eines Modells der realen Welt
 3. Umsetzung des Modells auf einen Rechner
 - Auswahl der Implementierungswerzeuge
 4. Testen und Dokumentieren der Lösung

Typische Beispiele realer Probleme

- Erstellung eines Online-Portals für Bank XYZ
- Erstellung eines Energiecockpits für ein mittelständisches Unternehmen.
- Erstellung eines Raumverwaltungssystems für die Uni Marburg

Informationsverarbeitung

- Reales Problem
 - Gegeben: Tagesumsätze eines Unternehmens
 - Gesucht: Wochen-, Monats- und Jahresumsatz
- Software-Werkzeuge als Hilfsmittel, um schneller, einfacher, besser reale Probleme zu lösen.
 - Programmiersprache
 - Formale Sprache zur „einfachen“ Lösung des Problems
 - Andere Hilfsmittel
 - Editoren, Entwicklungsumgebungen, Software-Bibliotheken

Informatik zur Werkzeugentwicklung

- Gruppe von ähnlichen Problemen aus der realen Welt
 1. Generierung eines abstrakten Problems
 2. Lösung des abstrakten Problems
 3. Bereitstellung der Lösung als Werkzeug
→ „Framework“
- Wichtige Eigenschaft
 - Lösung konkreter Probleme wird einfacher, wenn die Lösung des abstrakten Problems verwendet wird.

Beispiel

- Entwicklung eines Informationssystems
 - Problemklasse
 - Viele Unternehmen besitzen wichtige Ressourcen, aber es gibt keine einheitliche Verwaltung der Ressourcen
 - Lösung
 - SAP/R3-System zur einfachen Verwaltung der Ressourcen

1.1 Programmiersprachen

- **Die Programmiersprache bildet die Schnittstelle zwischen Mensch und Computer bei der Entwicklung von Lösungen konkreter Probleme und Werkzeugen**
 - Mittels einer Sprache werden die zuvor entwickelten Lösungen auf einem Computer umgesetzt.
- **Analogie zu einem Dolmetscher**

Anforderungen an eine Sprache

- einfach erlernbar
- ausdrucksstark
 - Es können damit ganz viele Probleme gelöst werden.
- effiziente Ausführung der Befehle auf dem Computer
 - Kurze Laufzeit eines Programms
- schnelles Verarbeitung während der Entwicklung

Maschinensprache

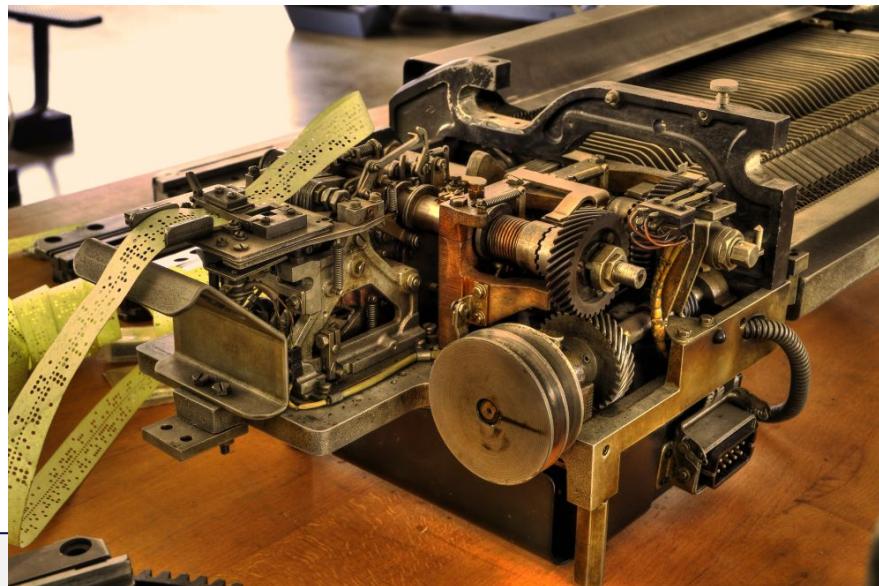
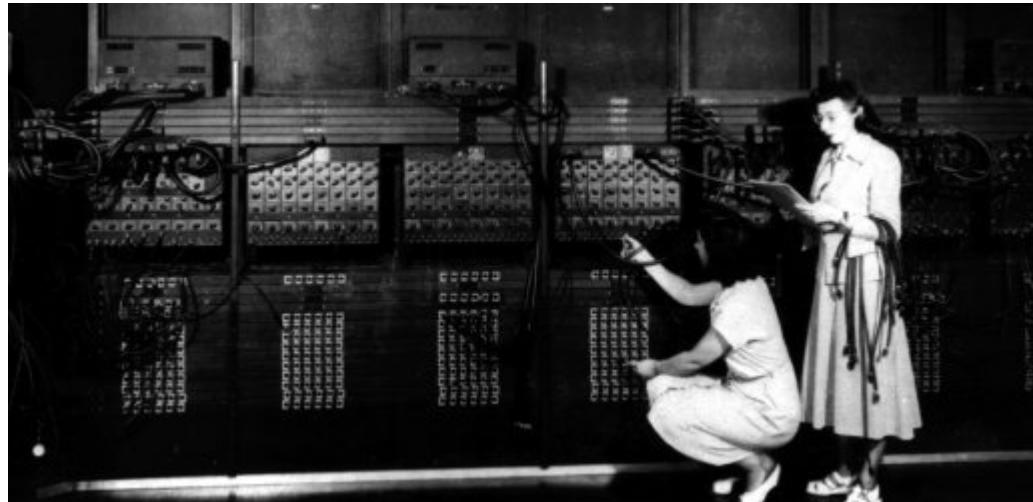
Computer verfügen bereits über

- einen Satz von elementaren Befehlen (Maschinensprache),
 - die direkt vom Computer verarbeitet werden können.
- und einfache Datentypen
 - die zur Repräsentation von Information genutzt werden können.
 - z. B. ganze Zahlen (bis zu einer gewissen Größe).

Maschinenprogramm

- Folge von Maschinenbefehlen zur Lösung eines Problems

Programmieren in der Antike



Beispiel

- Beispiel von Kommandos in Maschinensprache

ADD AX, FFh

INC AX

JMP 0A3

MOV AX, 5

- Diese Kommandos sind als Zahlen im Binärsystem kodiert (d.h. mit den Ziffern 0 und 1), etwa:

10001010 10100111 11010101 10010110 11010110 10101001 11110101 01011110
10110010 10100101

- Mit Maschinensprache kann man zwar prinzipiell alle Probleme lösen, die man auch in anderen Sprachen lösen kann, aber für Menschen sind Programme in Maschinensprache schwierig zu verstehen.

→ zu hohe Erstellungs- und Wartungskosten

Höhere Programmiersprachen

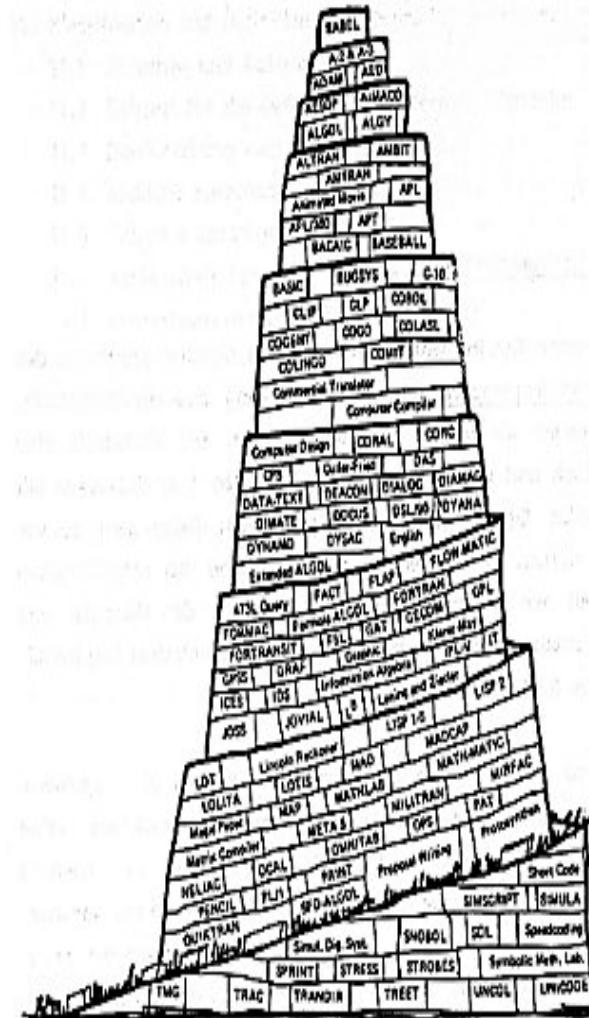
Idee

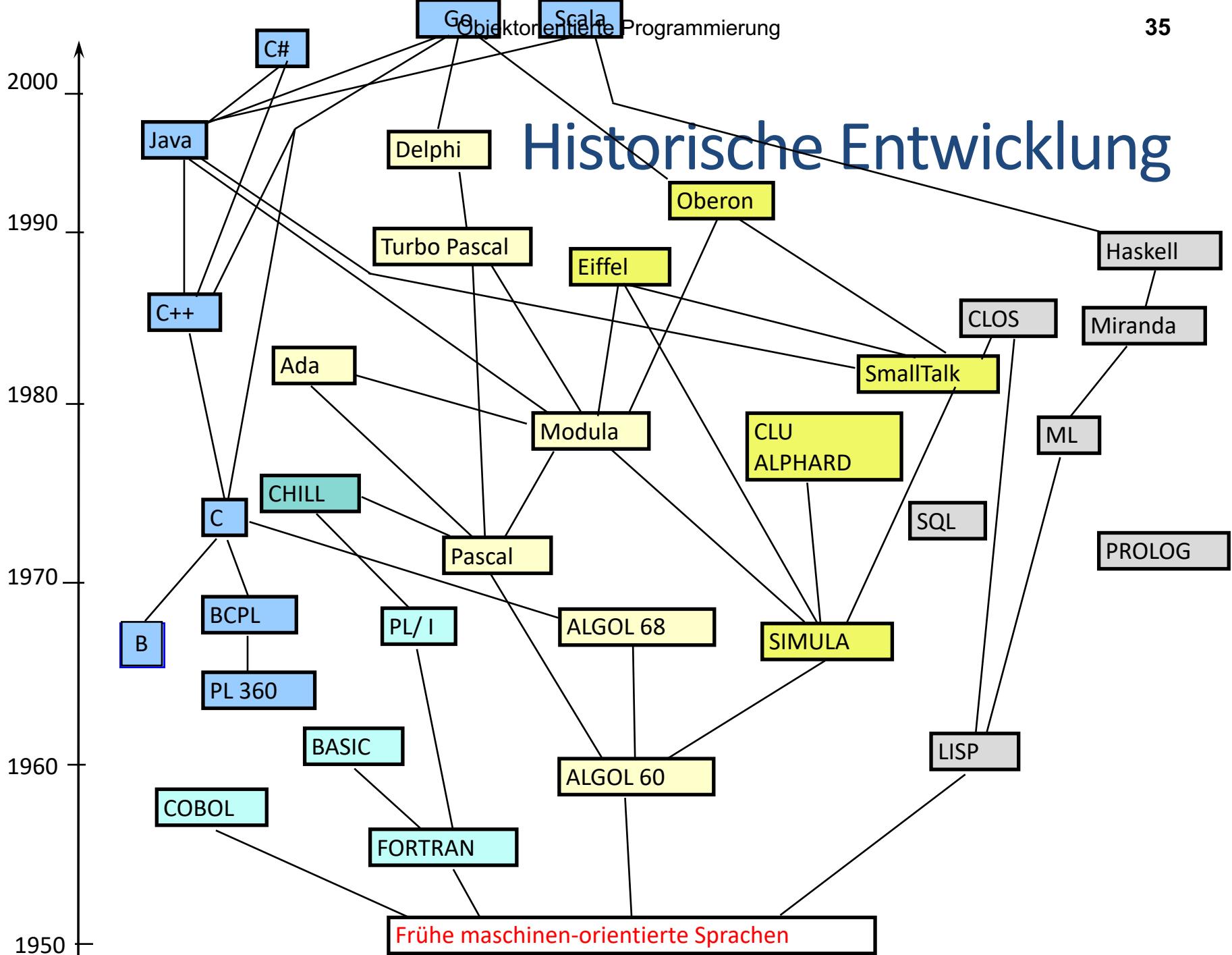
- Entwicklung von höheren Programmiersprachen, die eine einfache Umsetzung von Problemlösungen erlauben.
- Programmiersprachen bieten mächtige Methoden zur Abstraktion
 - a) Möglichkeit der Definition problemorientierter Datentypen (**Datenabstraktion**)
 - b) Mehrere Schritte, die sequentiell ausgeführt werden sollen, können zu einem Schritt zusammengefasst werden (**Kontroll- und Funktionsabstraktion**).
- Unterschiedliche Probleme erfordern verschiedene Programmiersprachen.
 - One-size-does-not-fit-all



Programmiersprachen

- Es gibt Tausende von Programmiersprachen
 - Allgemeine Sprachen, mit denen prinzipiell alle Fähigkeiten eines Rechners zugänglich sind
 - C, Pascal, C++, **Java**, C#, Scala, Go, Rust,...
 - Lisp, Prolog, ...
 - FORTRAN, ALGOL, LISP, BASIC, PL1, ...
 - Visual Basic, VB.NET
 - Spezialsprachen für bestimmte Anwendungen
 - JavaScript, ECMAScript, VBScript, ...
 - PHP, Perl, Python, Ruby ...
 - SQL, TeX, TCL/TK, ...
 - HTML, XHTML, XML, ...

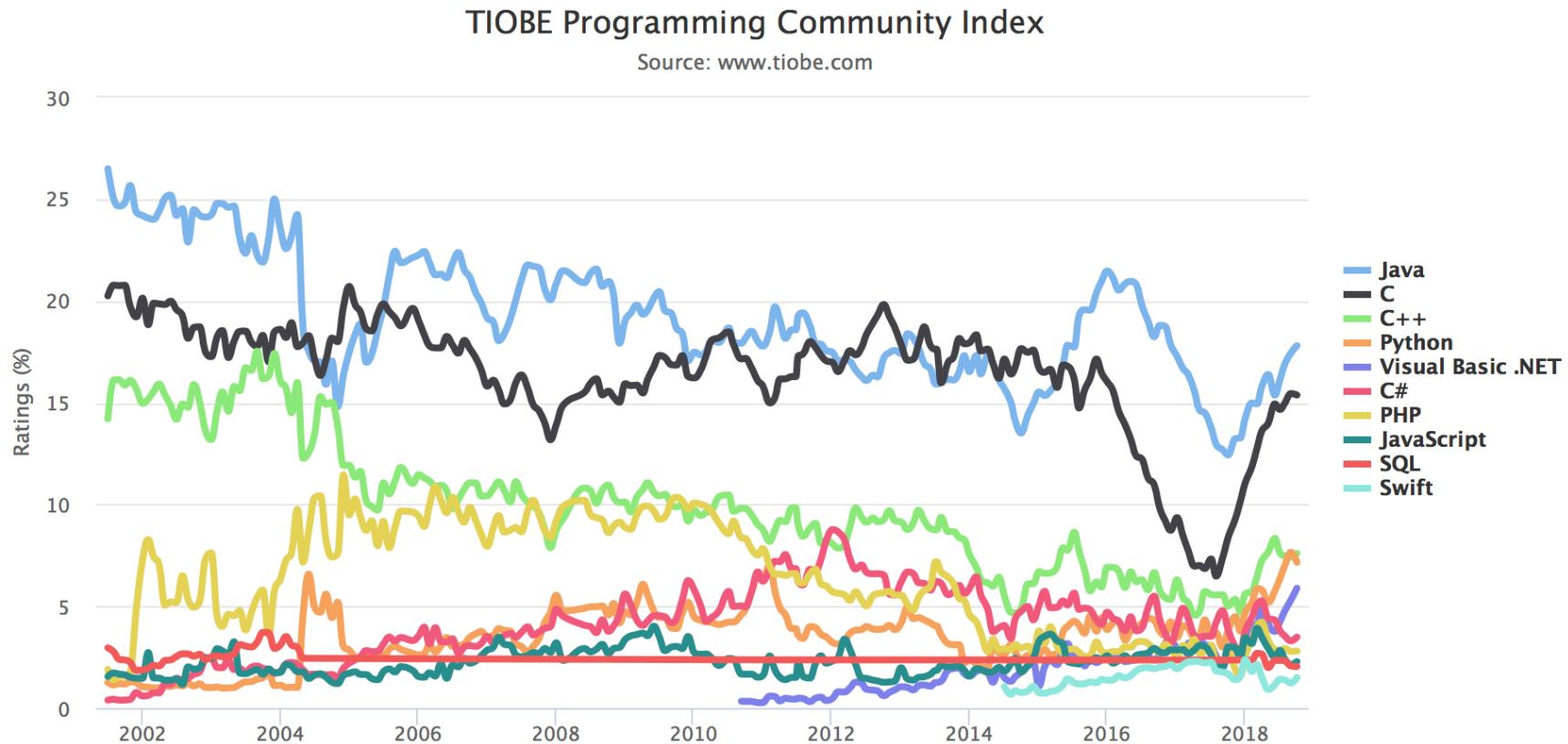




Klassen von Programmiersprachen

- **imperative Sprachen** unterstützen das zustandsorientierte Programmieren
 - Werte können in Variablen zwischengespeichert und verändert werden, auf die über einen Namen oder Adresse zugegriffen werden kann.
 - Beispiele: Pascal, C, Fortran, Cobol, ...
- **objektorientierte Sprachen** sind eine Weiterentwicklung imperativer Sprachen
 - Werte können nur innerhalb eines Objekts zwischengespeichert werden. Der Zustand eines Objekts kann durch Methoden verändert werden.
 - Beispiele: Eiffel, Smalltalk, C++, Java
- **funktionale Sprachen** betrachten ein Programm als eine mathematische Funktion, die zu einer Eingabe eine Ausgabe produziert
 - In diesen Sprachen gibt es keine explizite Zwischenspeicherung von Ergebnissen.
 - Beispiele: Lisp, Miranda, ML, Haskell
- **objekt-funktionale Sprachen** sind ein Hybrid zwischen objektorientierten und funktionalen Sprachen
 - Beispiele: C++14, C#, Scala, Java9
- **logik-basierte Sprachen**
 - Regeln zur Definition von Relationen.
 - Beispiel: Prolog, Datalog, SQL

Ranking von Programmiersprachen (?)





Die Geschichte von Java

- **1991:** James Gosling entwickelt mit einem kleinen Team bei der Kultfirma SUN die Programmiersprache **OAK** (Object Application Kernel).
 - Ziel zunächst: Programmierung von elektronischen Geräten der Konsumgüterindustrie
- **1995:** Java wird öffentlich vorgestellt und im gleichen Jahr bietet SUN kostenlos den Kern eines Programmiersystems **JDK** zusammen mit einer Implementierung der **JVM** an.
- **2010:** Mit der Übernahme von SUN durch die Firma Oracle verliert die Entwicklung von Java an Dynamik.
 - Andere JVM-basierende Programmiersprachen wie Scala werden als Alternative zu Java interessant.
- Im **Sept. 2019** wurde die Version 13 freigegeben und veröffentlicht.

Gründe für den Erfolg von Java

- Java ist zum richtigen Zeitpunkt veröffentlicht worden.
 - C-ähnliche Syntax hat viele dazu bewegt, den Schritt von C und C++ nach Java zu gehen.
 - Gelungene Umsetzung der objektorientierten Konzepte
- Freie Verfügbarkeit von Compiler und Infrastruktur
- Großes Angebot an vorgefertigten Werkzeugen in Java
- Java gilt als die Internet-Programmiersprache
 - Entwicklung einer Vielzahl von Apps und Spielen
 - z. B. Minecraft
- Java ist eine Anforderung bei vielen Jobs
→ [aktuelle Jobbörsen](#)

1.2 Erste Schritte mit Java

- Vorerst nur Entwicklung im REPL-Interface
 - Erste Schritte
 - Variablen
 - Deklaration
 - Zuweisung

Programmentwicklung

1. Klassische Entwicklungswerkzeuge

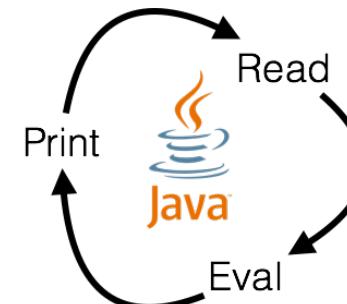
- Texteditor
- Übersetzer

2. Heute: Integrierte Entwicklungsumgebungen

- Anlegen von Projekten
- Zusätzlich werden Werkzeuge angeboten, um
 - Programme zu testen
 - Fehler zu finden
 - Programmentwicklung im Team

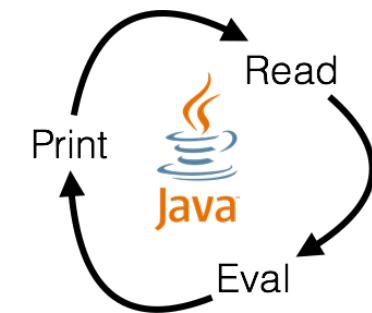


3. REPL-Interface (seit Java 9)



REPL-Schnittstelle

- Neue Möglichkeit um Java-Befehle direkt auszuführen
 - REPL = Read–Eval–Print Loop
- Vorgehensweise (Windows)
 - Kommandozeile cmd.exe aufrufen
 - Befehl jshell eingeben



A screenshot of a Microsoft Windows command prompt window titled "Eingabeaufforderung - jshell". The window shows the following text:
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\bhseeger>jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro

jshell> ■

Erste Befehle mit REPL

- Im Wesentlichen funktioniert alles wie bei einem Taschenrechner.
 - Die Ausgabe eines Ergebnisses ist leider etwas komplizierter.
- Beispiele

1. `System.out.print("Hallo Welt!")`

Gibt den Text zwischen den beiden Anführungszeichen aus.

2. `System.out.print(7)`

Gibt die ganze Zahl 7 aus.

3. `System.out.print(3.14)`

Gibt die Gleitpunktzahl 3.14 aus.

4. `System.out.print('a')`

Gibt das Zeichen a aus.

Erste Berechnungen

- Ähnlich zum Taschenrechner können im REPL Berechnungsschritte angegeben und ausgeführt werden.
- Beispiele

1. `System.out.print("Hallo" + " Hallo")`

Verknüpfung von zwei Texten und Ausgabe des Ergebnis.

2. `System.out.print(7 + 8)`

Addiert 7 + 8 und gibt die ganze Zahl 15 aus.

3. `System.out.print(3.14*2*2)`

Multipliziert die 3 Zahlen und gibt das Ergebnis aus.

Speichern von Zwischenergebnissen (1)

- Analog zum Taschenrechner kann man sich Werte im Zwischenspeicher merken.
 - Zwischenspeicher im REPL ist nahezu beliebig groß!
- Beispiele
 - 1. "Hallo Welt!"



Name des
Zwischenspeichers

```
C:\ Eingabeaufforderung - jshell
C:\Users\bhseeger>jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro

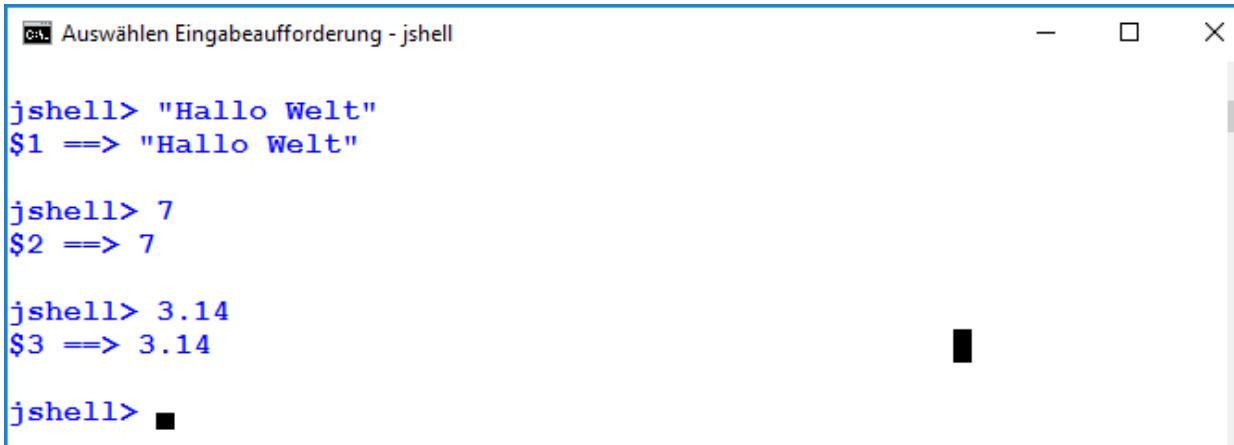
jshell> "Hallo Welt!"
$1 => "Hallo Welt!"

jshell> System.out.print($1)
Hallo Welt!
jshell> ■
```

Wiederverwendung des Werts
aus dem Zwischenspeicher.

Speichern von Zwischenergebnissen (2)

- Die Namen der Zwischenspeicher sind eindeutig.
 - Ein Name wird nur einmal vergeben.



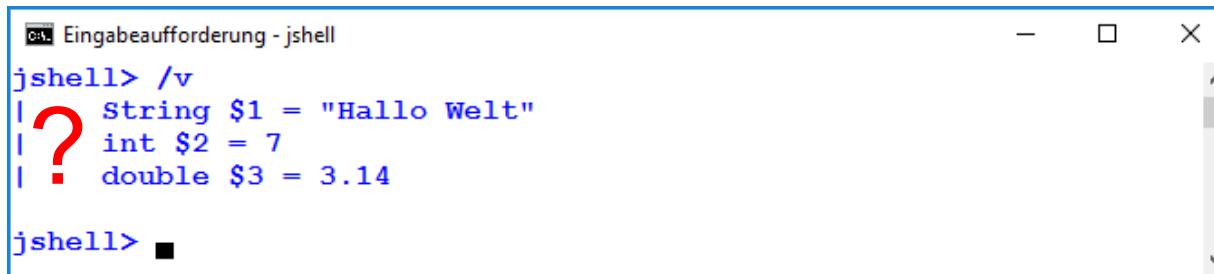
```
jshell> "Hallo Welt"
$1 ==> "Hallo Welt"

jshell> 7
$2 ==> 7

jshell> 3.14
$3 ==> 3.14

jshell> ■
```

- Die Liste aller gespeicherten Werte kann man durch den Befehl **/vars** oder kurz **/v** bekommen.



```
jshell> /v
| ? String $1 = "Hallo Welt"
| ? int $2 = 7
| ? double $3 = 3.14

jshell> ■
```

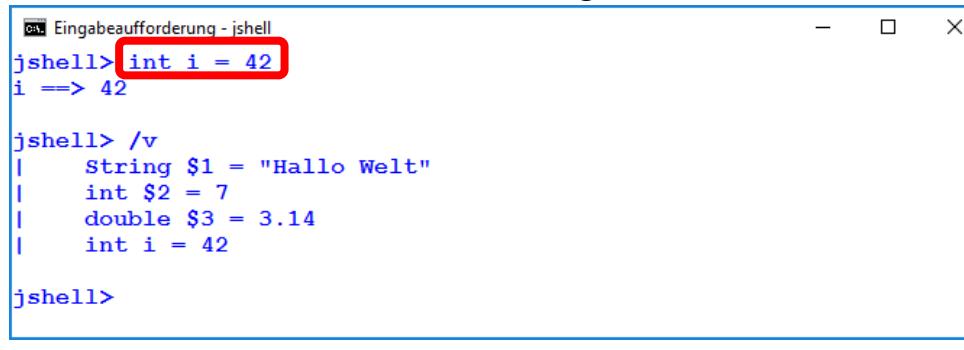
Datentypen

- Jeder Zwischenspeicher merkt sich noch zusätzlich aus welcher Wertemenge ein Wert ist.
 - "Hallo Welt!" hat den **Typ String** und repräsentiert die Menge aller Zeichenketten.
 - 7 hat den **Typ int** und repräsentiert die Menge der ganzen Zahlen in dem Bereich -2147483648 bis 2147483647
 - 3.14 ist vom **Typ double** und repräsentiert die Menge der Gleitpunktzahlen
- Zusätzlich zu der Wertmenge besitzt ein Datentyp **Operationen**
 - Datentyp int
 - + , - , * , / und %
 - Datentyp String
 - +



Variablen

- Statt nun einfach REPL zu überlassen, wie Speicherplätze benannt werden, kann man dies auch selbst tun.
 - Man spricht dann von einer **Variable**.
 - Bei der **Deklaration der Variable** muss immer der **Datentyp** angegeben werden.
 - Jede Variable darf nur einmal deklariert werden.
 - Zudem ist es empfehlenswert den Variablen einen Wert zuzuweisen.
 - Hierzu benutzt man den **Zuweisungsoperator** (Symbol =). Dies bedeutet, dass die Variable den Wert rechts von = zugewiesen bekommt.



```
jshell> int i = 42
i ==> 42

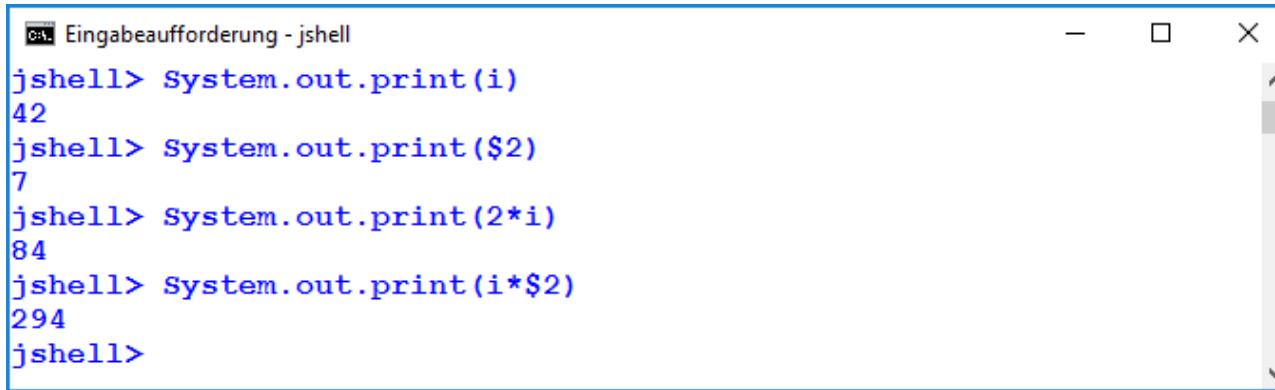
jshell> /v
|   String $1 = "Hallo Welt"
|   int $2 = 7
|   double $3 = 3.14
|   int i = 42

jshell>
```

The screenshot shows a terminal window titled 'Eingabeaufforderung - jshell'. It displays several lines of Java code being entered and executed. The first line declares an integer variable 'i' and assigns it the value 42. The second line starts a script file named 'v'. Inside the script, three variables are declared: '\$1' set to the string 'Hallo Welt', '\$2' set to the integer 7, and '\$3' set to the double 3.14. The final line in the script re-declares 'int i = 42'. The prompt 'jshell>' appears at the end of the session.

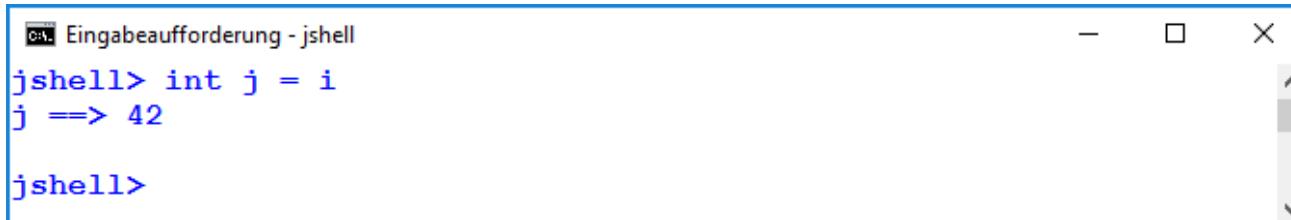
Verwendung von Variablen

- Variablen können immer dort stehen, wo auch der Wert der Variablen stehen darf.



```
Eingabeaufforderung - jshell
jshell> System.out.print(i)
42
jshell> System.out.print($2)
7
jshell> System.out.print(2*i)
84
jshell> System.out.print(i*$2)
294
jshell>
```

- Man kann einer Variablen den Wert einer anderen Variablen zuweisen.



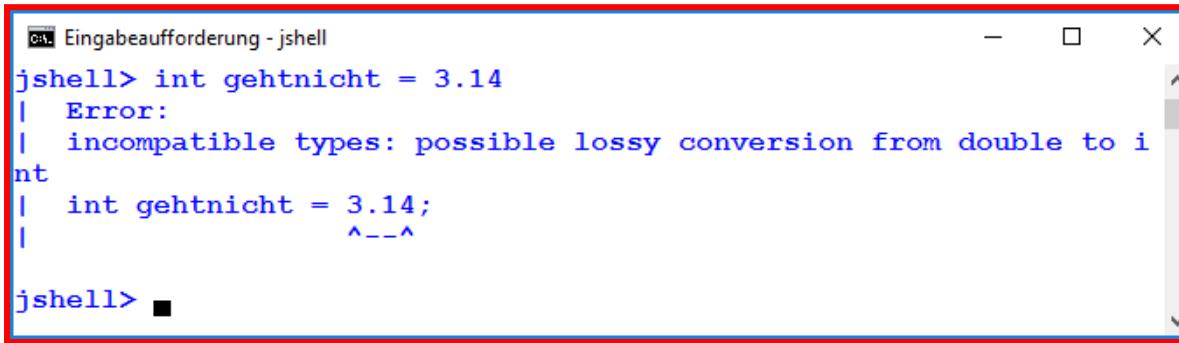
```
Eingabeaufforderung - jshell
jshell> int j = i
j ==> 42

jshell>
```

- Der Wert von i wird aus dem Speicher gelesen und dann in der Variable j gespeichert.

Dies ist jedoch zu beachten!

- Einer Variablen v kann nur ein Wert zugewiesen werden, wenn der Datentyp von v auch den Wert enthält.



```
Eingabeaufforderung - jshell
jshell> int gehtnicht = 3.14
| Error:
| incompatible types: possible lossy conversion from double to int
|   int gehtnicht = 3.14;
|           ^--^

jshell> ■
```

A screenshot of a terminal window titled "Eingabeaufforderung - jshell". The window contains the following text:
jshell> int gehtnicht = 3.14
| Error:
| incompatible types: possible lossy conversion from double to int
| int gehtnicht = 3.14;
| ^--^

The text from the second line onwards is highlighted with a red box, indicating the error message.
jshell> ■

- REPL liefert dann eine Fehlermeldung (Error) mit einer Begründung.
 - Der Datentyp int enthält nur ganze Werte. Eine Zuweisung von einer Gleitpunktzahl ist nicht möglich.



Überschreiben von Variablen

- Prinzipiell ist es möglich, bereits deklarierten Variablen einen neuen Wert zuzuweisen.
 - Hierfür verwendet man wieder den Zuweisungsoperator.
 - Der alte Wert ist dann nicht mehr verfügbar.

```
Eingabeaufforderung - jshell

jshell> /v
|   string $1 = "Hallo Welt!"
|   int $2 = 7
|   int i = 42

jshell> i = 23
i ==> 23

jshell> $1 = "Hallo Uni Marburg!"
$1 ==> "Hallo Uni Marburg!"

jshell> /v
|   String $1 = "Hallo Uni Marburg!"
|   int $2 = 7
|   int i = 23

jshell> ■
```

... und übrigens

- Verlassen von REPL mit
 - `/exit`
- Abspeichern der bisherigen Befehle mit
 - `/save meineREPLDatei`
- Einlesen der Befehle aus der Datei „meineREPLDatei“ beim Starten von REPL.
 - `jshell meineREPLDatei`

Zusammenfassung

- Erste Schritte mit Java unter Verwendung von REPL
- Wichtige Konzepte
 - Variablen
 - Datentypen
 - Wertemengen
 - Operationen
 - Zuweisungsoperator

2. Algorithmen und Methoden

- Vertiefende Diskussion über Algorithmen
 - Definition
 - Eigenschaften
 - Klassifizierung
- Algorithmen und Methoden in Java
 - Variablen
 - Einfache Anweisungen
- Kontrollstrukturen in Java
 - Anweisungsblock (Sequenz)
 - Selektion
 - Wiederholung
 - Rekursion



Al-Chwarizmi,
der Namensgeber
für *Algorithmus*

Motivation

- Algorithmen sorgen dafür, dass
 - Aufgaben effizienter erledigt werden können,
 - mathematische Probleme einfacher gelöst werden können,
 - Amazon Ihnen passende Produktvorschläge macht,
 - Sie den Weg zum Hörsaalgebäude finden.

Motivation

- Algorithmen lösen wichtige Probleme.
 - Sortieren
 - Eines der bekanntesten Probleme der Informatik.
 - Computer beschäftigen sich sehr oft mit dem Sortieren von Daten.
 - Berechnung des größten gemeinsamen Teilers
 - Euklid hat bereits dazu vor 2300 Jahren einen Algorithmus entwickelt.
 - Ranking von Web-Seiten
 - Algorithmus von Brin & Page wird in Suchmaschinen verwendet.
 - Verschlüsselung von Nachrichten
 - Verwendung von Algorithmen bereits im antiken Rom durch Caesar



Algorithmen im Alltag

- Das Möbelhaus Ikea bietet Online viele Montageanleitungen an.
- Jede Gebrauchsanleitung enthält Algorithmen.
 - Wie kann ich mit meiner Kaffeemaschine Kaffee kochen?
 - Wie kann ich die Kaffeemaschine entkalken?
- Wie kann ich einen Router konfigurieren?

Algorithmen und Programme

- Algorithmen können in verschiedener Art und Weise beschrieben werden.
 - So kann z. B. ein Algorithmus mit Hilfe von einer Programmiersprache formuliert werden.
 - Ein **Algorithmus wird als Methode** implementiert, die **Teil eines Programms** ist. Das Programm kann dann auf einem Rechner ausgeführt werden.
 - Viele Algorithmen werden jedoch informell beschrieben, um dann vom Menschen ausgeführt zu werden.
 - Beispiele
 - Algorithmus zum Aufbauen von Hemnes mit 8 Schubladen.
 - Algorithmus zum Entkalken der Kaffeemaschine
 - Algorithmus zum Kochen einer Tasse Kaffee

2.1 Algorithmen und ihre Spezifikationen

- Algorithmen sind **allgemeine Lösungsverfahren** für eine Problemklasse
 - Der Algorithmus bekommt Daten als Eingabe und liefert ein Ergebnis als Ausgabe.



- Problemspezifikation:
ist eine **vollständige, detaillierte und unzweideutige Problembeschreibung**.
 - vollständig:**
Angabe aller Rahmenbedingungen (**Eingabe**)
 - detailliert:**
Voraussetzungen über Hilfsmittel und Grundoperationen (**Prozessor**)
 - unzweideutig:**
klare Beschreibung, was der Algorithmus tun soll (**Ausgabe**).

Beispiel (Hemnes von Ikea)

- **vollständig:**

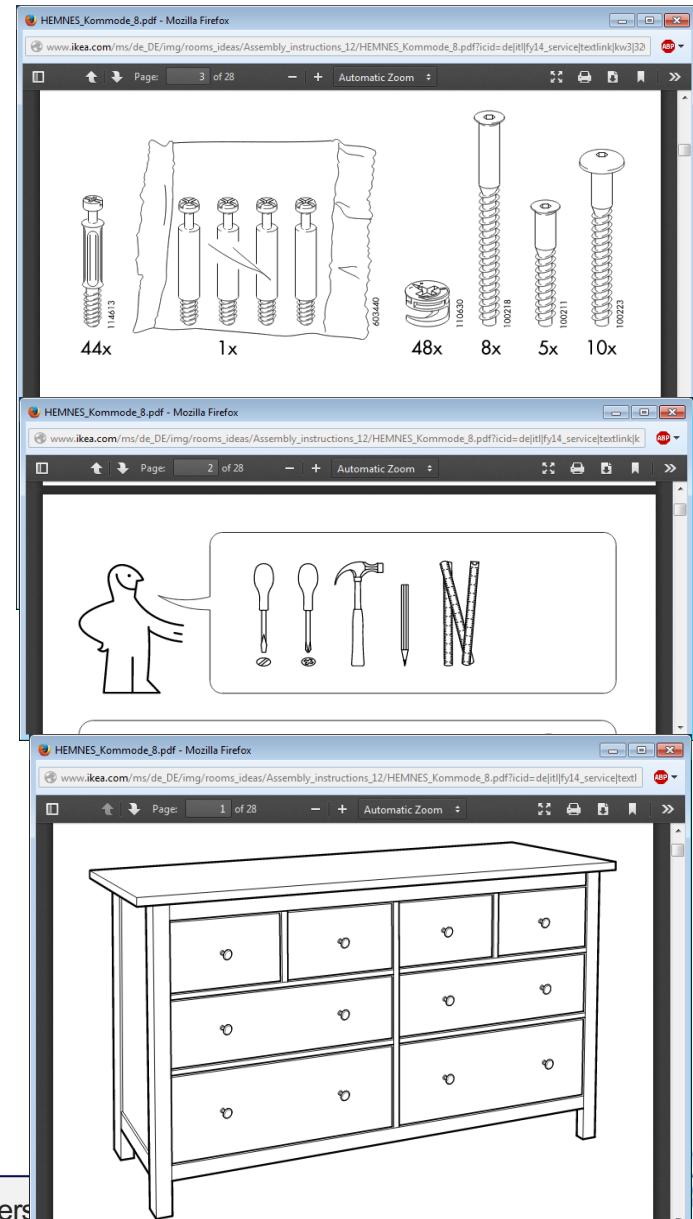
- Es wird genau beschrieben, welche Teile in der Kiste sein sollen.
- Alle Schrauben sind genau beschrieben.

- **detailliert:**

- Benötigte Werkzeuge werden gezeigt.

- **unzweideutig:**

- Eine Zeichnung zeigt das Endprodukt.



Formale Beschreibungsverfahren zur Spezifikationen von Problemen

- Informelle in Umgangssprache formulierte Spezifikationen genügen i.a. nicht den obigen Kriterien.



- Spezifikationen können in der Informatik ganz formal in speziellen, auf der Logik basierenden Spezifikationssprachen ausgedrückt werden.
 - Beispiele für formale Spezifikationssprachen: Z Notation, VDM

Beispiel: Suche nach einer Telefonnummer

- Informelle Beschreibung: Suche zu einem beliebigen Namen die zugehörige Telefonnummer.

Offene Fragen:

- Vollständigkeit:
 - Wie setzt sich ein Name zusammen?
 - Buchstaben des deutschen Alphabets
 - Sind auch Ziffern und Sonderzeichen erlaubt ? Beispiel „1&1“
 - Besteht ein Name aus Vor- und Nachname?
- Detailliertheit:
 - Welche Hilfsmittel stehen zur Verfügung?
 - Internet
 - Telefonbuch
 - CD-ROM
 - Was bieten die einzelnen Hilfsmittel an Funktionalität?
- Unzweideutigkeit:
 - Wie soll die Ausgabe meiner Suche aussehen?
 - Was passiert, wenn mehrere Telefonnummern gefunden werden?

Beispiel: ggT (größter gemeinsamer Teiler)

- Informelle Problembeschreibung: Für beliebige Zahlen M und N berechne den größten gemeinsamen Teiler

Offene Fragen:

- Vollständigkeit:
 - Welche Zahlen sind zugelassen?
 - Ganze Zahlen, rationale Zahlen, oder sonstige?
 - Dürfen M und N auch 0 sein?
- Detailliertheit:
 - Welche Operationen sind auf den ganzen Zahlen erlaubt?
 - +, - oder auch / (ganzzahlige Division), % (Divisionsrest)?
- Unzweideutigkeit:
 - Was bedeutet „ggt“ überhaupt?
 - Welche Zahlen sind als Ergebnis zugelassen?

Vorbedingung - Nachbedingung

- Probleme werden formal über ein Paar $\{P\} \{Q\}$ spezifiziert.
 - P (**Vorbedingung**)
 - Alle relevanten Eigenschaften, die anfangs vor der Ausführung des Algorithmus gelten.
 - Q (**Nachbedingung**)
 - Alle relevanten Eigenschaften, die am Ende des Algorithmus erfüllt sind.
- P und Q können als Boolesche Funktionen aufgefasst werden, die nur zwei Werte („wahr“ und „falsch“) als Resultat zulassen.

Beispiel (ggt)

- Vorbedingung P
 - M und N sind ganze Zahlen mit $M > 0$ und $N > 0$.
- Nachbedingung Q
 - Das Ergebnis z aus den ganzen Zahlen mit $z > 0$ erfüllt folgende Bedingungen:
 - z ist Teiler von M
 - z ist Teiler von N
 - für jede ganze Zahl y, die N und M teilt, gilt $y \leq z$

Algorithmus – wichtige Begriffe

- Definition
 - Ein **Algorithmus** ist eine präzise und endliche **Beschreibung** eines allgemeinen Verfahrens zur **schrittweisen Lösung** eines **Problems**. Der Algorithmus besteht aus einzelnen **Schritten** und **Vorschriften**, welche die **Ausführung der Schritte** kontrollieren.
 - Jeder Schritt muss
 - klar und eindeutig beschrieben sein und
 - mit endlichem Aufwand in endlicher Zeit ausführbar sein.

- Ein Algorithmus wird durch einen **Prozessor** ausgeführt.
- Ein (sequentieller) **Prozess** bezeichnet die sequentielle Folge von Ausführungsschritten, die durch einen Prozessor bei der Verarbeitung eines Algorithmus erzeugt wird.



Gegenbeispiele: Was ist kein Algorithmus!

- Beschreibung durch **Analogien**

Das Zerteilen von Fliesen läuft im Prinzip so wie das Zerteilen eines Ziegels.

- Beschreibung durch **Beispiele**

Berechne die ersten 10 Zahlen der Folge 2, 3, 5, 7, 11,

- Was ist die nächste Zahl der Folge?

- Verwendung **nicht-ausführbarer und nicht-Grundoperationen**

- Laufe die nächsten 100 Meter in 5 Sekunden.
 - Zur besseren Anbindung des Campus an die Stadt, soll ab dem WS 2018/19 ein Helikopter verwendet werden.

- **Verletzung der Allgemeinheit** (Problem: Addieren zweier Zahlen)

- 2 und 3 ergibt zusammen 5
 - Ein **Algorithmus** lässt sich **stets auf eine Klasse von Problemstellungen** anwenden (z.B. die Addition zweier Zahlen).
 - Jeder Algorithmus besitzt deshalb **Eingabeparameter**, die beim Start eines Prozesses **mit konkreten Werten zu belegen** sind.

Korrektheit von Algorithmen

- Zwingend notwendig ist die **Korrektheit**, d.h. der Algorithmus muss tatsächlich die **Spezifikation $\{P\}\{Q\}$ erfüllen**:
 - Für **jeden möglichen Prozess** eines Algorithmus muss folgender Sachverhalt gelten:
 - Falls beim Start des Prozesses die **Funktion P** wahr liefert,
 - muss auch am Ende des Prozesses die **Funktion Q** wahr liefern
 - Erfüllt ein Algorithmus A die Spezifikation $\{P\}\{Q\}$, so schreiben wir auch
$$\{P\} A \{Q\}$$
 - Die **Vorbedingung P** ist abhängig von den **Eingabe** und die **Nachbedingung Q** ist abhängig von der **Ausgabe** des Algorithmus A.



„Jetzt hat man jedoch beim Nachfolgemodell Ariane 5 nicht beachtet, dass die Ariane 5 eine schnellere Rakete ist. In dem Moment sind die Daten, die übermittelt worden sind, höhere Zahlen gewesen. Und diese Zahlen haben in den Bereich, den der Computer auffassen konnte, nicht mehr reingepasst.“

Nicht-funktionale Eigenschaften von Algorithmen

- Algorithmen sollten einfach zu **verstehen** sein.
 - Erläuterung des Prinzips durch ein Beispiel
 - Zusätzliche Beschreibung in Form von **Dokumentation**
- Laufzeit- und **Platzbedarf** sollten möglichst niedrig sein.
 - Effizienzanalyse von Algorithmen
 - Experimentelle Validierung der Kosten

Terminierung von Algorithmen

Definition (**Terminierung**)

- Ein **Prozess** terminiert, wenn er aus einer **endlichen Anzahl** von **Ausführungsschritten** besteht.
- Ein **Algorithmus** terminiert, wenn **jeder** seiner möglichen Prozesse terminiert.
- Bemerkungen
 - Oft ist „Algorithmus“ in der Literatur so definiert, dass die **Terminierung** bereits **inhärent gefordert** wird (z.B. Suche nach Webseiten).
 - **nicht-terminierende** Algorithmen sind aber **auch von Interesse**
 - Ampelsteuerung
 - Verarbeitung von Daten eines Sensors oder Tickers:
Kontinuierliche Berechnung des durchschnittlichen Börsenwertes einer Aktie innerhalb der letzten sieben Tage.

Deterministische Algorithmen

- **Determinismus** bezieht sich auf den **Prozess**
 - Ein Algorithmus ist **deterministisch**, wenn es zu jeder möglichen Eingabe genau einen Prozess gibt.
 - Kann bei nichtdeterministischen Algorithmen den zu einer Eingabe gehörenden **Prozessen** eine Wahrscheinlichkeit **zugeordnet** werden, so spricht man auch von **stochastischen** Algorithmen.
- **Beispiele** für nichtdeterministische Algorithmen
 - Spielzüge bei einem Schachspiel
 - Verkehrsreglung an einer Kreuzung

Determiniertheit von Algorithmen

- Determiniertheit bezieht sich auf das Ergebnis eines Algorithmus
 - Ein Algorithmus ist **determiniert**, wenn die Eingabe das Resultat des Algorithmus eindeutig bestimmt.
- Beispiele für determinierte Algorithmen
 - Aufbauanleitung bei Ikea
 - Größter gemeinsamer Teiler von zwei Zahlen
- Beispiele für nichtdeterminierte Algorithmen
 - Spielzüge bei Spielen mit Würfeln
 - Approximative Suchverfahren für Probleme, zu denen es keinen schnellen Algorithmus gibt.

Sequentiell: Schritt für Schritt, nacheinander. Parallel: Mehrere Schritte gleichzeitig, um Zeit zu sparen und effizienter zu arbeiten.

Sequentielle und parallele Algorithmen

sequentiell: ein schritt kann erst ausgeführt werden wenn der vorherige schon abgeschlossen ist d.h. es gibt eine klare reihenfolge

- Diese Begriffe beziehen sich auf die Prozesse.
 - Ein Prozess heißt **parallel**, wenn Einzelschritte gleichzeitig ausgeführt werden.
 - Ein Algorithmus heißt **parallel**, wenn er parallele Prozesse zulässt. Andernfalls heißt er **sequentiell**.
- Beispiele:
 - Beim **Hausbau** kann eine Wand aus Ziegeln gemauert werden und gleichzeitig können die Ziegel zugeschnitten werden.
 - Bei der **Berechnung von $a^2 + b^2$** kann die Quadratberechnung der Zahlen a und b parallel zueinander ablaufen.
 - **Suchen in Telefonbüchern** kann parallel erfolgen (wenn es mehrere Telefonbücher gibt, ansonsten ist Parallelität nicht sehr effektiv)

Zusammenfassung

- Formale Spezifikation von Problemen
 - Vorbedingung und Nachbedingung
- Algorithmus
- Korrektheit von Algorithmen
- Nicht-funktionale Eigenschaften
 - Laufzeit und Speicherverbrauch eines Algorithmus
- Arten von Algorithmen
 - Deterministisch
 - Determiniert
 - Sequentielle und parallele Algorithmen

Live Vote

Beispiel

PIN: M4I2

Deterministische Algorithmen:

Ein Algorithmus ist deterministisch, wenn er bei gleicher Eingabe immer das gleiche Ergebnis liefert.
Alles, was der Algorithmus macht, ist vorhersehbar und festgelegt.

Determiniertheit von Algorithmen:

Determiniertheit bedeutet, dass ein Algorithmus keine Zufälle oder Unsicherheiten enthält.
Das Verhalten ist immer gleich, wenn die Eingaben gleich sind

Deterministischer Algorithmus:

Rezept: Du folgst genau dem Rezept: 200g Mehl, 100g Zucker, 2 Eier, 30 Minuten bei 180°C backen.
Ergebnis: Jedes Mal, wenn du das Rezept befolgst, bekommst du denselben Kuchen.

Das Rezept ist deterministisch, weil es immer zum gleichen Kuchen führt, wenn du es genauso machst.

<https://ilias.uni-marburg.de/vote/M4I2>



Hier Text eingeben

2.2 Methoden in Java

- Algorithmen lassen sich in Java direkt als **Methoden** implementieren.
- Beispiel: Algorithmus zur Berechnung der Kreisfläche.

Dokumentation

```
/** Die Methode kreisFlaeche liefert zum Radius r die Kreisfläche.  
 * @param r ist der Radius  
 * @return liefert die Fläche eines Kreises mit Radius r  
 */
```

Methodenkopf

```
double kreisFlaeche (double r)
```

Methodenrumpf

```
{  
    double pi = 3.14;  
    return pi*r*r;  
}
```

- Am Ende des Kapitels sollten Sie dieses Beispiel verstehen!

2.2 Methoden in Java

- Algorithmen lassen sich in Java direkt als **Methoden** implementieren.
- Beispiel: Algorithmus zur Berechnung der Kreisfläche.

Dokumentation

```
/** Die Methode kreisFlaeche liefert zum Radius r die Kreisfläche.
```

```
* @param r ist der Radius  
* @return liefert die Fläche
```

In einem
Methodenrumpf muss
immer ein Semikolon (;)
am Ende jeder
Anweisung stehen.

Methodenkopf

```
double kreisFlaeche (do
```

```
{
```

```
    double pi = 3.14;  
    return pi*r*r;
```

```
}
```

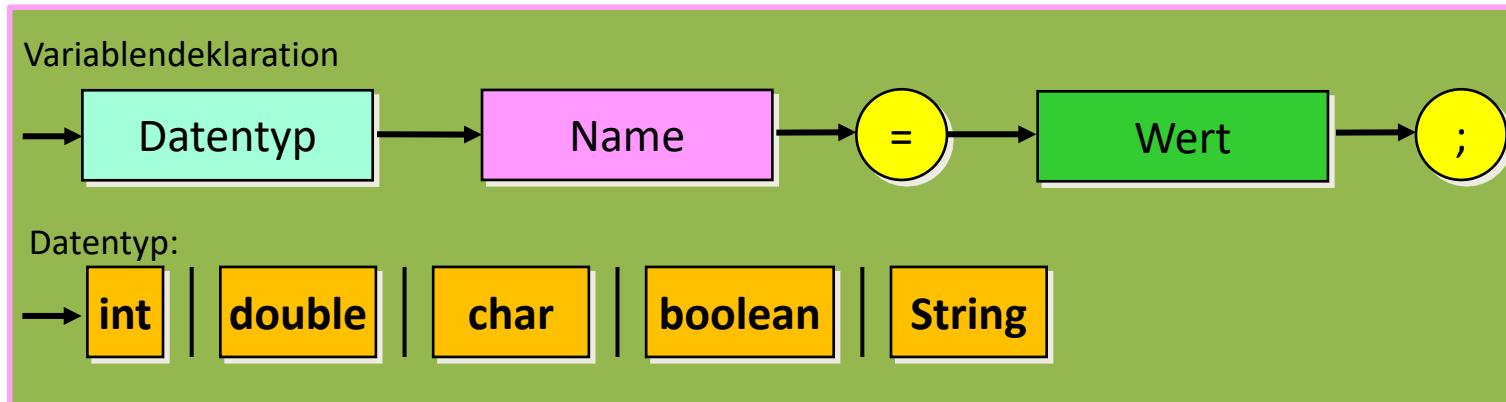
- Am Ende des Kapitels sollten sie dieses Beispiel verstehen!

Syntax einer Sprache

- Unter der **Syntax einer formalen Sprache** versteht man die Regeln, um aus elementaren Bausteinen korrekte Programme zu erstellen.
 - Elementare Bausteine einer Programmiersprache sind
 - Schlüsselwörter
 - Operatoren
 - Werte
 - Bezeichner eines Benutzers
 - Beispiel (Deklaration einer Variable)
 - `int i = 42;`
- Darstellung der Regeln durch Syntaxdiagramme

Syntaxdiagramme

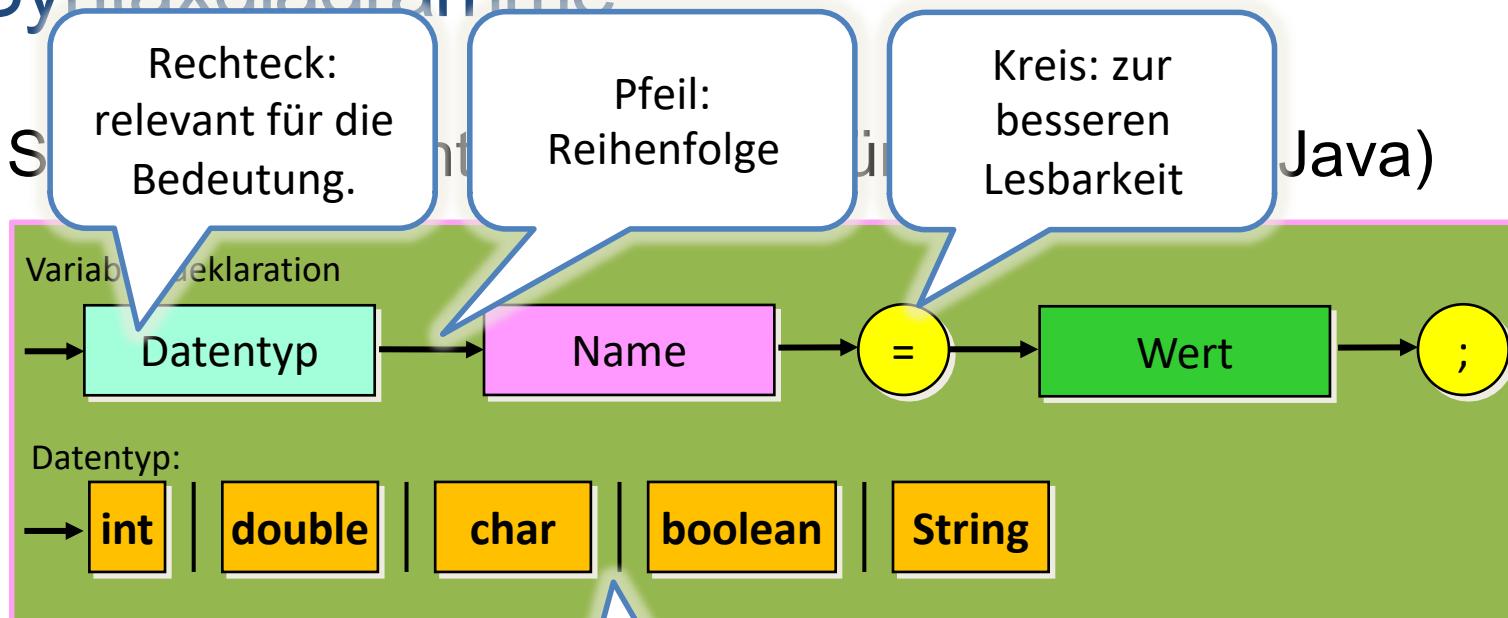
- Sehr vereinfachte Beispiele (für die Sprache Java)



- Die erste Regel besagt, dass eine Variablen-deklaration aus einem Datentyp, einen Namen, dem Zuweisungsoperator und einem Wert besteht.
- Die zweite Regel besagt, dass ein Datentyp entweder int, double, char, boolean oder String ist.

Syntaxdiagramme

- Syntaxdiagramme

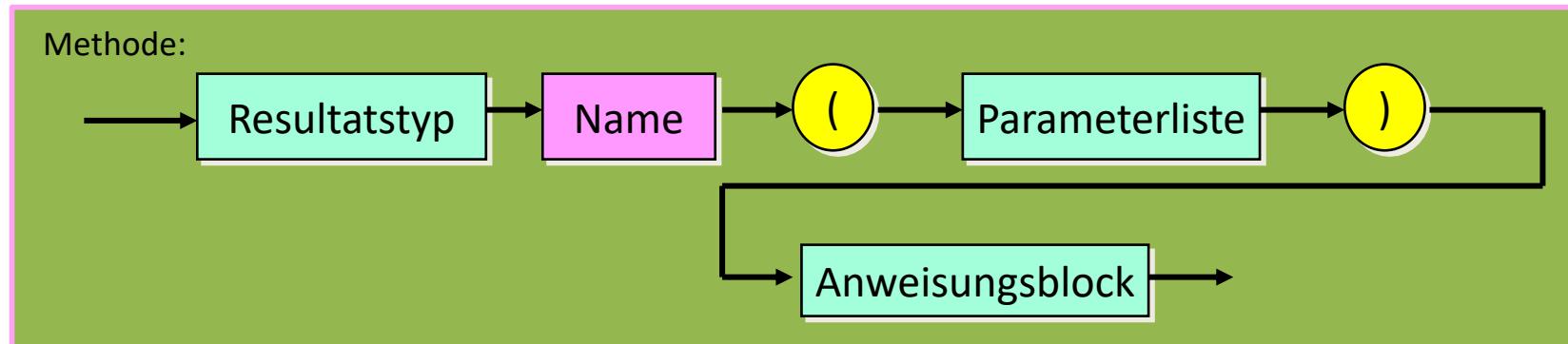


- Die erste Regel besagt, dass eine Variablen-deklaration aus einem Datentyp, einem Namen, einem Zuweisungsoperator und einem Wert besteht.
- Die zweite Regel besagt, dass der Datentyp entweder int, double, char, boolean oder String ist.

Senkrechter Strich trennt Alternativen

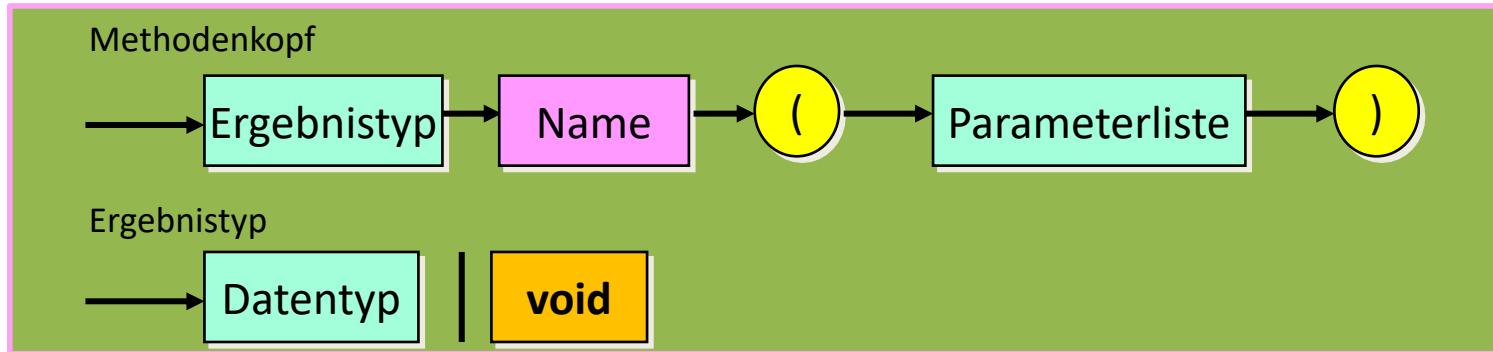
Syntax einer Methode (für die jshell)

- Eine Methode kann durch folgende Regel beschrieben werden.



- Der **Methodenrumpf** entspricht einem **Anweisungsblock** (auch **Sequenz** genannt).

2.2.1 Methodenkopf: Ergebnistyp



- **Ergebnistyp**

`double kreisFlaeche (double r)`

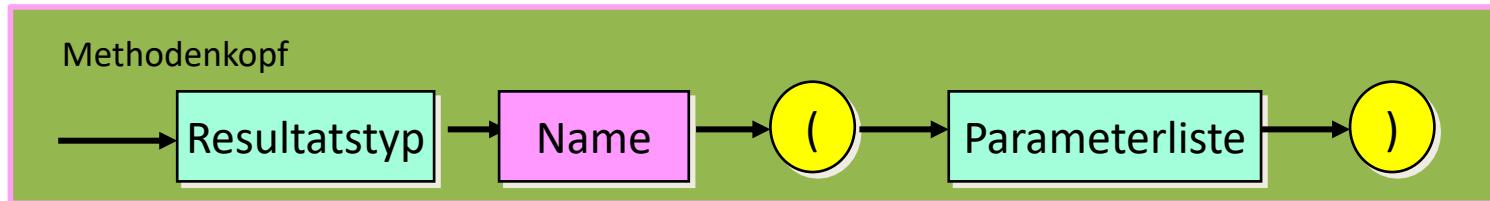
- *Datentyp der Ausgabe*

- In unserem Beispiel `kreisFlaeche` ist es der **Typ double**.
 - Die Methode muss ein Ergebnis aus der Wertemenge der Gleitkommazahlen liefern.

- *Schlüsselwort void* wird verwendet, wenn die Methode kein Ergebnis produziert.

- Die Methode `System.out.println` liefert kein Ergebnis, sondern eine Ausgabe auf dem Bildschirm.

Methodenkopf: Name der Methode

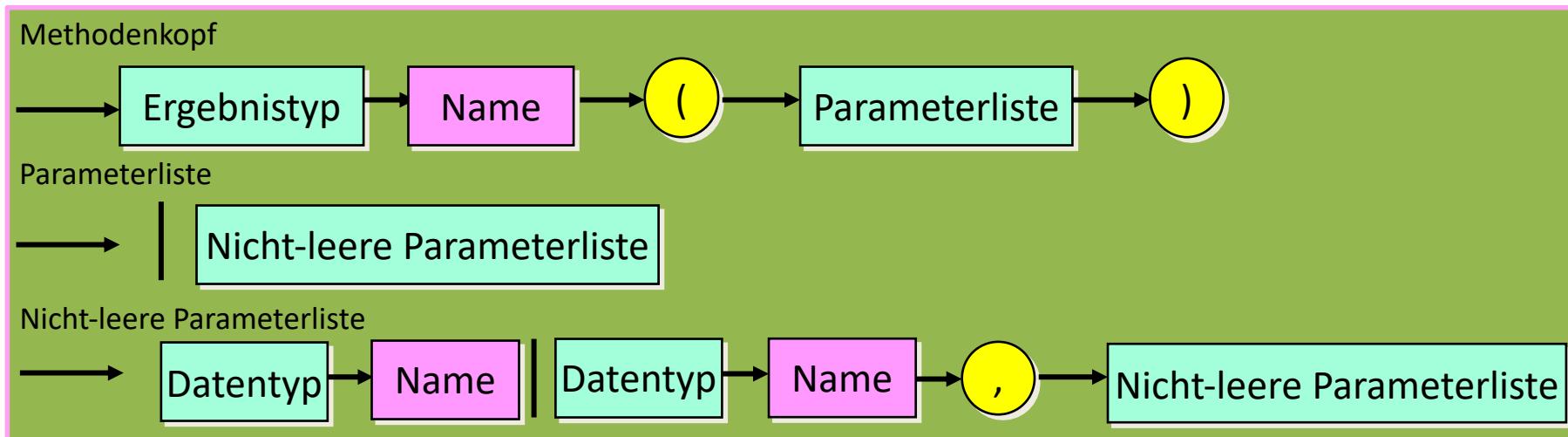


- **Name der Methode**

- Dieser wird vom Programmierer gewählt werden und muss eindeutig sein.
 - In unserem Beispiel `kreisFlaeche`.
 - Bei der Namenswahl müssen noch gewisse Regeln beachtet werden.
(→ Details dazu später)

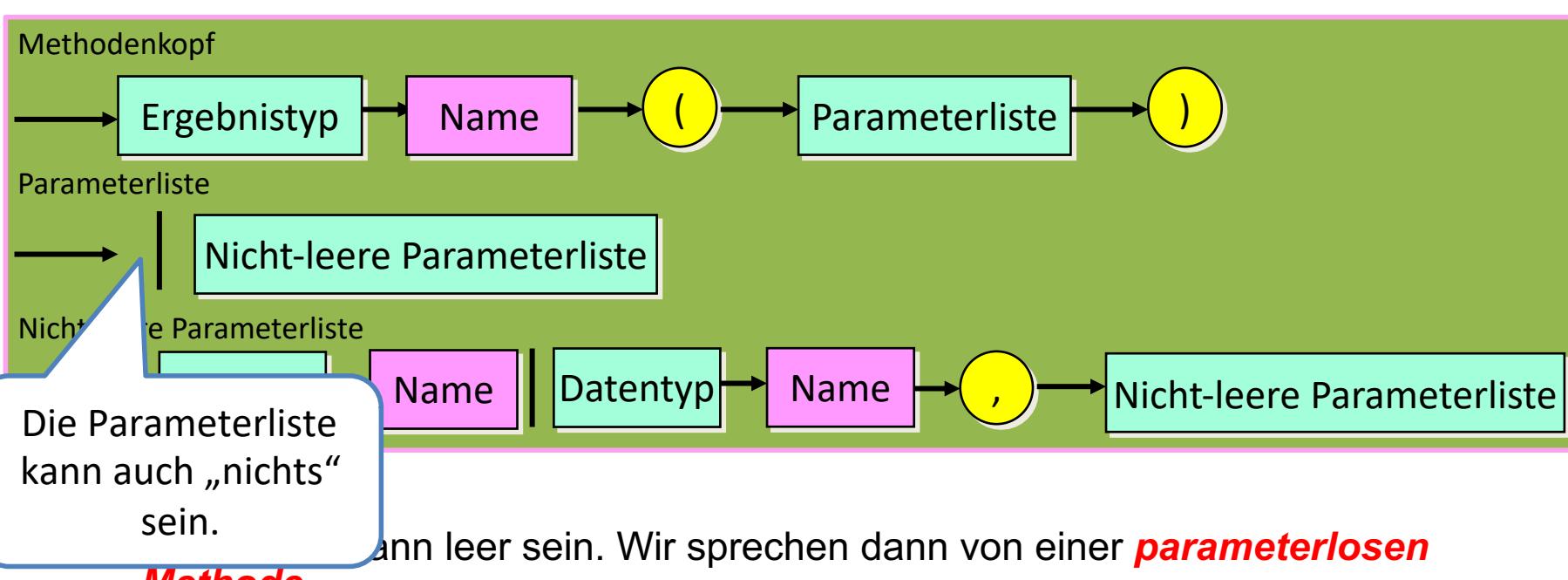
`double kreisFlaeche (double r)`

Methodenkopf: Parameterliste



- **Parameterliste**
 - Diese Liste kann leer sein. Wir sprechen dann von einer **parameterlosen Methode**.
- **Nicht-leere Parameterliste**
 - Diese bestehen entweder aus einem Parameter, der aus einem Datentyp und einem Namen besteht.
 - Oder mehreren Parametern, die mit Komma getrennt sind.
 - Dies wird durch eine rekursive Regel definiert. (→ Details Rekursion später)

Methodenkopf: Parameterliste



- Die Liste kann nicht leer sein.
- Nicht-leere Parameterliste
 - Diese bestehen entweder aus einem Parameter, der aus einem Datentyp und einem Namen besteht.
 - Oder mehreren Parametern, die mit Komma getrennt sind.
 - Dies wird durch eine rekursive Regel definiert. (→ Details Rekursion später)

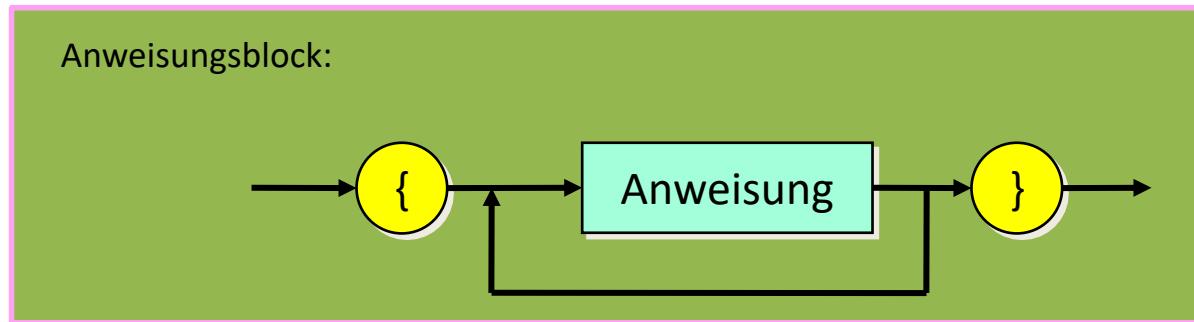
Definition und Aufruf einer Methode in jshell

- Eine Methode wird durch **Angabe des Methodennamens** und eines **Werts oder eines Ausdrucks für jeden Parameter** aufgerufen.
 - Der Methodenrumpf hat keinen Einfluss auf die Syntax des Aufrufs.

```
jshell> double kreisFlaeche(double r) {  
...>     double pi = 3.14;  
...>     return r*r*pi;  
...> }  
|  created method kreisFlaeche(double)  
  
jshell> double a = Math.sin(2.0)  
a ==> 0.9092974268256817  
  
jshell> double b = kreisFlaeche(2.0)  
b ==> 8.56  
  
jshell> double c = kreisFlaeche(b)  
c ==> 156.805504  
  
jshell> double c = kreisFlaeche(a * b)  
c ==> 129.65021070295182
```

2.2.2 Anweisungsblock in Java-Methoden

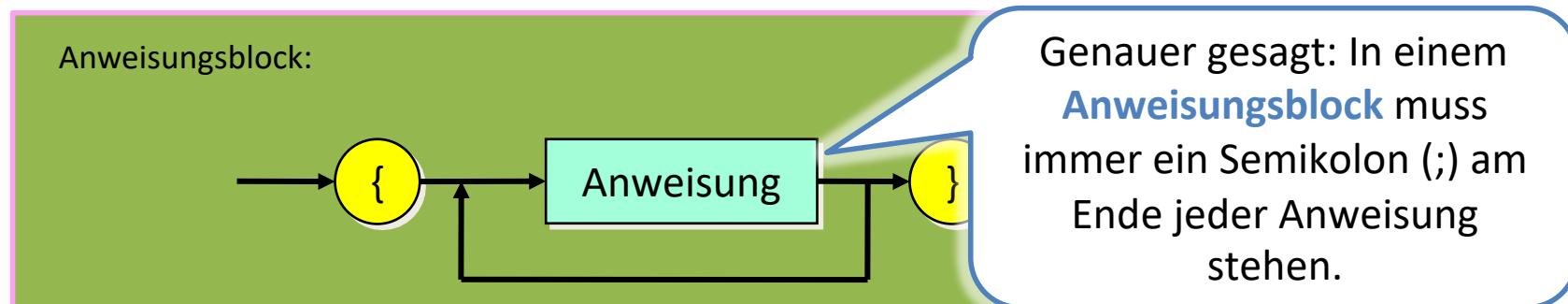
- In Java wird ein Verarbeitungsschritt als **Anweisung** bezeichnet.
- Eine Sequenz von Anweisungen, die durch ein Paar von geschweiften Klammern umgeben ist, wird als **Anweisungsblock** bezeichnet.



- Ein Anweisungsblock kann **logisch wieder als eine einzige Anweisung** aufgefasst werden!
 - Wir benutzen deshalb auch dann den Begriff Anweisung.
- Zur Unterscheidung werden Anweisungen, die nicht weiter strukturiert sind, auch als **atomar** bezeichnet.

2.2.2 Anweisungsblock in Java-Methoden

- In Java wird ein Verarbeitungsschritt als **Anweisung** bezeichnet.
- Eine Sequenz von Anweisungen, die durch ein Paar von geschweiften Klammern umgeben ist, wird als **Anweisungsblock** bezeichnet.



- Ein Anweisungsblock kann **logisch wieder als eine einzige Anweisung** aufgefasst werden!
 - Wir benutzen deshalb auch dann den Begriff Anweisung.
- Zur Unterscheidung werden Anweisungen, die nicht weiter strukturiert sind, auch als **atomar** bezeichnet.

Beispiel: Methoden kreisFlaeche

```
double kreisFlaeche (double r)
```

```
{
```

```
    double pi = 3.14;
```

```
    return pi*r*r;
```

```
}
```

————— (atomare) Anweisung

————— (atomare) Anweisung

Anweisungsblock

- *Der Methodenkörper besteht aus einem Anweisungsblock, der aus zwei atomaren Anweisungen besteht.*

Methoden mit verschachtelten Anweisungsblöcken

```
double kreisFlaechePlusBlock(double r)
{
    double pi = 3.14;
    {
        double tmp = 0.0;
        String str = " ";
    }
    return pi*r*r;
}
```

The diagram illustrates the structure of the code. A blue rectangular border surrounds the entire method body. Inside, a green rectangular box encloses the inner block starting with '{' and ending with '}'. Red arrows point from the text '(atomare) Anweisung' to the assignment 'double pi = 3.14;' and the return statement 'return pi*r*r;'. A green arrow points from the text 'Anweisungsblock' to the inner block's braces '{}'. A blue arrow points from the text 'Anweisungsblock' to the outer blue border.

Anweisungsblock

- *Der Methodenrumpf besteht aus einem Anweisungsblock, der aus zwei atomaren Anweisungen und einem Anweisungsblock besteht.*
 - Es ist eine beliebige Verschachtelung von Anweisungsblöcken möglich.

Abarbeitung eines Anweisungsblocks

- Ein **Prozessor** führt einen Anweisungsblock nach folgenden Regeln aus:
 - Die Bearbeitung des Anweisungsblocks **beginnt mit der ersten Anweisung**.
 - Zu einem **Zeitpunkt** wird **genau eine Anweisung** ausgeführt.
 - **Jede Anweisung wird genau einmal ausgeführt**.
 - Keine Wiederholung, kein Auslassen einer Anweisung. (Mehr dazu später)
 - **Die Reihenfolge** der Bearbeitung der Anweisungen ist **identisch mit** der im **Algorithmus**.
 - Die Bearbeitung des Anweisungsblocks **endet mit der letzten Anweisung**.

Kurze Zusammenfassung von der Bearbeitung :

- 1) Die Bearbeitung beginnt mit der 1. Anweisung
- 2) zu einem zeitpunkt wird eine Anweisung bearbeitet
- 3) Jede Anweisung wird einmal ausgeführt es werde keine Anweisungen wiederholt oder weggelassen
- 4) Die Reihenfolge der Bearbeitung entspricht dem Algorithmus
- 5) Bearbeitung endet mit der letzten Anweisung

Beispiel

```
double kreisFlaechePlusBlock(double r)
```

```
{  
    double pi = 3.14;  
    {  
        double tmp = 0.0;  
        String str = " ";  
    }  
    return pi*r*r;  
}
```

(atomare) Anweisung

Anweisungsblock

(atomare) Anweisung

Anweisungsblock

- Der blaue Anweisungsblock besteht aus drei Anweisungen
 - Die zweite Anweisung ist ein Anweisungsblock.
- Die Abarbeitung verläuft sequentiell.
 1. final double pi = 3.14;
 2. Komplette Abarbeitung des grünen Anweisungsblock
 3. return pi*r*r;

Anweisungen

- In unserem Beispiel gibt es zwei Arten von Anweisungen.
 - Eine Variablen-deklaration mit einer Initialisierung.

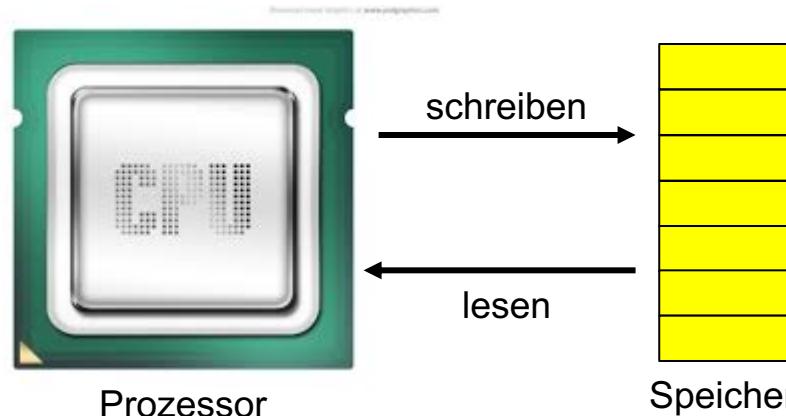
```
double pi = 3.14;
```
 - Eine return-Anweisung.

```
return pi*r*r;
```
- Im Folgenden werden wir nochmals genauer das Thema Variablen behandeln.

2.2.3 Variablen

Variablen bieten die Möglichkeit Werte zu speichern , geben uns die einzige Möglichkeit um auf den Speicher zuzugreifen
Man kann auf eine Variablen ("Speicherzelle") lesend und schreiben zugreifen

- Beim Ausführen einer Methode durch den Prozessor müssen häufig Werte aus Berechnungen in einer Speicherzelle des Computers zwischengespeichert werden.



- **Variablen** bieten in höheren Programmiersprachen, wie z. B. Java, die einzige Möglichkeit, um auf den Speicher des Computers zuzugreifen.
 - Man kann mit einer Variablen den Inhalt einer Speicherzelle **lesen**.
 - Man kann mit einer Variablen den Inhalt einer Speicherzelle **schreiben**.

Variablen

- Unter einer **Variablen** verstehen wir einen **Speicherbereich** und einen **Datentyp** (wie z. B. `int`, `double`, `String`), auf den über einen **Variablennamen (Bezeichner)** zugegriffen werden kann.

- Variablen müssen vor der ersten Benutzung in Java **deklariert** werden.

```
double y;           // Variable y vom Typ double wird deklariert.  
int x = 23;        // Variable x vom Typ int wird deklariert und initialisiert.
```

Bezeichner	Wert	Typ
x	23	int
y	uninitialisiert	double

Bemerkungen

- Der **Bezeichner** und der **Typ** einer Variablen ändern sich während der Lebenszeit **nicht!**
- Der **Wert** einer Variablen **kann sich ändern**
 - Schreibe den Wert 42 in die Variable x ergibt dann

Bezeichner	Wert	Typ
x	42	int
y	uninitialisiert	double

Variablen

- Unter einer **Variablen** verstehen wir einen **Speicherbereich** und einen **Datentyp** (wie z. B. `int`, `double`, `String`), auf den über einen **Variablennamen** (**Bezeichner**) zugegriffen werden kann.

- Variablen müssen vor der ersten Benutzung in Java **deklariert** werden.

```
double y           // Variable y vom Typ double wird deklariert.  
int x = 23        // Variable x vom Typ wird deklariert und initialisiert.
```

Bezeichner	
x	23
y	uninitialisiert

Variable darf erst nach
Initialisierung gelesen werden!
(Ausnahme: in JShell außerhalb von
Methoden werden deklarierte
Variablen automatisch mit einem
Standardwert initialisiert)

Bemerkungen

- Der **Bezeichner und der Typ** einer Variablen sind festgelegt
- Der **Wert** einer Variablen kann später verändert werden

- Schreibe den Wert 42 in die Variable x ergibt dann

Bezeichner	Wert	Typ
x	42	int
y	uninitialisiert	double

benszeit **nicht!**

Variablen in Java-Methoden

■ Zunächst unterscheiden wir zwei Arten von Variablen

– Parametervariablen

- In der Literatur wird auch der Begriff *formale Parameter* benutzt.
- Diese Variablen werden im Methodenkopf deklariert.

– Lokale Variablen

- Deklaration erfolgt in einem Anweisungsblock einer Methode.

```
double kreisFlaeche (double r) {  
    double pi = 3.14;  
    return pi*r*r;  
}
```

Parametervariable

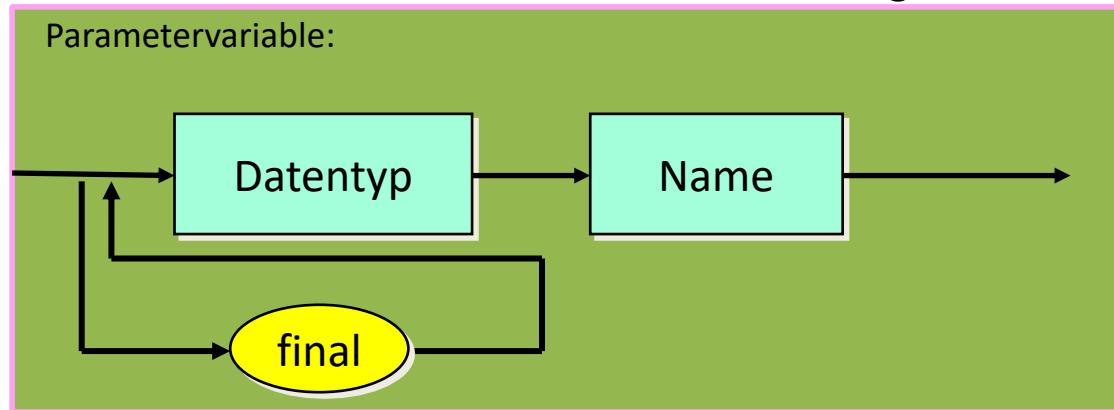
Lokale Variable

Wichtige Eigenschaften von Variablen

- **Gültigkeit** von Variablen
 - Teil des Programmtexs, in dem man die Variable verwenden **darf**.
- **Lebensdauer** von Variablen
 - Zeitspanne von der Erzeugung der Variablen im Speicher bis zum Löschen aus dem Speicher

Parametervariablen (mit final)

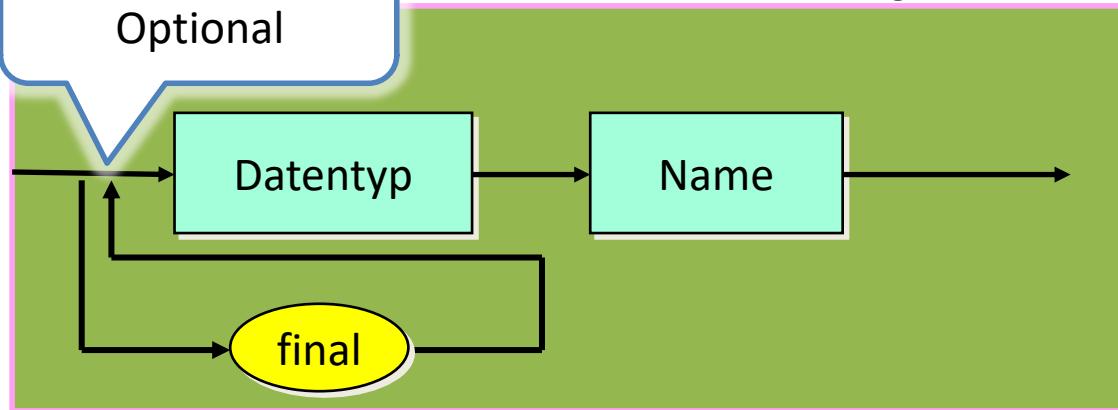
- Die Deklaration dieser Variablen erfolgt im Methodenkopf.



- Zusätzlich können wir das Schlüsselwort **final** voranstellen.
 - Alle Variablen, die als **final** deklariert werden, erhalten **genau einmal einen Wert**.
- Gültigkeit
 - Parametervariablen sind nur im Methodenrumpf verwendbar.
- Lebensdauer
 - Die Variable wird beim Methodenaufruf erzeugt und bekommt beim Aufruf einen Wert.
 - Nach dem Ende der Methode wird die Variable vom Speicher entfernt.

Parametervariablen (mit final)

- Verzweigung: dieser Variablen erfolgt im Methodenkopf.
Optional



- Zusätzlich können wir das Schlüsselwort **final** voranstellen.
 - Alle Variablen, die als **final** deklariert werden, erhalten **genau einmal einen Wert**.
- Gültigkeit
 - Parametervariablen sind nur im Methodenrumpf verwendbar.
- Lebensdauer
 - Die Variable wird beim Methodenaufruf erzeugt und bekommt beim Aufruf einen Wert.
 - Nach dem Ende der Methode wird die Variable vom Speicher entfernt.

Parametervariablen beim Methodenaufruf

- Eine Methode wird durch **Angabe des Methodennamens** und eines **Werts oder eines Ausdrucks für jede Parametervariable** aufgerufen.
 - kreisFlaeche(2.0)
 - kreisFlaeche(input)
- Beim Aufruf der Methode werden folgende Schritte ausgeführt.
 - **Erzeugung der Parametervariablen** im Speicher.
 - **Initialisierung der Parametervariablen** durch die Werte, die beim Aufruf übergeben werden.
 - **Schrittweise Ausführung** der Methode
 - Am Ende der Methode
 - Löschen aller erzeugter Variablen
 - Rücksprung an die Stelle im Programm, von der die Methode aufgerufen wurde.



Beispiel

```
double kreisFlaeche (double r) {  
    final double pi = 3.14;  
    return pi*r*r;  
}
```

```
double input = 2.5;  
double result = kreisFlaeche(input);  
System.out.println("Flaecheninhalt von Kreis mit Radius " + result);
```

Aufruf der Methode kreisFlaeche

Ausführen der Methode kreisFlaeche

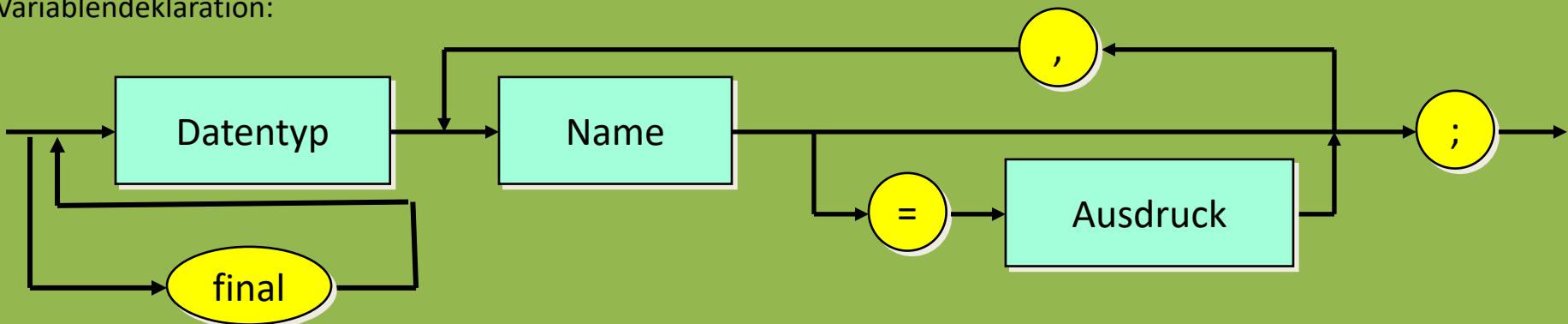
- Wert der Variable input wird gelesen und an Variable r übergeben.
- Am Ende wird das Ergebnis an den Aufrufenden geliefert.
- Parametervariablen und lokale Variablen werden gelöscht.

Es erfolgt ein Rücksprung an die Stelle des Aufrufs.

Lokale Variablen

- Deklaration erfolgt in einem der Anweisungsblöcke der Methode.

Variablen-deklaration:



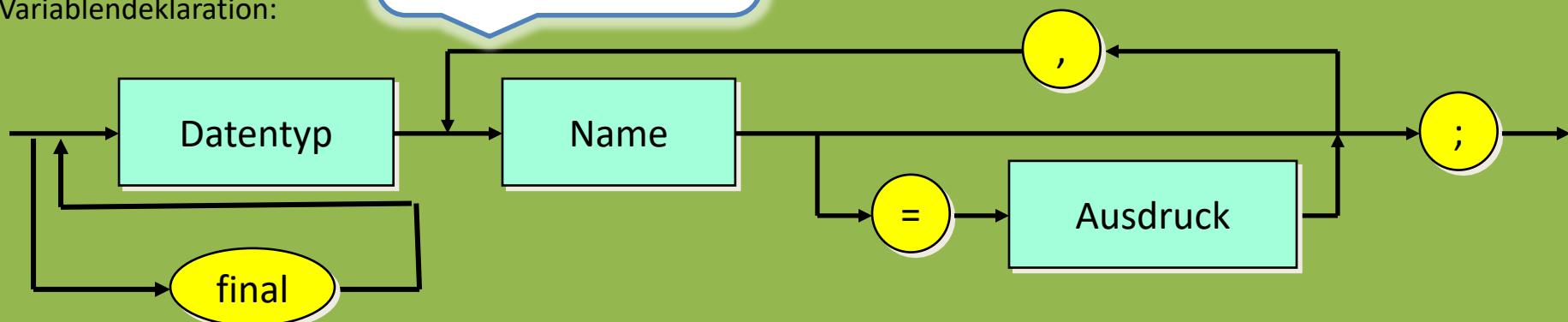
- Unterschiede zu Parametervariablen
 - Es können durch Komma getrennt **mehrere Variablen** in einer einzigen Deklaration definiert werden.
 - Alle Variablen haben den gleichen Datentyp.
 - Optional kann jede Variable einen initialen Wert bei der Deklaration erhalten.

Lokale Variablen

- Deklaration einer Variablen ist eine Kette von Anweisungsblöcke der Methode.

Rückverweis: Pfad kann wiederholt werden

Variablen-deklaration:



- Unterschiede zu Parametervariablen
 - Es können durch Komma getrennt **mehrere Variablen** in einer einzigen Deklaration definiert werden.
 - Alle Variablen haben den gleichen Datentyp.
 - Optional kann jede Variable einen initialen Wert bei der Deklaration erhalten.

Gültigkeit und Lebensdauer

- Gültigkeitsbereich der lokalen Variable
 - erstreckt sich von der Deklaration bis zum Ende des umschließenden Blocks.
 - Innerhalb des Gültigkeitsbereich einer Variablen ist es nicht möglich eine Variable mit gleichem Namen zu deklarieren.
- Lebensdauer der lokalen Variable
 - Die Variable wird angelegt, wenn die Deklaration ausgeführt wird.
 - Die Variable wird am Ende der Ausführung des umschließenden Blocks gelöscht.
- Beispiel

```
double kreisFlaeche (double r) {  
    final double pi = 3.14;  
    return pi*r*r;  
}
```

Gültigkeitsbereich der Parametervariable r

Gültigkeitsbereich der lokalen Variable pi

Beispiel (Gültigkeitsbereich)

```
int test1(int a) {  
    int y;  
    y = 2;  
    int z;  
    ...  
}  
  
int test2() {  
    int b;  
    ...  
    {  
        int c;  
        ...  
    }  
}
```

The diagram illustrates the scope of variables in two functions. In `test1`, the variable `a` is shown with its scope bracketed by curly braces. Inside this scope, `y` and `z` are declared. In `test2`, the variable `b` is shown with its scope bracketed by curly braces. Inside this scope, `c` is declared. Vertical double-headed arrows indicate the boundaries of each variable's scope. The text "Gültigkeitsbereich von a" is positioned above the first brace, "Gültigkeitsbereich von y" is between the first and second braces, and "Gültigkeitsbereich von z" is below the second brace. Similarly, "Gültigkeitsbereich von b" is above the brace for `b`, and "Gültigkeitsbereich von c" is between the braces for `b` and `c`.

Live Vote

```
double flaecheZylinder(double h, double r) {  
    double kreisFlaechen;  
    {  
        double eineKreisflaeche = 3.14 * r * r;  
        kreisFlaechen = 2 * eineKreisflaeche;  
    }  
    double mantelFlaechen = 2 * 3.14 * h; hier fehlt glaube ich *r  
    return kreisFlaechen + mantelFlaechen;  
}
```

- Wie viele Variablen sind an der markierten Stelle gültig?

Kreisflaechen Formel:
$$(2\pi r^2 + 2\pi r \cdot h)$$

$$= 2\pi r(r+h)$$

Der Code muss
double flaecheZylinder (double h, double r){
 double pi=3.14;
 double kreisFlaechen=2*pi*r*r;
 double mantelFlaechen=2*pi*r*h;
 return kreisFlaechen+mantelFlaechen;}

PIN: EQYH

<https://ilias.uni-marburg.de/vote/EQYH>



```
double flaechenZylinder2(double h, double r){  
    double pi=3.14;  
    return 2*pi*r*(r+h);}
```

Initialisierung lokaler Variablen

Zur Erinnerung

- Bei der Deklaration einer Variable wird ein Bezeichner eingeführt.
 - Vor dem Bezeichner steht der Datentyp.

```
int x = 42;  
double y;
```

Programmschnipsel

Bezeichner	Wert	Typ
x	42	int
y	uninitialisiert	double

Speicher

- Im Unterschied zu Parametervariablen müssen diese explizit initialisiert werden.
 - Dies erfolgt unter Verwendung des Zuweisungsoperators
 - Entweder bereits bei der Deklaration (siehe Variable x)
oder
 - zu einem späteren Zeitpunkt.
 - Das Lesen der Variablen vor ihrer Initialisierung ist nicht erlaubt.

Schreibender Zugriff auf Variablen

- Schreibend
 - Ein Schreiben der Variable erfolgt, wenn sie links von dem Zuweisungsoperator "=" steht.

```
x = 99;
```

- Wiederholung
 - Das erste Schreiben einer Variable ist die Initialisierung.
 - Eine final-Variable kann nach der Initialisierung nicht wieder geschrieben werden.

Lesender Zugriff auf Variablen

- Lesend
 - Ein Lesen der Variablen erfolgt, wenn
 - sie auf der **rechten Seite einer Zuweisung** auftaucht.
$$z = 2*x;$$
 - sie als **aktueller Parameter** beim Aufruf einer Methode verwendet wird.
$$\text{kreisFlaeche}(x);$$
Dabei bekommt die Parametervariable r den Wert von x zugewiesen, was der Zuweisung $r = x;$ entspricht.
 - Lesen erfordert, dass zuvor die Variable initialisiert wurde.

Lesend und schreibender Zugriff auf Variablen

- Lesend und schreibender Zugriff in einer Anweisung

$$x = 2*x;$$

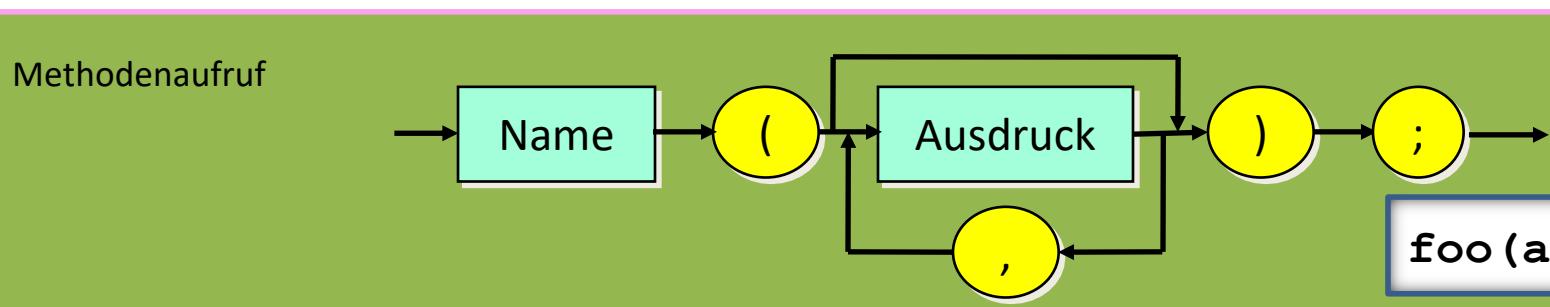
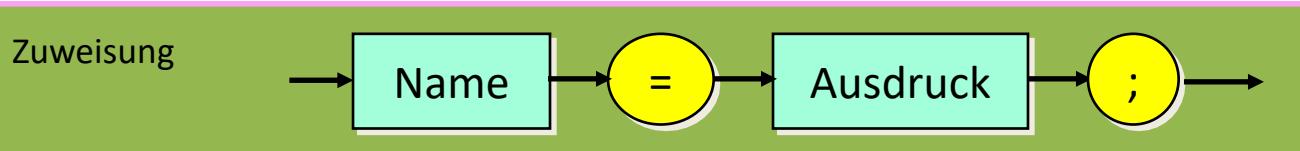
Dabei wird zunächst der Wert von x gelesen, mit zwei multipliziert und dann das Ergebnis wieder in x geschrieben.

- Diese Anweisungen haben mir schlaflose Nächte bereitet, bis ich verinnerlicht hatte, dass

'=' der Zuweisungsoperator ist (und nichts mit der mathematischen Gleichheit zu tun hat).

2.2.4 Erste Anweisungen

- Wie bei allen höheren Programmiersprachen gibt es auch in Java **einfache** und **strukturierte Anweisungen**, die auch als **Kontrollstrukturen** bezeichnet werden.
- Einige **einfache Anweisungen** wie z. B. die Variablen-deklaration, Zuweisung und Methodenaufruf haben wir bereits behandelt.
 - Eine atomare Anweisung wird stets mit einem Semikolon abgeschlossen.



return-Anweisung in Methoden

- Methoden mit einem echten Ergebnistyp (ungleich void) **müssen** durch eine return-Anweisung beendet werden.
 - Der Wert in dem Ausdruck nach dem Schlüsselwort return wird als Ergebnis der Methode an den Aufrufer geliefert.
- Methoden mit dem Schlüsselwort void **können** durch eine leere return-Anweisung (ohne Angabe eines Werts bzw. Ausdrucks) beendet werden.
 - Ansonsten enden sie mit dem letzten Befehl im Methodenrumpf.

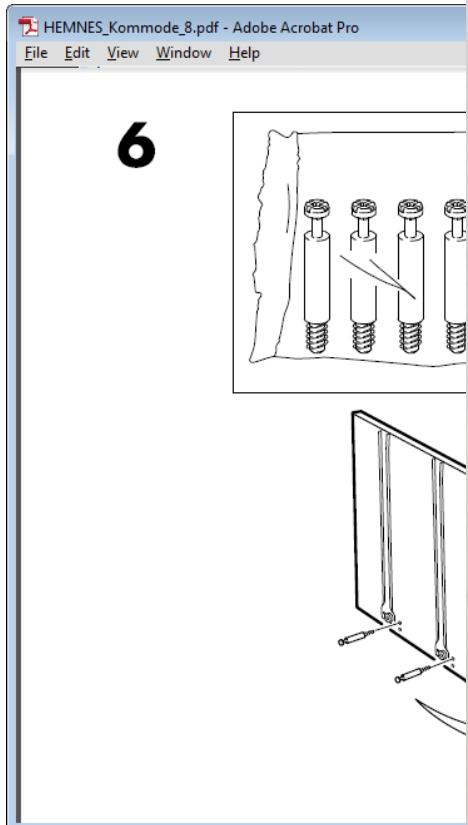
```
void tutNix(double x) {  
    System.out.println("Hallo Tutnix " + x);  
    return;  
}
```

2.3 Elementare Kontrollstrukturen

- Kontrollstrukturen sind Bestandteile von Algorithmen, die den **Ablauf der Schritte innerhalb eines Algorithmus dynamisch steuern**.
 - Aus einer endlichen Beschreibung können unendlich lange Prozesse entstehen.
- Dem **Prozessor** muss die Bedeutung der Kontrollstrukturen klar sein. Ansonsten würde zu einem Algorithmus und einer Eingabe nicht der korrekte Prozess erzeugt.
- Wichtige Kontrollstrukturen:
 - **Sequenz** (Anweisungsblock) – bereits kennengelernt
 - **Selektion** (bedingte Anweisung)
 - **Schleifen**



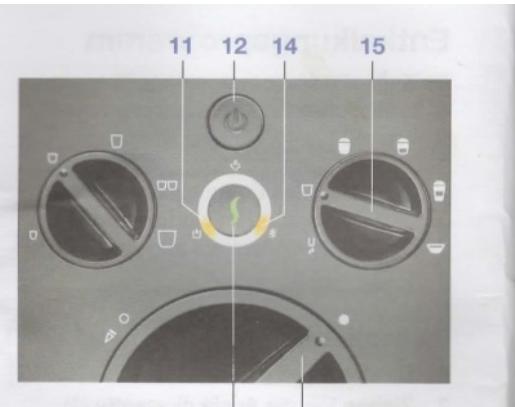
Motivation



DE

Abschnitt 1 - Entkalken

1. Stellen Sie den rechten Drehknopf (15) auf das Symbol für Kaffee oder Heißwasser.
2. Stellen Sie sicher, dass die leere Padkassette eingeschoben ist und drehen Sie den Stellknopf (17) auf Position ●.
3. Stellen Sie ein mindestens 1,8 Liter fassendes Auffanggefäß (z. B. Topf) in die Mitte auf das Aufstellgitter.
4. Schalten Sie das Gerät an der Ein/Aus-Taste ⏪ (12) ein.
5. Drücken Sie gleich nach dem Einschalten beide Tasten (12) und (16) gleichzeitig und halten Sie sie so lange gedrückt, bis die Pumpe anläuft. Das Entkalkungsprogramm läuft, die Verkalkungsanzeige ☹ (14) leuchtet dauerhaft. Nach ca. 35 Minuten ist der erste Abschnitt des Entkalkungsprogramms beendet. Die Füllstandsanzeige ⌂ (11) blinkt.



Abschnitt 2 - Spülen

1. Füllen Sie den Wassertank bis zu dessen **max**-Marke mit Leitungswasser und setzen Sie ihn wieder in das Gerät ein.
2. Stellen Sie das entleerte Auffanggefäß wieder in die Mitte auf das Aufstellgitter.
3. Drücken Sie die grün leuchtende Start/Stopp-Taste ⏸ (16). Das Spülen beginnt.
4. Sobald die Füllstandsanzeige ⌂ (11) wieder blinkt, spülen Sie noch einmal. Wiederholen Sie dazu die Schritte 1. bis 3.



Nach dem zweiten Spülen schaltet das Gerät in den Aus-Zustand.

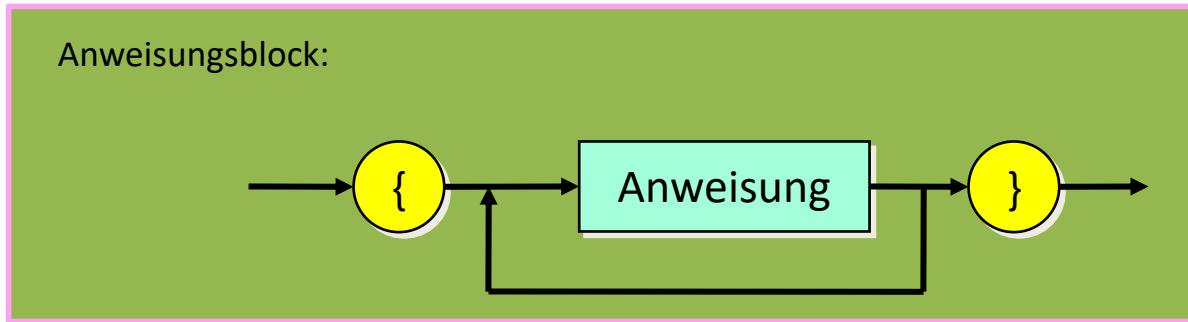


2.3.1 Wiederholung: Anweisungsblock

- Ein Anweisungsblock (auch Sequenz genannt) ist eine zusammenhängende **Teilfolge von Schritten** innerhalb eines Algorithmus.
- Prozessoren führen einen Anweisungsblock nach folgenden Regeln aus:
 - Bearbeitung der Anweisungsblock **beginnt mit dem ersten Schritt** des Anweisungsblocks
 - zu einem **Zeitpunkt** wird dann **nur ein Schritt** ausgeführt
 - **jeder Schritt wird genau einmal** ausgeführt: keine Wiederholung, kein Auslassen
 - **Reihenfolge** der Bearbeitung der Schritte ist **identisch mit** der des Algorithmus
 - Bearbeitung des Anweisungsblocks **endet mit dem letzten Schritt**

Wiederholung: Anweisungsblock in Java

- In Java wird ein Schritt als **Anweisung** bezeichnet. Die Anweisungen stehen dabei in einem Paar von Klammern.

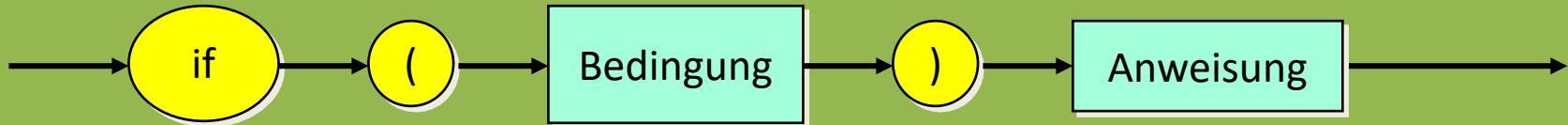


- Ein **Anweisungsblock** kann **logisch** wieder als **ein einzige Anweisung** aufgefasst werden!
- Die **Implementierung** eines **Algorithmus** besitzt **mindestens einen Anweisungsblock**, aber typischerweise gibt es **mehrere Anweisungsblöcke**.

2.3.2 Selektion

- Algorithmen erzeugen in **Abhängigkeit von den Eingabeparametern** verschiedene Prozesse.
- Bei Algorithmen benötigt man eine **von einer Bedingung abhängige Selektionsmöglichkeit**, mit welchen Schritten fortgefahrene wird.
- In Java spricht man von der **bedingten Anweisung**, die mit dem Schlüsselwort **if** beginnt und folgendermaßen aufgebaut ist:

If - Anweisung:



- Die Bedingung liefert entweder **wahr** oder **falsch**.
 - Zunächst wird die Bedingung ausgewertet.
 - Bei **wahr** wird die Anweisung ausgeführt und bei **falsch** übersprungen.

If-Anweisung in Methoden

- Häufig wird eine If-Anweisung verwendet, um inkorrekte Eingabeparameter zu korrigieren.
 - Wir wollen in der Methode kugelVolumen verhindern, dass eine negative Eingabe verarbeitet wird.
- Eine Lösung

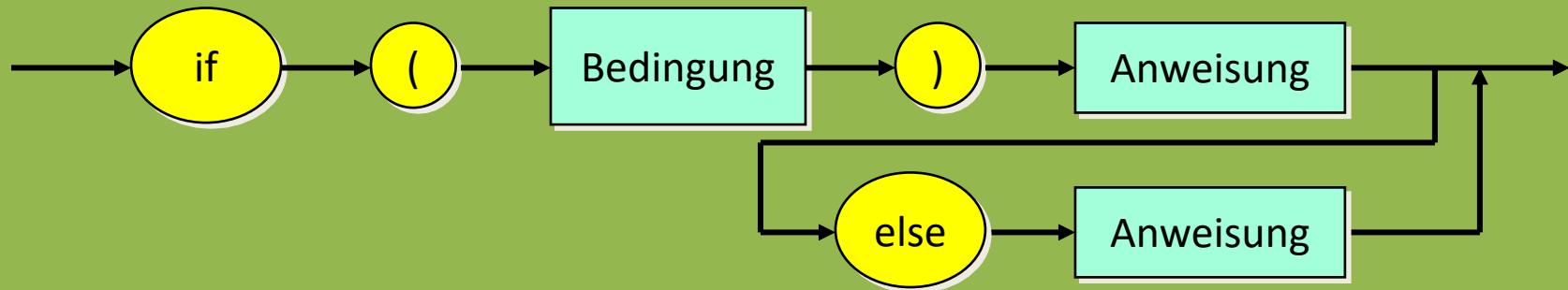
If-Anweisung in Methoden

- Häufig wird eine If-Anweisung verwendet, um inkorrekte Eingabeparameter zu korrigieren.
 - Wir wollen in der Methode kugelVolumen verhindern, dass eine negative Eingabe verarbeitet wird.
- Eine Lösung

```
double kugelVolumen (double r) {  
    final double pi = 3.14;  
    if (r < 0.0)  
        r = -1.0*r;  
    return 1.333*pi*r*r*r;  
}
```

Verallgemeinerung der Selektion

If – else - Anweisung:



- **Abarbeitung:**
 - Wiederum wird **zunächst** die **Bedingung** ausgewertet.
 - Falls die **Bedingung wahr** ist, wird mit **erster Anweisung** fortgefahrene.
 - Andernfalls (**Bedingung falsch**) wird mit **zweiter Anweisung** fortgefahrene.
 - Nach vollständiger Abarbeitung von einer der beiden Anweisungen wird mit der nachfolgenden Anweisung (nach der if-else-Anweisung) fortgefahrene.
- Eine Anweisung kann also atomar oder ein Anweisungsblock

if-else-Anweisung in Methoden

- Problem

- Wir wollen in der Methode kreisFlaeche verhindern, dass eine negative Eingabe verarbeitet wird **und der aufrufenden Methode mitteilen, dass die Eingabe nicht richtig war.**

- Eine Lösung

```
double kreisFlaeche (double r) {  
    final double pi = 3.14;  
    if (r >= 0.0)  
        return pi*r*r;  
    else  
        return -1.0;  
}
```

Später werden wir noch eine bessere Lösung sehen.

Verschachtelte if-Anweisungen

- Seien B_1 und B_2 Bedingungen und A_1 und A_2 Anweisungen. Wie ist folgende Anweisung auszuwerten?

```
if(B1) if(B2) A1 else A2
```

- Welches if und else gehören zusammen?

1

```
if(B1) if(B2) A1 else A2
```

2

```
if(B1) if(B2) A1 else A2
```

if(B1){
if(B2){
Mr.

?
} else
A2;

if(B1){
if(B2)
A1;
}
else
A2;

Verschachtelte if-Anweisungen

- Seien B_1 und B_2 Bedingungen und A_1 und A_2 Anweisungen. Wie ist folgende

```
if(B1) if(B2) A1
```

A2 wird ausgeführt,
wenn B_1 erfüllt und B_2
nicht erfüllt ist.

$B_1 = w$
 $B_2 = f$

- Welches if und else gehören zusammen?

1

```
if(B1) if(B2) A1 else A2
```

?

2

```
if(B1) if(B2) A1 else A2
```

$B_1 = f$

A2 wird ausgeführt,
wenn B_1 nicht erfüllt ist.

Live Vote

PIN: KC3W

x

<https://ilias.uni-marburg.de/vote/KC3W>



Verschachtelte if-Anweisungen

- Seien B_1 und B_2 Bedingungen und A_1 und A_2 Anweisungen. Wie ist folgende Anweisung auszuwerten?

```
if(B1) if(B2) A1 else A2
```

- Welches if und else gehören zusammen?

1

```
if(B1) if(B2) A1 else A2
```

2

```
if(B1) if(B2) A1 else A2
```



- Auflösung

Wie auch bei anderen Programmiersprachen üblich, verwendet man hier die Regel: Ein **else** ergänzt das letzte unergänzte **if**. (Antwort 1)

Formel für Schaltjahre:

Ist die Jahreszahl durch vier teilbar, aber nicht durch 100, ist es ein Schaltjahr. 2008 fällt unter diese Regel.

Ist die Jahreszahl durch 100 teilbar, aber nicht durch 400, ist es kein Schaltjahr. 2100 wird kein Schaltjahr sein.

Ist die Jahreszahl durch 400 teilbar, dann ist es ein Schaltjahr. Deshalb war das Jahr 2000 ein Schaltjahr.

Klammerung von if-Anweisungen

- Es soll ein Algorithmus geschrieben werden, um die Anzahl der Tage zu berechnen. Die Regeln hierfür sind:
 - Jedes durch 4 teilbare Jahr ist ein Schaltjahr mit Ausnahme eines Jahrs, das durch 100, aber nicht durch 400 teilbar ist.

ein schaltjahr ist ist durch 4 teilbar und nicht durch 100 teilbar
ein Schaltjahr ist durch 4 teilbar und durch 100 teilbar und durch 400 teilbar

Zeilen- nummern	<pre>1 if (jahr % 4 != 0) // Jahre nicht durch 4 teilbar 2 tage = 365; 3 else // durch 4 teilbar 4 if (jahr % 100 != 0) // nicht durch 100 teilbar 5 tage = 366; 6 else // durch 100 teilbar 7 if (jahr % 400 != 0) // nicht durch 400 teilbar 8 tage = 365; 9 else 10 tage = 366; // durch 400 teilbar 11 return tage;</pre>
--------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

- Wie könnte zu dem Codeschnipsel eine komplette Methode aussehen?

Live Vote

PIN: RT33

x

<https://ilias.uni-marburg.de/vote/RT33>



Klammerung von if-Anweisungen

```
1  if (jahr % 4 != 0) // Jahre nicht durch 4 teilbar
2      tage = 365;
3  else // durch 4 teilbar
4      if (jahr % 100 != 0) // nicht durch 100 teilbar
5          tage = 366;
6      else // durch 100 teilbar
7          if (jahr % 400 != 0) // nicht durch 400 teilbar
8              tage = 365;
9          else
10             tage = 366; // durch 400 teilbar
11 return tage;
```

- Weitere Fragen
 - Welche Zeilen werden bei folgender Eingabe ausgeführt? - 2048, 2015, 2800, 2200
 - Die Zuweisung in welcher Zeile wird jeweils bei der Eingabe für "jahr" ausgeführt? – 2800, 2200
 - Welche Eingabe für "jahr" führt dazu, dass die Zuweisung in Zeile 5 ausgeführt wird? Welche führt zu der Ausführung von Zeile 10?

2.3.3 while-Schleife

- Bei vielen Problemen werden Schritte solange wiederholt wie eine Bedingung wahr ist.
 - In fast allen Programmiersprachen wird deshalb als Kontrollstruktur die while-Schleife angeboten. In Java lautet die genaue Syntax:

while - Anweisung:



- Verhalten der while-Schleife
 - Die Bedingung wird ausgewertet. Wenn das Ergebnis **wahr** ist, wird die Anweisung abgearbeitet und die Bedingung erneut ausgewertet. Wenn das Ergebnis wahr ist, wird die Anweisung abgearbeitet und die Bedingung erneut ausgewertet usw....
 - Wenn die Bedingung zum ersten Mal **falsch** ist, wird die Anweisung **nicht ausgeführt** und die **while-Schleife beendet**.
- Problem:
 - Wenn die Schleifenbedingung **nie falsch** liefert → **keine Terminierung**.

Beispiele: Sternenmuster

- Schreibe eine Methode, um in einer Zeile beginnend von der ersten Position k Sterne auszugeben.

```
/** Die Methode printStars gibt Sterne in einer Zeile aus.
 * @param k Anzahl der Sterne
 */
void printStars(int k) {
    int i = 0;
    while (i < k) { // Blockanweisung
        System.out.print("*");
        i = i + 1;
    }
}
```

Beispiele: Sternenmuster

- Methode um n Zeilen mit Sternen auszugeben, $n > 0$.
 - Jede Ausgabe in Zeile i beginnt an der ersten Position und gibt i Sterne aus.

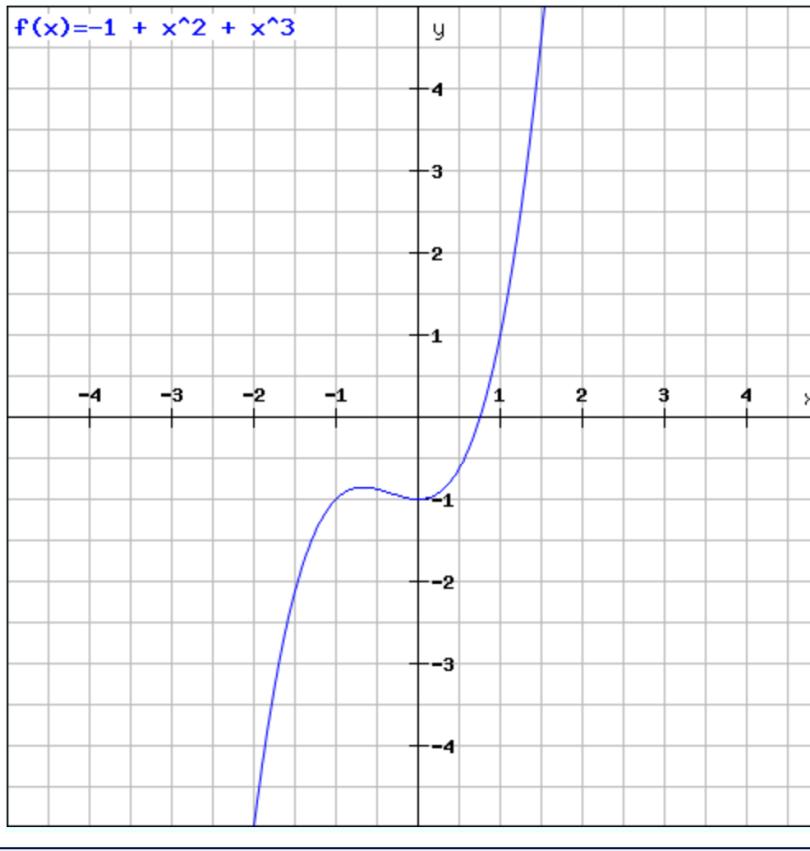
```
/** Die Methode printStarRows gibt Zeilen mit abnehmender
 * Anzahl an Sternen aus.
 * @param k Anzahl der Sterne
 */
void printStarRows(int k) {
    while (k > 0) { // Blockanweisung
        printStars(k);
        System.out.println();
        k = k - 1;
    }
}
```

Beispiel: Bisektionsverfahren

- Problem
 - Gegeben ist eine stetige reelle Funktion $f: \mathbb{R} \rightarrow \mathbb{R}$, zu der wir nicht analytisch die Nullstellen berechnen können.
 - Wir nutzen ein Näherungsverfahren, bei dem ein reellwertiges x gesucht ist, so dass
$$|f(x)| < \varepsilon$$
ist.
 - Wenn wir zwei Punkte a, b kennen mit $a < b$ und $f(a)*f(b) < 0$ lässt sich ein sogenanntes Bisektionsverfahren anwenden.
- Die Lösung gibt es dann Live in der Vorlesung.

Beispielfunktion

- Gegeben die Funktion $f(x) = -1 + x^2 + x^3$
- Wir sehen: die Nullstelle liegt zwischen 0 und 1





Beispiel: Berechnung von π

- Die Kreiszahl kann näherungsweise über die von dem Mathematiker Gottfried Wilhelm Leibniz entwickelten Summenformel berechnet werden.

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

- Entwicklung einer Methode für die Summenformel
 - Iterative Berechnung mit einer while-Schleife
 - Abbruch der Schleife nach dem i-ten Schritt, falls die Änderungen der partiellen Summe des i-ten Schritts im Vergleich zum (i-1)-ten Schritt unter einer Schranke ε liegt.
 - ε ist dabei ein Eingabeparameter der Methode
 - Die Methode liefert dann den Wert der partiellen Summe nach i Durchläufen.

Java-Methode

```
/**  
 * Der Algorithmus berechnet approximativ die Kreiszahl pi  
 * @param epsilon Abbruch der Iteration, wenn | Summand | < epsilon  
 * @return Approximativer Wert für die Zahl pi  
 */  
double getPi(double epsilon) {  
    double sum = 1.0;                      // Erste partielle Summe  
    double sign = -1.0;                     // Vorzeichen  
    double nenner = 3.0;                    // Divisor  
    while (1.0/nenner > epsilon) {          // Abbruchkriterium  
        sum = sum + sign/nenner;           // Nächste partielle Summe  
        nenner = nenner + 2;               // Erhöhung des Divisors  
        sign = -1.0*sign;                 // Vorzeichenwechsel  
    }  
    return 4*sum;                          // Ergebnis  
}
```

Beispiel: Euklidscher Algorithmus

- Gegeben
 - Zwei ganze Zahlen m und n mit $m > 0$ und $n > 0$.
- Berechnung des größten gemeinsamen Teilers z
 - z ist Teiler von m und z ist Teiler von n
 - z ist der größte gemeinsame Teiler

Beispiel: Euklidscher Algorithmus

- Idee des Algorithmus
 - Annahme: $n > m$, z sei ggt von n und m

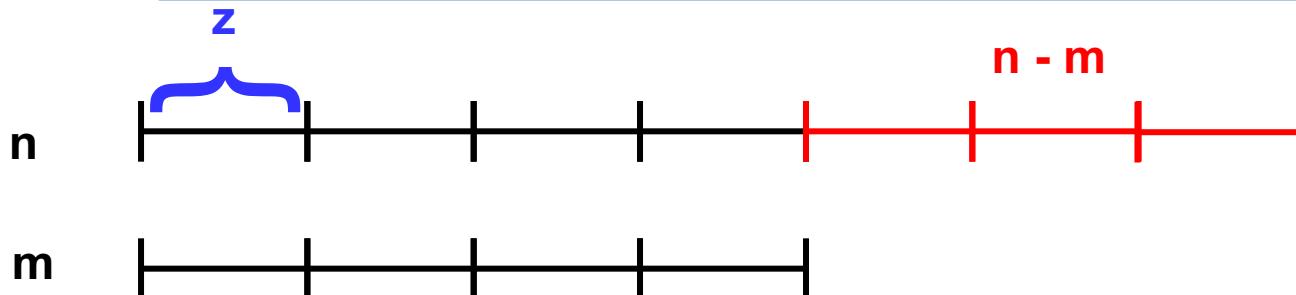
Mathematische Überlegungen

Man kann sich überlegen:

1. z teilt auch $n-m$
2. es gibt keinen größeren Teiler für $n-m$ und m

Zudem gilt noch eine Eigenschaft:

- Das Problem für $n-m$ und m ist einfacher!
- Im besten Fall gilt sogar: $n-m == m \rightarrow$ Dann sind wir fertig!



Aufgabe ggt mit modulo operation

Algorithmus in Java

```
/**  
 * Der Algorithmus berechnet den ggt von zwei ganzen Zahlen > 0  
 * @param n erste ganze Zahl, n > 0  
 * @param m zweite ganze Zahl. m > 0  
 * @return grter gemeinsamer Teiler von n und m  
 */  
int ggt(int n, int m) {  
    while (n != m)  
        if (n > m) // Hier gilt n > m  
            n = n - m;  
        else // Hier gilt m > n  
            m = m - n;  
    // Nach der Schleife gilt n == m  
    return n;  
}
```

Zusammenfassung

- Wichtige Konzepte der imperativen Programmierung
 - Datentypen: Wertemenge und Operationen
 - Variablen
 - Zuweisungsoperator
 - Algorithmen und Methoden
 - Funktionen: Methoden mit echtem Datentyp als Rückgabe
 - Prozeduren: Methode mit dem Schlüsselwort void.
 - Parameterliste
 - Kontrollstrukturen
 - Anweisungsblock: {...}
 - Bedingte Anweisung: if (B) ... else ...
 - While-Schleife: while (B) ...

3. Entwurfsprinzipien von Algorithmen

- Konventionen für lesbaren Code
- Entwurfsprinzipien
 - Schrittweise Verfeinerung
 - Rekursive Algorithmen

3.1 Kommentare & Konventionen

- Standard Java Konventionen:
<https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>
- Namenskonventionen
- Formatierungsanweisungen
- Kommentarregeln
- Programmierstil
- **Siehe auch Konventionen auf ILIAS**

Namenskonventionen

- Parameter / Variablen:
 - Beginn mit Kleinbuchstaben; Anfangsbuchstaben weiterer Wortteile groß; in der Regel ein Nomen
 - `radius`
 - `lowerBound`
- Methoden:
 - Beginn mit Kleinbuchstaben; Anfangsbuchstaben weiterer Wortteile groß; meistens ein Verb (Imperativ)
 - `ggt`
 - `kreisFlaeche`
 - `oeffne`
- Konstanten:
 - ausschließlich Großbuchstaben; Trennung von Wortteilen durch Unterstrich (`_`)
 - `PI`
 - `MAX_VALUE`

Formatierung

- Hilfsmittel
 - Zeilenumbruch
 - Einrückung
- Eine Deklaration pro Zeile, am Blockanfang

- Beispiele
 - Format von Methodenköpfen

- sample(int i, int j) {
 }
 ...

Bei if, while, etc. am besten immer Anweisungsblöcke statt atomarer Anweisungen

- Format von Anweisungen

- if (*condition*) {
 }
 } else {
 ...
}

Blöcke konsistent einrücken

Kommentare

- Wichtige Zusatzinformationen zur Dokumentation von Algorithmen
 - Bedeutung der Eingabe (Vorbedingungen)
 - Bedeutung der Ausgabe (Nachbedingungen)
- Kommentare werden vom Java-Compiler ignoriert.
- Arten von Kommentaren in Java

Kommentartyp	Beginn	Ende
einzeilig	//	Zeilenende
mehrzeilig	/*	*/
JavaDoc	/**	*/

Dieser Kommentar wird von dem Werkzeug javadoc genutzt, um aus dem Inhalt des Kommentars z.B. html-Seiten zu erzeugen.

Das Werkzeug javadoc

- javadoc erzeugt aus einer java-Datei unter Verwendung der Kommentare `/** ... */` eine **html-Datei**
 - und noch ganz viele andere Dateien.
- Spezielle **Tags mit dem Präfix @** im Kommentar haben eine Bedeutung.
 - Allgemein verwendbare Tags
 - `@author` für Namen des Autors
 - `@version` für die Version des Programms
 - `@see` für Verweise
 - Tags für Methoden
 - `@param` für die Methodenparameter
 - `@return` für das Ergebnis
 - `@see` für Verweise
- Erster Satz muss eine vollständige Kurzbeschreibung sein
- Weitere Sätze für zusätzliche Erläuterungen

Verwendung später,
wenn wir Klassen haben.

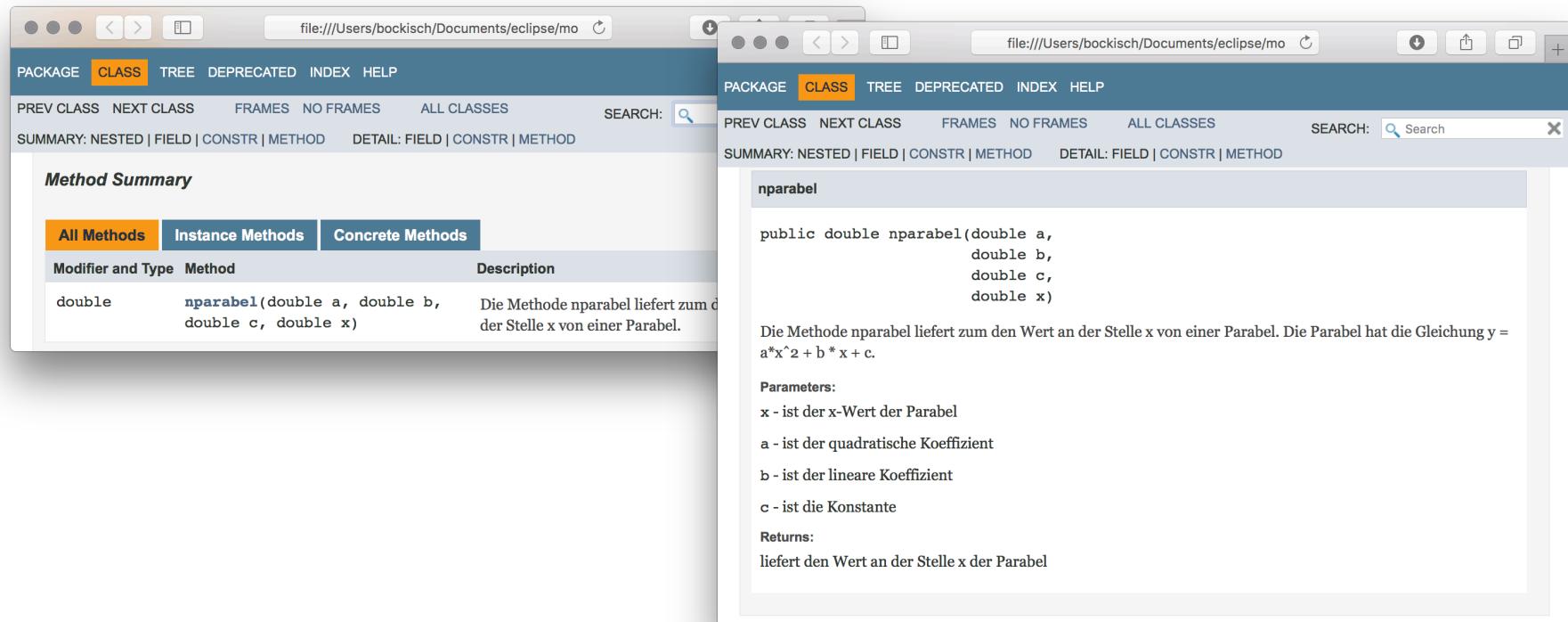
Beispiel: Parabel (1)

How to generate java doc with cmd ?

Generate javadoc with intellij idea:
go to tools and then generate javadoc

```
/** Die Methode nparabel liefert zum den y-Wert an der Stelle x von einer Parabel.  
 * Die Parabel hat die Gleichung y = a*x^2 + b * x + c.  
 * @param x ist der x-Wert der Parabel  
 * @param a ist der quadratische Koeffizient  
 * @param b ist der lineare Koeffizient  
 * @param c ist die Konstante  
 * @return liefert den Wert an der Stelle x der Parabel  
 */  
public double nparabel(double a, double b, double c, double x) {  
    return a*x*x + b*x + c;  
}
```

JavaDoc



- JavaDoc Tool ist Klassenbasiert
 - Daher Anwendung erst später
 - Aber: kommentieren Sie Methoden bereits nach diesen Regeln!

Lesbarkeit von Programmen

- Die anderen Arten von Kommentaren sind primär **für Programmentwickler** gedacht und sind deshalb auch wichtig.
 - Methoden werden nicht nur benutzt, sondern können bei großen Softwareprojekten durch einen anderen Entwickler verändert werden.
- Generell ist beim Verstehen von Programmcode auch die **Formatierung des Programmtextes** wichtig.
 - Die Formatierung hat keinen Einfluss auf die syntaktische Korrektheit eines Programms, sondern dient „nur“ dazu, die Lesbarkeit zu verbessern.
 - Zeilenumbrüche, Leerzeichen und Einrücken sollten verwendet werden, auch wenn dies nicht durch die Programmiersprache gefordert wird.
 - *Wir werden deshalb darauf achten, die Programmierrichtlinien (siehe ILIAS) einzuhalten.*

3.2 Schrittweise Verfeinerung von Algorithmen

- Entwurf der Algorithmen ist relativ einfach solange die Problemstellungen einfach sind.
 - Leider sind nahezu alle interessanten Problemstellungen komplex!

Schrittweise Verfeinerung (**Top-Down Ansatz**)

- Idee
 1. Zerlege das Problem geeignet in mehrere Einzelprobleme
 2. Verfahre mit jedem in Schritt 1 erzeugten Teilproblem folgendermaßen:
 - Falls das Teilproblem genügend einfach ist: Entwirf Algorithmus für das Problem
 - Ansonsten: Wende das Verfahren der schrittweisen Verfeinerung auf dieses Problem erneut an.

Beispiel

- Algorithmus Entkalken-Komplett
 1. Vorbereitung
 2. Entkalken
 3. Spülen
- Jeder dieser Schritte muss wieder durch einen eigenen Algorithmus genau beschrieben werden.



Algorithmus zum Spülen der Kaffeemaschine

1. Setze den vollen Wassertank in das Gerät ein.
2. Stelle das Auffanggefäß in die Mitte des Auffanggitters.
3. Drücke die grün leuchtende Taste.

Beispiel

- **Problem**
 - Schritt 1 ist noch nicht detailliert genug, d.h. der Schritt kann nicht interpretiert und deshalb auch nicht ausgeführt werden.
- **Erneute Verfeinerung des Algorithmus**

Algorithmus zum Einsetzen des vollen Wassertanks

 1. Fülle den Wassertank mit Trinkwasser bis zur max-Marke auf.
 2. Klappe die Wassertankhalterung des Geräts auf.
 3. Setze den vollen Tank auf die Halterung.
 4. Bringe die Wassertankhalterung wieder in die senkrechte Position.

Aspekte bei der schrittweisen Verfeinerung

- Kenntnisse über die Fähigkeiten des Prozessors sind von zentraler Bedeutung bei der schrittweisen Verfeinerung.
 - Wenn ein Schritt noch nicht vom Prozessor ausgeführt werden kann, muss noch eine Verfeinerung vorgenommen werden.
 - Wünschenswert ist auch eine Verfeinerung von Schritten, deren Ausführung noch verbessert werden kann.
 - Die Fähigkeiten des Prozessors legen die Richtung der Verfeinerung fest.
- Bisher entwickelte Algorithmen, die der Prozessor ausführen kann, können wieder beim Entwurf anderer Algorithmen als elementare Schritte genutzt werden, z.B. „Einsetzen Wassertank“
 - Wichtig beim Spülen der Kaffeemaschine und beim Zubereiten von Kaffee → Prozessor erweckt dadurch den Anschein, komplexere Operationen direkt ausführen zu können.

Beispiel: Maximum – Version 1

- Aufgabe
 - Berechnung des Maximums von 3 Zahlen
- Lösungsidee
 - Nehmen wir an, dass wir bereits eine Methode max2 hätten, um das Maximum von zwei ganzen Zahlen zu berechnen.
 - Dann können wir das Maximum von drei Zahlen darauf zurückführen.

```
/**  
 * Der Algorithmus berechnet das Maximum von drei ganzen Zahlen  
 * @param m erste ganze Zahl  
 * @param n zweite ganze Zahl  
 * @param p dritte ganze Zahl  
 * @return Maximum von m, n und p  
 */  
int max3(int m, int n, int p) {  
    int tmp = max2(n,p); // Berechne zunächst das Maximum von n und p  
    return max2(m, tmp); // Danach das Maximum von m und tmp  
}
```

Beispiel: Maximum – Version 1

- Aufgabe
 - Berechnung des Maximums von 3 Zahlen
- Lösungsidee
 - Nehmen wir an, dass wir bereits eine Methode max2 hätten, um das Maximum von zwei ganzen Zahlen zu berechnen.
 - Dann können wir das Maximum von drei Zahlen darauf zurückführen.

```
/**  
 * Der Algorithmus berechnet das Maximum von drei ganzen Zahlen  
 * @param m erste ganze Zahl  
 * @param n zweite ganze Zahl  
 * @param p dritte ganze Zahl  
 * @return Maximum von m, n und p  
 */  
int max3(int m, int n, int p) {  
    return max2(m, max2(n,p));  
}
```

Die Variable tmp können wir einsparen, indem wir den Aufruf direkt dort hinschreiben, wo auf tmp lesend zugegriffen wird.

Vorteile der schrittweisen Verfeinerung

- Komplexe Problemstellungen werden in einfachere Probleme aufgeteilt. Jedes der Probleme kann durch eine eigene Methode gelöst werden.
 - Änderungen einer Methode haben keinen Einfluss auf das aufrufende Programm, solange das Verhalten der Methode gleich bleibt.
 - Beim Benutzen einer Methode ist nur erforderlich, **was** die Methode leistet, aber **nicht, wie** die Methode dies erledigt (**prozedurale Abstraktion**).
 - Korrektheit eines großen Programms kann einfacher überprüft werden, da dies auf die Korrektheit der Methoden abgebildet werden kann.
 - Methoden können unabhängig voneinander im **Team** entwickelt werden.
- Fertig implementierte Methoden können beim Entwurf anderer Algorithmen **wiederverwendet** werden (→ **Softwarebibliotheken**)
 - In der Java-Klasse Math sind bereits viele mathematische Funktionen vorhanden. Wir können diese benutzen, indem wir vor dem Methodennamen noch den Klassennamen gefolgt von einem Punkt schreiben.

Math.log(5)

3.3 Rekursion

- Methodenaufruf
 - Die Methodenausführung hat einen eigenen lokalen Speicher für lokale Variablen

```
/** JavaDoc Kommentar fehlt aus Platzgründen
 */
int max3(int m, int n, int p) {
    int tmp = max2(n,p); // Berechne zunächst das Maximum von n und p
    return max2(m, tmp); // Danach das Maximum von n und tmp
}
/** JavaDoc Kommentar fehlt aus Platzgründen
 */
int max2(int m, int n) {
    if (m > n) {
        return m;
    }
    else {
        return n;
    }
}
int tmp = 21;
max3(1, 2, 3);
tmp;
```

3.3 Rekursion

- Methodenaufruf
 - Die Methodenausführung hat einen eigenen lokalen Speicher für lokale Variablen

```
/** JavaDoc Kommentar fehlt aus Platzgründen
 */
int max3(int m, int n, int p) {
    int tmp = max2(n,p), // Berechnung von tmp
        return max2(m, tmp); // Danach
}
/** JavaDoc Kommentar fehlt aus Platzgründen
 */
int max2(int m, int n) {
    if (m > n) {
        }
    else {
        }
}
int tmp = 21;
max3(1, 2, 3);
tmp;
```

Zuweisen von 3 an tmp
im Kontext der
Methodenausführung

n und p

Zuweisen von 21
an tmp im
Standard-Kontext.

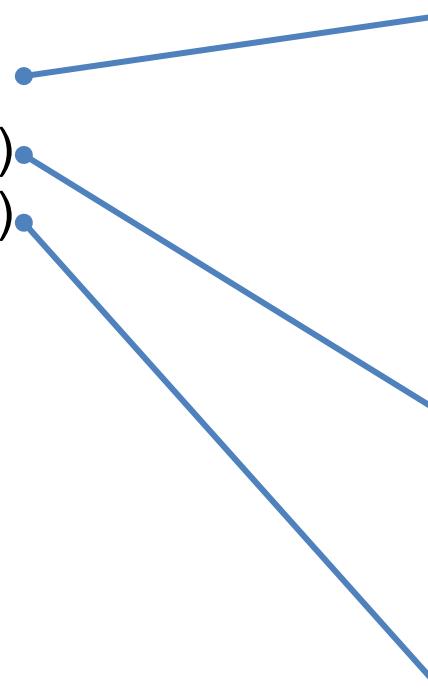
Ergibt 21, da wir im
Standard-Kontext sind.

Speicherreservierung

Aufruf von `max3(1, 2, 3)`

→ Aufruf von `max2(2, 3)`

→ Aufruf von `max2(1, 3)`



Variablenname	Wert
m	1
n	2
p	3
tmp	3

Variablenname	Wert
m	2
n	3

Variablenname	Wert
m	1
n	3

Speicherreservierung

Aufruf von `max3(1, 2, 3)`

→ Aufruf von `max2(2, 3)`

→ Aufruf von `max2(1, 3)`

Ein neuer Speicherbereich
wird für **jeden Aufruf**
einer Methode angelegt!

Lokale Variablen für `max3`

Variablen

	vwert
m	1
n	2
p	3
tmp	3

Lokale Variablen für `max2`

Variablen

m	2
n	3

Lokale Variablen für `max2`

Variablen

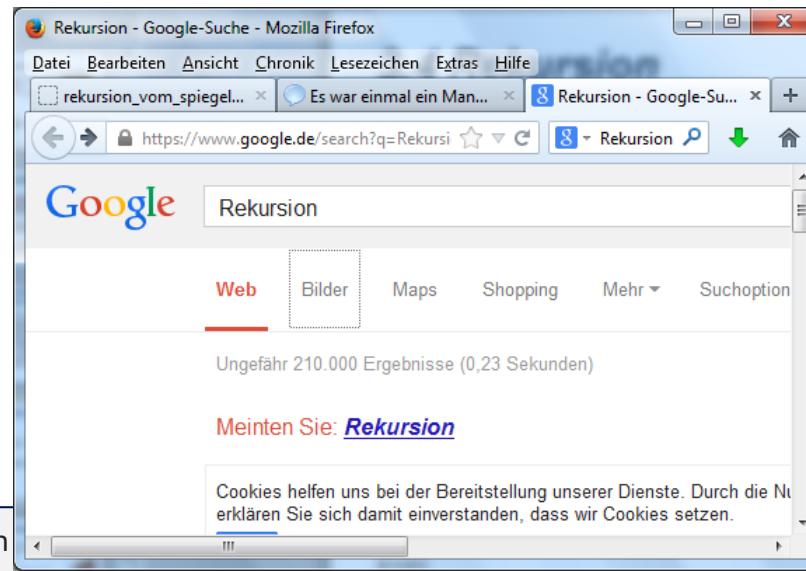
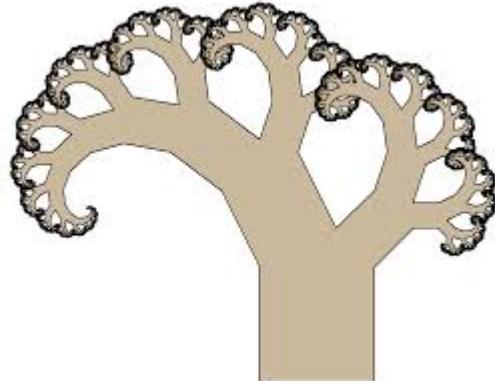
m	1
n	3

Wichtige Eigenschaft

- Bei jedem Aufruf einer Methode wird eine neue Methodeninstanz erzeugt. Diese bekommt ihren eigenen Speicherbereich, in dem insbesondere die Werte der Parametervariablen und der lokalen Variablen hinterlegt werden.
- Nach dem Beenden dieser Methodeninstanz wird dieser Speicher zurückgegeben.
- Dies gilt auch, wenn dieselbe Methode mehrmals aufgerufen wurde und somit mehrere Instanzen der Methode existieren können.

Rekursion

- Bisher
 - Lösung eines Problems P werden auf die Lösung eines anderen Problems Q zurückgeführt.
- Jetzt
 - Lösung eines Problems P durch Zurückführen auf die Lösung eines Problems vom Typ P oder mehrerer einfachere Probleme vom Typ P.



Beispiel (Matrjoschka-Puppen)

```
/* Öffnet rekursiv die Puppen, entfernt die massive Puppe und schließt die Puppen. */
```

```
void nimmMassivePuppe(Puppe M) {
```

```
    if (M ist massiv) {
        nimm M;
    }
    else {
        öffne M;
        nimmMassivePuppe(Inhalt von M);
        schließe M;
    }
}
```

- Zweiter Schritt im **else**-Zweig bewirkt, dass die gleiche Methode wieder aufgerufen wird, jetzt aber mit der nächst kleineren Puppe.
- Algorithmus arbeitet in 2 Phasen
 1. Phase: Öffnen aller Puppen
 2. Phase: Schließen aller Puppen



Berechnung der Fakultät

- Aufgabenstellung: Berechnung von $n!$, wobei
$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n = (1 \cdot 2 \cdot 3 \cdots (n-1)) \cdot n = ((n-1)!) \cdot n$$
- Wir nutzen dabei aus, dass folgende Eigenschaft gilt.

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

- Direkte Umsetzung als rekursive Methode in Java

```
int fact(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fact(n-1);  
}
```



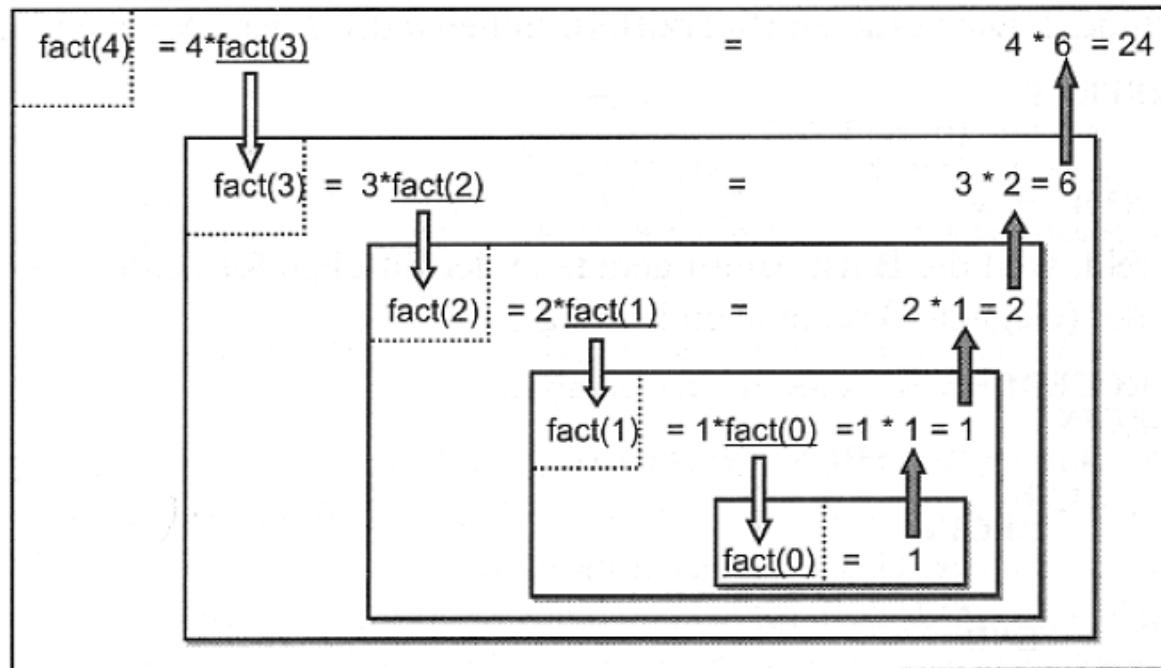
Veranschaulichung des Prozesses für fact(4)

Methodenaufrufe

Ergebnisrückgabe

Unterprogrammaufrufe

Ergebnisrückgabe



Wichtige Anforderungen an rekursive Algorithmen

- Folge von rekursiven Methodenaufrufen muss endlich sein
 - Eingabe für den nächsten rekursiven Aufruf muss „einfacher“ werden.
 - Eine bedingte Anweisung (bzw. eine entsprechende Kontrollstruktur) muss den einfachsten Fall behandeln, an dem die Rekursion nicht fortgesetzt wird.
- Beispiel
 - Matrjoschka
 - Bei jedem Rekursionsaufruf nähern wir uns der massiven Puppe.
 - Eine bedingte Anweisung führt zu dem Fall, wenn wir auf die massive Puppe treffen.
 - Fakultätsberechnung
 - Bei jedem Rekursionsaufruf nähern wir uns der Null.
 - Eine bedingte Anweisung behandelt den Fall der Berechnung von $0!$.



Lineare Rekursion (auch: Struktureller Rekursion)

- Problem P wird auf genau ein einfacheres Problem des gleichen Typs rekursiv zurückgeführt.
 - Diese Art der Probleme können dann auch direkt durch eine while-Schleife gelöst werden (und umgekehrt).
- Aufgabe: Berechne die Fakultät der Zahl n ohne Verwendung einer Rekursion.

```
int facultaeWhile(int n) {  
    int erg = 1;  
    while (n > 0) {  
        erg = erg*n;  
        n = n-1;  
    }  
    return erg;  
}
```

Baumartige Rekursion

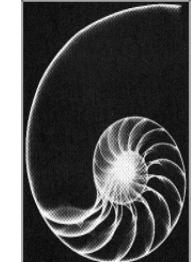
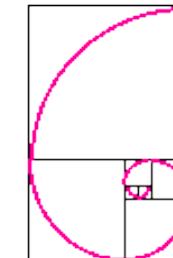
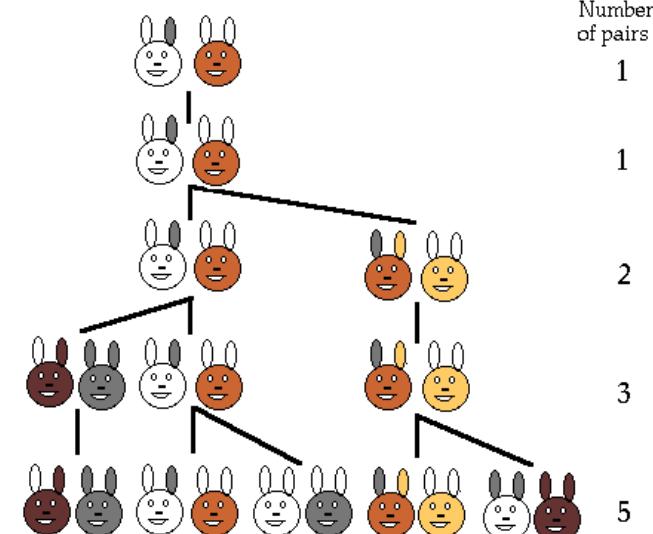
- Problem P wird in einem Rekursionsschritt auf **mehrere Probleme** des gleichen Typs zurückgeführt.
 - Jedes dieser neuen Probleme muss einfacher sein als das ursprüngliche Problem.
 - Diese Programme sind i. A. nicht mehr so einfach in nicht-rekursive Programme mit Schleifen umzusetzen.

Die Fibonacci-Folge

- $(F_n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$
- Bildungsgesetz:
 - $F_0 = 1, F_1 = 1,$
 - $F_{n+1} = F_n + F_{n-1}$

Trivia:

- 1202 entwickelt um Wachstum von Kaninchenpopulationen zu beschreiben
- Weitere Wachstumsprozesse: Blütenstände, Tannenzapfen, Schneckenhäuser
- Verhältnis einer Fibonacci-Zahl zur vorangegangenen nähert sich Goldenem Schnitt (Ästhetik) an



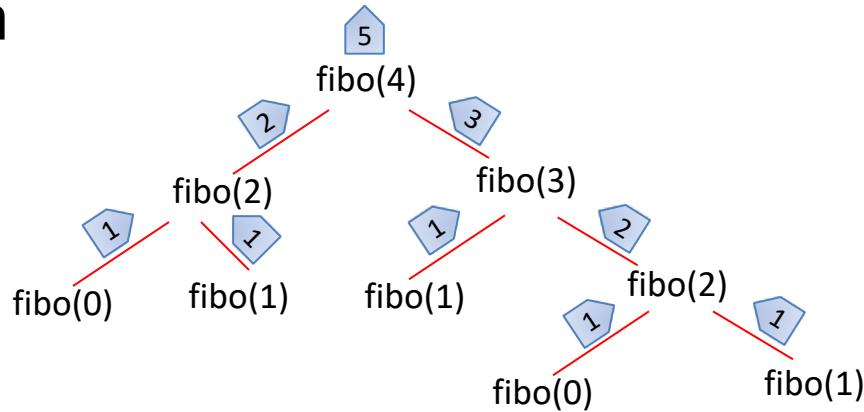


Lösung

- Direkte Umsetzung als rekursive Methode

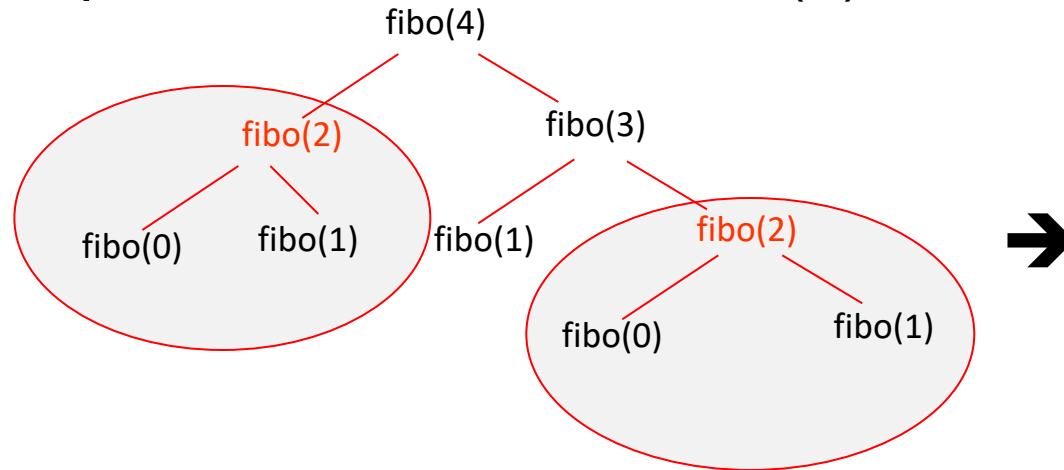
```
int fibo(int n){  
    if (n < 2)  
        return 1;  
    else  
        return fibo(n-2)+fibo(n-1);  
}
```

- Aufrufbaum



3.3.1 Laufzeit

- Durch eine schlecht programmierte Rekursion kann ein Programm viel Zeit benötigen.
 - Das gleiche Problem in einer Rekursion kann **mehrfach berechnet** werden.
- Beispiel: Aufrufbaum von $\text{fibo}(4)$



→
5-mal $\text{fibo}(1)$ oder $\text{fib}(0)$
2-mal $\text{fibo}(2)$
1-mal $\text{fibo}(3)$

Laufzeit von fibo

- Um es etwas einfacher zu machen, sei n gerade.
- Für die Lösung von $\text{fibo}(n)$ muss für $n > 2$ mindestens 2-mal die Lösung von $\text{fibo}(n-2)$ berechnet werden. Damit folgt:

- 4-mal $\text{fibo}(n-4)$ [bei $n > 4$]
- 8-mal $\text{fibo}(n-6)$ [bei $n > 6$]
- 16-mal $\text{fibo}(n-8)$ [bei $n > 8$]
-
- $2^{n/2} - \text{mal fibo}(0)$

- **Vermutung**
 - Falls $n > 1$ gerade ist, wird $\text{fib}(0) 2^{n/2}$ -mal berechnet.
 - (Formaler Beweis mit Hilfe vollständiger Induktion möglich, wird aber in dieser Vorlesung nicht gezeigt)



Die Laufzeit der Fibonacci-Algorithmus lässt sich optimieren. Siehe VL "Datenstrukturen und Algorithmen", "Deklarative Programmierung".

3.3.2 Terminierung

- Wichtig dabei ist die Vereinfachung des Problems und die Behandlung des einfachsten Sonderfalls.
- Einfachheit lässt sich nicht immer auf einen ganzzahligen Parameter zurückführen.
- Beispiel ggt

```
int ggtRekursiv(int a, int b) {  
    if (a==b) // Stoppbedingung für die Rekursion  
        return a;  
    else {  
        if (a>b) // Rückführung auf ein einfaches Problem  
            return ggtRekursiv(a-b,b);  
        else  
            return ggtRekursiv(b-a,a);  
    }  
}
```

Terminierung rekursiver Funktionen

- Damit eine rekursive Funktion

```
typerg rekFunk(typ1 p1, ..., typk pk) {  
    ... rekFunk(e1, ..., ek) ...  
}
```

terminiert, müssen die Parameterwerte e_1, \dots, e_k des rekursiven Aufrufes „einfacher“ sein als die Parameterwerte p_1, \dots, p_k des originalen Aufrufs

- Was heißt „einfacher“ ?

Es muss eine mathematische Funktion

$$F : t_1 \times \dots \times t_k \rightarrow \mathcal{N}$$

geben mit

$$F(p_1, \dots, p_k) > F(e_1, \dots, e_k) \geq 0.$$

Beispiele

```
static int fibo(int n){  
...  
}
```

$$F(n) = n$$

```
static int ggT(int m,int n){  
...  
}
```

$$F(m,n) = m+n$$

```
static int McCarthy(int n){  
    if (n > 100)  
        return n-10;  
    else  
        return McCarthy(McCarthy(n+11));  
}
```

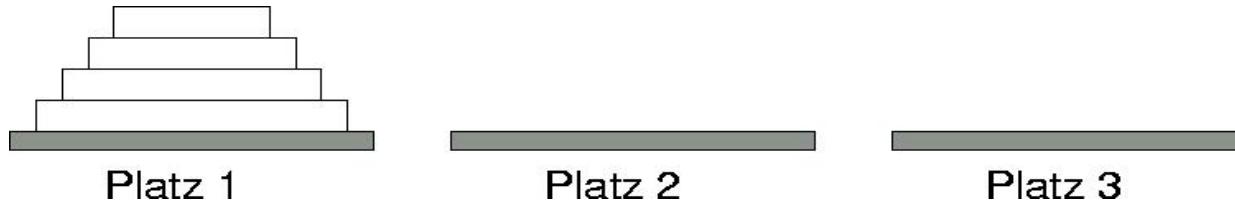
- Experimentieren Sie mit dieser Funktion
- Zeigen Sie, dass sie immer terminiert

John McCarthy war ein Logiker und Informatiker, dem seine großen Beiträge im Feld der Künstlichen Intelligenz den Turing Award von 1971 einbrachten.

3.3.3 Türme von Hanoi

Problem

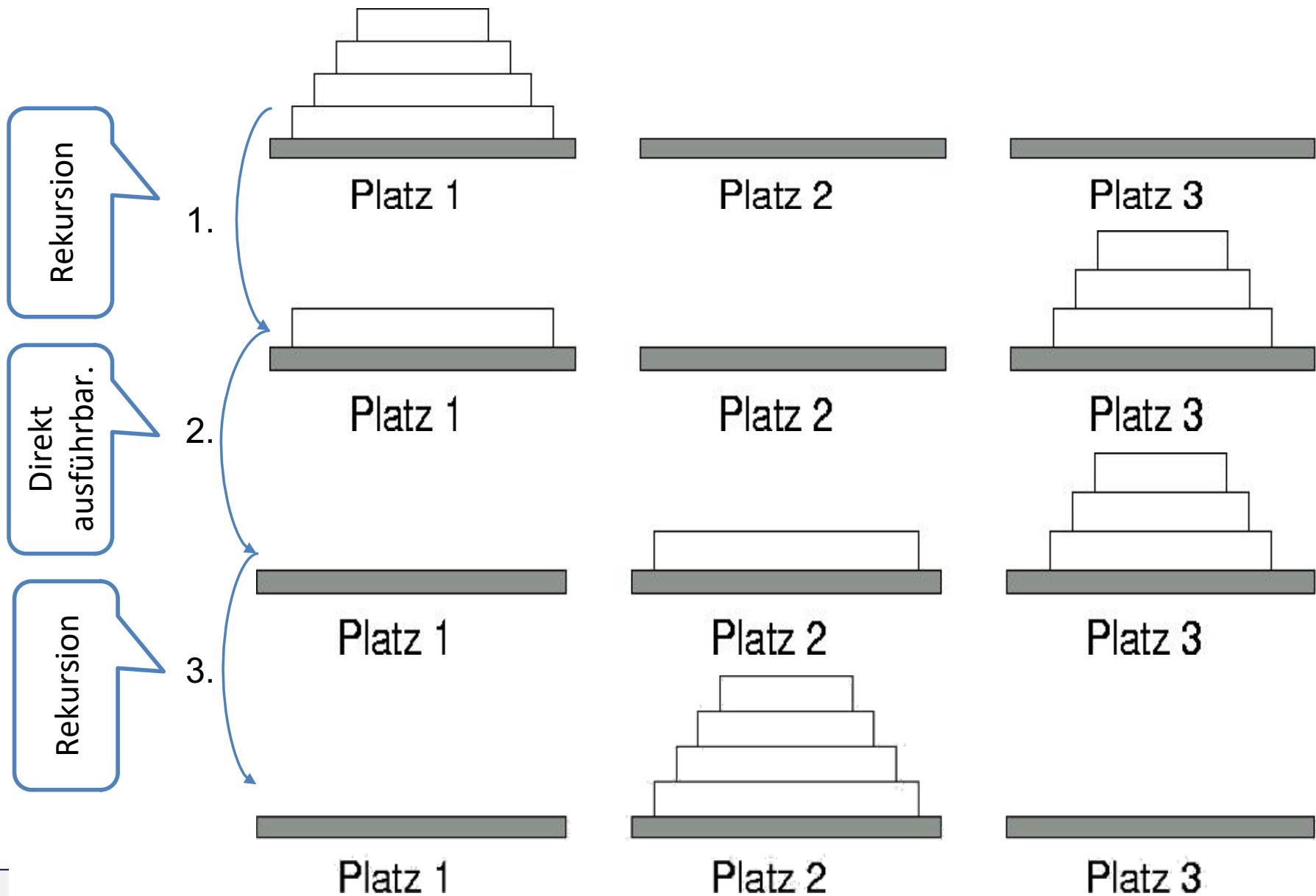
- Gegeben 3 Plätze und ein Stapel von Scheiben, die auf einem der Plätze ihrer Größe nach aufeinander liegen



- Ziel ist es den Stapel unter Beachtung folgender Regeln auf einen anderen Platz zu legen.
 - Pro Zug wird immer **nur eine Scheibe** bewegt.
 - Es darf **nie eine größere auf einer kleineren Scheibe** liegen.
 - Es dürfen alle drei Plätze zur Zwischenlagerung von Scheiben benutzt werden.

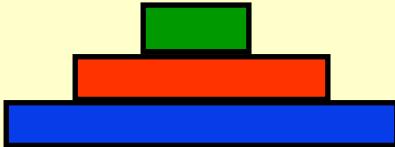
Rekursive Lösung

- Algorithmus für 4 Scheiben von Platz 1 auf Platz 2 (3 dient als Zwischenlager)
 1. Übertrage die oberen 3 Scheiben von Platz 1 nach Platz 3 (Rekursion!)
 2. Bewege die untere Scheibe von Platz 1 nach Platz 2
 3. Übertrage die 3 Scheiben von Platz 3 nach Platz 2 (wieder Rekursion!)
- Verallgemeinerung für n Scheiben von Platz 1 auf Platz 2
 1. Übertrage die oberen $n-1$ Scheiben von Platz 1 nach Platz 3 (Rekursion!)
 2. Bewege die untere Scheibe von Platz 1 nach Platz 2
 3. Übertrage die $n-1$ Scheiben von Platz 3 nach Platz 2 (wieder Rekursion!)

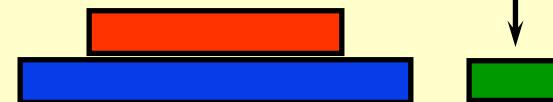


Türme von Hanoi mit drei Scheiben

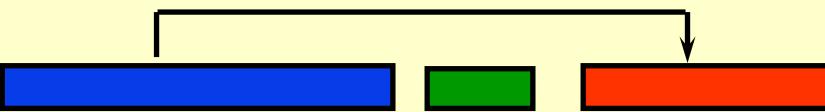
Ausgangslage



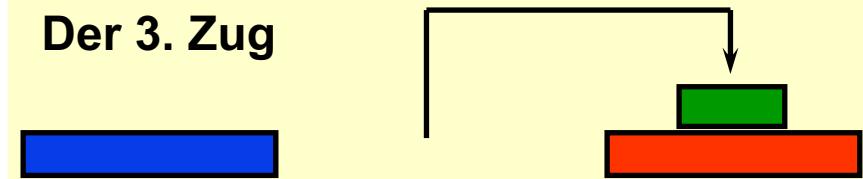
Der 1. Zug



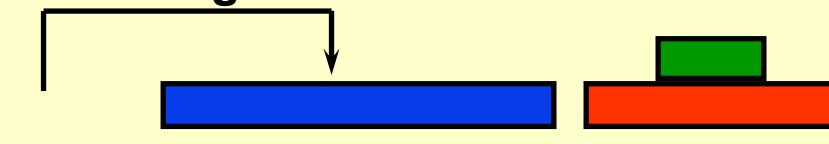
Der 2. Zug



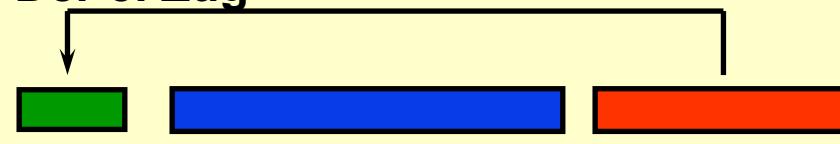
Der 3. Zug



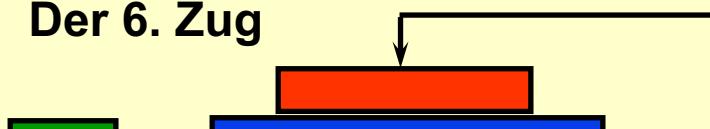
Der 4. Zug



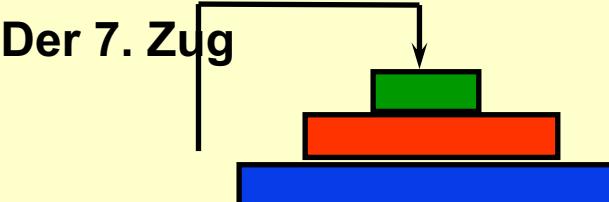
Der 5. Zug



Der 6. Zug



Der 7. Zug



fertig !

Rekursiver Algorithmus

```
/** Überträgt einen Stapel von n Scheiben von Platz „von“ auf Platz „nach“ und
 * benutzt
 * dabei den Platz „über“. n ist dabei die Anzahl der Scheiben
 */
void bewegeStapel (int n, int von, int nach, int ueber) {
    if (n == 1) // Stopbedingung für die Rekursion
        bewegeScheibe(von, nach);
    else {      // Zurückführung auf einfachere Probleme
        bewegeStapel(n - 1, von, ueber, nach);
        bewegeScheibe(von, nach);
        bewegeStapel(n - 1, ueber, nach, von);
    }
}
```

- Was ist die Ausgabe des Algorithmus
 - In der Methode bewegeScheibe wird der Zug mit System.out.println ausgegeben.
 - Geht dies nicht etwas besser?
 - Ja, aber die Antwort dazu kommt in einem späteren Kapitel.

Anzahl der Züge beim Hanoi-Spiel

- Wenn wir **eine** Scheibe haben, benötigen wir **einen** Zug.
- Wenn wir **zwei** Scheiben haben, benötigen wir **drei** Züge.
- **A(n)** sei die Anzahl der Züge, falls wir **n** Scheiben haben.
- Dann gilt offenbar:

- $A(1) = 1$
- $A(n+1) = A(n) + 1 + A(n) = 1 + 2 \cdot A(n)$

- Die resultierende Folge lautet:
 $1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots$
- Allgemein scheint zu gelten, dass

$$A(n) = 2^n - 1$$

Mehr dazu in
späteren Semestern.

- Dies kann induktiv bewiesen werden:

$$A(1) = 1 = 2^0$$

$$A(n+1) = 1 + 2 \cdot A(n) = 1 + 2 \cdot (2^n - 1) = 1 + 2^{n+1} - 2 = 2^{n+1} - 1$$

Zusammenfassung

- Entwurf von Algorithmen durch schrittweise Verfeinerung
 - Wiederverwendung
- Rekursion ist ein wichtiges Entwurfsprinzip
 - Lösungen werden auf ein Problem geringerer Komplexität des gleichen Typs zurückgeführt.
 - Arten von Rekursion
 - Linear und baumartig
 - Terminierung
 - Laufzeit
- Was kann alles mit Rekursion berechnet werden?
 - Genauso viel wie auf einer Turing-Maschine → Theoretische Informatik.



4. Datentypen

- Beschreibung von elementaren Datentypen
boolean, byte, short, int, long, float, double, char
- Auswertung von Ausdrücken
- Arrays

Bisher

- Algorithmus benötigt als Eingabe **Daten** und liefert als Ergebnis wieder **Daten** zurück.



- Der Algorithmus macht dabei gewisse Annahmen über die **Eingabedaten**.
 - Voraussetzung für die Korrektheit des Algorithmus (**Vorbedingung**)
- Der Algorithmus liefert als Ergebnis nur bestimmte Daten aus einer **Wertemenge** zurück. (**Nachbedingung**)
- Beispiel
 - ggt
 - Liefert zu zwei ganzen Zahlen größer Null wieder eine ganze Zahl größer Null
 - getPi
 - Liefert zu einer Gleitpunktzahl (Abbruchbedingung) eine Gleitpunktzahl zurück.

Datentypen

- **Datentypen**

- Repräsentation einer Menge von Datenelementen
- und die zugehörigen Operationen, die auf den Datenelementen ausgeführt werden können.

Datentyp ist eine Sort bzw. Menge zu dieser Menge gehören Elemente und auf diese Menge sind besondere Grundoperationen definiert die auf diesem Datentyp angewendet werden können aber auf andere Datentypen nicht

- **Beispiel**

- **Datentyp int**

- Repräsentation aller ganzen Zahlen in dem Intervall
-2.147.483.648 ... 2.147.483.647
- Operationen +, -, /, *, %

Datentyp ist eine Sorte bzw. Menge besteht aus :
1) Elemente
2) Grundoperationen

zu dem Datentyp int gehören diese Werte das sind Literale,
und nur diese Werte sind zulässig

Operationen auf diesen Datentyp

- Verwendung von Datentypen bei der Deklarationen von Variablen

double x; // Variable x ist vom Typ double

int a; // Variable a ist vom Typ int

Datentypen in Java

- Java bietet nur wenige vorgefertigte Datentypen an.
 - Boolesche Werte: boolean
 - Ganzzahlige Datentypen: byte, char, short, int, long Primitive Datentypen
 - Gleitpunktzahlen: float, double
- Zusätzlich wird noch ein Datentyp String unterstützt.
 - Zeichenketten: String Das ist kein primiver Datentyp, eigentlich Strings sind in Java Objekte
- Java bietet Techniken zur Konstruktion eigener Datentypen
 - Arrays
 - Aufzählungstypen
 - Klassen

Repräsentation von Information

- Viele Abstraktionsebenen

Interpretation



ISBN 978-3-486-70641-3

01001001 01010011 01000010 01001110
00100000 00111001 00110111 00111000
00101101 00110011 00101101 00110100
00111000 00110110 00101101 00110111
00110000 00110110 00110100 00110001
00101101 00110011

Repräsentation



Motivation

mögliche **Abstraktionshierarchie** in einem Informationssystem der Universität:

logische
Ebene



Vorlesungen	Termin hinzufügen, ...	Termine, Räume, Teilnehmer, ...
Kalenderdaten	Tage addieren, ...	Tag, Monat, Jahr, ...
ganze Zahlen	addieren, ...	-2.147.483.648, 0, 1, 1024
Bits	and, or, ...	00011001,00001100, 00000010, ...

technische
Ebene

Ein Rechner verfügt zunächst nur über wenige, primitive Operationen!

Motivation

mögliche Abstraktionshierarchie in einem Informationssystem der Universität:



Termin hinzufügen, ...

Tage addieren, ...

addieren, ...

and, or, ...

Termine, Räume,
Teilnehmer, ...

Tag, Monat, Jahr,
...

-2.147.483.648,
0, 1, 1024

00011001,00001100,
00000010, ...

Bemerkung:

- Der Benutzer einer Abstraktionsebene hat i.A. **keine Kenntnisse über die Repräsentation der Daten**, sondern kennt **nur die Bedeutung der Operationen!**

4.1 Beschreibung von Datentypen

- Ein **Datentyp** besteht aus einer **Grundmenge** (**Sorte**, **Trägermenge**) und einer Menge von **Operationen**. Jede der zugehörigen Operationen besitzt als Parameter oder als Ergebnis **zumindest einmal die Trägermenge**.
- Beziehen sich die Parameter und Ergebnisse der Operationen **nur** auf die **Trägermenge**, so sprechen wir von **einsortigen Datentypen**. Ansonsten von **mehrsortigen** oder **heterogenen Datentypen**.

Beispiel eines einsortigen Datentyps (**boolean**)

SORT	boolean	
OPS	true:	→ boolean
	false:	→ boolean
	not: boolean	→ boolean
	and: boolean × boolean	→ boolean
	or: boolean × boolean	→ boolean
	xor: boolean × boolean	→ boolean

„Signaturen“ der Operationen.
Definieren nur die Syntax nicht
die Semantik!

Begriffe

- Die **Stelligkeit** einer Operation bezeichnet die **Anzahl ihrer Parameter**.
- Die **Signatur** einer Operation besteht aus der Folge der zu den **Parametern** zugeordneten Datentypen und dem Datentyp des **Resultats**.
 - In Java ist bei einer Methode **der Name der Methode Teil der Signatur**, aber nicht der Ausgabetyp.
(Es können keine zwei Methoden, die sich nur im Rückgabetyp unterscheiden, existieren.)
- Operationen mit **Stelligkeit 0** wie z.B. true liefern nur einen Wert zurück; sie können somit als **Konstanten** aufgefasst werden.

Schreibweisen

- Es gibt verschiedene Schreibweisen für den Aufruf einer Operation f:

f	nullstellige Operation
f(a, b, ...)	Methodenaufruf
a f b	Infixform (nur für 2-stellige Operationen)
f a b ...	Präfixform
a b ... f	Postfixform
ā	Spezialnotationen

- Beispiele
 - true
 - fibo(5);
 - 2 + 3
 - 2 3 + 5* entspricht $(2+3)*5$
 - ! b Spezialnotation für die der Negation
- Bisher haben wir noch nicht geklärt, was die einzelnen Operationen leisten!

HP 12c: Postfix-Taschenrechner



Semantik der Operationen durch eine Funktionstabelle

- Angabe der Semantik der Operationen durch Funktionstabellen

x	not x
true	false
false	true

x	y	x and y	x or y	x xor y
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

- Bemerkungen

- Diese Herangehensweise erfordert eine **kleine Trägermenge!**
- Gesetzmäßigkeiten** wie z.B.

$$\text{not } (x \text{ and } y) == (\text{not } x) \text{ or } (\text{not } y)$$

sind anhand von Funktionstabellen nur **schwierig erkennbar**.

- Bei diesen komplexen Ausdrücke ist bereits die Reihenfolge der **Auswertung** wichtig zu wissen.
 - Durch Klammerung kann die Reihenfolge festgelegt werden.

Literale

- **Literale** sind (unmittelbar vom Programmierer eingebbare) Bezeichnungen für die **Werte** eines Datentyps.
- Es gibt folgende Arten von Literalen:
 - Die Literale des Datentyps *boolean*
 - `false` und `true`
 - *Ganzzahlige* Literale
 - z. B. `2`, `17`, `-3` und `32767`
 - Literale für *Gleitpunktzahlen*
 - z. B. `3.14`, `1E-6`
 - Literale für *Zeichen* und *Zeichenketten*
 - z. B. `'A'`, `"Hallo OOP"`

4.1.1 Der Datentyp boolean

- Boolesche Werte können in Variablen gespeichert werden.

```
boolean b = true;  
boolean test = false;
```

- Sie entstehen bei Vergleichen und können dann zugewiesen werden.

```
b = 3 < 7;  
test = x != y;
```

- Sie werden vor allem dort benutzt, wo **Bedingungen** ausgewertet werden müssen, z.B. in **if-Anweisungen**, **while-Schleifen** etc.:

```
if (var1 < var2) ...  
while (x != y) ...
```

```
if (b) ...  
while (test) ...
```

Boolesche Operationen in Java

- Für die Verknüpfung Boolescher Werte gibt es in Java verschiedene Operatoren:

logisches und (and):	& und &&
Logisches oder (or):	und
Exklusives oder (xor):	^
Logische Negation (not):	!

- Beispiele:

```
boolean b = true;
boolean test = false;
b = (var1 < var2) & !(x != y);
test = (x != y) || (b ^ test);
```

- & und | tragen die normale Bedeutung von **and** bzw. **or**. Sie werden „unbedingt“ ausgewertet, d.h.
 - Zuerst wird der linke und dann der rechte Operand ausgewertet.
 - Danach wird das Ergebnis laut Operationstabelle bestimmt.
- Dagegen werden && und || zur sog. verkürzten Auswertung verwendet.

Verkürzte Auswertung (1)

- Bei der Auswertung der binären Operatoren `&&` und `||` gilt ebenfalls, dass zunächst der linke Operand ausgewertet wird.
 - Was können wir bereits aussagen, wenn das Ergebnis des ersten Operanden `true` bzw. `false` ist?
- Für Boolesche Werte gilt immer (d.h. für alle $x \in \{\text{true}, \text{false}\}$):

```
true or x ergibt true  
false and x ergibt false
```

- Dies eröffnet die Möglichkeit, Boolesche Ausdrücke mit den Operatoren `&&` und `||` besonders effizient auszuwerten.

Verkürzte Auswertung (2)

- Auswertungsregel für **b1 || b2**:
 - Falls **b1** den Wert **true** liefert, muss **b2** nicht mehr ausgewertet werden, denn das Gesamtergebnis **true** steht bereits fest. Es gilt also:
 - **b = (b1 || b2)** \Leftrightarrow **if (b1) b = true; else b = b2;**
- Analog für **b1 && b2**:
 - Falls **b1** den Wert **false** liefert, ist das Ergebnis **false**:
 - **b = (b1 && b2)** \Leftrightarrow **if (!b1) b = false; else b = b2;**
- Damit ist auch
 - if ((x != 0) && (y/x > 4)) . . .**„sicher“, da eine Division durch 0 vermieden wird.

Prioritäten

- Operatoren besitzen **Prioritäten**, welche die **Reihenfolge der Auswertung** bestimmen.
 - Aus der Schule bereits bekannt: **Punkt- vor Strichrechnung**.
 - Boolesche Operationen besitzen auch unterschiedliche Prioritäten

niedrige Priorität	Zuweisungsoperator:	=
	logisches Oder (verkürzt)	
	logisches Und (verkürzt)	&&
	logisches Oder	
	logisches Und	&
	Vergleichsoperatoren	<, <=, ==, !=, =>, >
hohe Priorität	logische Negation	!

- Die Reihenfolge der Auswertung kann durch **Klammerung** verändert werden.

```
boolean x = a < b && (c < d || e < f);
```

4.1.2 Der ganzzahlige Datentyp (int)

SORT int

OPS $+, -, *, /, \%:$ int \times int \rightarrow int
 $=:$ int \rightarrow int

Weiter Zuweisungsoperatoren:
 $+=$ addiert den Ergebniswert zur Variablen (z.B. $a += 1$). Genauso z.B.: $-=$, $*=$, $/=$.

- Sorte besteht aus einer Teilmenge von { ... -3, -2, -1, 0 , 1, 2, 3, ... },
- Für die aufgeführten Operatoren gelten die **üblichen Rechengesetze**.
 - Der Operator / bezeichnet die **ganzzahlige Division** (Rest vernachlässigt).
 - Der Operator % liefert den **Rest der ganzzahligen Division**. Somit bleibt die folgende Gleichung erhalten:
$$x == y * (x/y) + (x \% y)$$
- Priorität: Punkt-vor-Strich-Regel
- Weitere Operatoren auf den ganzen Zahlen sind die **üblichen Vergleichsoperatoren**

Ganze Zahlen in Java

- Der Bereich von int ist **endlich!**
 - Die ganzen Zahlen liegen zwischen **- 2^{31} und $2^{31}-1$** .
 - Neben int können ganze Zahlen in Java durch die Datentypen **byte**, **short** und **long** repräsentiert werden.
 - Wertebereich von byte: **- 2^7 und 2^7-1**
 - Wertebereich von short: **- 2^{15} und $2^{15}-1$**
 - Wertebereich von long: **- 2^{63} und $2^{63}-1$**
- Warum benötigt man diese verschiedenen Typen?**
- Was passiert bei einem **Unterlauf** bzw. **Überlauf**, d.h., wenn das Ergebnis außerhalb des zulässigen Bereichs fällt?

Darstellung von ganzen positiven Zahlen

- Binärdarstellung:

$$b_{m-1}b_{m-2}\dots b_3b_2b_1b_0$$

- Wert der Binärzahl:

$$\sum_{i=0}^{m-1} 2^i \cdot b_i = 2^0 b_0 + 2^1 b_1 + 2^2 b_2 + \dots + 2^{m-1} b_{m-1}$$

Basis der Zahlendarstellung

- Beispiel (m=16):

$$\begin{aligned} & 0110\ 1001\ 0011\ 1101_{(2)} \\ & 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 \\ & + 1 \cdot 16 + 1 \cdot 32 + 0 \cdot 64 + 0 \cdot 128 \\ & + 1 \cdot 256 + 0 \cdot 512 + 0 \cdot 1024 + 1 \cdot 2048 \\ & + 0 \cdot 4096 + 1 \cdot 8192 + 1 \cdot 16384 + 0 \cdot 32768 = 26941_{(10)} \end{aligned}$$

Umrechnen: Dezimal → Binär

- Hornerschema:

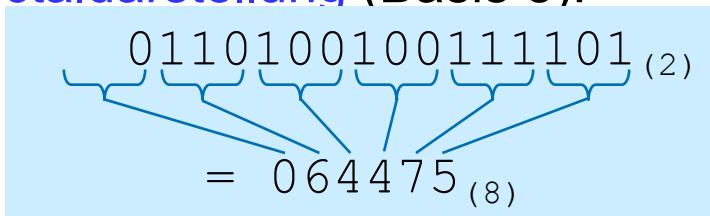
26972	/	2	=	13486	Rest	0
13486	/	2	=	6743	Rest	0
6743	/	2	=	3371	Rest	1
3371	/	2	=	1685	Rest	1
1685	/	2	=	842	Rest	1
842	/	2	=	421	Rest	0
421	/	2	=	210	Rest	1
210	/	2	=	105	Rest	0
105	/	2	=	52	Rest	1
52	/	2	=	26	Rest	0
26	/	2	=	13	Rest	0
13	/	2	=	6	Rest	1
6	/	2	=	3	Rest	0
3	/	2	=	1	Rest	1
1	/	2	=	0	Rest	1

mit Nullen aufgefüllt bis 16 Bit,
wegen 16 Bit Worten

0110 1001 0101 1100

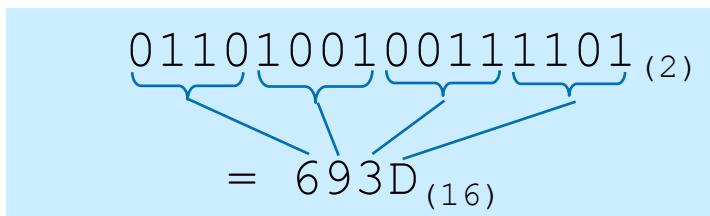
Schreibweisen für Bitfolgen

- Binärdarstellung: 0110 1001 0011 1101
 - Zahl wird als Zahl zur Basis 2 geschrieben
- Octaldarstellung (Basis 8):



000 ₍₂₎	=	0 ₍₈₎	100 ₍₂₎	=	4 ₍₈₎
001 ₍₂₎	=	1 ₍₈₎	101 ₍₂₎	=	5 ₍₈₎
010 ₍₂₎	=	2 ₍₈₎	110 ₍₂₎	=	6 ₍₈₎
011 ₍₂₎	=	3 ₍₈₎	111 ₍₂₎	=	7 ₍₈₎

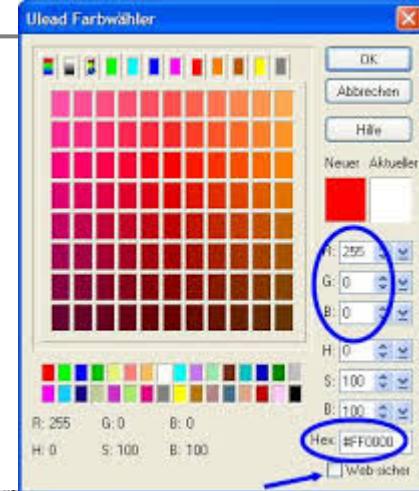
- Hexadezimaldarstellung (Basis 16):



0000 ₍₂₎	=	0 ₍₁₆₎	1000 ₍₂₎	=	8 ₍₁₆₎
0001 ₍₂₎	=	1 ₍₁₆₎	1001 ₍₂₎	=	9 ₍₁₆₎
0010 ₍₂₎	=	2 ₍₁₆₎	1010 ₍₂₎	=	A ₍₁₆₎
0011 ₍₂₎	=	3 ₍₁₆₎	1011 ₍₂₎	=	B ₍₁₆₎
0100 ₍₂₎	=	4 ₍₁₆₎	1100 ₍₂₎	=	C ₍₁₆₎
0101 ₍₂₎	=	5 ₍₁₆₎	1101 ₍₂₎	=	D ₍₁₆₎
0110 ₍₂₎	=	6 ₍₁₆₎	1110 ₍₂₎	=	E ₍₁₆₎
0111 ₍₂₎	=	7 ₍₁₆₎	1111 ₍₂₎	=	F ₍₁₆₎

Beispiele

- Das RGB-Farbformat beschreibt die Mischung von drei Farben, deren Intensität zwischen 0 und 255 liegen.
 - 3 **byte** werden zur Darstellung einer Farbmischung verwendet.
- Die Anzahl der Kursteilnehmer in "Objektorientierter Programmierung" kann nicht durch 1 Byte dargestellt werden.
 - Wir benötigen hierfür einen Wert vom Typ **short**.
- 71137 ist die Anzahl der Plätze in der Allianz Arena.
 - Wir benötigen einen Wert vom Typ **int**.
- Weltbevölkerung: 7 750 008 172
 - Der Wert muss durch den Typ **long** repräsentiert werden.



Rechnen mit Binärzahlen

Rechnen mit
Binärzahlen funktioniert
genauso wie mit
Dezimalzahlen:

$$\begin{array}{r}
 0110\ 1001\ 0101\ 1100_{(2)} = 26972_{(10)} \\
 + 0011\ 1010\ 1110\ 0101_{(2)} = 15077_{(10)} \\
 \hline
 1010\ 0100\ 0100\ 0001_{(2)} = 42049_{(10)}
 \end{array}$$

$$\begin{array}{r}
 0110\ 1001\ 0101\ 1100_{(2)} = 26972_{(10)} \\
 - 0011\ 1010\ 1110\ 0101_{(2)} = 15077_{(10)} \\
 \hline
 0010\ 1110\ 0111\ 0111_{(2)} = 11895_{(10)}
 \end{array}$$

Problem: mit einer
beschränkten Anzahl
von Bits kann man
nicht beliebig große
Zahlen darstellen:

$$\begin{array}{r}
 0110\ 1001\ 0101\ 1100_{(2)} = 26972_{(10)} \\
 + 1011\ 1010\ 1110\ 0101_{(2)} = 47845_{(10)} \\
 \hline
 10010\ 0100\ 0100\ 0001_{(2)} = 74817_{(10)}
 \end{array}$$

Überlauf!

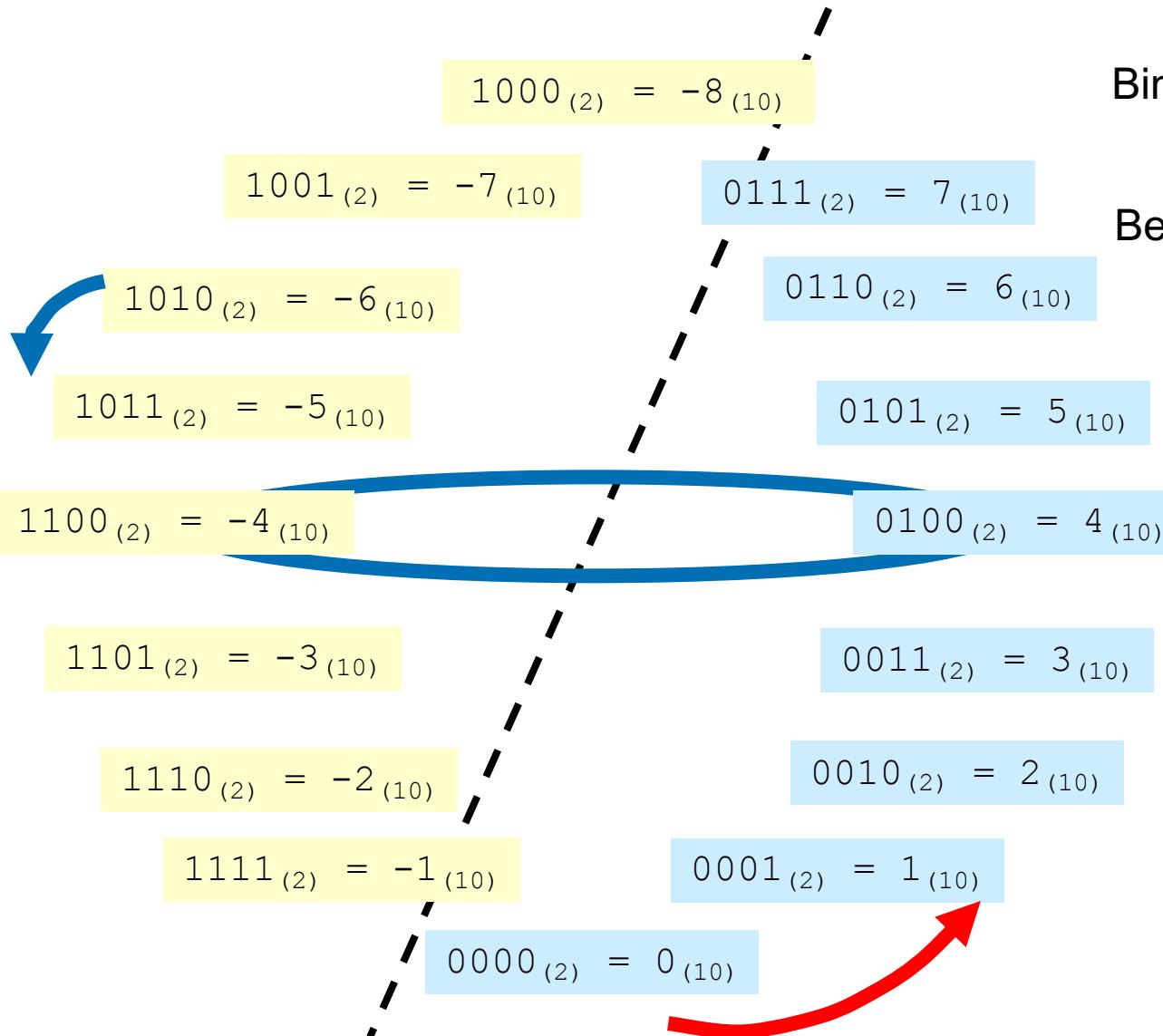
Begrenzung der darstellbaren Zahlen

- Mit m Bits lassen sich die Zahlen von 0 bis $2^m - 1$ darstellen.

m	$2^m - 1$
4	15
8	255
16	65 535
32	4 294 967 296
64	18 446 744 073 709 551 616

- Was ist eigentlich mit negativen Zahlen?**

Darstellung ganzer Zahlen als Zweierkomplement



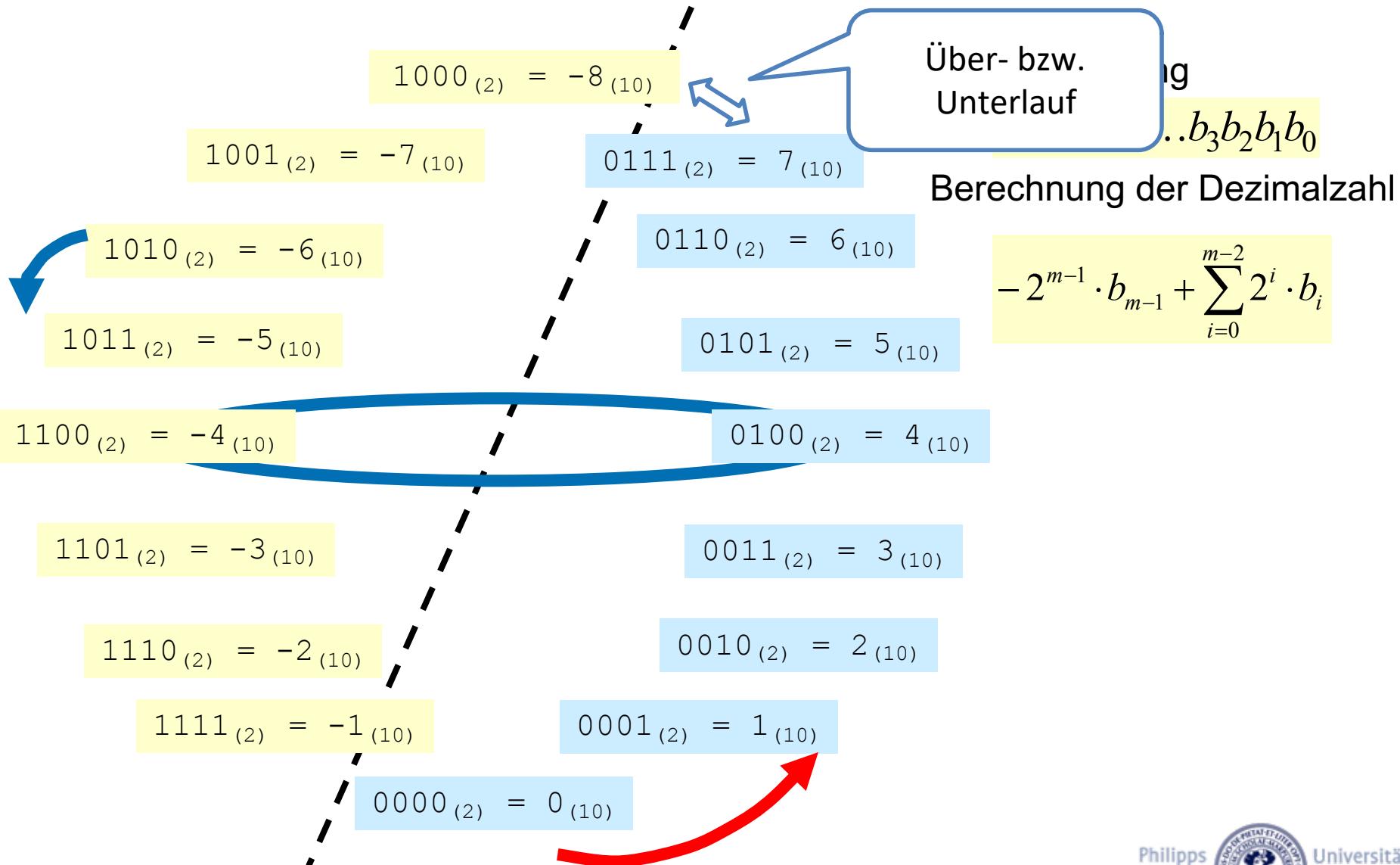
Binärdarstellung

$$b_{m-1}b_{m-2}\dots b_3b_2b_1b_0$$

Berechnung der Dezimalzahl

$$-2^{m-1} \cdot b_{m-1} + \sum_{i=0}^{m-2} 2^i \cdot b_i$$

Darstellung ganzer Zahlen als Zweierkomplement



Vorzeichenwechsel & Rechnen

- Vorzeichenwechsel durch:
 1. Komplementieren
(0 und 1 austauschen)
 2. Addieren von 1

Beispiel: (4-Bit)

$$5 = \begin{array}{r} 0101B \\ \hline \end{array}$$

Zum Vorzeichenwechsel:

Komplementieren $\begin{array}{r} 1010B \\ \hline \end{array}$

Addieren von 1 $\begin{array}{r} 0001B \\ \hline \end{array}$

$$\begin{array}{r} -5 = 1011B \\ \hline \end{array}$$

Zum Vorzeichenwechsel:

Komplementieren $\begin{array}{r} 0100B \\ \hline \end{array}$

Addieren von 1 $\begin{array}{r} 0001B \\ \hline \end{array}$

$$\begin{array}{r} 5 = 0101B \\ \hline \end{array}$$

- Addition und Subtraktion arbeiten „normal“:

$$\begin{array}{r} 5 \quad \quad \quad 0101B \\ + -7 \quad \quad \quad 1001B \\ \hline -2 \quad \quad \quad 1110B \\ \hline 3 \quad \quad \quad 0011B \\ - 4 \quad \quad \quad 0100B \\ \hline -1 \quad \quad \quad 1111B \\ \hline \end{array}$$

Vorzeichenwechsel & Rechnen

- Vorzeichenwechsel durch:

1. Komplement "B" steht für Binärzahl.
(0 und 1) Hat dieselbe Bedeutung
2. Addieren von 1 wir Index $\binom{2}{2}$.

Beispiel: (4-Bit,

$$5 = \begin{array}{r} 0101 \\ \hline \end{array} B$$

Zum Vorzeichenwechsel:

Komplementieren $1010B$

Addieren von 1 $0001B$

\hline

$$-5 = \begin{array}{r} 1011 \\ \hline \end{array} B$$

Zum Vorzeichenwechsel:

Komplementieren $0100B$

Addieren von 1 $0001B$

\hline

$$5 = \begin{array}{r} 0101 \\ \hline \end{array} B$$

- Addition und Subtraktion arbeiten „normal“:

$$\begin{array}{r} 5 \\ + -7 \\ \hline \end{array} \quad \begin{array}{r} 0101B \\ 1001B \\ \hline \end{array}$$

$$\begin{array}{r} -2 \\ \hline \end{array} \quad \begin{array}{r} 1110B \\ \hline \end{array}$$

$$\begin{array}{r} 3 \\ - 4 \\ \hline \end{array} \quad \begin{array}{r} 0011B \\ 0100B \\ \hline \end{array}$$

$$\begin{array}{r} -1 \\ \hline \end{array} \quad \begin{array}{r} 1111B \\ \hline \end{array}$$

Vorzeichenwechsel & Rechnen

- **Problem:** das Ergebnis kann eine nicht darstellbare Zahl sein!
 - Über- oder Unterlauf
 - Gilt aber für alle Zahlendarstellungen mit beschränktem Platz

5 0101B
+ 11 1011B

0 10000B

Wird
abgeschnitten.

Weitere Operationen auf Integer-Datentypen

- Inkrement/Dekrement-Operationen

`++`, `--`

- Häufig werden Variablen ganzer Zahlen um 1 erhöht bzw. erniedrigt.

- Durch

`++i;`

bzw.

`i++;`

wird der Wert von `i` um 1 erhöht. Dies hat die gleiche Wirkung wie
`i += 1` oder `i = i + 1`;

- Durch

`--i;`

bzw.

`i--;`

wird der Wert von `i` um 1 erniedrigt. Dies hat die gleiche Wirkung wie
`i -= 1` oder `i = i - 1`;

- **Wir werden später noch auf Besonderheiten dieser Operatoren eingehen.**

Bit-Operatoren

- Diese Operatoren interpretieren eine ganze Zahl als Folge von Bits.
- Operatoren: `&`, `|`, `^`, `~`, `<<`, `>>`, `>>>`
 - Die Bit-Operationen `&`, `|`, `^` und `~` (entspricht **and**, **or**, **xor** und **Komplement**) verknüpfen ganzzahlige Werte bitweise - ohne Rücksicht auf Zahlenwerte.
 - Die Shift-Operationen `<<`, `>>` und `>>>` schieben ganzzahlige Werte bitweise.
 - `<<` Schiebe nach links und rechts wird mit Nullen aufgefüllt.
 - `>>` Schiebe nach rechts und links wird das Vorzeichenbit erhalten.
 - `>>>` Schiebe nach rechts und links wird mit Nullen aufgefüllt.

Multiplikation und Division mit 2

- Eine Multiplikation einer Zahl mit 2^i kann durch eine Shift-Operation erfolgen.

```
int x = 42;                      // Operand der Multiplikation
int i = 3;                        // Exponent
System.out.println(x << i);      // Liefert x * 2i
```

- Entsprechend kann eine ganzzahlige Division durchgeführt werden.

```
int x = -42;                     // Divisor
int i = 3;                       // Dividend 2i
System.out.println(x >> i);     // Shift rechts und Auffüllen mit
                                // Vorzeichenbit.
```

Ganzzahlige Literale

- Einfache **ganzzahlige Literale**:
 - 2 17 -3 32767 -889275714
- Für ganzzahlige Literale gibt es neben der Standardschreibweise auch noch eine weitere:
 - Die **binäre** Notation beginnt mit den Zeichen Null: **0b**
 - 0b101 entspricht $5_{(10)}$ ($=1 \cdot 4 + 0 \cdot 2 + 1$)
 - Die **oktale** Notation beginnt mit einer Null: **0**
 - 0747 entspricht $487_{(10)}$ ($=7 \cdot 64 + 4 \cdot 8 + 7$)
 - Die **hexadezimale** Notation einer Zahl beginnt mit den Zeichen **0x**. Danach können normale Ziffern oder hexadezimale Ziffern (**A ... F**) folgen. Diese können groß oder klein geschrieben werden.
 - $889275714_{(10)}$ entspricht 0xCafeBabe und 0xCAFEBAE
- Ganzzahlige Literale bezeichnen zunächst Werte des Datentyps **int**.
- Will man sie als **long** kennzeichnen, muss man **L** (oder **I**) anhängen.
 - 4242424242L oder 0xC0B0L

Live Vote

PIN: P5BK

x

<https://ilias.uni-marburg.de/vote/P5BK>



4.1.3 Der Datentyp float

- Repräsentation von Fließpunktzahlen auf dem Rechner.
 - Oft werden diese Zahlen zur Repräsentation von den reellen Zahlen aus der Mathematik benutzt.

SORT float

OPS

Divisionsrest auch auf float anwendbar!

$+ , - , * , / , \% : \text{float} \times \text{float} \rightarrow \text{float}$

$= , += , == , *= , /= : \text{float} \rightarrow \text{float}$

- Für die Operatoren gelten die **üblichen Rechengesetze** aus der Mathematik für reelle Zahlen.

Besonderheiten auf Rechner

- Wie sieht die Überlaufbehandlung aus?
- Zusätzlich muss mit Rundungsfehlern gerechnet werden! Auf dem Rechner (auch in Java) gilt z.B. nicht mehr das Assoziativgesetz. Somit i.A.
$$(x + y) + z \neq x + (y + z)$$
- Neben float (32 Bits) können reelle Zahlen auch durch double (64 Bits) repräsentiert werden. Operationen sind für double entsprechend definiert.
- Wie kann ich einen Wert vom Typ int in einen Wert vom Typ float konvertieren?

Kontrollrechnung

- Ungenauigkeit durch Längenbeschränkung
 - z. B. kann $1/10$ nicht auf einem Computer genau dargestellt werden.
 - Weiteres Problem: Fehler kumulieren in Berechnungen!
-
- Kleinste darstellbare positive Zahl: $\approx 1,4 * 10^{-45}$
 - Größte darstellbare positive Zahl: $\approx 3,4 * 10^{38}$
 - Der Datentyp **double** mit 64 Bits arbeitet zwar genauer, besitzt aber die gleichen Probleme wie der Datentyp **float**.

Kontrollrechnung

- Ungenauigkeit durch Längenbeschränkung
 - z. B. kann $1/10$ nicht auf einem Computer genau dargestellt werden.
- Weiteres Problem: Fehler kumulieren in Berechnungen!

Achtung: bei der Ausgabe von Floats rundet Java automatisch. So kann das Runden umgangen werden:

- Kleiner Fehler:
`System.out.println(String.format("%.20f", 1.0f / 10.0f));`

- Größte darstellbare positive Zahl: $\approx 3.4 \cdot 10^{38}$

Float in formatierten Text konvertieren. Verwenden von 20 Nachkommastellen.

- Der Datentyp **double** mit 64 Bits arbeitet mit den gleichen Problemen wie der Datentyp **float**.

Beispiel (Berechnung der Zahl e):

Formel zur Berechnung der Zahl e:

$$e = \sum_{i \geq 0} \frac{1}{i!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

```
/** Diese Methode berechnet die Zahl e.  
 * @result Liefert einen approximativen Wert von e  
 */  
float eulerLeftToRight () {  
    float sum = 1.0f;  
    int i = 1;  
    while (i <= 15) {  
        sum = sum + 1.0f / fakultaet(i);  
        i = i+1;  
    }  
    return sum;  
}  
...
```

Beispiel (Berechnung der Zahl e):

Formel zur Berechnung der Zahl e:

$$e = \sum_{i \geq 0} \frac{1}{i!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

```
/** Diese Methode berechnet die Zahl e.  
 * @result Liefert einen approximativen Wert von e  
 */  
float eulerRightToLeft () {  
    float sum = 0.0f;  
    int i= 15;  
    while (i >= 1) {  
        sum = sum + 1.0f / fakultaet(i);  
        i = i-1;  
    }  
    return sum + 1.0f;  
}  
...
```

```
...
/** Diese Methode berechnet die Fakultät.
 * @param Eingabeparameter für die Berechnung
 * @return Liefert als Ergebnis n!
 */
long fakultaet(int n) {
    long res = 1;
    while (n >= 1) {
        res = res*n;
        n = n - 1;
    }
    return res;
}
```

- Als Ergebnisse der Methoden `eulerLeftToRight` und `eulerRightToLeft` bekommen wir folgende Ausgabe:

2.718282

2.7182817

Differenz: 2.3841858E-7

Da stimmt doch
irgendetwas nicht?

Binärdarstellung von Fließpunktzahlen

- Darstellung von Fließpunktzahlen durch $\pm \text{Mantisse} \cdot 2^{\text{Exponent}}$
- Es lassen sich nur Zahlen exakt darstellen, die Brüche mit einer Zweierpotenz im Nenner sind

IEEE 754 Converter (JavaScript), V0.21

	Sign	Exponent	Mantissa
Value:	-1	2^{-1}	1.0
Encoded as:	1	126	0
Binary:	<input checked="" type="checkbox"/>	<input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>	<input type="checkbox"/>
Decimal representation	-0.5		
Value actually stored in float:	-0.5		
Error due to conversion:	<input type="checkbox"/> +1 <input type="checkbox"/> -1		
Binary Representation	101111100000000000000000000000000000000000000000000000000000000		
Hexadecimal Representation	0xbfffff		

<https://www.h-schmidt.net/FloatConverter/IEEE754de.html>

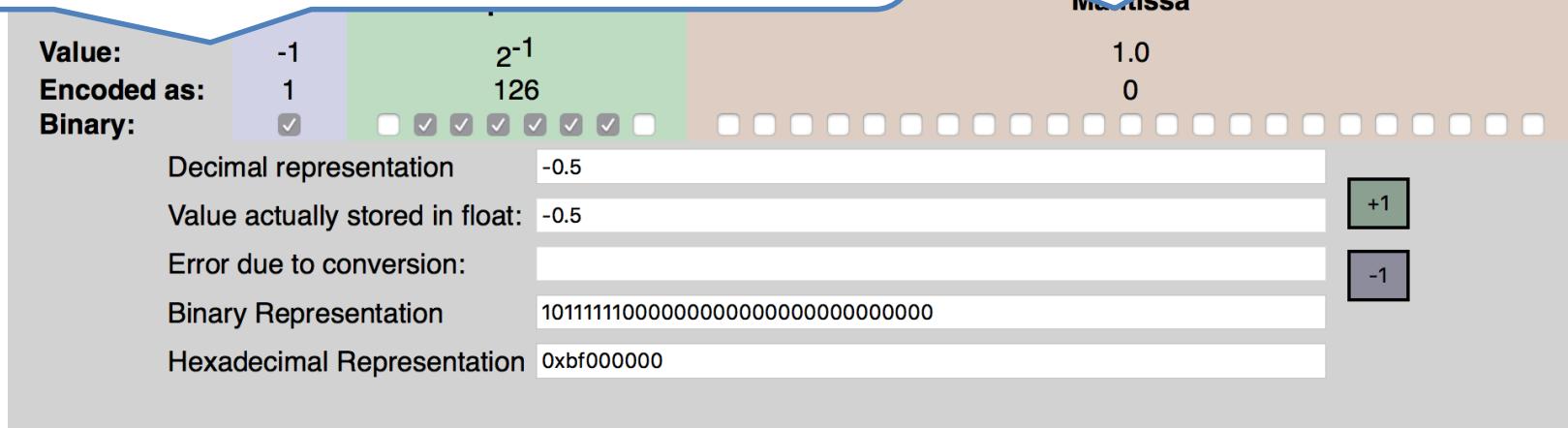
Binärdarstellung von Fließpunktzahlen

- Darstellung von Fließpunktzahlen durch $\pm \text{Mantisse} \cdot 2^{\text{Exponent}}$
- Es lassen sich nur Zahlen exakt darstellen, die Brüche mit

Kein 2er-Komplement. Stattdessen:

Exponent = (binär codierte positive Ganzzahl) - 127

Mantissen-Bitmuster (M)
steht für: 1,M
 $1 \leq \text{Mantisse} < 2$



<https://www.h-schmidt.net/FloatConverter/IEEE754de.html>

Binärdarstellung von Fließpunktzahlen

- Darstellung von Fließpunktzahlen durch $\pm \text{Mantisse} \cdot 2^{\text{Exponent}}$
- Es lassen sich nur Zahlen exakt darstellen, die Brüche mit einer Zweierpotenz im Nenner sind

IEEE 754 Converter (JavaScript), V0.21

	Sign	Exponent	Mantissa
Value:	+1	2^{-4}	1.600000023841858
Encoded as:	0	123	5033165
Binary:	<input type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/>
You entered	0.1		
Value actually stored in float:	0.100000001490116119384765625		
Error due to conversion:	1.490116119384765625E-9		
Binary Representation	001110110011001100110011001101		
Hexadecimal Representation	0x3dcccccd		

<https://www.h-schmidt.net/FloatConverter/IEEE754de.html>

Weitere Operationen für float

- Auf den Datentypen für Fließpunktzahlen sind die üblichen arithmetischen Operationen definiert:

- Inkrement/Dekrement-Operationen**

`++`, `--`

rechnen einfach +1 und -1

- Vergleichs-Operationen**

`==`, `!=`, `<`, `<=`, `>`, `>=`

problematisch wegen Rechengenauigkeit!

Literale für Fließpunktzahlen

- Zur Erinnerung: in Java sind **float** (floating point number, 32 Bit) und **double** (double precision number, 64 Bit).
 - Fließpunkt-Literale bezeichnen normalerweise Werte des Datentyps **double**.
 - Durch Anhängen eines der Suffixe **F** oder **D** (bzw. **f** oder **d**) spezifiziert man sie explizit als Werte des Datentyps **float** bzw. **double**.
 - Beispiele für Fließpunkt-Literale des Datentyps **float** sind :

1e1f 2.f .3f 0f 1.0f 2.0f 3.14f 6.022137e+23f

Exponentendarstellung

- Beispiele für Fließpunkt-Literale des Datentyps **double**:

1e1 2. .3 0.0 2.0 42.42 3.14 1e-9d 1e137

Live Vote

PIN: 6DJ7

x

<https://ilias.uni-marburg.de/vote/6DJ7>



4.2 Ausdrücke

- Ein Ausdruck setzt sich zusammen aus **Konstanten**, **Variablen** und **Operationen/Operationsaufrufen**.
 - Durch **Auswertung eines Ausdrucks** wird ein **Wert** berechnet.
 - Jeder Ausdruck ist einem **Datentyp** zugeordnet.
- Beispiel: $\text{sum} + 1.0f / \text{fakultaet}(i)$

Formale Definition (rekursiv):

- Ein **Ausdruck A vom Typ D entspricht genau einem der folgenden drei Fälle:**
 - i. A ist eine **Konstante** vom Typ D,
 - ii. A ist eine **Variable** vom Typ D,
 - iii. A ist eine **Operation** $f(t_1, \dots, t_m)$ mit $f: D_1 \times \dots \times D_m \rightarrow D$ und t_1, t_2, \dots, t_m sind Ausdrücke der Typen D_1, \dots, D_m .

Auswertung von Ausdrücken

- Ein Ausdruck lässt sich folgendermaßen **rekursiv** berechnen:
 - i. Falls der Ausdruck aus genau einer **Konstanten** vom Typ D besteht, so entspricht der Wert des Ausdrucks dem der Konstanten zugeordneten Werts des Datentyps D.
 - ii. Falls der Ausdruck aus genau einer **Variablen** besteht, so ist der Wert des Ausdrucks gleich dem aktuellen Wert der Variablen.
 - iii. Falls der Ausdruck aus einer **Operation** $f(t_1, \dots, t_m)$ besteht, so wird **zunächst** der Wert w_i des Ausdrucks t_i berechnet, $i = 1, \dots, m$. Der Wert des Ausdrucks $f(t_1, \dots, t_m)$ ergibt sich dann aus $f(w_1, \dots, w_m)$.
- Auswertungsreihenfolge:
 - Bei der Auswertung von Ausdrücken bestimmt die **Priorität der einzelnen Operatoren** die Auswertungsreihenfolge.
 - bekannte Regel: **Punkt- vor Strichrechnung**
 - Auswertungsreihenfolge kann durch Setzen von **Klammern** geändert werden.

Auswahl von Operatoren

(nach Prioritäten absteigend sortiert)

Priorität aufsteigend ↑

- Methodenaufruf
- Postfix-Operatoren
- unäre Operatoren
- Typumwandlung
- Multiplikationsoperatoren
- Additionsoperatoren
- Vergleichsoperatoren
- Gleichheitsoperatoren
- Bitoperatoren

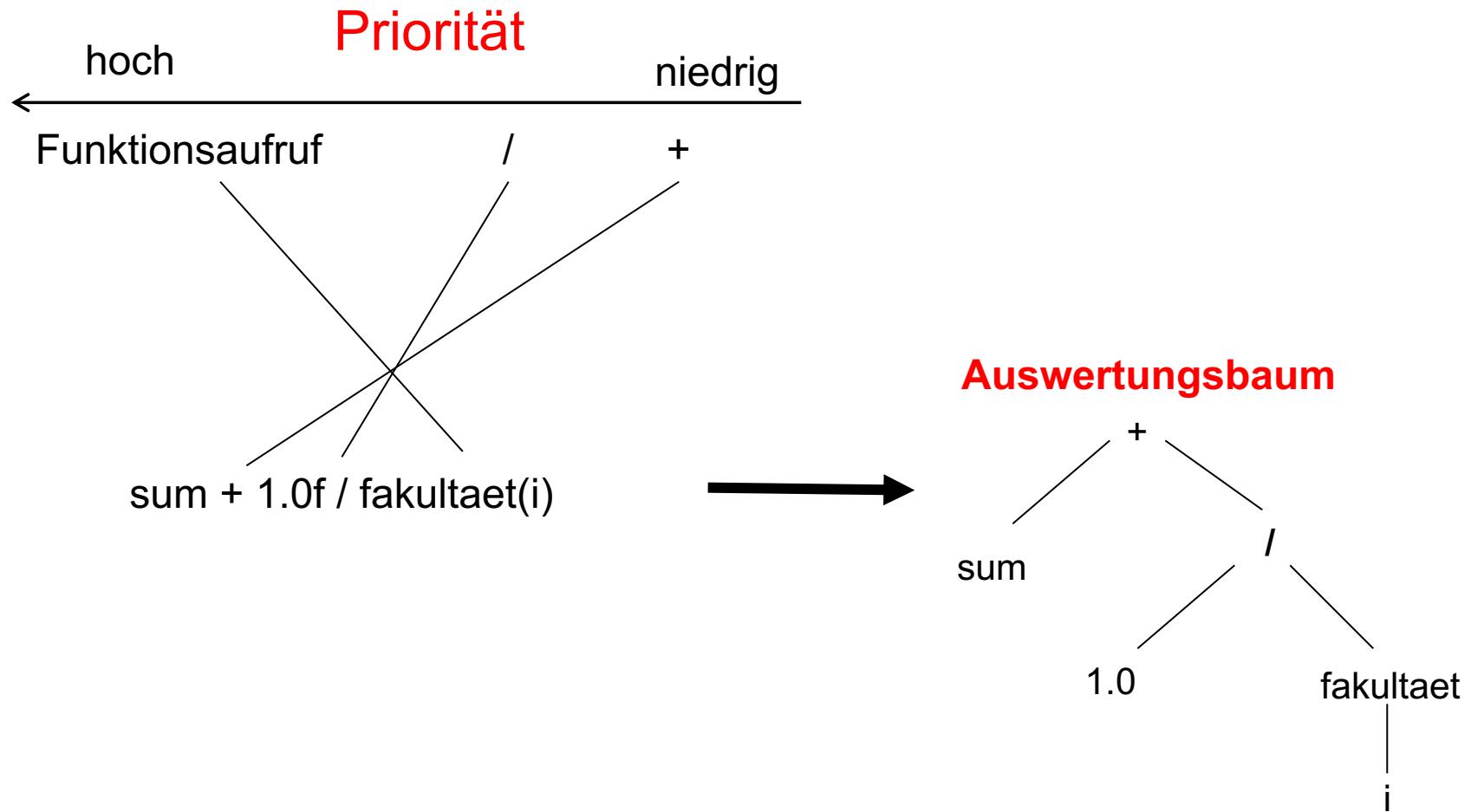
- verkürztes logisches Und
- verkürztes logisches Oder
- bedingter Zuweisungsoperator
- Zuweisungsoperatoren

$f(\text{Parameterliste})$
 $\text{var}++ \text{ var}--$
 $++\text{var} \text{ --var} \text{ +expr} \text{ -expr} \text{ !expr} \text{ ~expr}$
 $(\text{Typ}) \text{ expr}$
 $\text{expr}^* \text{expr} \text{ expr/expr} \text{ expr \%expr}$
 $\text{expr}+ \text{expr} \text{ expr-expr}$
 $\text{expr}< \text{expr} \text{ expr}> \text{expr} \text{ expr}>= \text{expr} \text{ expr}<= \text{expr}$
 $\text{expr}== \text{expr} \text{ expr}!= \text{expr}$
 $\text{expr} \& \text{expr}$
 $\text{expr} \wedge \text{expr}$
 $\text{expr} \mid \text{expr}$
 $\text{expr} \&& \text{expr}$
 $\text{expr} \parallel \text{expr}$
 $\text{expr} ? \text{expr} : \text{expr}$
 $\text{var}=\text{expr}, \text{var}+=\text{expr}, \text{var}^*=\text{expr}, \text{var}/=\text{expr}$

var steht für Variable

expr steht für Ausdruck

Beispiel



Seiteneffekte in Java

- Bisher haben wir bei der Auswertung eines Ausdrucks angenommen, dass die **Reihenfolge der Auswertung der Operanden** keinen Einfluss auf den Wert hat.
- Bei einer imperativen Programmiersprache kann der Wert aber von der Reihenfolge der Auswertung abhängen (**Seiteneffekt**).
- In Java ist daher festgelegt, dass die Auswertung der Ausdrücke $f(t_1, \dots t_m)$ bzw. $(s \text{ op } t)$ **von links nach rechts** erfolgt
 - d.h., t_i wird vor t_{i+1} für $i = 1, \dots, m-1$ bzw. s vor t ausgewertet

Beispiel:

- Der Ausdruck $((y \neq 0) \&\& (x/y > 1))$ ist eine **Kurzform** für die Schreibweise

```
if (y == 0) return false; else return x/y > 1;
```
- Seiteneffekte können z.B. auftreten, wenn **lesend und schreibend** auf eine gemeinsame **Variable** zugegriffen wird.

Ausdrücke vs. Anweisungen

In Java wird zwischen **Ausdrücken** und **Anweisungen** nicht sauber unterschieden.

$v = A$

ist keine Anweisung, sondern ein **Zuweisungsausdruck**. D.h. er hat einen Wert **und** bewirkt einen **Seiteneffekt**. Die Auswertung eines solchen Ausdrucks erfolgt (im Gegensatz zum sonstigen Vorgehen) **von rechts nach links**.

Wenn v eine Variable und A ein Ausdruck ist, dann bewirkt die **Auswertung** von

- $v = A$
1. **Ergebnis** des Zuweisungsausdrucks ergibt sich aus dem Ergebnis von A .
 2. Das **Ergebnis** wird v zugewiesen.

Dies lässt sich zur folgenden **Mehrfach-Zuweisung** verallgemeinern:

$vn = \dots = v1 = A$

Dieser Ausdruck wird **rechts-assoziativ** ausgewertet, d.h. alle Variablen $v1$,
 \dots , vn bekommen den Wert von A .

1. A wird ausgewertet
2. Das Ergebnis wird $v1$ zugewiesen.
- ...
- (n+1). Das Ergebnis wird vn zugewiesen.

Weitere Zuweisungsoperatoren

Eine **verkürzte Schreibweise** für bestimmte Zuweisungsausdrücke lautet

v op= A und steht für **v = v op A**,

wobei **op** ein (arithmetischer oder Schiebe-) Operator ist.

Für die besonders häufigen Zuweisungen

v += 1
v -= 1

bzw.

v = v + 1
v = v - 1

gibt es die weitergehenden Abkürzungen

v++
v--

und

++v
--v

Diese sog. „**Autoinkrement-Operatoren**“ sind ebenfalls **Ausdrücke**, es gibt sie in **Präfix**- und in **Postfix**-Form (mit unterschiedlicher Bedeutung):

++v // erhöht v um 1 und liefert den erhöhten Wert

v++ // erhöht v um 1 und liefert den ursprünglichen Wert

Seiteneffekt und Inkrementierung

- Wir betrachten die Methode printInts mit zwei Parametern.

```
void printInts(int n, int m){  
    System.out.println(n + " " + m);  
}
```

- Was passiert beim Aufruf von printInts?

```
int i = 42;  
  
printInts(i++, i);
```

```
int i = 42;  
  
printInts(++i, i);
```

```
int i = 42;  
  
printInts(i, i++);
```

Live Vote

Empfehlungen

- Seiteneffekte sollten möglichst vermieden werden.
- Mehrere Inkrement- und Dekrementoperationen sollten in einem Ausdruck generell vermieden werden.
 - Verwenden Sie stattdessen den Zuweisungsoperator `+=` (z. B. `i += 1;`), um Seiteneffekte zu vermeiden und die Lesbarkeit zu verbessern.
- Wenn mehrere Inkrement- und Dekrementoperationen in einem Ausdruck verwendet werden, dann sollten diese sich auf unterschiedliche Variablen beziehen.

Typumwandlung

- Manchmal ist es vorteilhaft, die **strengen Typkonventionen** bei der Auswertung von Ausdrücken zu **umgehen**.
- Java und andere Sprachen bieten deshalb Möglichkeiten, Werte eines Typs in einen Wert eines anderen Typs umzuwandeln (engl: **casting**)
 - Dies ist aber nur für eine sehr eingeschränkte Menge von Paaren von Typen möglich. So ist z.B. eine Typumwandlung von **boolean** nach **int** (und umgekehrt) **nicht** möglich.
 - Dieses Codefragment funktioniert z.B. **nicht** in Java:

```
int n = 10;
while (n) A;
```
 - bei jeder Typumwandlung muss das **Ergebnis** (d.h. der Wert) **klar** sein.

Explizite und implizite Typumwandlung

- Wir betrachten folgende Variablen

```
int i = 1;  
long l = 12345678901;  
float f = 3.14f;
```

- Unterscheidung zwischen

- **expliziter** Typumwandlung (z.B. von **float→int**)
 - Angabe des Typs in Klammern vor dem Ausdruck. Z. B. :
 $i = (\text{int}) f;$
- **impliziter** Typumwandlung (z.B. von **int→float** oder **long→float**)
 - Dies ist nur möglich, wenn der Zieltyp einen größeren Wertebereich besitzt, z.B.

```
f = i;  
f = l;
```

Typausweitung ist immer erlaubt!

Explizite und implizite Typumwandlung

```
int i=1;  
long l=12345678901;  
float f = 3.14f;  
  
// explizite Typumwandlung (z.B. von float→int)  
i = (int) f;  
// implizite Typumwandlung  
// (z.B. von int→float oder long→float)  
f = i;  
f = l;
```

- Eine implizite Typumwandlung ist immer dann möglich, wenn der Zieltyp einen größeren Wertebereich besitzt.

Typanpassung bei Java

- Typausweitung ist immer erlaubt!

```
byte vb = 127;  
short vs = 255;  
int vi = 600000;  
long vl = 123456789;  
  
vs = vb; vi = vs; // usw.
```

- Umgekehrt geht es nicht!
- Was oft geht, ist eine explizite Anpassung mit einem sog. **type cast**.
 - Diese führt aber ggf. zu *Datenverlust*.
 - Zu jedem Datentyp **T** gibt es einen *type cast-Operator (T)*.
 - Ob man ihn auch anwenden darf, hängt vom Kontext ab.
 - Generell gehen Anpassungen: *Zahlen* → *Zahlen*
 - Keinen Sinn macht z.B.: *String* → *Zahlen*

~~vb = vs; vi = vl; // usw.~~

```
int vix = 600000;  
short vsx = (short) vix;  
System.out.print(vsx);
```



10176

Beispiele für Java-Ausdrücke

- Seien **x**, **y**, **z** Variablen vom Typ **int**

Welchen Typ haben die folgenden Ausdrücke?

x+2*(y+z)

int

x + 3.25 * (y + 0.1)

double

x + 5 == z - 28

boolean

4.3 Die Datentypen char und String

- Ein weiterer einfacher Datentyp ist der Datentyp **char**.
 - Der Datentyp repräsentiert **ganze positive Zahlen** im Bereich $0, 1, \dots, 65535$ ($= 2^{16}-1$). Es gelten also die typischen Regeln für ganze Zahlen:

```
char ch1 = 87;  
int intch1 = ch1;  
char ch3 = (char) intch1;
```

- Zudem stehen als Operationen `++`, `--` und die Zuweisungsoperatoren `=`, `+=`, `-=`, ... zur Verfügung.
- Es gibt aber **Unterschiede** zu den anderen ganzzahligen Datentypen.
 - Als Literale stehen zusätzlich noch die Zeichen des **UCS/Unicode-Zeichensatz** zur Verfügung.
char ch2 = 'x'; *Das Zeichen steht zwischen 2 Apostrophen!*
 - Zudem wird bei der Ausgabe eines Werts vom Typ `char` durch `System.out.print` das Zeichen des Unicode-Zeichensatzes verwendet.

Zeichensätze

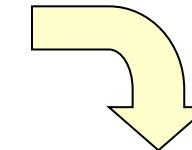
- Java nutzt den **UCS/Unicode**-Zeichensatz, der neben den lateinischen Schriftzeichen auch kyrillische, chinesische, japanische, koreanische und zahlreiche weitere Zeichensätze enthält.
- Die ersten 256 Zeichen entsprechen dem **ASCII**-Zeichensatz

000	NUL	033	!	066	B	099	c	132	ä	165	Ñ	198	ää	231	þ
001	Start Of Header	034	"	067	C	100	d	133	å	166	¤	199	Å	232	Þ
002	Start Of Text	035	#	068	D	101	e	134	å	167	°	200	łł	233	ú
003	End Of Text	036	\$	069	E	102	f	135	ç	168	¿	201	ƒƒ	234	ú
004	End Of Transmission	037	%	070	F	103	g	136	é	169	®	202	łł	235	ú
005	Enquiry	038	&	071	G	104	h	137	ë	170	˜	203	łł	236	ý
006	Acknowledge	039	?	072	H	105	i	138	è	171	½	204	łł	237	Ý
007	Bell	040	(073	I	106	j	139	í	172	¼	205	=	238	-
008	Backspace	041)	074	J	107	k	140	ł	173	ı	206	‡‡	239	.
009	Horizontal Tab	042	*	075	K	108	l	141	ı	174	«	207	¤¤	240	-
010	Line Feed	043	+	076	L	109	m	142	À	175	»	208	đ	241	±
011	Vertical Tab	044	,	077	M	110	n	143	Á	176	„„	209	Đ	242	-
012	Form Feed	045	-	078	N	111	o	144	È	177	‰‰	210	€	243	¼
013	Carriage Return	046	.	079	O	112	p	145	œ	178	֍֍	211	ÊÊ	244	††
014	Shift Out	047	/	080	P	113	q	146	Œ	179		212	ÈÈ	245	§§
015	Shift In	048	0	081	Q	114	r	147	ó	180	†	213	ı	246	÷
016	Delete	049	1	082	R	115	s	148	ö	181	À	214	í	247	:
017	-- frei --	050	2	083	S	116	t	149	ò	182	Á	215	î	248	º
018	-- frei --	051	3	084	T	117	u	150	ú	183	À	216	í	249	-
019	-- frei --	052	4	085	U	118	v	151	ü	184	®	217	j	250	.
020	-- frei --	053	5	086	V	119	w	152	ÿ	185	‡‡	218	ѓ	251	‘
021	Negative Acknowledge	054	6	087	W	120	x	153	ö	186		219	■■	252	»
022	Synchronous Idle	055	7	088	X	121	y	154	ú	187		220	■■	253	»
023	End Of Transmission Block	056	8	089	Y	122	z	155	ø	188		221	ı	254	■
024	Cancel	057	9	090	Z	123	{	156	£	189	¢	222	ı	255	-
025	End Of Medium	058	:	091	[124		157	ø	190	¥	223	■		
026	Substitute	059	:	092	\	125	}	158	×	191	¬	224	ó		
027	Escape	060	<	093]	126	~	159	ƒ	192	Ł	225	ß		
028	File Separator	061	=	094	^	127	¤	160	á	193	Ł	226	ó		
029	Group Separator	062	>	095	-	128	ç	161	í	194	Ł	227	ó		
030	Record Separator	063	?	096		129	ú	162	ö	195	Ł	228	ó		
031	Unit Separator	064	@	097	a	130	é	163	ú	196	-	229	ó		
032		065	A	098	b	131	â	164	ñ	197	+	230	µ		

Operationen auf char-Datentypen

- Inkrement/Dekrement-Operationen `++` und `--` liefern Vorgänger und Nachfolger-Zeichen
- Zuweisungsoperationen (mit ganzer Zahl): `+=`, `-=`,
- Vergleichs-Operationen `==`, `!=`, `<`, `<=`, `>`, `>=`
- Beispiel:

```
char c = 40;
while (c <= '9')
    System.out.print(c++);
System.out.println();
```



`() *+, -./0123456789`

- Warnung:

```
c++; // ok
c += 1; // ok
c = c + 1; // nicht ok
```

Im 3. Fall bekommt die rechte Seite durch Typanpassung den Typ `int`, der **nicht** implizit an den Typ `char` angepasst wird!

String-Datentypen in Programmiersprachen

- Der Typ **String** ist ein erstes Beispiel eines **zusammengesetzten** Typs.
 - Ein **String** ist eine **Zeichenkette** und besteht aus einer Folge von Zeichen (von Werten des Datentyps **char**).
- Mit der Menge aller solcher Zeichenketten wird die **Datentyp String** gebildet.
- Operationen sind z.B.:

+ :	String × String → String	Konkatenation
length :	String → int	Stringlänge
charAt :	String × int → char	Zugriff auf Zeichen
indexOf :	String × char → int	Position eines Zeichens

Zeichen- und Zeichenketten-Literale

- Ein Zeichen-Literal (**char**-Literal) ist ein einzelnes, in einfache Apostrophe eingeschlossenes Unicode-Zeichen:

```
'a' '%t' 'a' 'w' 'α' 'Ω' 'æ' 'ç' 'Ã' 'ß'
```

- Ein Zeichenketten-Literal (**String**-Literal) ist eine Folge von Unicode-Zeichen, in Doppel-Apostrophien:

```
"Dies ist ein String."
```

- Ein **String**-Literal muss auf genau einer Zeile beginnen und enden.
- Allerdings können String-Literale mit dem **+** Operator verkettet (konkateniert) werden. Sie bilden dann ein zusammengefasstes **String**-Literal.
- Beispiele für weitere **String**-Literale:

```
"Dies ist ein String, der auf " +  
"zwei Zeilen verteilt wurde."
```

```
"Tar" + "tar" + " ist " +  
"keine Käsesorte!"
```

```
"Ввгдтъ Юњяы Швгд Итњяыш Жл"
```

Ersatzdarstellungen

- In Zeichen- oder String-Literalen können bzw. müssen Ersatzdarstellungen benutzt werden.
- Falls das eingeschlossene Zeichen selbst ein Apostroph oder ein \ sein soll, oder ein nicht-druckbares Zeichen ist, muss eine der folgenden Ersatzdarstellungen, auch Escape-Sequenzen genannt, verwendet werden.

```
\" für ein "
\' für ein '
\\ für ein \
\t für einen horizontalen Tabulator (HT: ASCII-Wert 9)
\n für einen Zeilenwechsel (LF: ASCII-Wert 10)
```

Weitere Ersatzdarstellungen:

```
\b für einen Rückwärtsschritt (BS: ASCII-Wert 8)
\f für einen Formularvorschub (FF: ASCII-Wert 12)
\r für einen Wagenrücklauf (CR: ASCII-Wert 13)
```

```
'\t' '\\"' '\"'
```

```
"\tDieser Text\r\n\twurde formatiert."
```

Aufruf von Operationen

- Der +-Operator verbinden zwei Zeichenketten zu einer neuen Zeichenkette.
 - String str1 = "Uni";
String str2 = "Marburg";
String str3 = str1 + " " + str2;
- Die anderen Operationen werden anders aufgerufen.
 - Der erste Parameter (vom Typ String) kommt zuerst, dann folgt ein Punkt und der Aufruf der Methode ohne den ersten Parameter.
 - Beispiele:
 - int len = str1.length(); // Liefert die Länge der Zeichenkette str1
 - char ch = str2.charAt(3); // Liefert das Zeichen an Position 3
 - int pos = str3.indexOf(' '); // Liefert die Position des Leerzeichens
 - Vor dem Punkt darf ein beliebiger Ausdruck vom Typ String stehen.
 - len = (str1 + str2).length(); // Liefert die Länge der beiden Zeichenkette str1 und str2

Beispiel: Mustersuche in Text

```
/** Prüft, ob ein Textmuster in einer Zeichenkette an einer Position vorkommt.  
 * @param text Zeichenkette  
 * @param pattern Muster der Länge l  
 * @param pos Position  
 * @return text[pos, pos + 1, ... ,pos + l - 1] == pattern  
 */  
  
boolean isSubStringAtPosition(String text, String pattern, int pos) {  
    int i = 0;  
    while (i < pattern.length()) {  
        if (i + pos < text.length() && text.charAt(i + pos) == pattern.charAt(i))  
            i = i + 1;  
        else  
            return false;  
    }  
    return true;  
}
```

Zusammenfassung

- Datentypen
 - Wertemenge
 - Operationen
- Primitive Datentypen in Java
 - boolean, ganze Zahlen, Fließpunktzahlen
- Ausdruck und Anweisung
 - Ausdrucks hat einen Typ
 - Auswertung unter Verwendung von Prioritäten der Operatoren
 - Seiteneffekte in Java
- Typumwandlung
 - Explizite und implizite

```
//Meine Lösung  
//Idee zwei Indexe laufen lassen eines über den Text und eines über pattern  
//Natürlich muss sichergestellt werden, dass die Indexe nicht weit laufen sonst haben  
//wir ein Exception
```

```
boolean isSubStringAtPosition(String text, String pattern, int pos) {  
    int i=pos;  
    int j=0;  
    while (j< pattern.length()) {  
        if ( (i < text.length() ) & (text.charAt(i) == pattern.charAt(j) ) )  
        {  
            i = i + 1;  
            j++;  
        }  
        else  
            return false;  
    }  
    return true;  
}
```

4.4 Arrays und for-Schleifen

- In vielen Anwendungen gibt es das Problem eine Folge von Daten des gleichen Typs zu verarbeiten.
 - Gegeben die Notenpunkte der Studierenden des Moduls Praktische Informatik 1. Gesucht ist die Durchschnittsnote.
- Arrays repräsentieren Folgen von Datenelementen des gleichen Typs
 - Mathematisch: x_0, x_1, x_2, x_3
 - Java: $x[0], x[1], x[2], x[3]$
 - In Java können wir mit Hilfe **einer Variable** auf **alle** Folgenelemente eines Arrays zugreifen.
 - Im Gegensatz zu den bisher bekannten Variablen, muss der Speicherplatz eines Arrays **im Programm explizit reserviert** werden.



4.4.1 Der Array Datentypen

- Voraussetzung
 - T ist bereits ein bekannter Datentyp
- Formale Definition des Datentyps und seine Operationen
SORT T[]

OPS

length : array	→ int
new : int	→ T-array
[] : T-array x int	→ T

Deklaration

- Zu jedem beliebigen Typ T kann ein Array-Typ definiert werden.
 - **T[]**
 - Beispiele für Typen
 - **int[]**
 - **double[]**
- Wie für jeden anderen Typ können zu einem Array-Typ Variablen deklariert werden.
 - **int[] x;** // x ist eine Variable für eine Folge von ganzen Zahlen
 - **double[] r;** // r ist eine Variable für eine Folge von Gleitpunktzahlen
- Im Gegensatz zu Variablen primitiver Datentypen **verweisen** diese Variablen nur **auf den Speicherplatz** eines Arrays (Details später).
 - Wir sprechen dann von einer **Referenzvariablen**.
 - Der Speicherplatz für das Array wird durch die Variablen-deklaration noch nicht reserviert.



Beispiel

```
/** Erster Entwurf einer Methode zur Speicherung der Zahlen 1,2,3,... in einem Array
 * und der Berechnung der Summe.
 */
int gaussSumme () {
    int[ ] arr;
    int sum = 0;
    int i = 0;
    // Code zur Speicherplatzreservierung und Initialisierung des Arrays arr
    // Code zur Berechnung der Summe
    return sum;
}
...
```

Erzeugung eines Arrays

- Die Speicherplatzreservierung für Arrays erfolgt durch den new-Operator
 - **arr = new int[10];**
// Folge arr[0], arr[1], … , arr[9] wird erzeugt; **arr** verweist auf die Folge
 - Bei dem new-Operator muss der Typ des Arrays und die Größe des Arrays zwischen den eckigen Klammern angegeben werden.
- Der new-Operator liefert als Ergebnis die Speicheradresse des Arrays. Diese Speicheradresse wird in der Referenzvariable hinterlegt.

```
// Erster Entwurf einer Methode zur Berechnung von einer Summe
int gaussSumme () {
    int[] arr;
    int sum = 0;
    int i = 0;
    arr = new int[10];
    // Code für die Initialisierung
    // Code zur Berechnung der Summe
    return sum;
}
...
```

Initialisierung eines Arrays

- Ein Array kann elementweise initialisiert werden.
- Auf jedes Element des Array kann **schreibend zugegriffen** werden, in dem der **Selektionsoperator []** benutzt wird.

```
arr[1] = 2;
```

- Entsprechend kann auch durch Verwendung des Selektionsoperators **lesend** auf die Elemente des Array **zugegriffen** werden.

```
sum += arr[1];
```

```
// Erster Entwurf einer Methode zur Berechnung von einer Summe
int gaussSumme () {
    int[] arr;                                // Deklaration der Array-Variable
    int sum = 0;
    int i = 0;
    arr = new int[10];                         // Erzeugung des Arrays mit 10 Elementen
    while (i < 10) {                           // Initialisierung (Wertzuweisung)
        arr[i] = ++i;
    }
    // Code zur Berechnung der Summe
    return sum;
}
```

...

Länge eines Arrays

- Manchmal ist in der Methode nicht bekannt, wie lang das Array ist.
- Die **Länge des Arrays** arr erhält man stets durch den Ausdruck
arr.length
- Diese **Schreibweise mit dem Punkt** werden wir später noch öfter benutzen.

```
// Erster Entwurf einer Methode zur Berechnung von einer Summe
int gaussSumme () {
    int[] arr;
    int sum = 0;
    int i = 0;
    arr = new int[10];
    while (i < arr.length) {
        arr[i] = ++i;
    }
    // Code zur Berechnung der Summe
    return sum;
}
...
```

Länge eines Arrays

- Manchmal ist in der Methode nicht bekannt, wie lang das Array ist.
- Die **Länge des Arrays** arr erhält man stets durch den Ausdruck
arr.length
- Diese **Schreibweise mit dem Punkt** werden wir später noch öfter benutzen.

```
// Erster Entwurf einer Met
int gaussSumme () {
    int[] arr;
    int sum = 0;
    int i = 0;
    arr = new int[10];
    while (i < arr.length) {
        arr[i] = ++i;
    }
    // Code zur Berechnung der Summe
    return sum;
}
...
```

Achtung: Die Indizes beginnen bei 0. Daher hat ein Array der Länge 10 die Indizes 0 .. 9.

4.4.2 Speicherplatz im Programm



- Jedes Programm besitzt zwei Arten von Speicher
 - Stack-Speicher
 - Dort werden beim Aufruf einer Methode der Speicherplatz für die Variablen der Methode abgelegt (**Methodeninstanz**).
 - Heap-Speicher
 - Dort werden die Arrays (und Objekte) abgelegt, die mit dem **new-Operator** erzeugt werden.
 - Diese Arrays können nicht direkt angesprochen werden, sondern nur indirekt über eine **Referenzvariable**.
 - Der Wert einer Referenzvariable kann entweder im Stack-Speicher oder im Heap-Speicher liegen.
- Beim Aufruf der Methode `gaussSumme` werden drei Variablen auf den Stack-Speicher abgelegt.

Stack-Speicher

Aufrufer

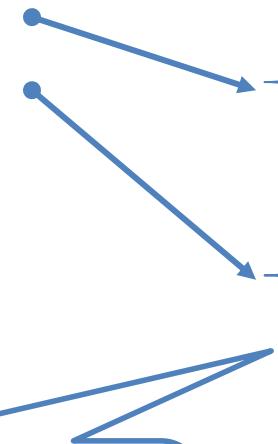
→ Aufruf von gaussSumme()

Stack-Speicher

Bezeichner	Wert
?	?
?	?
arr	<siehe nächste Folie>
sum	0
i	0

Stack-Speicher

Aufrufer



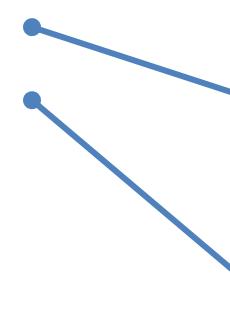
Stack-Speicher

Bezeichner	Wert
?	?
?	?
arr	<siehe nächste Folie>
sum	0
i	0

Speicherplatz kann wieder verwendet werden, wenn Aufrufer eine neue Methode aufruft.

Stack-Speicher

<Programmende>



Stack-Speicher

Bezeichner	Wert
?	?
?	?
arr	<siehe nächste Folie>
sum	0
i	0

Stack-Speicher

Aufrufer

→ Aufruf von gaussSumme()

arr ist eine
Referenzvariable. Größe
des Array-Werts ist nicht
immer bekannt.

B	?
?	
arr	?
sum	0
i	0

Durch den Aufruf des new-Operators wird
Speicherplatz im Heap-Speicher reserviert.

- arr = new int[10];

Adresse	Wert
1	?
2	?
...	...
41	?
42	0
43	0
44	0
45	0
46	0
47	0
48	0
49	0
50	0
51	0
52	?
...	

Heap-Speicher

Stack-Speicher

Aufrufer

→ Aufruf von gaussSumme()

Bezeichner	Wert
?	?
?	?
arr	
sum	0
i	0

Adresse	Wert
1	?
2	?
...	...
41	?
42	0
43	0
44	0
45	0
46	0
47	0
48	0
49	0
50	0
51	0
52	?
...	

Werte der
Array-
Elemente

Durch Verwendung des []-Operators können wir schreibend und lesend auf den Inhalt eines Arrays zugreifen.

- `arr[i] = i+1;`
- `arr2 = arr;`

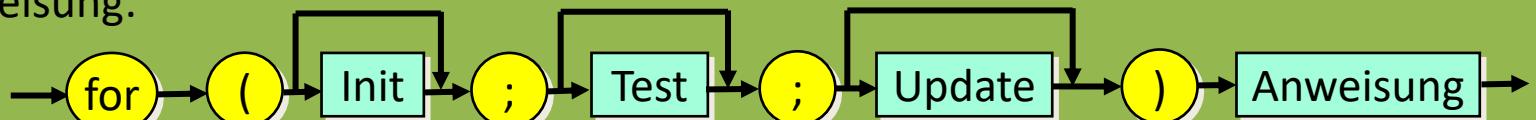
Speicheradresse (vereinfacht):
Array-Start + Index.

Heap-Speicher

4.4.3 for-Schleife

- Statt einer while-Schleife ist es oft einfacher eine for-Schleife zu benutzen.

for- Anweisung:



- Der Schleifenkopf der for-Schleife setzt sich zusammen aus
 - **Init** steht für eine oder mehrere durch Kommata getrennte Zuweisungen oder Variablendeklarationen mit Initialisierung.
 - `int i = 0`
 - **Test** steht für eine Bedingung, die meistens testet, ob in **Init** genannte Variablen eine Schranke überschreiten.
 - `i < max`
 - **Update** steht für eine oder mehrere durch Kommata getrennte Anweisungen die meist die in **Init** genannten Variablen verändern.
 - `i += 1`

for-Schleife und while-Schleife

- ▶ Die **for-Anweisung** ist äquivalent zu folgender **while-Anweisung** und standardisiert somit genau diesen Typ von while-Anweisungen:

```
{  
    Init;  
    while ( Test ) {  
        Anweisung;  
        Update;  
    }  
}
```

- ▶ Eigentlich wird die for-Schleife nicht benötigt, sondern wird nur zur Vereinfachung der Programmierung angeboten.
- ▶ Jede der drei Komponenten einer for-Anweisung können auch leer sein. Daher ist folgende Anweisung die kürzest mögliche for-Anweisung:

```
for(;;);
```

Keine Abbruchbedingung!
Endlosschleife.

for-Schleife: Einfache Beispiele

- Das folgende Beispiel zeigt eine for-Schleife, die die **Fakultäts-Funktion** iterativ berechnet und für $i=1, 2, \dots, 5$ ausgibt:

```
int fak = 1;
for (int i=1; i < 6; i++){
    fak = i*fak;
    System.out.println("Fakultät von " + i + " ist: " + fak);
}
```

- Analog die *Fibonacci-Funktion*:

```
int fibo0 = 1, fibo1 = 1, fiboneu;
for (int i=2; i < 6; i++){
    fiboneu = fibo0 + fibo1;
    fibo0 = fibo1;
    fibo1= fiboneu;
}
```

Verschachtelte for-Schleife: Beispiele

- Schleifen können auch verschachtelt sein.

```
int max = 10;
for (int i=0; i < max; i++){
    for ( int k=0 ; k < max-i-1 ; k++ )
        System.out.print(" ");
    for ( int k=0 ; k < 2*i-1 ; k++ )
        System.out.print("*");
    System.out.println();
}
```

- In diesem Beispiel werden 10 Zeilen ausgegeben.
 - In der 1. Zeile 1 Stern
 - In der 2. Zeile 3 Sterne
 - In der 3. Zeile 5 Sterne
 - usw.
 - Die Sterne sollen zentriert werden, d.h.
 - Vor der Sternausgabe müssen auch noch passend viele Leerzeichen ausgegeben werden.

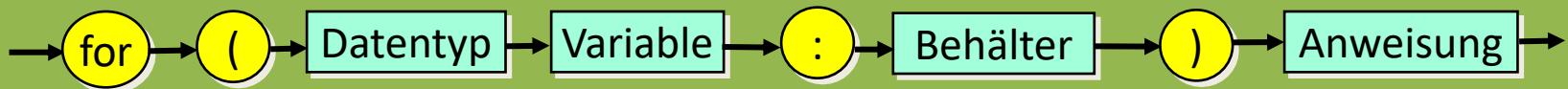
for-Schleife und Arrays

- for-Schleifen sind sehr gut geeignet, um Arrays zu durchlaufen.
 - Beachte: Die **Länge des Arrays** arr bekommt über **arr.length**

```
/** Erster Entwurf einer Methode zur Berechnung von einer Summe
int gaussSumme () {
    int[] arr;                                // Deklaration einer Array-Variable
    int sum = 0;
    arr = new int[10];                         // Reservierung des Speicherplatz
    for (int i = 0; i < arr.length; i++) {      // Initialisierung
        arr[i] = i+1;
    }
    for (int i = 0; i < arr.length; i++) {      // Lesender Zugriff auf das Array
        sum += arr[i];
    }
    return sum;
}
...
```

Syntax der foreach-Schleife

foreach- Anweisung:



- Dies ist die allgemeine Form der **foreach-Anweisung**. **Behälter** steht für den Namen einer Variablen einer **Behälter-Datenstruktur**.
 - Derzeit kennen wir in diesem Zusammenhang nur **Arrays** als Behälter.
- Der Datentyp am Anfang muss der Datentyp der Elemente des Behälters sein, also der Typ der Array Elemente.
 - Der folgende Code-Schnipsel zeigt die Schleife in unserem Beispiel bei der Berechnung der Summe.
 - Leider kann mit der foreach-Schleife **nicht schreibend** auf die Elemente des Arrays zugegriffen werden.

```
int[] arr = new int[10];
int sum = 0;
for (int i = 0; i < arr.length; i++) { // Hier müssen wir die alte for-Schleife verwenden.
    arr[i] = i+1;
}
for (int elem : arr) { // Hier funktioniert die neue Schleife.
    sum += elem; } // Lesender Zugriff ist erlaubt.
```

4.4.4 Besonderheiten bei Arrays

- Eine Zuweisung von Array-Variablen ist wie bei allen anderen Variablen möglich.

```
int[] arr1 = new int[5], arr2;
for (int i = 0; i < arr.length; i++)
    arr1[i] = i+1;
arr2 = arr1;
```

- Aber was passiert dabei?

- Der Wert der Variablen arr1 und arr2 ist eine Speicheradresse.
- Somit wird bei der oberen Zuweisung dieser Wert von arr1 an arr2 übergeben.

Bezeichner	Wert
arr1	
arr2	

Adresse	Wert
1	?
2	?
...	...
41	?
42	1
43	2
44	3
45	4
46	5

Werte der Array-Elemente

Solche Zuweisungen passieren insbesondere beim Aufruf einer Methode mit einem Array als Parametervariable.

4.4.4 Besonderheiten bei Arrays

- Eine Zuweisung von Array-Variablen ist wie bei allen anderen Variablen möglich.

```
int[] arr1 = new int[5], arr2;
for (int i = 0; i < arr.length; i++)
    arr1[i] = i+1;
arr2 = arr1;
```

- Aber was passiert dabei?

- zwei verschiedene Referenzvariablen können auf das gleiche Array zugreifen
- $arr1[1] = 6;$
 $if (arr2[1] == 6) System.out.println("Achtung");$
// erfüllt.

Bezeichner	Wert
arr1	
arr2	



The diagram illustrates the state of memory after the execution of the code. A table shows memory addresses (Adressen) from 1 to 46, with their corresponding values (Werte). An arrow points from the variable declarations in the table below to the memory representation here. Two pointers, arr1 and arr2, both point to the same element at address 42, which contains the value 1. Other elements in the array are filled with values 6, 3, 4, and 5 respectively.

Adresse	Wert
1	?
2	?
...	...
41	?
42	1
43	6
44	3
45	4
46	5

arr1[0] und
arr2[0]

arr1[1] und
arr2[1]
Elemente

Geteilte Referenzvariablen

- Wir ein Array an eine Methode übergeben, so kann diese den Arrayinhalt ändern

```
/** Methode zur Berechnung von einer Summe eines Arrays
 * @param arr ein Array mit ganzen Zahlen.
 * @return Die Summe der Zahlen des Arrays.
 */
int getSumme (int[] arr) {          // Array-Parametervariable
    for (int i = 1; i < arr.length; i++) {
        arr[0] += arr[i];
    }
    return arr[0];
}

// Aufruf in der jshell
int arr[] = {1, 2, 3};
int sum = getSumme(arr);
System.out.println("Summe = " + sum);
for (int e : arr) { System.out.print(e + ", "); }
```

Verändert den
Arrayinhalt.

Geänderter Inhalt auch
beim Aufrufer.

Array Literale

- Ähnlich wie bei anderen Typen können **konstante Arrays in einer speziellen Syntax** angegeben werden.
 - Die Array-Elemente stehen dabei mit Komma getrennt in einem Mengenklammerpaar.

{1,2,3,4,5}

- Diese Array-Literale können **ausschließlich** bei der

- Deklaration einer lokalen Array-Variable

int[] arr = {1,2,3,4,5};

- und beim Aufruf einer Methode verwendet werden. Hierbei muss aber noch der Typ des Arrays zusätzlich angegeben werden.

long res = getSumme(new int[] {1,2,3,4,5});

Arrays als Parametervariablen

- Die Methode bekommt ein Array übergeben und liefert die Summe zurück.

```
/** Methode zur Berechnung von einer Summe eines Arrays
 * @param arr ein Array mit ganzen Zahlen.
 * @return Die Summe der Zahlen des Arrays.
 */
long getSumme (int[] arr) {          // Array-Parametervariable
    long sum = 0;
    for (int elem : arr) {           // Neue Schleifensyntax
        sum += elem;
    }
    return sum;
}

// Aufruf in der jshell
long l = getSumme(new int[]{1,2,3}); // Literal als Parameter
System.out.println("Summe = " + l);
```

Standardwerte für Array-Elemente

- Bisher: Zugriff auf Variablen-Werte erst nach deren Initialisierung

- Ist das erlaubt?

```
int[] notenspiegel = new int[15];  
System.out.println(notenspiegel[14]);
```

- Ja, es ist erlaubt: Für Arrayelemente gelten nicht die gleichen Regeln wie für lokale Variablen.

- Aber: welchen Wert hat ein nicht-initialisiertes Arrayelement?

- Java initialisiert Arrayelemente mit Standard-Werten:
 - 0, 0L für int, long, etc.
 - 0.0f, 0.0d für float, double
 - false für boolean
 - null für String, Arrays, etc.

Später mehr dazu.

Zuweisungs- und Inkrementoperatoren

- In Bezug auf die nötige Initialisierung unterscheiden sich Array-Elemente von lokalen Variablen.
- Die Zuweisungs- und Inkrementoperatoren funktionieren aber auch hier.

```
int arr[] arr = {1, 2, 3};  
arr[0]++;           // arr: {2, 2, 3}  
arr[1] *= 10;       // arr: {2, 20, 3}
```

Vergleich von Arrays

- Beim **Vergleich von** zwei Array-Variablen werden die **Referenzen** verglichen.
 - Beim folgenden Vergleich sind die Referenzen gleich. Somit ergibt der Vergleich also **true**

```
int[] a = { 17, 42, 47 };  
int[] b = a;  
System.out.println(a == b);
```

- *Beim folgenden Vergleich sind die Referenzen verschieden, da b eine Kopie von a referenziert.*
 - *Der Vergleich ergibt also **false** obwohl a eine identische Kopie von b ist!*

```
int[] a = { 17, 42, 47 };  
int[] b = new int[3];  
for (int i = 0; i < 3; i++)  
    b[i] = a[i];  
System.out.println(a == b);
```

Die Nullreferenz

- Eine Array-Variable kann mit Default-Wert **null** initialisiert werden.

```
int[] a = null;
```

- Eine bisher anders verwendete Array-Variable kann auch durch Zuweisung einer null-Referenz **außer Betrieb genommen werden**.

```
a = null;
```

- Dabei können Arrays entstehen, die nicht mehr referenziert werden.

- Diese Arrays können nicht mehr genutzt werden. Sie sind also Datenmüll.
 - Die „Java-Müllabfuhr“ (**garbage collector**) sammelt den Müll ein und der Speicherplatz kann **recycelt** werden.

```
int[] a = { 17, 42, 47 };  
...  
a = null;
```

a: **null**

nicht mehr
referenziert
Müll

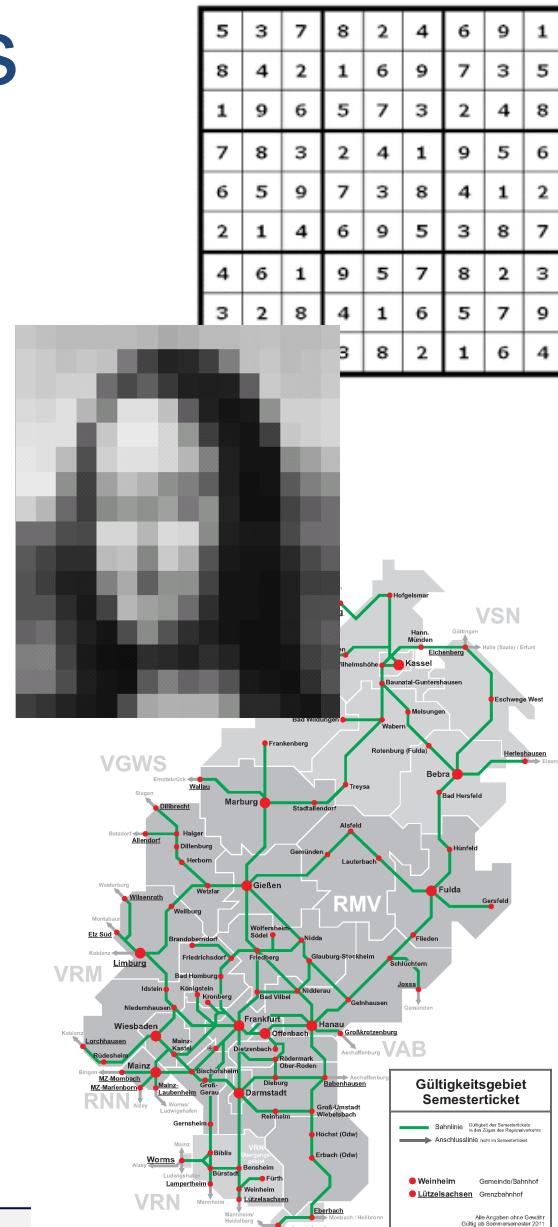
17
42
47

- Benutzen wir später nochmal a, kommt es zu einem Abbruch des Programms. Eine sogenannte Exception, genauer NullPointerException, wird geworfen.



4.4.5 Mehrdimensionale Arrays

- Arrays kann man mit beliebigem Komponententyp T bilden.
 - Insbesondere kann der Datentyp selbst wieder ein Array sein.
 - Der zum Datentyp $T[]$ gehörende Array-Datentyp ist somit:
- $T[][].$
- Wir sprechen dann von einem zweidimensionalen und allgemein von mehrdimensionalen Arrays.
 - Man benutzt mehrdimensionale Arrays z.B. zur Speicherung und Bearbeitung von
 - Bildern
 - Graphen
 - Wer ist mit wem im Netzwerk befreundet?
 - Welche Städte haben eine direkte Zugverbindung?
 - Distanztabellen



Deklaration und Erzeugung

- Deklaration der Array-Variablen

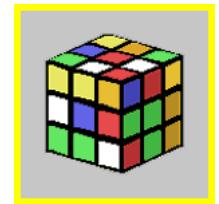
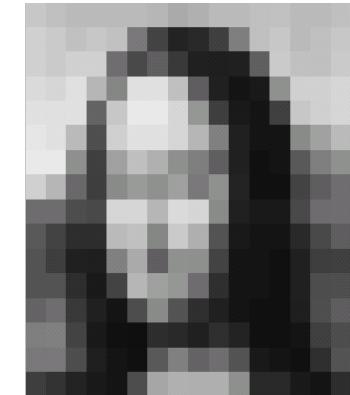
- `int [] [] greyMonaLisa;`
- `int [] [] bildschirm;`
- `Color[][][] rubik ;`

- Erzeugung eines Arrays

- `bildschirm = new int [1024][748];`
- `rubik = new Color[3][3][3];`

- Deklaration einer Array-Variable mit direkter Initialisierung

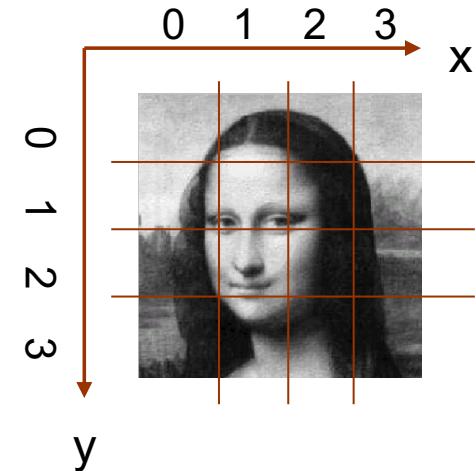
- `boolean[][] xorTabelle = {{false, true},{true, false}};`
- `int[][] entfernung = { { 0, 213, 419, 882}, {213, 0, 617, 720}, {419, 617, 0, 521}, {882, 720, 521, 0} };`



Beispiel: Bildbearbeitung

- Graphik als Matrix von Grauwerten

- ```
int[][] monaGrey = { {21,26,72,66},
 {38,22,33,60},
 {50,59,59,63},
 {23,45,72,80} } ;
```



- Aufhellen

- ```
for (int x = 0; x < breite; x++)
    for(int y = 0; y < hoehe; y++)
        monaGrey[x][y] = monaGrey[x][y]*9/10 ;
```

- Negieren

- ```
for (int x = 0; x < breite; x++)
 for(int y =0; y < hoehe; y++)
 monaGrey[x][y] = 255-monaGrey[x][y] ;
```



- Schwarzweiss

- ```
for (int x = 0; x < breite; x++)
    for(int y =0; y < hoehe; y++)
        monaGrey[x][y] = (monaGrey[x][y] < 128)? 0: 255;
```

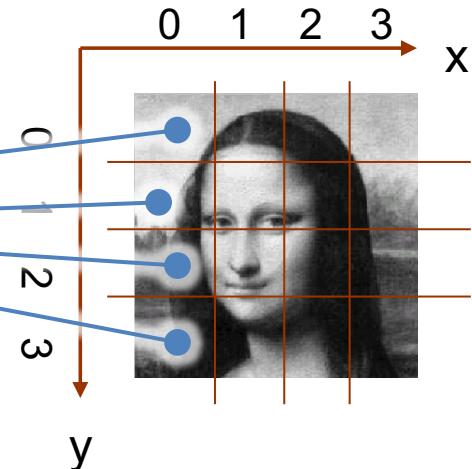


*Fragezeigeoperator
(BoolExpr) ? Expr1 : Expr2*

Beispiel: Bildbearbeitung

- Graphik als Matrix von Grauwerten

```
• int[][] monaGrey = { {21, 20, 72, 60},  
                      {38, 22, 33, 60},  
                      {50, 59, 59, 63},  
                      {23, 45, 72, 80} } ;
```



- Aufhellen

```
• for (int x = 0; x < breite; x++)  
    for(int y = 0; y < hoehe; y++)  
        monaGrey[x][y] = monaGrey[x][y]*9/10 ;
```



- Negieren

```
• for (int x = 0; x < breite; x++)  
    for(int y = 0; y < hoehe; y++)  
        monaGrey[x][y] = 255-monaGrey[x][y] ;
```



- Schwarzweiss

```
• for (int x = 0; x < breite; x++)  
    for(int y = 0; y < hoehe; y++)  
        monaGrey[x][y] = (monaGrey[x][y] < 128)? 0: 255;
```

*Fragezeigeoperator
(BoolExpr) ? Expr1 : Expr2*

Speicherrepräsentation von Arrays

- Ein zweidimensionales Array ist ein Array von Spalten
- Ein dreidimensionales Array ist ein Array von zweidimensionalen Arrays
- ...

```
int[][][] monaGrey =
{ {21,26,72,66},  

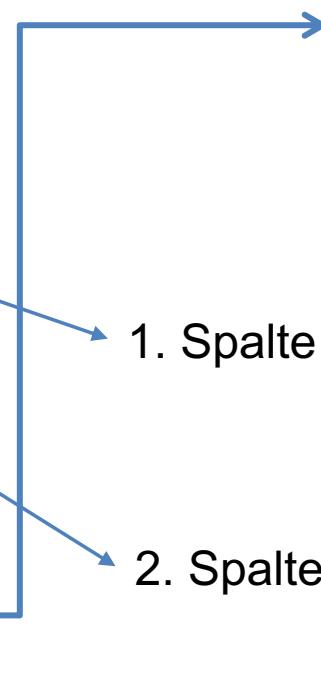
  {38,22,33,60},  

  {50,59,59,63},  

  {23,45,72,80} } ;
```

Bezeichner	Wert
monaGrey	42

Stack-Speicher



Adresse	Wert
1	?
...	...
42	46
43	50
44	54
45	58
46	21
47	26
48	72
49	66
50	38
51	22
52	33
...	...

Heap-Speicher

Zusammenfassung

- Array als Datentyp
- Stack- und Heap-Speicher
 - Explizite Erzeugung von Arrays im Heap-Speicher
- Referenzvariablen
 - Verweisen auf ein Array
 - Lesender und schreibender Zugriff
- Mehrdimensionale Arrays
- Neue Schleifen
 - for-Schleife
 - foreach-Schleife



5. Klassen

- Klassendefinition
- Klasse als Typ
- Klasse als Objektfabrik
- Konstruktoren
- Objektmethoden
- Statische Felder und Methoden

Motivation

- In vielen Anwendungen besitzen reale Objekte unterschiedliche Eigenschaften:
 - Studierende besitzen einen Vornamen, Nachnamen, Matrikelnummer, Fachbereich und das Fachsemester.
 - Jeder dieser Eigenschaften kann ein Datentyp zugeordnet werden.
 - Nachname ist vom Typ String
 - Fachsemester vom Typ int
 - Ein Bankkonto hat eine Kontonummer, aktuellen Kontostand und eine Kundennummer.
 - Eine Kontonummer ist vom Typ String.
 - Der aktuelle Kontostand vom Typ double.
- Wünschenswert wäre, wenn wir alle Daten eines Studierenden bzw. eines Bankkontos über eine Variable ansprechen könnten.
 - Was ist der Typ einer solchen Variable

Max, Mustermann, 12345, 12, 1

54321, 2500.23, 77

Motivation

- In vielen Anwendungen besitzen reale Objekte unterschiedliche Eigenschaften:
 - Studierende besitzen einen Vornamen, Nachnamen, Matrikelnummer, Fachbereich und das Fachsemester.
 - Jeder dieser Eigenschaften kann ein Datentyp zugeordnet werden.
 - Nachname ist vom Typ String
 - Fachsemester vom Typ int
 - Ein Bankkonto hat eine Kontonummer, aktuellen Kontostand und eine Kundennummer.
 - Eine Kontonummer ist vom Typ String.
 - Der aktuelle Kontostand vom Typ double.
- Wünschenswert wäre, wenn wir ...
... eines Studierenden
... eines Bankkontos über
... Was ist der Typ einer solchen V

Max, Mustermann, 12345, 12, 1

54321, 2500.23, 77

Achtung: wegen möglicher Rundungsfehler lassen sich nicht alle Geldbeträge als double darstellen. Mit Klassen lassen sich hier zutreffendere Datentypen erstellen.

Datentypen und Operationen

- Bestandteil von Datentypen
 - Wertemenge
 - Menge von erlaubten Operationen.
- Beispiel Bankkonto
 - Wertemenge
 - Das Kreuzprodukt String x int x double repräsentiert die Eigenschaften eines Kontos: Kontonummer, Kundennummer und Kontostand.
 - Operationen
 - Geld abheben Bankkonto x double → double
 - Geld einzahlen Bankkonto x double →
 - Kontostand abfragen Bankkonto → double
- Studierende haben andere Eigenschaften und benötigen andere Operationen.
 - Prüfe, ob ein Studierender an einem gegebenen Fachbereich eingeschrieben ist. Student x String → boolean



5.1 Klassen als eigene Datentypen

- Mit Hilfe von sogenannten **Klassen** können in objektorientierten Sprachen (wie Java) sehr flexibel **eigene Datentypen** definiert werden.
 - Datenelemente unterschiedlicher Typen werden dabei in einem neuen Typ zusammengefasst.
 - Die zu dem neuen Datentyp zugeordneten Operationen werden durch Angabe von Methoden zur Verfügung gestellt.
- Verwendung von **Klassen**
 - Wie bei Arrays können **Variablen mit Klassen-Typen** deklariert werden.
 - Wie bei Arrays muss der Speicherplatz für eine Instanz des neuen Datentyps **explizit reserviert** werden.
 - Reservierung geschieht ebenfalls auf dem Heap
 - Statt Instanz verwenden wir den Begriff **Objekt**.

Beispiel - Konto

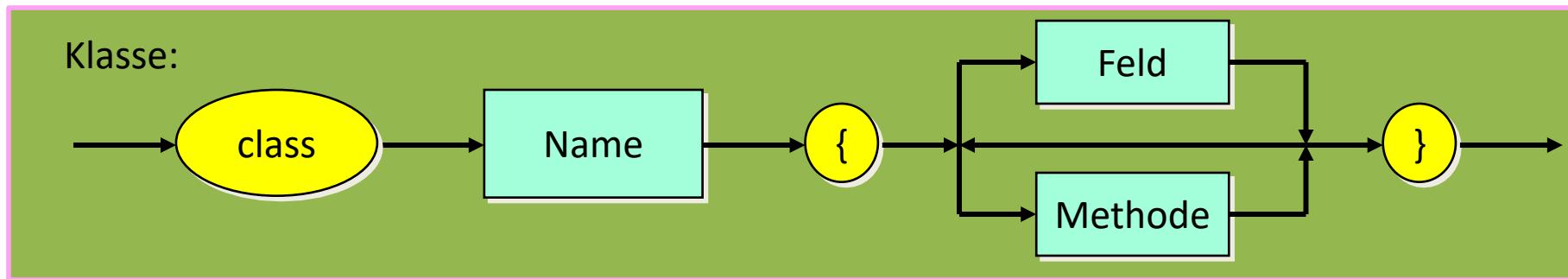
```
class Konto{  
    String kontoNr;  
    double kontoStand;  
    int kundenNr;  
  
    /** ... */  
    void einzahlen(double geld) {  
        ...  
    }  
    /** ... */  
    double abheben(double wunschBetrag) {  
        ...  
    }  
    /** ... */  
    double getKontoStand() {  
        ...  
    }  
}
```

Datenfelder

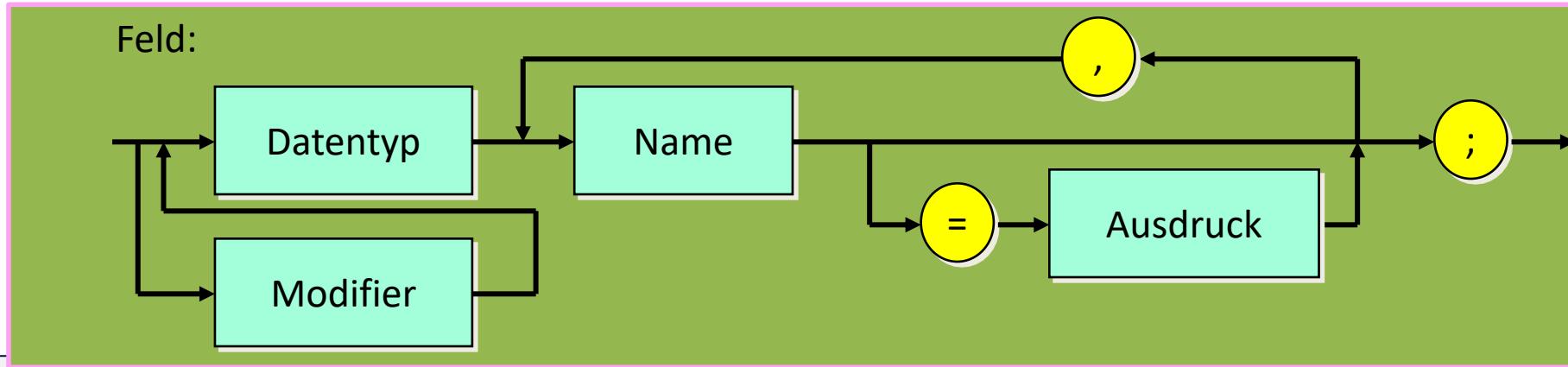
Objekt-methoden

Formale Definition einer Klasse

- **Klassen**-Definitionen haben die folgende – vereinfachte – syntaktische Struktur

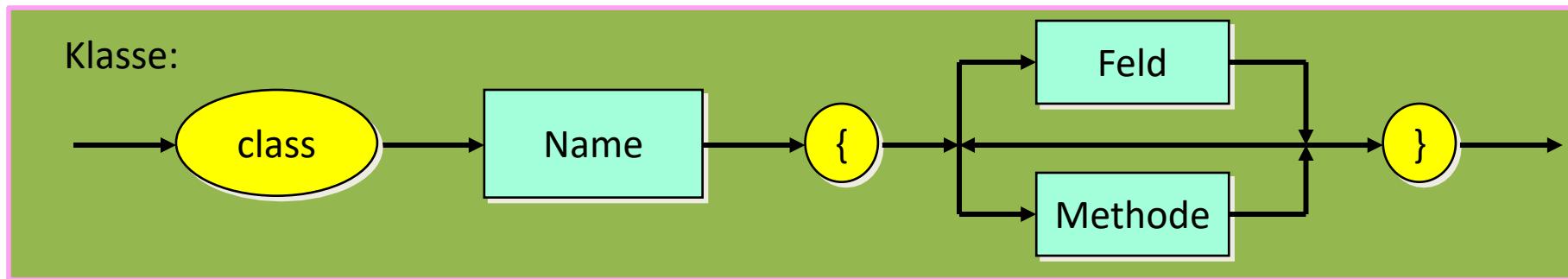


- Felder in einer Klasse werden ähnlich zu lokalen Variablen einer Methode deklariert.

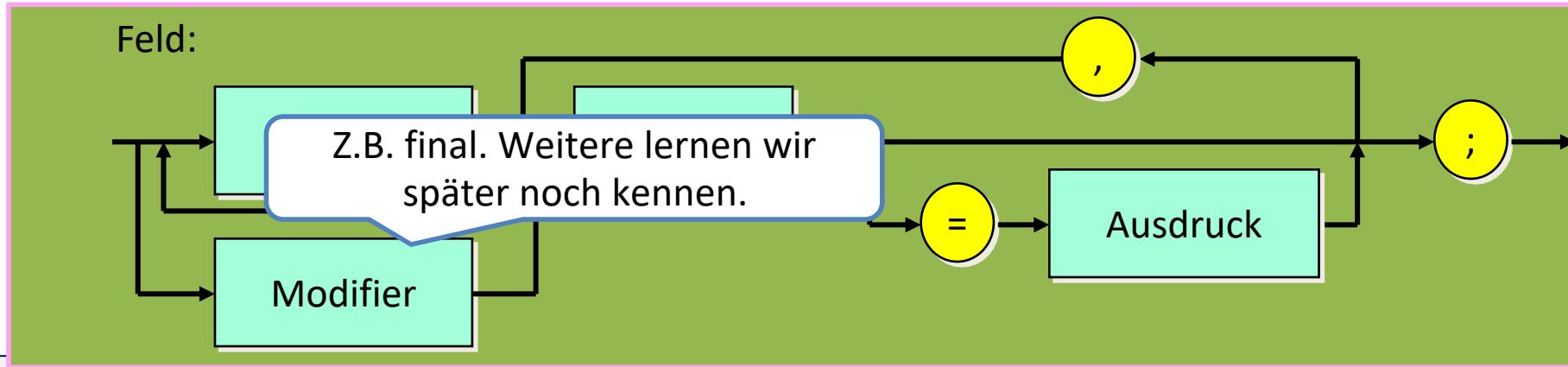


Formale Definition einer Klasse

- **Klassen**-Definitionen haben die folgende – vereinfachte – syntaktische Struktur



- Felder in einer Klasse werden ähnlich zu lokalen Variablen einer Methode deklariert.



Beispiel – Klasse Student

```
class Student {  
    String vname;  
    String nname;  
    int matrnr;  
    int fb;  
    int fachsemester;  
  
    /** Prüft die Zugehörigkeit zu einem Fachbereich  
     * @param fb Fachbereich  
     * @return true, wenn der Studierende vom Fachbereich fb  
     * ist.  
     */  
    boolean istImFachbereich(int fb) {  
        ...  
    }  
}
```

Datenfelder in Klassendefinition

- Ausgehend von beliebigen Typen T₁,...,T_m kann durch eine Klasse T ein neuer Typ definiert werden.

```
class T {  
    T1 a1;  
    T2 a2;  
    ...  
    Tm am;  
}
```

- Die Klasse T hat m Felder. Jedes Feld hat einen eindeutigen Namen.
- Beispiel

```
public class Student {  
    String vname;  
    String nname;  
    int matrnr;  
    int fb;  
    int fachsemester;  
}
```

oder

```
public class Student {  
    String vname, nname;  
    int matrnr, fb, fachsemester;  
}
```

Deklaration von Variablen

- Zu einer Klasse T können Referenzvariablen deklariert werden.

T myRefVar;

Beispiele:
Student fritz;
Konto vb;

- Analog zu Arrayvariablen verweisen diese Referenzvariablen nur auf den im Heap-Speicher liegenden Speicherplatz eines Objekts der Klasse.
 - Ein Objekt gibt es nach der Variablen-deklaration nicht.
- Genau wie bei Arrays kann einer Referenzvariable mit Klassentyp der spezielle Wert null zugewiesen werden.
 - Es gelten dann dieselben Besonderheiten wie bei den Arrays vorgestellt

5.2 Erzeugung der Objekte

Klasse Student



vorname:
nachname:
fb:
matrNr:

Konstruktoren



Objekte vom
Typ Student



vorname: Max
nachname: Mustermann
fb: 12
matrNr: 12345



vorname: Ute
nachname: Musterfrau
fb: 13
matrNr: 54321

Klassen als Objektfabriken

- Objekte werden durch den **new-Operator** erzeugt.

```
fritz = new Student();
```

```
vb = new Konto();
```

- Bei dem new-Operator muss der Klassename gefolgt von einem Klammerpaar angegeben werden.
 - Ähnlich zu einem Methodenaufruf können Parameter übergeben werden. Hierzu werden sogenannte **Konstruktoren** der Klasse benötigt.
 - Objekte werden fast immer über Variablen angesprochen, aber dies ist nicht zwingend erforderlich (→ wir werden später darauf eingehen).
- Der new-Operator liefert eine **Referenz** auf das erzeugte Objekt.
 - Diese Referenz wird in einer **Referenzvariable** hinterlegt.
 - Der Typ der Variable muss zu dem Typ des Objekts passen.
- Die Objekte selbst werden im Heap-Speicher gespeichert.

Klassen als Objektfabriken

- Objekte werden durch den **new-Operator** erzeugt.

```
fritz = new Student();
```

```
vb = new Konto();
```

- Bei dem new-Operator muss der Klassename gefolgt von einem Klammerpaar angegeben werden.
 - Ähnlich zu einem Methodenaufruf können Parameter übergeben werden. Hierzu werden sogenannte **Konstruktoren** der Klasse benötigt.
 - Objekte werden fast immer über Referenzen erstellt. Das ist nicht zwingend erforderlich (\rightarrow warum?)
- Der new-Operator liefert eine **Referenz** auf das Objekt.
 - Diese Referenz wird in einer **Referenzvariable** gespeichert.
 - Der Typ der Variable muss **zu dem Typ des Objekts passen**.
- Die Objekte selbst werden im Heap-Speicher gespeichert.

Wenn der Typ des Objekts und der Variablen gleich sind, ist das *passend*. Es gibt noch weitere Regeln, die wir noch kennenlernen.

Stack- und Heap-Speicher

Variable	Wert (Referenz)
fritz	99
sparkonto	42

Stack-Speicher

Adresse	Wert
1	...
2	...
...	...
42	?
43	?
44	?
...	...
99	?
100	?
101	0
102	0
103	0
...	...

Heap-Speicher

Objektinitialisierung

- Eine **Initialisierung** der Felder sollte entweder **direkt bei deren Deklaration** oder durch einen Konstruktor stattfinden.
 - Direkte Initialisierung

```
class Student {  
    String vname = "Max";  
    String nname = "Mustermann";  
    int matrnr = 12345;  
    int fb = 12;  
    int fachsemester = 1;  
  
    ...  
}
```

```
class Konto {  
    String kontoNr = "12345";  
    double betrag = 5.0;  
    int kundenNr = 42;  
}
```

- Damit bekommen alle Objekte der Klasse initial diese Werte.
 - Im Fall der Klasse Student ist dies nicht empfehlenswert, da typischerweise die Werte individuell gesetzt werden sollen.
 - Stattdessen empfiehlt sich die Benutzung von **Konstruktoren**.

Objektinitialisierung

- Eine **Initialisierung** der Felder sollte entweder **direkt bei deren Deklaration** oder durch einen Konstruktor stattfinden.
 - Direkte Initialisierung

```
class Student {  
    String vname  
    String nname  
    int matrnr =  
    int fb = 12;  
    int fachsemes  
    ...  
}
```

Spezielle Methoden, die nur bei Objekterzeugung ausgeführt werden.

```
s Konto {  
    String kontoNr = "12345";  
    double betrag = 5.0;  
    int kundenNr = 42;
```

- Damit bekommen alle Objekte der Klasse initial diese Werte.
 - Im Fall der Klasse Student ist dies nicht empfehlenswert, da typischerweise die Werte individuell gesetzt werden sollen.
 - Stattdessen empfiehlt sich die Benutzung von **Konstruktoren**.

Konstruktoren (1)

- Konstruktoren dienen der Initialisierung neu erzeugter Objekte.
 - Ähnlich zu einem Formular müssen dabei Angaben gemacht werden, um die Objekte zu erzeugen.
- In Java besitzen Konstruktoren den **Namen der Klasse**; ein **Ergebnistyp wird nicht angegeben**.
 - Beim Erzeugen von Objekten der Klasse mit new wird stets ein Konstruktor aufgerufen.
 - Dies ist die einzige Möglichkeit Konstruktoren zu nutzen.
 - Sie dienen nur dazu, einem neuen Objekt einer Klasse einen **initialen Zustand zu geben**.

Konstruktoren (2)

- Eine Klasse kann **keinen, einen oder mehrere unterschiedliche Konstruktoren** besitzen.
 - Sollte **kein Konstruktor** zur Verfügung gestellt werden, wird der **Default-Konstruktor** (parameterlos und mit leerem Rumpf) automatisch hinzugefügt.
 - Wird **mindestens ein Konstruktor** in der Klasse zur Verfügung gestellt, wird **kein Default-Konstruktor** hinzugefügt.
 - Diese Konstruktoren verfügen dann i. A. über Parameter, die zur Erzeugung der Objekte genutzt werden.

Beispiel eines Konstruktors

```
class Student {  
    String vorname;  
    String nachname;  
    int matrnr;  
    int fb;  
    int fachsemester = 1;  
  
    /** Ein Konstruktor der Klasse Stud.  
     */  
  
    Student(String v, String n, int mnr, int f) {  
        vorname = v;  
        nachname = n;  
        matrnr = mnr;  
        fb = f;  
    }  
}  
  
Student s = new Student("Max", "Mustermann", 12345, 12);
```

Ausgabe auf jshell:

s ==> Student@6b09bb57

Mehrere Konstruktoren

- In einer Klasse können mehrere Konstruktoren existieren.
 - Diese Konstruktoren müssen sich in Ihrer **Signatur** (Liste der Typen der Parametervariablen) unterscheiden.

```
class Student {  
    ...  
    Student(String v, String n, int mnr, int f, int fs) {  
        vorname = v;  
        nachname = n;  
        matrnr = mnr;  
        fb = f;  
        fachsemester = fs;  
    }  
    ...  
}
```

Initialisiertes Feld darf überschrieben werden, wenn es nicht final ist.

Standardwerte

- Datenfelder ohne explizite Initialisierung werden implizit mit einem Standardwert (genau wie bei Arrayelementen) initialisiert
- Auf Datenfelder darf daher bereits vor der ersten Zuweisung zugegriffen werden.

```
class Student {  
    ...  
    Student() {  
        System.out.println(  
            "Vorname: " + vorname + ",\n" +  
            "Nachname: " + nachname + ",\n" +  
            "Matrikelnummer: " + matrnr);  
    }  
    ...  
}
```

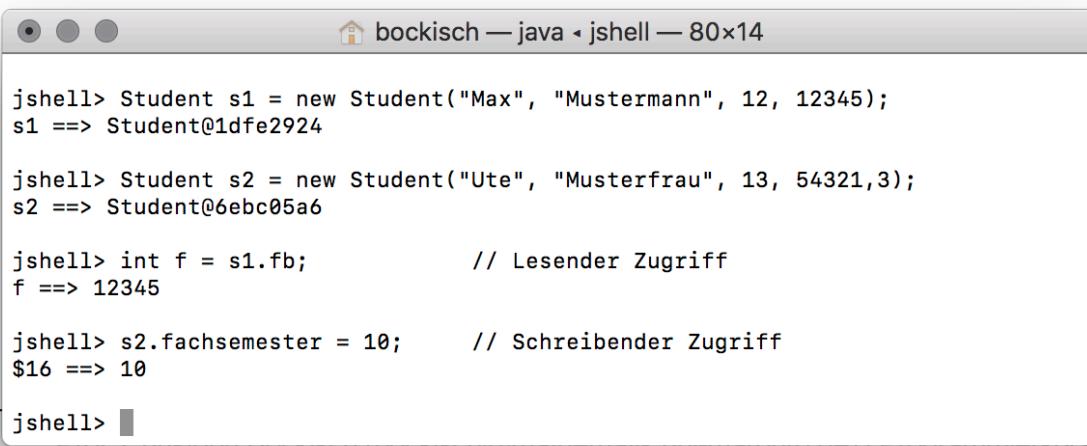
5.3 Zugriff auf Datenfelder

- Ist ein Objekt erzeugt und initialisiert worden, ist ein Zugriff auf die Datenfelder schreibend und lesend möglich.
 - Hierzu verwendet man typischerweise eine Variable gefolgt von einem Punkt und den Namen des Datenfelds.

```
Student s1 = new Student("Max", "Mustermann", 12, 12345);
Student s2 = new Student("Ute", "Musterfrau", 13, 54321,3);

int f = s1.fb;                      // Lesender Zugriff

s2.fachsemester = 10;                // Schreibender Zugriff
```



```
jshell> Student s1 = new Student("Max", "Mustermann", 12, 12345);
s1 ==> Student@1dfe2924

jshell> Student s2 = new Student("Ute", "Musterfrau", 13, 54321,3);
s2 ==> Student@6ebc05a6

jshell> int f = s1.fb;              // Lesender Zugriff
f ==> 12345

jshell> s2.fachsemester = 10;        // Schreibender Zugriff
$16 ==> 10

jshell>
```

Zugriff auf Felder: ja oder nein?

- Oft ist es **gefährlich**, Datenfelder der Objekte direkt zu verändern.
 - Dadurch kann ein Objekt mit einem nicht erlaubten Zustand entstehen.
 - Beispiel (Klasse Konto)
 - Ein direkter Zugriff auf das Datenfeld `kontoStand` könnte dazu führen, dass dies ohne **Überweisung, Einzahlung oder Abheben** geändert wurde.
- Zugriffsrechte **public** und **private**
 - **public** erlaubt wie bisher den uneingeschränkten Zugriff auf die Datenfelder.
 - **private** nur dann, wenn man sich in der Klasse, z. B. in einer Methode, befindet.
 - Tatsächlich gibt es noch mehr Optionen für die Zugriffsrechte, auf die wir später zu sprechen kommen.

[Ich bin hier](#)

Geschützte Datenfelder

- Im Allgemeinen sollten alle Datenfelder einer Klasse als **private** deklariert werden.

```
class Konto{  
    private String kontoNr;  
    private double kontoStand;  
    private int kundenNr;  
  
    public void einzahlen(double geld) {  
        ...  
    }  
    public double abheben(double wunschBetrag) {  
        ...  
    }  
    public double getKontoStand() {  
        ...  
    }  
}
```

- Damit ist der Zugriff auf die Datenfelder nur noch innerhalb der Klasse möglich.
 - Den Zugriff von außerhalb ermöglichen wir indirekt über die **public**-Methoden.

5.4 Objektmethoden

- Die Operationen des neuen Datentyps, der durch die Klasse bereitgestellt wird, werden durch Objektmethoden realisiert.
- Objektmethoden werden innerhalb des Klassenrumpfs definiert
 - Diese Methoden haben stets **Zugriff auf alle Datenfelder eines Objekts** als wären es lokale Variablen der Methode.

Beispiel

```
class Konto {  
    private String kontoNr;  
    private double kontoStand;  
    private int kundenNr;  
  
    public void einzahlen(double betrag) {  
        if (betrag > 0.0)  
            kontoStand += betrag;  
    }  
    public double abheben(double wunschBetrag) {  
        ...  
    }  
    public double getKontoStand() {  
        return kontoStand;  
    }  
}
```

Aufruf von Objektmethoden

- Objektmethoden können nur im **Kontext eines Objekts** genutzt werden.
 - Beim Aufruf wird **erst das Objekt dann ein Punkt und dann der Methodename** mit den Parametern angegeben.
 - Damit kann in der Methode auf alle **Datenfelder des Objekts** lesend und schreibend zugegriffen werden.
 - Beispiel

```
Konto k = new Konto("12345", 0.0, 7);  
// Hier wird ein neues Konto mit Nummer 12345 für Kunde mit  
// Kundennummer 7 erstellt. Der Kontostand ist am Anfang auf 0  
  
k.einzahlen(1000.0);  
// In der Methode einzahlen kann jetzt auf die Datenfelder des Kontos  
// zugriffen werden, auf das k verweist.  
.  
.  
.  
System.out.println("Aktueller Kontostand: " + k.getKontoStand());
```



5.5 Das Schlüsselwort this

Wie referenziere ich mich selbst?

- Problem
 - Parametervariablen einer und Datenfelder einer Klassen können den gleichen Namen haben. → Namenskonflikt
 - Wie kann man den Namenskonflikt auflösen?
- Lösung: Verwendung von **this**
 - Durch das Schlüsselwort **this** bekommt man die Referenz des **Objekts**, in dem man sich befindet.
 - **this** kann in der Klasse **wie eine Variable vom Typ der Klasse** verwendet werden.

Beispiel

- Anwendung von this in der Klasse Konto zur Auflösung von Namenskonflikten

```
class Konto {  
    private String kontoNr;  
    private double kontoStand;  
    private int kundenNr;  
  
    Konto(String kontoNr, double ks, int kundenNr) {  
        this.kontoNr = kontoNr;  
        kontoStand = ks;  
        this.kundenNr = kundenNr;  
    }  
    ...  
}
```

Zugriff auf das
Datenfeld

Zugriff auf die
Parametervariable

Selbstreferenz eines Objekts

- Problem
 - Wie kann ein Objekt in einer Methode eine Referenz auf sich selbst als Ergebnis liefern?
- Lösung: Verwendung von **this**
 - Durch das Schlüsselwort **this** bekommt man die **Referenz des Objekts**, in dem man sich befindet.
 - **this** kann als **Ergebnis einer Methode** nach außen geliefert werden.

Beispiel

- Anwendung von this in der Klasse Konto als Selbstreferenz

```
class Konto {  
    private double kontoStand;  
    ...  
  
    /** ToDo: Kommentar fehlt! */  
    Konto einzahlen (double betrag) { // neuer Rückgabetyp!!  
        kontoStand += betrag;  
        return this;  
    }  
    ...  
}
```

this liefert die Referenz
auf das Objekt

Selbstbezug bei Konstruktoren

- Problem
 - Konstruktoren sehen oft sehr ähnlich aus und unterscheiden sich oft nur in einem Parameter.
 - Kann bei der Bereitstellung von Konstruktoren auch ein anderer Konstruktor benutzt werden?
- Lösung: Verwendung von **this**
 - Durch das Schlüsselwort **this** kann in einem Konstruktor ein **anderer Konstruktor** der gleichen Klassen **aufgerufen werden**.
 - **this** wird dann **wie ein Methodenname** genutzt.
 - Nebenbedingung
 - Der **this**-Konstruktoraufruf muss die **erste Anweisung** in einem Konstruktor sein.

Beispiel

- Anwendung von this zum Aufruf eines anderen Konstruktors

```
class Konto {  
    private String kontoNr;  
    private double kontoStand;  
    private int kundenNr;  
  
    Konto(String kontoNr, double ks, int kundenNr) {  
        this.kontoNr = kontoNr;  
        kontoStand = ks;  
        this.kundenNr = kundenNr;  
    }  
  
    Konto(String kontoNr, int kundenNr) {  
        this(kontoNr, 0, kundenNr);  
    }  
}
```



5.6 Das Schlüsselwort static

- Das Schlüsselwort **static** kann vor
 - einer Methode,
 - einem Datenfeld,
 - und einem Initialisierungsblock einer Klasse
→ wird nicht in dieser Vorlesung besprochenstehen.
- Alle mit static gekennzeichneten Komponenten einer Klasse, sind **Bestandteile der Klasse**.
 - Diese Datenfelder und Methoden gehören nicht zu einem Objekt der Klasse.

static Datenfelder

- Manchmal werden Datenfelder benötigt, die unabhängig von den Objekten einer Klasse sind.
 - Für alle Objekte der Klasse sollen die Felder den gleichen Wert haben.
- In der Klasse Konto soll der dispo als statisches Feld gespeichert werden.
 - Diese Felder können durch das Schlüsselwort static definiert werden.

```
static double dispo = 5000.0;
```

- Oft handelt es sich dabei um Konstanten, weshalb static zusammen mit final benutzt wird.

```
static final double PI = 3.14;
```

- Der Zugriff von außerhalb der Klasse erfolgt durch den Namen der Klasse oder einem Objekt der Klasse:

Konto.dispo

static Methoden

- Klassenmethoden, die unabhängig von Objekten einer Klasse sind, werden ebenfalls mit dem Schlüsselwort static deklariert.
 - In diesen Methoden steht kein „this“ zur Verfügung.
 - Sie dürfen nur auf static Felder und Methoden der Klasse zugreifen.
- So enthält Java eine Klasse Math, in der nützliche mathematische Funktionen wie z. B. sin, cos, max, min, random, etc. als statische Methoden implementiert sind:

```
static double random() { ... };
```

- Der Aufruf der Methode erfolgt über den Klassennamen, ohne ein Objekt von Math zu erzeugen:

```
double zufall = Math.random();
```

Unterschied zwischen Klassenmethoden und Objektmethoden

- **Klassenmethoden**
 - Schlüsselwort static
 - Zugriff nur auf mit static-deklarierten Methoden und Datenfelder der eigenen Klasse möglich.
 - Aufruf (typischerweise) über den Klassennamen
 - Beispiel: Math.sqrt(2)
 - Genauso für Klassen-Felder: System.out
- **Objektmethoden** besitzen **nicht** das Schlüsselwort static.
 - Diese Methoden haben stets Zugriff auf **alle** Datenfelder und Methoden eines Objekts.
 - Aufruf einer Objektmethode erfolgt über ein Objekt
 - Beispiel: meinKonto.einzahlen(10000), out.println("Hallo OOP")
 - Genauso für Objekt-Felder: student.vorname



static und private Datenfelder

- Problem
 - Man möchte static-Datenfelder nutzen, aber den Gebrauch außerhalb der Klasse verbieten.
- Lösung
 - Dann ist es sinnvoll **zusätzlich den Modifier private** zu benutzen.

```
private static double dispo = 5000.0;
```

- Damit wird eine unkontrollierte Veränderung des Datenfelds verhindert, da nur **static-Methoden der Klasse** Zugriff auf das Datenfeld haben.

Geheimnis gelüftet - print

- Bisher haben wir folgende Methode zur Ausgabe einer Zeichenkette benutzt.

```
System.out.println("Hallo Welt");
```

- Was steckt dahinter?
 - **System** ist eine Klasse
 - System hat ein statisches Datenfeld **out**
 - Der Typ des Datenfelds out ist die **Klasse PrintStream**.
 - Die Klasse PrintStream hat Objektmethoden **print** und **println**.

5.7 Die main-Methode in Java

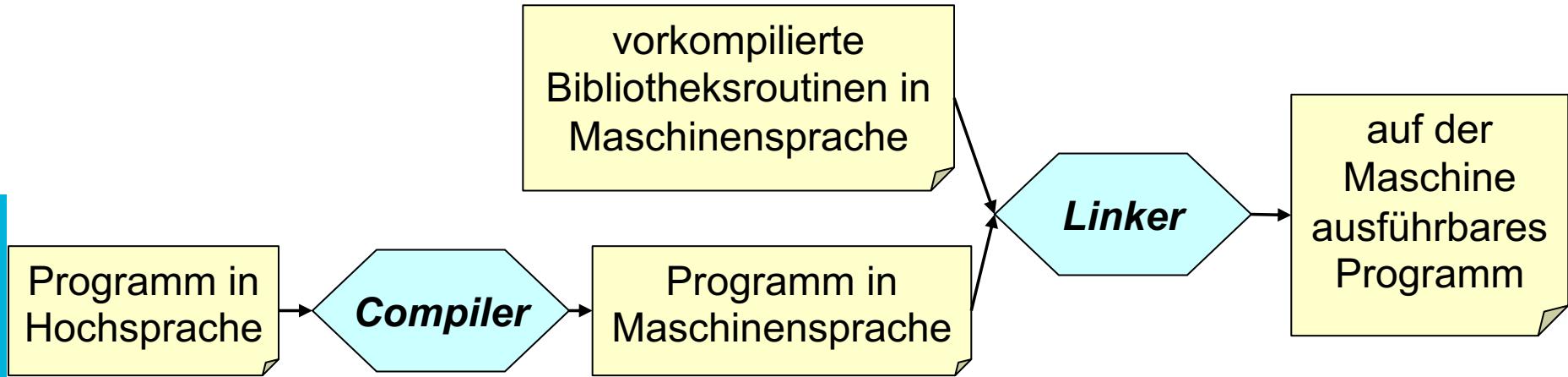
- Bisher haben wir in der Vorlesung jshell benutzt.
 - Vorteil: Schnelles und einfaches Erstellen von Java-Programmen
- Bevor es jshell gab, war die main-Methode der klassische Zugang zum ersten Java-Programm.

```
class Hallo {  
    public static void main(String[] args) {  
        System.out.println("Hallo Welt!");  
    }  
}
```

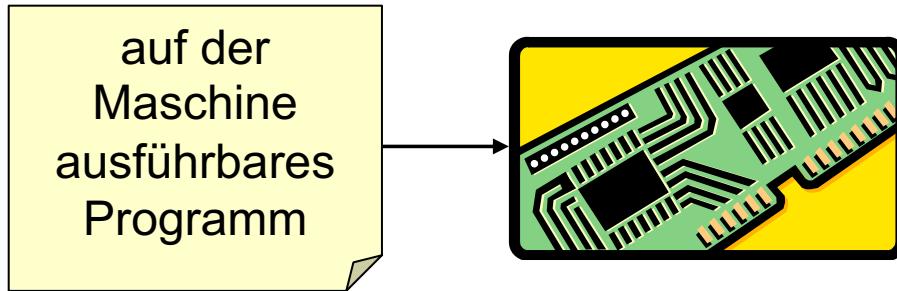
- Abspeicherung der Klasse in einer Datei mit dem Namen der Klasse und der Dateiendung "java".
 - In unserem Fall Hallo.java.

Der Weg zum ausführbaren C-Programm

- Zur „Übersetzungszeit“



- Zur „Laufzeit“



Vorteil:

Schnelle Ausführung des fertig übersetzten Programms zur Laufzeit!

Nachteil:

Nur auf einem Maschinentyp (Prozessortyp) ausführbar!

Jede Maschine hat andere Befehle (Ein Programm auf dem Pc läuft nicht auf Mac und umgekehrt)

Betriebssysteme stellen Werkzeuge bereit (Dateiverwaltung, Speicherverwaltung etc.), Programme rufen die Funktionen der Werkzeuge von Betriebssystemen auf (z.B. writefile, send etc.)

Am Ende ein Programm auf bestimmten Rechentyp mit einem bestimmten Betriebssystem (Pc mit Linux oder Mac mit MacOS)

Rechentyp mit Betriebssystem nennt man Plattform

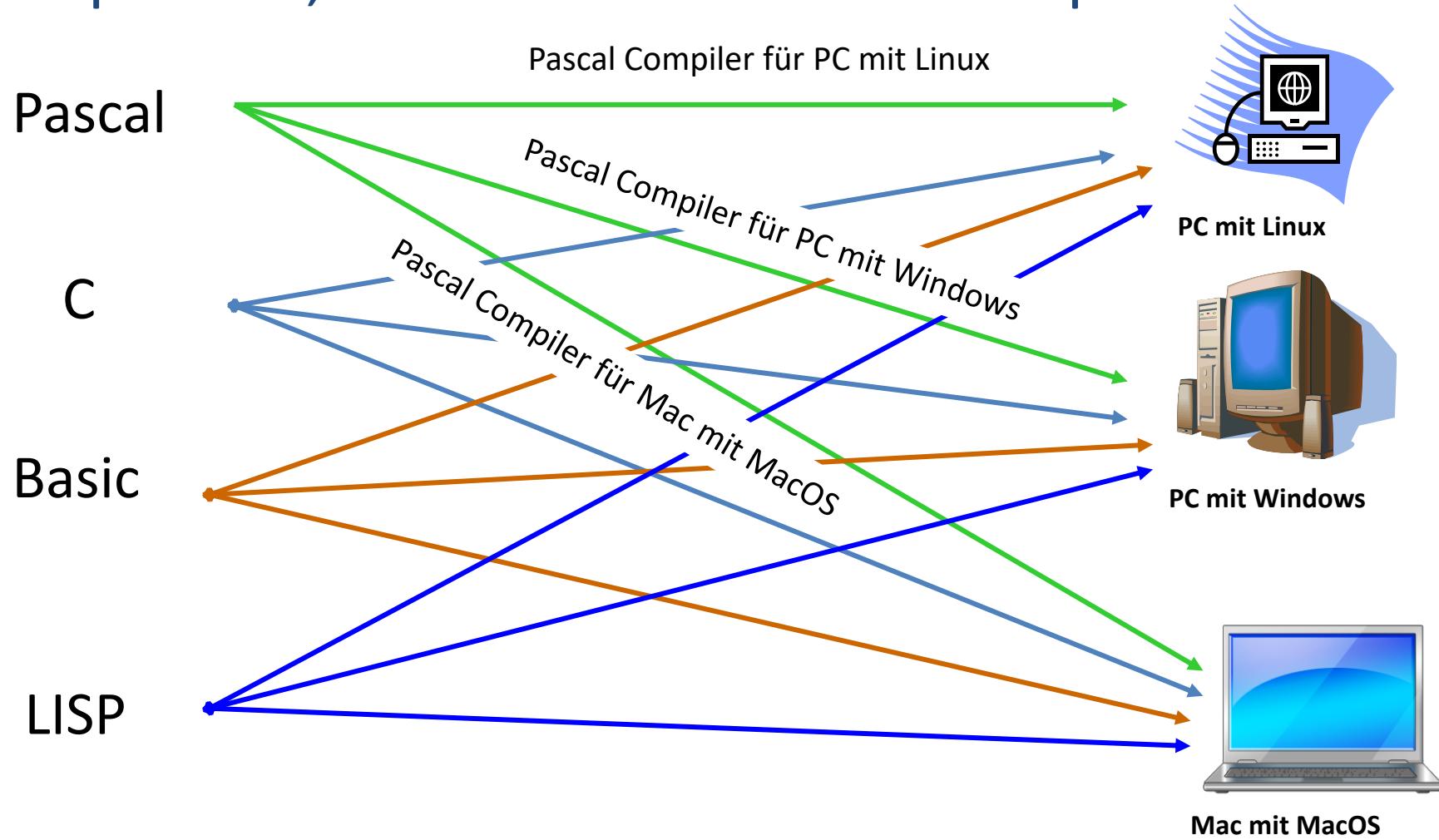
Maschinenabhängigkeit

- Jede Maschine hat andere Befehle
 - Ein Programm für einen **PC** läuft nicht auf dem **Mac** und umgekehrt
- **Betriebssysteme abstrahieren** von dem konkreten Rechner und stellen **Werkzeuge** bereit, die von Programmen genutzt werden können.
 - Dateiverwaltung, Speicherverwaltung, Prozessverwaltung, Input/Output, Hilfsprogramme...
- Programme rufen die **Funktionen** der Werkzeuge von Betriebssystemen auf.
 - writefile, print, readfile, send, receive, out, ...

Konsequenz:

- Jedes Programm läuft nur auf einem bestimmten Rechentyp mit einem bestimmten Betriebssystem
 - z.B. nur auf **PC** mit **Linux**, oder nur auf **Mac** mit **MacOS**
- Einen Rechentyp zusammen mit dem darauf laufenden Betriebssystem nennt man auch **Plattform**.

m Sprachen, n Plattformen $\rightarrow m \cdot n$ Compiler



Virtuelle Maschinen

- Eine Virtuelle Maschine ist ein gedachter Computer VM.
 - Eine VM bietet auch eine gedachte Maschinensprache an. Diese Sprache wird auch als *Bytecode* bezeichnet.
- VM wird auf jedem realen Computer emuliert.
- Für jede Sprache L benötigt man nur einen Compiler von L nach VM Maschinencode.

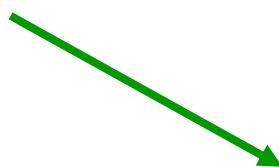
so wie ich hier verstehe VM für Windows , für Linux und für MacOs

Wir brauchen nur ein Compiler, von der Programmiersprache in die Maschinencode für die VM
wir hatten vorher 1 Sprache dafür brauchen wir 3 Compiler auf jede Plattform
Jetzt nur ein Compiler und VM auf jede Plattform

D.h. wenn wir mit BASIC programmieren , die Entwickler stellen die Programmiersprache bereit mit 3 verschiedenen Compiler für jede Plattform(Rechnertyp + Compiler)
Das Programm auf einem PC mit Windows laufen kann, wird nicht laufen auf eine PC mit linux laufen oder MacOS
Lösung ist die Virtuelle Maschiene

Virtuelle Maschine

Pascal



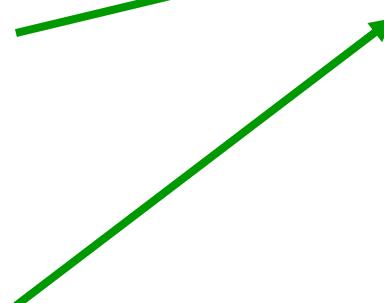
C



Basic



LISP



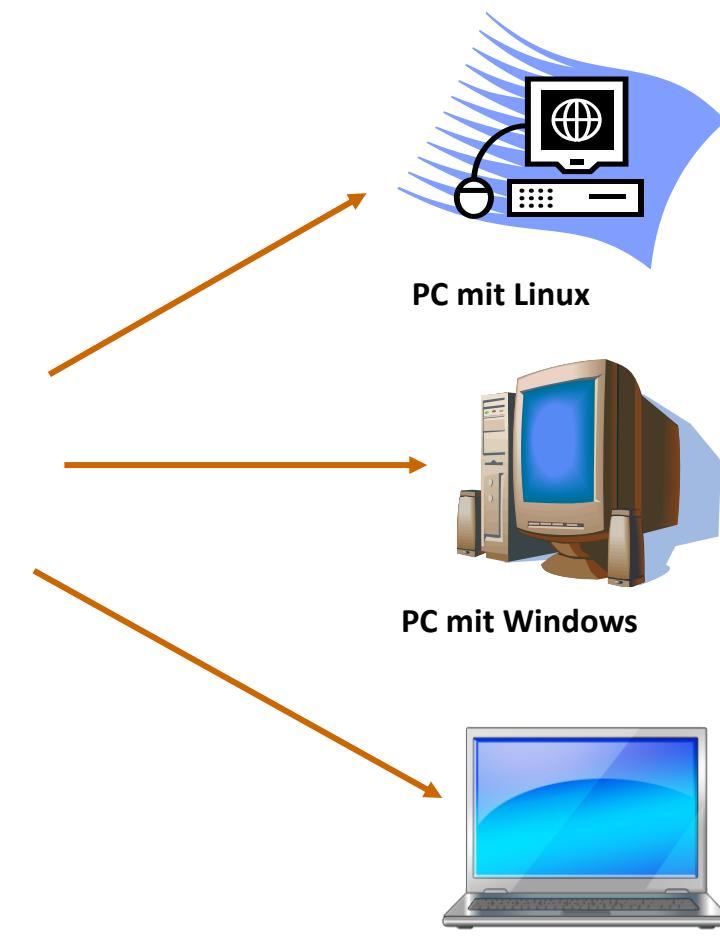
Virtuelle Maschine

M Sprachen



M Compiler, N Implementierungen der VM

N Plattformen



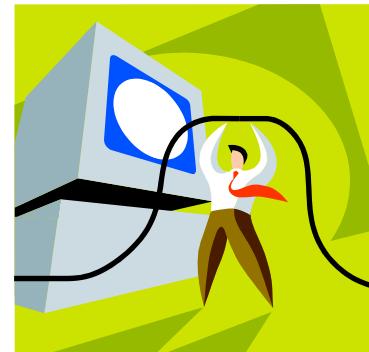
WORA: Write-Once-Run-Anywhere

- Durch die Verwendung einer VM als Zielsystem können lauffähige Programm portiert werden.
 - Entwicklung auf PC unter Windows
 - Ausführung eines Programms auf dem Mac unter Mac OS

Java Virtuelle Maschine

- Die Java-Runtime Engine ist eine virtuelle Maschine
 - die zunächst speziell für die Sprache **Java** entwickelt wurde.
- Sie ist auf fast allen Plattformen implementiert.
- Lässt sich in der Regel über das Kommando „**java**“ aufrufen

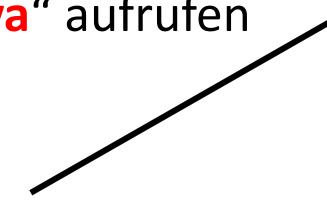
Ich verstehe hier es gibt JVM AUF jede Plattform



JVM
(Java Virtual Machine)



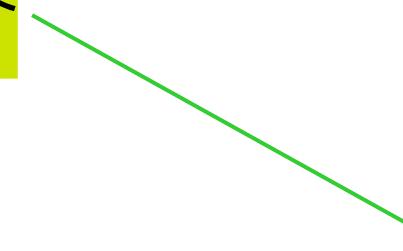
PC mit Linux



PC mit Windows

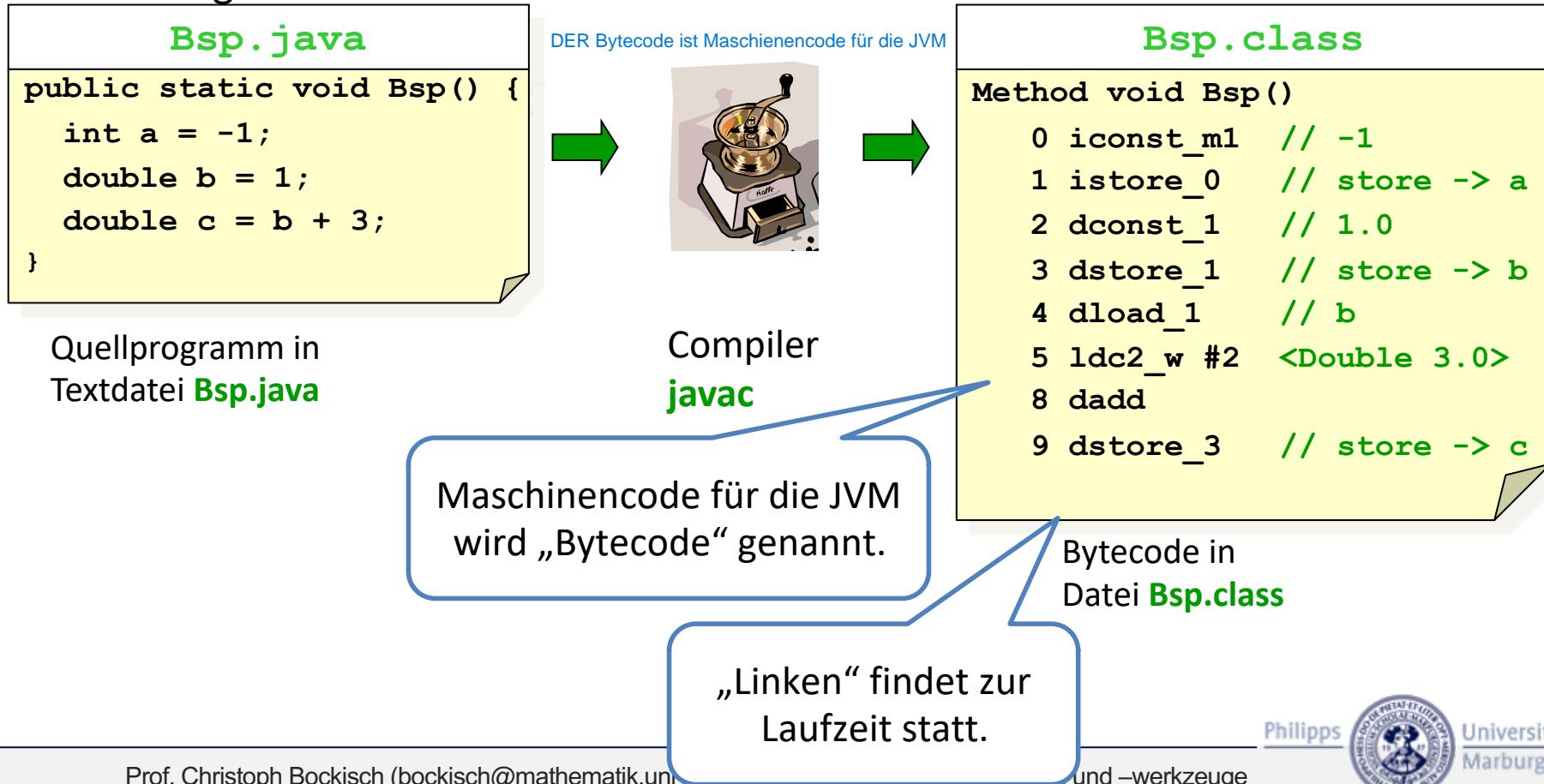


Mac mit MacOS



Übersetzen von Java-Programmen (1)

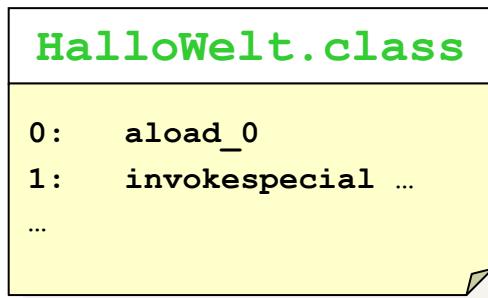
- Aus einer Textdatei mit der Endung **.java** erzeugt der Compiler **javac** eine Datei mit gleichem Namen, aber Endung **.class**
- Diese sogenannte class-Datei enthält den **Maschinencode** für die **JVM**



Ausführung

- Die **class**-Datei mit dem Bytecode wird der **JVM** übergeben.
 - Der Bytecode wird zur Laufzeit in Maschinencode der zugrundeliegenden Plattform übertragen.

der Bytecode(Maschinenkode für die JVM) wird zur Laufzeit in Maschinencode der Plattform umgewandelt



Java Virtual Machine
java



Unter der Haube der JShell

```
bockisch — java • jshell — 80x23
jshell> /l

1 : class Student {
    String vorname;
    String nachname;
    int matrnr;
    int fb;
    int fachsemester = 1;

    /** Ein Konstruktor der Klasse Student.
     */
    Student(String v, String n, int mnr, int f) {
        vorname = v;
        nachname = n;
        matrnr = mnr;
        fb = f;
    }
}
2 : Student s1 = new Student("Max", "Mustermann", 12, 12345
3 : s1.fachsemester = 10;
4 : int f = s1.fb;

jshell>
```

Student.java

```
class Student {
    String vorname;
    String nachname;
    int matrnr;
    int fb;
    int fachsemester = 1;

    /** Ein Konstruktor der Klasse Stud.
     */
    Student(String v, String n, int mnr, int f) {
        vorname = v;
        nachname = n;
        matrnr = mnr;
        fb = f;
    }
}
```

Temp.java

```
class Temp {
    public static void main(String[] args) {
        Student s1 = new Student("Max",
            "Mustermann", 12, 12345);
        s1.fachsemester = 10;
        int f = s1.fb;
    }
}
```

Unter der Haube der JShell

```
bockisch — java · jshell — 80x23
jshell> /l

1 : class Student {
2     String vorname;
3     String nachname;
4     int matrnr;
5     int fb;
6     int fachsemester
7
8     /** Ein Konstruktor */
9     Student(String v, String n, int m, int f, int fs) {
10        vorname = v;
11        nachname = n;
12        matrnr = m;
13        fb = f;
14        fachsemester = fs;
15    }
16
17    public void setFachsemester(int fs) {
18        fachsemester = fs;
19    }
20
21    public int getFachsemester() {
22        return fachsemester;
23    }
24
25    public String getName() {
26        return vorname + " " + nachname;
27    }
28
29    public String getMatrnr() {
30        return String.valueOf(matrnr);
31    }
32
33    public String getFb() {
34        return String.valueOf(fb);
35    }
36
37    public String getFachsemesterString() {
38        return String.valueOf(fachsemester);
39    }
40
41    public void print() {
42        System.out.println("Name: " + getName());
43        System.out.println("Matrnr: " + getMatrnr());
44        System.out.println("FB: " + getFb());
45        System.out.println("Fachsemester: " + getFachsemesterString());
46    }
47
48    public static void main(String[] args) {
49        Student s1 = new Student("Mustermann", "Musterfrau", 12, 12345, 10);
50        s1.print();
51        int f = s1.fb;
52    }
53}
```

Student.java

Klassen werden behandelt als ob sie in einer eigenen .java-Datei stehen.

```
class Student {  
    String vorname;  
    String nachname;  
    int matrnr;  
    int fb;  
    int fachsemester = 1;  
  
    /** Ein Konstruktor der Klasse Stud.  
     */  
}
```

JShell führt aus:
javac Student.java
javac Temp.java
java Temp

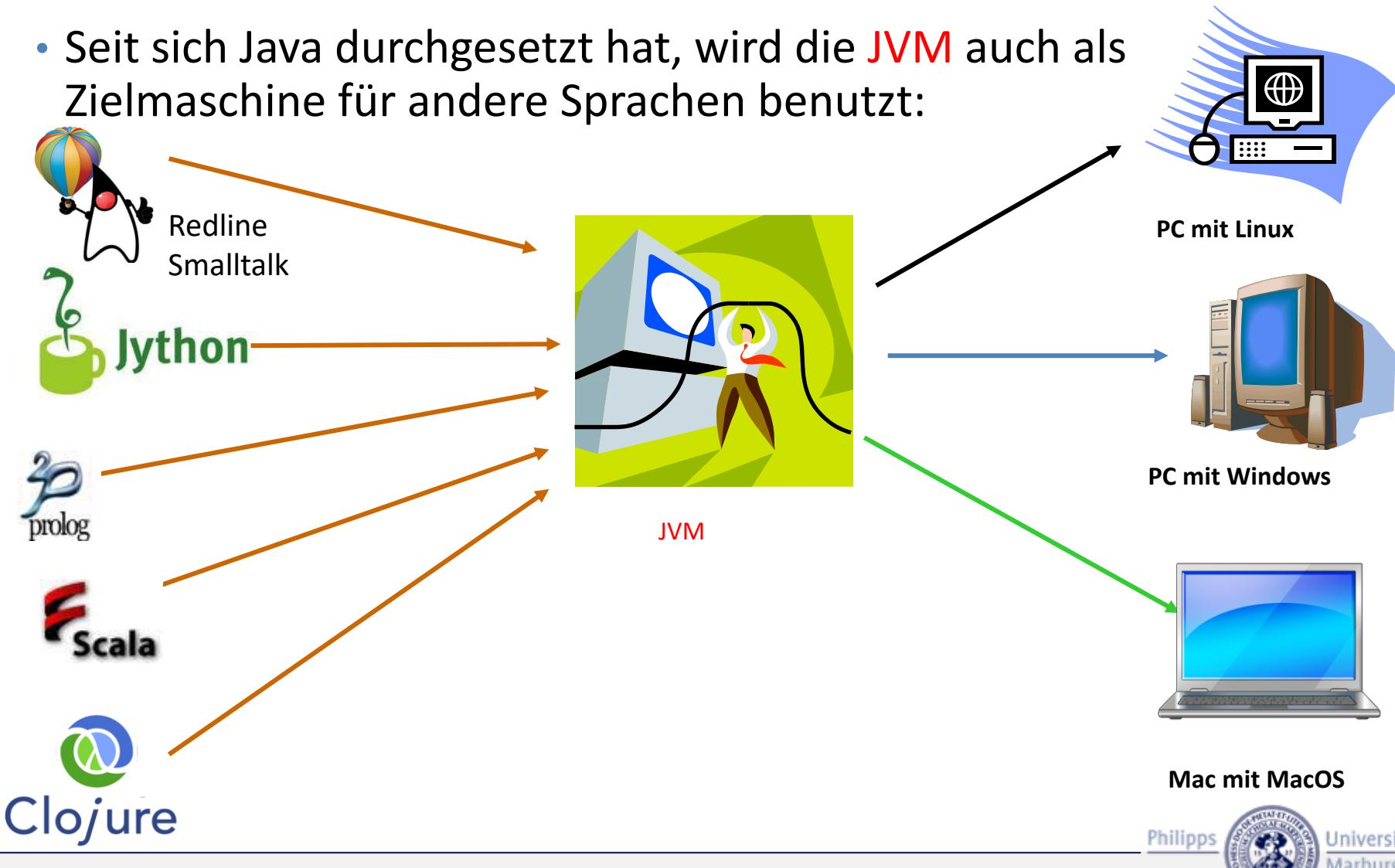
Alle direkt eingegebenen Befehle werden behandelt als ob sie in der main-Methode einer eigenen Java-Klasse stehen.
Gegebenenfalls werden Semikolons am Ende der Anweisungen ergänzt.

Alles gab es schon mal ...

- Virtuelle Maschinen für
 - **eine** Sprache und
 - **multiple** Plattformengab es schon früher, aber sie haben sich nicht durchgesetzt.
- Früher existierende virtuelle Maschinen für verschiedene Sprachen:
 - Pascal : p-Maschine (Anfang der 80-er Jahre)
 - Smalltalk : Smalltalk Bytecode Interpreter
- **Smalltalk**
 - Vorläufer von Java
 - teilweise mit besserer Umsetzung der Objektorientierung im Vergleich zu Java
 - Konzepte waren ihrer Zeit viele Jahre voraus ...
 - ... leider auch den damaligen Möglichkeiten der Hardware.

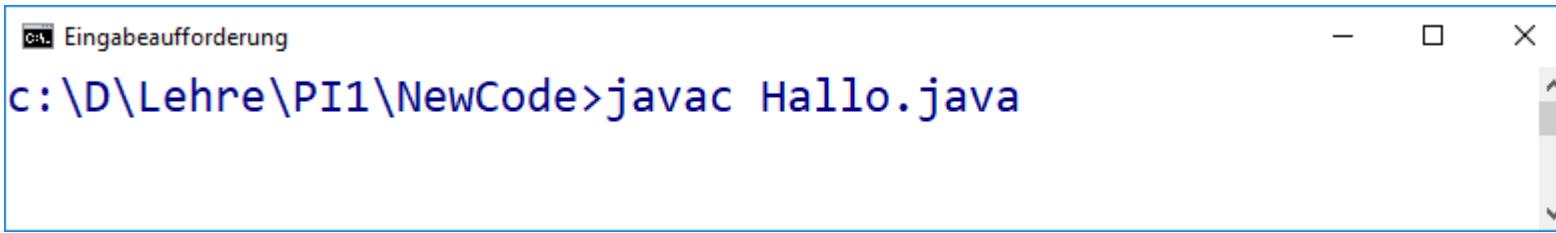
Erfolg steckt an ...

- Seit sich Java durchgesetzt hat, wird die **JVM** auch als Zielmaschine für andere Sprachen benutzt:



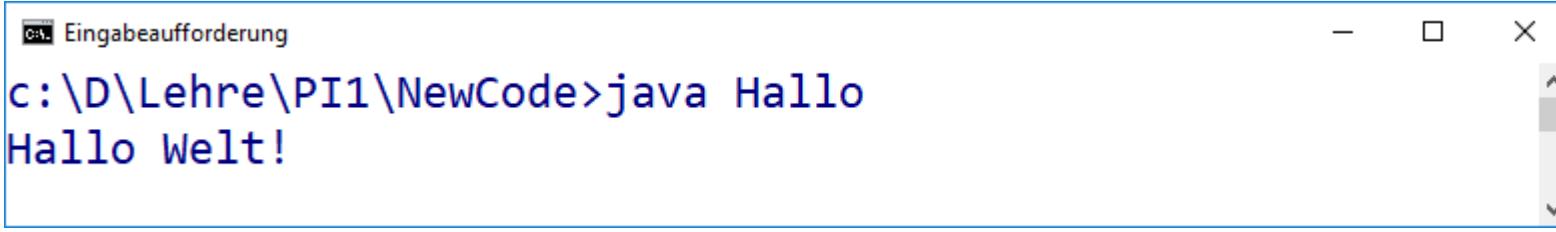
Übersetzen und Ausführen in der Kommandokonsole

- Übersetzen des Programms durch Aufruf des java-Compilers in cmd.



```
c:\D\Lehre\PI1\NewCode>javac Hallo.java
```

- In dem Verzeichnis wurde die Datei Hallo.class erzeugt.
- Ausführen des Programms durch Aufruf von java.



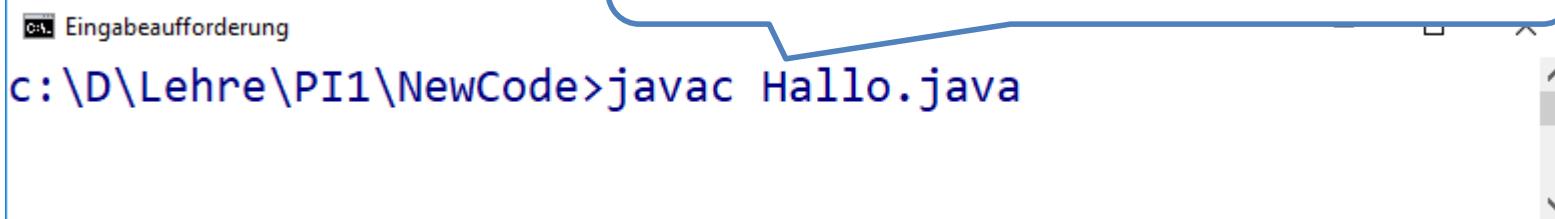
```
c:\D\Lehre\PI1\NewCode>java Hallo
Hallo Welt!
```

- Die Java-Laufzeitumgebung startet das Programm Hallo mit dem Aufruf der main-Methode.

Übersetzen und Ausführen in der Kommandokonsole

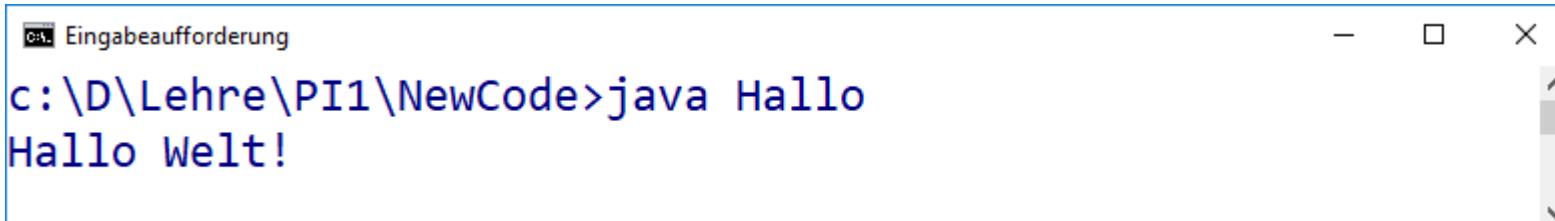
- Übersetzen des Programms mit dem Java-Compiler in cmd.

Die .java-Dateien werden in einem einfachen Text-Editor (später: IDE) geschrieben.



```
c:\D\Lehre\PI1\NewCode>javac Hallo.java
```

- In dem Verzeichnis wurde die Datei Hallo.class erzeugt.
- Ausführen des Programms durch Aufruf von java.



```
c:\D\Lehre\PI1\NewCode>java Hallo
Hallo Welt!
```

- Die Java-Laufzeitumgebung startet das Programm Hallo mit dem Aufruf der main-Methode.

Aufbau der main-Methode

```
public static void main(String[] args) {  
    ...  
}
```

- Was steckt hinter der main-Methode?
 - Die Methode ist **öffentlich nutzbar**.
 - Die Methode main ist eine **statische Methode**.
 - Die Methode liefert **kein Ergebnis**.
 - Die Methode besitzt eine **Parametervariable vom Typ String[]**.
 - Damit kann man beim Aufruf des Programms beliebig viele Parameter vom Typ String der main-Methode übergeben.

Parameter args in main

- Folgendes Programm gibt alles aus, was in der Kommandozeile nach dem Programmnamen kommt.

```
public class Echo {  
    public static void main (String[] args) {  
        for (String s: args)  
            System.out.println(s);  
    }  
}
```

Kommandozeilen-Parameter werden zu Elementen des Array-Parameters von main.

- Beispiel

```
Eingabeaufforderung  
c:\D\Lehre\PI1>NewCode>java Echo  
Grünkohl  
mit  
Mettwurst  
und  
Salzkartoffeln
```

Eingabe von Zahlen

- Das Programm Euklid zur Berechnung des größten gemeinsamen Teilers soll zwei Zahlen übergeben bekommen.
- Folgender Aufruf

java Euklid 152343 7439823

gibt folgende Ausgabe:

ggt von 152343 und 7439823 ist gleich 9.

- Erforderlich ist dabei die Umwandlung von String nach int.
 - Statische Methode `parseInt(String s)` aus der **Klasse Integer**
 - Falls die Zeichenkette keine Zahl als Parameter hat, bekommt man eine **Fehlermeldung**, eine sogenannte **Exception**, geliefert.

Eingabe von Zahlen

- Das Programm Euklid zur Berechnung des größten gemeinsamen Teilers soll zwei Zahlen übergeben bekommen.
- Folgender Aufruf

java Euklid 152343 7439823

gibt folgende Ausgabe:

ggt von 152343 und 7439823 ist gleich 9.

- Erfordert

- Static

- Falls

eine

```
public static void main(String[] args) {  
    int x = Integer.parseInt(args[0]);  
    int y = Integer.parseInt(args[1]);  
    System.out.println("ggt von " + x + " und " + y + " ist " + ggt(x, y));  
}
```

5.8 JavaDoc Kommentare für Klassen

- Spezielle Tags mit dem Präfix @ in Kommentaren für Klassen.
 - Allgemein verwendbare Tags
 - @author für Namen des Autors
 - @version für die Version der Klasse/Methode
 - @see für Verweise

Lesbarkeit von Programmen

- Die Kommentare für javadoc dienen primär den **Benutzern von Klassen**, um die Klasse korrekt anzuwenden.
 - Die Kommentare werden im Normalfall nur für public Klassen, Methoden und Felder erzeugt.
 - Trotzdem auch private Klassen, Methoden und Felder kommentieren!
- Aktualisierte Coding Conventions: siehe ILIAS

Zusammenfassung

- Klassen in Java
 - Definition eigener Datentypen
 - Wertemenge
 - Operationen
 - Verwendung von Klassen
 - Klassen als Datentypen
 - Klassen als Objektfabriken
- Konzepte von Klassen
 - Datenfelder und Methoden
 - Konstruktoren
- Schlüsselwort static



6. Übersetzungseinheiten und Pakete

- Themen des Kapitels
 - Ausführbare Java-Programme
 - Übersetzungseinheiten und Gesamtprogramm
 - Laden von Klassen
 - Der Speicherbereich „Method Area“
 - Pakete (Packages)
 - Standard-Pakete
 - Erstellung eigener Pakete
 - Umgebungsvariable Classpath

Ausführbare Java-Programme

- Java-Programme bestehen nur aus Klassen und Interfaces. Auch ausführbare Programme werden in Java mithilfe von Klassen definiert.
- Eine Klasse kann als Programm gestartet werden, wenn sie eine spezielle Klassenmethode namens main mit der Signatur

```
public static void main(String[] args)
```

enthält.

- Übersetzungseinheit
 - Eine zu kompilierende Java-Datei ist eine Textdatei mit dem Suffix .java. Diese wird auch als Übersetzungseinheit bezeichnet.
 - Sie kann den Quelltext einer oder mehrerer Java-Klassen enthalten.
- Bei erfolgreicher Übersetzung mit einem Java-Compiler entstehen in dem Verzeichnis, das die Übersetzungseinheit enthält, Dateien mit dem Suffix .class, und zwar eine pro übersetzter Klasse.

Ausführbare Java-Programme

- Java-Programme bestehen nur aus Klassen und Interfaces. Auch ausführbare Programme werden in Java mit von Klassen definiert.
- Eine Klasse kann als Programm gestartet werden, wenn sie eine spezielle Klassenmethode namens main mit der

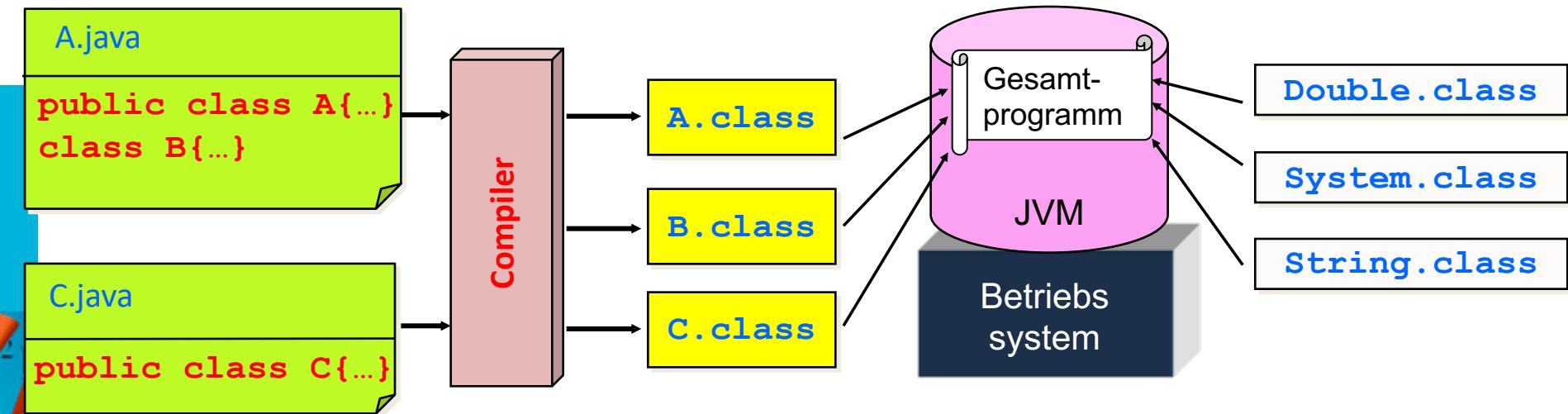
Lernen wir nächste Woche kennen.

```
public static void main(String[] args)
```

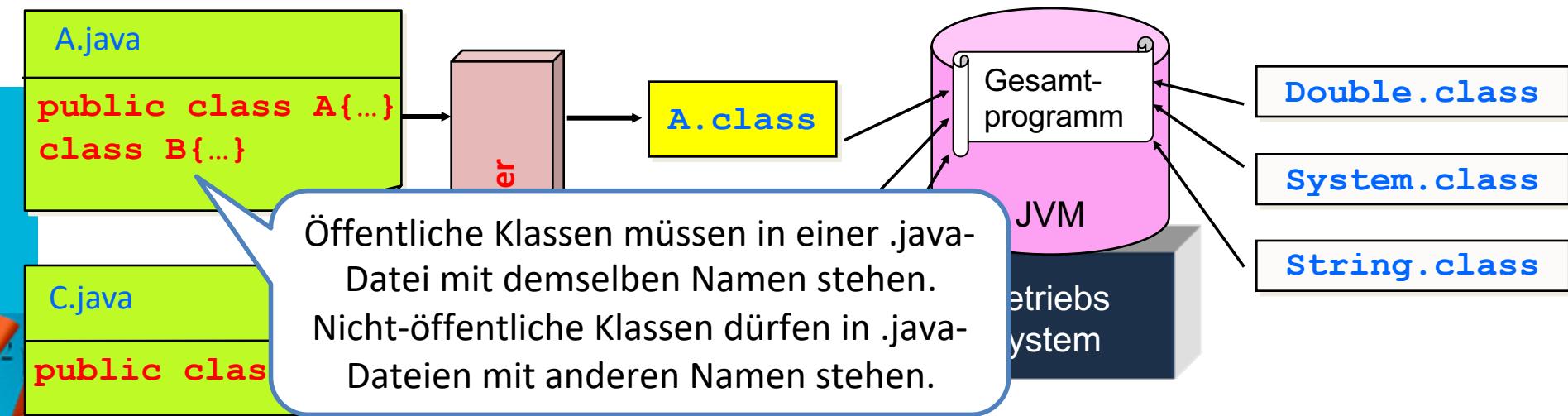
enthält.

- Übersetzungseinheit
 - Eine zu kompilierende Java-Datei ist eine Textdatei mit dem Suffix .java. Diese wird auch als Übersetzungseinheit bezeichnet.
 - Sie kann den Quelltext einer oder mehrerer Java-Klassen enthalten.
- Bei erfolgreicher Übersetzung mit einem Java-Compiler entstehen in dem Verzeichnis, das die Übersetzungseinheit enthält, Dateien mit dem Suffix .class, und zwar eine pro übersetzter Klasse.

Übersetzung und Ausführung von Java-Programmen



Übersetzung und Ausführung von Java-Programmen

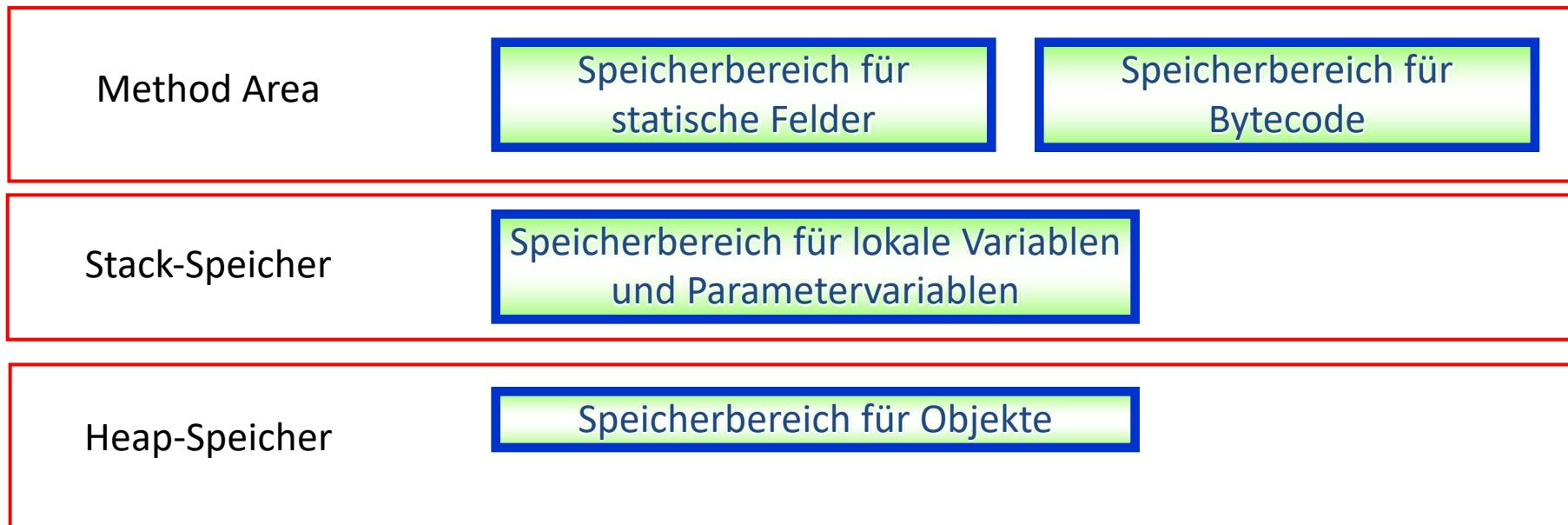


Entstehung des Gesamtprogramms

- Die Ausführung von Java-Programmen beginnt mit einer Klasse, die eine wie oben spezifizierte Methode **main** enthält.
 - Diese wollen wir **Hauptprogramm-Klasse** nennen.
- Die Hauptprogramm-Klasse kann andere Klassen benutzen, jene wieder andere Klassen etc.. Dies passiert erstmals durch:
 - Aufruf von statischen Methoden, Verwendung von statischen Datenfeldern und Aufruf von Konstruktoren
 - z. B. System.out
- Das **Gesamtprogramm** besteht aus der **Hauptprogramm-Klasse** und allen direkt oder indirekt **benutzten Klassen**.
 - Wenn eine Klasse in einem Programm zum **ersten Mal angesprochen** wird, wird sie zum Gesamtprogramm hinzugefügt. Man sagt dann, dass die Klasse **geladen** wird.
 - Der Zeitpunkt des Ladens kann auch tatsächlich etwas früher sein, aber das spielt aber für unsere Vorlesung keine Rolle.

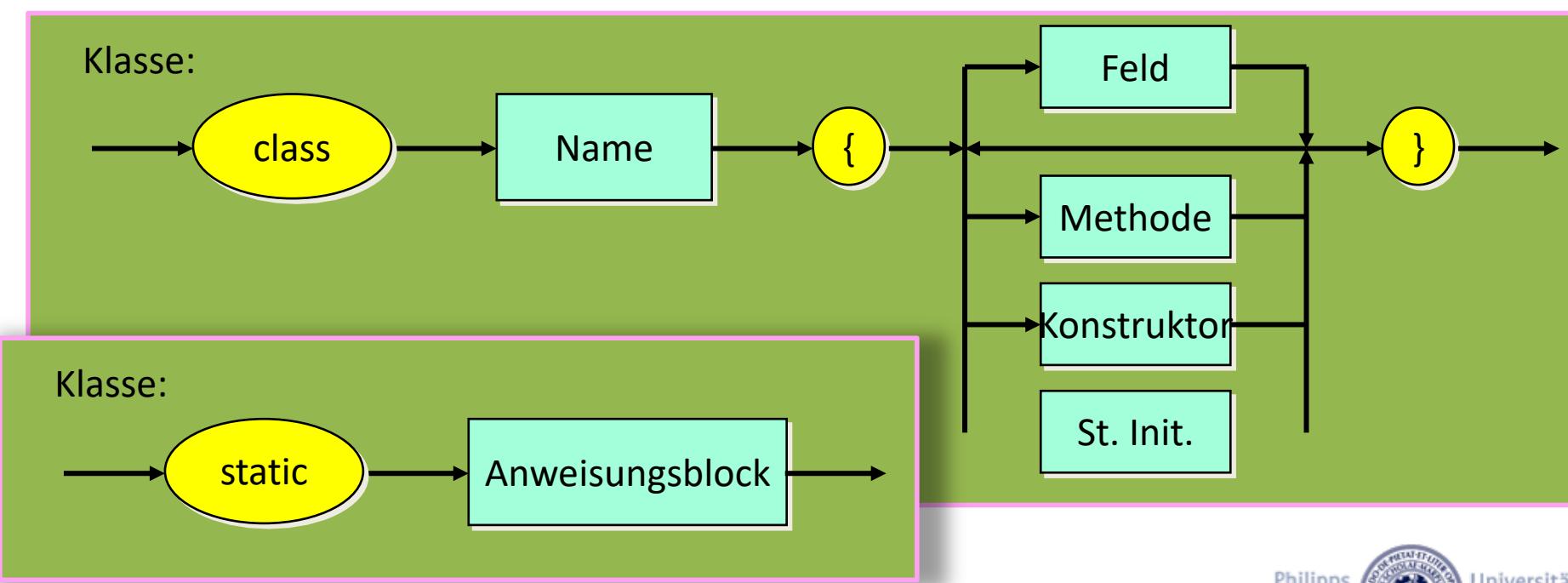
Initialisierung von Klassen

- Beim Laden einer Klasse wird ein **Speicherbereich für die statischen Felder** angelegt (und eine **Initialisierung der Klasse** vorgenommen).
 - Es werden noch weitere Informationen zur Klasse gespeichert.
- Zusätzlich zu dem Stack-Speicher und dem Heap-Speicher gibt es einen eigenen Speicherbereich (**Method Area**) für die Daten, die beim Laden einer Klasse angelegt werden.



Initialisierung von Klassen

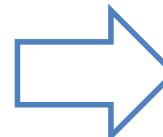
- Übrigens: der "Static Initializer" einer Klasse ist so etwas wie eine Methode, die ausgeführt wird, wenn eine Klasse geladen wird.
 - Spezielle Syntax



Initialisierung von Klassen

- Static Initializer

```
class Main {  
    static {  
        System.out.println("Laden von Klasse Main");  
    }  
    public static void main(String[] args) {  
        System.out.println("In Klasse Main");  
        A a = new A();  
        a.m();  
    }  
}  
  
class A {  
    static {  
        System.out.println("Laden von Klasse A");  
    }  
    public A() {  
        System.out.println("Konstruktor von Klasse A");  
    }  
    public static void m() {  
        System.out.println("Methode m von Klasse A");  
    }  
}
```



Laden von Klasse Main
In Klasse Main
Laden von Klasse A
Konstruktor von Klasse A
Methode m von Klasse A

Klassen, Klassen, Klassen, ...

- Durch die vielen Klassen (und Interfaces), die man typischerweise für das Gesamtprogramm erstellt, verliert man leicht den Überblick.
 - Wo finde ich passende Methoden?
 - Welche Klassen (und Interfaces) passen inhaltlich zueinander?



- Für die Erstellung großer Programme ist deshalb eine bessere Strukturierung hilfreich.

Klassen, Klassen, Klassen, ...

- Klassenname: sollte das Konzept beschreiben, das die Klasse implementiert
- Was passiert, wenn zwei unterschiedliche Klassen durch denselben Namen beschrieben werden?
 - Zum Beispiel, weil Sie verschiedene Aspekte des Konzepts umsetzen
 - „Spieler“ in einem Online-Spiel: 1. als Akteur, 2. Speicherung der Nutzerdaten?
 - Zum Beispiel, weil der Name mehrere Bedeutungen hat
 - „Feder“ als Produkt in einem Shop werden: 1. Schreibfeder, 2. Druckfeder
 - Zum Beispiel, weil eine wiederverwendete Implementierung einen generischen Namen wie „Setup“ verwendet



Pakete - Packages

- **Pakete** (engl.: packages) sind **Zusammenfassungen** von Java-Klassen für einen bestimmten Zweck oder einen bestimmten Typ von Anwendungen.
 - Dies hilft uns unsere eigenen Klassen besser zu strukturieren.
 - Funktionalität aus anderen Klassen leichter zu finden.
- Pakete sind **Namensräume**!
 - Klassennamen müssen innerhalb eines Pakets eindeutig sein, dürfen sich aber in anderen Paketen wiederholen
 - Der „voll qualifizierte“ (vollständige) Klassename wird folgendermaßen gebildet:
Paketname.Klassename
- Zum Beispiel
 - logik.Spieler und daten.Spieler
 - schreibwaren.feder und eisenwaren.feder
 - meinprogramm.Setup und wiederverwendet.Setup

Pakete - Packages

- Java-Programme können vordefinierte Standard-Pakete benutzen, die auf jedem Rechner mit einer Java-Ausführungsumgebung zu finden sein müssen.
- Das wichtigste Paket, `java.lang` enthält alle Klassen, die zum Kern der Sprache Java gezählt werden.
- Daneben enthält das JDK eine Unmenge weiterer Pakete, wie z.B.
 - `javax.swing` mit Klassen zur Programmierung von Fenstern und Bedienelementen,
 - `java.net` mit Klassen zur Netzwerkprogrammierung,
 - `java.util` mit nützlichen Klassen wie `Calendar`, `Date`, `ArrayList` und Schnittstellen `Collection`, `List`, `Iterator`, etc.

Das Paket `java.lang`

- Kern dieser Standard-Pakete ist `java.lang` (lang steht für language.)
 - Dieses Paket enthält die wichtigsten vordefinierten Klassen, wie z. B. `System`, `String`.
 - Zusätzlich gibt es die „Wrapper-Klassen“ für die primitiven Datentypen: `Boolean`, `Character`, `Integer`, `Short`, `Byte`, `Long`, `Float` und `Double`.
 - Die Klasse `Math` enthält mathematische Standardfunktionen wie z.B. `sin`, `cos`, `log`, `abs`, `min`, `max`, etc. und mathematische Konstanten `E` und `PI`.
- Die Klassen aus `java.lang` können direkt verwendet werden.
- Für alle anderen Standard-Pakete muss man entweder `import-Anweisungen` hinzufügen, oder man muss alle Namen (inkl. des Paketnamens) voll ausschreiben.

Import von Standardpaketen

- Um eine Klasse eines Paketes anzusprechen kann man grundsätzlich die Paketadresse dem Namen voranstellen, etwa

```
double[] arr = java.util.Arrays.copyOf(meinArray, 4711);  
java.util.Random r = new java.util.Random();
```

- Einfacher ist es, wenn man eine sogenannte **import-Anweisung** benutzt.
- Damit können eine oder alle Klassen eines Paketes **direkt angesprochen** werden, ohne ihnen jeweils die Paket-Adresse voranzustellen.
 - Schickt man also eine Import-Anweisung wie

```
import java.util.Random;
```

voraus, so kann man anschließend Objekte und Methoden der Klassen Calendar benutzen, ohne den vollen Pfad zu ihnen anzugeben:

```
Random r = new Random();
```

- Die Namen von Standardpakete beginnen grundsätzlich mit
 - java**, oder **javax**
- Die zugehörigen Dateien liegen im Java-System.
 - Compiler und Interpreter wissen wie man diese Dateien findet.

Einfach, aber nicht empfehlenswert

- Um alle Klassen eines Pakets zu importieren, kann man das Zeichen „*“ als **Wildcard** einsetzen. So werden z.B. mit

```
import java.util.*;
```

sämtliche Klassen des Pakets **util** importiert oder genauer gesagt, werden die im Programm angesprochene Klassen **lediglich an den bezeichneten Stellen aufgesucht**.

- Nachteil dieser Variante
 - Man erkennt nicht mehr ohne weiteres die Abhängigkeiten seiner Klasse mit anderen Klassen, was aber bei der Fehlersuche in großen Programmen sehr nützlich ist.
- Seit Java 1.5 gibt es als zusätzliche Möglichkeit die **import-static-Anweisung**. Diese ermöglicht den unqualifizierten Zugriff auf statische Felder und Methoden einer Klasse.
- Man sollte diese Möglichkeit aber auch **mit Vorsicht** einsetzen. Ein kurzes Beispiel:

```
import static java.lang.Math.*;
import static java.lang.System.*;
class TestImportStatic {
    public static void main(String[] args) {
        out.println(sqrt(PI + E));
    }
}
```

2.420717761749361

Einfach, aber nicht empfehlenswert

- Um alle Klassen eines Pakets zu importieren, kann man das Zeichen „*“ als **Wildcard** einsetzen. So werden z.B. mit

```
import java.util.*;
```

- sämtliche Klassen des Pakets „java.util“ importiert.
- Nachteil:
 - Man kann wieder zu Namenskonflikten kommen.
Was aber bei der Fehlerbehandlung sehr nützlich ist.
- Seit Java 7 ist es möglich, mit **import-static** Namenskonflikte mit lokalen Definitionen zu verhindern. Z.B. könnte ein Feld „out“ in einer Klasse `TestImportStatic` nicht mehr unterschieden werden.
- Man sollte jedoch gesagt, werden die im ezeichneten Stellen aufgesucht.



```
import static java.lang.Math.*;
import static java.lang.System.*;
class TestImportStatic {
    public static void main(String[] args) {
        out.println(sqrt(PI + E));
    }
}
```

2.420717761749361



Beispiel: Scanner-Klasse aus java.util

- Seit dem JDK 1.5 wird die Klasse **Scanner** angeboten, mit deren Methoden man auf einfache Weise Daten einfacher Typen vom Konsolenfenster einlesen kann.
 - Methoden: String next(), int nextInt(), double nextDouble(),
- Dieses kleine Testprogramm kann von der Kommandozeile gestartet werden.

```
import java.util.Scanner;
public class TestScanner {
    public static void main (String[] args){
        Scanner s = new Scanner(System.in);
        System.out.print("Dein Vorname bitte: ");
        String name = s.next();
        System.out.println("Hallo " + name + "\nWie alt bist Du ?");
        int value = s.nextInt();
        if(value > 30)
            System.out.println("Hallo alter Hase");
        else
            System.out.println("Noch lange bis zur Rente ...");
        s.close();
    }
}
```

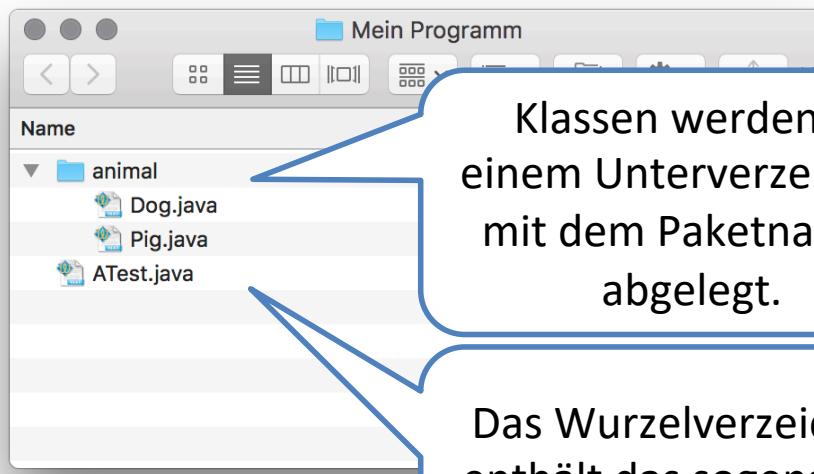
Erstellung eigener Pakete

- Um selber Pakete zu erzeugen, kann man veranlassen, dass die Klassen einer Java-Datei zu einem bestimmten Paket gehören sollen.
- Dies erreicht man mit der **package-Anweisung** am Anfang der Übersetzungseinheit:
 - `package meinPaketName;`

```
package animal;  
public class Pig{ ... }
```

```
package animal;  
public class Dog{ ... }
```

...



Klassen werden in
einem Unterverzeichnis
mit dem Paketnamen
abgelegt.

Wurzelverzeichnis des
Projekts nicht des
Dateisystems.

Das Wurzelverzeichnis
enthält das sogenannte
„**Default Package**“.

Bei Klassen in diesem
Package darf keine package-
Anweisung angegeben sein.

Verwendung eigener Pakete

- Sollen Klassen, Methoden und Felder aus anderen Paketen benutzt werden, müssen diese als **public** deklariert sein.
- Es soll ein Testprogramm erstellt werden, das die Klasse **Dog** unseres Pakets **animal** nutzt.
 - Dieses Testprogramm kann in irgendeinem Verzeichnis sein. Dieses wird damit zum **Default-Package**.

```
import animal.Dog;
public class ATest {
    public static void main(String args[]) {
        Dog a = new Dog();
    }
}
```

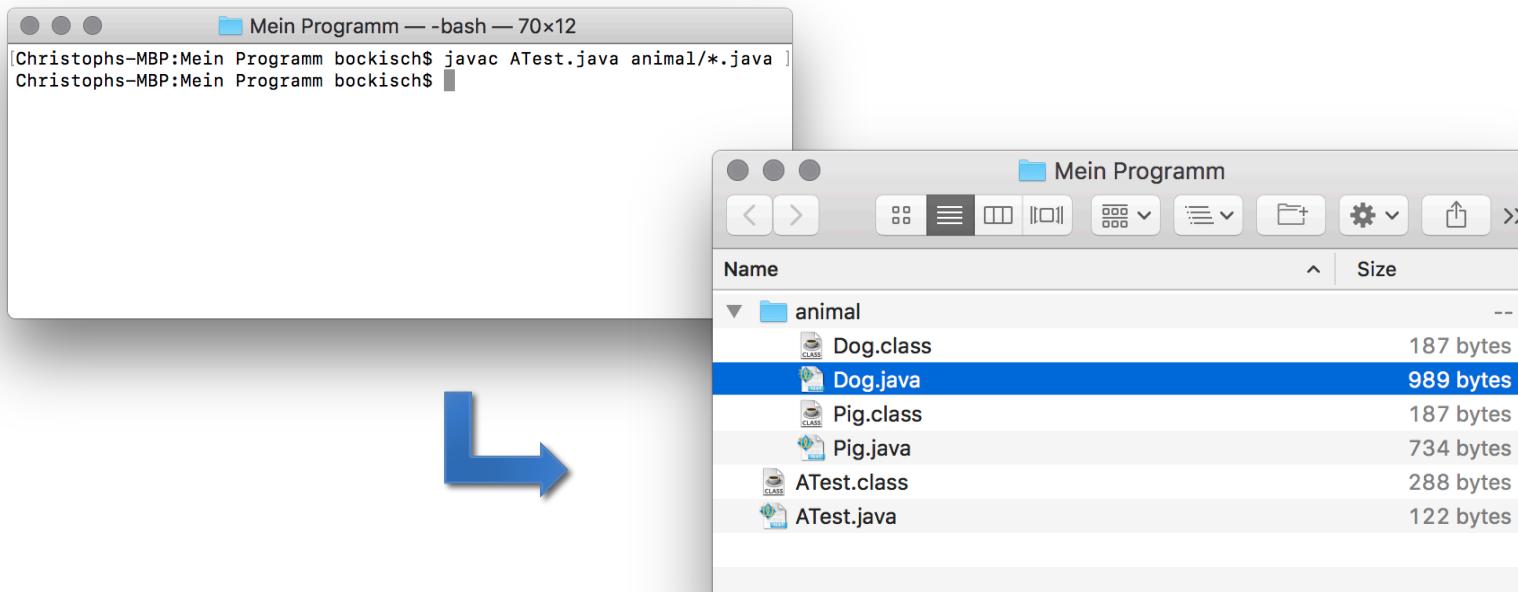
- *Die Klassen des Pakets animal müssen in dem Unterverzeichnis animal liegen*

Durch Setzen von CLASSPATH
kann diese starre Regel noch
aufgeweicht werden (→ später).

Verwendung eigener Pakete

```
import animal.Dog;
public class ATest {
    public static void main(String args[]) {
        Dog a = new Dog();
    }
}
```

wie kann ich Klassen in einem Paket und eine andere Klasse die diese Klasse importiert einfach kompilieren und ausführen mit cmd



Namenskonventionen

- Wir hatten bisher einen **einfachen Namen** für das Package gewählt.
 - Man kann hier aber auch mit Paket- und Verzeichnishierarchien arbeiten.
- Wenn die Klasse **K** in dem Package **a.b.c** sein soll, müssen wir folgende Importanweisung verwenden:
`import a.b.c.K;`
- Im Dateiverzeichnis muss **a** dann ein Unterverzeichnis des aktuellen Verzeichnisses (Default Packages) sein, **b** ein Unterverzeichnis von **a**, **c** ein Unterverzeichnis von **b** und **K.class** muss in **c** enthalten sein.
- Die Namen von Packages sollten eindeutig sein, weshalb es gewisse Konventionen gibt, die aber hier in der Vorlesung keine Rolle spielen.
 - Um Pakete von Klassen zu unterscheiden, sollten wir den Paktnamen mit einem kleinen Buchstaben und Klassennamen mit einem großen Buchstaben beginnen.

Speicherorte der Pakete

- Pakete können in irgendeinem Verzeichnis auf dem eigenen Rechner liegen.
- In frühen Versionen von Java wurde propagiert, sie könnten auch auf einem Rechner irgendwo im Internet sein. Davon ist man wohl abgekommen:
 - aus Sicherheitsgründen,
 - aus Effizienzgründen und
 - um die Konsistenz von Versionen besser kontrollieren zu können.

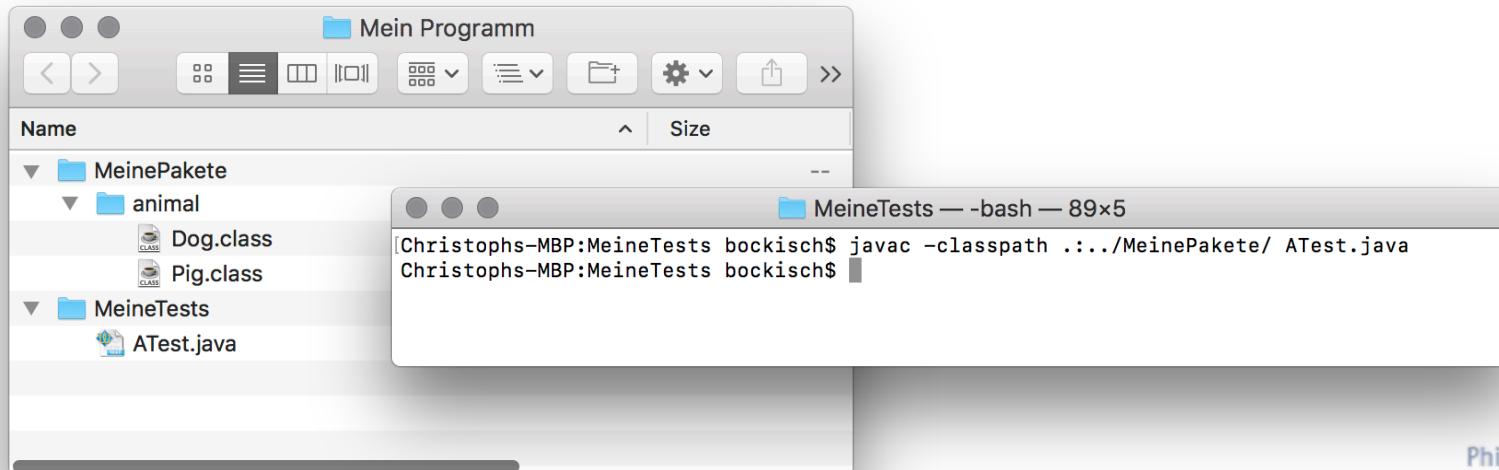
Die Umgebungsvariable CLASSPATH

Problem

- Bisher haben wir immer nur den Fall betrachtet, das die Pakete in Unterverzeichnissen des aktuellen Verzeichnisses (Default Package) zu finden sind.
 - Das ist natürlich wenig flexibel und selten der Fall, wenn vorgefertigte Pakete benutzt werden sollen.

Lösung

- Dem Compiler kann über die Option **-classpath** die Position anderer Pakete mitgeteilt werden



Die Umgebungsv

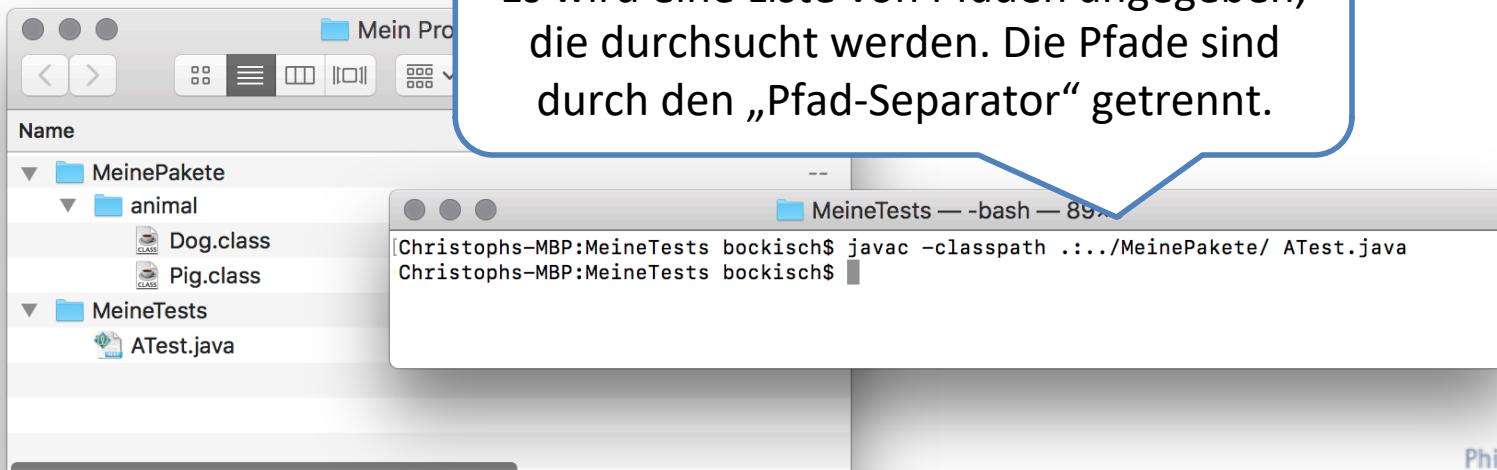
Problem

- Bisher haben wir immer nur den Fall betrachtet, dass die Pakete Unterverzeichnisse des Quellcodeverzeichnisses sind.
 - Das ist natürlich praktisch, aber nicht immer sinnvoll.
- Achtung: Der Pfadseparator ist plattformspezifisch.
- Für Linux und macos Doppelpunkt (:).
- Für Windows Semikolon (;)
- Auch Pfadangaben selbst sind plattformspezifisch.

Lösung

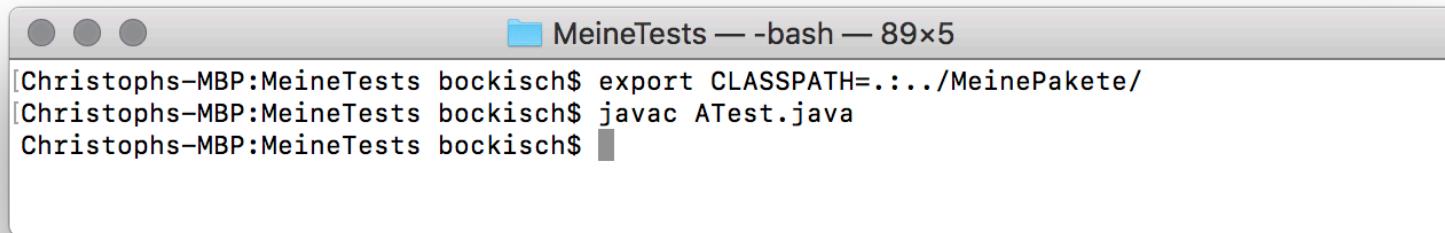
- Dem Compiler kann über die Option **-classpath** die Position anderer Pakete mitgeteilt werden

Es wird eine Liste von Pfaden angegeben, die durchsucht werden. Die Pfade sind durch den „Pfad-Separator“ getrennt.



Die Umgebungsvariable CLASSPATH

- Wenn häufig derselbe Classpath benötigt wird, kann auch die Umgebungsvariable **CLASSPATH** gesetzt werden.



```
[Christophs-MBP:MeineTests bockisch$ export CLASSPATH=.:./MeinePakete/
[Christophs-MBP:MeineTests bockisch$ javac ATest.java
[Christophs-MBP:MeineTests bockisch$ ]]
```

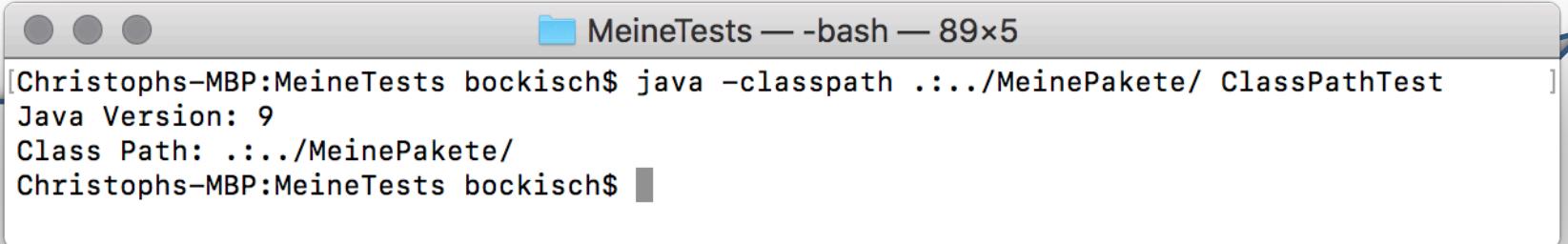
- Achtung: die Umgebungsvariable wird gelöscht, wenn Sie die Konsole schließen
 - Unter Linux/macos können Sie das entsprechende Kommando der Datei `.bash_profile` hinzufügen
 - Für Windows siehe: <https://www.java.com/de/download/help/path.xml>

Ich habe einfach die Klasse kopiert und in eine Datei gespeichert
Mit diesem command hat es geklappt javac -classpath .;.. ClassPathTest.java
dann java ClassPathTest

Inspektion des Classpath

- Den Wert des Classpath kann man mit der Methode `getProperty` der Klasse `System` bekommen. Ein Beispiel hierzu:

```
class ClassPathTest {  
  
    public static void main(String[] args) {  
        System.out.println("Java Version: " + System.getProperty("java.version"));  
        System.out.println("Class Path: " + System.getProperty("java.class.path"));  
    }  
}
```



```
[Christophs-MBP:MeineTests bockisch$ java -classpath .;./MeinePakete/ ClassPathTest  
Java Version: 9  
Class Path: .;/MeinePakete/  
Christophs-MBP:MeineTests bockisch$ ]
```

- In diesem Fall enthält der **CLASSPATH** zwei Verzeichnisse:
 - Das **aktuelle Verzeichnis**, gekennzeichnet durch den Punkt (vor dem Pfadseparatoren)
 - Achtung: Wenn man den Punkt im CLASSPATH weglässt, bekommt man Probleme
 - ./MeinePakete/**
 - Neben normalen Verzeichnissen sind auch **zip- und jar-Verzeichnisse** erlaubt

Zusammenfassung

- Erstellung eines Gesamtprogramms
 - Bedarfsorientiertes Laden und Initialisieren der Klassen
- Aufteilen von Klassen in Pakete
 - Nutzen von Klassen aus anderen Paketen
 - Erstellen neuer Pakete
- Umgebungsvariable CLASSPATH

bei windows ich hatte ein Ordner heißt MeinePakete und in diesem Verzeichnis gibt es das Verzeichnis animal und die Klasse ATest.java und in dem Verzeichnis animal gibt es die Datei Dog.java dog ist im package animal und die Klasse ATest.java importiert die Klasse Dog.java Du musst zu jedem Verzeichnis gehen und dort kompilieren mit javac -classpath ;... Dog.java dann zurück zum Verzeichnis wo ATest ist also mit cd.. dann mit javac -classpath ;.. MeinePakete\ATest.java dann java ATest.java hat kein Fehler gegebenen also alles gut

```
C:  
\Users\omara\Desktop\MeinePakete\animal>javac -classpath ;...  
Dog.java
```

```
C:  
\Users\omara\Desktop\MeinePakete\animal>javac -classpath ;...  
MeinePakete\ATest.java  
Fehler: Datei nicht gefunden: MeinePakete\ATest.java  
Verwendung: javac <Optionen> <Quelldateien>  
Mit --help können Sie eine Liste der möglichen Optionen aufrufen
```

```
C:\Users\omara\Desktop\MeinePakete\animal>cd ..
```

```
C:\Users\omara\Desktop\MeinePakete>javac -classpath ;...  
MeinePakete\ATest.java  
Fehler: Datei nicht gefunden: MeinePakete\ATest.java  
Verwendung: javac <Optionen> <Quelldateien>  
Mit --help können Sie eine Liste der möglichen Optionen aufrufen
```

```
C:\Users\omara\Desktop\MeinePakete>javac -classpath ;...  
ATest.java
```

```
C:\Users\omara\Desktop\MeinePakete>java ATest.java
```

```
C:\Users\omara\Desktop\MeinePakete>
```

Philipps

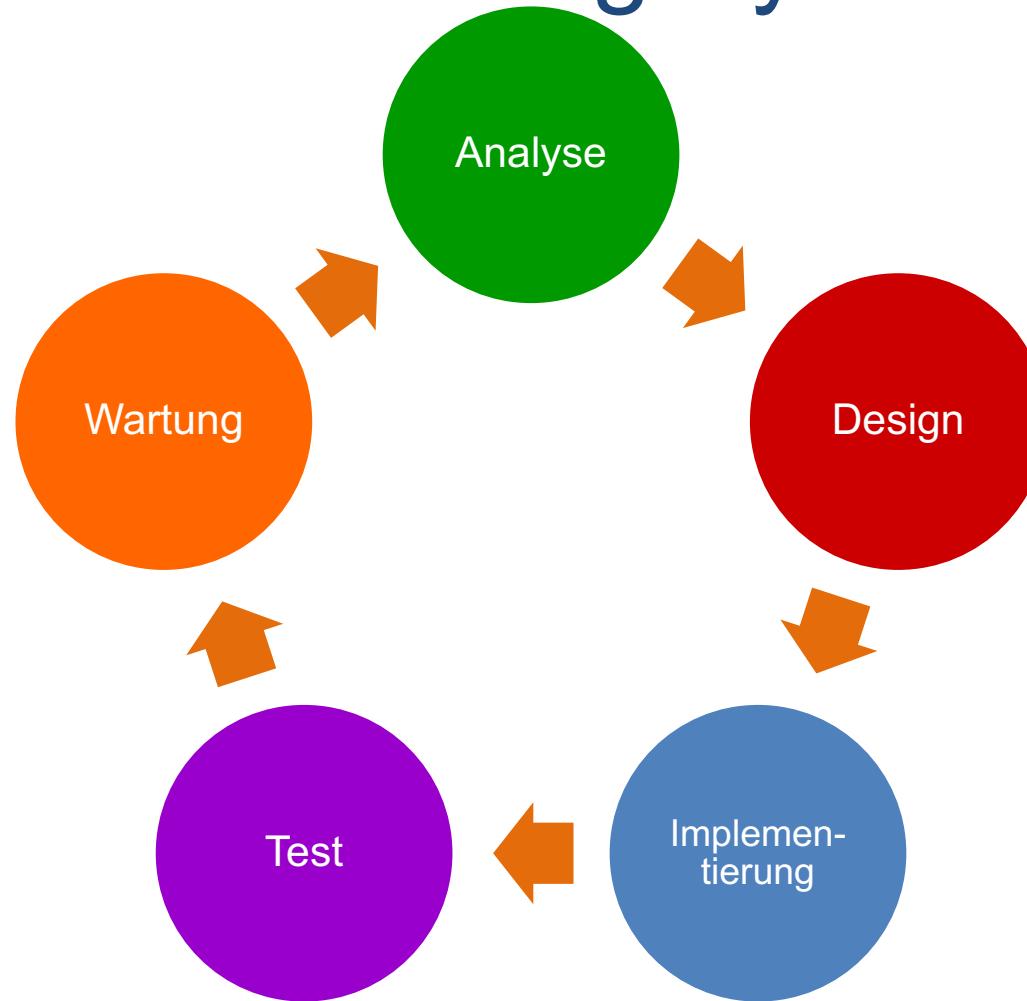


Universität
Marburg

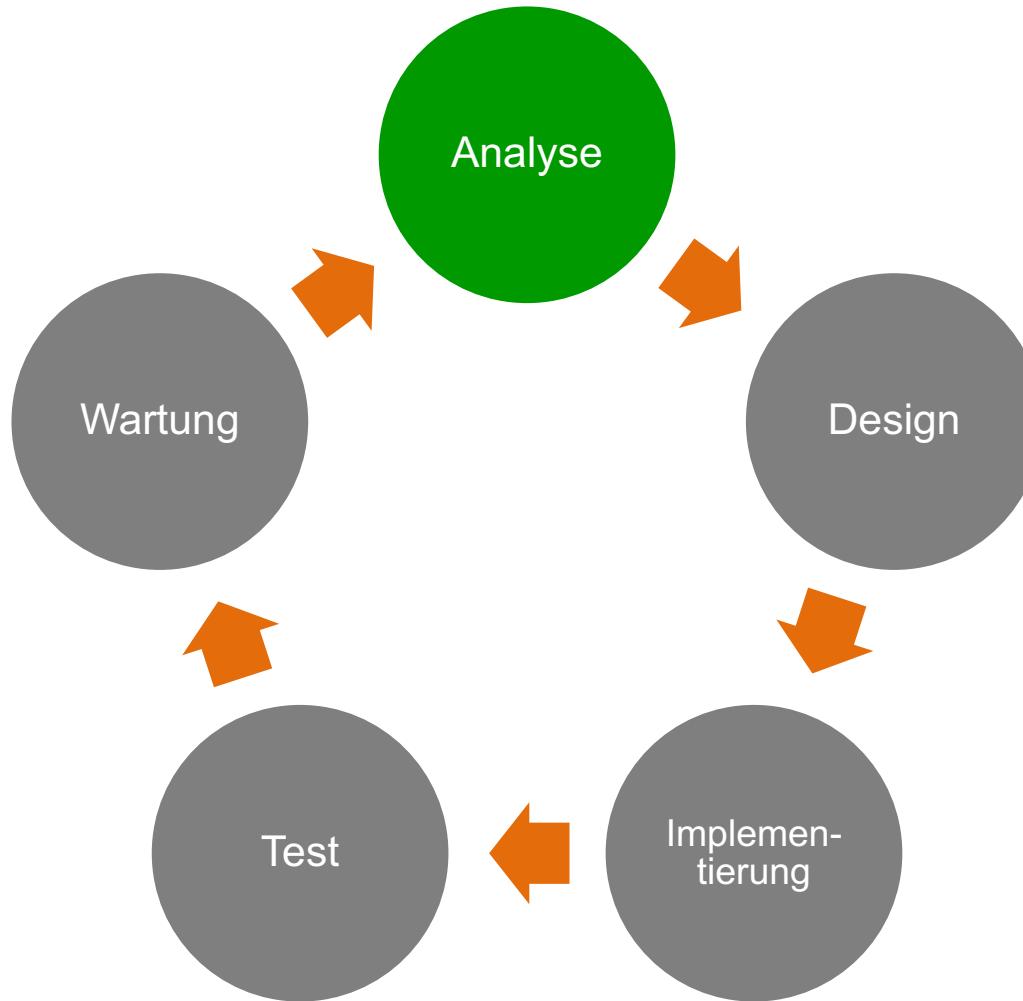
7. Programmieren im Großen

- Entwicklung: IntelliJ IDEA, Bibliotheken, Versionsverwaltung
- Testen: Unit-Tests, Integrationstests
- Fehlerbehebung: Debugger
- Auslieferung: JAR-Dateien

Software-Entwicklungszyklus



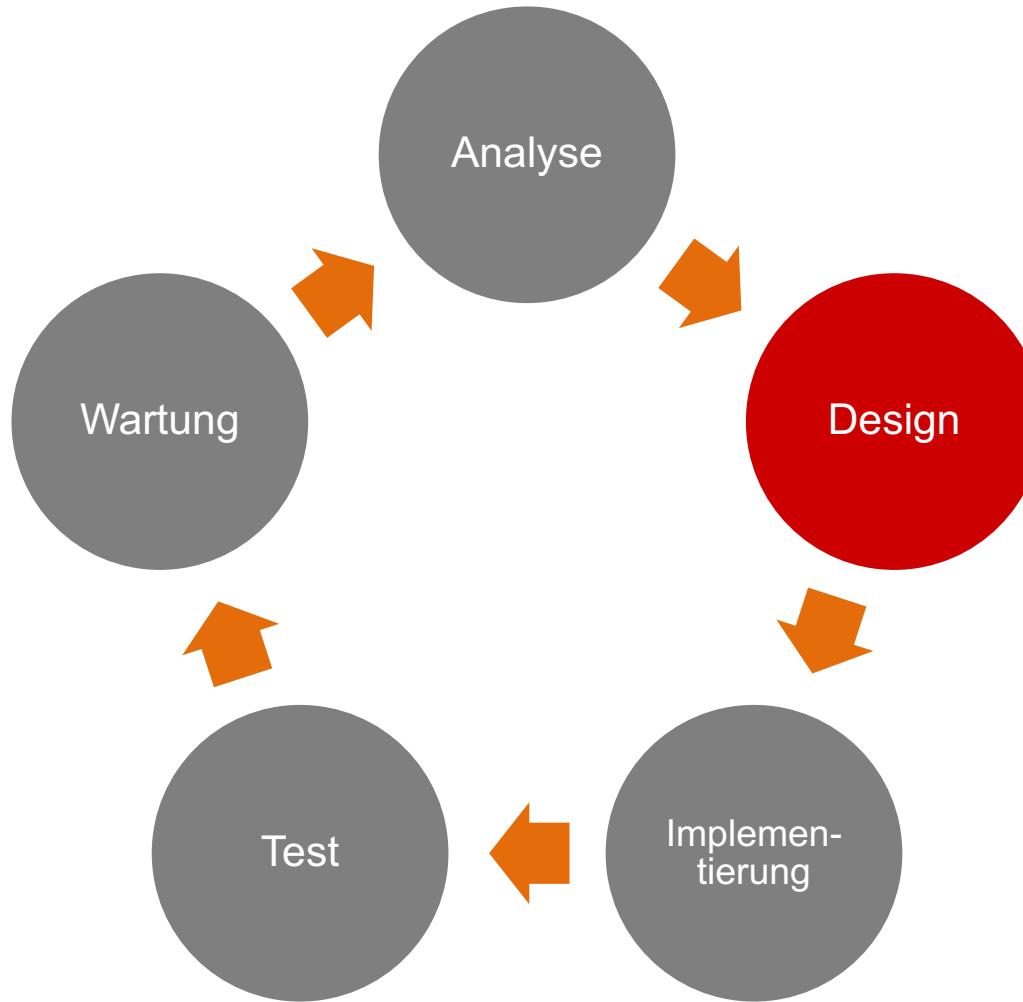
Übersicht



Anforderungsanalyse

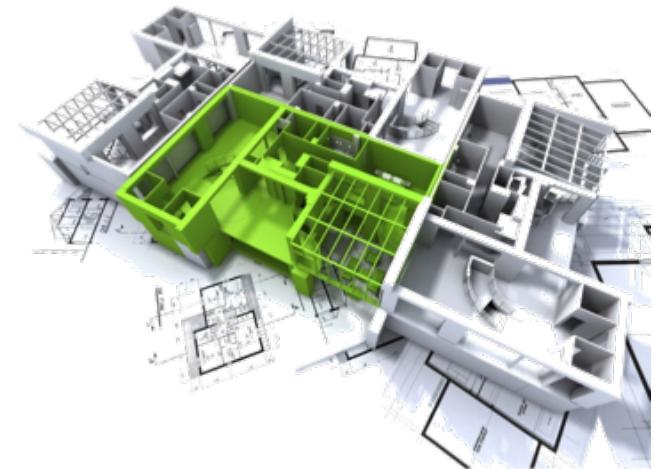
- Was möchte der Auftraggeber?
- Sammlung aller Anforderungen
 - Anwender und Entwickler haben oft nicht die gleiche Sichtweise
- Überprüfung der Anforderungen
 - Machbarkeit, Abhängigkeiten, Konsistenz
- Sorgfältige Analyse ist wichtig, da sich Fehler über den gesamten Entwicklungszyklus erstrecken und enorme Kosten verursachen können.

Übersicht

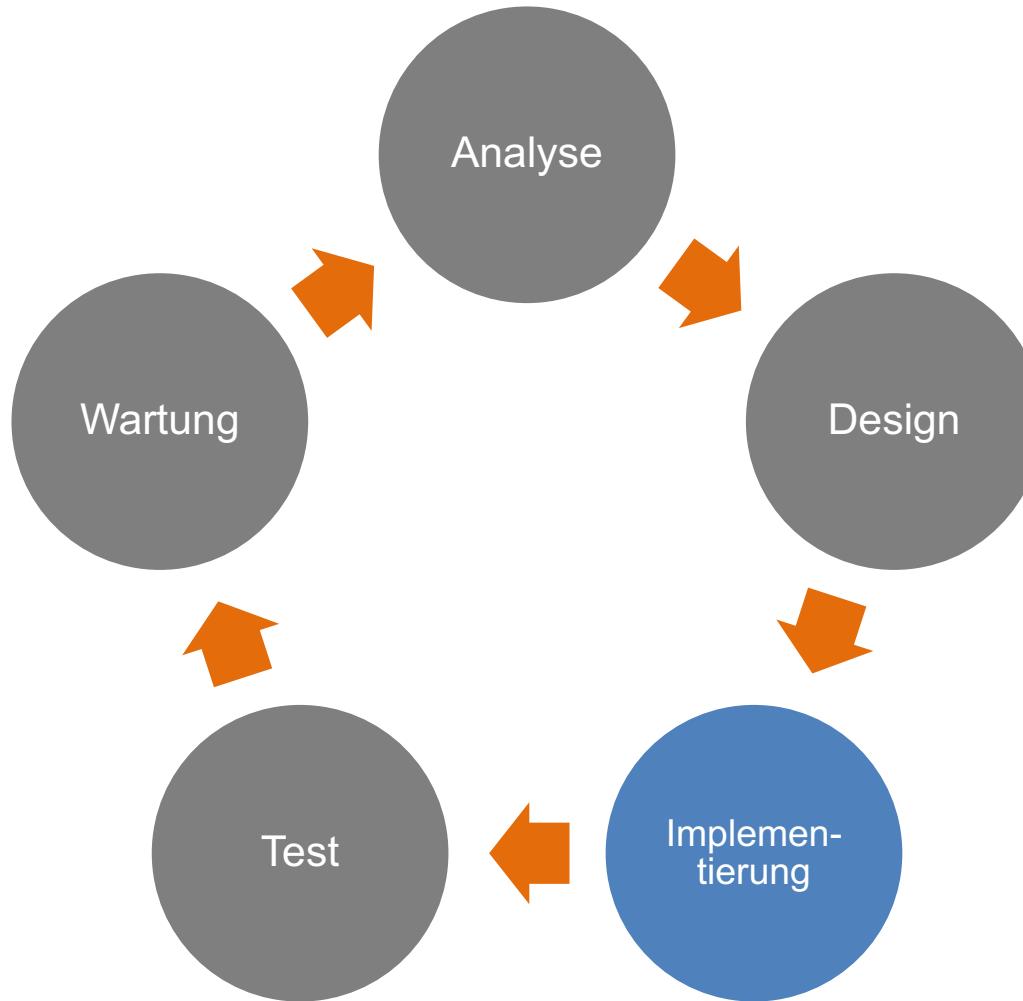


Softwaredesign

- Programmieren im Kleinen: Implementierung eines Algorithmus in Java, Aufteilung in Methoden
- Programmieren im Großen: Entwurf der Systemarchitektur
 - Bestimmung von Komponenten des Systems (Modularisierung)
 - Definition von **Schnittstellen** zwischen den Komponenten



Übersicht



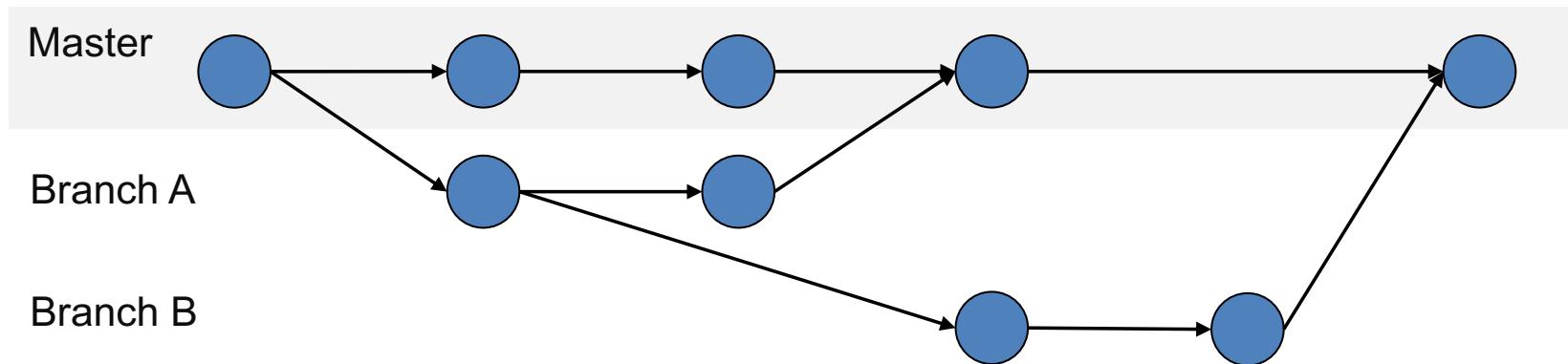
Implementierung

- Nach der Designphase sind die wichtigen Schnittstellen definiert
- Implementierung erfolgt üblicherweise unabhängig voneinander (oft sogar in getrennten Teams)
- Klare Trennung der Komponenten mit Hilfe von Schnittstellen erleichtert die parallele Entwicklung
- Aber: die Komponenten an sich können auch wieder sehr umfangreich sein und mehrere Personen beschäftigen

→ Mehrere Personen arbeiten am gleichen Code

Versionsverwaltung

- Software wird oft in großen Teams und an verschiedenen Orten gleichzeitig entwickelt
 - gemeinsame Code-Basis und Synchronisierung notwendig, um **Inkonsistenzen** zu vermeiden
- Änderungen der Software sollen nachvollziehbar und ggf. auch reversibel sein



Versionsverwaltungssysteme

- **Git** ist das mit Abstand am verbreitetste Tool
 - Dezentrales System ohne notwendige Client-Server-Struktur
 - Änderungen werden feingranular lokal versioniert (*commits*) und dann mit Teammitgliedern synchronisiert
 - Konflikte werden über *merges* gelöst
- github.com ist eine populäre Git-Hosting-Plattform
 - Frei nutzbar für Open-Source-Projekte
 - 24 Mio. Benutzer (5 Mio. in Europa)
 - 67 Mio. Repositories (25 Mio. aktiv in 2017)
 - Linux-Kernel mit über 700k Commits, Git selbst
 - Microsoft VSCode mit über 15k Beteiligungen

Implementierung mit einer IDE

- Bisher:
 - Entwicklung mit Hilfe eines Editors (z.B. Notepad++)
 - Übersetzen mit `javac` auf der Konsole
 - Ausführen mit `java` auf der Konsole
- Auf Dauer sehr lästige Vorgehensweise!
- Abhilfe: Integrierte Entwicklungsumgebungen (IDE)
 - Umfangreiche Werkzeuge für die Entwicklung von Programmen
 - Bekannte IDEs für Java: **IntelliJ IDEA**, Eclipse, NetBeans



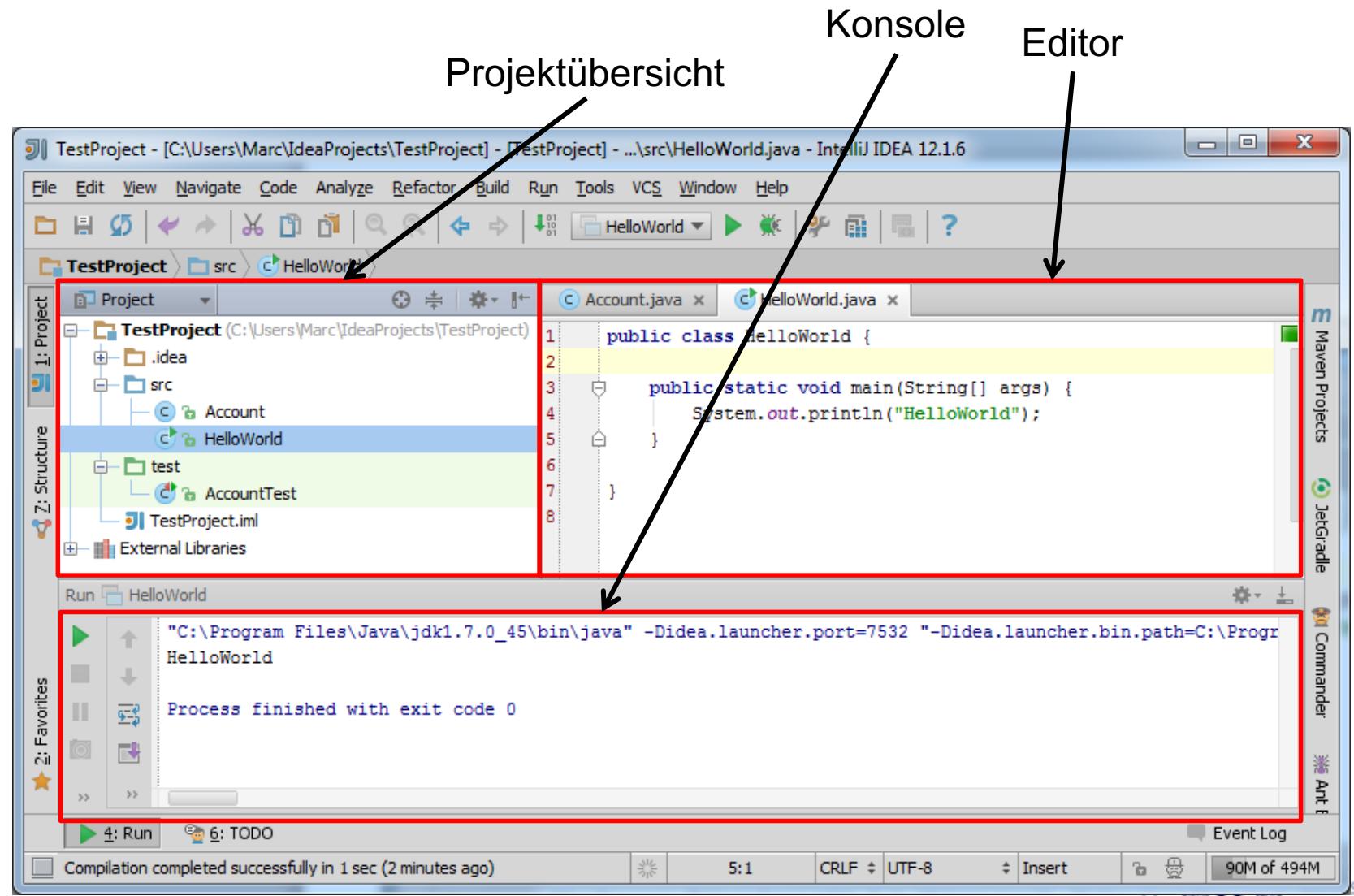
IntelliJ IDEA

IntelliJ IDEA

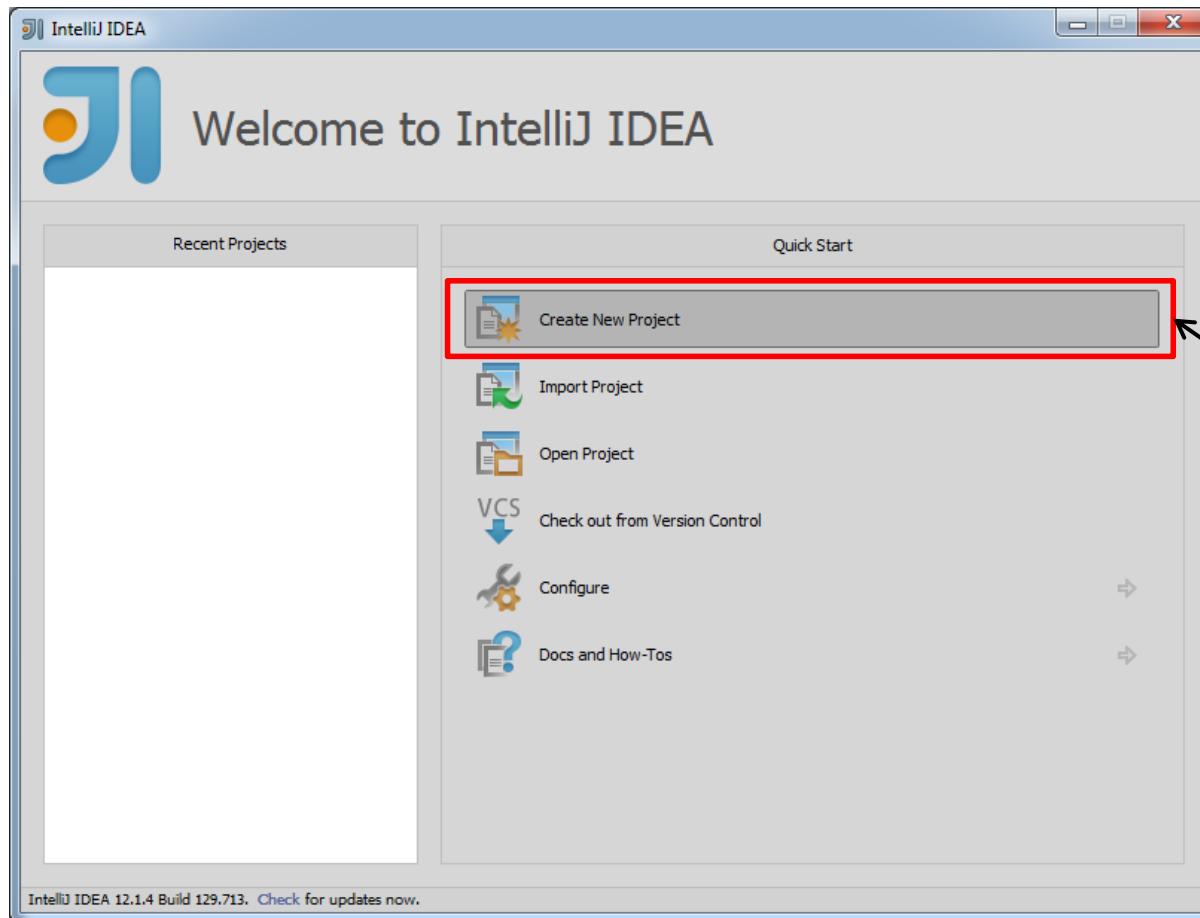
- Wir werden zukünftig die Open-Source Community Edition von IntelliJ IDEA (kurz: IntelliJ) nutzen (<http://www.jetbrains.com/idea/>)
 - Verwaltet Programmcode in Projekten
 - Quellcode liegt im Verzeichnis „src“
 - Zeigt Fehlermeldungen des Compiler direkt im Editor an
 - Übersetzt den Programmcode automatisch
 - Auto vervollständigung im Editor
 - Integrierte Konsole
 - ... und weitere nützliche Features

Hilft nicht in der Klausur. Deshalb sollte man sich auf diese Funktionalität nicht verlassen.

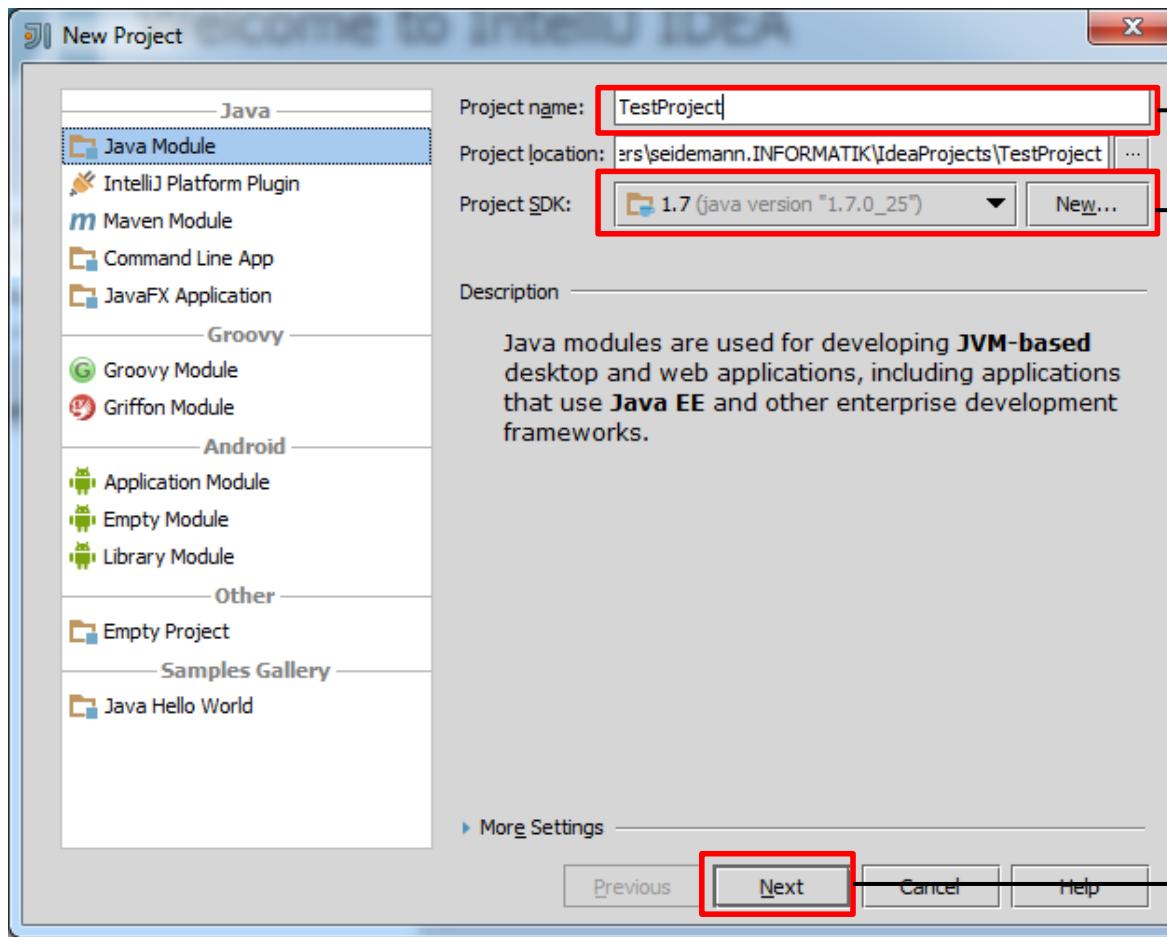
IntelliJ: Übersicht



IntelliJ: Einrichtung



IntelliJ: Einrichtung

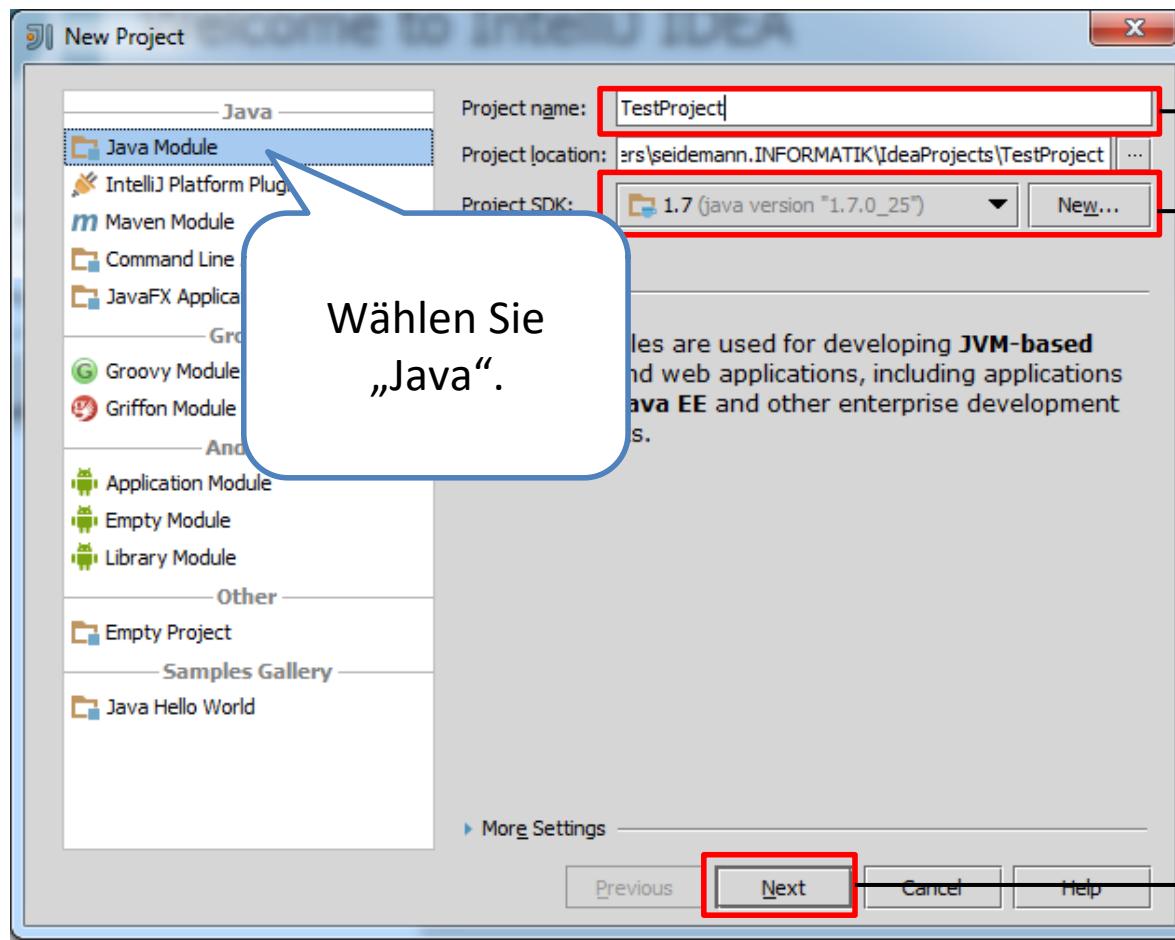


Name des Projekts

Falls nicht vorhanden:
über „New...“ den Pfad
zum JDK-Verzeichnis
auswählen

„Next“ und im
anschließenden Fenster
„Finish“ klicken

IntelliJ: Einrichtung

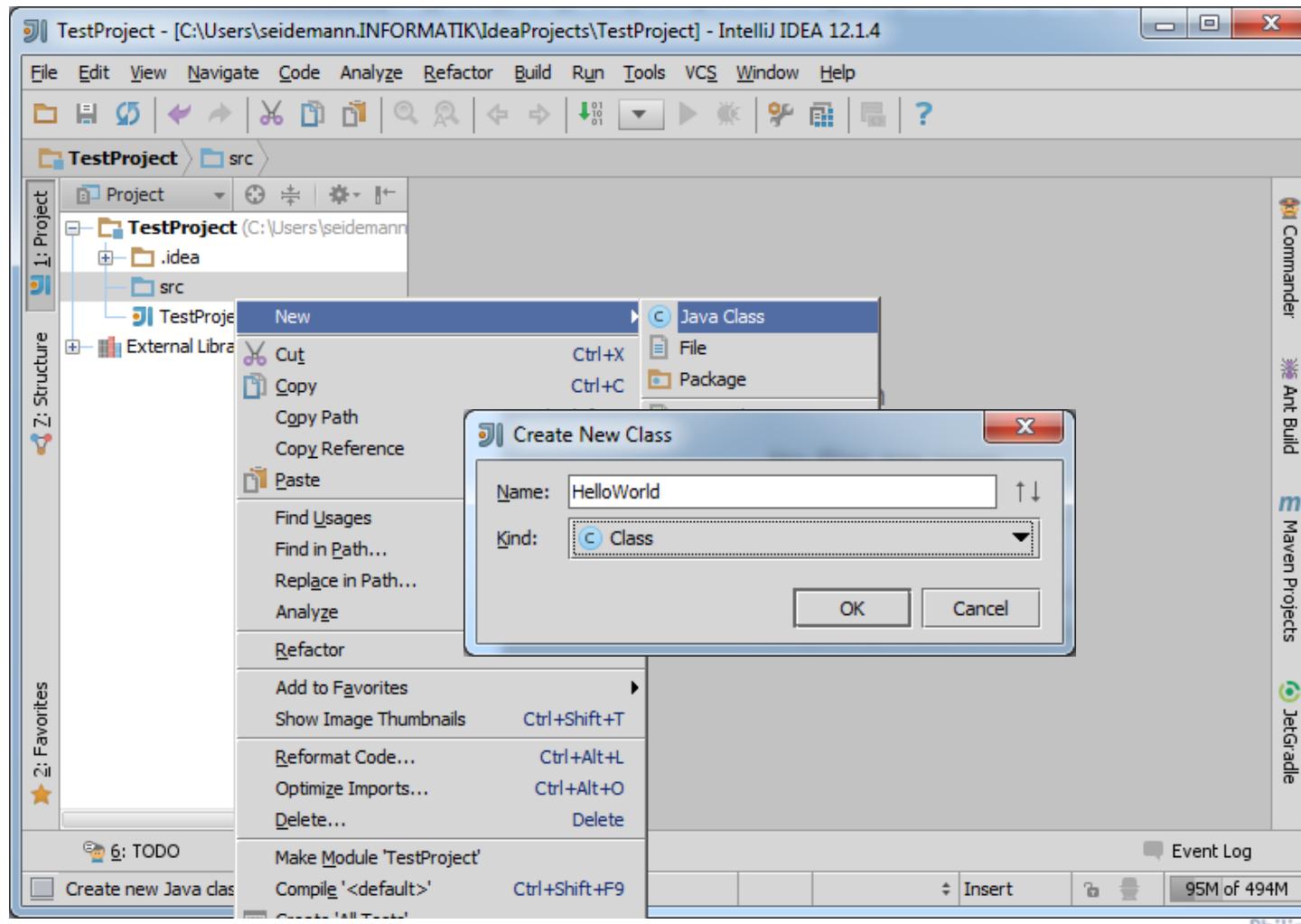


Name des Projekts

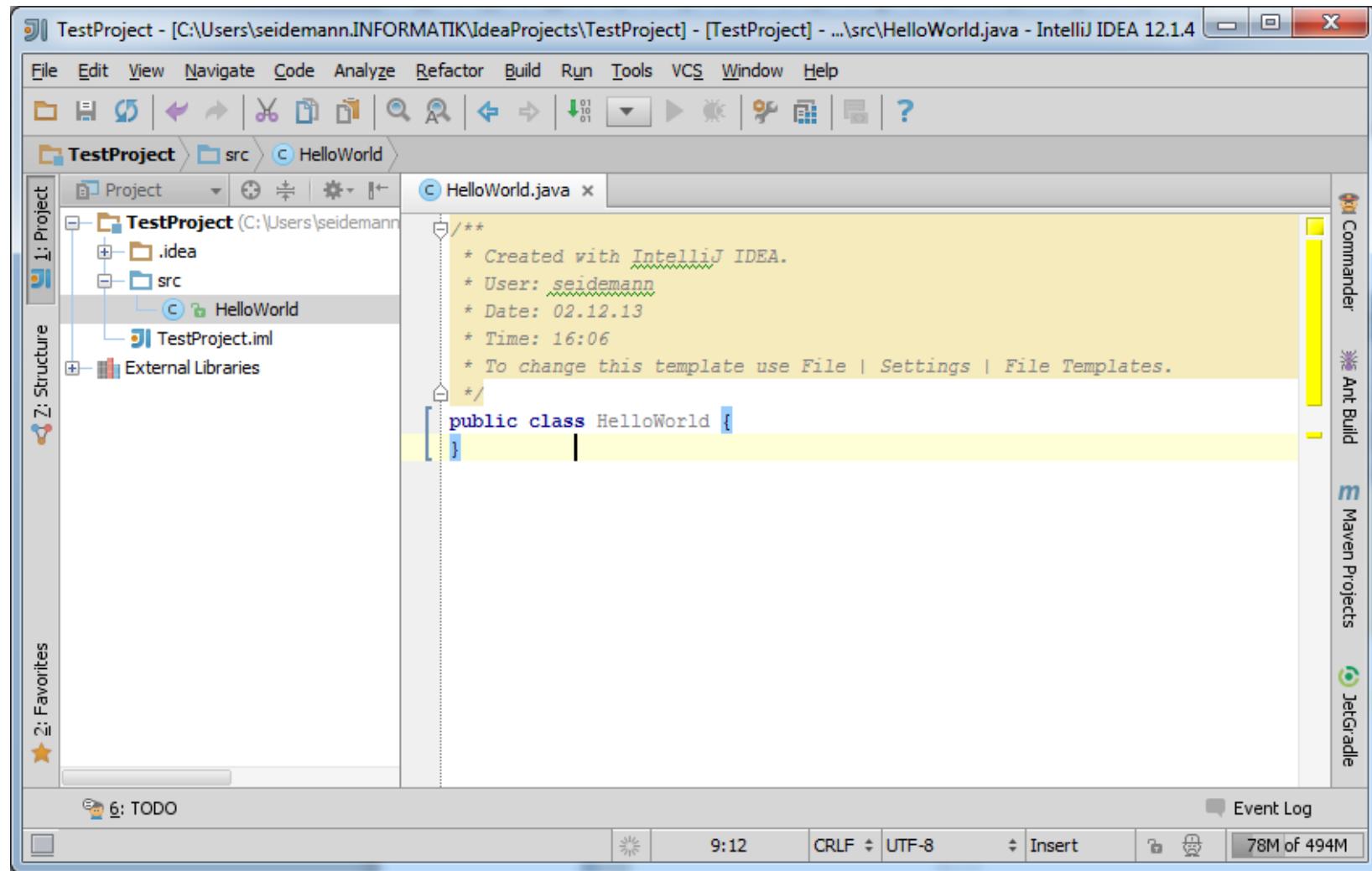
Falls nicht vorhanden:
über „New...“ den Pfad
zum JDK-Verzeichnis
auswählen

„Next“ und im
anschließenden Fenster
„Finish“ klicken

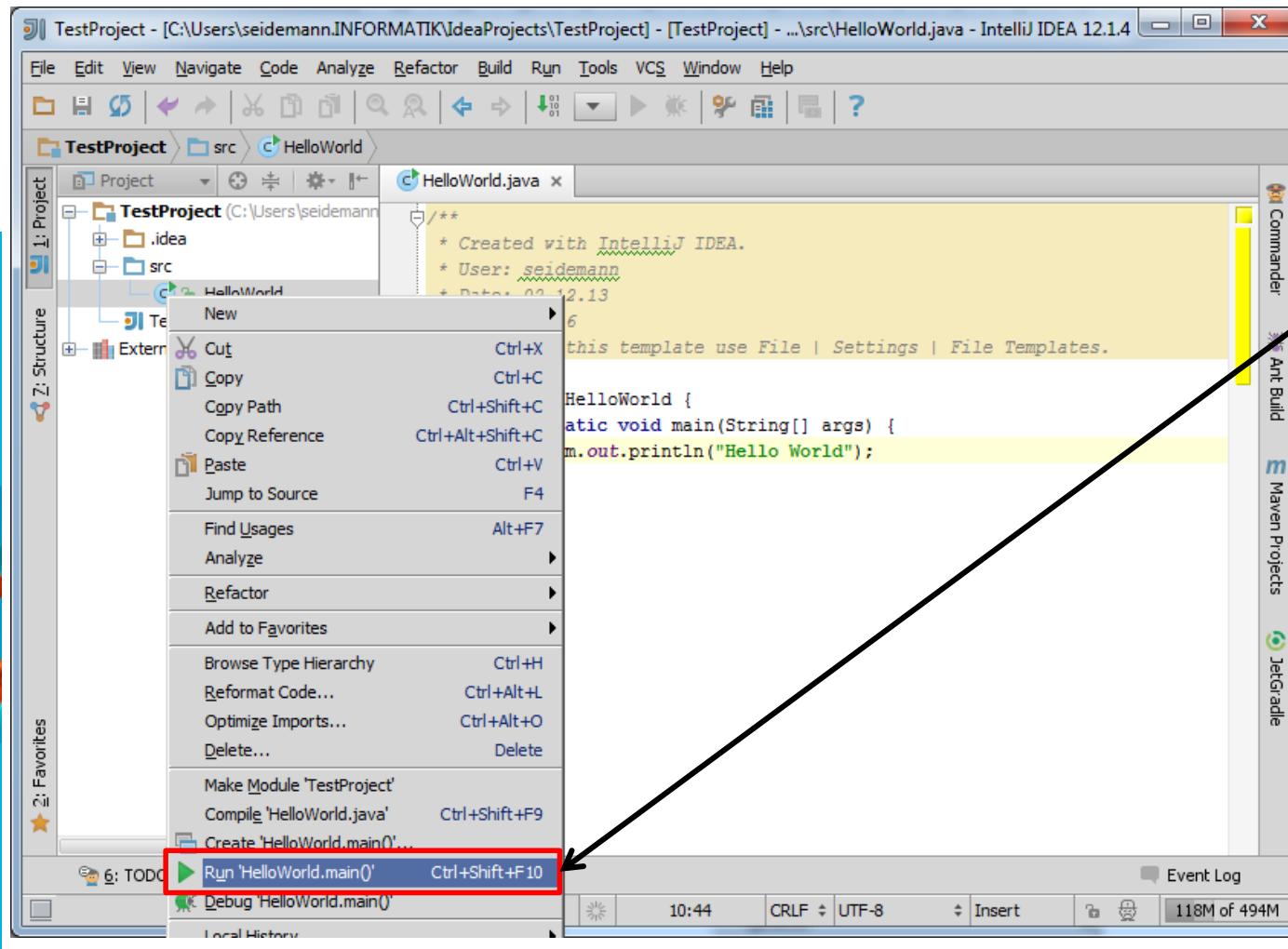
IntelliJ: HelloWorld



IntelliJ: HelloWorld

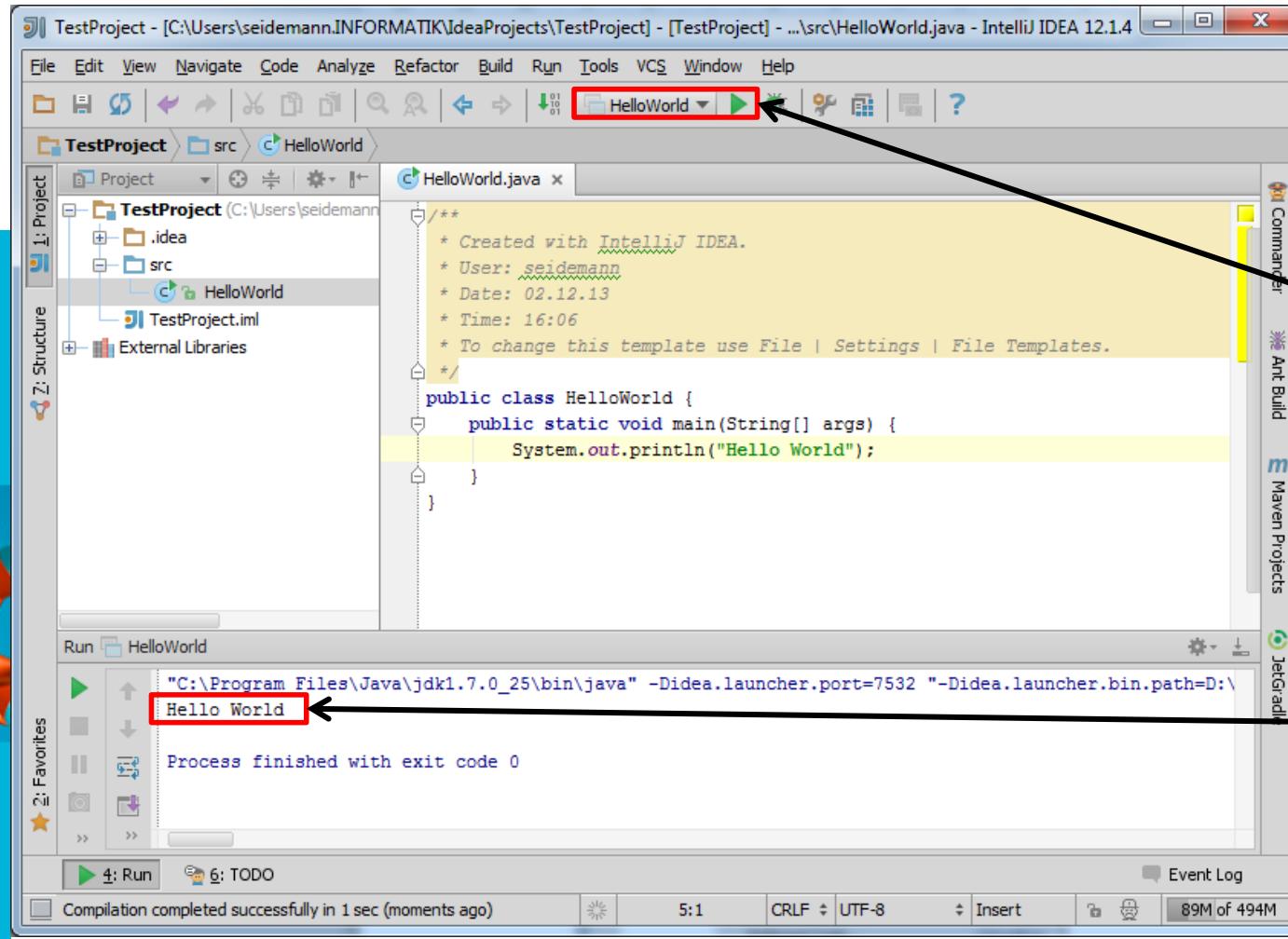


IntelliJ: Programm ausführen

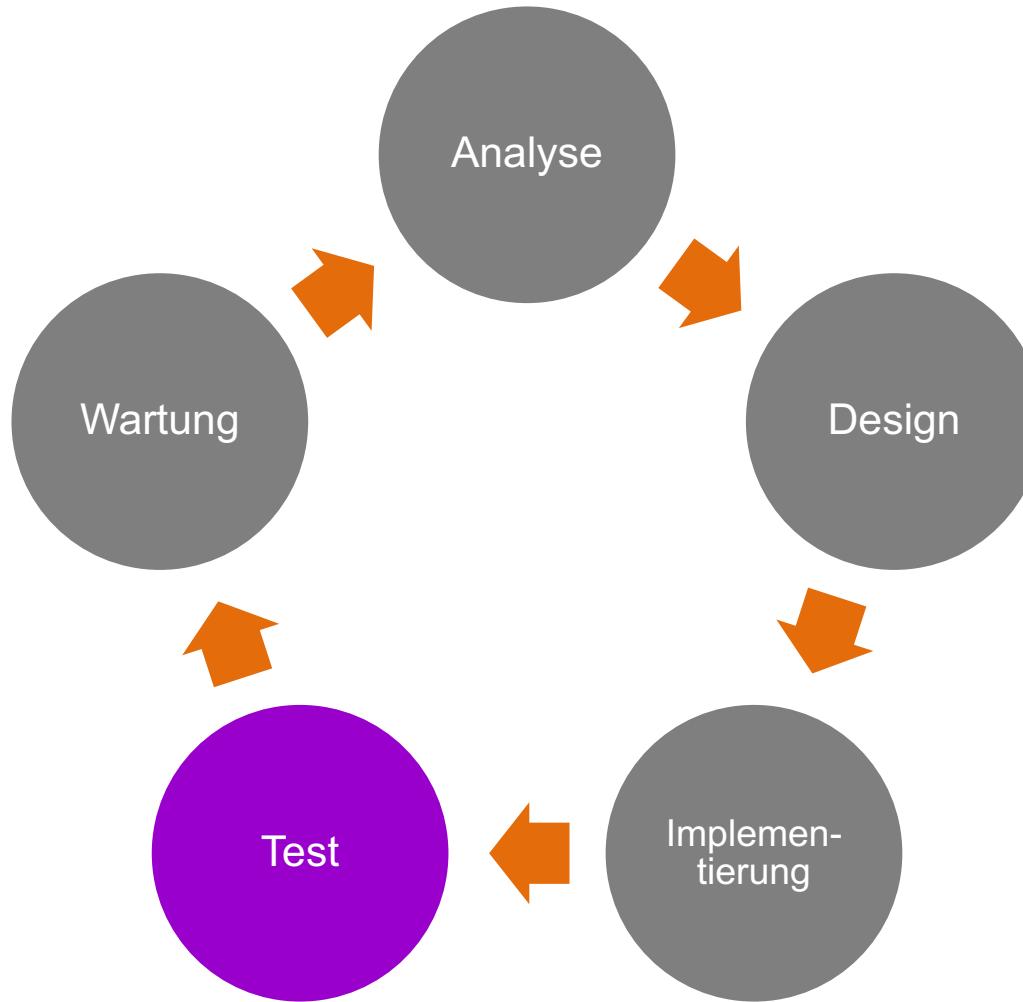


Programm
ausführen
([STRG]+[SHIFT]+[F10])

IntelliJ: Programm ausführen



Übersicht



Testen

- Auf 1000 Zeilen Programmcode fallen durchschnittlich 1-25 Fehler
- Wird ein Fehler erst nach der Auslieferung der Software gefunden, kostet die Behebung nicht selten das **zehnfache**
- Oder auch mehr...

Start einer Ariane 5 (1996):
Fehlerhafte Umwandlung einer 64-bit Gleitkommazahl in einen 16-bit Short-Wert führte zu einem Ausfall der Steuerung

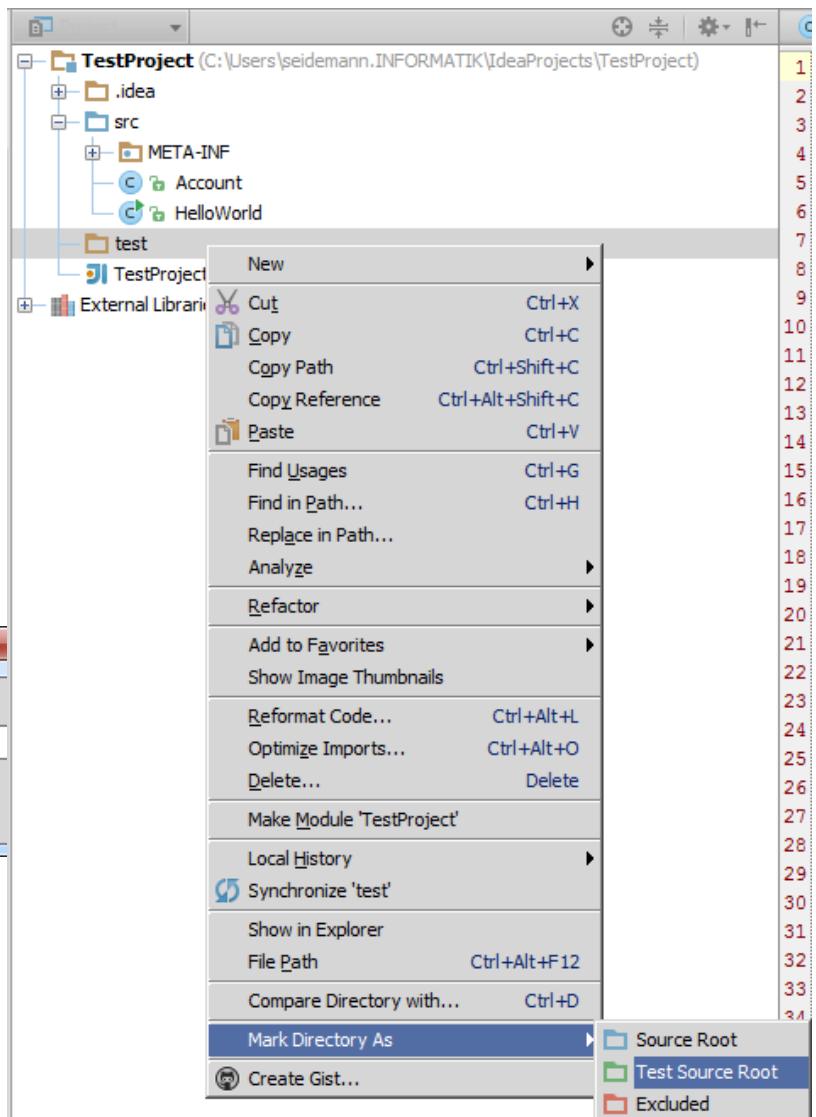
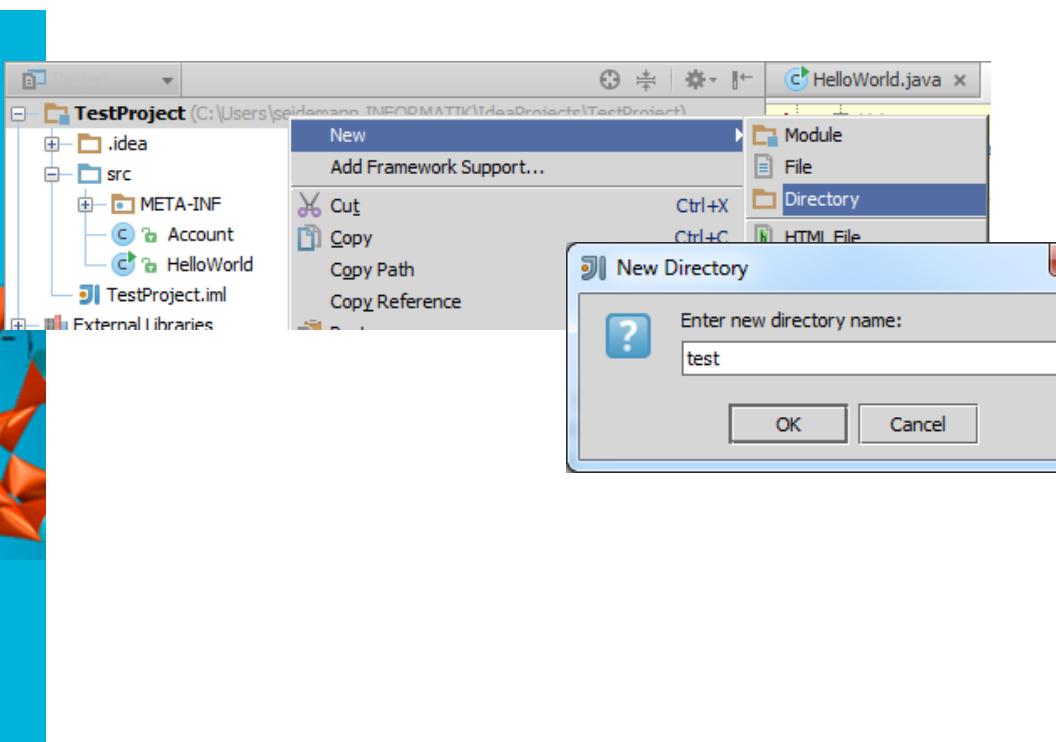


Unit-Tests

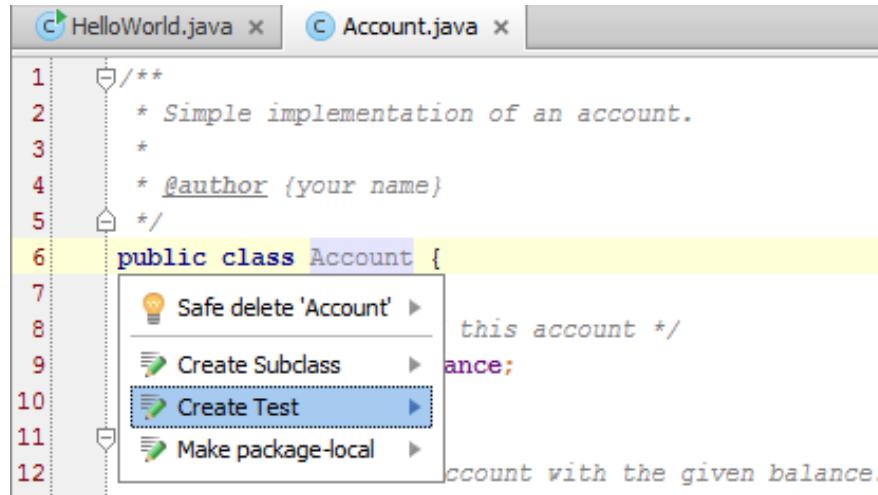
- Test einer kleinen Einheit (z.B. Klasse) für sich
- Für jede Klasse und jede Methode ein Test
 - mit Ausnahme von trivialen Methoden
- Prinzipien
 - Jeder Test spezifiziert ein unabhängiges und wiederholbares Szenario.
 - Vorbedingungen und Nachbedingungen werden in Form von **Assertions** geprüft.
 - Unit-Tests laufen selbstständig ab und besitzen keine Interaktion mit dem Anwender oder anderen Komponenten (idealerweise).

JUnit

Zuerst:
Verzeichnis für
Tests erstellen



JUnit



The screenshot shows an IDE interface with two tabs: 'HelloWorld.java' and 'Account.java'. The 'Account.java' tab is active, displaying the following code:

```
1  /**
2   * Simple implementation of an account.
3   *
4   * @author {your name}
5   */
6  public class Account {
7      ...
8  }
9
10
11
12 }
```

A context menu is open at the end of the line 'public class Account {'. The menu items are:

- Safe delete 'Account'
- Create Subclass
- Create Test** (highlighted)
- Make package-local

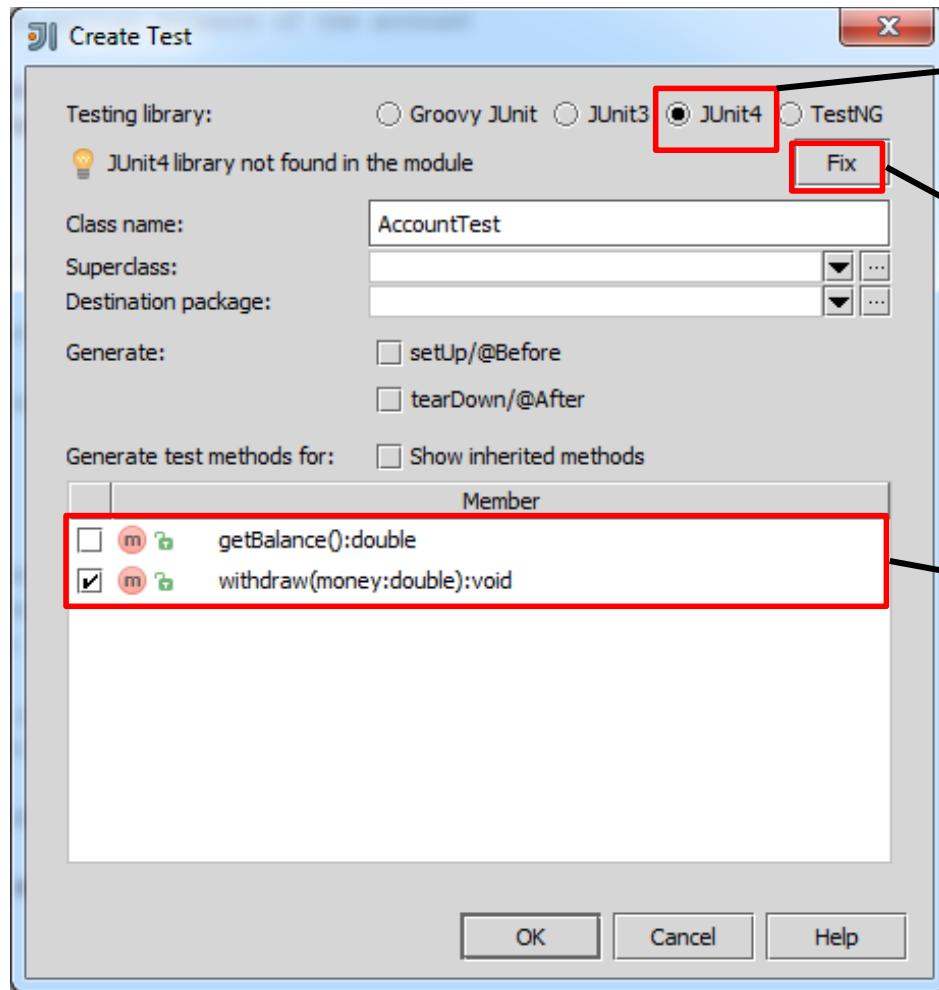
The code continues below the class definition:

```
this account */  
ance;  
account with the given balance.
```

Testklasse erstellen

- Cursor auf Klassennamen
- [ALT]+[ENTER]
- Create Test

JUnit

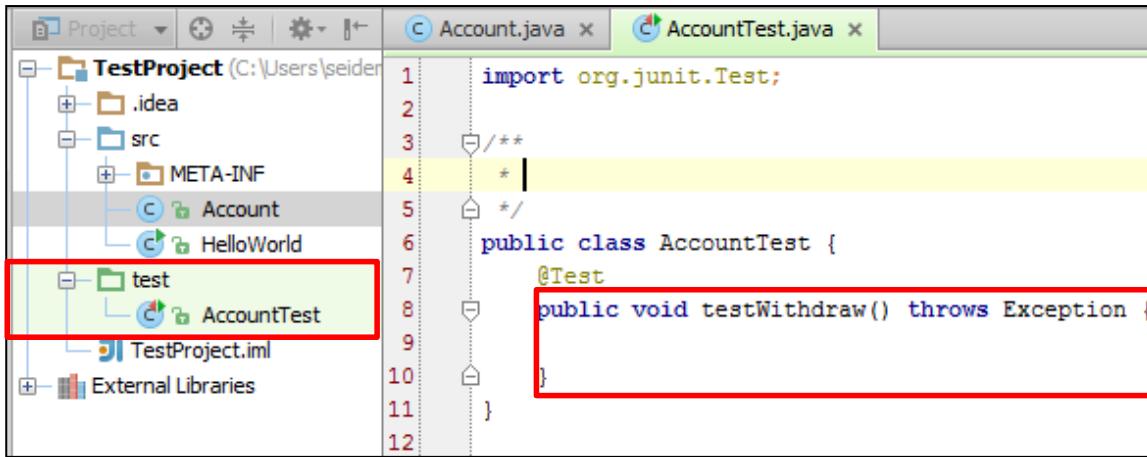


Wir benutzen **JUnit4**

Beim ersten Mal: **Fix** klicken

Auswahl der Methoden, für die eine Testmethode erstellt werden soll

JUnit



The screenshot shows a Java development environment with the following details:

- Project View:** Shows a project named "TestProject" located at "C:\Users\seider". It contains a ".idea" folder, a "src" folder with "META-INF", "Account", and "HelloWorld" packages, and a "test" folder containing "AccountTest". A red box highlights the "test" folder.
- Code Editor:** Two tabs are open: "Account.java" and "AccountTest.java".
 - "Account.java" contains a single line: `import org.junit.Test;`
 - "AccountTest.java" contains the following code:

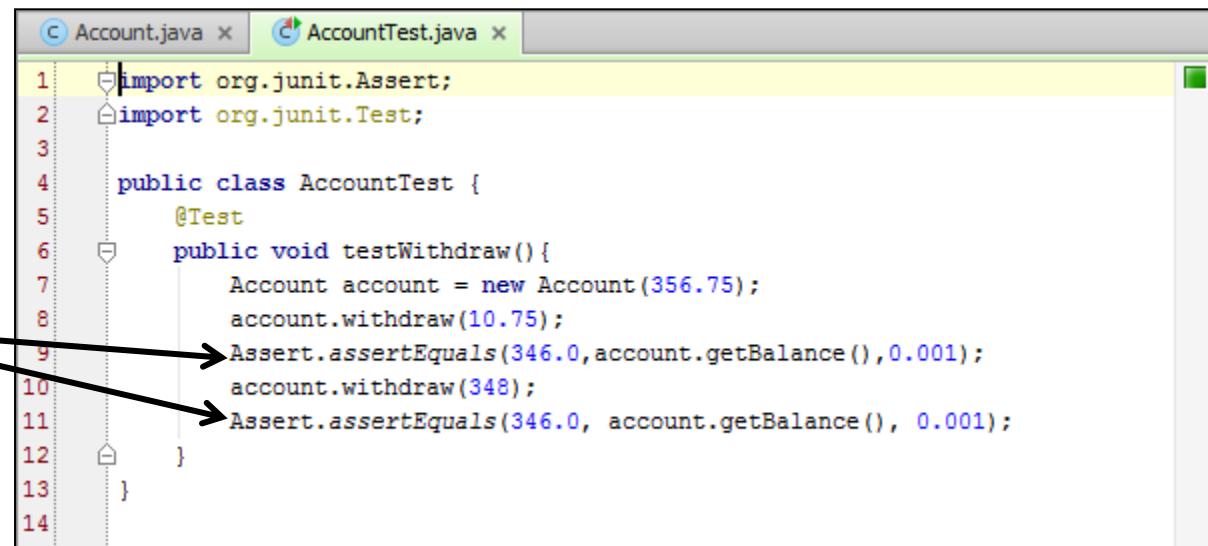
```
1 import org.junit.Test;  
2  
3 /**  
4  *  
5 */  
6 public class AccountTest {  
7     @Test  
8     public void testWithdraw() throws Exception {  
9         // Test implementation  
10    }  
11 }  
12 }
```

A red box highlights the entire content of the "AccountTest.java" code editor.

Die Testklasse befindet sich nun im Ordner „test“

JUnit

- Weitere Assertions
 - Assert.assertFalse(boolean condition)
 - Assert.assertTrue(boolean condition)
 - ...



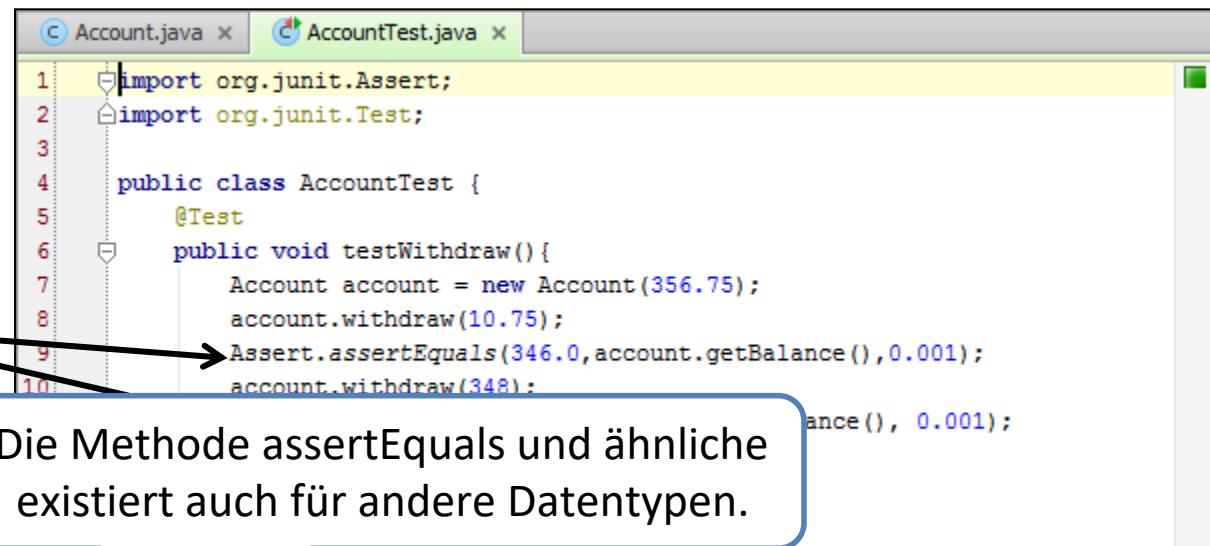
```
Account.java x AccountTest.java x
1 import org.junit.Assert;
2 import org.junit.Test;
3
4 public class AccountTest {
5     @Test
6     public void testWithdraw() {
7         Account account = new Account(356.75);
8         account.withdraw(10.75);
9         Assert.assertEquals(346.0, account.getBalance(), 0.001);
10        account.withdraw(348);
11        Assert.assertEquals(346.0, account.getBalance(), 0.001);
12    }
13 }
14
```

Überprüfung der
Nachbedingung

Assert.assertEquals(double expected, double actual, double difference)

JUnit

- Weitere Assertions
 - Assert.assertFalse(boolean condition)
 - Assert.assertTrue(boolean condition)
 - ...



```
1 import org.junit.Assert;
2 import org.junit.Test;
3
4 public class AccountTest {
5     @Test
6     public void testWithdraw() {
7         Account account = new Account(356.75);
8         account.withdraw(10.75);
9         Assert.assertEquals(346.0, account.getBalance(), 0.001);
10        account.withdraw(348);
```

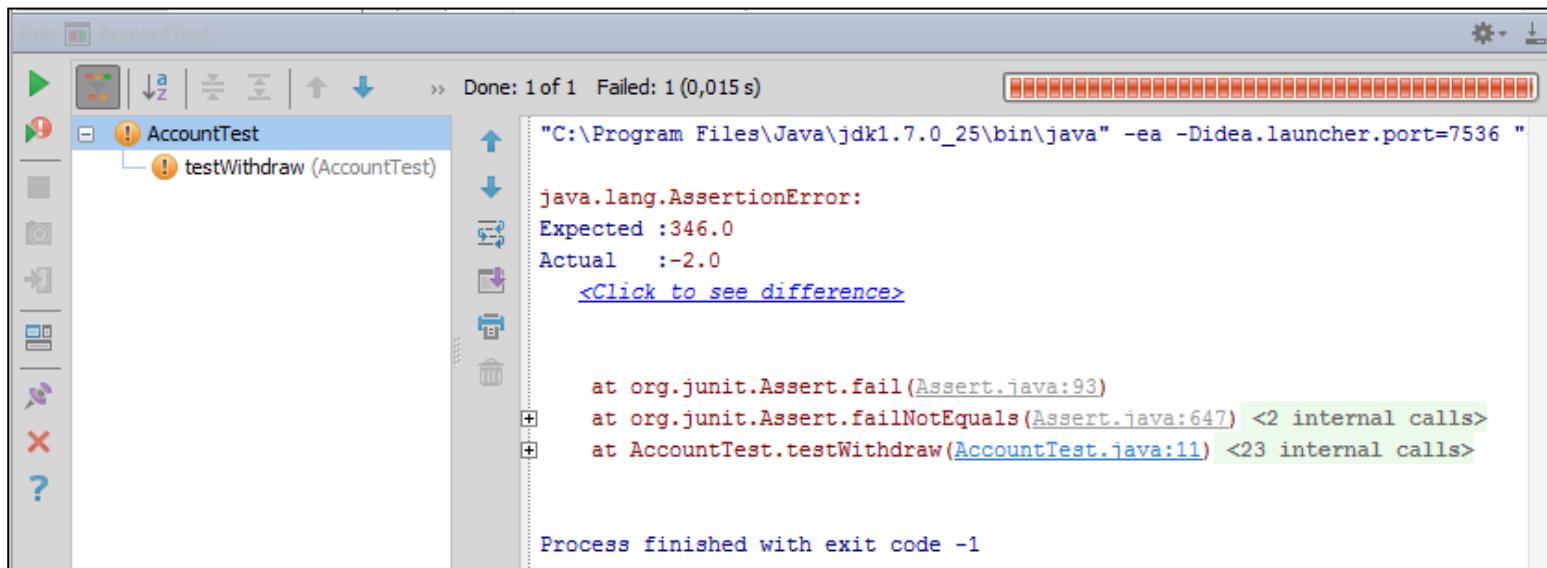
Überprüfung der Nachbedingung

Die Methode assertEquals und ähnliche existiert auch für andere Datentypen.

Assert.assertEquals(double expected, double actual, double difference)

JUnit

- Test ausführen: [STRG]+[SHIFT]+[F10]



```
public void withdraw(double money) {  
    this.balance -= money;  
}
```

Negative Kontostände sind nicht erlaubt!

Integrationstests

- Unit-Tests alleine genügen nicht!
 - Eine Software muss immer auch im Zusammenspiel mit externen Schnittstellen betrachtet werden (Klassen-übergreifend, Festplatten, Netzwerk, ...)
- Integrationstests dienen dazu, eine Komponente im Zusammenspiel mit anderen Komponenten zu testen
- Andere Komponenten sind durch externe Faktoren beeinflusst
 - Eine Methode möchte eine Datei auf der Festplatte anlegen
 - Festplatte wird auch von anderen Programmen verwendet
 - Zum Beispiel: Test geht 100x gut, beim 101-ten Test ist jedoch die Festplatte voll

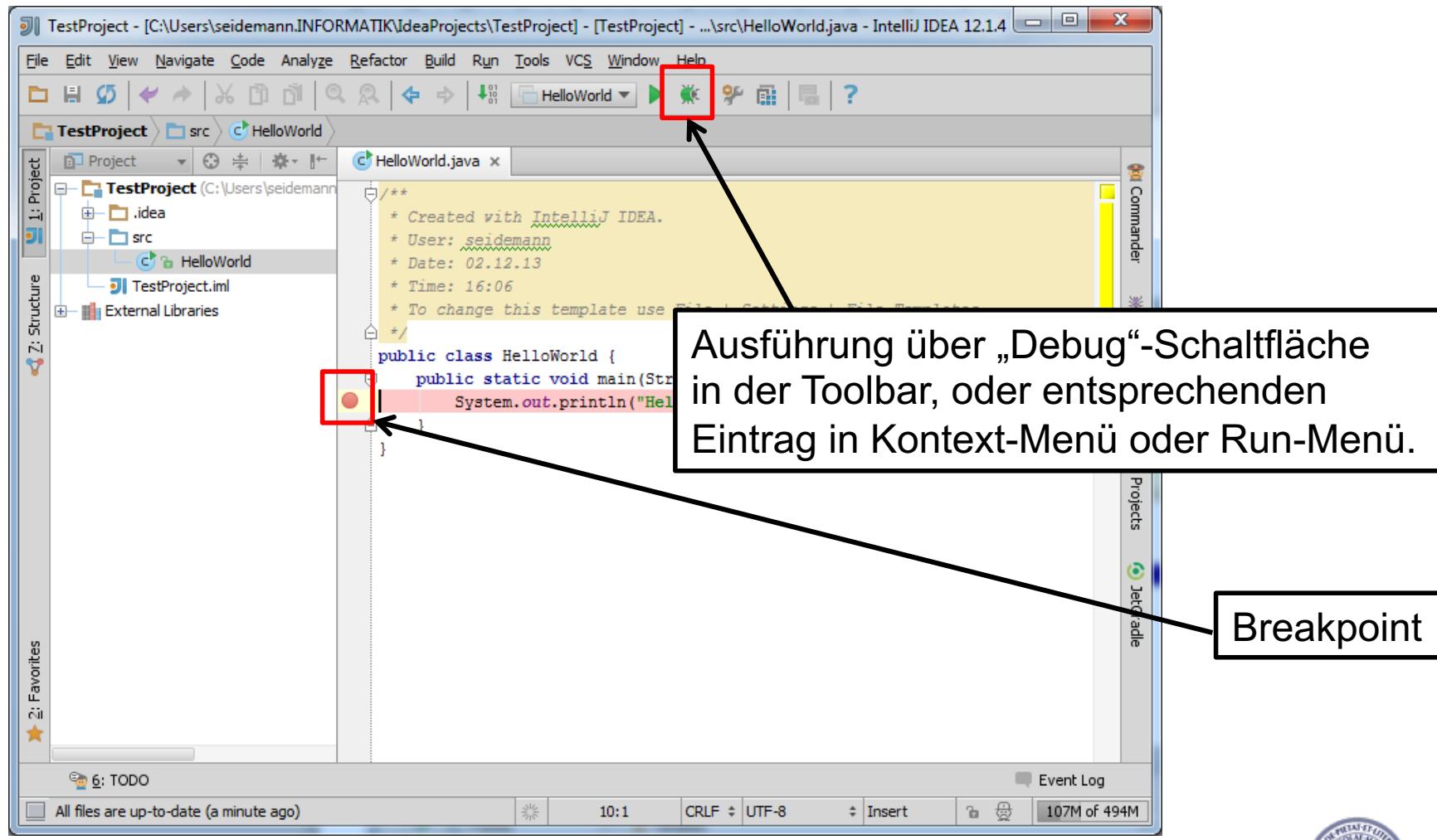


Wenn es schief geht: Debugging

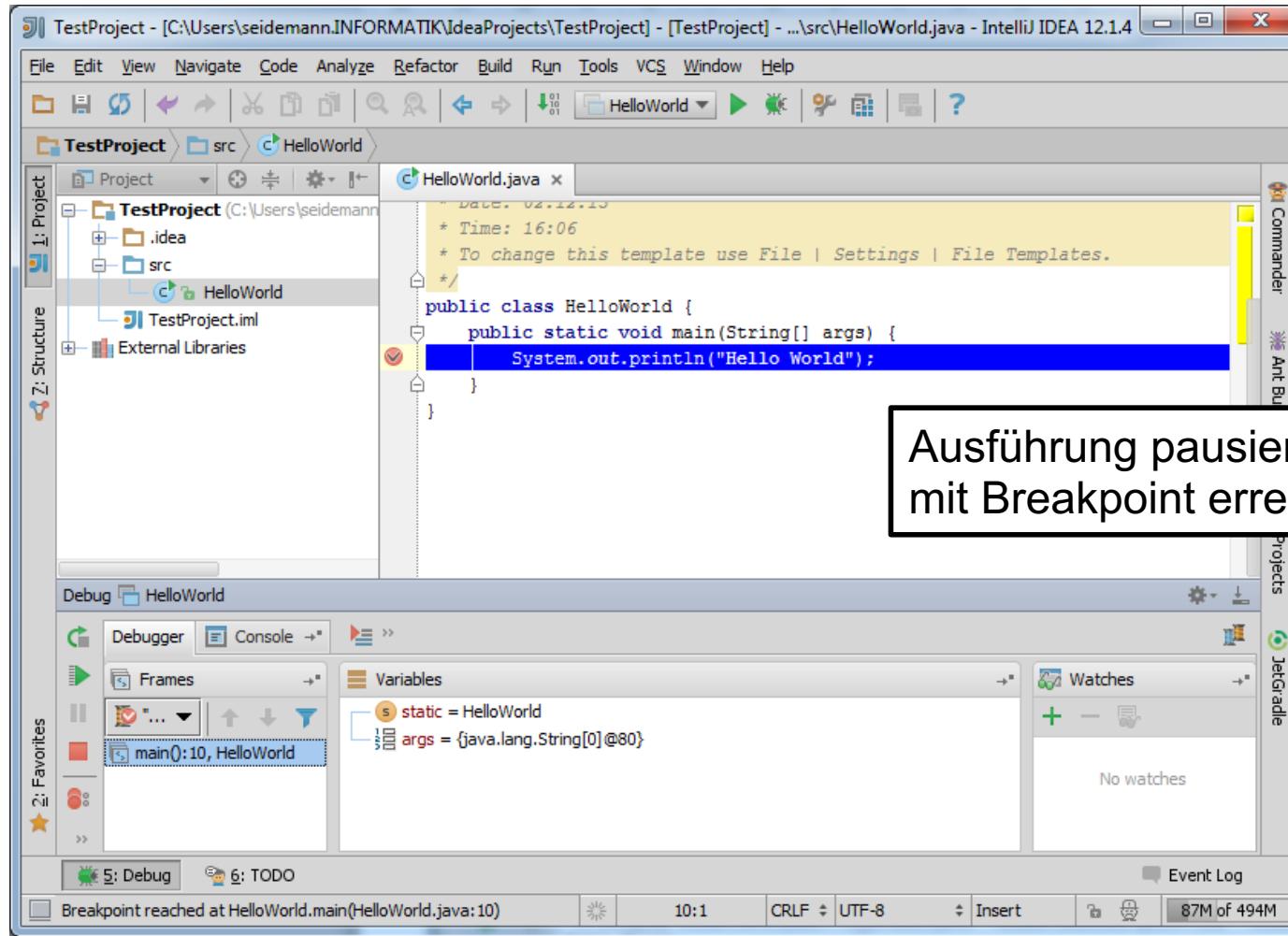
- Schlägt ein Testfall fehl, beginnt die Fehlersuche
- Old School:
 - Ausgabe von Variablen auf der Konsole → sehr umständlich
 - Nicht empfehlenswert: das Programm wird zur Fehlersuche modifiziert!
- Debugger
 - Teil der Entwicklungsumgebung
 - Kann mit Hilfe von **Breakpoints** (Haltepunkten) die Ausführung des Programms gezielt an einer gewünschten Stelle unterbrechen
 - Nachvollziehen des Programmflusses
 - Navigation in der Programmausführung (Hinein- und Herausspringen aus Methoden)
 - Beobachtung und ad-hoc-Veränderung von Variablen während der Laufzeit



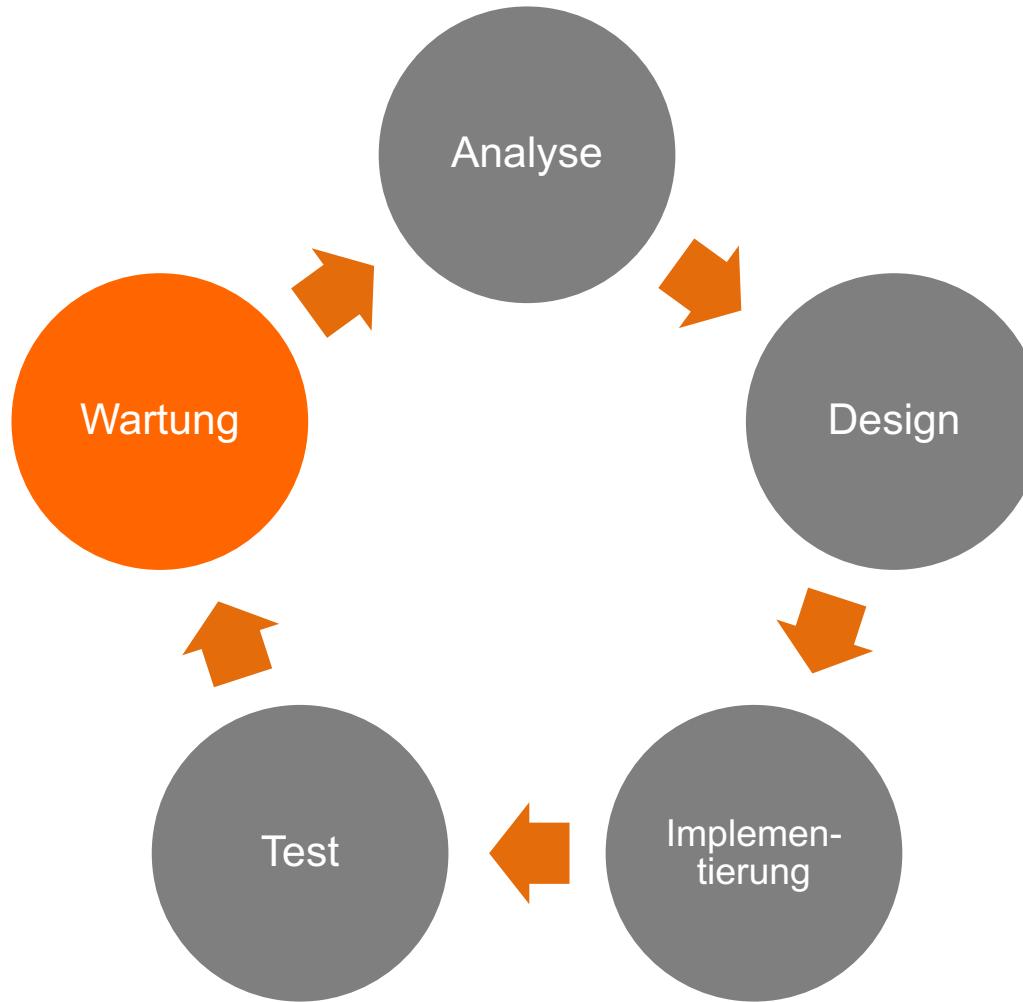
Debugger in IntelliJ



Debugger in IntelliJ



Übersicht



Auslieferung: JAR-Dateien

- Nachdem alle Tests erfolgreich bestanden wurden, kann das Programm ausgeliefert werden
- In Java: ausführbare **JAR-Datei** erzeugen
 - JAR-Dateien sind im Wesentlichen ZIP-Dateien, die neben den übersetzten Class-Dateien eine **Manifest-Datei** enthält
 - Manifest-Datei spezifiziert Metadaten, wie z.B. den Classpath und die Klasse, die die **main**-Methode enthält
- Ausführung der JAR-Datei:
`java –jar MeineJar.jar`

JAR-Manifest

- Die Metadaten befinden sich in der Datei „META-INF/MANIFEST.MF“
- Auszug aus einer Manifest-Datei

Manifest-Version: 1.0
Main-Class: HelloWorld
Class-Path: <Jar1>

enthält die **main**-Methode

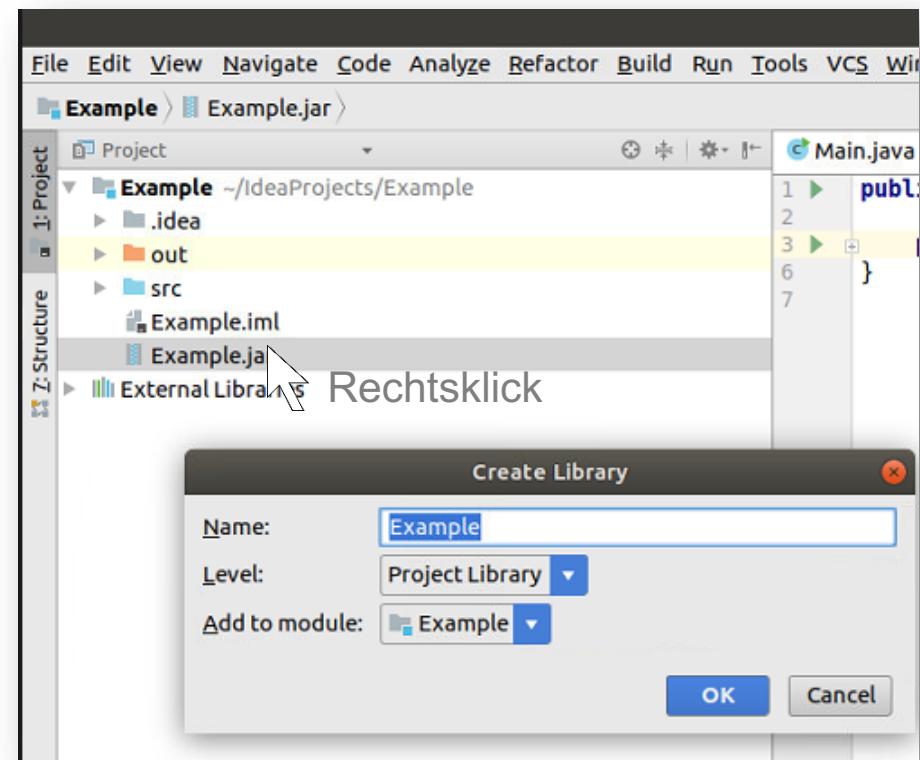
Bibliotheken

- JAR-Dateien dienen nicht nur als ausführbare Programme, sondern auch als Bibliotheken
 - Idee dahinter: nicht ständig das Rad neu erfinden, sondern bestehenden Code nutzen
 - Klassen werden in JAR-Dateien gebündelt und können in anderen Projekten eingebunden und genutzt werden
- Problem
 - Wir müssen dem Java-Compiler mitteilen, in welchen Bibliotheken (JAR-Dateien) die referenzierten Klassen liegen.
 - CLASSPATH muss gesetzt werden (siehe auch Kapitel 6)

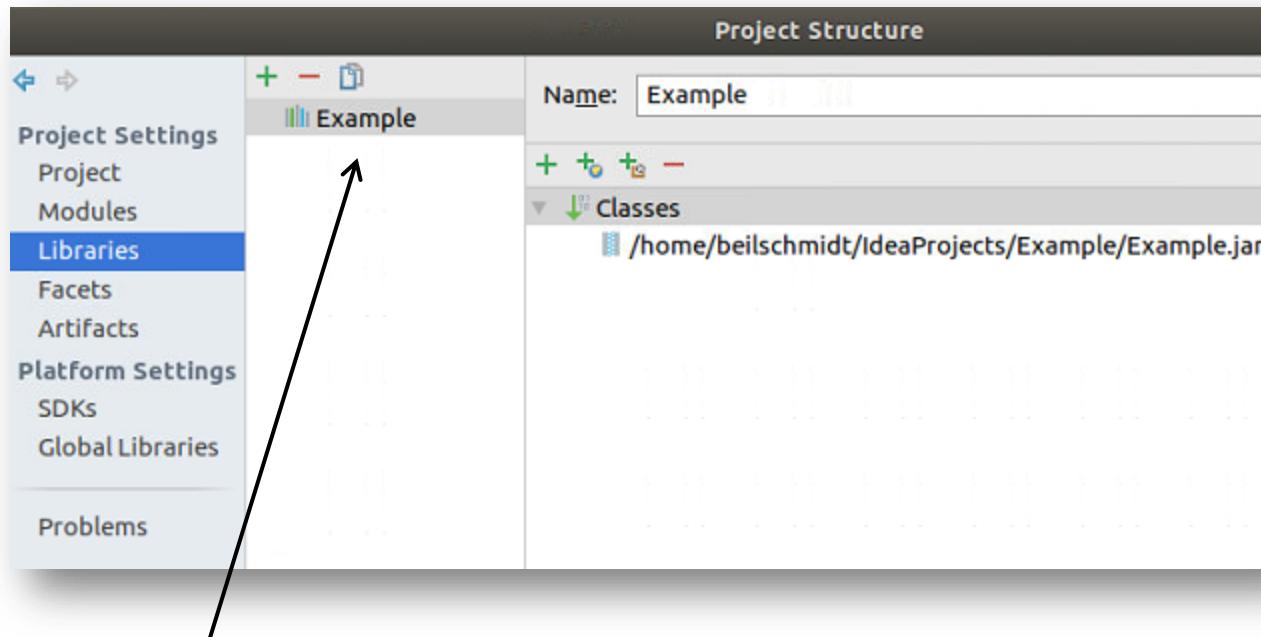


Classpath

- Compiler muss Programmcode finden
- Classpath definiert die Verzeichnisse, in denen gesucht wird
 - Ausführungsverzeichnis
 - Umgebungsvariable CLASSPATH
 - Optionaler Parameter -classpath
- In IntelliJ:
 - Rechtsklick → Kontextmenü -> Add as library ...



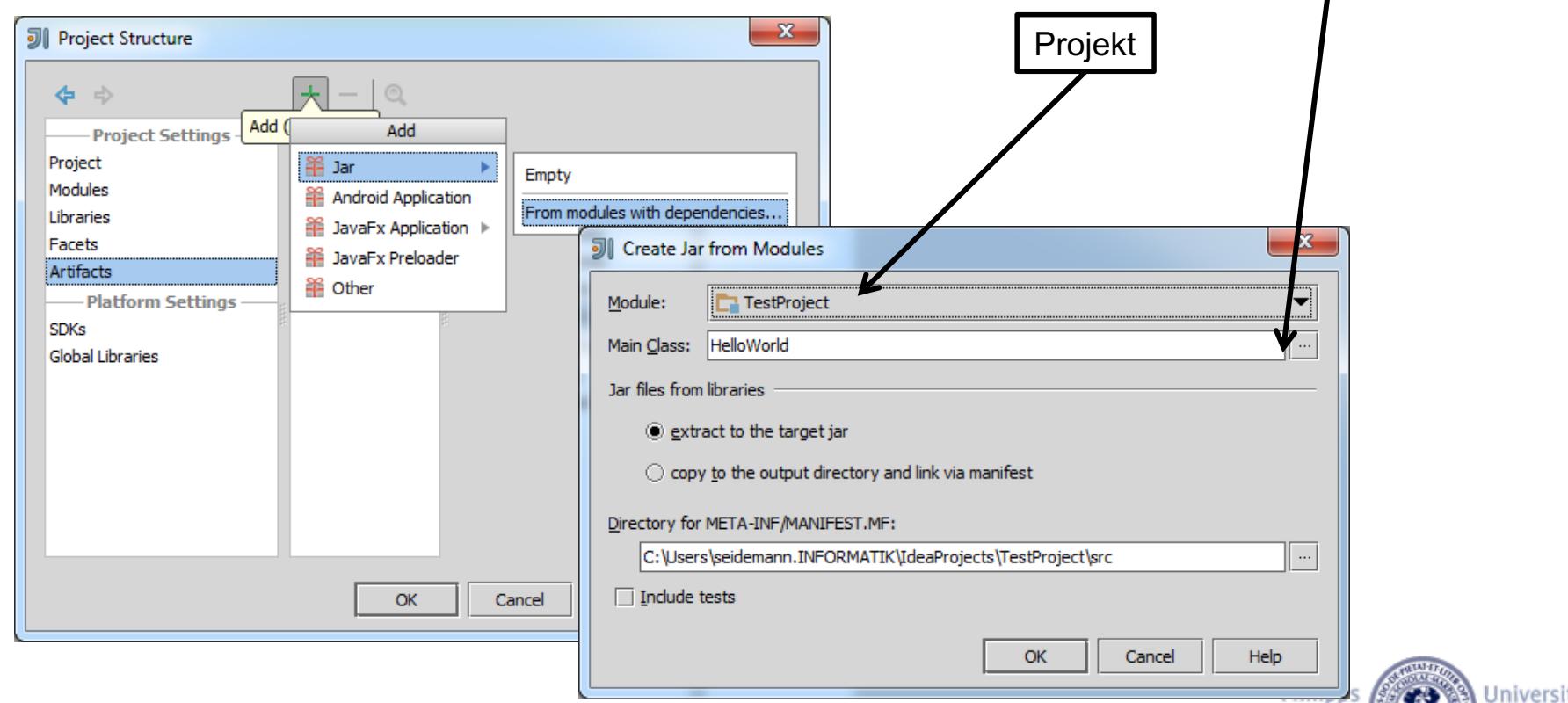
Classpath (2)



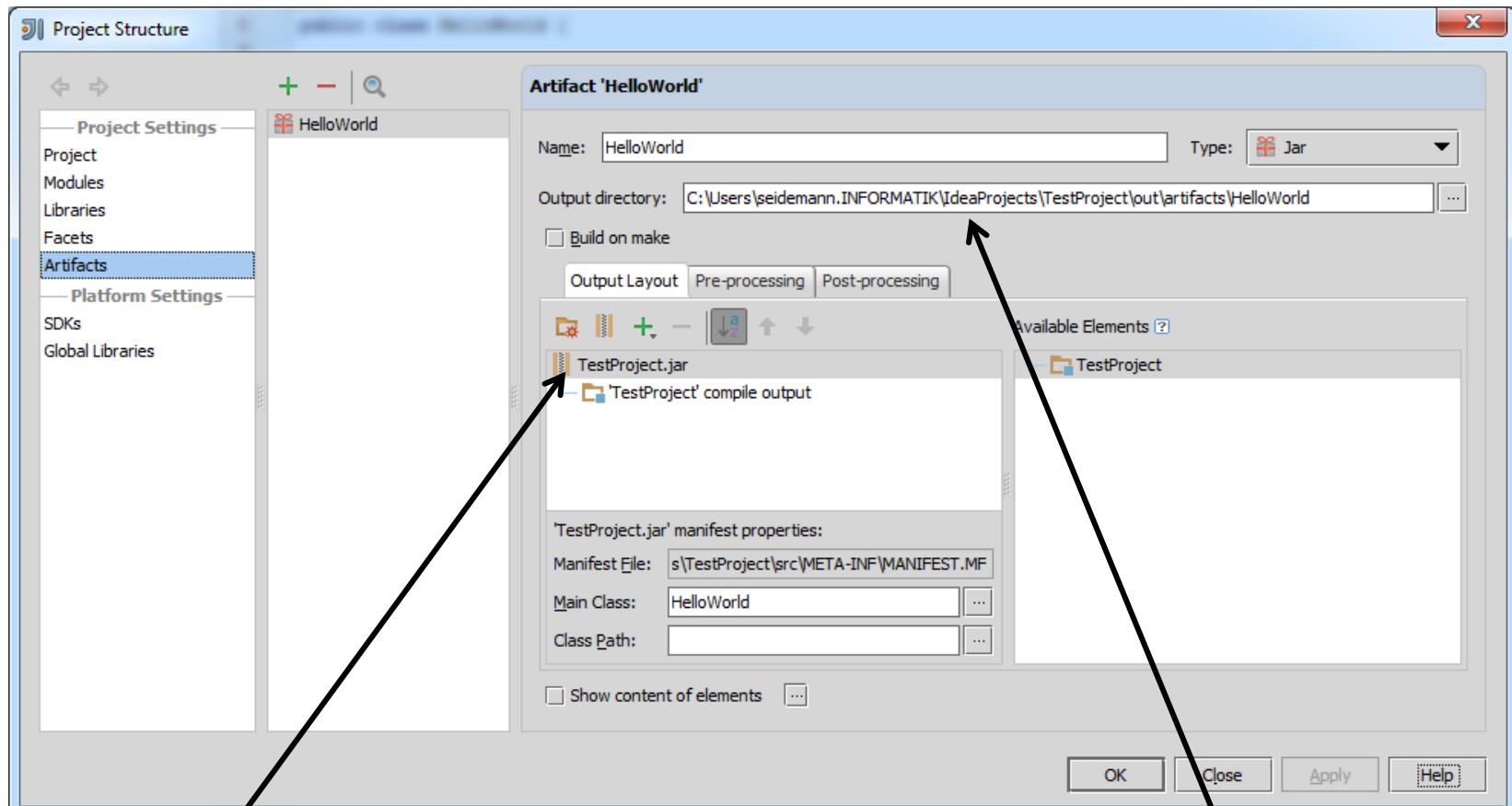
Die unter "Libraries" hinzugefügten Bibliotheken werden automatisch auch dem CLASSPATH hinzugefügt.

JAR-Datei mit IntelliJ

- Rechtsklick auf Projekt →  Open Module Settings → Artifacts



JAR-Datei mit IntelliJ



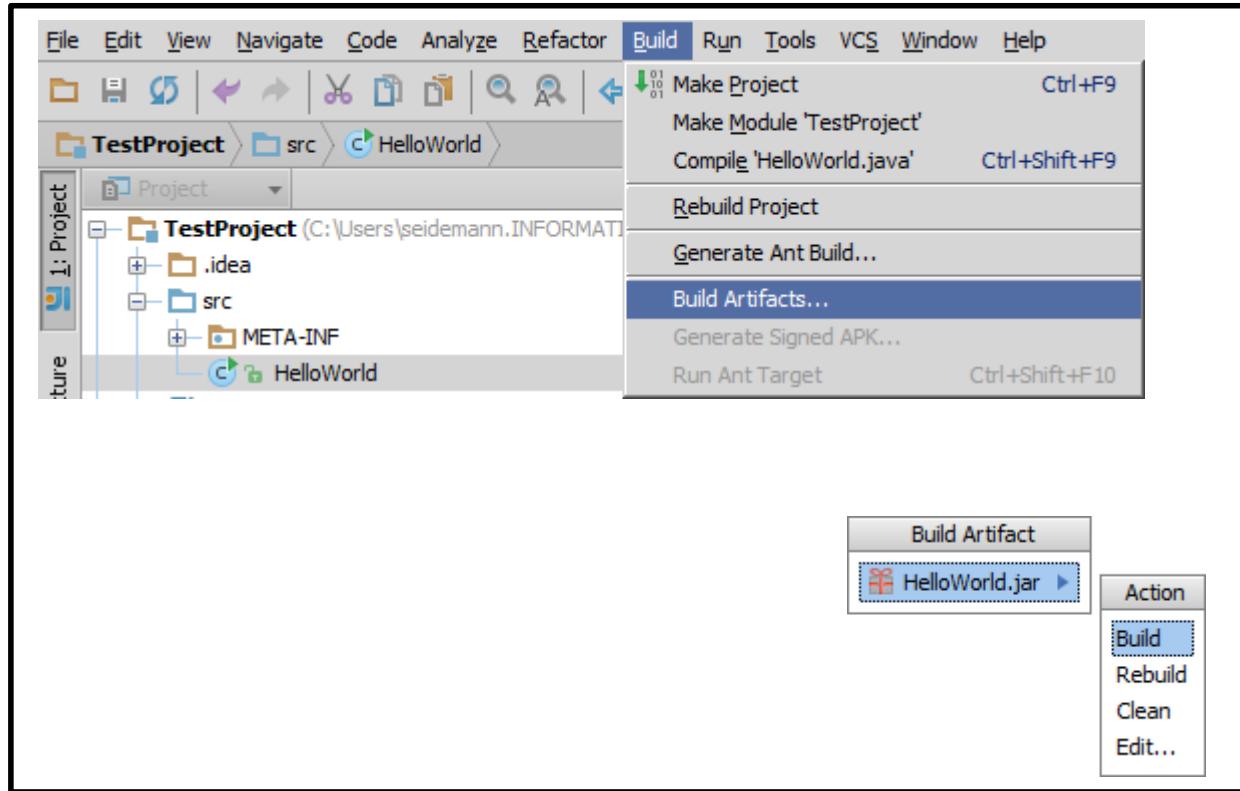
Name der JAR-Datei

Ausgabepfad der JAR-Datei

wofür ist Build Artifacts

JAR-Datei mit IntelliJ

- JAR-Datei muss nach einer Änderung neu erstellt werden



Besser: Build-System

- Maven, Gradle, etc. bilden Pfad von...
 - Abhängigkeiten (Bibliotheken)
 - Kompilieren
 - Testen
 - Ausliefern (Jar)
- ...ab.
- Spezifikation in einer Datei (XML, Gradle Script DSL)
- Arbeitsfluss ist IDE-unabhängig
- Abhängigkeiten aus Online-Repositories (search.maven.org)

Jedoch nicht Inhalt dieser Vorlesung.

Wartung

- Nach der ersten Auslieferung: Software muss gewartet werden, denn ...
 - Früher oder später tauchen Fehler auf, die behoben werden müssen, oder
 - Der Kunde möchte die Software um neue Funktionen erweitern, oder
 - Bestimmte Teile des Codes sollen aus Performancegründen überarbeitet werden, oder
 - ...
- Software muss gut zu warten sein
 - Umfassende Dokumentation
 - Kommentare an kritischen Stellen
 - Keine „schmutzigen Tricks“ (der Kollege muss es auch verstehen)
 - Einhaltung von **Konventionen**



Konventionen in Java

- Konventionen sind nicht zwingend vorgeschrieben (der Compiler akzeptiert auch schlecht formatierten den Code), gehören aber zum guten Ton
 - Wenn ein Java-Entwickler den Bezeichner „MyNumber“ liest, vermutet er eine Klasse und keine Integer-Variable!
- Bezeichner: in Java wird CamelCase verwendet
 - Variablen, Methoden und Felder beginnen mit Kleinbuchstaben
 - Klassen beginnen mit einem Großbuchstaben
 - Pakete beginnen hinter einem Punkt mit Kleinbuchstaben: z.B. java.lang
- <http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html>

8. Schnittstellen in Java

- Problem
 - Bei der Erstellung von *großen Programmpaketen* sollten Klassen **wiederverwendet werden**.
 - Typischerweise werden dabei in einer Klasse Datenfelder vom Typ einer andere Klasse bereitgestellt.

```
public class Konto {  
    private String kontoNr;  
    private double kontoStand;  
    private Kunde k;           // k verweist auf ein Objekt der Klasse Kunde  
    ...  
}
```

- Dadurch wird die Klasse Konto von der Klasse Kunde abhängig!
 - Was passiert bei einer Änderung der Klasse Kunde?

8.1 Motivation

- Man stelle sich nun ein größeres Programm P vor, das aus sehr vielen, voneinander abhängigen Klassen besteht.
 - ➔ Dies ist keine gute Software, da kleine Änderungen immer wieder dazu führen, dass P neu übersetzt und getestet werden muss.
- Probleme bei zu vielen Abhängigkeiten im Programm
 - Schlechte Wiederverwendbarkeit einzelner Klassen
 - Klasse Konto kann nicht ohne Klasse Kunde verwendet werden.
 - Viele Fehler durch Änderungen
 - Änderungen in der Klasse Kunde kann dazu führen, dass Klasse Konto nicht mehr funktioniert.
 - Schlechte Erweiter- und Anpassbarkeit an neue Probleme
 - Schwierige Aufteilung eines Programms P in einzelne Teile, die unabhängig voneinander im Team entwickelt werden können.

Modulares Programmieren

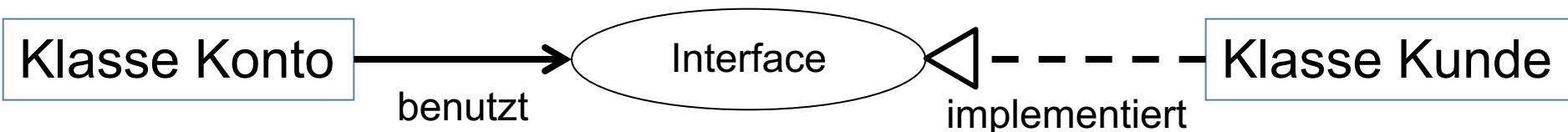
- Um diese Probleme zu beheben, wurde bereits in den 70er Jahren des letzten Jahrhunderts das modulare Programmieren postuliert.
 - Nikolaus Wirth hat dafür eine neue Programmiersprache Modula-2 entwickelt .
- Für **große Programmieraufgaben** ist es wünschenswert,
 - die Aufgabenstellung in sinnvolle Teile (**Module**) zu zerlegen,
 - diese Teile **unabhängig voneinander** zu programmieren, zu übersetzen und zu testen.
- Die **Vorteile** aus einer solchen Vorgehensweise sind :
 - Die Problemlösung wird **einfacher darstellbar** und **übersichtlicher**.
 - Mehrere Personen können **gleichzeitig** an einem Programm arbeiten.
 - Teile können leichter **getestet**, **verändert** und **gepflegt** werden.
 - Teile können in anderen Programmen **wiederverwendet** werden.



8.2 Schnittstellen in Java

- Modulares Programmieren wird in Java durch Interfaces realisiert.
 - Statt Klassen werden **Interfaces als Datentypen** verwendet.
→ Klassen sind damit unabhängig voneinander.
- Der deutsche Begriff für Interface ist **Schnittstelle**.

Schnittstellen als Vertrag



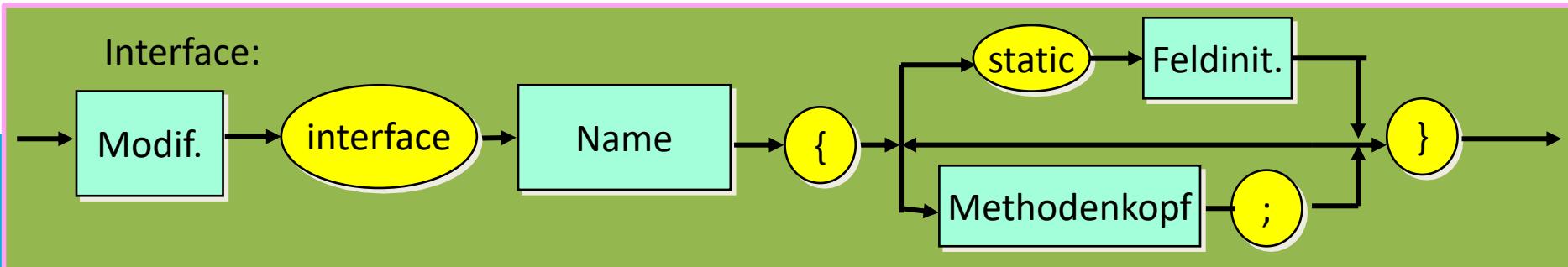
- Die Klasse Konto benutzt die Klasse Kunde, möchte aber unabhängig von der Klasse Kunde bleiben.
 - Es wird jetzt ein **Interface** vereinbart, das von der **Klasse Kunde implementiert** wird und von der **Klasse Konto verwendet** wird.
 - Ein Interface ist wie ein **Vertrag** zwischen den beiden **Klassen**, in dem die **Zusammenarbeit** geregelt wird.
 - Es wird dabei festgelegt, **was aber nicht wie** etwas geleistet werden soll.

Interface in Java - Eigenschaften

- Ein **Interface** ist **keine Klasse** - kann aber teilweise ähnliche Dienste anbieten.
 - Ein **Interface** kann in Java als **Typ von Variablen** verwendet werden.
 - Ein Interface hat jedoch **keine Konstruktoren** für die Objekterzeugung.
- Ein Interface ist primär eine Sammlung **abstrakter Methoden**
 - Eine abstrakte Methode besteht nur aus dem **Methodenkopf**, hat aber **keinen Rumpf**.
 - Zusätzlich können auch noch **Konstanten** (d.h. Felder mit den Schlüsselwörtern **final** und **static**) definiert werden.
 - Seit Java 8 können aber inzwischen auch implementierte Methoden Teil eines Interface sein. Wir werden diese sogenannten **Default-Methoden** später betrachten.
- Klassen können die abstrakten Methoden eines Interfaces implementieren.

Syntax eines Interface

- Ein vereinfachter Aufbau eines Interface hat die folgende Gestalt:



- Ähnlich zu Klassen kann einem Interface noch Schlüsselwörter vorangestellt werden, die den Zugriff auf die Schnittstelle definieren.

Beispiel

```
public interface RealFunc {  
    /** Eine Methode zur Auswertung von reellen Funktionen  
     * @param x Der Wert, an dem eine Funktion ausgewertet werden kann.  
     * @return Der Wert der Funktion an der Stelle x.  
     */  
    public double eval(double x);  
  
    /** Eine Methode zur Berechnung der Ableitung einer Funktion  
     * @return Liefert die Ableitung der Funktion.  
     */  
    public RealFunc derive();  
  
    /** Eine Methode zur Repräsentation einer Funktion als Zeichenkette.  
     * Diese Methode wird von System.out.print für die Ausgabe verwendet.  
     */  
    public String toString();  
}
```

Implementierung eines Interface

- Klassen können ein Interface implementieren.
 - Dazu muss hinter dem Klassennamen das Schlüsselwort **implements** und dann der Name der Interfaces folgen, die implementiert werden.
- Beispiel

```
public class Exp implements RealFunc {  
    ...  
}
```

- Diese Klassen müssen zu jeder abstrakten Methode des Interface auch eine Implementierung anbieten.
 - Sie können aber die im Interface definierten Konstanten nutzen.
- Es können gleichzeitig mehrere Interfaces (mit Komma getrennt) angegeben werden.
 - Dann müssen alle Interfaces auch implementiert werden (Beispiel später).

Die Klasse Exp

ich habe versucht zu übersetzen die Klase Exp hat nicht funktioniert such bitte online wie stackoverflow hier implementiert die Klasse Exp das Interface RealFunc

```
public class Exp implements RealFunc {  
    private double a;  
    private double b;  
  
    public Exp(double f, double g) {          // Konstruktor  
        a = f;  
        b = g;  
    }  
  
    public double eval(double xval) {          // Implementierung der eval-Methode  
        return a*Math.exp(b*xval);  
    }  
  
    public String toString() {                // Implementierung der toString-Methode  
        return "" + a + "e^(" + b + "*x)";  
    }  
  
    public RealFunc derive() {                // Implementierung der derive-Methode.  
        return new Exp(a*b, b);  
    }  
}
```

Klassen mit mehreren Interfaces

- Klassen können kein, ein oder mehrere Interfaces implementieren.
- Betrachten wir folgendes Szenario
 - Die Klasse Exp soll zwei Interfaces implementieren:
 - Das bisherige Interface RealFunc mit den Methoden eval, derive und toString.
 - Ein weiteres Interface Integrable mit der Methode antiDerive().
 - In der Klasse Exp müssen dann nach dem Schlüsselwort implements beide Interfaces angegeben werden.

```
public class Exp implements RealFunc, Integrable {  
    ...  
}
```

- Im Rumpf der Klasse müssen die vier Methoden implementiert sein.

Interface als Datentyp

- Wir können statt Klassen auch Interfaces nutzen, um Variablen und Datenfelder zu deklarieren.

```
RealFunc rf;
```

D.h. wir können variablen vom Typ Interface deklarieren also eine Referenzvariable aber auch in der Klasse selbst können wir ein Datenfeld vom Typ der Interface deklarieren
- Diese Variablen können auf Objekte der Klassen verweisen, die das Interface implementieren.

```
rf = new Exp(2., 3.);
```
- Es können nun über diese Variablen alle Methoden aufgerufen werden, die im Interface angegeben wurden.
 - Tatsächlich werden beim Aufruf der Methoden, die Methoden des Objekts benutzt, auf das die Variable verweist.
 - Man spricht dann von *dynamischen Binden*.

Dynamisches Binden

- Es wird zur Laufzeit des Programms entschieden wird, welche konkrete Methode **beim Aufruf** ausgeführt wird.
 - Es wird die Methode ausgeführt, die in der Klasse des Objekts implementiert wurde.
- Beispiele

```
RealFunc rf = new Exp(2.,3.);  
double res;  
  
res = rf.eval(2.0); // Aufruf der Methode aus der Klasse Exp  
  
// Annahme die Klasse Polynom implementiert ebenfalls das Interface RealFunc  
  
rf = new Polynom(new double[]{1.,2.});  
  
res = rf.eval(2.0); // Aufruf der Methode aus der Klasse Polynom
```

Man kann nicht alles nutzen!

- Wird ein Objekt über eine Interface-Variable an, so stehen **ausschließlich nur die Methoden aus dem Interface** zur Verfügung.
 - Die Klasse selbst kann über weitere Methoden verfügen.
 - Z. B. könnte in der Polynom-Klasse eine Methode int getGrad() den höchsten Exponenten des Polynoms liefern (2 im Fall einer Parabel).

```
RealFunc rf = new Polynom(new double[]{1.,2.});  
  
double res = rf.eval(2.0); // Aufruf der Methode aus der Klasse Polynom  
  
int grad = rf.getGrad(); // Funktioniert so nicht !
```

Typkonvertierung hilft!

- Das Problem kann behoben werden, indem man ein Cast auf den gewünschten Typ durchführt.

```
RealFunc rf = new Polynom(new double[]{1.,2.});  
  
double res = rf.eval(2.0); // Aufruf der Methode aus der Klasse Polynom  
  
Polynom p = (Polynom) rf;    // Cast: RealFunc → Polynom  
int grad = p.getGrad();      // Jetzt funktioniert alles wieder.
```

- Wenn der gewünschte Typ jedoch nicht passt, bekommen wir ein richtiges Problem!
 - Das Programm wirft dann eine Exception und wird möglicherweise beendet.

Was ist jetzt der Vorteil?

- Da das Interface RealFunc durch die beiden Klassen Polynom und Exp implementiert wird, vereinheitlicht sich unser Testprogramm.

```
RealFunc p = new Exp(new Double(args[0]), new Double(args[1]));
RealFunc d = p.derive();
System.out.println(p);
System.out.println(d);
for (double elem: new double[]{1.0, 2.0, 3.0, 4.0, 5.0}) {
    res = p.eval(elem);
    System.out.println("f( " + elem + " ) = " + res);
}
```

- Mit Ausnahme der Konstruktoraufrufe ist das Codefragment komplett unabhängig von den KlassenExp und RealFunc.
 - Wir haben also fast die Unabhängigkeit sichergestellt.

Was ist jetzt der Vorteil?

- Da das Interface RealFunc durch die beiden Klassen Polynom und Exp implementiert wird, vereinheitlicht sich unser Testprogramm.

```
RealFunc p = new Exp(new Doub  
RealFunc d = p.derive();  
System.out.println(p);  
System.out.println(d);  
for (double elem: new double[] {  
    res = p.eval(elem);  
    System.out.println("f( " + elem + " ) = " + res);  
}
```

Gilt genauso für andere Programmteile, nicht nur Tests!

- Mit Ausnahme der Konstruktoraufrufe ist das Codefragment komplett unabhängig von den Klassen Exp und RealFunc.
 - Wir haben also fast die Unabhängigkeit sichergestellt.

Objekterzeugung in separaten Fabriken

Ich habe es nicht verstanden

- Soll dieser Makel der Abhängigkeit von Konstruktoren beseitigt werden, kann eine sogenannte **Factory** implementiert werden.

```
class RealFuncFactory {  
    public static RealFunc getRealFunc(String criteria, double[] args) {  
        if ( criteria.equals("exp") )  
            return new Exp(args[0], args[1]);  
        else  
            return new Polynom(args);  
    }  
}
```

- Diese Klasse lässt sich noch beliebig erweitern, wenn noch weitere Klassen hinzugefügt werden, welche die Schnittstelle RealFunc implementieren.

Interfaces aus der Java-Bibliothek

- Das Java-System bietet bereits eine Vielzahl vordefinierter Klassen und Interfaces an.
- Für geordnete Daten verwendet man meist das **Interface Comparable**, das eine Methode für zum Vergleichen von Objekten vorgibt.
- Das Ergebnis von `compareTo` liefert ein Ergebnis vom Typ int. Es gilt folgende Konvention:
 - Wenn `a.compareTo(b)` negativ ist, interpretiert man dies als $a < b$.
 - Wenn `a.compareTo(b)` 0 ist, interpretiert man dies als $a = b$.
 - Wenn `a.compareTo(b)` positiv ist, interpretiert man dies als $a > b$.
- Wir werden später auf diese und andere Interfaces aus der Java-Bibliothek noch genauer eingehen.

8.3 Default-Methoden in Interfaces

- Interfaces können neben abstrakten Methoden auch **implementierte Methoden mit Rumpf** besitzen.
 - Diese Methoden werden als **Default-Methoden** bezeichnet.
 - Default-Methoden dürfen Methoden im Rumpf verwenden, die im Interface deklariert wurden
 - Die Methoden stehen dann auch in allen Klassen zur Verfügung, die dieses Interface implementieren.

Ich bin hier

Beispiel (1)

```
public interface RealFunc {  
    public double eval(double x);  
    public RealFunc derive();  
    public String toString();  
  
    default public double[] bulkEval(double[] arr) {  
        double[] res = new double[arr.length];  
        for (int i = 0; i < arr.length; i+=1)  
            res[i] = eval(arr[i]);  
        return res;  
    }  
}
```

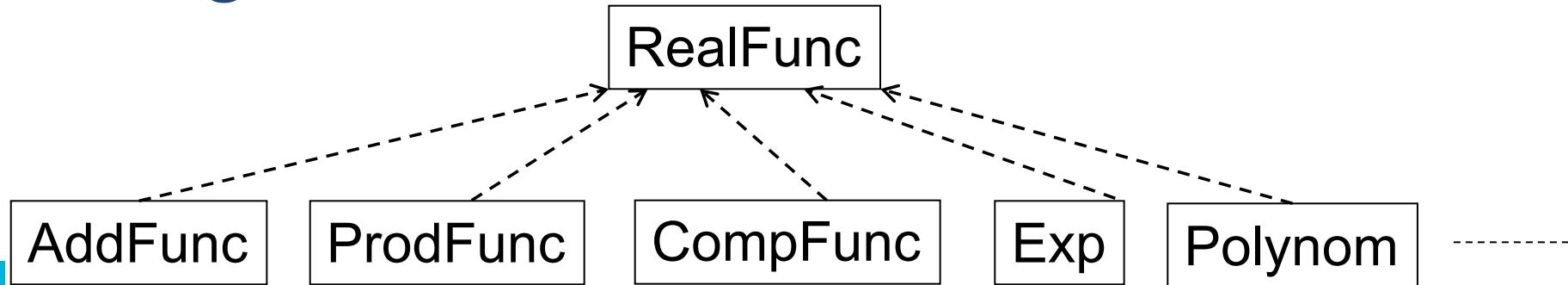
- In der Schnittstelle RealFunc soll eine Methode hinzugefügt werden, um die Funktion für jeden Wert eines double-Arrays auszuführen und die Ergebnisse als Array zu liefern.
- Diese Methode kann komplett mit Hilfe der anderen Methoden aus der Schnittstelle implementiert werden.

Beispiel (2)

- Damit kann man die Default-Methode bulkEval für Objekte der Klassen Exp nutzen.

```
public static void main(String[] args){  
    RealFunc q = new Exp(2.0, 3.0);  
  
    double[] rarr = q.bulkEval(new double[]{0, 1, 2, 3, 4});  
  
    for (double y:rarr)  
  
        System.out.println("Wert " + y);  
}
```

Programm mit vielen Klassen



- Schnittstelle RealFunc kann von vielen Klassen implementiert werden.
 - Exp – Klasse der Exponentialfunktionen
 - Polynom – Klasse der Polynome $a_0 + a_1 \cdot x^1 + \dots + a_n \cdot x^n$
 - Die drei Klassen auf der linken Seite stellen für Funktionen $f(x)$ und $g(x)$ folgende Funktionen zur Verfügung.
 - AddFunc $f(x) + g(x)$
 - ProdFunc $f(x) * g(x)$
 - CompFunc $f(g(x))$

Klasse AddFunc

```
public class AddFunc implements RealFunc {  
    private RealFunc left;  
    private RealFunc right;  
  
    public AddFunc(RealFunc f, RealFunc g) {  
        left = f;  
        right = g;  
    }  
  
    public double eval(double x) {  
        return left.eval(x) + right.eval(x);  
    }  
  
    public RealFunc derive() {  
        return new AddFunc(left.derive(), right.derive());  
    }  
  
    public String toString(){  
        return left.toString() + " + " + right.toString();  
    }  
}
```

Ableitungsregel:
 $(f+g)' = f' + g'$



Klasse ProdFunc

```

public class ProdFunc implements RealFunc {
    private RealFunc left;
    private RealFunc right;

    public ProdFunc(RealFunc f, RealFunc g) {
        left = f;
        right = g;
    }

    public double eval(double x) {
        return left.eval(x) * right.eval(x);
    }

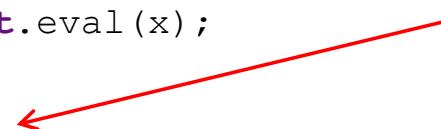
    public RealFunc derive() {
        return new AddFunc(new ProdFunc(left.derive(), right),
                          new ProdFunc(left, right.derive()));
    }

    public String toString() {
        return "(" + left.toString() + " * (" + right.toString() + ")";
    }
}

```

weil eine Addition laut Ableitungsregel gilt muss wir irgendwie $(f' * g)$ und $(f * g')$ die wir darstellen können durch den ProdFunc Konstruktor und wir brauchen den AddFunc Konstruktor wir brauchen AddFunc Konstruktor aber dürfen wir einfach so AddFunc Konstruktor so nutzen obwohl es keine Abhängigkeit gibt zwischen AddFunc und ProdFunc

Ableitungsregel:
 $(f*g)' = f'*g + f*g'$



Zusammenfassung

- Interfaces sind in Java ein **wichtiges** Konzept, um Klassen unabhängig voneinander zu machen.
 - Interfaces als Datentypen bei der Variablen Deklaration
- Interfaces können durch Klassen implementiert werden.
 - Eine Klasse kann mehrere Interfaces implementieren.
- Dynamisches Binden
 - Wird eine Methode über eine Interface-Variable aufgerufen, wird die konkrete Methode durch die Klasse des Objekts bestimmt.
- Factory-Klassen
 - Unabhängigkeit von den Konstruktoren der Klasse, in dem wir zu einem Interface noch eine solche Klasse bereitstellen.
- Default-Methoden

Das ist nicht möglich bei Vererbung
also eine Klasse kann nur eine Klasse
vererben aber mehrere Interface
implementieren

9. Die Klasse String

- Übersicht
 - Strings, String-Objekte, Literale
 - Stringerzeugung
 - Konkatenation von Strings
 - Einige Methoden für Strings
 - Vergleiche von Strings
 - Strings und char-Arrays
 - StringBuilder,
 - Die Methode **format**

Motivation

- Die Informatik beschäftigt sich sehr oft mit der **Verarbeitung von Zeichenketten.**
 - Analyse von Emails
 - Twitter-Nachrichten
- Die **Klasse String** bietet als Datentyp **nicht-veränderbare Zeichenketten** mit folgenden Diensten an.
 - Deklaration von Variablen des Typs String
 - Diverse Konstruktoren zur Erzeugung
 - Viele Hilfsmethoden, um z. B.
 - die Länge eines Strings festzustellen: **length()**
 - zwei Strings auf Gleichheit zu testen: **equals()**
 - das i-te Zeichen in einem String zu lesen: **charAt()**

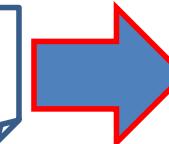
String-Erzeugung

- Man kann eine Variable/ ein Feld wie üblich definieren: `String bsp;`
- Meist nutzt man aber eine Definition mit Initialisierung:
 - Dabei erhält `gruss1` eine Referenz auf das Objekt `"Halli hallo"`
- Man kann auch die üblichen Konstruktoren nutzen
 - Dieser Konstruktor erzeugt eine Referenz auf ein leeres String-Objekt.
 - Die gleiche Wirkung hat: `String leer2 = "";`
 - Es gibt auch einen Konstruktor, mit einem String-Parameter
 - Dieser erzeugt eine Referenz auf eine Kopie des angegebenen Strings.

Konkatenation von Strings

- Für die Konkatenation von Strings wird der „+“ – Operator genutzt.
 - "Hallo" + " Welt" ergibt: "Hallo Welt"
 - "Bitte" + "nicht" + "stören" ergibt: "Bitte nicht stören"
- Bei der Konkatenation wird jeweils ein neues zusammengesetztes String-Objekt gebildet.
- Bei der Konkatenation wird, falls möglich, automatisch eine Umwandlung vorgenommen. Die Umwandlung erfolgt von links nach rechts:
 - "Hallo" + 1 ergibt: "Hallo1"
 - "Hallo" + 1 + 1 ergibt: "Hallo11"
 - Erst wird die erste 1 umgewandelt und konkateniert, dann die zweite 1 ...
 - "Hallo" + (1 + 1) ergibt: "Hallo2"
 - Die Klammerung bewirkt das erst addiert und dann umgewandelt wird

```
System.out.println("1 und 1 ist "+(1+1));  
System.out.println("1 und 1 ist "+1+1);
```



```
1 und 1 ist 2  
1 und 1 ist 11
```

Einige Methoden

- Die Länge eines Strings:

```
String gruss1 = "Halli hallo";
System.out.println(gruss1.length());
String gruss2 = new String("Halli hallo");
System.out.println(gruss2.length());
String leer1 = new String();
System.out.println(leer1.length());
String leer2 = "";
System.out.println(leer2.length());
System.out.println(" ".length());
```

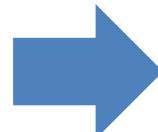
11
11
0
0
1

- *Zeichen an Position:* `System.out.println(gruss1.charAt(4));` → i
- *Index von Zeichen:* `System.out.println(gruss1.indexOf('l'));` → 2
- *indexOf* gibt es auch in Varianten: Mit einem weiteren Parameter der die Position angibt, von der an gesucht werden soll.
- *Teilstring* `System.out.println(gruss1.substring(3,5) +
gruss1.substring(9));` → lilo
- Und viele andere mehr...

Vergleiche von Strings

- Wie für Objekte üblich werden beim Vergleich von zwei Objekten der Klasse String immer die **Referenzen** verglichen.
- Für den Vergleich der **Inhalte** von Strings sind in der Klasse String einige passende Methoden vordefiniert: **equals**, **equalsIgnoreCase**, **compareTo**, **compareTolgnoreCase**, **contains**.

```
String gruss1 = "Halli hallo";
String gruss2 = new String("Halli Hallo");
System.out.println(" gruss1 equals gruss2 ist " + gruss1.equals(gruss2));
System.out.println(" gruss1 equalsIgnoreCase gruss2 ist " +
                  gruss1.equalsIgnoreCase(gruss2));
```



gruss1 equals gruss2 ist false
gruss1 equalsIgnoreCase gruss2 ist true

Vergleiche von Strings

- Die Methode `compareTo` vergleicht die Strings lexikografisch
 - Das Ergebnis ist 0, wenn sie gleich sind,
 - negativ, wenn der erste String lexikografisch kleiner ist als der zweite
 - positiv, wenn der erste String lexikografisch grösser ist als der zweite
- Die Methode `compareTo` stammt aus dem Interface Comparable, das von String implementiert wird
 - **ACHTUNG:** trotzdem darf der `compareTo` Methode für ein String-Objekt nur ein anderes String-Objekt übergeben werden

Strings sind nicht veränderbar

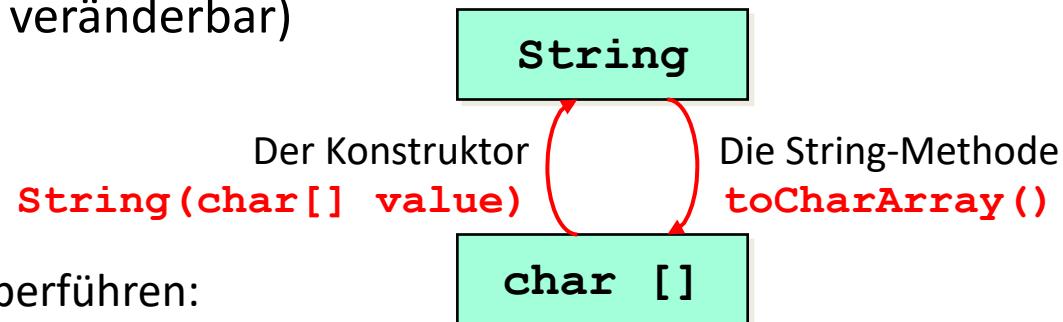
- Die Klasse String bietet nur Methoden, um den Inhalt auszulesen
- Wenn ein Text verändert werden soll, muss ein neues String-Objekt mit dem veränderten Wert erzeugt werden

```
class Reverse{  
  
    /** Methode zum Umdrehen von einer Zeichenkette.  
     * @param s Zeichkette, die umgedreht werden soll.  
     * @return Umgedrehte Zeichenkette  
     */  
    public static String reverseInefficient(String s) {  
        String newString = "";  
        for (int i = 0; i < s.length(); i++) {  
            newString = s.charAt(i) + newString;  
        }  
        return newString;  
    }  
    ...  
}
```

Konkatenation erzeugt einen String.
Im Beispiel: in jedem Schleifendurchlauf.

Strings und char Arrays (1)

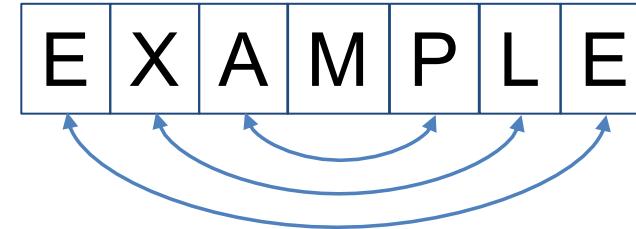
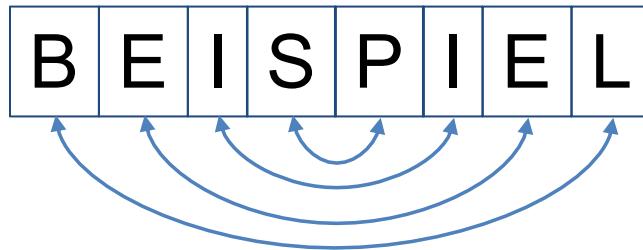
- Umweg über char-Array (das ist veränderbar)
- Strings und char-Arrays haben zwar viele Ähnlichkeiten, sind aber verschiedene Typen.
 - Man kann sie aber ineinander überführen:



```
...
/** Methode zum Umdrehen von einer Zeichenkette
 * @param s Zeichkette, die umgedreht werden soll.
 * @return Umgedrehte Zeichenkette
 */
static String reverse(String s){
    char[] car = s.toCharArray(); // String → char[]
    int lastIndex = car.length - 1;
    for (int i = 0; i < (lastIndex + 1)/2; i++)
        swap(car, i, lastIndex - i);
    return new String(car);           // char[] → String
}
```

Strings und char Arrays (2)

```
...
/** Methode zum Umdrehen von einer Zeichenkette
 * @param s Zeichkette, die umgedreht werden soll.
 * @return Umgedrehte Zeichenkette
 */
static String reverse(String s){
    char[] car = s.toCharArray(); // String → char[]
    int lastIndex = car.length - 1;
    for (int i = 0; i < (lastIndex + 1)/2; i++)
        swap(car, i, lastIndex - i);
    return new String(car);           // char[] → String
}
```



- *Noch besser wäre gewesen, wenn wir zuvor geschaut hätten, ob eine solche Funktion bereits in einer anderen Klasse implementiert wurde.*

Strings und char Arrays (2)

```
...
/** Die Methode vertauscht in einem char-Array zwei Zeichen.
 * @param a das char-Array
 * @param i erste Position im Array a
 * @param k zweite Position im Array a
 */
static void swap(char[] a, int i, int k){
    char t = a[i];
    a[i]=a[k];
    a[k]=t;
}
```

- *Noch besser wäre gewesen, wenn wir zuvor geschaut hätten, ob eine solche Funktion bereits in einer anderen Klasse implementiert wurde.*

Die Klasse StringBuilder

- Die Klasse **StringBuilder** erlaubt Zeichenketten zu verändern, ohne dabei immer wieder neue Objekte zu erzeugen. Die wesentlichen Operationen sind dabei:
 - **append** zum Anhängen am Ende,
 - **insert** zum Einfügen an einer beliebigen Stelle,
 - **delete** zum Entfernen eines beliebigen Teilstrings.
- Objekte der Klasse **StringBuilder** haben eine variable **Kapazität**, um Zeichen bis zu der Größe aufzunehmen.
 - Mit **trimToSize** kann die Kapazität explizit verkleinert werden, mit **ensureCapacity** kann sie explizit vergrößert werden.
 - Wenn die Kapazität nicht mehr ausreicht, wird automatisch zusätzlicher Speicher angefordert – in größeren Inkrementen.
- Die Klasse **StringBuilder** bietet auch eine Methode **reverse!**
 - **Wir nutzen diese Methode** und vermeiden somit eine Neuentwicklung.



Ausgabe von Zeichenketten

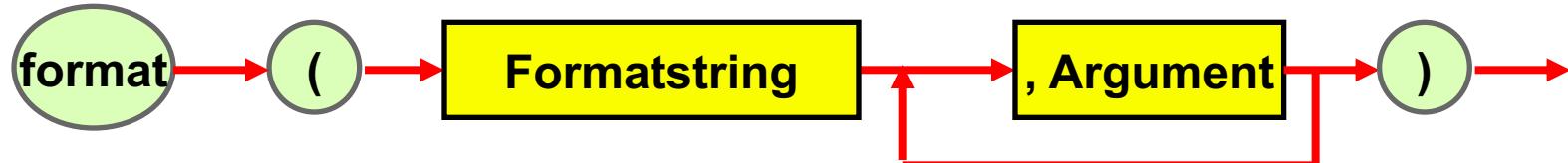
- Problem
 - Direkte Ausgabe einer Zeichenkette oder Zahl soll schöner werden.

```
System.out.println("Diese Zahl ist zu lang: " + 42.12345678901234);
```
 - Es gibt folgende zwei Arten das Format der Ausgabe zu ändern.
 - In der **Klasse PrintStream** gibt es die Methode **printf**.
 - In der **Klasse String** gibt es die static-Methode **format**.
 - Beide Methoden beruhen auf dem gleichen Prinzip, das man eine sogenannte **Formatzeichenkette** als Parameter verwendet.

Die Methode format

- Zweck
 - Formatierte Ausgabe von Daten.

- Syntax



- Die statische Methode liefert als Ergebnis die formatierte Zeichenkette zurück und kann beliebig viele Parameter besitzen.

Aufbau eines Formatstrings

- Der Formatstring setzt sich aus **Text und Formatanweisungen** zusammen.
- Eine Formatanweisung besteht aus einem **Prozentzeichen und einem weiteren Zeichen für den Datentyp**.
 - Die Formatanweisung wird durch das nächste noch nicht benutzte Argument der format-Methode ersetzt.
- Formatanweisungen für spezifische Datentypen
 - ganze Zahlen
 - Gleitpunktzahlen
 - Datum

Formatstring für Zahlen

- Ganze Zahlen (Beispiele)

- %d Ausgabe als Dezimalzahl
- %o Ausgabe als Oktalzahl ohne Vorzeichen
- %x Ausgabe als Hexadezimalzahl

- Gleitpunktzahlen (Beispiele)

- %f Ausgabe von float und double im Format [-]m.d (# Nachpunktstellen = 6).
- %e Ausgabe von float und double im Format [-]m.dex (#Nachpunktstellen= 6)
- %g Ausgabe von float und double im Format %e oder %f, je nach Größe des Exponenten

Formatstring für Zahlen

- Ganze Zahlen (Beispiele)

- %d Ausgabe als Dezimalzahl
- %o Ausgabe als Oktalzahl ohne Vorzeichen
- %x Ausgabe als Hexadezimalzahl

- Gleitpunktzahlen (Beispiele)

- %f Ausgabe von float und double im Format
[-]m.d (#x) .
x: Exponent
- %e Ausgabe von float und double im Format
[-]m.dex (#Nachpunktstellen= 6)
- %g Ausgabe von float und double im Format %e oder %f,
e nach dem Exponenten

m: Mantisse (Vor-Komma-Stellen)

d: Dezimalstellen

Anstelle führender Nullen werden Leerzeichen vorangestellt.

Beispiele

```
public static void main(String[] args) {
    int vi = 42;
    String svi = String.format(
        "Der Wert von vi ist dezimal %3d, oktal %3o und hexadezimal %3x " ,
        vi, vi, vi);
    System.out.println(svi);

    String test = String.format(
        "Einige Zufallszahlen: %10.4f %10.4f %10.4f %10.4f %10.4f " ,
        Math.random(), Math.random(), Math.random(), Math.random(),
        Math.random());
    System.out.println(test);
}
```

Der Wert von vi ist dezimal 42, oktal 52 und hexadezimal 2a
Einige Zufallszahlen: 0,0861 0,1296 0,0395 0,4423 0,3328

Beispiele

```
public class Main {  
    public static void main(String[] args) {  
  
        String svi = String.format( // Mindestens 3 Zeichen, evtl. mit  
            "Der Wert von vi ist dezimal %3d, ", // vorangestellten Leerzeichen.  
            vi, vi, vi);  
        System.out.println(svi);  
  
        String test = String.format( // Mindestens 10 Zeichen (evt.  
            "Einige Zufallszahlen: %10.4f %10.4f %10.4f %10.4f %10.4f " ,  
            Math.random(), Math.random(), Math.random(), Math.random(),  
            Math.random());  
        System.out.println(test);  
    }  
}
```

Mindestens 10 Zeichen (evt.
Leerzeichen vorangestellt),
davon 4 für Nachkommastellen.

Verwendet lokale Einstellungen für
Formate. Daher wird z.B. hier das Komma
als Dezimaltrennzeichen ausgegeben.

Der Wert von vi ist dezimal 12, oktal 52 und hexadezimal 2a
Einige Zufallszahlen: 0,0861 0,1296 0,0395 0,4423 0,3328

Zusammenfassung

- Die Klasse String kurz vorgestellt.
 - Objekte repräsentieren unveränderbare Zeichenketten.
- Funktionalität der Klasse
 - Konstruktoren
 - Diverse Methoden
 - Formatieren von Zeichenketten
 - **Suche nach Zeichenketten in einer Zeichenkette (siehe Übungen)**
- Assoziierte Klassen
 - StringBuilder

Live Vote

PIN: 8VX3

x

<https://ilias.uni-marburg.de/vote/8VX3>

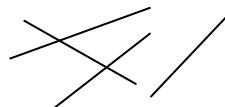


10. Klassenerweiterung

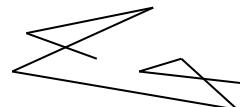
- Klassenerweiterung
- Überschreiben von Methoden, Polymorphie
- Konstruktoren, Attribute
- Abstrakte Klassen

10.1 Motivation

- Bei der Entwicklung von Programmen, die aus mehreren Klassen bestehen, ist oft zu beobachten, dass Klassen sehr ähnlich zueinander sind.
- Betrachten wir dazu folgende Aufgabe:
 - In einem Projekt sollen Klassen zur Manipulation von geometrischen Objekten implementiert werden. Folgende Objekttypen sind von Interesse:



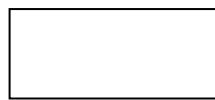
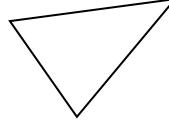
Kantenmenge



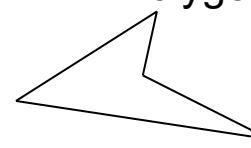
Kantenzug



Polygon

orthogonales Rechteck
(O-Rechteck)

Dreieck



Viereck

- Wir beschließen für jeden Objekttyp eine Klasse zu implementieren.
 - Alle Klassen sollen eine Methode `length()` haben, um die Gesamtlänge aller Kanten zu berechnen.
 - Für Polygone soll noch eine Methode `area()` den Flächeninhalt liefern.

Beispiel des Codes (1. Versuch)

```
// Kantenmenge
public class EdgeSet{
    private Edge[] edges;

...
    public double length() {
        double sum = 0.0;
        for (Edge e:edges)
            sum += e.length();
        return sum;
    }
}
```

```
public class Polygon {
    private Edge[] edges;
```

...

```
public double length() {
    double sum = 0.0;
    for (Edge e:edges)
        sum += e.length();
    return sum;
}
```

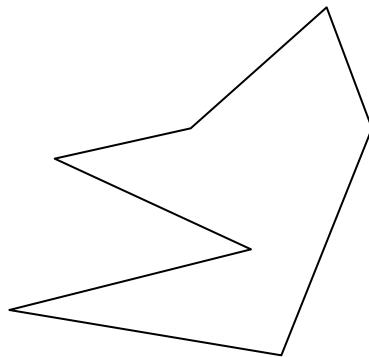
```
public double area() {
...
}
```

Diskussion

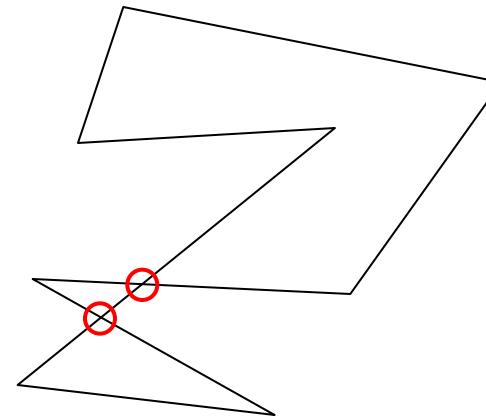
- Unsere beiden Klassen sehen sehr ähnlich aus. Die Methode `length()` ist in beiden Klassen gleich.
 - Dies würde auch bei der Implementierung der anderen Klassen der Fall sein.
 - Quellprogramm ist auf Grund der Redundanz **nicht wartungsfreundlich**.
- Aufgrund der Unabhängigkeit der Klassen kann **ein Polygon nicht dort verwendet werden, wo eine Kantenmenge erwartet wird**.
 - Aber ein Polygon kann als Spezialfall einer Kantenmenge mit folgenden Zusatzeigenschaften angesehen werden.
 - Der Anfangspunkt einer Kante ist der Endpunkt der Vorgängerkante.
 - Kanten schneiden sich nicht (abgesehen in den Endpunkten zwei aufeinanderfolgender Kanten).

Einschub: Begriff des Polygons

- Ein Polygon ist ein „**Vieleck**“ mit einer Folge von Kanten.
 - Zwei aufeinanderfolgende Kanten berühren sich in Ihren Endpunkten.
 - Ansonsten gibt es keine Schnitte zwischen Kanten.



ist ein Polygon



ist kein Polygon

Beispiel des Codes (2. Versuch)

```
public class OneClass {  
    private Edge[] edges;  
    boolean istPolygon;  
  
    public OneClass(Edge[] in, boolean poly) {  
        this.edges = new Edge[in.length];  
        istPolygon = poly;  
        for (int i = 0; i < in.length; i++) {  
            this.edges[i] = in[i];  
        }  
        if (poly)  
            // Hier müssen noch einige Überprüfungen erledigt werden.  
    }  
  
    public double length() {  
        double sum = 0.0;  
        for (Edge e:edges)  
            sum += e.length();  
        return sum;  
    }  
  
    public double area() {  
        // Wie kann man verhindern, dass die Methode für ein  
        // Objekt aufgerufen wird bei dem istPolygon „false“ ist??  
    }  
}
```



Diskussion

- Lösung mit einer Klasse OneClass
 - Die Methode length() wird nur noch einmal implementiert.
 - Wir können jetzt Objekte, die Polygone sind, auch nutzen, wenn nur ein EdgeSet erwartet wird.
 - Umgekehrt geht es aber leider auch!!
 - Wir müssen jetzt selbst aufpassen, dass gewisse Methoden, wie z. B. die Methode area(), nur für Polygone, aber nicht für Kantenmenge aufgerufen werden.
 - Das kann sehr leicht zu Fehlern in unserem Programm führen!
 - Wir benötigen zusätzlich eine Boolesche Variable, um zu unterscheiden, ob das Objekt ein Polygon oder eine Kantenmenge ist.
 - Dies verkompliziert an vielen Stellen die Programmierung
- **Beide bisherige Lösungen sind also nicht wirklich gut.**
 - Mit dem Konzept der Klassenerweiterung können wir aber diese Nachteile vermeiden.



10.2 Klassenerweiterung in Java

- Zunächst sollen folgende **vereinfachende Annahmen** gelten:
 - Klassen, Methoden und Datenfelder sind **public**
- Sei K eine Klasse. Dann lässt sich eine neue Klasse IsA_K

```
public class IsA_K extends K { ... }
```

erzeugen, wobei IsA_K **alle Objekteigenschaften** (Datenfelder **ohne** Schlüsselwort static und Objektmethoden) der Klasse K besitzt.

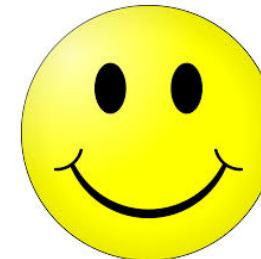
Neben diesen Eigenschaften kann IsA_K **weitere Eigenschaften** besitzen, die dann wie gewohnt **in die Klassendefinition mit aufgenommen werden**.

Beispiel des Codes (3. Versuch)

```
public class EdgeSet{  
    private Edge[] edges;  
  
    ...  
  
    public double length() {  
        double sum = 0.0;  
        for (Edge e:edges)  
            sum += e.length();  
        return sum;  
    }  
}
```



```
public class Polygon extends EdgeSet {  
    // edges bereits in EdgeSet definiert  
    // Methode length wird aus EdgeSet  
    // übernommen.  
  
    public Polygon(Edge[] edges) {  
        ...  
        // Details dazu später  
    }  
  
    public double area() {  
        ...  
    }  
}
```

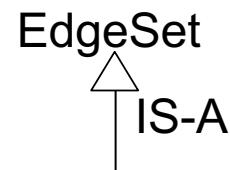


Verwendung der Klassen

- Wir können die Klassen wie bisher verwenden.
 - Erzeugen von Objekten
 - `Polygon poly = new Polygon(/* ggf. Parameter einfügen */);`
 - `EdgeSet edges = new EdgeSet(/* ggf. Parameter einfügen */);`
 - Folgende Methodenaufrufe sind möglich:
 - `double r1 = edges.length();`
 - `double r2 = poly.area();`
 - `double r3 = poly.length();`
 - Dies ist möglich, da die Klasse Polygon alles bekommt, was auch in der Klasse EdgeSet vorhanden ist.
 - ~~`double r4 = edges.area()`~~
 - Dies ist nicht möglich, da die Methode area nicht in der Klasse EdgeSet existiert. → Fehler fällt bereits dem Compiler auf!

Wichtige Begriffe

- Die erweiternde Klasse wird als **Unterklasse** (Subklasse) und die erweiterte Klasse als **Oberklasse** (Superklasse) bezeichnet.
- In der Literatur wird statt **Klassenerweiterung** auch der Begriff **Vererbung** verwendet.
 - In dieser Vorlesung werden wir Klassenerweiterung als Begriff benutzen.
- Graphisch können solche Klassenerweiterungen mit einem Pfeil von der Unterklasse zur Oberklasse dargestellt werden.
 - Um zu verdeutlichen, dass die Unterklasse über alle Eigenschaften der Oberklasse verfügt, können wir an der Kante das Label „IS-A“ hinzufügen.
 - Ein Polygon-Objekt ist ein (IS-A) EdgeSet-Objekt.
 - Standard „UML“ Notation: Pfeil mit nicht-gefülltem Dreieck



Polygon

Klassenerweiterung ist transitiv

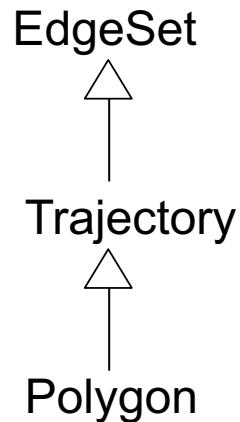
- Man kann eine Klasse erweitern, die bereits eine andere Klasse erweitert hat.

- public class Trajectory extends EdgeSet { ... }
- public class Polygon extends Trajectory { ... }
 - Damit bekommt Polygon alles, was in EdgeSet und Trajectory **nicht** mit dem Modifier static definiert wurde.

- Beispiel

- Falls EdgeSet die Methode length() besitzt, dann ist die Methode so auch in Polygon vorhanden.

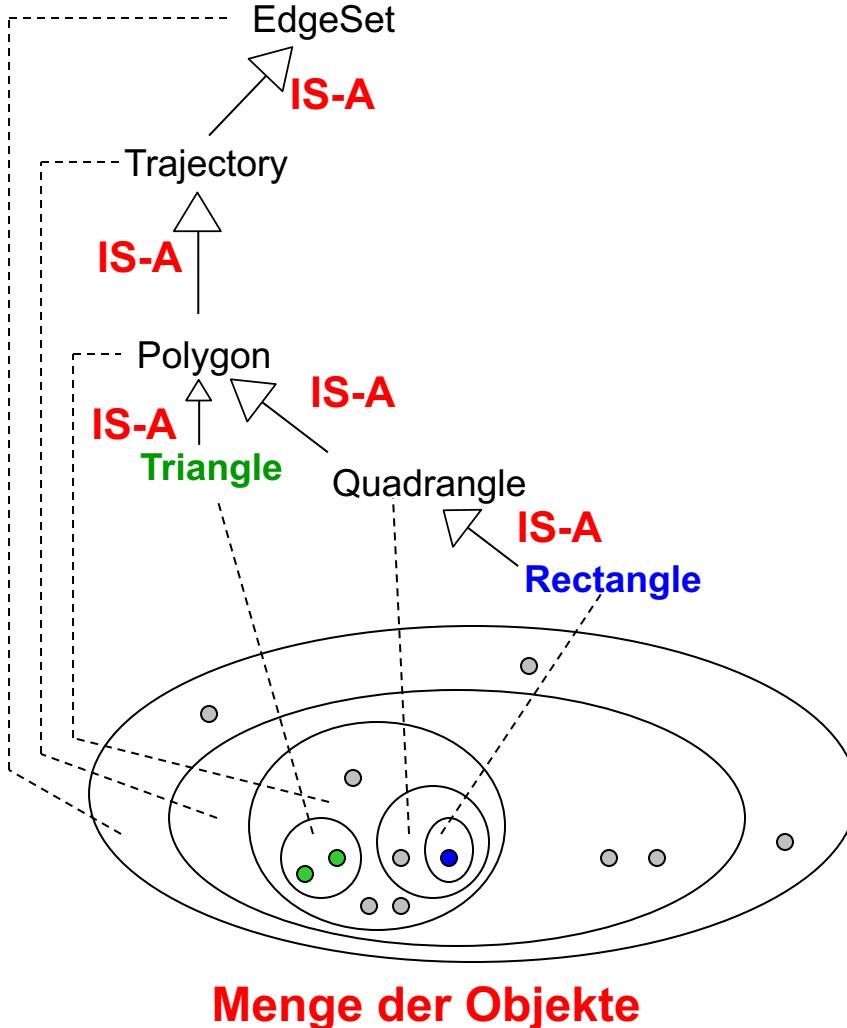
```
Polygon p = new Polygon(points);  
double res = p.length();
```



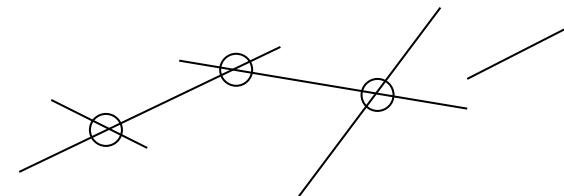
Oberklasse Object

- In Java gibt es eine **Klasse Object**, von der **alle Klassen direkt oder indirekt** erben.
 - Die Klasse Object hat keine Oberklasse.
 - Die Klasse Object besitzt z. B. eine Methode `toString()`, um ein Objekt als String zu repräsentieren.
 - Diese Methode wird bei der Ausgabe des Objekts von den Methoden `print` und `println` genutzt.
 - Klassen, die keine `extends`-Klausel besitzen, erweitern trotzdem die Klasse Object.
 - Wir werden später noch im Detail über diese Klasse sprechen.

Veranschaulichung von IS-A-Beziehungen



- Eigenschaften von EdgeSet
 - Anzahl der Kanten in der Menge
 - Methode length()
 - Methode intersection() zur Berechnung aller Schnittpunkte



Methode intersection()

- Objekte der anderen Klassen können zusätzlich noch weitere Eigenschaften besitzen z.B.
 - Trajectory: maxAngle
 - Polygon: area
 -

10.3 Überschreiben

- Die Klassenerweiterung in Java ermöglicht, dass Methoden aus Oberklassen auch in Unterklassen nochmals neu implementiert werden.
 - Ein Grund dafür ist, dass man in den Unterklassen häufig die spezifischen Eigenschaften der Objekte ausnutzen kann, um die Funktionalität schneller zu implementieren.
 - Ein zweiter Grund ist, dass bei einer Methode, die den Zustand des Objekts verändert, man noch die spezifischen Eigenschaften des Objekts der Unterkasse sicherstellen muss.

Beispiel (Schnittpunkte)

- Klasse EdgeSet



der Klasse EdgeSet müssen wir jede Kante mit allen anderen Kanten auf einen Schnitt testen.
Bei n Kanten ist die Anzahl der Vergleiche $(n+1)*n/2$.

- Klasse Polygon

- In der Klasse Polygon wissen wir bereits, dass nur die Endpunkte der Kanten sich berühren, aber sonst keine weiteren Schnitte vorliegen.
 - Es genügt also hier linear alle n Kanten zu besuchen und deren Endpunkte auszugeben.
 - Wir bieten deshalb eine Neuimplementierung der Methode in der Klasse Polygon an.

A cartoon illustration of a brown dog with a white patch on its chest, running towards the right. There are motion lines behind it.

```
public Point[] intersection() {
    Point[] points = new Point[edges.length];
    for (int i=0; i < points.length; i++)
        points[i] = edges[i].getStart();
    return points;
}
```

Beispiel (Einfügen von Kanten)

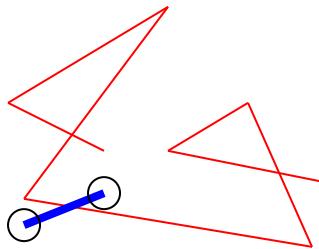
- In der Klasse EdgeSet soll eine Methode angeboten werden, um neue Kanten in die Menge einzufügen.
 - Unsere Lösung ist deshalb in dem Array etwas mehr Platz zu reservieren als notwendig und die Kanten im vorderen Teil des Arrays abzuspeichern.
 - Wir merken uns in einem Datenfeld `size` die aktuell belegten Plätze im Array.
 - Beim Einfügen einer neuen Kante wird dann einfach die nächst freie Position benutzt.

```
public class EdgeSet {  
    private Edge[] edges;  
    private int size;  
  
    ...  
  
    public void insert(Edge newEdge) {  
        if (size < edges.length)  
            edges[size++] = newEdge;  
        else  
            System.out.println("Storage Overflow");  
    }  
}
```

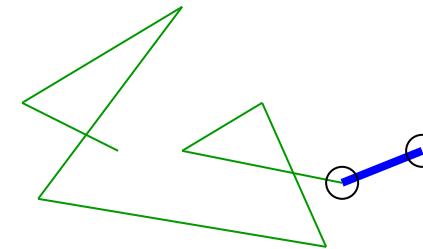
Kommentar: Irgendwann wird das Array zu klein sein. Wir lernen später Datenstrukturen kennen, die wachsen können.

Beispiel (Einfügen von Kanten)

- In der Klasse Trajectory können wir leider die bisherige Methode aus der Klasse EdgeSet nicht benutzen.
 - Es muss sichergestellt werden, dass ein Objekt der Klasse Trajectory aus einer zusammenhängenden Folge von Kanten besteht.
- Diese Eigenschaft kann sichergestellt werden, wenn wir das Einfügen einer neuen Kante nur am Ende oder Anfang des Kantenzugs erlauben.
 - Die neue Kante newEdge muss also einen gemeinsamen Eckpunkt mit der ersten Kante oder der letzten Kante im Kantenzug haben.



Nicht erlaubt!



Erlaubt

- Dadurch wird garantiert, dass das Objekt nach dem Einfügen weiterhin die Eigenschaft eines Kantenzugs besitzt.

Definition (Überschreiben)

- Beim Überschreiben einer Methode m_O aus einer Oberklasse O durch eine Methode m_U in einer Unterklassse U gelten folgende Eigenschaften:
 - Die Rückgabetypen der Methoden m_U und m_O müssen gleich sein.
 - Die Signaturen der Methoden m_U und m_O müssen übereinstimmen.
- Bei einer Änderungsoperation wird sichergestellt, dass die Eigenschaften des Objekts einer Klasse stets erhalten bleiben.
 - Diese Eigenschaften werden auch als **Klasseninvarianten** bezeichnet.
 - Alle Objekte der Klasse erfüllen die Klasseninvarianten.
 - Beispiel einer Klasseninvariante
 - Objekte der Klasse Trajectory repräsentieren eine Folge von Kanten, die sich in ihren Endpunkten berühren.

Definition (Überschreiben)

- Beim Überschreiben einer Methode m_O aus einer Oberklasse O durch eine Methode m_U in einer Unterklasse U gelten folgende Eigenschaften:
 - Die Rückgabetypen der Methoden m_U und m_O müssen gleich sein.
 - Die Signaturen der Methoden müssen übereinstimmen. Java stellt das nicht automatisch sicher, bietet aber Unterstützung, die wir später kennenlernen werden.
- Bei einer Änderungsoperation wird sichergestellt, dass die Eigenschaften des Objekts einer Klasse stets erhalten bleiben.
 - Diese Eigenschaften werden auch als **Klasseninvarianten** bezeichnet.
 - Alle Objekte der Klasse erfüllen die Klasseninvarianten.
 - Beispiel einer Klasseninvariante
 - Objekte der Klasse Trajectory repräsentieren eine Folge von Kanten, die sich in ihren Endpunkten berühren.

Verwendung überschriebener Methoden

- Für die Objekte der Unterklasse steht **nicht** mehr die ursprüngliche **Implementierung** der Methode zur Verfügung.
- Beispiel

```
Edge[] edges = new Edge[10];
```

```
...
```

```
Trajectory trajectory = new Trajectory(edges);
```

```
Point p = new Point(1.0, 2.0), q = new Point(2.0, 3.0);
```

```
// Hier wird immer die Methode aus der Klasse Trajectory verwendet.  
trajectory.insert(new Edge(p,q));
```

Verbieten von Überschreiben

- Soll das Überschreiben einer Methode m_O aus einer Oberklasse O in einer Unterklassse U verhindert werden, so muss bei der Definition von m_O das Schlüsselwort **final** verwendet werden.
 - Beispiel
 - Die Methode `length()` aus der Klasse `EdgeSet` soll in den Unterklassen nicht mehr überschrieben werden.

```
public class EdgeSet{  
    private Edge[] edges;  
  
    ...  
    public final double length() {  
        ...  
    }  
}
```

- In einer Unterklassse kann die `final`-Eigenschaft der Methode nicht mehr verändert werden.

Einmal final immer final.

Unterschiede zum Überladen

- In Java kann es in einer Klasse mehrere **Methoden mit dem gleichen Namen** geben.
 - Man spricht dann vom **Überladen der Methode**.
- Methoden mit gleichem Namen **müssen sich aber in ihrer Signatur unterscheiden**.
 - Die Signatur besteht aus dem Namen der Methode und dem Namen der Typen der Parametervariablen.
 - Der Rückgabetyp hat dabei keine Relevanz!
- Beispiel (Einfügen neuer Kanten)
 - Zusätzlich zu der Methode zum Einfügen einer einzelnen Kante, soll noch ein Einfügen eines Array von Kanten unterstützt werden.
 - Wir überladen deshalb die Methode insert und bieten die Methode nochmal mit dem Parametertyp Edge[] an.

Beispiel (Überladen)

- Hinzufügen einer zweiten Methode `insert` zu EdgeSet
 - Zusätzlich zu der Methode zum Einfügen einer einzelnen Kante, soll noch ein Einfügen eines Array von Kanten unterstützt werden.
 - Die Methode `insert` wird überladen.

```
public class EdgeSet {  
    private Edge[] edges;  
    private int size;  
  
    ...  
  
    public void insert(Edge newEdge) {  
        if (size < edges.length)  
            edges[size++] = newEdge;  
        else  
            System.out.println("Storage Overflow");  
    }  
  
    public void insert(Edge[] newEdges) {  
        if (size + newEdges.length < edges.length)  
            for (int i = 0; i < neue.length; i++)  
                edges[size++] = newEdges[i];  
        else  
            System.out.println("Storage Overflow");  
    }  
}
```

Neudeklaration von Datenfeldern

- Ähnlich wie bei Methoden können auch **Datenfelder** hinzugefügt werden.
- Es gibt aber einen **fundamentalen Unterschied** zum Überschreiben von Methoden:
 - Es wird dadurch nur **ein weiteres Datenfeld gleichen Namens** angelegt!
 - Das **ursprüngliche Datenfeld existiert weiterhin!**

Für Datenfelder gilt:

- Objekte der erweiterten Klasse UK können **sowohl auf das neu deklarierte Datenfeld** als auch auf das **ursprüngliche Datenfeld** zugreifen.
- Sei o eine Referenzvariable der Klasse UK und df ein Datenfeld der Oberklasse OK, das in der Klasse UK neu deklariert wurde. Dann gilt:
 - Zugriff auf das neu deklarierte Datenfeld: **o.df**
 - Zugriff auf das ursprüngliche Datenfeld: **((OK) o).df**
- **Prinzipiell sollte das Definieren von Datenfeldern mit gleichem Namen wie Datenfelder in der Oberklasse vermieden werden.**

Beispiel: Neudeklaration von Datenfelder

```
class OK {  
    public float a = 14.0f;  
  
    OK() {}  
}
```

```
class UK extends OK {  
    public int a = 27;  
  
    UK() {}  
}
```

```
public class UKTest {  
    public static void main(String[] args) {  
        UK k = new UK();  
        System.out.println(k.a);  
        System.out.println(((OK) k).a);  
    }  
}
```

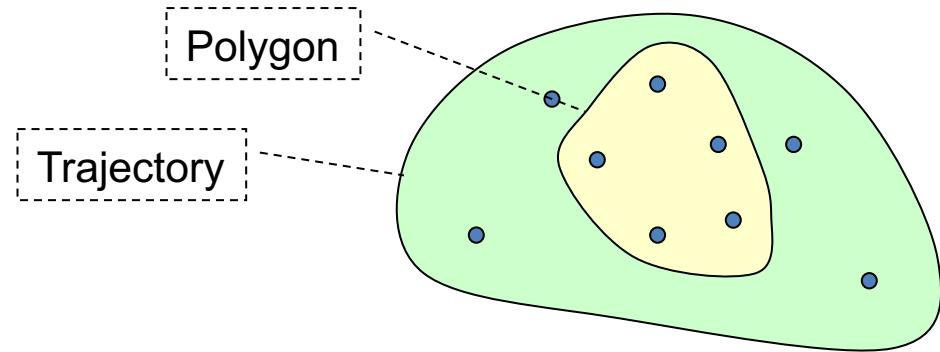
Ein Objekt der
Klasse UK hat beide
Datenfelder!

10.4 Polymorphie

- Bereits bekannt:
 - Klassen können als Datentypen von Variablen genutzt werden.
- Bei Ober- und Unterklassen können wir das bei Schnittstellen (Interfaces) bekannte Prinzip verwenden:
 - Eine **Referenzvariable** vom Typ der Oberklasse darf auf ein Objekt der Klasse K verweisen.

Beispiel

```
Polygon poly; // kann auf Polygon-Objekte verweisen  
Trajectory traj; // kann auf Polygon- und Kantenzug-Objekte verweisen
```



```
poly = new Polygon();  
traj = poly; // Polygon ist auch ein Kantenzug  
Object o = traj; // Kantenzug ist ein Objekt  
Trajectory[ ] geoArr = {new Polygon(), new Trajectory()};
```

Zentrale Regel der Polymorphie

1. Beim **Methodenaufruf** bestimmt stets die **Klasse des Objekts**, welche Implementierung benutzt wird.
2. Beim Zugriff auf ein **Datenfeld** bestimmt stets die **Klasse der Referenzvariablen**, welche Deklaration gültig ist.

Beispiel 1:

- Bei den folgenden Methodenaufrufen

```
Trajectory[ ] geoArr = {new Polygon(), new Trajectory()};  
Point[] points = geoArr[0].intersection();  
points = geoArr[1].intersection();
```

werden verschiedene Implementierungen von intersection()
angesprochen.

Zentrale Regel der Polymorphie

Beispiel 2:

```
class EdgeSet {  
    public final String name = "EdgeSet";  
    public String getName() {  
        return name;  
    }  
}
```

```
class Trajectory extends EdgeSet{  
    public final String name = "Trajectory";  
    public String getName() {  
        return name;  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        EdgeSet es1 = new EdgeSet();  
        Trajectory t = new Trajectory();  
        EdgeSet es2 = t;  
        System.out.println("Feldzugriff:");  
        System.out.println(es1.name);  
        System.out.println(t.name);  
        System.out.println(es2.name);  
        System.out.println("\nMethodenaufruf:");  
        System.out.println(es1.getName());  
        System.out.println(t.getName());  
        System.out.println(es2.getName());  
    }  
}
```

Welche Ausgabe ist korrekt:

- a) Feldzugriff:
 EdgeSet
 Trajectory
 EdgeSet

 Methodenaufruf:
 EdgeSet
 Trajectory
 EdgeSet
- b) Feldzugriff:
 EdgeSet
 Trajectory
 EdgeSet

 Methodenaufruf:
 EdgeSet
 Trajectory
 Trajectory
- c) Feldzugriff:
 EdgeSet
 EdgeSet
 Trajectory

 Methodenaufruf:
 EdgeSet
 Trajectory
 Trajectory

Live Vote

PIN: 1UQN

x

<https://ilias.uni-marburg.de/vote/1UQN>



Zentrale Regel der Polymorphie

Beispiel 2:

```
class EdgeSet {  
    public final String name = "EdgeSet";  
    public String getName() {  
        return name;  
    }  
}
```

```
class Trajectory extends EdgeSet{  
    public final String name = "Trajectory";  
    public String getName() {  
        return name;  
    }  
}
```

Welche Ausgabe
ist korrekt:

a)

Feldzugriff:
EdgeSet
Trajectory
EdgeSet

Methodenaufruf:
EdgeSet
Trajectory
EdgeSet

b)

Feldzugriff:
EdgeSet
Trajectory
EdgeSet

Methodenaufruf:
EdgeSet
Trajectory
Trajectory

c)

Feldzugriff:
EdgeSet
Edge Set
Trajectory

Methodenaufruf:
EdgeSet
Trajectory
Trajectory

Zentrale Regel der Polymorphie

Beispiel 2:

```

class EdgeSet {
    public final String name = "EdgeSet";
    public String getName() {
        return name;
    }
}

class Trajectory extends EdgeSet{
    public final String name = "Trajectory";
    public String getName() {
        return name;
    }
}

class Main {
    public static void main(String[] args) {
        EdgeSet es1 = new EdgeSet();
        Trajectory t = new Trajectory();
        EdgeSet es2 = t;
        System.out.println("Feldzugriff:");
        System.out.print(es1.name);
        System.out.print(t.name);
        System.out.print(es2.name);
        System.out.println("\nMethodenaufruf:");
        System.out.println(es1.getName());
        System.out.println(t.getName());
        System.out.println(es2.getName());
    }
}

```

Feldzugriff:
 EdgeSet
 Trajectory
 EdgeSet

Methodenaufruf:
 EdgeSet
 Trajectory
 Trajectory

Typkonvertierung von Objekten

- Wurde ein **Objekt einer Unterklasse** einer **Referenzvariablen** vom Typ der **Oberklasse** zugewiesen, so ist es über eine **explizite Konvertierung** erlaubt, dieses Objekt **wieder** an eine Referenzvariable der **Unterklasse** zu übergeben.
- **Beispiele:**



```
Trajectory traj1 = new Polygon();           // Iz1 verweist auf Objekt der Klasse Polygon
Polygon poly1 = (Polygon) traj1;           // Konvertierung ist somit erlaubt.
```



```
Trajectory traj2 = new Trajectory();
Polygon poly2= (Polygon) traj2;
// Dies wird zwar ohne Fehlermeldung übersetzt, führt aber bei der
// Programmausführung zu einem Fehler, der eine sogenannte Exception auslöst.
```

10.5 Konstruktoren

- Eigentlich haben wir bereits die wichtigsten Aspekte der Klassenerweiterung geklärt.
- Im Folgenden gehen wir noch auf die Erzeugung der Objekte einer Unterklasse ein.
 - Dies stellt sich als technisch relativ schwierig heraus.

• Wichtige Beobachtungen

- Konstruktoren der Oberklasse reichen alleine nicht für die Initialisierung eines Objekts der Unterklasse aus.
 - Neue Datenfelder in der Unterklasse können nicht initialisiert werden.
- Konstruktoren der Unterklasse reichen alleine nicht für die Initialisierung der Datenfelder der Oberklasse aus.
 - Privat-deklarierte Datenfelder der Oberklasse können nicht zugegriffen werden.

Konstruktoren in Java

- Zur Erinnerung
 - Klassen können eine **beliebige Anzahl von Konstruktoren** (mit unterschiedlichsten Parametern) besitzen.
 - Genau wie bei überladenen Methoden müssen diese Konstruktoren sich in Ihrer Signatur unterscheiden.
 - Wird zu einer Klasse **kein Konstruktor explizit definiert**, so wird **automatisch** der **parameterlose Konstruktor** erzeugt.
 - Wird ein Konstruktor aufgerufen, so wird **in der zugehörigen Klasse** nach einer entsprechenden **Implementierung** gesucht. Wenn **eine** passende Implementierung **nicht** vorhanden ist, gibt es einen **Übersetzungsfehler**.

Besonderheit bei der Klassenerweiterung in Java

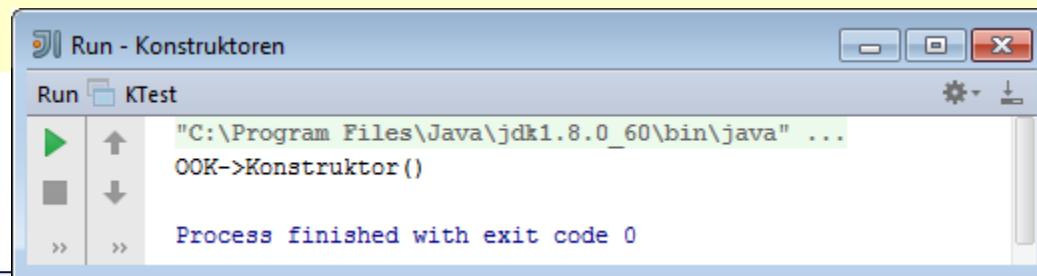
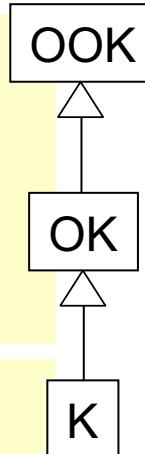
- Die Konstruktoren einer Klasse werden **nicht** vererbt.

Konstruktoren und Oberklassen

- Jeder Konstruktor einer Klasse **muss** (implizit oder explizit, direkt oder indirekt) einen Konstruktor der Oberklasse aufrufen!
 - *Expliziter und direkter Aufruf* erfolgt durch Angabe des Schlüsselworts **super** und den entsprechenden Parametern für den Konstruktor der Oberklasse.
 - Dieser Aufruf muss erster Befehl im Konstruktor der UnterkLASSE sein.
 - *Indirekter Aufruf* eines Konstruktors der Oberklasse erfolgt dann, wenn zunächst ein anderer Konstruktor der gleichen Klasse gerufen wird.
 - Dazu verwenden wir das Schlüsselwort **this** und die entsprechenden Parameter. Dieser Aufruf muss erster Befehl im Konstruktor sein.
 - Ein gegenseitiges (bzw. zyklisches) Aufrufen der Konstruktoren innerhalb einer Klasse ist verboten!
 - In allen anderen Fällen (erster Befehl ist nicht ein Konstruktorauftrag mit this und super) erfolgt ein **impliziter Aufruf** des **parameterlosen Konstruktors**, der dann in der Oberklasse verfügbar sein muss.

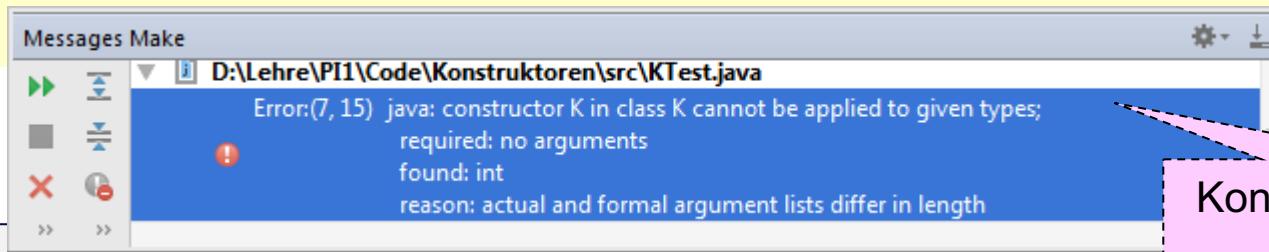
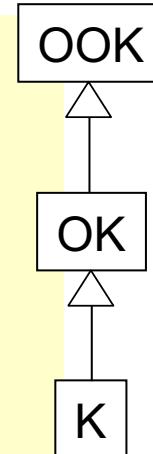
Beispiel 1 (impliziter Konstruktoraufruf)

```
class OOK{  
    OOK() {  
        System.out.println("OOK->Konstruktor()");  
    }  
    OOK(int i1) { System.out.println("OOK->Konstruktor(int)"); }  
}  
  
class OK extends OOK{  
}  
  
class K extends OK{  
}  
  
public class KTest {  
    public static void main(String[] args) {  
        K k = new K();  
    }  
}
```



Beispiel 2 (keine Vererbung bei Konstruktoren)

```
class OOK{  
    OOK() {  
        System.out.println("OOK->Konstruktor()");  
    }  
    OOK(int i1) {  
        System.out.println("OOK->Konstruktor(int)");  
    }  
}  
class OK extends OOK{  
}  
class K extends OK{  
}  
  
public class KTest {  
    public static void main(String[] args) {  
        K k = new K(5);  
    }  
}
```



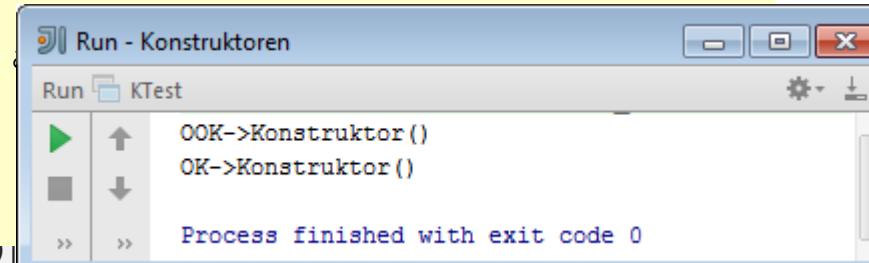
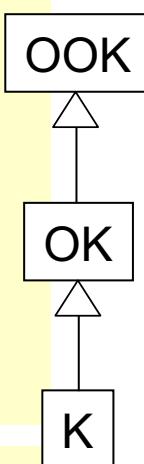
Konstruktoren werden eben
nicht vererbt!

Beispiel 3 (Reihenfolge der Konstruktoraufrufe)

```
class OOK{  
    OOK() {  
        System.out.println("OOK->Konstruktor()");  
    }  
    OOK(int i1) {  
        System.out.println("OOK->Konstruktor(int)");  
    }  
}
```

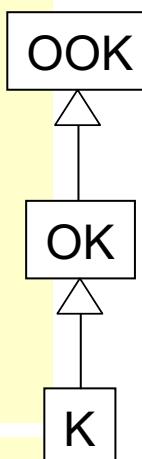
```
class OK extends OOK {  
    OK() {  
        System.out.println("OK->Konstruktor()");  
    }  
}  
  
class K extends OK {  
}
```

```
public class KTest {  
    public static void main(String[])  
        K k = new K();  
    }  
}
```



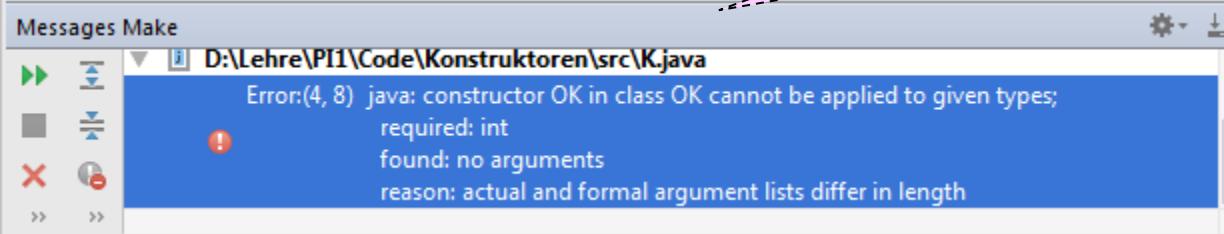
Beispiel 4 (Kein parameterloser Konstruktor)

```
class OOK{  
    OOK() {  
        System.out.println("OOK->Konstruktor()");  
    }  
    OOK(int i1) {  
        System.out.println("OOK->Konstruktor(int)");  
    }  
}  
  
class OK extends OOK {  
    OK(int x) {  
        System.out.println("OK->Konstruktor(int)");  
    }  
}  
  
class K extends OK {  
}
```



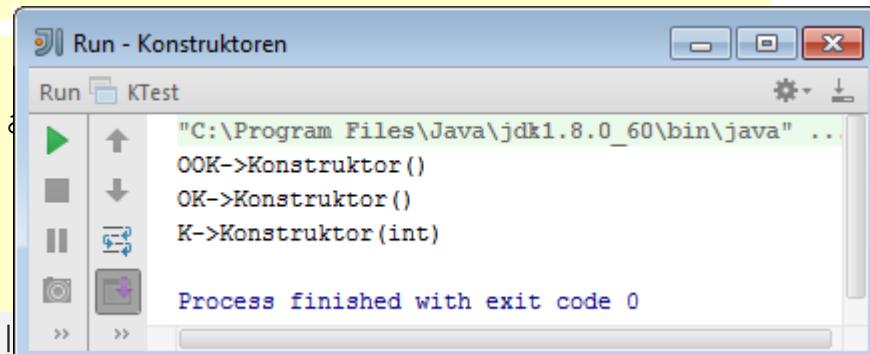
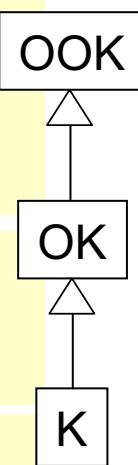
Die Klasse OK hat nicht mehr automatisch einen parameterlosen Konstruktor!

Dieser wird aber von dem automatisch erzeugten Konstruktor für Klasse K aufgerufen!



Beispiel 5 (Konstruktor mit Parameter)

```
class OOK{  
    OOK() { /* siehe oben */ }  
    OOK(int i1) { /* siehe oben */ }  
}  
  
class OK extends OOK {  
    OK() { System.out.println("OK->Konstruktor()"); }  
}  
  
class K extends OK {  
    K() {  
        System.out.println("K->Konstruktor()");  
    }  
    K(int x) {  
        System.out.println("K->Konstruktor(int)");  
    }  
}  
  
public class KTest {  
    public static void main(String[] args) {  
        K k = new K(5);  
    }  
}
```



Beispiel 6 (Aufruf eines Konstruktors mit this)

```

class OOK{
    /* siehe oben */
}

class OK extends OOK {
    OK() { /* siehe oben */ }
}

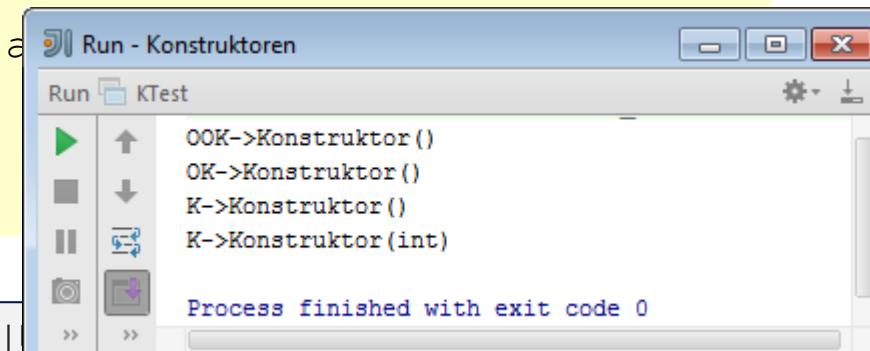
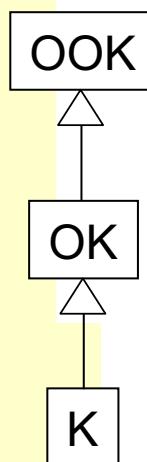
class K extends OK {
    K() {
        System.out.println("O-Konstruktor()");
    }

    K(int x) {
        this();
        System.out.println("K->Konstruktor(int)");
    }
}

public class KTest {
    public static void main(String[] args) {
        K k = new K(5);
    }
}

```

Weil `this()` aufgerufen wird, wird für **diesen** Konstruktor nicht mehr der parameterlose Konstruktor der Oberklasse aufgerufen!



Beispiel 7 (Aufruf eines Konstruktors mit super)

```

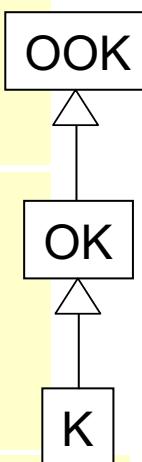
class OOK{
    /* siehe oben */
}

class OK extends OOK {
    OK(int x) {
        System.out.println("OK->Konstruktor(int)");
    }
}

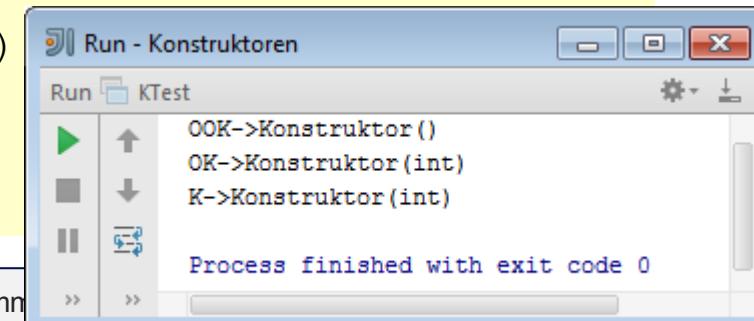
class K extends OK {
    K() { /* siehe oben */ }
    K(int x) {
        super(x);
        System.out.println("K->Konstruktor(int)");
    }
}

public class KTest {
    public static void main(String[] args)
        K k = new K(5);
    }
}

```

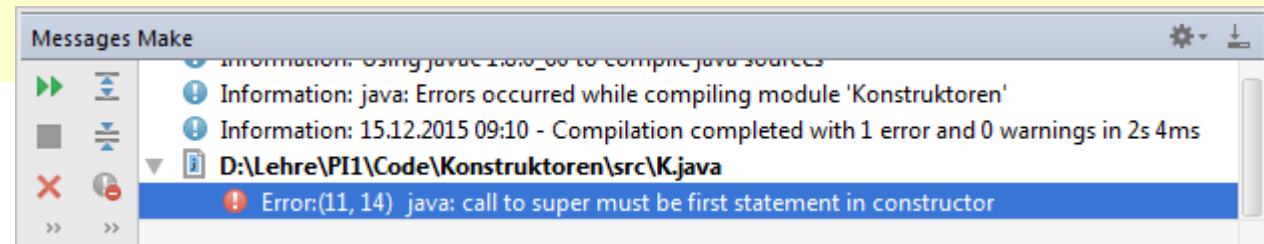


Jetzt wird mit super der Konstruktor der Oberklasse gerufen.



Beispiel 8 (Position des super-Aufrufs)

```
class OOK{  
/* siehe oben */  
}  
  
class OK extends OOK {  
    OK() /* siehe oben */  
    OK(int i) /* siehe oben */  
}  
  
class K extends OK {  
    K() {  
        System.out.println("K->Konstruktor()");  
    }  
    K(int x) {  
        System.out.println("K->Konstruktor(int)");  
        super(x);  
    }  
}
```



Beispiel für die Klassen Polygon und Trajectory

```
class Polygon extends Trajectory {  
    int count; // Abspeicherung der Anzahl der Kanten  
    /** Erzeugt ein Polygon mit edges.length Kanten  
     */  
    Polygon(Edge[] edges) {  
        super(edges);  
        count = edges.length();  
        If (edges[count-1].endPoint != edges[count[0].startPoint)  
            // Fehler: Programm abbrechen  
        // Prüfe Überschneidungsfreiheit der Kanten  
    }  
    double area(){...}      // Berechnung des Flächeninhalts  
    ...  
}
```

Initialisierung von Objekten

- **Algorithmus bei Aufruf eines Konstruktors der Klasse K**
 1. Reservierung des Speichers und Basisinitialisierung der Datenfelder der Klasse K:
 - 0 für Zahlen,
 - false für boolean-Datentyp,
 - null für Referenzvariablen.
 2. Aufruf der entsprechenden Konstruktoren der Oberklassen
 3. (Zweite) Initialisierung der Datenfelder durch ihre Initialisierungsausdrücke
 4. Ausführung des Rumpfs des Konstruktors

Aber eigentlich ist alles ganz einfach!

- Einfach den Konstruktor so schreiben dass er die Datenfelder des Objekts korrekt initialisiert.
 - Initialisierung der Datenfelder, die in der Klasse definiert wurden.
- Gegebenenfalls kann man **in der ersten Zeile** des Konstruktors mit „super“ und mit „this“ gezielt andere Konstruktoren aufrufen.
- Da der Rumpf des „lokalen“ Konstruktors als letztes ausgeführt wird, kann man hier den Datenfeldern neue Werte geben.

Die Bedeutung von `super`

Schlüsselwort `super` tritt in zwei verschiedenen Bedeutungen auf

- Aufruf von Konstruktoren der Oberklasse
- Zugriffsmöglichkeit auf die **Datenfelder** und **Methoden** der Oberklasse:
 - `super.<Datename>`
 - `super.<Methodename>(...)`

Beim Methodenaufruf wird hier die **Methode aus der Oberklasse** benutzt und nicht die aus der erweiterten Klasse.

Bemerkung

- In Java ist der Ausdruck `super.super.<Name>` **nicht** erlaubt.
 - Seien OOK, OK und K drei Klassen, wobei OOK direkte Oberklasse von OK und OK direkte Oberklasse von K ist.
 - Sei `m()` eine Methode von OOK, die in OK und K jeweils überschrieben wird.
 - Dann kann in der Klasse K (über `super`) **nur die Methode `m()` der Klasse OK, aber nicht die Methode `m()` der Klasse OOK** aufgerufen werden.

10.6 Zugriffsrechte

- In Java gibt es vier verschiedene Zugriffsrechte
 - public
 - private
 - protected
 - package private

Zugriffsrecht public

- **Zugriffsschutz public steht vor dem Datenfeld bzw. vor der Methode**
 - Datenfelder und Methoden sind **aus jeder anderen Klasse zugreifbar**.

Zugriffsrecht private

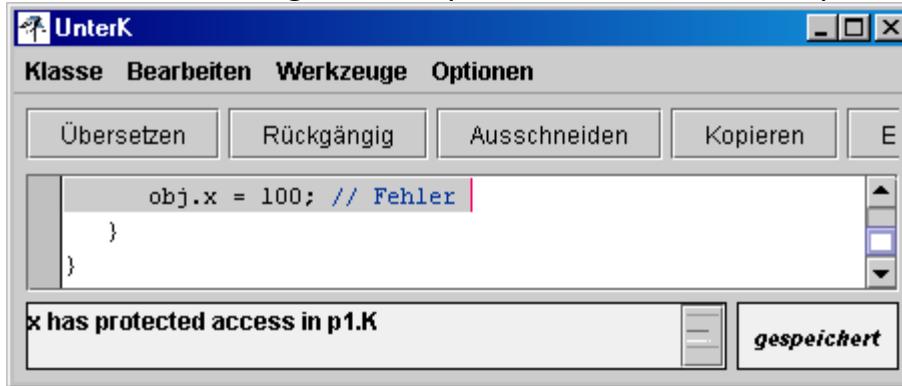
- **Zugriffsschutz private steht vor dem Datenfeld bzw. vor der Methode**
 - Datenfelder und Methoden sind **aus keiner anderen Klasse zugreifbar**.
- Methoden und Datenfelder sind **nur in der Klasse zugreifbar**, in der sie auch deklariert wurden.
 - Auch in Unterklassen kann nicht auf die Datenfelder und Methoden zugegriffen werden.

Diskussion

- Sei OK eine Klasse und **m()** eine Methode aus OK, die mit dem Attribut **public** gekennzeichnet ist und die auf **private** gekennzeichnete **Datenfelder** zugreift.
- Sei K eine direkte Unterklasse von OK.
 - Dann kann die Methode **m()** mit einem Objekt der Klasse K aufgerufen werden.
 - Privat deklarierte Datenfelder sind auch in den Objekten der Unterklasse vorhanden. Sie können jedoch nicht mehr direkt angesprochen werden.

Zugriffsrecht protected

- Zugriffsschutz **protected** steht vor dem Datenfeld bzw. vor der Methode
 - Datenfelder und Methoden sind über Objekte in den Klassen, die **zum gleichen Paket** gehören, zugreifbar.
 - Datenfelder und Methoden sind in **Unterklassen**, die in **anderen Paketen** definiert werden, zugreifbar (aber nur über "this")



```
package p1;
public class OK {
    protected int x;
    // ...
}
```

```
package p2;
import p1.OK;

class UK extends OK{
    void ok() {
        this.x = 100;
    }
    void nichtOk() {
        OK obj = new OK();
        obj.x = 100; // Fehler
    }
}
```



- im gleichen Paket ist der Zugriff aber erlaubt:

```
package p1;

public class OK {
    protected int x;
    // ...
}
```

```
package p1;

class UK extends OK{
    void ok() {
        this.x = 100;
    }

    void hierDochOk() {
        OK obj = new OK();
        obj.x = 100; // Korrekt!
    }
}
```



Warum wird überhaupt **protected** benötigt?

- Programme treten in mindestens zwei verschiedenen Rollen auf:
 - Programme werden durch einen Systementwickler erstellt und gewartet.
 - Programme werden durch Anwender benutzt.
- Entwickler sollten Zugriffsrechte für viele Datenfelder und Methoden eines Programms besitzen.
 - Java: Entwickler können die als „public“ und „protected“ gekennzeichnete Methoden und Datenfelder verwenden.
- Ein Anwender besitzt im allgemeinen weniger Zugriffsrechte als ein Entwickler.
 - Java: Anwender dürfen alle als „public“ gekennzeichnete Eigenschaften verwenden.

Zugriffsrecht package private

- **Keine explizite Angabe eines Zugriffsschutzes vor dem Datenfeld bzw. vor der Methode**
 - Datenfelder und Methoden sind über Objekte in den Klassen, die **zum gleichen Paket** gehören, zugreifbar.
- Vergleich mit Zugriffsschutz `protected`
 - Das Zugriffsrecht `package private` ist restriktiver als `protected`.
 - In Unterklassen, die in einem anderen Paket liegen, ist der Zugriff nicht mehr erlaubt.

Zugriffsrechte und Vererbung

- Beim Überschreiben von Methoden ist es erlaubt die **Zugriffsrechte zu verändern**. Dabei gilt:
 - Die Zugriffsrechte in einer überschriebenen Methode können **aufgeweicht werden**, aber nicht restiktiver werden.
 - Z. B.: Eine protected-Methode aus der Oberklasse kann zu einer public-Methode in der Unterklasse werden, aber nicht umgekehrt.
 - Hierzu wird die Methode überschrieben und der public-Modifier benutzt
 - Die überschreibende Methode kann z.B. einfach über „super“ die Implementierung aus der Oberklasse aufrufen und diese so zugreifbar machen.

10.7 Abstrakte Klassen

- Abstrakte Klassen sind spezielle Klassen, für die **keine direkten Objekte** erzeugt werden können (ähnlich wie bei Interfaces).
 - Abstrakte Klassen können **abstrakte Methoden** besitzen, die keinen Rumpf haben und in abgeleiteten Klassen noch überschrieben und implementiert werden müssen.
 - Unterschied zu Interfaces
 - Abstrakte Klassen können Datenfelder, Konstruktoren und nicht-abstrakte Methoden besitzen.
 - Eine UnterkLASSE kann nur **eine** abstrakte Klasse erweitern.
- Abstrakte Klassen fassen typischerweise **gemeinsame Eigenschaften der Unterklassen** zusammen, ohne dass Objekte erzeugt werden können!
 - In einer Universität gibt es bereits die drei Klassen Professoren, Mitarbeiter und Nicht-wissenschaftliche Angestellte.
 - Diese drei Klassen haben folgende Felder gemeinsam: PersonalId, Abteilung.
 - Durch Anlegen einer abstrakten Klasse Personal können diese Felder (und entsprechende Methoden) in der abstrakten Klasse zur Verfügung gestellt werden.

Syntax

- Eine abstrakte Klasse wird mit dem Schlüsselwort **abstract** gekennzeichnet, das direkt vor **class** (bzw. den Modifikatoren der Klasse) steht.
- Eine abstrakte Methode hat ebenfalls das Schlüsselwort **abstract** vorangestellt.
 - Eine abstrakte Methode besitzt in Java keinen Methodenrumpf.

```
public abstract class GeoObjectWithExtent {  
    private double x, y;  
  
    GeoObjectWithExtent(double a, double b) {  
        x = a; y = b;  
    }  
    public abstract Rectangle envelope();  
  
    public abstract double area();  
  
    public double coverage() {  
        return area() / envelope().area();  
    }  
}
```

Konstruktoren können bereits in abstrakten Klassen vorhanden sein.

Die Implementierung von Methoden kann bereits abstrakte Methoden der Klasse verwenden.

Implementierung einer abstrakten Klasse

- Die konkreten Unterklassen einer abstrakten Klasse müssen die abstrakten Methoden implementieren.

```
public class Circle extends GeoObjectWithExtent {  
    private double radius;  
  
    public Circle(double a, double b, double r) {  
        super(a,b); ——————  
        radius = r;  
    }  
  
    @Override  
    public Rectangle envelope() {  
        return new Rectangle (x, y, radius, radius);  
    }  
  
    @Override  
    public double area() {  
        return Math.PI*radius*radius;  
    }  
}
```

Aufruf des Konstruktors der Oberklasse.

Überschreiben der abstrakten Methode area.

Zusammenfassung

- Klassenerweiterung
 - UnterkLASSE erweitert eine OberKLASSE
 - Überschreiben von Methoden
 - Konstruktoren
 - Zugriffsrechte
 - Polymorphie
 - Variablen vom Typ der Oberklasse können auf Objekte der UnterkLASSE verweisen.
 - Dynamisches Binden
 - Klasse des Objekts bestimmt, welche Implementierung bei überschriebenen Methoden verwendet wird.
- Abstrakte Klassen
 - Keine Objekterzeugung

11. Ausnahmebehandlung

- Motivation
- Behandeln mit try-Anweisungen
- Werfen von Ausnahmen
- Weiterreichen mit throws

11.1 Motivation

- Unterscheidung zwischen zwei Arten von Fehlern
 - **Übersetzungsfehler**
 - Der Compiler erkennt fehlerhafte Anweisungen/Deklarationen im Programm und erzeugt deshalb auch keinen ausführbaren Code.
 - Der Programmentwickler wird über die entsprechenden Fehler informiert.
 - **Laufzeitfehler**
 - Die Ausführung eines Programms erzeugt unerwartete Ergebnisse bzw. bricht völlig ab.
 - Der Benutzer des Programms bekommt eine unverständliche Fehlermeldung und kann diesen Fehler nicht selbst beheben.

A screenshot of a Java IDE's terminal window titled "Run TestRealFunc". It shows the following output:

```
Exception in thread "main" 2.0 + 3.0*x^1 + 4.0*x^2 + 5.0*x^3
8.0 + 30.0*x^1
Wert des Polynoms p(42.0) = 377624.0
2.0e^(3.0*x)
6.0e^(3.0*x)
java.lang.ClassCastException: Exp cannot be cast to Polynom
        at TestRealFunc.main(TestRealFunc.java:21) <5 internal calls>
Process finished with exit code 1
```

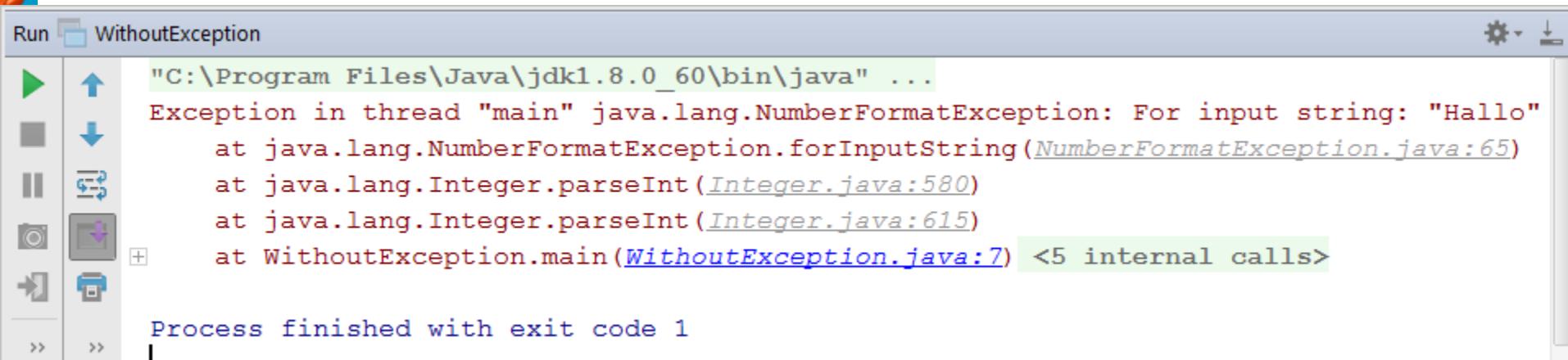


**Und solche Laufzeitfehler
führen zur Verzweiflung!**

Schon mal gesehen?

```
public class WithoutException {  
  
    public static void main(String[] args) {  
        int i = Integer.parseInt(args[0]);  
        System.out.println("Mein Zahl: " + i);  
    }  
}
```

- Zwei Ausgaben des Programms
 - Beim Start des Programms mit dem Argument 42.
Mein Zahl: 42
 - Beim Start des Programms mit dem Argument "Hallo Welt".



The screenshot shows an IDE interface with a toolbar on the left containing icons for run, stop, and other operations. The title bar says "Run WithoutException". The main area displays a stack trace:

```
"C:\Program Files\Java\jdk1.8.0_60\bin\java" ...  
Exception in thread "main" java.lang.NumberFormatException: For input string: "Hallo"  
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)  
    at java.lang.Integer.parseInt(Integer.java:580)  
    at java.lang.Integer.parseInt(Integer.java:615)  
    at WithoutException.main(WithoutException.java:7) <5 internal calls>  
  
Process finished with exit code 1
```

Was sieht man in der Ausgabe?

- Bei einem Laufzeitfehler im Java-Programm, der nicht behandelt wird, wird der sogenannte **Stack-Trace** ausgegeben.
- Der Stack-Trace ist die Liste der **Methodenaufrufe**, die beim Zeitpunkt des Programmabbruchs aktiv waren, mit der **Zeilenummer des letzten Befehls** in der Methode.
 - In unserem Fall bestand die Methode aus folgenden Aufrufen.
 - main-Methode unserer Klasse WithoutException (Zeile 7)
 - parseInt-Methode der Klasse Integer (Zeile 615)
 - parseInt-Methode der Klasse Integer (Zeile 580)
 - forInputString-Methode der Klasse NumberFormatException (Zeile 65)
 - Diese Methode hat zum Abbruch des Programms geführt.

Typische Ursachen für Laufzeitfehler

- Arrayzugriff mit einem Index außerhalb des erlaubten Bereiches
 - Zugriff auf nicht richtig initialisierte Variablen
 - Zugriff auf ein Objekt wird versucht, aber Referenz ist „null“
 - Konvertieren von Daten mit verschiedenen Repräsentationen
 - Cast-Operation eines Objekts in eine nicht-erlaubte Klasse
 - Schreibversuch in eine schreibgeschützte Datei oder Zugriff auf eine nichtvorhandene Datei
 - Missachtung von Vor- und Nachbedingungen
 - Inkorrekte Spezifikation von Methodenparametern
- Tritt während der Programmausführung ein Laufzeitfehler auf, so gerät das Programm in einen **Ausnahmezustand**.

Ausnahmen und Fehler

- Fehler (Error)
 - ist ein nicht reparierbarer Laufzeitfehler oder ein Hardware-Problem, das zum Absturz des Programms führt.
- Ausnahme (Exception)
 - ist meistens kein eigentlicher Fehler, sondern ein unerwarteter Fall, auf den das Programm reagieren sollte.

Erster Lösungsansatz

- In Programmiersprachen ohne explizite Unterstützung für die Ausnahmebehandlung wird typischerweise ein Fehlercode bei Methoden zurückgegeben.
- In dem folgenden Programmfragment ist das der Wert **-1**.

```
/** Die Methode durchsucht ein Array nach einem Element.  
 * @param arr Array mit ganzen Zahlen  
 * @param was Das zu suchende Element in dem Array  
 * @return Index im Array, an dem das gesuchte Element liegt.  
 */  
int search(int[] arr, int was){  
    int n = arr.length;  
    for (int i = 0; i < n; i++)  
        if (arr[i] == was) return i;  
    return -1;  
}
```

Nachteile von Fehlercodes (1)

- Vermischung von Programmlogik und Ausnahmebehandlung führt zu unleserlichen Programmen mit vielen bedingten Anweisungen
- Hier ein Beispiel als Pseudocode, wie so etwas aussehen könnte:

```
int readFile {
    int errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        compute length of file;
        if (gotTheFileLength) {
            allocate memory;
            if (gotEnoughMemory) {
                read file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        }
    }
}
```

```
    else {
        errorCode = -3;
    }
    close the file;
    if (fileDidntClose && errorCode == 0) {
        errorCode = -4;
    } else {
        errorCode = errorCode && -4;
    }
} else {
    errorCode = -5;
}
return errorCode;
}
```

Nachteile von Fehlercodes (2)

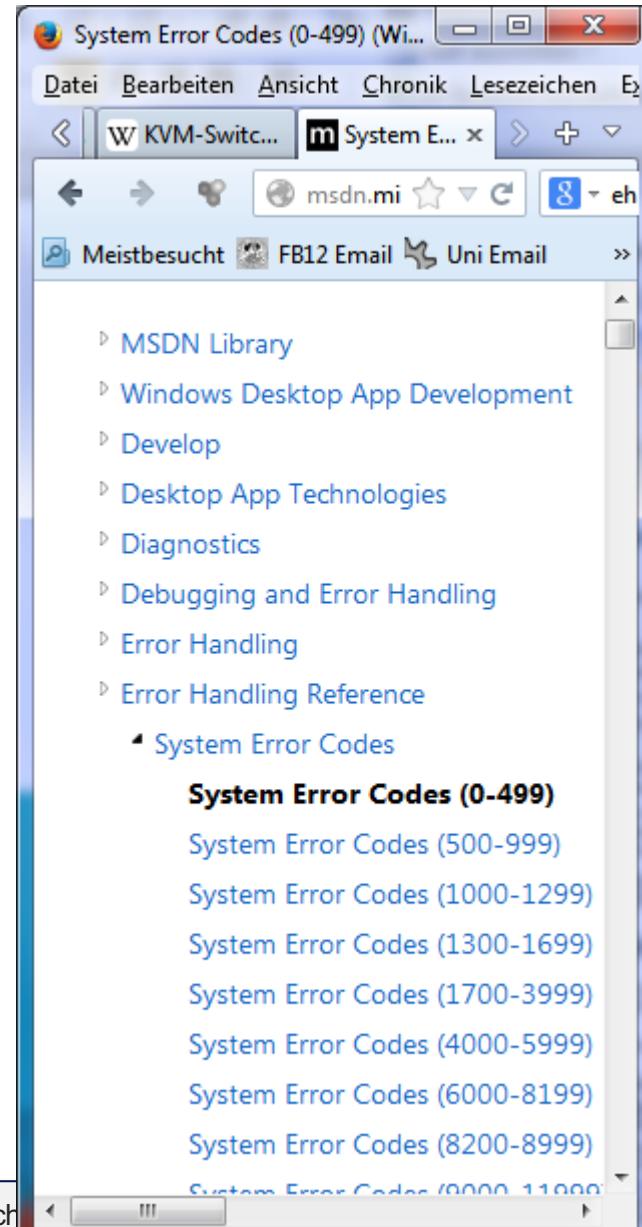
- Aufwändiges Durchreichen von Ausnahmen
- Hier ein Beispiel als Pseudocode, wie so etwas aussehen könnte:

```
void method1 () {  
    ErrorCodeType error = method2 ();  
    if (error < 0)  
        doErrorProcessing;  
    else  
        proceed1(); // Pseudocode  
}  
  
int method2() {  
    int error = method3();  
    if (error < 0)  
        return error;  
    else  
        proceed2(); // Pseudocode  
}
```

```
int method3() {  
    int error;  
    error = readFile( ....) ;  
    if (error < 0)  
        return error;  
    else  
        proceed3(); // Pseudocode  
}
```

Nachteile von Fehlercodes (3)

- Fehlercodes bieten **keinerlei Strukturierung** an.
 - Keine Gruppierung und Klassifizierung von Codes

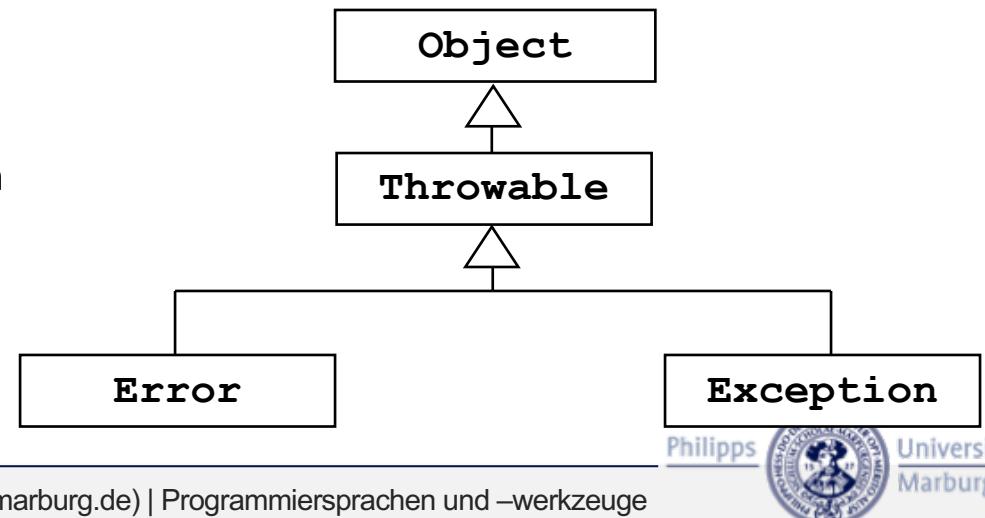


Ziele

- Bereitstellung einer Methodik, um Ausnahmen eleganter zu behandeln als mit Fehlercodes
 - z. B.: Ausgabe eines aussagekräftigen Texts zum Fehler
- Ausnahmen sollten nicht zum Abbruch des Programms führen.
 - Effektive Behandlung von Ausnahmen im Programm
- Trennung vom normalen Programmcode und dem Code zur Ausnahmebehandlung.
 - Vermeidung stark verschachtelter if-else-Anweisungen

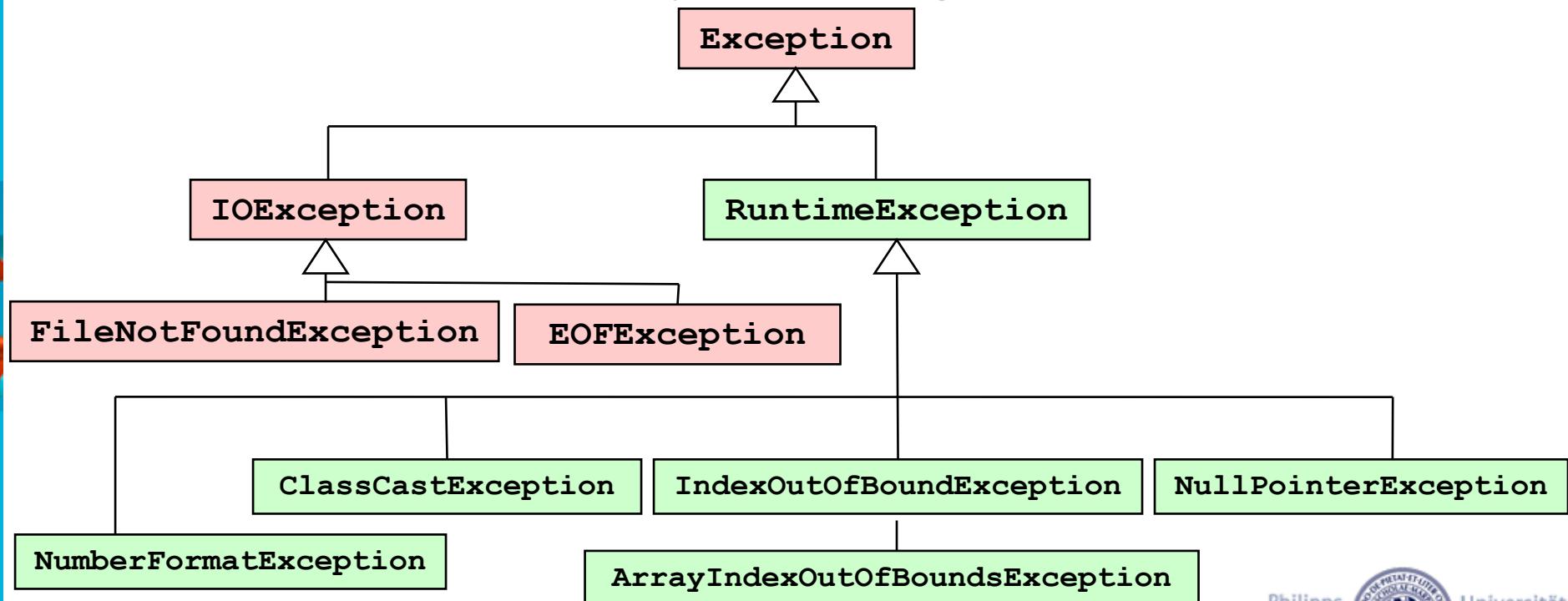
11.2 Ausnahmebehandlung in Java

- Java stellt für die Erkennung von (erwarteten) Fehlern, eine sogenannte **Ausnahmebehandlung** zur Verfügung.
 - Mittels den Oberklassen **Throwable**, **Error** und **Exception** und davon abgeleiteten Unterklassen werden Ausnahmeobjekte zur Verfügung gestellt.
 - Ausnahmen können **erzeugt** („geworfen“) werden: „**to throw an exception**“.
 - Falls eine Ausnahme auftritt, kann dieses Ereignis **abgefangen** werden: „**to catch an exception**“.
- Die Klasse **Throwable** ist die Oberklasse aller für Fehler und Ausnahmen zuständigen Klassen.
 - Geworfen** werden können nur Objekte dieser Klasse bzw. ihrer Unterklassen.
 - Analog: Nur **Throwable**-Objekte können **abgefangen** und **behandelt** werden.



Die Klassenhierarchie Exception

- ▶ In dieser Übersicht sind einige aber nicht alle Unterklassen von **Exception** aufgeführt.
- ▶ Die Klasse RunTimeException und deren Unterklassen haben einige Besonderheiten, auf die wir später noch eingehen werden.



Ein erstes Beispiel

```
public class WithException {  
  
    public static void main(String[] args) {  
        int i;  
        try {  
            i = Integer.parseInt(args[0]);  
        }  
        catch (NumberFormatException except) {  
            i = 7;  
            System.out.println("Defaultzahl = " + i);  
        }  
        System.out.println("Meine Zahl: " + i);  
    }  
}
```

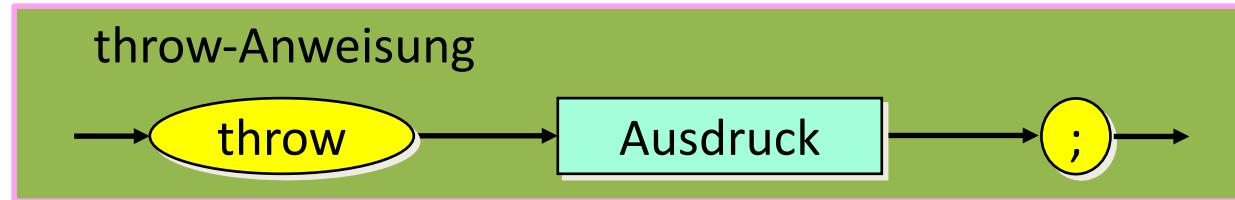
- Verwendung der try-catch-Anweisung
 - try-Block besteht aus dem eigentlichen Programm
 - catch-Block besteht aus der Fehlerbehandlung.
 - Parameter des catch-Blocks ist ein Objekt einer Exception-Klasse

Übersicht zu Exceptions in Java

- Eine Ausnahme kann **geworfen** werden.
 - Durch eine throw-Anweisung wird ein Objekt einer Exception-Klasse erzeugt.
- Eine Ausnahme kann **gefangen** werden.
 - Hierfür gibt es den try-catch-Block
- Eine Ausnahme kann **weitergereicht** werden.
 - Für sogenannte **ungeprüfte Ausnahmen** (Objekte der Klasse RuntimeException oder einer Unterklasse) ist dies ohne weitere Angaben möglich.
 - Für alle anderen Exceptions (**geprüfte Ausnahmen**) wird noch das throws-Schlüsselwort im Methodenkopf benötigt.

11.2.1 Werfen von Ausnahmen

- Das Werfen einer Ausnahme erfolgt durch die **throw-Anweisung**.



- In dem Ausdruck muss (in der Regel) ein **Konstruktor** der **Exception-Klasse** oder einer Klasse, welche die Klasse Exception erweitert, aufgerufen werden.
- Bei der Ausführung der Anweisung wird ein Objekt der entsprechenden Exception-Klasse erzeugt und der übliche **Ablauf der Methode gestoppt**.
 - Die Fortführung der Programmausführung hängt davon ab, ob die Ausnahme
 - gefangen**
 - oder weitergereicht** wird.

Beispiel

- Beim Erzeugen eines Objekts der Klasse Edge können zwei Runtime-Exceptions geworfen werden:
 - Falls einer der Punkte den Wert null hat.
 - NullPointerException
 - Falls die Punkte gleich sind.
 - EqualPointException ist eine eigene RuntimeException-Klasse.

```
public class Edge{  
    ...  
    Edge(Point2D p, Point2D q) {  
        if (p == null || q == null)  
            throw new NullPointerException("Eingabe: null");  
        if (p.equals(q))  
            throw new EqualPointException(q);  
  
        start = p;  
        end = q;  
    }  
    ...  
}
```

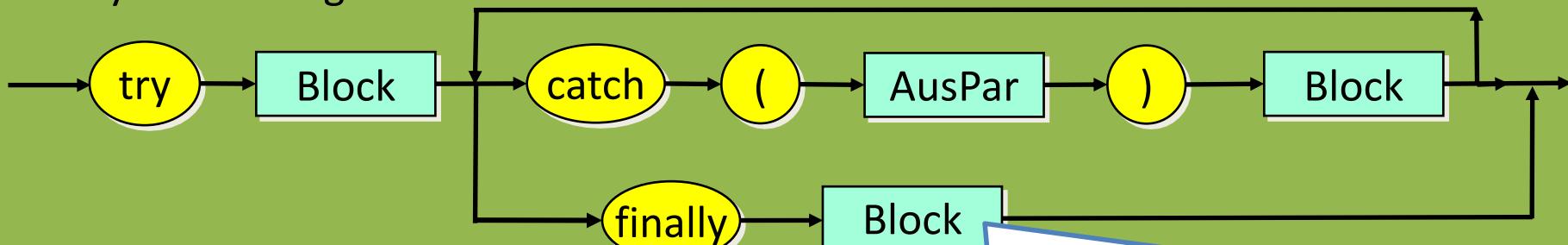
Wie eigene Exception-Klassen definiert werden, sehen wir später.

Falls p oder q null sind, wurde eine Exception geworfen. Der weitere Code wird nicht mehr ausgeführt. Daher ist es OK auf p eine Methode aufzurufen.

11.2.2 Fangen von Ausnahmen

- Eine Ausnahme kann durch eine **try-Anweisung** gefangen werden.
 - Dabei steht das eigentliche Programm in dem **try-Block**.
 - In dem Block können auch Exceptions geworfen werden.
 - Das Fangen der Ausnahmen erfolgt in den **catch-Blöcken**.
 - Im Kopf vor dem catch-Block wird eine Variable vom Typ einer Ausnahmeklasse deklariert.
 - Optional können in einem **finally-Block** noch die letzten Schritte der try-Anweisung (nach einem catch-Block oder try-Block) angegeben werden.

try-Anweisung:



AusPar:



Ablauf einer try-Anweisung

- Der Rumpf der try-Anweisung wird entweder komplett oder bis zum Werfen der ersten Ausnahme ausgeführt.
- Beim Werfen einer Ausnahme wird ein Exception-Objekt erzeugt. Dann werden die catch-Böcke von oben nach unten daraufhin überprüft, ob sie die Ausnahme behandeln können.
 - *Eine Behandlung ist dann möglich, wenn das geworfene Exception-Objekt der Variable im Kopf des catch-Blocks zugewiesen werden kann.*
 - *Wenn so der erste catch-Block gefunden wurde, wird das Exception-Objekt der Variablen zugewiesen und dann die Anweisungen im eigentlichen Block ausgeführt.*
 - *Falls kein passendes catch-Konstrukt gefunden wurde, wird die Ausnahme an die rufende Methode weitergereicht. (→ Details später)*
- Falls der finally-Block vorhanden ist, wird dieser **immer** am Schluss ausgeführt.



Beispiel (mehrere catch-Blöcke)

- Wir behandeln in unserem bisherigen Programm noch das Problem, wenn ein Benutzer kein Argument liefert.

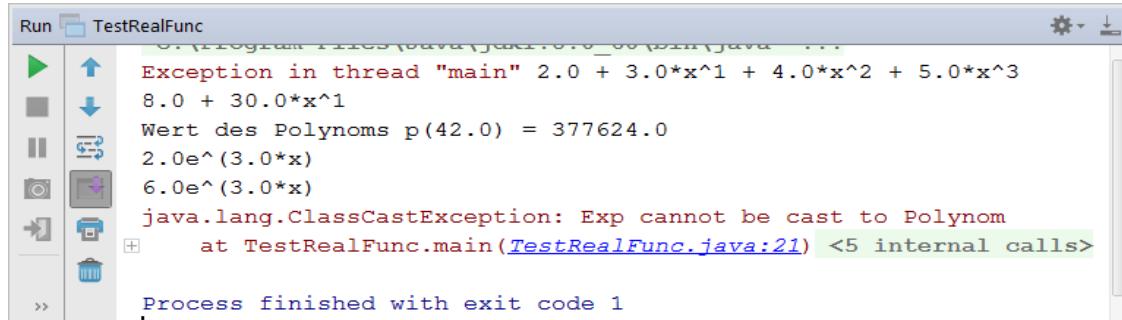
```
public class WithException {  
  
    public static void main(String[] args) {  
        int i;  
        try {  
            i = Integer.parseInt(args[0]);  
        }  
        catch (NumberFormatException abc) {  
            i = 7;  
            System.out.println("Zahl (bei falscher Eingabe) " + i);  
        }  
        catch (ArrayIndexOutOfBoundsException except) {  
            i = 0;  
            System.out.println("Zahl (bei keiner Eingabe) " + i);  
        }  
        System.out.println("Meine Zahl: " + i);  
    }  
}
```

Reihenfolge der catch-Blöcke

- Die Reihenfolge der catch-Blöcke ist wichtig, da die Fehlerbehandlung beim ersten Block stattfindet, bei dem die geworfene Ausnahme passt.
 - Ausnahmeobjekt kann der Parametervariablen im catch-Block zugewiesen werden.
- In der Liste der catch-Blöcke sollte ein Block mit einer speziellen Exception-Klasse vor dem einer allgemeinen Klasse stehen.
 - Weshalb?

11.2.3 Weiterreichen von Ausnahmen

- Werden ausgelöste Ausnahmen innerhalb einer Methode nicht behandelt, so werden diese an die aufrufende Methode weitergereicht.
 - In der **main-Methode** führt eine nicht-behandelte Ausnahme zum **Abbruch des Programms und Ausgabe des Stack-Trace**.



The screenshot shows a Java IDE's run window. The title bar says "Run TestRealFunc". The output pane displays the following stack trace:

```
Exception in thread "main" 2.0 + 3.0*x^1 + 4.0*x^2 + 5.0*x^3
8.0 + 30.0*x^1
Wert des Polynoms p(42.0) = 377624.0
2.0e^(3.0*x)
6.0e^(3.0*x)
java.lang.ClassCastException: Exp cannot be cast to Polynom
    at TestRealFunc.main(TestRealFunc.java:21) <5 internal calls>
Process finished with exit code 1
```

- Beim Weiterreichen von Ausnahmen ist die Unterscheidung zwischen zwei Arten von Ausnahmen wichtig.
 - **Ungeprüfte** Ausnahmen
 - **Geprüfte** Ausnahmen

Geprüfte und ungeprüfte Ausnahmen

- Java unterscheidet zwei Arten von Ausnahmen
 - **Ungeprüfte Ausnahmen** sind Objekte der Klasse `RuntimeException` und ihrer Unterklassen.
 - Wird eine solche Ausnahme nicht in einer Methode behandelt, wird diese automatisch an die aufrufende Methode weitergeleitet.
 - Im Programm ist hierfür **kein zusätzlicher Code** erforderlich.
 - **Geprüfte Ausnahmen** sind Objekte der Klasse `Exception` und ihrer Unterklassen, die **nicht** eine RuntimeException sind.
 - z. B. Öffnen einer Datei, die nicht existiert.
 - In der Methode, in der diese Aufnahme geworfen wird, muss diese Ausnahme entweder behandelt oder **explizit** für die Weiterreichung **angemeldet werden**.
- Entwickler entscheiden, ob sie **neue Ausnahmen als geprüfte oder ungeprüfte Exceptions implementieren**.

Weiterreichen geprüfter Ausnahmen

- Werden **geprüfte Ausnahmen** innerhalb einer Methode nicht behandelt, so müssen diese weitergereicht werden.
 - Mit Hilfe der **throws-Klausel** muss **im Methodenkopf** festgelegt, welche geprüften Ausnahmen von einer Methode weitergereicht werden.
 - Beim **Überschreiben von Methoden** ist darauf zu achten, dass die Ausnahmen bei der überschriebenen Methode angegeben werden.
 - Eine Spezialisierung von Exceptions in der throws-Klausel ist erlaubt.
- Beispiel
 - Nehmen wir an, dass die Klasse EqualPointException eine geprüfte Ausnahme erzeugt (also keine Unterklasse von RuntimeException ist).

```
Edge(Point2D p, Point2D q) throws EqualPointException {  
    // Beste Loesung in diesem Fall!  
    if (p.equals(q))  
        throw new EqualPointException(q);  
    start = p;  
    end = q;  
}
```



11.2.4 Eigene Exception-Klasse

- Regel Nr. 1
 - Bevor eine eigene Exception-Klasse geschrieben wird, sollte man prüfen, ob eine der bereits in Java existierenden Klassen benutzt werden kann.
- Eine eigene Exception-Klasse muss von Exception bzw. von RuntimeException erben.
 - Klasse Exception
 - Konstruktoren
 - Exception()
 - Exception(String str)
 - Methoden (von der Klasse Throwable geerbt)
 - getMessage, printStackTrace, toString,

Beispiel

```
package geo;

public class EqualPointsException extends RuntimeException {
    private Point p;

    public EqualPointsException(Point in){
        super("Punkte sind gleich: " + in);
        p = in;
    }
}
```

Zusammenfassung

- Ausnahmen in Java
 - Nutzen vorhandener Exception-Klassen
 - Neuentwicklung eigener Exception-Klassen
- Unterscheidung
 - Geprüfte Ausnahmen
 - Ungeprüfte Ausnahmen
- Wichtige Funktionalität
 - Werfen von Ausnahmen mit throw
 - Fangen von Ausnahmen mit try-catch-finally
 - Weiterreichen von geprüften Ausnahmen mit throws-Liste

12. Generische Klassen

- Problem bei der Erstellung von Software sind die sehr **hohen Kosten**.
 - Kosten für die Erstellung von 1 Zeile Programmcode: **10 – 50 Euro**
 - Wartungskosten von existierendem Code: **50–75% der Gesamtkosten**
- Eine Lösungsansatz für die Kostenreduktion ist der Einsatz von **vorgefertigten Komponenten** zur Entwicklung von Anwendungen, um die Anzahl der noch benötigten Zeilen eigenen Programmcodes zu minimieren.
- Beispiele
 - Statt I/O selbst zu programmieren greift man auf Standardsoftware zurück wie z. B. Datenbanksysteme
 - Java bietet mit dem JDK eine Vielzahl vorgefertigter Klassen, die man für die Entwicklung eigener Anwendungsprogramme nutzen kann.
- Die in Java unterstützen generischen Klassen erlauben es in einfacher Weise wiederverwendbare Komponenten selbst zu entwickeln.

Aufbau des Kapitels

- Fallbeispiel: Listen mit 2-dimensionalen Punkten
 - Konzept einer Liste
- Generische Liste mit Object
 - Nachteile dieser Lösung
- Generische Listen als parametrisierte Klassen
- Generische Klassen aus der Java-Bibliothek
 - Schnittstellen Comparable, List und Iterable
- Details zu generischen Klassen
 - Schlüsselwort extends
 - Wildcards

12.1 Fallbeispiel: Listen

- Wichtige Aufgabenstellung in der Informatik
 - Dynamische Verwaltung einer Menge **gleichartiger Datenobjekte**, um diese wiederzufinden.
 - Gleichartig bedeutet, dass die Objekte zu einer Klasse gehören.
- Beispiele
 - Wir haben innerhalb einer betriebswirtschaftlichen Anwendung Klassen für Kunden, Produkte, Aufträge, Bestellungen, ... erstellt.
 - In jeder dieser Klassen werden Datenfelder definiert, die den Zustand der Objekte aus diesen Klassen beschreiben.
 - In einem Unternehmen sollen jetzt Mengen von Objekten verwaltet werden, die aus einer Klasse stammen. Z. B. eine Menge aller Kunden eines Unternehmens.
 - Es soll möglich sein, neue Kunden in die Menge einzufügen und Kunden aus der Menge zu löschen.
 - Zudem soll es möglich sein, Kunden in der Menge unter Angabe von Suchprädikaten zu finden.
 - Suchprädikate hängen dabei von den Datenfelder des Objekts ab.
 - Suche nach dem Kunden mit KundenNr = 1234
 - Suche nach Kunden mit Wohnort = „Marburg“

Mögliche Implementierungen

- Menge als Array
 - Ist die Maximalzahl möglicher Datenobjekte bekannt, so sollte am besten ein **Array** verwendet werden.
- Menge als einfach verkettete Liste
 - Ist die Anzahl der Datenobjekte nicht bekannt, kann man die Objekte dynamisch z.B. in **einfach verketteten Listen** verwalten.
 - Im Gegensatz zu einem Array kann eine einfache verkettete Liste dynamisch (also zur Laufzeit) verlängert bzw. verkürzt werden.
 - Durch das Einfügen neuer Elemente wird eine Liste **verlängert**.
 - Wenn Elemente gelöscht werden, wird die Liste verkürzt.

Was soll also eine Liste leisten?

- Die Minimalanforderung besteht aus folgenden Methoden

- Hinzufügen eines Elements an das Ende der Liste

boolean add(Element e)

- Löschen eines Elements aus der Liste

boolean remove(Element e)

- Suchen nach einem Element in der Liste

boolean contains(Element e)

- Prüfen, ob die Liste leer ist.

boolean isEmpty()

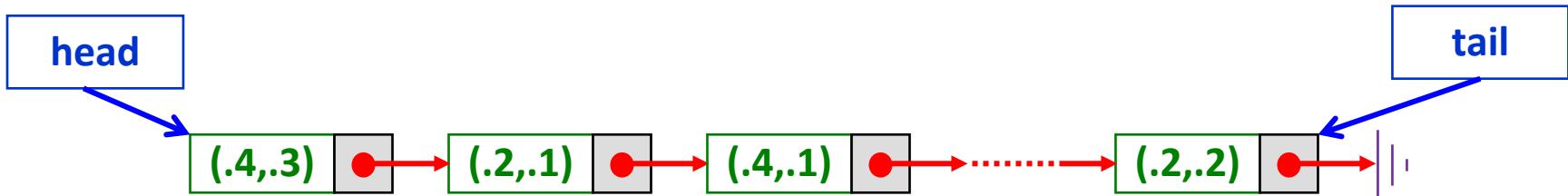
- Ausgabe des i-ten Elements aus der Liste

Element get(int i)

- Was ist unter dem Datentyp *Element* zu verstehen?
 - Wir wollen Objekte mit beliebigem Typ speichern können!
(dazu später mehr)

Implementierung einer einfach verketteten Liste

- Zunächst machen wir folgenden Annahme
 - Der Datentyp Element entspricht der Klasse *Point2D*, die wir bereits aus früheren Kapiteln kennen. Wir wollen also eine Liste von Punkten verwalten.
- Eine Liste von Punkten besteht aus miteinander verketteten Listenelementen. Jedes Listenelement hat
 - ein Objekt der Klasse *Point2D*
 - eine Referenz auf das nächste Listenelement



- Zusätzlich merken wir uns zwei Verweise **head** und **tail** auf das erste und letzte Listenelement.
 - Dadurch können schnell am Anfang **und am Ende** neue Punkte eingefügt werden.

Umsetzung

- Zur Implementierung von Listen werden zwei neue Klassen implementiert.
 - Eine Klasse `ListElement` für die **Listenelemente**.
 - Eine Klasse `LinkedList` zur Verwaltung der Liste. Diese enthält:
 - Datenfelder `head` und `tail`
 - *Konstruktoren*
 - Methoden von Folie 567
 - Weitere Methoden wie z. B. `equals` und `toString`
- Für die Nutzung der Liste wird noch die Klasse `Point2D` benötigt.
 - Diese Klasse liegt bereits vollständig implementiert vor.

Klasse ListElement

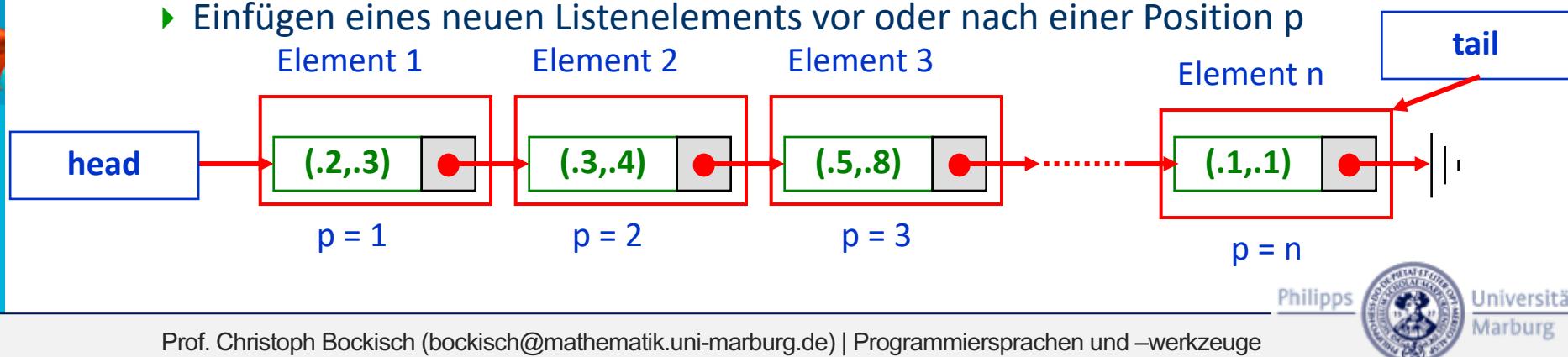
- ▶ Diese Klasse hat zwei Datenfelder:
 - ▶ Das Datenfeld *data* referenziert ein Objekt der Klasse Point2D.
 - ▶ Das Datenfeld *next* auf ein Objekt der Klasse ListElement
- ▶ Da innerhalb der Klasse wiederum auf Objekte der Klasse Bezug genommen wird, sprechen wir auch von einer rekursiven Klasse.
 - ▶ Viele der Algorithmen für Listen könnten deshalb auch rekursiv formuliert werden.
- ▶ Konstruktoren der Klasse
 - ▶ Einer der beiden Konstruktoren definiert ein neues Objekt vom Typ ListElement mit gegebenem *data* und *null* als *next*.
 - ▶ Erzeugung eines Listenelements ohne einen Nachfolger
 - ▶ Der andere Konstruktor definiert ein neues Objekt vom Typ ListElement mit gegebenem *data* und gegebenem Verweis *next* auf das nächste ListElement.
 - ▶ Erzeugung eines Listenelements mit Nachfolger

Klasse ListElement

```
public class ListElement{  
    protected Point2D data;  
    protected ListElement next; // rekursive Klasse  
  
    /**  
     * Konstruktor der Klasse mit einem Parameter vom Typ Point2D  
     * @param Objekt der Klasse Point2D, auf das Datenfeld data referenziert.  
     */  
    ListElement(Point2D in){  
        this(in, null);  
    }  
  
    /**  
     * Konstruktor der Klasse  
     * @param in Objekt der Klasse Punkt, das in dem ListElement referenziert  
     * werden soll.  
     * @param ref Ein gültiger Verweis auf ein ListElement.  
     */  
    ListElement(Point2D in, ListElement ref){  
        next = ref;  
        data = in;  
    }  
}
```

Klasse LinkedList

- ▶ Diese Klasse enthält:
 - ▶ Die Verweise `head` und `tail`.
 - ▶ Konstruktoren
 - ▶ Methoden
 - ▶ Einfügen (am Ende), Löschen, Suchen, Prüfen auf leer, Liefern des ersten Elements
 - ▶ `toString`, um eine Liste einfach auszugeben.
- ▶ Optional könnte man noch weitere Methoden implementieren
 - ▶ Ermitteln der Listenlänge
 - ▶ Einfügen eines neuen Listenelements vor oder nach einer Position p



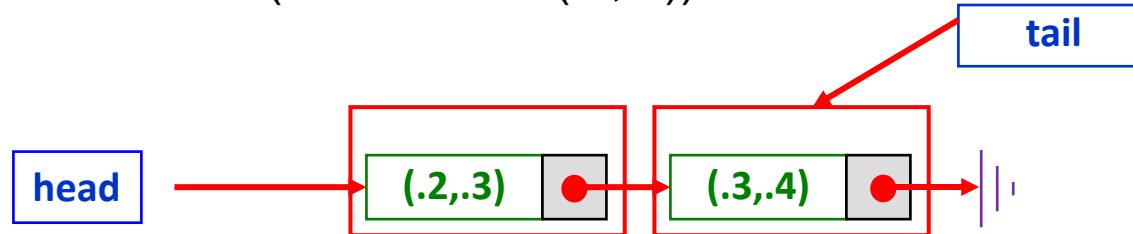
Einfügen eines neuen Punkts

```
public class LinkedList {  
  
    private ListElement head, tail;  
  
    LinkedList() {  
        head = tail = null;  
    }  
  
    /** Fügt einen neuen Punkt in die Liste ein.  
     * @param p der neue Punkt  
     * @return ob der Punkt erfolgreich eingefügt wurde.  
     */  
    public boolean add(Point2D p) {  
        if (p == null) return false;                      // Solche Punkte nicht!  
        if (head == null)                                // Liste ist leer  
            head = tail = new ListElement(p);  
        else {  
            tail.next = new ListElement(p);              // Liste ist nicht leer  
            tail = tail.next;                            // Einfügen am Ende  
        }  
        return true;  
    }  
}
```

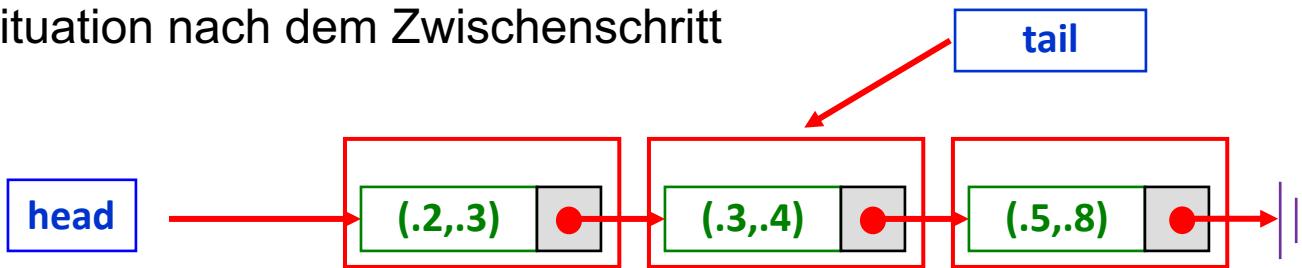
...

Beispiel

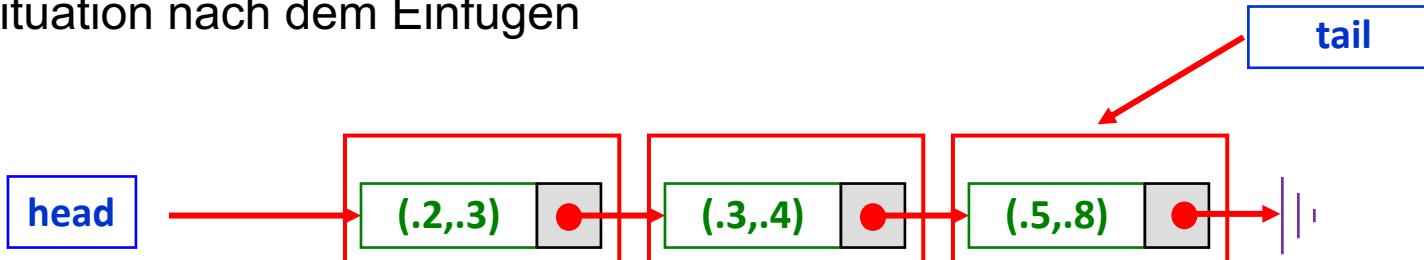
- Aufruf von `add(new Point2D(.5,.8))` auf einer Liste mit 2 Elementen.



- Situation nach dem Zwischenschritt



- Situation nach dem Einfügen



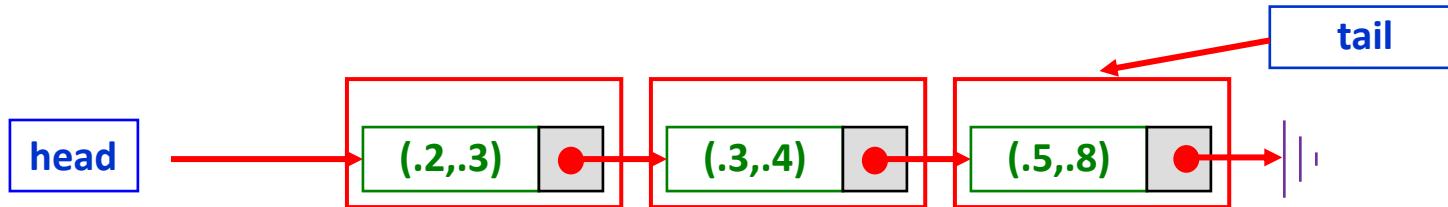
Löschen eines Punkts

- Für einen gegebenen Punkt p wird nur das **erste** Listenelement mit einem Punkt q, der q.equals(p) erfüllt, gelöscht.
- Beim Durchlauf durch die Liste werden **prev** und **cur** verwendet.

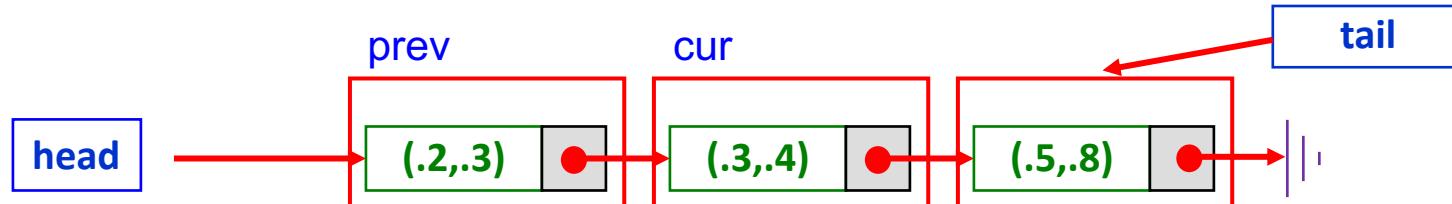
```
public boolean remove(Point2D p) {  
    ListElement prev = null;  
    for (ListElement cur = head; cur !=null; cur = cur.next) {  
        if (p.equals(cur.data)) {  
            if (prev == null)  
                if (head == tail)  
                    head = tail = null;  
                else  
                    head = head.next;  
            else  
                prev.next = cur.next;  
            return true;  
        }  
        prev = cur;  
    }  
    return false;  
}  
// Überprüfe auf Gleichheit  
// Vorgänger existiert nicht  
// Liste hat ein Element  
// Löschen des 1. Elements  
// Freigeben des Speichers  
// Löschen erfolgreich  
// Element p nicht gefunden  
// Anpassen von prev  
// Kein Element gelöscht
```

Beispiel

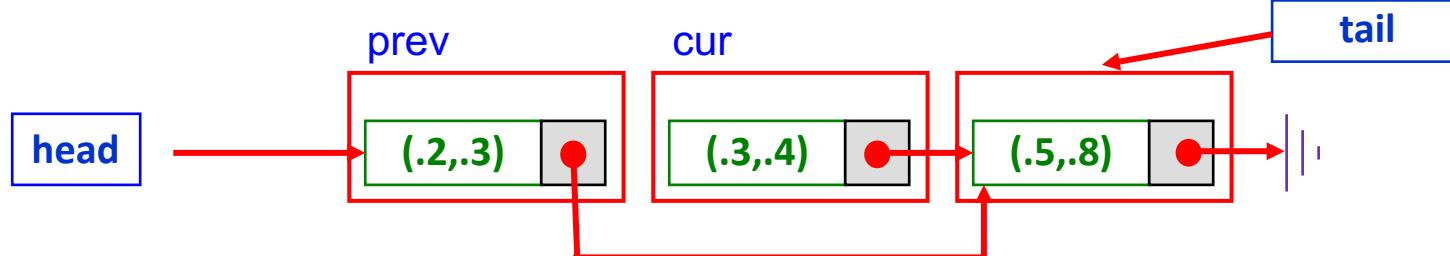
- Aufruf von remove (new Point2D(.3,.4)) auf folgender Liste.



- Situation nach dem Finden des zu löschen Elementes

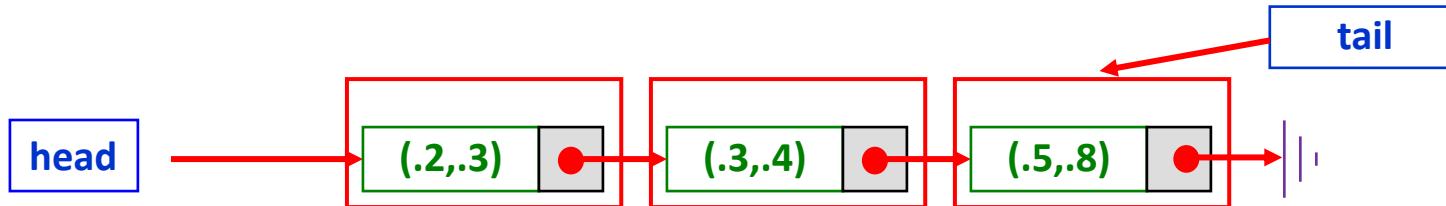


- Umhängen des Verweises

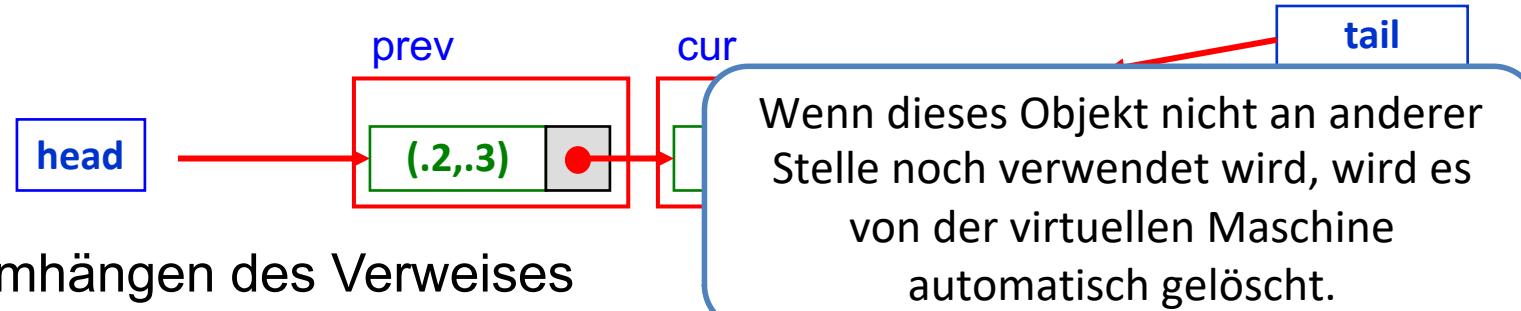


Beispiel

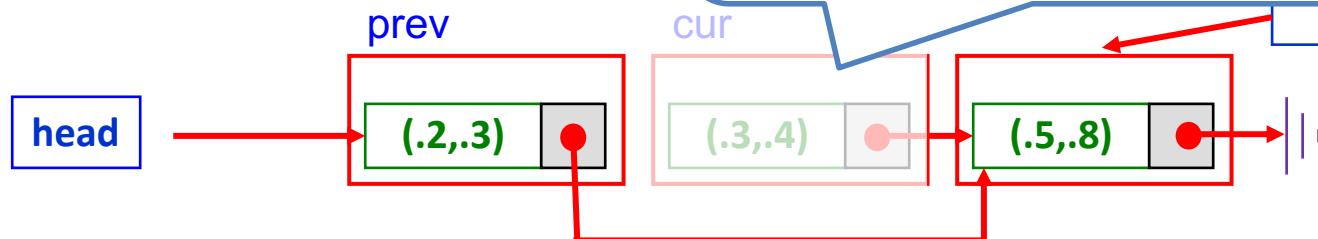
- Aufruf von remove (new Point2D(.3,.4)) auf folgender Liste.



- Situation nach dem Finden des zu löschen Elementes



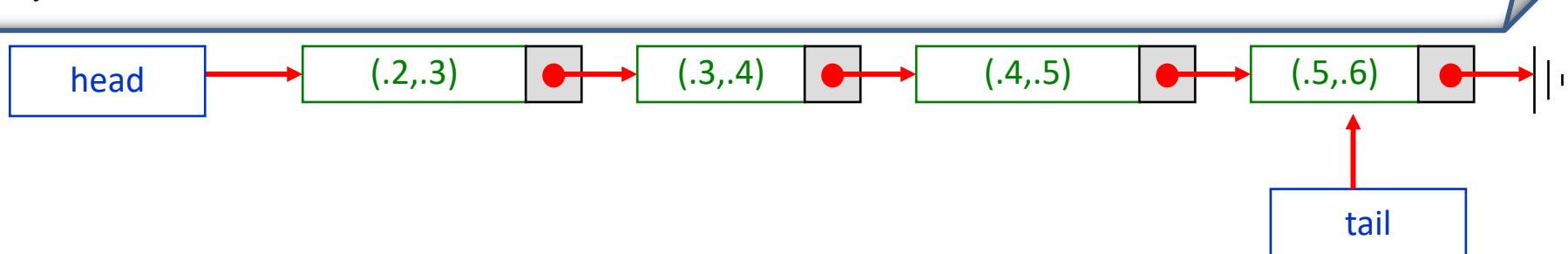
- Umhängen des Verweises



Verwendung einer LinkedList

- Zunächst erzeugen wir eine Liste, dann fügen wir verschiedene Punkte ein:

```
public static void main(String[] args) {  
    LinkedList list = new LinkedList();  
    list.add(new Point2D (.2,.3));  
    list.add(new Point2D (.3,.4));  
    list.add(new Point2D (.4,.5));  
    list.add(new Point2D (.5,.6));  
  
    System.out.println("Liste " + list); // toString sollte überschrieben sein.  
    System.out.println("Gefunden: " + list.contains(new Point2D (.3,.3)));  
    System.out.println("x des 1. Punkts:" + list.get(0).getX());  
}
```



Diskussion

- Die Klasse LinkedList erfüllt nur die Anforderungen für eine Liste mit Objekten der Klasse Point2D.
- Die Implementierung ist **nicht generisch** genug.
 - Soll eine Liste für die Klasse Konto erstellt werden, müsste eine komplett neue Liste implementiert werden.
- Erster Lösungsvorschlag des Problems
 - Ersetze in der bisherigen Implementierung die Klasse Point2D durch die Klasse Object.
 - Damit kann die Implementierung für die Verwaltung von Objekten beliebiger Klassen verwendet werden.
 - Was wäre der Nachteil dieser Lösung?

12.2 Generische Listen mit Object

- Im Folgenden soll die zuvor genannte generische Lösung einer Liste vorgestellt werden.
 - *Verwendung der Klasse Object (statt Point2D) als Typ für das Datenfeld data.*
- Dadurch ergeben sich folgende Änderungen in der Klasse ListElement.

```
class ListElement{  
    protected Object data;  
    protected ListElement next;  
  
    /** siehe oben */  
    ListElement(Object in){  
        this(in, null);  
    }  
  
    /** siehe oben */  
    ListElement(Object in, ListElement ptr){  
        next = ptr;  
        data = in;  
    }  
}
```

Einfügen eines Objekts in LinkedList

```
public class LinkedList implements DynamicSet {  
    // Das Interface DynamicSet muss ebenfalls angepasst werden.  
    private ListElement head, tail;  
  
    // Hier müssen noch Konstruktoren eingefügt werden.  
  
    /** siehe oben */  
    public boolean add(Object p) {  
        if (p == null) return false;  
        if (head == null)  
            head = tail = new ListElement(p);  
        else {  
            tail.next = new ListElement(p);  
            tail = tail.next;  
        }  
        return true;  
    }  
    /** Löschen eines Elements aus der Liste */  
    public boolean remove(Object e) {...}  
  
    /** Suchen nach einem Element in der Liste */  
    public boolean contains(Object e) {...}  
  
    ...  
}
```

Ausgabe eines Elements

```
public class LinkedList implements DynamicSet {  
    private ListElement head, tail;  
  
    // Hier müssen noch Konstruktoren eingefügt werden.  
    /** siehe oben */  
    public boolean add(Object p) { ... }  
    /** Löschen eines Elements aus der Liste */  
    public boolean remove(Object e) {...}  
    /** Suchen nach einem Element in der Liste */  
    public boolean contains(Object e) {...}  
  
    /** Prüft, ob eine Liste leer ist. */  
    public boolean isEmpty() {  
        return head == null;  
    }  
  
    /** Liefert das i-te Element aus der Liste */  
    public Object get(int i) {  
        if ((i < 0) || i >= size()) throw new IndexOutOfBoundsException();  
        ListElement cur = head;  
        for (int j = 0; j < i; j += 1)  
            cur = cur.next;  
        return cur.data;  
    }  
    ...  
}
```

LinkedList von Object

- Erster Lösungsvorschlag des Problems
 - Ersetze in der bisherigen Implementierung die Klasse Point2D durch die Klasse Object.

```
public static void main(String[] args) {  
    LinkedList list = new LinkedList();  
    list.add(new Point2D (.2,.3));  
    list.add(new Point2D (.3,.4));  
    list.add(new Point2D (.4,.5));  
    list.add(new Point2D (.5,.6));  
  
    System.out.println("Liste " + list); // toString Methode überschrieben sein.  
    System.out.println("Gefunden: " + list.contains(new Point2D (.3,.3)));  
    System.out.println("x des 1. Punkts:" + list.get(0).getX());  
}
```

Das ist nicht erlaubt, da
getX() nicht in Object
deklariert ist.

Nachteile der generischen Lösung

- Es kann nicht sichergestellt werden, dass eine Liste nur **Objekte einer Klasse** enthält.
 - Prinzipiell wäre es möglich, eine gemischte Liste mit Objekten der Klasse Konto und der Klasse Point2D zu erstellen.
- Die Methode get(int) liefert als Ergebnis nur **Objekte der Klasse Object**, auch wenn zuvor Objekte der Klasse Point2D eingefügt wurden.
 - Zusätzlich müssen diese Objekte mit einem *Down-Cast* noch in die entsprechende Klasse zurücktransformiert werden.
 - Dies funktioniert nur dann, wenn sich in der Liste nur Objekte einer Klasse befinden.



12.3 Generische Listen mit parametrisierten Typen

- Seit Java 5.0: parametisierte Datentypen, meist generische Datentypen genannt.
 - Man spricht auch von parametrischer Polymorphie.
- Generische Datentypen können von einem oder mehreren Typ-Parametern abhängen.
- Es können sowohl Klassen als auch Schnittstellen auf diese Weise definiert werden.
 - Dadurch werden unsere Probleme bei der bisherigen generischen Implementierung einer Liste gelöst.

Definition und Anwendung von generischen Datentypen

- In Java werden Typ-Parameter in spitzen Klammern nach dem Klassen- bzw. Schnittstellennamen angegeben.
 - Beispiele
 - interface List<E> { ... }
 - class LinkedList<E> { ... }
 - Ähnlich zu Parametervariablen in Methoden wird dadurch einer Klasse bzw. einer Schnittstelle ein Parameter übergeben. Der Unterschied ist, dass es sich dabei um einen **Datentyp (Klasse)** handelt.

Definition und Anwendung von generischen Datentypen

- Innerhalb der Schnittstellen und Klassen kann der Typparameter (E) als (fast) normaler Datentyp verwendet werden, z.B.
 - Als Typ von Variablen und Parametern
 - Als Rückgabetyp
 - Als Typparameter
 - **ABER NICHT:** für Konstruktoraufrufe oder in der extends oder implements Klausel
- Erst bei der **Deklaration von Variablen und der Erzeugung der Objekte** einer generischen Klasse muss die konkrete Klasse für den Datentyp E festlegt werden.
 - Man spricht dann von einer **Instanziierung** des generischen Datentyps.
`LinkedList<Konto> list = new LinkedList<Konto>();`
 - Dadurch wird der Typ E in der Klasse `LinkedList` durch die Klasse `Konto` ersetzt.

Modifizierte Klasse

```
class ListElement<E> implements List<E> {  
    protected E data;  
    protected ListElement<E> next;  
  
    /**  
     * Konstruktor der Klasse  
     * @param in Objekt des Elementtyp E, das in dem ListElement  
     *         referenziert werden soll.  
     */  
    ListElement(E in){  
        this(in, null);  
    }  
  
    /**  
     * Konstruktor der Klasse  
     * @param in Objekt des Typs E, das in dem ListElement  
     *         referenziert werden soll.  
     * @param ptr Ein gültiger Verweis auf ein ListElement  
     */  
    ListElement(E in, ListElement<E> ptr){  
        next = ptr;  
        data = in;  
    }  
}
```

Oberklasse kann auch generisch sein!
Typ-Parameter kann in der Klassenerweiterung verwendet werden. Typparameter wird wieder als Typparameter verwendet.

Verwendung generischer Klassen

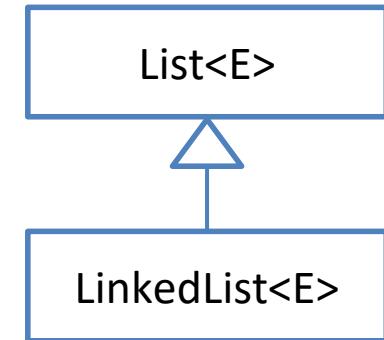
- Wie wir gesehen haben können in unserem Beispiel die Klasse und Schnittstelle relativ einfach generisch gemacht werden.
 - Zu beachten ist, dass bei der Klasse `LinkedList<E>` die Schnittstelle `List<E>` implementiert wird.
- Bevor die Details erläutert werden, wird zunächst die Verwendung einer generischen Klassen vorgestellt.
 - Erstellung einer Liste mit Objekten der Klasse `Point2D`

Beispiel: Liste mit Punkten

Vorteile

- In die Punktliste können nur Punkte eingefügt werden. Versucht man ein Objekt der Klasse Konto einzufügen, wird der **Compiler** ein Fehler melden.
- Beim Lesen eines Objekts der Klasse Point2D aus der Liste wird **keine Cast-Operation** mehr benötigt. Wir bekommen das Objekt mit der gewünschten Klasse zurück.
- Die Verwendung von generischen Klassen erfordert **wenig Aufwand**.

```
public static void main (String[] args) {  
    List<Point2D> list = new LinkedList<Point2D>();  
    list.add(new Point2D(.2,.3));  
    list.add(new Point2D(.3,.4));  
    list.add(new Point2D(.4,.5));  
    list.add(new Point2D(.5,.6));  
  
    System.out.println("Liste " + list);  
    System.out.println("Gefunden: " +  
        list.contains(new Point2D (.3,.3)));  
    System.out.println("x des 1. Punkts:" +  
        list.get(0).getX());  
}
```



Das ist OK: wir wissen, dass Element ein Point2D Sein muss.

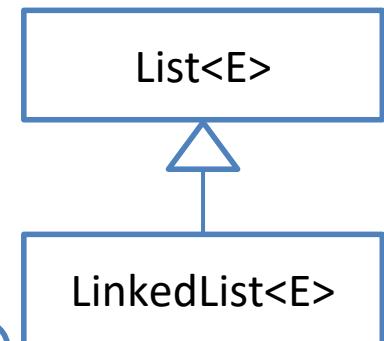
Beispiel: Liste mit Punkten

Vorteile

- In die Punktliste können nur Punkte eingefügt werden. Versucht man ein Objekt der Klasse Konto einzufügen, wird der **Compiler** ein Fehler melden.
- Beim Lesen eines Objekts der Klasse Point2D aus der Liste wird **keine Cast-Operation** mehr benötigt. Wir bekommen das Objekt mit der gewünschten Klasse zurück.
- Die Verwendung von generischen Klassen erfordert **wenig Aufwand**.

```
public static void main (String[] args) {  
    List<Point2D> list = new LinkedList<Point2D>();  
    list.add(new Point2D(.2,.3));  
    list.add(new Point2D(.3,.4));  
    list.add(new Point2D(.4,.5));  
    list.add(new Point2D(.5,.6));  
    list.add(new String("..."));  
}
```

Gibt bereits beim
Einfügen einen Fehler.



12.4 Generische Klassen und Schnittstellen der Java API

- Das Paket `java.util` bietet unter anderem Algorithmen, Klassen und Schnittstellen zur (effizienten) Verwaltung von Mengen beliebiger Datenobjekte.
 - Listen
 - Suchbäume
 - Sortierverfahren
- Wichtige generische Schnittstellen
 - `Comparable`
 - Vergleich von zwei Objekten
 - `Iterator`
 - Durchlauf durch eine Datenstruktur unabhängig von der konkreten Implementierung

Interface List<E>

- Das Interface ist eine zentrale Schnittstelle in `java.util`.
 - Das Interface hat noch weitere Ober-Interfaces, wie z. B. `Iterable<E>`. Dieses Interface werden wir gleich betrachten.
 - Das Interface `List` wird von vielen Klassen implementiert, wie z. B.
 - `ArrayList<E>`
 - `LinkedList<E>`
 - `SortedList<E>`
- Wichtige Methoden im Interface
 - `boolean add(E e)`
 - `E get(int index)`
 - `List<E> subList(int fromIndex, int toIndex)`

Interface Comparable<T>

- Für geordnete Daten verwendet man das **Interface Comparable<T>**, das folgende Methode für Objekte **o** vom Typ **T** vorschreibt.

```
int compareTo(T o);
```

- Das Ergebnis von **compareTo** ist vom Typ int. Es gilt folgende Konvention:
 - Wenn **a.compareTo(b)** negativ ist, interpretiert man dies als **a < b**.
 - Wenn **a.compareTo(b)** 0 ist, interpretiert man dies als **a == b**.
 - Wenn **a.compareTo(b)** positiv ist, interpretiert man dies als **a > b**.
- Beispiel

```
class Point2D implements Comparable<Point2D> {  
    ...  
    public int compareTo(Point2D o) { // lexikografischer Vergleich  
        int xc = Double.compare(x, o.x); // Verwenden  
        if (xc < 0)  
            return -1;  
        else  
            return (xc == 0) ? Double.compare(y, o.y) : 1;  
    }  
}
```

Interface Iterator<E>

- Ein Behälter ist ein generischer Datentyp zur Verwaltung von Mengen beliebiger Objekte.
 - Listen sind nur ein Beispiel für einen Behälter.
- Ein **Iterator** liefert die Elemente eines Behälters (oder eines Teils) in einer spezifischen Reihenfolge („Aufzählung der gewünschten Elemente“).
 - Die Methode **hasNext()** liefert **true**, wenn bei der aktuellen Aufzählung der Elemente des Behälters noch weitere Elemente anstehen.
 - Die Methode **next()** produziert das nächste Element der Aufzählung.
 - Die Methode **remove()** entfernt das Element aus dem Behälter, das zuletzt mit **next** abgeliefert wurde. Diese Methode ist optional – d.h. kann auch weggelassen werden und führt dann ggf. zu einer Ausnahme.

```
public interface Iterator<E> {  
    public boolean hasNext();  
    public E next();  
    public void remove();  
}
```

Beispiel: Ein Iterator für Listen

```
import java.util.List;
/**
 * Ein Iterator, der alle Elemente in einer Liste liefert.
 */
public class MyListIterator<E> implements Iterator<E> {
    List<E> list; // Zu durchlaufende Liste

    public MyListIterator(List<E> l) {
        list = l;
    }

    public boolean hasNext() {
        return !list.isEmpty();
    }

    public E next() {
        E tmp = list.get(0);
        list = list.subList(1, list.size()); // Liefert die Restliste
        return tmp;
    }
}
```

Die remove()-Methode ist als Default-Methode im Interface Iterator implementiert und wirft dort eine UnsupportedOperationException.

Interface Iterable<E>

- Analog zu einem Array kann auch eine Liste mit der **for-each**-Schleife durchlaufen werden.
- Voraussetzung hierfür ist, dass die Liste noch die generische Schnittstelle Iterable<E> implementiert.

```
public interface Iterable<E> {  
    public Iterator<E> iterator();  
}
```

- Die Schnittstelle List erweitert die Schnittstelle Iterable

```
public interface List<E> extends Iterable<E>{ ... }
```

und die Klasse LinkedList implementiert die Methode.

```
public class LinkedList<E> implements List<E>{  
    ...  
  
    public Iterator<E> iterator() {  
        return new MyListIterator<E>(this);  
    }  
}
```

Anwendung der for-each Schleife

- Da unsere Klasse LinkedList die Schnittstelle Iterable implementiert, kann der Durchlauf durch die Listen mit einer for-each-Schleife erfolgen.

```
public static void main(String[] args) {  
    List<Point2D> list = new LinkedList<Point2D>();  
    list.add(new Point2D (.2,.3));  
    list.add(new Point2D (.3,.4));  
    list.add(new Point2D (.4,.5));  
    list.add(new Point2D (.5,.6));  
  
    // Ausgabe aller Punkte mit der for-each Schleife  
    for (Point2D p: list)  
        System.out.println("Punkt: " + p);  
}
```

- Die for-each-Schleife kann für alle Klassen, die Iterable implementieren, genauso wie bei Arrays benutzt werden.

12.5 Java Generics im Detail

- Bisher haben wir am Beispiel von Listen die wichtigsten Konzepte generischer Klassen erklärt.
- In diesem Abschnitt sollen weitere Details von Java Generics behandelt werden.

Syntax

- Java Generics unterstützt die Parametrisierung mit Typen bei
 - Klassen
 - Schnittstellen
 - Methoden
- Anzahl der Parameter
 - beliebig, aber i. A. ist die Anzahl kleiner 3.
- Syntax
 - Angabe der Parameterliste in eckigen Klammernpaar „< ... >“
 - Bei Klassen und Schnittstellen nach dem entsprechenden Namen.
 - Parameter werden durch Kommata getrennt.

Beschränkungen

- **Keine primitive Typen** als Argument für einen Typparameter.

- **Keine Aufrufe von Konstruktoren** des Typparameters T

`T x = new T();` funktioniert nicht!

- **Kein Aufruf von statischen Methoden** des Typparameters T

`double avg = T.myStaticMethod();` funktioniert nicht!

- **Keine Verwendung** des Typparameters T **in statischen Methoden/bei statischen Felddeklarationen**

- **Keine Allokation eines Arrays von** Typparameter T

`T[] arr = new T[12];` funktioniert nicht!

`List<Point2D>[] larr = new List<Point2D>[12];` funktioniert nicht!

`List<Point2D>[] larr = new List[12];` **funktioniert (ab Java 1.7)!**

Beschränkungen

- **Keine primitive Typen** als Argument für einen Typparameter.

- ~~K~~ ~~List<int> li;~~

T
funktioniert nicht!

- **//das Folgende funktioniert aber:**
List<Integer> = ...;
li.add(1);
// dabei wird der int 1 in ein
// automatisch in ein Integer-Objekt
// umgewandelt.

ers T
funktioniert nicht!

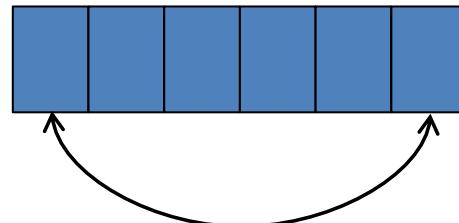
List<Point2D>[] larr = new List[12];

funktioniert nicht!
funktioniert nicht!
funktioniert (ab Java 1.7)!

Generische Methoden

- Objektmethoden dürfen Typparameter der Klasse verwenden
- Außerdem erlaubt Java, sowohl statischen Methoden als auch Objektmethoden eigene Typparameter zu deklarieren.
 - Die Liste der generischen Typen wird vor dem Rückgabetyp der Methode angegeben.
- Beispiel

```
// class Main
public static <T> void swap(T[] arr) {
    T tmp = arr[0];
    arr[0] = arr[arr.length-1];
    arr[arr.length-1] = tmp;
}
```



Automatische Typ-Bestimmung

- Oft kann der Java-Compiler den Typ für Typparameter selbst bestimmen
- Bei Erzeugung einer Instanz:
 - //explizite Typangabe:
`List<Integer> li = new LinkedList<Integer>();`
 - //Automatische Typbestimmung
`List<Integer> li = new LinkedList<>();`
- Beim Aufruf einer Methode: (gegeben: Integer[] arr = null;)
 - //explizite Typangabe:
`Main.<Integer>swap(arr);`
 - //Automatische Typbestimmung
`Main.swap(arr);`

Einschränkung des Typparameters

- Bei den bisherigen generischen Typen erlauben wir beliebige Typen bei der Instanziierung.

```
public static void main(String[] args) {  
    // Liste mit Punkten  
    List<Point2D> listp = new LinkedList<>();  
  
    // Liste mit Konten  
    List<Konto> listk = new LinkedList<>();  
  
    // Liste mit Integern  
    List<Integer> listi = new LinkedList<>();  
  
    // Liste mit Objekten  
    List<Object> listo = new LinkedList<>();  
}
```

- Dies ist nicht immer erwünscht, da in bestimmten Fällen von Typen gewisse Eigenschaften gefordert werden.
 - Z. B. sollte eine **geordnete Liste** nur mit Typen instanziert werden, die die **Schnittstelle Comparable** unterstützen.

extends-Klausel

- Bei einer generischen Klasse kann diese Eigenschaft durch Angabe des Schlüsselworts **extends** und einer Klasse oder Schnittstelle gefordert werden.
- Beispiel
 - Es soll eine generische Klasse erstellt werden, um beliebige Zahlen zu addieren.
 - Anmerkung:
Die **Klasse Number** aus der Java API ist die Oberklasse von all diesen Klassen wie z. B. der Klasse Integer.
- Lösung

```
class Accumulator<T extends Number> {  
    ...  
    // Über T kann jetzt auf die Methoden der Klasse  
    // Number zugriffen werden.  
}
```

```
Accumulator<Integer> ai = new Accumulator<>(); // funktioniert
```

```
Accumulator<String> ai = new Accumulator<>(); // funktioniert nicht.
```



extends-Klausel mit Schnittstellen

- Um eine Ordnung in der Liste sicherzustellen, sollte der generische Typ die Schnittstelle Comparable unterstützen.
 - Zusätzlich müssten wir in unserer Listenimplementierung die Methode

```
boolean add(T elem)
```

noch so ändern, dass die Objekte entsprechend der Ordnung in der Liste liegen.
- Entsprechend zu Klassen kann auch die **extends-Klausel bei Schnittstellen** verwendet werden.
 - Die Schnittstelle Comparable ist aber wiederum generisch.
- Der generische Typ T darf wieder als Typparameter der Schnittstelle/Klasse verwendet werden, die T implementieren/erweitern soll.

```
class SortedList<T extends Comparable<T>> { ... }
```

extends-Klausel mit mehreren Schnittstellen



```
public interface Cat {  
    void miau();  
}
```

```
public interface Dog {  
    void wuff();  
}
```

- Die Methode catDog erwartet ein Objekt einer Klasse, die sowohl das Interface Cat als auch das Interface Dog implementiert

```
public <T extends Dog & Cat> void catDog(T t) {  
    t.miau();  
    t.wuff();  
}
```

extends-Klausel mit Klasse und Schnittstelle

- In manchen Situationen ist es also nützlich, dass der Typparameter mehrere Schnittstellen unterstützen soll. Es ist sogar Folgendes möglich:

- Sei A eine Klasse oder Schnittstelle und seien BI, CI Schnittstellen, dann wird durch

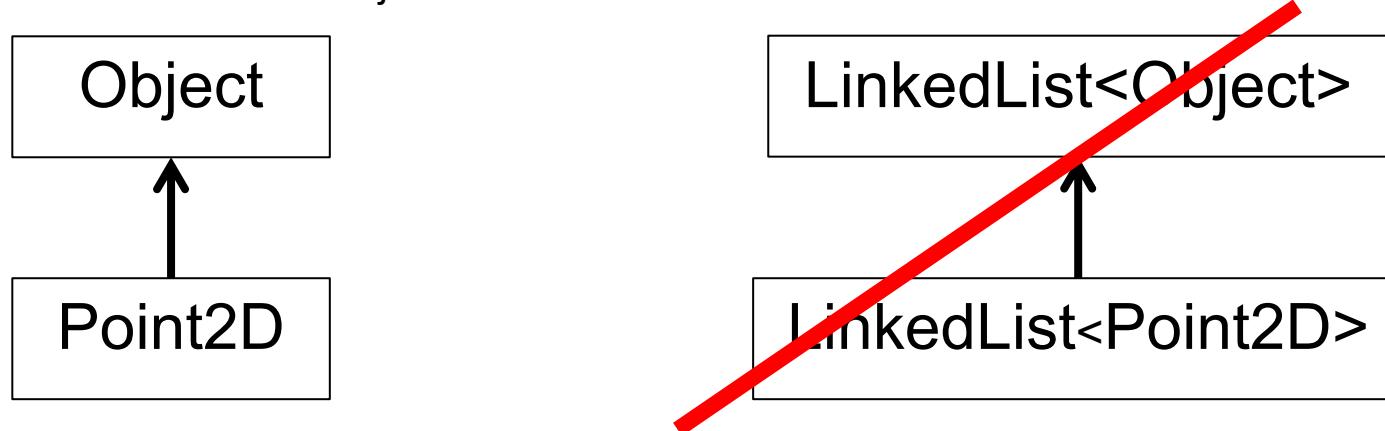
```
class MyGenericClass<T extends A & BI & CI> { ... }
```

gefordert, dass bei der Instanziierung nur Klassen verwendet werden können, die alle Schnittstellen implementieren und (im Fall, dass A eine Klasse ist) A erweitern.

- Wichtig dabei ist, dass nur **der erste Typparameter eine Klasse** sein darf. Alle **anderen Parameter müssen Schnittstellen** sein.

12.6 Wildcards – Motivation (1)

- Bei den bisherigen Möglichkeiten in Generics gibt es noch ein paar Probleme.
 - Die Klasse Object ist Oberklasse von Point2D.
 - Jedoch ist `LinkedList<Object>` keine Oberklasse von `LinkedList<Point2D>` !



- Damit ist dieser Programmschnipsel **nicht** korrekt.

```
LinkedList<Point2D> points = new LinkedList<Point2D>(); // funktioniert
```

```
LinkedList<Object> objects = new LinkedList<Point2D>(); // nicht erlaubt
```

- Es gibt also keine Polymorphie zwischen den beiden Listen-Klassen.

Warum ist es nicht erlaubt?

- Betrachten wir folgende Situation
 - Eine Methode zum Einfügen eines neuen Konto-Objekts in eine Liste vom Typ `LinkedList<Object>`.

```
void addSomethingToList (LinkedList<Object> lif) {  
    // Wir fügen jetzt etwas zu lif hinzu, wie z. B. ein Konto:  
    lif.add(new Konto());  
}
```

- Aufruf der Methode mit einem Parameter vom Typ `LinkedList<Point2D>`
 - Das sollte aber verhindert werden, da **Konto keine Unterklasse von Point2D ist.**

```
public static void main(String[] args) {  
    // Liste mit Punkten  
    LinkedList<Point2D> points = new LinkedList<>();  
    points.add(new Point2D(0.2,0.3));  
  
    addSomethingToList(points);      // Funktioniert nicht!  
}
```



Warum sollte es erlaubt sein?

- Es soll eine Methode bereitgestellt werden, um eine beliebige Liste auszugeben.

```
void printList (LinkedList<Object> lif) {  
    for (Object e: lif)  
        System.out.println(e);  
}
```

- Der Aufruf der Methode printList für die Liste points ist leider nicht erlaubt!

```
public static void main(String[] args) {  
    // Liste mit Punkten  
    LinkedList<Point2D> points = new LinkedList<>();  
    points.add(new Point2D(.2,.3));  
  
    printList(points);           // Funktioniert nicht!  
}
```

Wildcard-Typ

- Durch Verwendung eines Wildcard-Typs wird **dieses Problem** behoben.

```
void printList (LinkedList<?> lif) {  
    for (Object e: lif)  
        System.out.println(e);  
}
```

- Der Aufruf der Methode für die Liste points funktioniert jetzt!

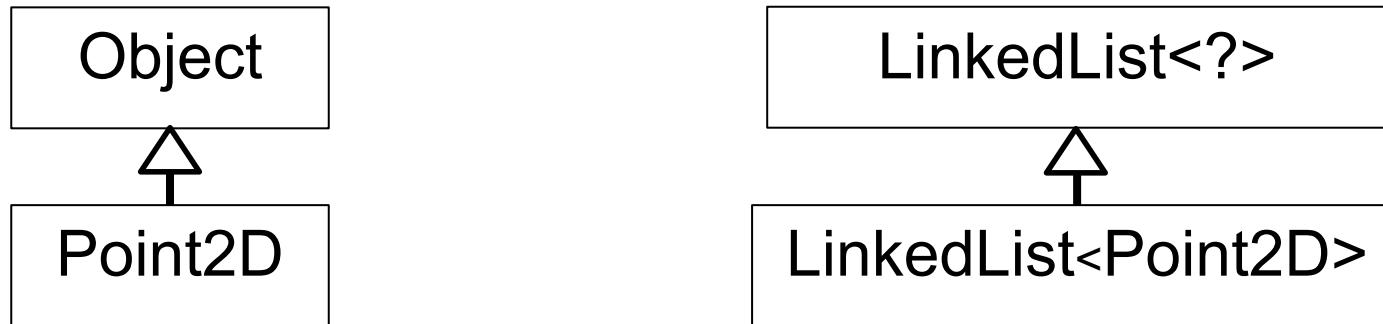
```
public static void main(String[] args) {  
    // Liste mit Punkten  
    LinkedList<Point2D> points = new LinkedList<>();  
    points.add(new Point2D(.2,.3));  
    printlist(points);           // alles in Ordnung!  
}
```

- Wichtige Beobachtungen

- In der Methode printList dürfen wir nicht die Methode add aufrufen, da **diese Methode den Typparameter in der Parameterliste verwendet (boolean add(T t) {...})**
- **Grund hierfür ist, dass der Typparameter der generischen LinkedList<?> unbekannt ist.**

Beziehung zwischen Klassen

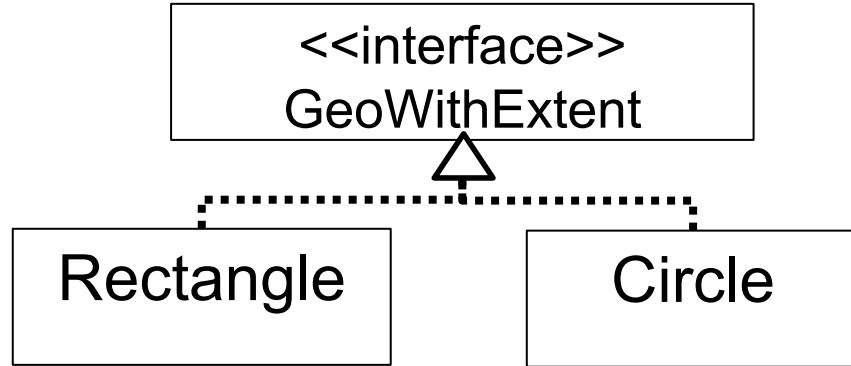
- Durch `LinkedList<?>` wird ein Obertyp für alle `LinkedList`-Klassen zur Verfügung gestellt.



- Jedoch gelten folgende zwei Einschränkungen:
 1. Liefern Methoden den Typparameter der `LinkedList<?>` als Ergebnis, können wir nur davon ausgehen, dass das Ergebnis zur Klasse `Object` gehört.
 2. Es darf kein Aufruf einer Methode von `LinkedList<?>` erfolgen, in der die Parameterliste den Typparameter verwendet.
- Die erste kann durch nach oben beschränkte Wildcards gelockert werden.
- Die zweite durch nach unten beschränkte Wildcards aufgehoben werden.

Motivation für oben beschränkte Wildcards

- Betrachten wir folgende Klassenhierarchie



- Die Schnittstelle `GeoWithExtent` besitzt eine Methode `area()` zur Flächenberechnung.
- Die Klassen `Circle` und `Rectangle` sind zwei Klassen, die die Schnittstelle implementieren.
- Wir betrachten im Folgenden drei verschiedene Listen.

```
LinkedList<GeoWithExt> geos = LinkedList <>();
LinkedList<Rectangle> rects = LinkedList <>();
LinkedList<Circle> circles = LinkedList <>();
```

- Können wir eine generische Methode bereitstellen, um für alle drei Listen die Summe der Flächeninhalte der Objekte zu berechnen?

Obere Schranke für Wildcards

- Benutzen wir den normalen Wildcard `LinkedList<?>` steht uns die Methode `double area()` nicht zur Verfügung.
- Deshalb gibt es nach **oben beschränkte Wildcards**, bei der wir nach `?` das Schlüsselwort **extends** und ein Typ als obere Schranke hinzufügen können.
- In unserem Beispiel:

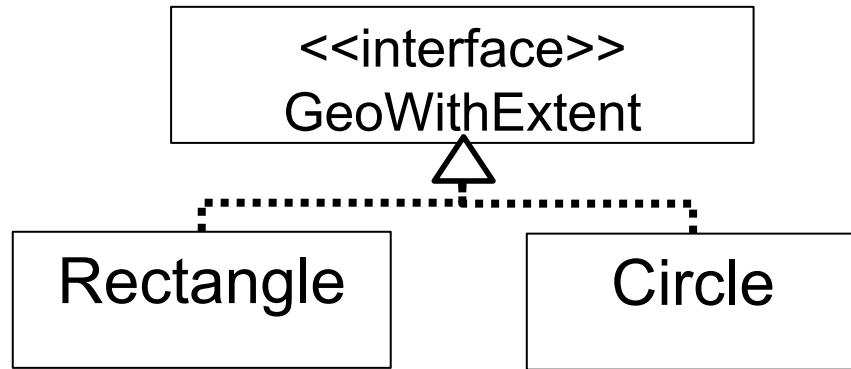
```
static double areaOverAll(LinkedList<? extends GeoWithExtent> list) {  
    double res = 0.0;  
    for (GeoWithExtent s: list) {  
        res += s.area();  
    }  
    return res;  
}
```

- Diese Methode kann für alle Listen parametrisiert mit einer UnterkLASSE von `GeoObjectsWithExtent` verwendet werden.

```
double resg = areaOverAll(geos);           // Rechtecke und Kreise  
double resr = areaOverAll(rects);          // Rechtecke  
double resc = areaOverAll(circles);         // Kreise
```

Motivation für unten beschränkte Wildcards

- Wir betrachten wieder die Klassenhierarchie



- Die Klasse Circle soll die Methode `double getRadius()` besitzen.
- Ist es sinnvoll, ein Objekt der Klasse `LinkedList<GeoWithExt>` an eine Variable der Klasse `LinkedList<Circle>` zu übergeben?
 - Wenn ja, wie können wir dies in Java unterstützen?
- Wir betrachten im Folgenden zwei Beispiele.

Beispiel 1

```
static double getAllRadii(LinkedList<Circle> circles) {
    double res = 0;
    for (Circle c: circles) {
        res += c.getRadius();
    }
    return res;
}

public static void main(String[] args) {
    LinkedList<GeoWithExt> geos = new LinkedList<GeoWithExt>();
    geos.add(new Rectangle());
    double total = getAllRadii(geos); // // Funktioniert nicht!
}
```

- In diesem Fall (lesender Zugriff) darf die Übergabe des Objekts der Listenoberklasse an einer Variable einer Listenunterklasse **nicht** erfolgen.
 - **Grund:** In einer Liste vom Typ `LinkedList<GeoWithExt>` können Objekte einer Klasse sein, für die die **keine** Methode `getRadius()` existiert.
- Der Compiler zeigt zu Recht einen Fehler an!

Beispiel 2

```
static void addCircle(LinkedList<Circle> circles) {
    circles.add(new Circle());
}

public static void main(String[] args) {
    LinkedList<GeoWithExt> geos = new LinkedList<GeoWIthExt>();
    geos.add(new Rectangle());
    addCircle(geos);                                // Funktioniert leider auch nicht!
}
```

- In diesem Fall könnte man eigentlich den Aufruf der Methode addCircle erlauben, aber der Compiler ist zu restriktiv.



Untere Schranke

- Damit der Compiler solche Programme akzeptiert gibt es in Java nach unten beschränkte Wildcards.
 - Nach dem ? folgt das **Schlüsselwort super** und eine Klasse L als untere Schranke der Klassen K, die beim Aufruf verwendet werden dürfen.
- Beispiel

```
static void addCircle(LinkedList<? super Circle> circles){  
    circles.add(new Circle()); // Schreibender Zugriff erlaubt  
}  
  
public static void main(String[] args){  
    LinkedList<GeoWithExt> geos = new LinkedList<GeoWIthExt>();  
    geos.add(new Rectangle());  
    addCircle(geos); // Funktioniert  
}
```

- Der Aufruf von add(new Circle()) ist erlaubt, da nur Listen mit folgenden Klassen möglich sind: `LinkedList<Circle>`, `LinkedList<GeoWithExtent>`, ..., `LinkedList<Object>`
- In all diesen Listenklassen darf ein Objekt der Klasse Circle oder einer Unterklasse von Circle hinzugefügt werden.

Zusammenfassung Wildcards

- Wir haben zwei unterschiedlich beschränkte Wildcards
 - nach oben beschränkte Wildcards: <? extends OT>
 - **nur Methodenaufrufe ohne Typparameter in der Parameterliste**
 - Verwendung der null-Referenz beim Aufruf immer möglich
 - nach unten beschränkte Wildcards: <? super UT>
 - **nur Objekte der Klasse UT (und Unterklassen) als Parameter bei Methodenaufruf mit Typparameter in der Parameterliste.**
 - Im Fall, dass die Methode als Ergebnistyp den Typparameter hat, kann man das Ergebnis nur an eine Variable vom Typ Object übergeben.
- Wildcards sind ziemlich fortgeschrittene, aber nützliche Konzepte.
 - Beispiel: Methode copy aus der Klasse java.util.Collections

```
/** Copies all elements from one list into another */
static <T> void copy(List<? super T> dest, List<? extends T> src){...}
```

13. Lambdas

- In diesem Kapitel geht es um wichtige Konzepte in Java, um den Programmieraufwand bei typischen Problemen zu reduzieren. Dies bedeutet:
 - wenige Zeilen von Code
 - wenige Fehler

→ Niedrigere Kosten bei der Softwareerstellung
- Folgende Konzepte werden vorgestellt:
 - Funktionale Programmierung in Java (Lambdas)
 - Streams

Motivation

- Problem

- Verwaltung einer Menge von Objekten, wie z. B. POIs (Points of Interests) oder Musiker, in einer Liste.
- Unterstützung von Suchoperationen, auch als **Filter** bezeichnet.
 - Welche POI (Points Of Interests) liegen in der Umgebung von Marburg?
 - Zeige mir die Musiker mit Namen "Wilson"?
 - Jede Suchoperation soll die Ergebnisse elementweise als Iterator liefern.

- Erste Lösung

- Die Elemente aus der Liste<Musician> können die Eingabe über die Methode iterator() als Iterator geliefert werden.
- Für jede Anfrage wird eine eigene Iterator-Klasse implementiert.

Lösung mit einem Iterator

```
class FilterMusician implements Iterator<Musician> {
    Iterator<Musician> it;      // Iterator, der über alle Element geht.
    Musician cur = null;        // Musician, der aktuell gefunden wurde.

    public FilterMusician(Iterator<Musician> in) {
        it = in;                // Speichern des Iterators
        cur = computeNext();     // Nächstes Ergebnis berechnen
    }

    private Musician computeNext() {
        while (it.hasNext()) {           // Durchlauf bis zum nächsten Ergebnis
            Musician tmp = it.next();
            if (tmp.getName().equals("Wilson")) // Prüfen der Bedingung
                return tmp;               // Treffer
        }
        return null;                  // Kein Ergebnis
    }

    public boolean hasNext() {
        return (cur != null);
    }

    public Musician next() {
        Musician tmp = cur;
        cur = computeNext(); // Nächstes Ergebnis berechnen
        return tmp;
    }
}
```

Lösung mit einem Iterator

```
class FilterMusician implements Iterator<Musician> {  
    Iterator<Musician> it;      // Iterator, der über alle Element geht.  
    Musician cur = null;       // Musician, der aktuell gefunden wurde.  
  
    public FilterMusician(Iterator<Musician> in) {  
        it = in;                // Speichern des Iterators  
        cur = computeNext();     // Nächstes Ergebnis berechnen  
    }  
  
    private Musician computeNext() {  
        while (it.hasNext()) {  
            Musician tmp = it.next();  
            if (tmp.getName().equals("Wilson")) // Prüfen der Filterbedingung  
                return tmp;                  // Treffer gefunden  
            }                                // Kein Erfolg  
        return null;  
    }  
  
    public boolean hasNext() {  
        return cur != null;  
    }  
  
    public Musician next() {  
        Musician result = cur;  
        cur = computeNext();             // Nächstes Ergebnis berechnen  
        return result;  
    }  
}
```

- aufwändig
- redundant
- fehleranfällig
- viel Programmcode für wenig Funktionalität

Für alle Filterbedingung müssten eine eigene Implementierungen angefertigt werden, die sich nur in wenigen Punkten unterscheiden.

Besser: Test in eigene Klasse auslagern

```
class FilterMusician implements Iterator<Musician> {
    Iterator<Musician> it;          // Iterator, der über alles geht.
    MusicianPredicate condition;    // Filterbedingung
    Musician cur = null;           // Musician, die aktuell gefunden wurde.

    public FilterMusician(Iterator<Musician> in,
                          MusicianPredicate c) {
        it = in;                  // Speichern des Iterators
        condition = c;            // Speichern der Bedingung
        cur = computeNext();      // Nächstes Ergebnis berechnen
    }

    private Musician computeNext() {
        while (it.hasNext()) {      // Durchlauf bis zum nächsten Ergebnis
            Musician tmp = it.next();
            if (condition.test(tmp)) // Prüfen der Bedingung
                return tmp;         // Treffer
        }
        return null;               // Kein Ergebnis
    }

    ...
}
```

Noch besser: Generics benutzen

```
class FilterIterator<T> implements Iterator<T> {
    Iterator<T> it;                      // Iterator, der über alles geht.
    Predicate<T> condition;   // Filterbedingung
    T cur = null;                      // Element, die aktuell gefunden wurde.

    public FilterIterator(Iterator<T> in, Predicate<T> c) {
        it = in;                      // Speichern des Iterators
        condition = c;                // Speichern der Bedingung
        cur = computeNext();          // Nächstes Ergebnis berechnen
    }

    private T computeNext() {
        while (it.hasNext()) {        // Durchlauf bis zum nächsten Ergebnis
            T tmp = it.next();
            if (condition.test(tmp))   // Prüfen der Bedingung
                return tmp;           // Treffer
        }
        return null;                 // Kein Ergebnis
    }

    ...
}
```

Diskussion

- Vorteile
 - Wiederverwendung der Implementierung des FilterIterator
 - Minimale Implementierung der Filterbedingung
- Keine Duplizierung des Codes

```
public class WilsonFilter implements Predicate<Musician> {  
    boolean test( Musician p ) {  
        return p.getName().equals("Wilson");  
    }  
}
```

- Nachteile
 - Bei vielen Filterbedingungen gibt es auch viele Klassen
 - Diese werden meistens nur an einer Stelle genutzt
- Benennung der Klasse scheint überflüssig

13.1 Lambda-Ausdrücke in Java 8

- Motivation
 - Wir werden im Folgenden sehen, dass durch das in Java 8 neue Konzept von **Lambda-Ausdrücken** (kurz **Lambdas**) sich Filter sehr einfach hinschreiben lassen.
- Grundlage hierfür sind sogenannte funktionale Schnittstellen (engl.: Functional Interfaces) in Java.
 - Interface mit nur einer abstrakten Methode (mit Ausnahme der Methoden, die es in der Klasse Object noch gibt).
 - Da bei diesen Interfaces nur eine Methode existiert, kann man den Datentyp des Interfaces als Datentyp dieser Methode betrachten.
 - Zuweisung an Variablen
 - Übergabe als Parameter
 - Rückgabewert von Methoden



Vorhandene funktionale Interfaces

- In dem Paket **java.util.function** werden die in Java vorhandenen funktionale Interfaces bereitgestellt.
 - **Consumer<T>** mit der Methode **void accept(T)**
 - **Predicate<T>** mit der Methode **boolean test(T)**
 - Damit lassen sich sehr gut Filterbedingungen ausdrücken, um aus Mengen von Objekten einen Teil zu extrahieren.
 - **Function<T,R>** mit der Methode **R apply(T)**
 - Damit lassen sich beliebige Transformationen von dem Typ T auf den Typ R ausdrücken.
 - **Supplier<T>** mit der Methode **T get()**
 - Damit kann die Erzeugung von Objekten abgebildet werden.
- ...

Beispiel: Funktionales Interface Consumer

- Implementierung des Codes, der für jedes Listenelement ausgeführt werden soll als **Lambda**
- Die Methode (der funktionalen Schnittstelle) wird ohne umgebende Klasse implementiert:

```
public static void main(String[] args) {
    List<Musician> mlist = new LinkedList<>();
    ...
    mlist.forEach(
        p -> {
            if (p.getName().equals("Wilson"))
                System.out.println(p);
        } );
}
```

Beispiel: Funktionales Interface Consumer

- Implementierung des Codes, der für jedes Listenelement ausgeführt werden soll als **Lambda**
- Die Methode (der funktionale Konsument) der umgebende Klasse implementiert

```
public static void main( String[] args ) {
    List<Musician> musicians = ...;
    musicians.stream()
              .filter( p -> p.getName().equals("Wilson") )
              .forEach( p -> System.out.println(p) );
}
```

Noch nicht exakt dasselbe wie der Filter, da hier die Filterung (Bedingung) und Ausgabe der Elemente (Ausgabe) zusammen implementiert ist. Eine bessere Lösung sehen wir noch später.



Kurzer Rückblick

- Das sogenannte Lambda-Kalkül wurde 1930 von Alonzo Church vorgestellt.
 - Mathematiker wie **Alonzo Church** und **Alan Turing** haben grundlegende **Modelle zur Berechenbarkeit** entwickelt.
 - Sie zeigten, dass **nicht alle Funktionen berechenbar** sind.
 - Turing-Berechenbarkeit und das Lambda-Kalkül sind gleichmächtig.
- Das Lambda-Kalkül ist die Grundlage von **funktionalen Programmiersprachen**, wie z. B. Lisp und Haskell.
 - siehe auch das Modul **Deklarative Programmierung**
- Inzwischen wird auch in vielen objektorientierten Sprachen, wie z. B. Java, C++, C# und Scala, das Konzept des Lambda-Kalküls unterstützt.

Was ist ein Lambda?

- Ein Lambda ist eine **anonyme Funktion**.
 - Eine anonyme Funktion ist eine Funktion ohne Namen.
 - Die typische Schreibweise besteht aus
(Liste der Parameter) -> {Funktionsrumpf} wie z. B.
 - `(int x, int y) -> {return x + y;}` // Addition
- In gewissen Situation darf man noch Dinge bei der Angabe eines Lambdas weglassen. Z. B.:
 - `x -> {return x * x;}` // Parameter ohne Typ
 - `() -> x` // Funktionsrumpf mit einer
// Codezeile.

Besonderheiten bei Lambdas

- Ein Lambda hat keinen, einen oder mehrere durch Komma getrennte Parameter.
 - Der Typ der Parameter muss nur dann angegeben werden, wenn dieser sich **aus dem Kontext nicht eindeutig** ergibt.
 - Bei **einem Parameter** (ohne Angabe des Typs) werden **Klammern nicht benötigt**.
 - () bezeichnet eine Funktion ohne Parameter
- Der Methodenrumpf besteht aus einer Folge von Anweisungen.
 - Bei nur **einer Anweisung** (ohne Kontrollstrukturen) können die Mengenklammern und das Semikolon weggelassen werden.



Weitere Vorgehensweise

- Exemplarisch wird die Syntax der Lambdas in Java 8 vorgestellt.
- Funktionstypen
- Methodenreferenzen
- Datenströme (Streams)

Beispiel 1: Ausgabe einer Liste

```
public class LambdaTest {  
  
    public static void main(String[] args) {  
        LinkedList<Integer> values = new LinkedList<Integer>();  
        values.addAll(Arrays.asList(42,33,7,88,23));  
        values.forEach(x -> System.out.println(x));  
    }  
}
```

- `x -> System.out.println(x)` ist ein Lambda mit einem Parameter und einer Anweisung.
 - Der Typ des Parameters ergibt sich aus dem Kontext. Da `values` eine Liste mit Integer-Objekten ist und `forEach` den Lambda für jedes Element aufruft, muss `x` vom Typ Integer sein.
- Bei einzeiligen Lambdas mit Rückgabetyp kann auch auf die Angabe des Schlüsselwort `return` verzichtet werden.
 - Das Ergebnis des Ausdrucks ist gleich dem Ergebnis der Methode.

Beispiel 2: Rumpf mit mehreren Anweisungen

```
public class LambdaTest {  
  
    public static void main(String[] args) {  
        LinkedList<Integer> values = new LinkedList<Integer>();  
        values.addAll(Arrays.asList(42,33,7,88,23));  
        values.forEach(x -> {  
            x += 7;  
            System.out.println(x);  
        });  
    }  
}
```

- Hierbei werden die Klammern benötigt, um aus mehreren Anweisungen einen Block zu erzeugen.

Beispiel 3: Lambda mit lokalen Variablen

```
public class LambdaTest {  
  
    public static void main(String[] args) {  
        LinkedList<Integer> values = new LinkedList<Integer>();  
        values.addAll(Arrays.asList(42,33,7,88,23));  
        values.forEach(x -> {  
            int y = x+7;  
            System.out.println(y);  
        });  
    }  
}
```

- Man kann wie üblich lokale Variablen im Rumpf eines Lambdas deklarieren.

Beispiel 4: Lambda mit expliziter Typangabe

```
public class LambdaTest {  
  
    public static void main(String[] args) {  
        LinkedList<Integer> values = new LinkedList<Integer>();  
        values.addAll(Arrays.asList(42,33,7,88,23));  
        values.forEach( Integer x ) -> {  
            int y = x+5;  
            System.out.println(y);  
        } ;  
    }  
}
```

- Bei Bedarf kann man auch explizit den Typ zu dem Parameter hinzufügen.
 - Hierzu muss aber der Parameter geklammert werden.

Lambda und Funktionstypen

- Lambdas besitzen einen **Typ**, der über eine funktionale Schnittstelle beschrieben ist.
 - z. B. `Consumer<T>` mit der abstrakten Methode `void accept(T)`.
- Ein Lambda mit einem Parameter kann man z. B. an eine **Variable** vom Typ `Consumer` zuweisen.
 - Die Variable repräsentiert dann diese Funktion.
 - Damit kann diese Funktion als Wert z.B. an eine Methode übergeben werden.
 - Die Funktion kann über den Methodennamen der funktionalen Schnittstelle angesprochen werden.

```
public class LambdaTest {  
  
    public static void main(String[] args) {  
        LinkedList<Integer> values = new LinkedList<Integer>();  
        values.addAll(Arrays.asList(42,33,7,88,23));  
        Consumer ic = x -> System.out.println(x);  
        values.forEach(ic);  
    }  
}
```

Lambda als Rückgabewert

- Man kann jetzt auch **Lambdas** in einer Methode erzeugen und diese dann **als Rückgabewert** liefern.
 - Der Typ des Rückgabewerts muss einer passenden funktionalen Schnittstelle entsprechen.

```
static Consumer<Integer> getConsumer() {  
    return x -> System.out.println(x);  
}  
  
public static void main(String[] args) {  
    LinkedList<Integer> values = new LinkedList<Integer>();  
    values.addAll(Arrays.asList(42, 33, 7, 88, 23));  
    values.forEach(getConsumer());  
}
```

13.2 Methodenreferenzen

- **Motivation**

- Eine vorhandene Methode (und ein Konstruktor) sollte statt eines Lambdas genutzt werden.
 - Ein Umweg über Lambdas sollte hier vermieden werden, wenn die Funktionalität bereits vorhanden ist.
 - Jedoch muss die Methode bzw. der Konstruktor zu der abstrakten Methode des Lambdas (funktionalen Interface) passen.

- **Definition**

- Eine Methodenreferenz ist ein Verweis auf eine Methode.
- Syntax
 - Methodenreferenzen verwenden den Methodennamen (bzw. bei Konstruktoren das Schlüsselwort new).
 - Davor steht noch :: und der Klassename bzw. die Objektreferenz.

Beispiel

- Die Signatur von `PrintStream.println(Object)` ist kompatibel zu `Consumer<T>.accept(T)`:

```
public class MusicianPrinter implements Consumer<Musician> {  
    public static void main(String[] args) {  
        List<Musician> ps = Arrays.asList(...);  
        ps.forEach( System.out::println );  
    }  
}
```

- Gibt alle Elemente der Liste auf der Konsole aus.

Methodenreferenzen (im Detail)

- Folgende vier Fälle werden bei Methodenreferenzen unterschieden:

Fall	Art der Methode	Syntax	Beispiel
1	Statische Methode	ClassName::StaticMethodName	String::valueOf
2	Konstruktor	ClassName::new	ArrayList::new
3	Objektmethode via Objekt	objectReference::MethodName	x::toString
4	Objektmethode via Klasse	ClassName::MethodName	Object::toString

Ziel-Objekt wird dann als erstes Argument übergeben.

Statische Methodenreferenzen

- Syntax der Referenzangabe
 - `ClassName::StaticMethodName`
- Beispiel

```
class A {  
    static void printInt(int x){  
        System.out.println(x);  
    }  
}  
  
public class TestMethodReferences {  
    public static void main(String[] args) {  
        LinkedList<Integer> values = new LinkedList<>();  
        values.addAll(Arrays.asList(1,2,3,4,5));  
        // Fall 1:  
        Consumer<Integer> c = A::printInt; // c: Integer --> ()  
        values.forEach(c);  
    }  
}
```

Konstruktoren

- Syntax der Referenzangabe
 - `ClassName::new`
- Beispiel

```
class A {  
    private int i;  
    public A(int x) { i = x; }  
    public String toString() { return "Klasse A: " + i; }  
}  
  
public class TestMethodReferences {  
    public static void main(String[] args) {  
        LinkedList<Integer> values = new LinkedList<>();  
        values.addAll(Arrays.asList(1,2,3,4,5));  
        // Fall 2:  
        Function<Integer, A> f = A::new; // f: Integer --> A  
        System.out.println("Ausgabe " + f.apply(42));  
    }  
}
```

Objektmethoden via Objekt

- Syntax der Referenzangabe
 - ObjectReference::MethodName
- Beispiel

```
class A {  
    private int i;  
    public A(int x) { i = x; }  
    public String toString() { return "Klasse A: " + i; }  
}  
  
public class TestMethodReferences {  
    public static void main(String[] args) {  
        LinkedList<Integer> values = new LinkedList<>();  
        values.addAll(Arrays.asList(1,2,3,4,5));  
        // Fall 3:  
        A a = new A(7);  
        Supplier<String> s = a::toString;    // () --> String  
        System.out.println("Ausgabe " + s.get());  
    }  
}
```

Objektmethode via Klasse

- Syntax der Referenzangabe
 - `ClassName::MethodName`
- Beispiel

```
class A {  
    private int i;  
    public A(int x) { i = x; }  
    public String toString() { return "Klasse A: " + i; }  
}  
  
public class TestMethodReferences {  
    public static void main(String[] args) {  
        LinkedList<Integer> values = new LinkedList<>();  
        values.addAll(Arrays.asList(1,2,3,4,5));  
        A a = new A(8);  
        // Fall 4:  
        Function<A, String> g = A::toString; // A --> String  
        System.out.println("Ausgabe " + g.apply(a));  
    }  
}
```

13.3 Datenströme (java.util.stream)

- Seit Java 8 gibt es das neue Paket `java.util.stream` im JDK
 - Das Paket enthält Funktionalität zur Verarbeitung eines Stroms von Werten (Objekten).
 - Damit wird es einfacher Lambdas für die Programmierung zu nutzen.

Operatoren auf Datenströmen

- Unterteilung in drei Klassen
 - Erzeugende Datenstromoperationen (**Generatoren**)
 - Dadurch können Container-Objekte (und andere), wie z. B. Listen, als Datenstrom zur Verfügung gestellt werden.
 - Transformationsoperationen (**Transformatoren**)
 - Operationen, die einen Datenstrom in einen anderen überführen.
 - Die Ausführung eines Transformators wird solange nicht begonnen bis dies von einer nachfolgenden Terminaloperation nicht explizit eingefordert wird.
 - **Terminaloperatoren**
 - Konsumieren des Datenstroms und Erzeugung eines Ergebnisses.

Generatoren

- Collections

```
List<Integer> li = new LinkedList<>();  
li.add(1);  
li.add(2);  
li.add(3);  
Stream<Integer> s = li.stream();
```

- Streams

```
Stream<Integer> s = Stream.of(1, 2, 3);
```

Generatoren

- Supplier

```
class NaturalNumbers implements Supplier<Integer> {  
    Integer next = 1;  
    @Override  
    public Integer get() {  
        return next++;  
    }  
}
```

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());
```

Warum hier keine Verwendung eines Lambda-Ausdrucks?

Wir müssen uns die letzte Zahl merken und Lambdas haben keinen Zustand!

Generatoren

- Supplier, Beispiel mit Methodenreferenz

```
Random rand = new Random(System.currentTimeMillis());
Stream<Integer> s2 = Stream.generate(rand::nextInt);
s2.limit(10).forEach(System.out::println);
```

Transformatoren (1)

- Filter
 - Eliminiere alle Elemente, die das gegebene Prädikat **nicht** erfüllen

```
Stream<Integer> s = Stream.of(1,2,3,4)  
    .filter( x -> x%2 == 0 );
```

- Map
 - Überführe Datenstrom von Typ I nach Datenstrom vom Typ O

```
Stream<String> s = Stream.of(3,2,4,1)  
    .map( x -> Integer.toString(x) );
```

Zurück zum Motivierenden Beispiel

- Jetzt können wir den Musiker-Filter implementieren
 - Keine Redundanz
 - Keine unnötigen Klassen
 - Filter Separat

```
List<Musician> mlist = new LinkedList<>();  
...  
mlist.stream()  
    filter(m -> m.getName().equals("Wilson")).  
    forEach(m -> System.out.println(m));
```

Transformatoren (2)

- FlatMap
 - Bilde jedes Objekt im Eingabestrom auf einen Datenstrom ab.
Liefere neuen Datenstrom, der alle Elemente der erzeugten Ströme enthält

```
Stream<List<Integer>> ls = Stream.of(  
    Arrays.asList( 1, 3, 5, 7 ),  
    Arrays.asList( 2, 4, 6, 7 ) );  
Stream<Integer> s = ls.flatMap( l -> l.stream() );
```

- Sorted / Sorted(<Comparator<T>>)
 - Liefert alle Elemente des Stroms sortiert nach ihrer natürlichen/der gegebenen Ordnung

```
Stream<Integer> = Stream.of(3,2,4,1)  
    .sorted();
```

Terminaloperatoren (1)

- `forEach`
 - Führe die gegebene Aktion für jedes Element des Streams aus

```
Stream.of(1,2,3,4)
      .forEach(System.out::println);
```

- `allMatch`
 - Überprüfe ob alle Elemente des Datenstroms das gegebene Prädikat erfüllen

```
boolean all = Stream.of(1,2,3,4)
                     .allMatch( x -> x < 10);
```

Terminaloperatoren (2)

- anyMatch
 - Überprüfe ob mindestens ein Element des Datenstroms das gegebene Prädikat erfüllt.

```
boolean any = Stream.of(1,2,3,4)  
    .anyMatch( x -> x % 2 == 0 );
```

- reduce
 - Reduziert die Elemente des Stroms auf genau ein Ausabeelement.

```
Integer sum = Stream.of(1,2,3,4)  
    .reduce( 0,  
            (acc, value) -> acc+value  
    );
```

Startwert acc

Das Ergebnis wird als
acc an den nächsten
Aufruf des Lamdas
übergeben.

Terminaloperatoren (3)

- collect
 - Reduziert die Elemente des Stroms auf einen veränderbaren (mutable) Akkumulator

```
List<Integer> ls = Stream.of(  
    Arrays.asList( 1, 3, 5, 7 ),  
    Arrays.asList( 2, 4, 6, 7 ) )  
    .flatMap( l -> l.stream() )  
    .collect(Collectors.toList());
```

Parametertyp: Collector
Keine funktionale Schnittstelle!

Bedeutung der Collector-Schnittstelle ist kompliziert und führt für diese Vorlesung zu weit.
Über die Methoden der Klasse Collectors lassen sich aber vorgefertigte Collector-Objekte erzeugen.

Beispiel Pipeline

```
Stream.of(5,4,7,2,10,8)
    .filter( x -> x % 2 == 0 )
    .map( x -> x*x )
    .sorted()
    .forEach( System.out::println );

// Ergebnis: ?
```



Beispiel Pipeline

```
Stream.of(5,4,7,2,10,8)
    .filter( x -> x % 2 == 0 )
    .map( x -> x*x )
    .sorted()
    .forEach( System.out::println );

// Ausgabe:
// 4
// 16
// 64
// 100
```

Lazy Evaluation (1)

- Auswertung wird ausschließlich durch Terminaloperatoren angestoßen:

```
Stream.of(5, 4, 7, 2, 10, 8)
    .filter( x -> {
        System.out.println("Filter: " + x);
        return x % 2 == 0;
    });

```

- Liefert keine Ausgabe, da kein Terminaloperator benutzt wird.

Lazy Evaluation (2)

- Auswertung wird ausschließlich durch Terminaloperatoren angestoßen:

```
Stream.of(5, 4, 7, 2, 10, 8)
    .filter( x -> { System.out.println("Filter: " + x);
                      return x % 2 == 0; })
    .forEach(
        x -> System.out.println("ForEach: " + x)
    );

// Ausgabe:
// Filter: 5
// Filter: 4
// ForEach: 4
// Filter: 7
// Filter: 2
// ForEach: 2
...
```

Lazy Evaluation (3)

- Operationen werden vertikal ausgeführt
 - Ein Element durchläuft alle Operationen der Pipeline, bevor das nächste ausgewertet wird
 - Es werden nur so viele Elemente betrachtet, wie zur Berechnung des Ergebnisses nötig sind
 - Reduziert ggf. die Anzahl der benötigten Operationen

```
Stream.of(5, 4, 7, 2, 10, 8)
    .map( x -> { System.out.println("Map: " + x);
                  return x*x; })
    .anyMatch( x -> { System.out.println("AnyMatch: " + x);
                      return x%2 == 0; });
// Ausgabe:
// Map: 5
// AnyMatch: 25
// Map: 4
// AnyMatch: 16 -- DONE
```

Ausnahme: Stateful Transformations

- Manche Transformatoren benötigen einen Status
 - Sortierung ist erst möglich, wenn alle Elemente bekannt sind.

```
Stream.of(5, 4, 7, 2, 10, 8)
    .sorted( (x, y) -> {
        System.out.println("Sort: " + x + ";" + y);
        return x.compareTo(y); } )
    .filter( x -> {
        System.out.println("Filter: " + x);
        return x % 2 == 0; } )
    .forEach(
        x -> System.out.println("ForEach: " + x)
    );
```

Ausgabe:

Sort: 4;5

Sort: 7;4

Sort: 7;5

Sort: 2;5

Sort: 2;4

Sort: 10;5

Sort: 10;7

Sort: 8;5

Sort: 8;10

Sort: 8;7

Filter: 2

ForEach: 2

Filter: 4

ForEach: 4

Filter: 5

Filter: 7

Filter: 8

ForEach: 8

Filter: 10

ForEach: 10

Order Matters

- Reduziere Anzahl der Operationen durch Umsortieren

```
Stream.of(5, 4, 7, 2, 10, 8)
    .filter( x -> {
        System.out.println("Filter: " + x);
        return x % 2 == 0; } )
    .sorted( (x, y) -> {
        System.out.println("Sort: " + x + ";" + y);
        return x.compareTo(y); } )
    .forEach(
        x -> System.out.println("ForEach: " + x)
    );
```

Eigentliche Ausgabe gleich.

Ausgabe:

Filter: 5
Filter: 4
Filter: 7
Filter: 2
Filter: 10
Filter: 8
Sort: 2;4
Sort: 10;2
Sort: 10;4
Sort: 8;4
Sort: 8;10
ForEach: 2
ForEach: 4
ForEach: 8
ForEach: 10

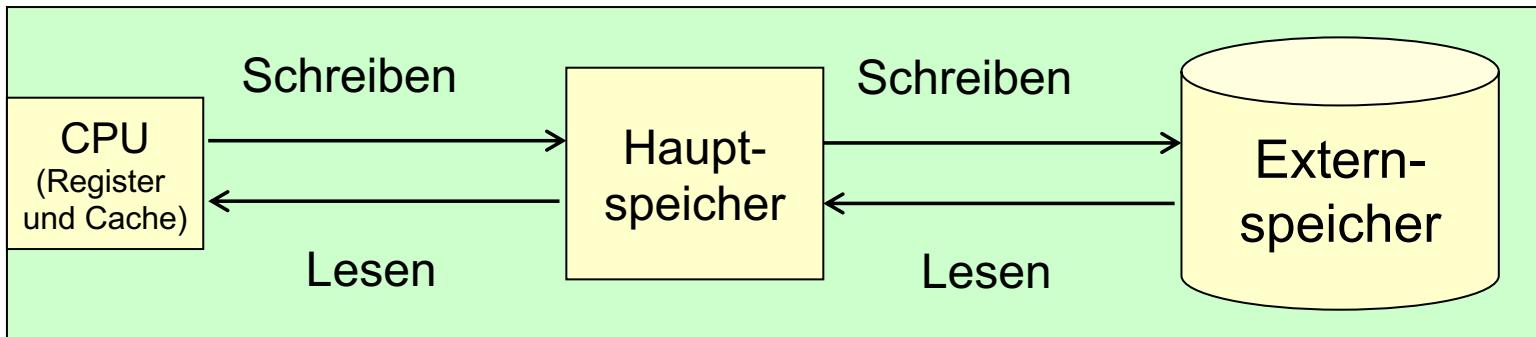
Zusammenfassung

- Innere Klassen
 - Insbesondere lokale (anonyme) Klassen haben zur Erleichterung der Programmierung beigetragen.
 - Klassen werden dort definiert, wo sie verwendet werden.
- Durch Lambdas werden in Java funktionale Programmierkonzepte zur Verfügung gestellt.
 - Vereinfachung der Programmierung
 - weniger Code, um das Gleiche ausdrücken.
 - Verbesserung der Codequalität
 - Typisierung wird zur Zeit der Übersetzung sichergestellt.
- Datenströme
 - Operatoren zur Verarbeitung von großen Daten
 - Anwendung im Bereich Data Science in Verarbeitungssystemen wie Apache Hadoop und Apache Spark

14. Ein- und Ausgabe

- Dateien
- Die Klasse `java.io.File`
- Binärdateien und die Klasse `java.io.RandomAccessFile`
- Datenströme
- Klassenhierarchie

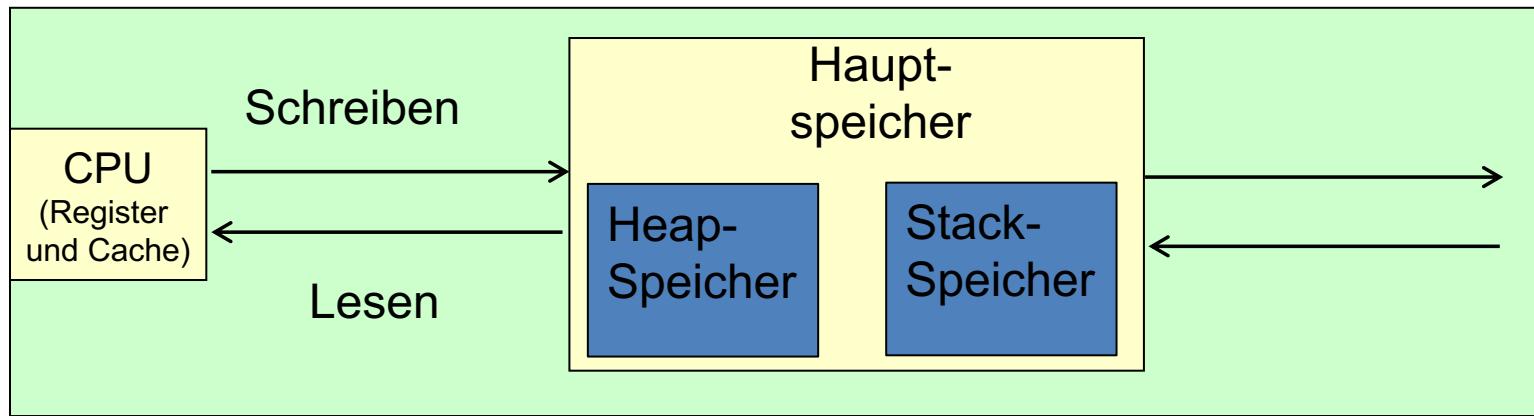
Einfaches Modell eines Computers



- CPU (central processing unit):
 - ist eine physikalische Einheit, die eine Menge von Befehlen verarbeiten kann
 - Hierfür benötigte Daten werden aus dem Hauptspeicher gelesen.
- Hauptspeicher ist ein Speichermedium mit folgenden Eigenschaften:
 - schneller Zugriff
 - relativ teuer
 - **flüchtig**: Verlust des Inhalts beim Abschalten des Computers
- Externspeicher (z. B. Festplatte, CD, Speicherstick,...)
 - langsamer Zugriff
 - relativ billig
 - **stabil**: kein Verlust des Inhalts beim Abschalten

Speicherverwaltung von Java

- Java verwaltet alle Variablen und Objekte im Hauptspeicher eines Computers



- Konsequenz

- Beim **Programmende** wird der vom Programm belegte Speicherplatz im Hauptspeicher **freigegeben**.

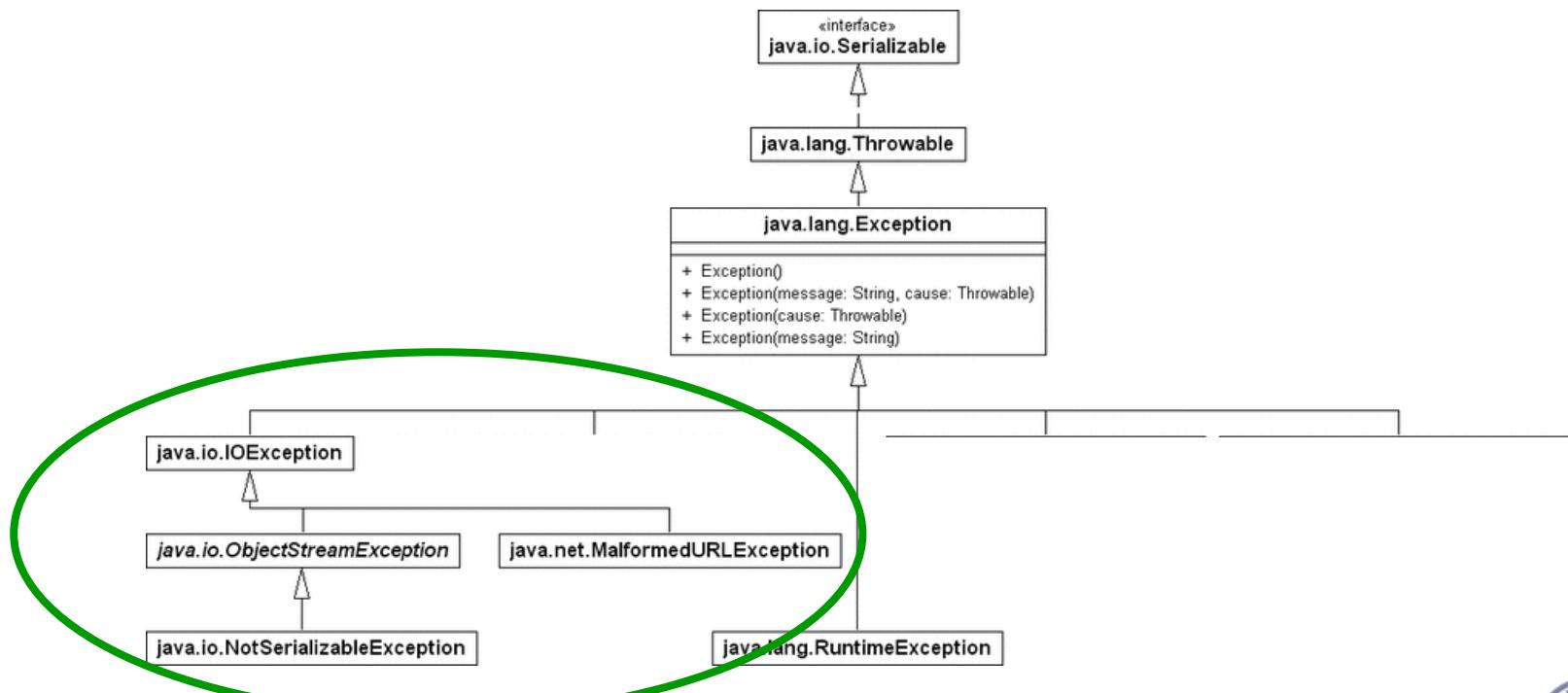
Programmende ⇒ Verlust der Werte aller Variablen und Objekte

Persistierung der Daten

- Wie kann man in Java es erreichen, dass
 - wichtige Daten auch noch nach dem Programmende existieren
 - und beim nächsten Programmlauf die Daten wieder im Programm vorhanden sind?
- Lösung
 - Rettung der Daten durch **Abspeichern auf dem Externspeicher**
 - Externspeicher „**überlebt**“ das Abschalten des Computers
 - Daten auf dem Externspeicher werden nicht nach dem Programmende vernichtet, sondern **bleiben** darüber hinaus **verfügbar**.
 - Einlesen der Daten beim nächsten Programmlauf
- Problem
 - Speicherverwaltung muss selbst programmiert werden.

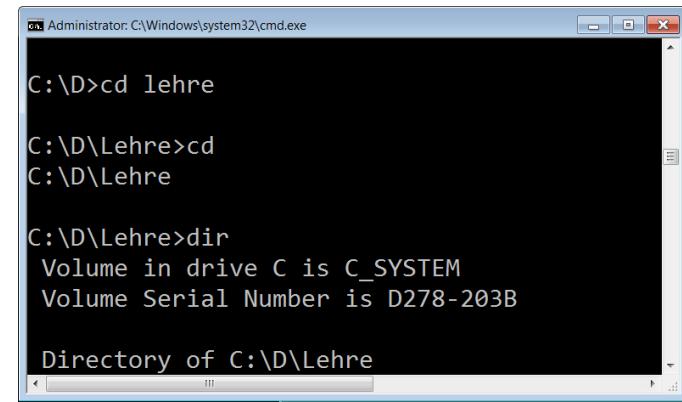
Exceptions für java.io

- Die Oberklasse `IOExceptions` und deren Unterklassen stellen geprüfte Ausnahmen für das Paket `java.io` zur Verfügung.
 - Diese Exceptions **müssen** gefangen oder weitergereicht werden!



14.1 Dateien und die Klasse File

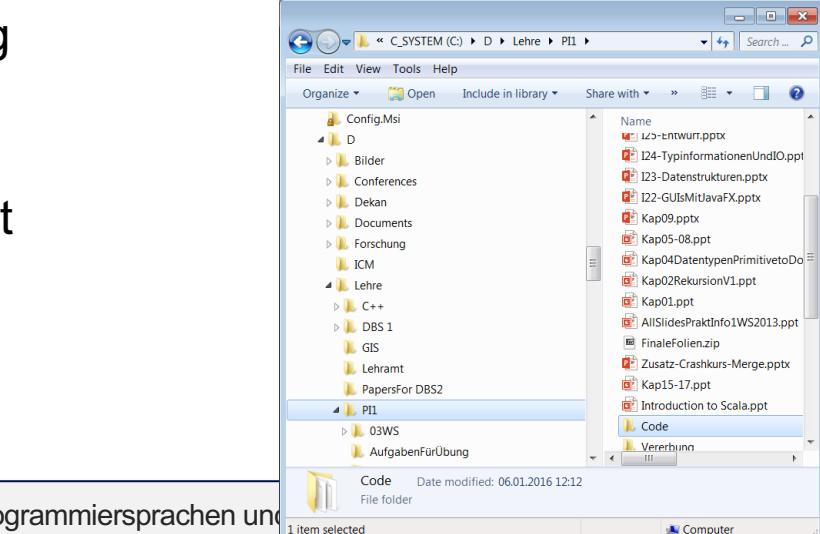
- Daten werden auf dem Externspeicher in **Dateien** gespeichert.
- Ein **Dateisystem** verwaltet die Dateien auf dem Externspeicher.
- Es gibt verschiedene Schnittstellen, um das Dateisystem anzusprechen.
 - Die **Kommandozeile** unter Windows und Linux stellt Befehle zur Verfügung, um mit Dateien zu arbeiten.
 - Eine andere Möglichkeit ist die Benutzung eines **Datei-Explorer**.
 - Die Programmiersprache Java bietet mit der **Klasse File** ebenfalls eine Möglichkeit Befehle des Dateisystems aufzurufen.



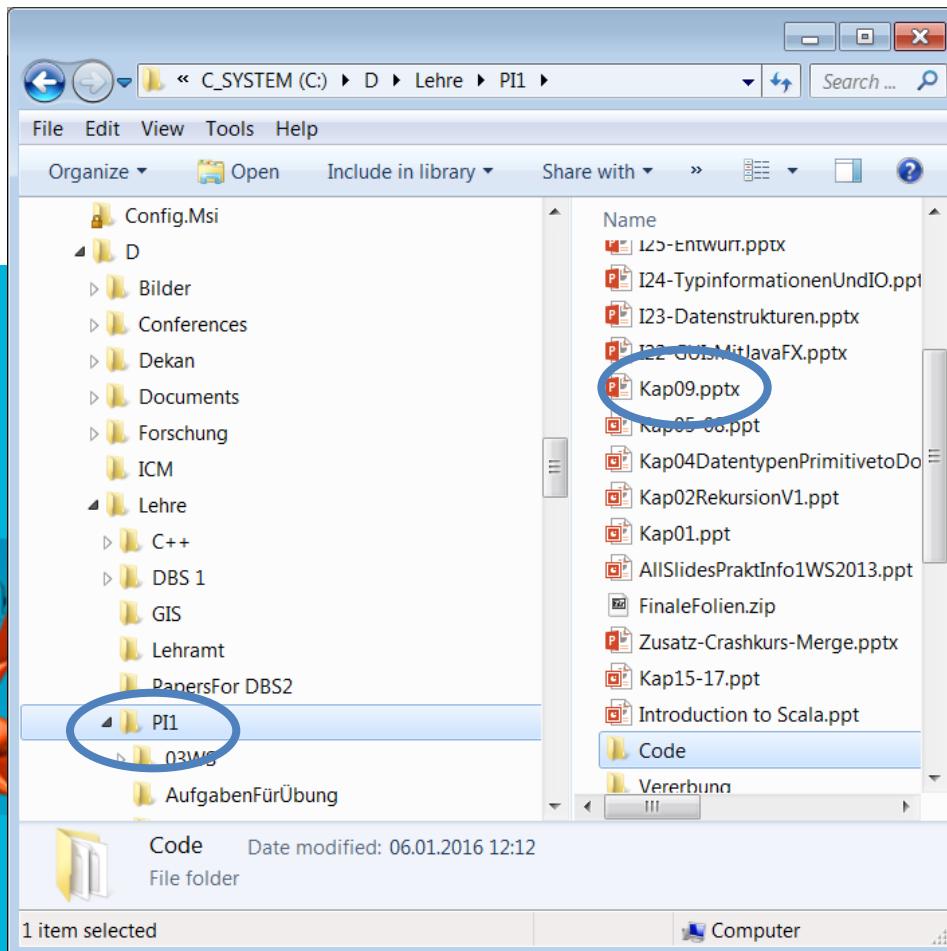
```
C:\D>cd lehre
C:\D\Lehre>cd
C:\D\Lehre

C:\D\Lehre>dir
Volume in drive C is C_SYSTEM
Volume Serial Number is D278-203B

Directory of C:\D\Lehre
```



Organisation des Externspeichers



- Daten auf dem Externspeicher werden in einem Dateisystem verwaltet.
 - Daten liegen in Dateien
 - z. B. Datei Kap09.pptx
 - Verzeichnisse werden genutzt, um Daten logisch aufzuteilen.
 - Verzeichnis PI1
- Eine Datei ist ein Behälter (potentiell beliebiger Größe)
 - Vor dem Gebrauch muss dieser Behälter **geöffnet** werden.
 - Danach kann man aus dem Behälter Daten **lesen** bzw. in den Behälter Daten **schreiben**.
 - Hat man alles erledigt, sollte eine Datei wieder **geschlossen** werden.

Dateinamen

- Der vollständige Dateiname setzt sich aus zwei Teilen zusammen:
 - **lokaler Dateiname**
 - ist eindeutig innerhalb des Verzeichnisses, dem die Datei zugeordnet ist.
 - **Pfadname**
 - Verknüpfung aller Verzeichnisnamen beginnend vom Wurzelverzeichnis bis zu dem Verzeichnis, in dem die Datei liegt.
- Abhängig vom Betriebssystem wird zwischen den einzelnen Teilen eines vollständigen Dateinamens ein **Separatorzeichen** verwendet:
 - DOS: „\“
 - UNIX: „/“
- Der Name des **Wurzelverzeichnisses** („/“) besteht nur aus dem Separatorzeichen **und im Fall von Windows dem Laufwerk**.
- Beispiel:
 - lokaler Dateiname: “Brief.Key”
 - vollständiger Dateiname (Windows): “C:\Cafe\Bin\Keys\Brief.Key”



Die Klasse File im Paket java.io

- File ist eine Klasse zum Zugriff auf einen baumstrukturierten Dateikatalog
 - Die Repräsentation abstrahiert vom zugrundeliegenden Betriebssystem.
- Konstruktoren
 - **public File(String path);**
 - **public File(String path, String name);**
 - **public File(File dir, String name);**
- Konstante
 - **public final static String separator;**
- Eine Auswahl von wichtigen Objektmethoden
 - **public String getPath();** // Resultat: Pfadname
 - **public String getParent();** // Resultat: Namen des Elternverzeichnis
 - **public boolean isDirectory();** // Resultat: true, falls Datei ein Verzeichnis ist.
 - **public long length();** // Resultat: Länge der Datei (in Bytes).
 - **public boolean exists();** // Resultat: true, falls zu diesem Dateinamen tatsächlich eine Datei existiert.
 - **public String[] list();** // Liste der Dateinamen für ein Verzeichnis

Systemabhängige Konstante für das Separatorzeichen als String.

Anwenden der Methoden

```
import java.io.File;

public class FileTest {
    public static void main(String[] args) throws Exception {
        File file;

        file = new File(args[0]);
        System.out.println("file.getPath()") : " + file.getPath());
        System.out.println("file.getCanonicalPath()") : " + file.getCanonicalPath());
        System.out.println("file.getParent()") : " + file.getParent());
        System.out.println("file.isDirectory()") : " + file.isDirectory());
        System.out.println("file.length()") : " + file.length());
        System.out.println("file.exists()") : " + file.exists());
        System.out.println("file.list().length") : " + file.list().length);
        System.out.println("file.list()[0]") : " + file.list()[0]);

    }
}
```

Beispiel

- **Aufgabenstellung**

Implementieren Sie eine Methode, die zu einem vorgegebenen **Namen eines Verzeichnisses** V und einem **Suffix** S, rekursiv alle Namen der Dateien ausgibt, die in V liegen und den Suffix S besitzen.

- Vorschlag für eine Implementierung:

```
import java.io.File;

public class FSearch {
    public static void search (File dir, String suffix) { ... }

    public static void main(String[] arg) throws Exception {
        File start = new File(arg[0]);
        if (start.exists() && start.isDirectory())
            search(start, "." + arg[1]);
        else
            System.out.println("Falsche Eingabe!");
    }
}
```

Der Rumpf der Methode search

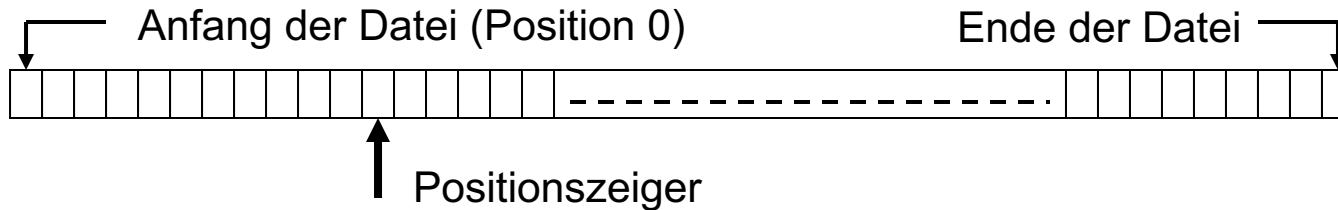
```
public static void search (File dir, String suffix) {  
    File next;  
    String[] list = dir.list();  
    int count = 0;  
  
    for (int i = 0; i < list.length; i++) {  
        if (list[i].endsWith(suffix)) {  
            // endsWith ist eine Methode der Klasse String  
            if (count == 0)  
                System.out.println("Directory: " + dir.getPath());  
            count++;  
            System.out.println(" " + list[i]);  
        }  
    }  
    for (int i = 0; i < list.length; i++) {  
        next = new File(dir, list[i]);  
        if (next.isDirectory())  
            search(next, suffix);  
    }  
}
```

Ausgabe der Dateinamen

rekursiver Aufruf

14.2 Wahlfreier Zugriff in Dateien

- Eine **Datei** ist eine Datenstruktur zur **Speicherung von Daten auf dem Externspeicher**.
 - Eine Datei **entspricht** dabei einer **Reihe von Werten mit dem Datentyp byte**. Zusätzlich gibt es einen **Positionszeiger**.
 - Eine Datei ermöglicht das Lesen an und Schreiben von der **Stelle**, wo der Positionszeiger gerade steht.



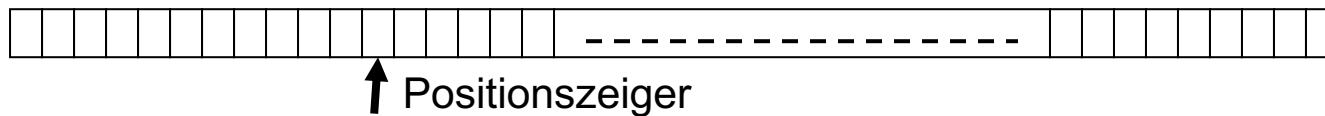
- Eine Datei kann erweitert werden, indem am Ende neue Daten hinzugefügt werden.

Öffnen und Schließen von Dateien

- Bevor eine Operation ausgeführt wird, muss eine Datei unter Angabe eines Namens geöffnet werden.
 - Hierbei werden Hilfsstrukturen im Hauptspeicher aufgebaut.
- Wird eine Datei nicht mehr benötigt, sollte man die Datei explizit schließen.
 - Hilfsstrukturen werden abgebaut und Speicherplatz freigegeben.
 - Danach muss die Datei wieder geöffnet werden, um wieder Operationen auf der Datei auszuführen.

Schreiben in eine Datei

- Der **Positionszeiger** muss zunächst auf die Stelle **gesetzt** werden, wo Daten geschrieben werden sollen.



- Nach dem Schreiben zeigt der Positionszeiger auf die Stelle in der Datei, die der zuletzt geschriebenen Stelle folgt.



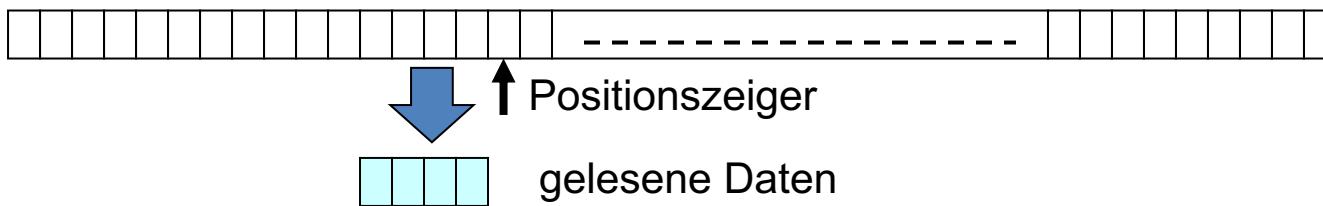
- Wird beim **Schreiben** das Ende der Datei überschritten, so wird die Datei dynamisch um eine entsprechend große Anzahl von Bytes **verlängert**.

Lesen aus einer Datei

- Der **Positionszeiger** muss zunächst auf die Stelle **gesetzt** werden, wo Daten geschrieben/gelesen werden sollen.



- Danach kann der Inhalt in den Hauptspeicher übertragen werden.
 - Der Positionszeiger steht hinter der zuletzt gelesenen Stelle.



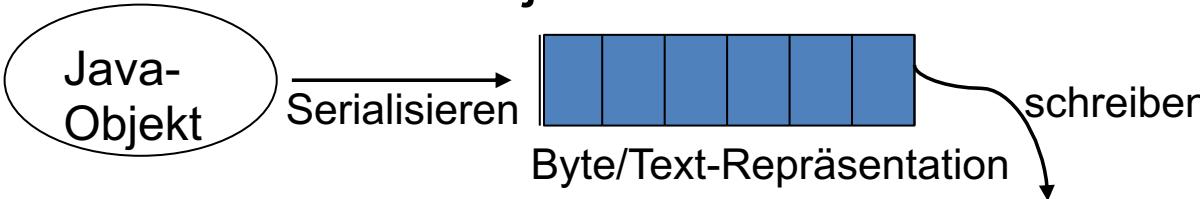
- Beim Lesen darf das Ende der Datei nicht überschritten werden.
 - Ansonsten wird eine Ausnahme ausgelöst.

Binär- und Textdateien

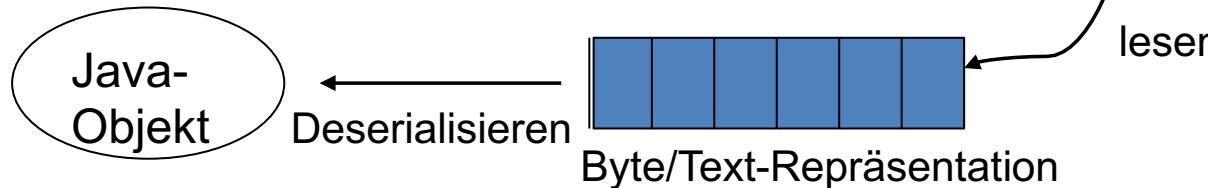
- Dateizugriff wird unterschieden, in welcher Form die Daten geschrieben/gelesen werden:
 - Daten können direkt in ihrem **Binärformat** in eine Datei geschrieben werden. Solche Dateien werden auch als **Binärdateien** bezeichnet
 - Werden Daten zuerst in einen **String** transformiert und wird dann der String in eine Datei geschrieben, so spricht man von einer **Textdatei**.

Operationen auf Dateien

- Schreiben eines Objekts in eine Datei



- Lesen aus einer Datei



- Bei einer Textdatei wird eine Serialisierung in Text und bei einer Binärdatei in Byte vorgenommen

Die Klasse RandomAccessFile

- Die Klasse stellt Dateien zur Verfügung, die den wahlfreien Zugriff auf Dateien unterstützen.
 - Zugriffsmethoden und Datenfelder zur Manipulation des Positionszeigers

Konstruktoren

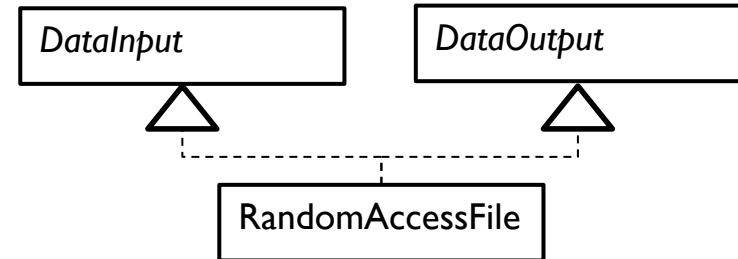
- **public RandomAccessFile(String name, String mode);**
 - Der Parameter *mode* zeigt an, ob auf die Datei nur *lesend* (*mode == "r"*) oder auch *schreibend* zugegriffen werden kann (*mode == "rw"*).
 - Der Parameter *name* enthält den *Dateinamen* (relativ zum Verzeichnis, wo das Programm gestartet wird oder absolut)
 - Nach einem Aufruf eines Konstruktors ist die *Datei geöffnet*, d.h. Operationen können ausgeführt werden. Der *Positionszeiger* steht *am Anfang* der Datei.
- **public RandomAccessFile(File handle, String mode);**
 - In diesem Konstruktor wird statt einem String eine Referenz eines Objekts der Klasse File übergeben.

Methoden zum Positionieren

- Lesen der Position des Positionszeigers
 - **public long** getFilePointer();
// Resultat: **aktueller** Wert des **Positionszeiger**
- Setzen der Position des Positionszeigers
 - **public void** seek(long pos);
// Resultat: Wert des Positionszeiger wird auf pos gesetzt.
- Abfrage nach der Länge der Datei (in Bytes)
 - **public long** length();

Gemessen in
Bytes.

Wichtige Schnittstellen



- Schnittstellen `DataOutput` und `DataInput`
 - Umwandlung von **primitiven Datentypen in eine Folge von Bytes** und umgekehrt.
 - Beispiele von Methoden
 - `DataOutput`
`void writeDouble(double v) throws IOException`
 - `DataInput`
`double readDouble() throws IOException`

Methoden zum Lesen

- Implementierung der Schnittstelle DataInput
- Methoden zum **Lesen** von Daten
 - Schnittstelle DataInput:
 - **boolean** readBoolean()
 - **double** readDouble()
 - **int** read(**byte[]** b, **int** off, **int** len)
 - Methode **liest** bis zu **len Bytes aus der Datei und schreibt diese** in das Array b an die Position off.
 - Resultat: Anzahl der tatsächlich gelesenen Bytes.
 - ...

Methoden zum Schreiben

- Implementierung der Schnittstelle DataOutput
- Methoden zum **Schreiben** von Daten
 - Schnittstelle DataOutput:
 - **void writeBoolean(boolean v)**
 - **void writeDouble(double d)**
 - **void write(byte[] b, int off, int len)**
 - Methode **schreibt** ab der Position off bis zu **len Bytes aus dem Array b** in die Datei.
 - ...

Beispiel

- Erweiterung der Klasse Point

```
import java.io.RandomAccessFile;

class Point {
    ...
    public void writeToFile(RandomAccessFile raf, int pointPos)
        throws IOException {
        raf.seek(pointPos*16);
        raf.writeDouble(x);
        raf.writeDouble(y);
    }

    public Point(RandomAccessFile raf, int pointPos)
        throws IOException {
        raf.seek(pointPos*16);
        x = raf.readDouble();
        y = raf.readDouble();
    }
    ...
}
```

Der wievielte „Point“ in der Datei.

seek und writeDouble können eine IOException werfen.

Ein „Point“ besteht aus zwei doubles. Je double werden 8 Byte benötigt.

Verwendung der Methoden

```
Point[] parr = new Point[MAX], pcopy = new Point[MAX];  
  
// ...  
File f = new File("Datei.pi1");  
  
RandomAccessFile raf;  
try {  
    raf = new RandomAccessFile(f, "rw");  
} catch (IOException e) {  
    System.out.println("Die Datei konnte nicht geöffnet werden"); return -1;  
}  
  
// Schreiben in die Datei  
try {  
    for (int i = 0; i < MAX; i++)  
        parr[i].writeToFile(raf, i);  
} catch (IOException e) {  
    System.out.println("Nicht alle Punkte konnten serialisiert werden"); return -1; }  
  
// Einlesen der Daten aus der Datei  
try {  
    for (int i = 0; i < MAX; i++)  
        pcopy[i] = new Point(raf, i);  
} catch (IOException e) {  
    System.out.println("Nicht alle Punkte konnten deserialisiert werden."); return -1; }  
return 0;
```

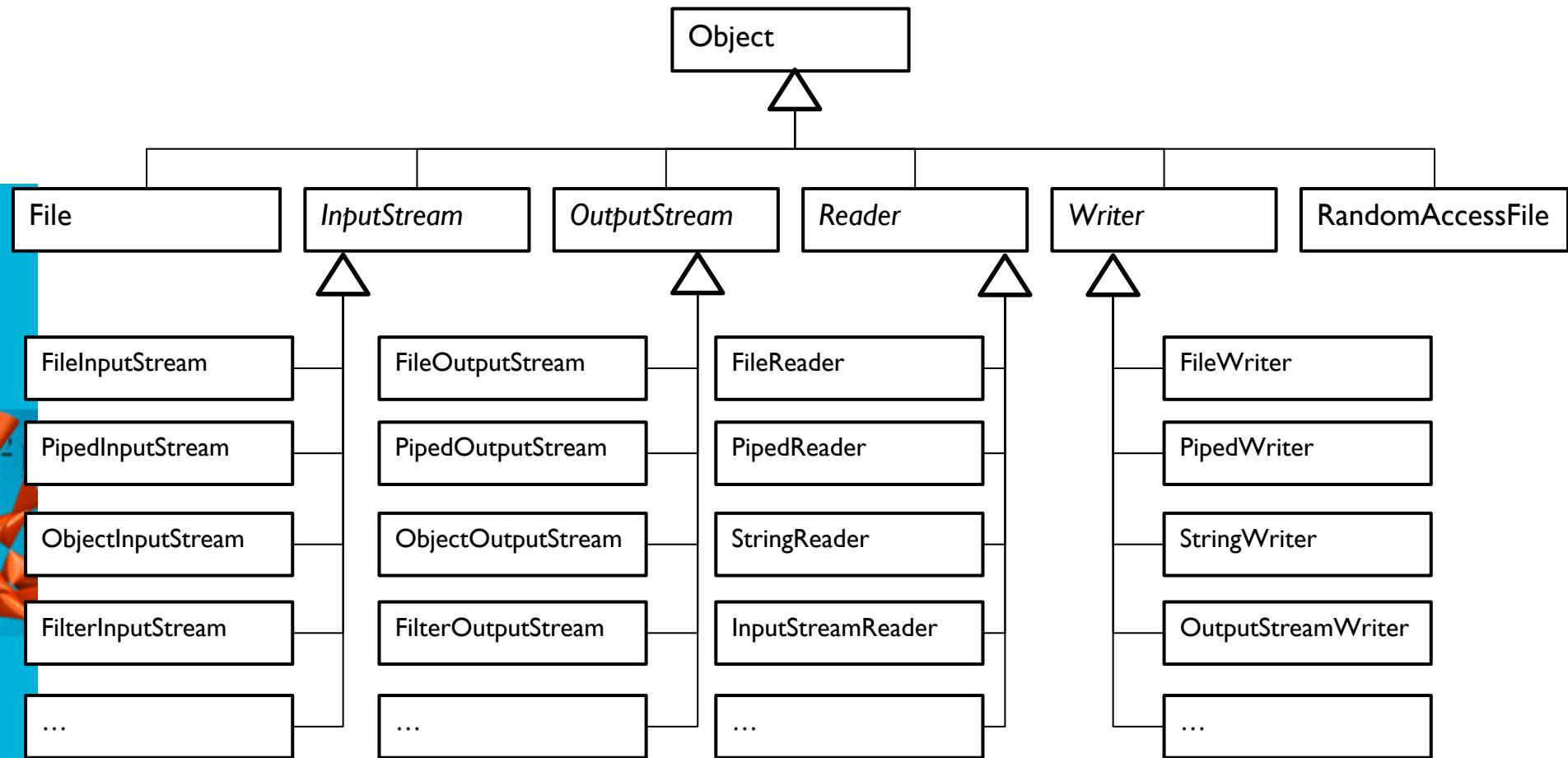
Diskussion

- Die Klasse RandomAccessFile ermöglicht den **wahlfreien Zugriff** auf Dateien.
 - Position in der Datei kann beliebig gesetzt werden.
 - Daten werden dann im Binärformat in die Datei geschrieben.
- Hoher Aufwand bei der Programmierung
- Hohe Fehleranfälligkeit
 - Auffangen von Exceptions
- Grundlegende Klasse zur Implementierung von
 - Datenbanksystemen
 - Indexstrukturen

14.3 Datenströme (java.io)

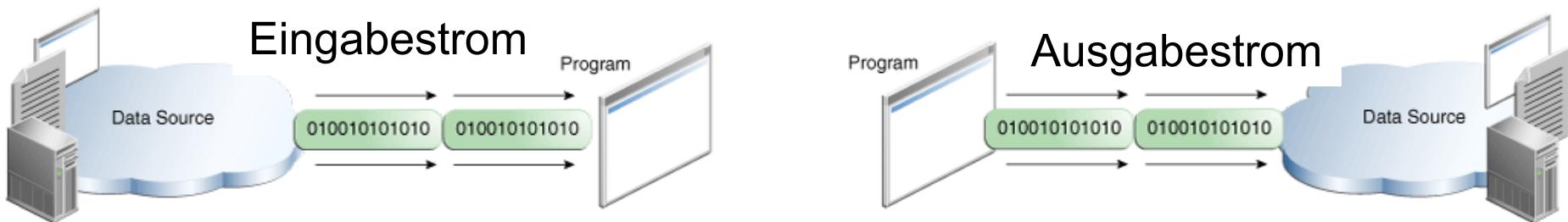
- Grundlegende Funktionalität für die **Verwaltung von Dateien** in einem Dateisystem
 - Klasse File
- Zugriff auf den **Inhalt der Dateien**: Unterscheidung zwischen strombasierten Zugriff und wahlfreien Zugriff auf Daten
 - **Strombasierter Zugriff**
 - Sequentielles Lesen aus einer Datei
 - Beginnend von vorne nach hinten
 - Sequentielles Schreiben in eine Datei
 - Typischerweise anhängen neuer Daten ans Ende der Datei
 - **Wahlfreier Zugriff**
 - Lesender und schreibender Zugriff auf beliebige Positionen in einer Datei

Übersicht zu den Klassen



Datenströme

- Das Paket `java.io` bietet mit dem Konzept des Datenstroms mehr als die reine Ein- bzw. Ausgabe (I/O) in Dateien.
- Datenströme sind grundlegend für die
 - Mensch-Maschine Kommunikation
 - Maschine-Maschine Kommunikation in einem Netzwerk
- Unterscheidung zwischen Ein- und Ausgabestrom



Datenströme (java.io)

- Datenströme bieten einen **sequentiellen Zugriff** auf Dateien an.
 - Alle Klassen zu Datenströmen befinden sich im Paket **java.io**.
- **Stream = abstraktes Konstrukt** mit der Fähigkeit, Zeichen auf ein imaginäres Ausgabegerät zu schreiben oder von diesem zu lesen.
 - Dieses imaginäre Gerät abstrahiert von den „echten“ Geräten (Bildschirm, Tastatur, Datei, String, Kommunikationskanal).
 - **Unterklassen binden** die Zugriffs Routinen **an** echte Ein- oder Ausgabegeräte.
- Streams können **verkettet** oder **verschachtelt** werden.
 - **Verkettung** erlaubt z.B. mehrere Dateien zusammenzufassen und als einen einzigen Stream darzustellen.
 - **Verschachteln** erlaubt die Implementierung von Zusatzfunktionen wie das Puffern von Zeichen.

Datenströme (java.io)

- Datenströme bieten einen **sequentiellen Zugriff** auf Daten.
- Alle Klassen zu Datenströmen befinden sich im Paket `java.io`.
- **Stream = abstraktes Konzept**: Ein Stream ist ein imaginäres Ausgabegerät zu schreiben.
- Dieses imaginäre Gerät kann eine Datei, den Bildschirm, Tastatur, oder Ausgabegeräte.
- Unter **Unterstromen** versteht man Stream, die mehrere Dateien zusammenfassen und als einen einzigen Stream darstellen.
- **Verschachtelung**: Ein Stream kann die Implementierung von Zusatzfunktionen wie das Puffern von Zeichen.

Achtung:
nicht verwechseln mit
`java.util.stream`

Byte- und Character-Streams

- In Java wird unterschieden zwischen
 - Byte-Streams
 - Diese Streams benutzen ein Byte als Einheit.
 - Character-Streams.
 - Diese verwenden grundsätzlich 16 Bit lange Unicode-Zeichen und arbeiten daher besser mit den String- und Zeichentypen von Java zusammen.
 - Brückenklassen erlauben eine Überführung von Character-Streams in Byte-Streams und umgekehrt.
- Weiterhin wird bei Streams noch unterschieden zwischen
 - Streams, auf die geschrieben werden kann.
 - Streams, von denen gelesen werden kann.

Übersicht der Klassen

	Byte-Streams	Character-Streams
lesenden Zugriff	<i>InputStream</i>	<i>Reader</i>
schreibenden Zugriff	<i>OutputStream</i>	<i>Writer</i>

- Die *Basisklassen für Byte-Streams sind *InputStream* und *OutputStream*.*
 - Diese sind jeweils Wurzel einer Hierarchie von Klassen.
- *Entsprechend beginnt für Character-Streams die Hierarchie bei den Klassen *Reader* und *Writer*.*
- *Beide Klassenhierarchien befinden sich im Paket *java.io*.*

14.3.1 Ausgabe Byte-Streams

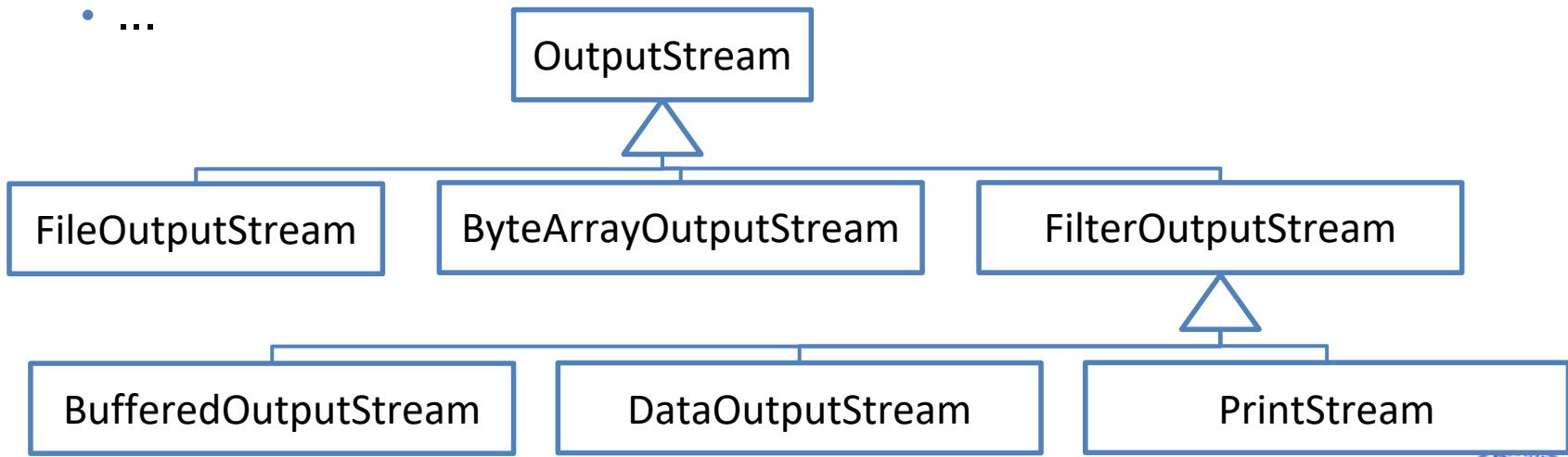
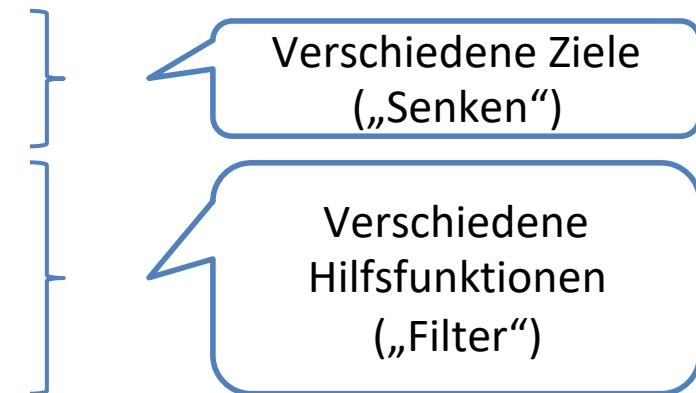
- Basis der Ausgabe-Streams ist die abstrakte Klasse **OutputStream**
- Sie stellt folgende **Methoden** zur Verfügung:
 - abstract public void write(int b)
 - Gibt lediglich die unteren 8 Bit aus und ignoriert alle übrigen.
 - Dies ist die einzige abstrakte Methode in der Klasse.
 - public void write(byte[] b)
 - Schreibt das Array in den Stream.
 - public void write(byte[] b, int offs, int len)
 - Schreibt den Teil des Arrays zwischen offs und offs+len-1 in den Stream.
 - public void flush()
 - Schreibt die gepufferten Daten auf das Ausgabegerät und leert alle Puffer.
 - public void close()
 - Schließt den Stream (und leert den Puffer). Danach sind keine weiteren Operationen erlaubt.
- All diese Methoden können eine IO Exception werfen

So kann das Argument als ein nicht-Vorzeichen-behafteter Wert angesehen werden.

Verschachtelung von Ausgabeströmen

- Was gibt es für Unterschiede zwischen Ausgabeströmen?

- Schreiben in Datei
- Schreiben in Byte-Array
- Schreiben von rohen Byte-Folgen
- Schreiben von primitiven Datentypen
- Pufferung der Daten
- ...



Verschachtelung von Ausgabeströmen

- Senken Ausgabeströme
 - Können eigenständig erzeugt und verwendet werden
 - Konstruktoren benötigen nur Angabe über Ziel des Bytestroms (z.B. Dateiname)
- Filter Ausgabeströme
 - Benötigen einen anderen Ausgabestrom, dem Hilfsfunktionalität hinzugefügt werden soll
 - Konstruktoren benötigen diesen Ausgabestrom
 - Alle Schreiboperationen werden dann intern an diesen Stream delegiert.
→ **Typisches Muster der Programmierung**
 - Schachtelung kann beliebig lang sein



Beispiele

ByteArrayOutputStream kann direkt erzeugt werden.

```
public static void main(String[] args) throws IOException {  
    ByteArrayOutputStream byteOS = new ByteArrayOutputStream();  
    BufferedOutputStream bufferedOS = new BufferedOutputStream(byteOS, 16);  
    DataOutputStream dataOS = new DataOutputStream(bufferedOS);  
}
```

dataOS: write-Methoden für Standard-Datentypen

bufferedOS:
Zwischenspeicherung von Daten

byteOS: Speicherung des Datenstroms in byte[]

Beispiele

ByteArrayOutputStream kann direkt erzeugt werden.

```
public static void main(String[] args) throws IOException {  
    ByteArrayOutputStream byteOS = new ByteArrayOutputStream();  
    BufferedOutputStream bufferedOS = new BufferedOutputStream(byteOS, 16);  
    DataOutputStream dataOS = new DataOutputStream(bufferedOS);  
  
    dataOS.writeInt(4);  
    dataOS.writeInt(3);  
    dataOS.writeInt(2);  
    dataOS.writeInt(1);  
    System.out.println(byteOS.toByteArray().length);  
    dataOS.writeInt(0);  
    System.out.println(byteOS.toByteArray().length);  
}
```

writeInt-Methode wird durch DataOutputStream hinzugefügt.

Ergibt „0“, da durch den BufferedOutputStream Daten noch zwischengespeichert werden.

Ergibt „16“, da der Puffer inzwischen voll war und die Daten (Bytes) an byteOS durchgeschrieben wurden.

FileOutputStream

- Erweiterung von OutputStream
 - Schreiben der Daten **in eine Datei**
- **Konstruktoren**
 - public FileOutputStream(String name) throws FileNotFoundException
 - public FileOutputStream(String name, boolean append) throws FileNotFoundException
 - public FileOutputStream() throws IOException
- **FileNotFoundException** wird „geworfen“, wenn
 - die Datei nicht existiert
 - die Datei existiert, aber die Datei ist ein Verzeichnis.
 - die Datei existiert, aber nicht erzeugt oder geöffnet werden kann.

Beispiel: FileOutputStream

```
import java.io.FileOutputStream;
import java.io.IOException;

public class FileOutputStreamTest {
    public static void main(String[] args) throws IOException {
        FileOutputStream out = null;
        try {
            out = new FileOutputStream(args[0], true);
            for (int i = 0; i < 256; ++i)
                out.write(i);
        }
        catch (IOException e) {
            e.printStackTrace();
            System.exit(1);      // Beendet das Programm
        }
        finally {
            out.close();          // Schließen der Datei als letzte Aktion
        }
    }
}
```

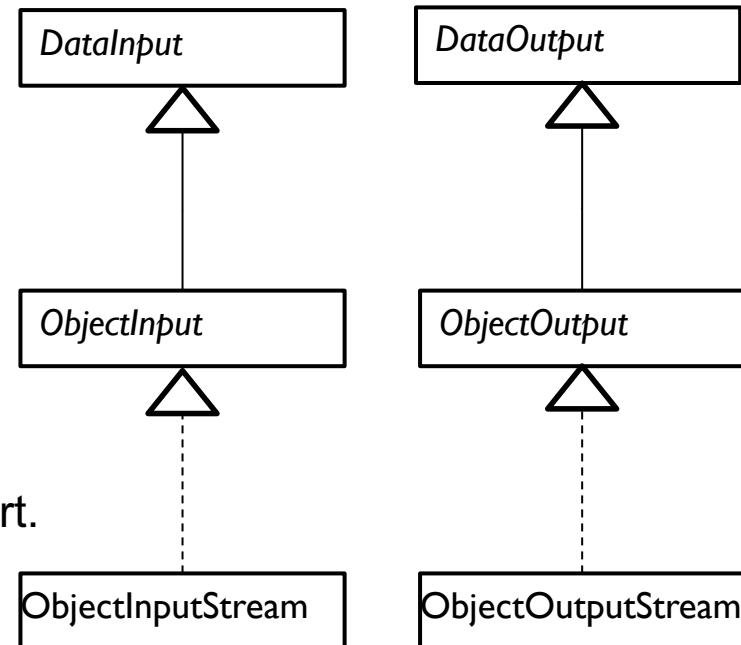
Dateiname

also
anhängen!

ObjectOutputStream

Besonderheiten der Klasse

- primitive Datentypen und komplette Objekte (inklusive aller referenzierten Objekte) können binär ausgegeben werden.
 - Die Objekte müssen jedoch die Schnittstelle **Serializable** implementieren.
 - Beispiel für einen **FilterOutputStream**
 - Die Schnittstelle **DataOutput** wird implementiert.
 - ObjectOutputStream ist die Basis für die **Serialisierung in Java**.
- Eine andere Möglichkeit Daten binär in einem Strom auszugeben, wird durch die Klassen DataOutputStream angeboten.
 - Diese Klasse implementiert ebenfalls **DataOutput**
 - Erlaubt aber nicht direkt das Serialisieren beliebiger Objekte



Beispiel

```
public void writeToFile(String fname) throws IOException {  
    // Der Einfachheit halber reichen wir die Exception weiter.  
    Point p = new Point(1.4, 3.14);  
    // Schreiben  
    FileOutputStream fos = new FileOutputStream(fname, true);  
    ObjectOutputStream oos = new ObjectOutputStream(fos);  
    oos.writeObject(p);  
    oos.close();  
  
    // Lesen – Erklärung noch später  
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream(fname));  
    Point q = (Point) ois.readObject();  
    System.out.println("Ausgabe: " + q);  
    ois.close();  
}
```

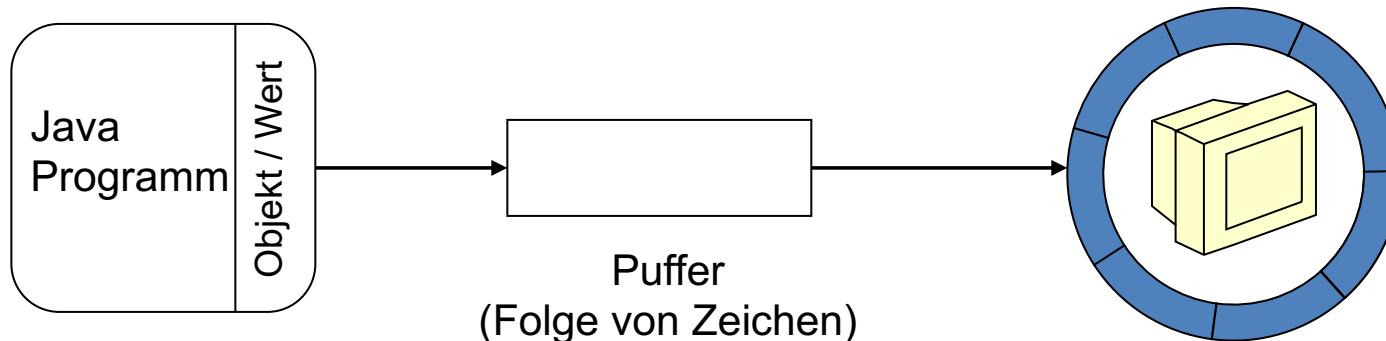
Ausgabe im Binärformat

```
import java.io.*;  
  
public class DataOutputStreamTest {  
    public static void main(String[] args) {  
        try {  
            DataOutputStream out = new DataOutputStream(  
                new BufferedOutputStream(  
                    new FileOutputStream("test.txt")));  
            out.writeInt(1);  
            out.writeInt(-1);  
            out.writeDouble(Math.PI);  
            out.writeUTF("häßliches");  
            out.writeUTF("Entlein");  
            out.close();  
        }  
        catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

C:\KonzepteDev\Programmierung\Test\test.txt										
0x00:	0000	0001	FFFF	FFFF	4009	21FB	5444	2D18	...	yyyy@.!üTD-
0x10:	000B	68C3	A4C3	9F6C	6963	6865	7300	0745	..hÄällches..E	ntlein
0x20:	6E74	6C65	696E							

Ausgabe mit der Klasse PrintStream

- Die Klasse **System** bietet eine Referenzvariable **out** mit einem Objekt der Klasse **PrintStream** an, das einen zeichenorientierten Bildschirm modelliert.



- Auswahl von Methoden aus PrintStream
 - `print (<Datentyp> x);` // x wird in den Puffer geschrieben
 - `flush();` // Der Puffer wird geleert.
 - `println(<Datentyp> x);` // entspricht: `print(x); print('\n');`

Ausgabe mit der Klasse PrintStream

- PrintStream ist ein **FilterOutputStream**
 - Bietet Konstruktor PrintStream(OutputStream)
 - Fügt so dem zugrunde liegenden Ausgabestrom die Möglichkeit hinzu eine Menschen-lesbare (Text-basierte) Darstellung von Werten zu erzeugen
- PrintStream bietet zusätzlich Konstruktoren, um den Datenstrom direkt in eine Datei zu schreiben
 - PrintStream(File datei), PrintStream(String dateName)
 - Intern wird dann ein FileOutputStream erzeugt, an den der Datenstrom weitergegeben wird.

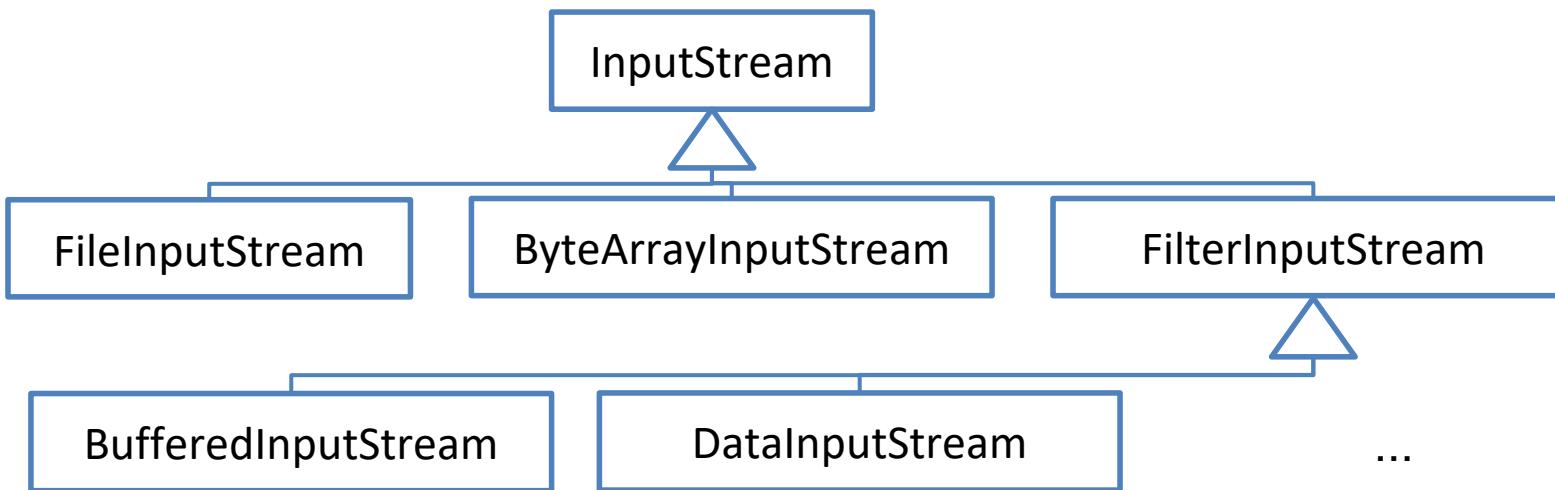


14.3.2 Eingabe Byte-Streams

- Basis der Eingabe-Streams ist die **abstrakte Klasse InputStream**.
- Sie stellt unter anderem folgende Methoden zur Verfügung:
 - `public abstract int read() throws IOException`
 - Diese Methode muss in den abgeleiteten Klassen implementiert werden.
 - `public int read(byte[] b) throws IOException`
 - Liest die Daten aus dem Stream in das Array
 - `public int read(byte[] b, int off, int len) throws IOException`
 - Liest len-off+1 Bytes aus der Eingabe und überträgt sie in das Array.
 - `public void close() throws IOException`
 - Schließt den Stream. Danach sind keine weiteren Operationen erlaubt.
 - `public void reset() throws IOException`
 - Setzt die Lesemarke auf den Anfang des Streams.

Verschachtelung von Eingabeströmen

- Analog zu Ausgabeströmen gibt es:
 - Eingabeströme, die direkt Daten einlesen („Quellen“)
 - Filter-Eingabeströme (`FilterInputStream`), die Funktionalität hinzufügen



```
import java.io.FileInputStream;
import java.io.FileOutputStream;

public class FileCopy {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("java FileCopy inputfile outputfile");
            System.exit(1);
        }
        try {
            FileInputStream in = new FileInputStream(args[0]);
            FileOutputStream out = new FileOutputStream(args[1]);
            byte[] buf = new byte[4096];
            int len;
            while ((len = in.read(buf)) > 0)
                out.write(buf, 0, len);
            out.close();
            in.close();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Ist der Aufruf korrekt?

beide Ströme initialisieren

je 4096 Byte lesen und schreiben
(bis auf den letzten Zugriff)

SequenceInputStream

- Ein `SequenceInputStream` dient dazu, zwei oder mehr `InputStreams` so miteinander zu verbinden, dass die `Daten nacheinander` aus den einzelnen Streams `gelesen` werden.
- Die beteiligten Streams können direkt an den Konstruktor übergeben werden:
 - `public SequenceInputStream(InputStream s1, InputStream s2)`
- Darüber hinaus wird es in der Klasse ermöglicht, dieses Prinzip auf beliebig viele `InputStreams` anzuwenden.

ObjectInputStream und DataInputStream

- Analog zu ObjectOutputStream gibt es eine Klasse **ObjectInputStream**, mit der die geschriebenen Daten eingelesen werden können.
- Beispiel für einen **FilterInputStream**
- ObjectInputStream **implementiert** das Interface **DataInputStream**, das folgende Methoden definiert:
 - ...
 - long readLong() throws IOException
 - double readDouble() throws IOException
 - Object readObject() throws IOException
 - ...
 - String readUTF() throws IOException

Beispiel

- Der folgende Konstruktor der Klasse Point liest die Daten aus der Datei und initialisiert damit das Objekt.

```
public Point(String fname) throws IOException {  
    FileInputStream fis = new FileInputStream(fname);  
    ObjectInputStream ois = new ObjectInputStream(fis);  
  
    Point p = (Point) ois.readObject();  
    // Klasse Point muss Serializable implementieren!  
  
    ois.close();  
}
```

15. Aufzählungstypen und die switch-Anweisung

- Weitere Konzepte in Java
 - Aufzählungstypen
 - switch-Anweisung

Zusammenfassung

- Einführung in das Paket java.io
 - Verwaltung von Dateien
 - Klasse File
 - Wahlfreier Zugriff auf Dateien
 - RandomAccessFile
 - Sequentieller Zugriff mit Datenströmen
- Datenströme unterscheiden zwischen
 - Byte-Datenströme
 - Klassen InputStream und OutputStream
 - Character-Datenströme
 - Reader und Writer
 - Funktionalität zum Lesen und Schreiben von Textdateien

15.1 Aufzählungstypen

- Seit dem JDK 1.5 kann man in Java auch **Aufzählungstypen** definieren.
 - Es handelt sich dabei um Klassen mit einer fest vorgegebenen Menge von Objekten.
- Eine solche Klasse wird durch eine **enum-Deklaration** erzeugt, in der alle möglichen Objekte aufgelistet werden, wie in:

```
enum Farbe {Rot, Grün, Blau, Weiss, Schwarz}  
enum Ampel {Grün, Gelb, Rot, GelbRot}  
enum Wochentag {Mo, Di, Mi, Do, Fr, Sa, So}
```

- Wochentag ist nun eine Klasse mit genau 7 Objekten, **weitere können nicht erzeugt werden**. Daher fehlt in der folgenden Zuweisung der gewohnte new-Operator:

```
Wochentag tag = Wochentag.Mi;
```

- Die Elemente der Klasse verhalten sich wie Objekte der Klasse und sind zugreifbar wie final static Datenfelder
- Objekte verschiedener enum-Typen sind nicht kompatibel, selbst wenn sie gleiche Name
 - Der Vergleich

```
Farbe.Grün == Ampel.Grün
```

muss somit zu einem **Typfehler** führen.

Konstruktoren

- Enum-Klassen können benutzerdefinierte Felder und Methoden erhalten.
- Selbst Konstruktoren sind möglich.
 - Diese können benutzt werden, um die vorhandenen Objekte geeignet zu initialisieren.
 - Eine Objekterzeugung mit dem new-Operator ist nicht möglich.
- In dem folgenden Beispiel reichern wir die Klasse Wochentag um ein Feld arbeitsTag und eine gleichnamige Methode an.
 - Der Konstruktor wird bei der Definition der enum-Klasse eingesetzt, um in den vorhandenen Objekten das Datenfeld arbeitsTag geeignet zu initialisieren:

```
enum Wochentag {  
    Mo(true), Di(true), Mi(true), Do(true),  
    Fr(true), Sa(false), So(false);  
  
    private boolean arbeitsTag;  
  
    boolean istArbeitsTag(){ return arbeitsTag; }  
  
    Wochentag(boolean mussArbeiten){ // Konstruktor  
        arbeitsTag = mussArbeiten;  
    }  
}
```

Oberklasse `java.lang.Enum`

- Alle enum-Klassen werden vom Java-Compiler in eine Unterklasse der **abstrakten Klasse `java.lang.Enum`** übertragen.
- Insbesondere folgende Methoden werden bereitstellt:
 - `boolean equals(Object other)`
 - `int compareTo(E o)`
 - Vergleich gemäß Definitionsreihenfolge der Objekte
 - `String toString()`
 - Zur Ausgabe der Objekte
 - `int ordinal()`
 - Liefert die Ordnungszahl des Enum-Objekts
 - `static <T extends Enum<T>> T valueOf(String)`
 - Zu einem Konstantennamen und einer Enum-Klasse T, wird das Enum-Objekt geliefert.

Zugriff auf die Objekte

- Die statische Methode **values()** liefert für jede Aufzählungsklasse ein Array mit allen Objekten der Klasse, diese kann man dann z.B. mit einer for-Schleife aufzählen:

```
for(WochenTag t : WochenTag.values())
    if(t.istArbeitsTag()) System.out.println(t);
```

Mo
Di
Mi
Do
Fr

- Eine weitere nützliche Methode **ordinal()** liefert die Nummer jedes Enum-Objektes. Daher hätten wir **istArbeitstag()** auch berechnen können:

```
boolean istArbeitsTag() {
    return this.ordinal() <= 4;
}
```

- Die Ordinalzahl entspricht dem Index im Array

Noch ein Beispiel

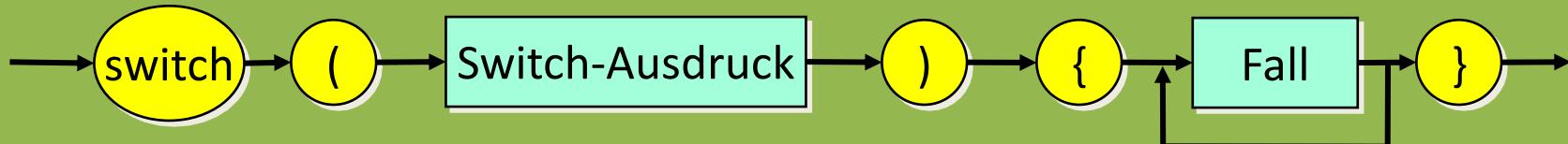
```
enum Farbe {  
    weiss, schwarz, rot, gold, gruen, blau;  
    static Farbe[][] fahnen = { {weiss, blau}, {schwarz, rot, gold},  
        {rot, weiss}, {rot}  
    };  
}  
  
class EnumFarbenTest{  
    public static void main(String[] args){  
        for(Farbe[] f: Farbe.fahnen) {  
            for (Farbe g: f)  
                System.out.print(g + " ");  
            System.out.println();  
        }  
    }  
}
```

```
weiss blau  
schwarz rot gold  
rot weiss  
rot
```

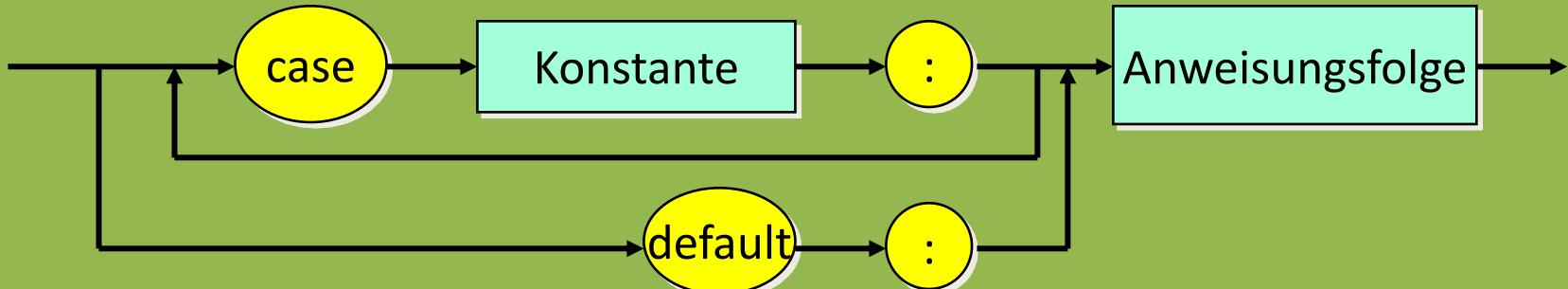
15.2 Die switch-Anweisung

- Wenn in **Abhängigkeit von dem Wert eines enum-Objekts** eine entsprechende Anweisung ausgeführt werden soll, kann man diese oft mit einer **switch-Anweisung** zusammenfassen.

Switch-Anweisung:



Fall:



Aufbau und Bedeutung der switch-Anweisung

- Der Typ des **switch-Ausdrucks** muss **char**, **byte**, **short**, **int**, **String** oder ein **enum-Typ** sein.
- Für alle **unterstützten** Ergebniswerte dieses Ausdruckes kann jeweils ein **Fall** formuliert werden.
 - Dieser besteht immer aus einer Konstanten und einer Folge von Anweisungen.
 - Schließlich ist noch ein **Default-Fall** erlaubt.
- Verhalten der switch-Anweisung
 - Wenn der **switch-Ausdruck** gleich dem Wert eines der **Fälle ist**, so wird die **zugehörige Anweisungsfolge** und die **aller folgenden Fälle (!)** ausgeführt.
 - Ansonsten wird, falls vorhanden, der **Default-Fall** und **alle folgenden Fälle (!)** ausgeführt.

Default-Fall sollte letzter Fall sein!

Verwendung von break und return

- Die **switch-Anweisung** ist gewöhnungsbedürftig und führt häufig zu unbeabsichtigten Fehlern.
- Die Regel, dass nicht nur die Anweisung eines Falles, sondern auch die Anweisungen aller auf einen Treffer folgenden Fälle ausgeführt werden, führt häufig zu Fehlern.
 - Tatsächlich ist dies ein Relikt von der Programmiersprache C.
- Wenn man möchte, dass immer nur genau ein Fall ausgeführt wird, so muss man jeden **Fall** mit einer
 - **return-Anweisung** oder
 - **break-Anweisung** enden lassen. (Es wird der durch die **switch-Anweisung** gebildete Block verlassen)

Beispiel: switch mit int

- Im folgenden Beispiel werden **Fälle** mit einer **return-Anweisung** beendet
- Wir definieren eine Methode, die die Anzahl der Tage eines Monats bestimmt.

```
static int tageProMonat(int j, int m){  
    switch (m) {  
        case 1: case 3: case 5: case 7:  
        case 8: case 10: case 12: return 31;  
        case 2: if (schaltJahr(j)) return 29; else return 28;  
        default: return 30;  
    }  
}
```

Beispiel: switch mit String

- Im folgenden Beispiel testen wir ein switch mit einem String-Ausdruck.

```
static int getMonatAlsZahl(String monat) {  
    int monatAlsZahl = 0;  
    if (monat == null) return monatAlsZahl;  
    switch (monat.toLowerCase()) {  
        case "januar":      monatAlsZahl = 1; break;  
        case "februar":     monatAlsZahl = 2; break;  
        case "märz":         monatAlsZahl = 3; break;  
        case "april":        monatAlsZahl = 4; break;  
        case "mai":          monatAlsZahl = 5; break;  
        case "juni":          monatAlsZahl = 6; break;  
        case "juli":          monatAlsZahl = 7; break;  
        case "august":        monatAlsZahl = 8; break;  
        case "september":    monatAlsZahl = 9; break;  
        case "oktober":      monatAlsZahl = 10; break;  
        case "november":     monatAlsZahl = 11; break;  
        case "dezember":     monatAlsZahl = 12; break;  
        default:             monatAlsZahl = 0; break;  
    }  
    return monatAlsZahl;  
}
```

Beispiel: switch mit enum

- Im folgenden Beispiel testen wir ein switch mit einem enum-Ausdruck.

```
enum Season{WINTER, SPRING, SUMMER, FALL};

public class SeasonTest {

    public static String getGermanName(Season season) {
        String str = null;
        switch (season) {
            case WINTER: str = "Winter"; break;
            case SPRING: str = "Frühling"; break;
            case SUMMER: str = "Sommer"; break;
            case FALL: str = "Herbst"; break;
        }
        return str;
    }

    public static void main(String[] args) {
        for(Season s : Season.values())
            System.out.println(s + " = " + getGermanName(s));
    }
}
```

16 Innere Klassen

- Klassen und Interfaces können **zur besseren Strukturierung** von Programmen verschachtelt werden.
 - Eine **innere Klasse** wird innerhalb einer anderen Klasse definiert.
 - Eine nicht-innere Klasse wird auch als **Top-Level Klasse** bezeichnet.

```
public class TopLevelKlasse {  
    ...  
    public class InnereKlasse {  
        ...  
    }  
    ...  
}
```

- Innere Klassen sind nützliche Konzepte
 - Sie bieten insbesondere die Grundlagen für die funktionalen Konzepte in Java.

Varianten von inneren Klassen

- Unterscheidung zwischen
 - Geschachtelte Top-Level-Klassen
 - Klassen und Interfaces, die innerhalb anderer Klassen definiert sind, aber sich **trotzdem wie Top-Level-Klassen verhalten**.
 - Elementklassen
 - innere Klassen, die **in anderen Klassen** definiert sind.
 - Lokale Klassen
 - Klassen, die **innerhalb einer Methode oder eines Java-Blocks** definiert werden.
 - Anonyme Klassen
 - Lokale Klassen **ohne Namen**

Geschachtelte Top-Level-Klassen

- Geschachtelte Top-Level-Klassen bzw. geschachtelte Top-Level-Interfaces verhalten sich wie andere Klassen bzw. Interfaces in einem Paket.
 - Einziger Unterschied
 - Ihrem Namen muss immer der Name der umgebenden Klasse als Präfix vorangestellt werden.
 - Motivation
 - Bessere Strukturierung durch geschachtelte Klassen
- Geschachtelte Klassen bzw. Interfaces werden mit **static** deklariert.

```
public class A{  
    ...  
    public static class B {  
        ...  
    }  
    ...  
}
```

```
// Verwendung der Klassen  
  
A a = new A();  
  
A.B ab = new A.B();
```

Elementklassen

- Im Gegensatz zu den geschachtelten Top-Level-Klassen handelt es sich bei Elementklassen um **echte innere Klassen**.
 - Die Definition der Klassen erfolgt **ohne das Schlüsselwort static**.
- Eine Instanz einer Elementklasse ("inner") existiert in einer Instanz der umgebenden Klasse ("outer")
- Eine Instanz einer Elementklasse hat vollen **Zugriff auf alle Datenfelder und Methoden** der umgebenden Instanz
 - Innere Klassen können dadurch den (**privaten**) Zugriffsschutz aufweichen.
 - Diese Lösung ist aber oft besser als die Klassen nebeneinander zu implementieren und den Zugriffsschutz von private auf package-protected abzusenken

Class-Dateien

- Die Elementklassen tragen den Namen der übergeordneten Klasse(n) und den eigenen, wobei diese mit \$ unterteilt werden.

```
public class A{  
    public class B {  
        public class C {  
        }  
    }  
}
```

javac A.java


Erzeugte class-Dateien

- A.class
- A\$B.class
- A\$B\$C.class

Objekte von Elementklassen

- Objekte von Elementklassen sind **immer von einem Objekt der umgebenden Klasse abhängig**.
 - Insbesondere der Konstruktor der Elementklasse kann nur mit Hilfe eines Objekts der äußeren Klasse aufgerufen werden.
 - Damit kann ein Objekt der Elementklasse auch auf die (privaten) Datenfelder und Methoden der äußeren Klasse zugreifen.

```
public class A{  
    private int i = 42;  
    public class B {  
        public int j;  
        B() {  
            j = i; // ok  
        }  
    }  
}
```

```
// Anwendung  
  
A a = new A();  
  
B b = a.new B();  
  
System.out.println(b.j);  
// Zugriff auf b.i funktioniert nicht
```

- Es gilt folgende Einschränkung:
 - Elementklassen haben **keine statischen Datenfelder und Methoden**.

Bsp.: Iterator als Elementklasse

```
public class LinkedList<E> implements Iterable<E> {
    private ListElement<E> head, tail;
    ...

    public Iterator<E> iterator() {
        return new ListenIterator();
    }

    /* Elementklasse lokal innerhalb von LinkedList */
    private class ListenIterator implements Iterator<E> {
        private ListElement<E> nextElem = head; // Zugriff auf head erlaubt !

        public E next() {
            ListElement<E> tmp = nextElem;
            nextElem = nextElem.next;
            return tmp.data;
        }

        public boolean hasNext(){
            return nextElem != tail;
        }
    }
}
```

Anonyme Klassen

- Anonyme Klasse werden wie lokale Klassen innerhalb von Anweisungsblöcken definiert.
 - Anonyme Klassen haben jedoch **keinen Namen!**
- Sie entstehen immer gleichzeitig zusammen mit **einem** Objekt, aber
 - sie haben **keinen Konstruktor**
 - sie haben die gleichen Beschränkungen wie lokale Klassen.
- Syntax von anonymen Klassen



- Für die Anonyme Klasse kann man **keine extends- oder implements-Klauseln** angeben.

Bsp.: Iterator als anonyme Klasse

```
/**  
 * Ein iterator mit einer anonymen Klasse innerhalb der Klasse  
 LinkedList.  
 */  
 public Iterator<E> iterator() {  
     return new Iterator<E>() {  
         ListElement nextPos = head;  
  
         public boolean hasNext() {  
             return nextPos != tail;  
         }  
  
         public E next() {  
             ListElement tmp = nextPos;  
             nextPos = nextPos.next;  
             return tmp.data;  
         }  
     };  
 };
```

// Semikolon der return-Anweisung nicht vergessen.

Dies ist möglich, obwohl
Iterator ein Interface ist.

Diskussion

- Durch die Verwendung von Generics und anonymen Klassen haben wir eine allgemein einsetzbare Lösung zum Filtern von Daten auf Iteratoren entwickelt.
- Jedoch lässt sich darüber streiten, ob dies wirklich eine elegante Lösung ist.
 - Wir müssen eine (anonyme) Klasse bauen, obwohl wir nur eine kleine Funktion (test) implementieren wollen.
 - Wir müssen die Typen aufeinander abstimmen, so dass auch alles zueinander passt.
- Dieses Probleme werden durch die Verwendung von den neuen funktionalen Konzepten von Java („Lambdas“) größtenteils behoben.

17. Parallelle Programmierung in Java mit Threads

- Überblick
 - Klassische Verwendung von Threads
 - Synchronisation
 - ExecutorService, Callable und Future

Motivation

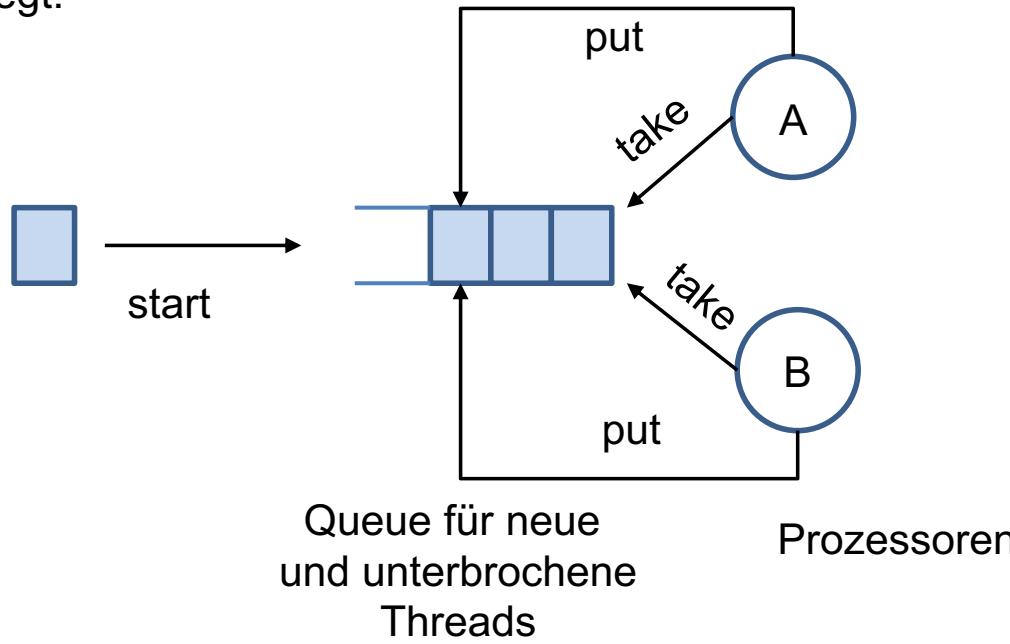
- Komplexe Programmsysteme müssen gleichzeitig verschiedene Benutzer bedienen (Pseudo-Parallelität).
 - Verschiedene Operationen laufen gleichzeitig miteinander (auch wenn nicht notwendigerweise mehrere CPUs vorhanden sind).
 - Beispiele
 - Betriebssysteme
 - Datenbanksysteme
- Parallele Rechnersysteme
 - Eine Aufgabe soll parallel programmiert werden, um alle Kerne eines Rechners zu nutzen und somit die Laufzeit eines Programms zu verringern.
 - Parallelisieren von klassischen Algorithmen wie Sortieren, Suchen, ...

Ausführungsthreads

- Eine moderne CPU besitzt mehrere Kerne
 - Durch Simultaneous Multi-Threading (SMT) kann jeder Kern mehrere Instruktionen parallel ausführen
 - Typische Desktop CPUs haben 4-8 Kerne, die jeweils 2 SMTs ausführen können
 - D.h., 8-16 Hardware-Threads
- Oft benötigen Anwendungen mehr Ausführungsthreads
 - Außerdem werden mehrere Anwendungen gleichzeitig ausgeführt
- Das Betriebssystem und die Programmiersprache unterstützen durch Software-Threads wesentlich mehr Ausführungsthreads
 - Größenordnung mehrere Tausend bis Zigtausend

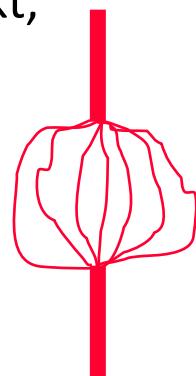
Realisierung von Software-Threads durch Betriebssystem/ Java Virtuelle Maschine

- Threads werden nach dem Start in einer Queue abgelegt.
- Prozessoren nehmen sich die Threads aus der Queue und verarbeiten diese bis ein Ereignis, wie z. B. das Zeitlimit ist überschritten, eintritt.
 - Falls Thread noch nicht fertig ist, wird dieser unterbrochen und in die Queue abgelegt.



17.1 Threads

- Bisher
 - Ein Java-Programm hat zu jedem Zeitpunkt genau eine aktive Methode.
- Threads erlauben die gleichzeitige Verarbeitung von mehreren aktiven Methoden in einem Programm.
 - Der Programmablauf verläuft gemeinsam bis zu einem gewissen Punkt, dann **teilt** sich die Ausführung in mehrere **Threads**.
 - Zu einem späteren Zeitpunkt kann die Ausführung dann wieder **gemeinsam** fortgesetzt werden.
- Wichtige Klassen und Interfaces in Java
 - **funktionales Interface Runnable**
 - Klasse Thread



Interface Runnable

- Funktionale Schnittstelle bietet die Methode `void run()`.
 - In der Methode wird die gewünschte Funktionalität implementiert, die über ein Objekt der Klasse Thread parallel ablaufen soll.
- Zwei Möglichkeiten
 - Implementierung über eine eigene Klasse

```
class TicTacToe implements Runnable{  
    String was;  
    TicTacToe(String s){ was = s; }  
    public void run(){  
        System.out.println(was);  
    }  
}
```

- Üblicher: Implementierung als anonyme Klasse oder Lambda

```
Runnable tic = () -> System.out.println("Tic");
```

Klasse Thread

- Die Klasse Thread wird benutzt, um ein Runnable-Objekt parallel auszuführen.

- Erzeugung eines Objekts der Klasse mit einem Runnable-Objekt

```
Thread t = new Thread(tic);
```

- Starten des Threads mit der Methode start()

```
t.start();
```

- Danach gibt es zwei aktive Methoden (Ausführungsfäden, Threads)
 - Hauptprogramm wird fortgeführt.
 - Dieser Ausführungsfaden endet mit der Methode main.
 - Methode start führt die Methode run des Objekts tic aus.
 - Dieser Ausführungsfaden endet mit der Methode run.
 - Beide laufen voneinander unabhängig.

Beispiel

- Betrachten wir folgendes Programm:

```
public static void main(String args[]) {  
    System.out.println("Thread Beispiel");  
    Runnable tic = () -> System.out.println("Tic ");  
    Runnable tac = () -> System.out.println("Tac ");;  
    Runnable toe = () -> System.out.println("Toe ");;  
    new Thread(tic).start();  
    new Thread(tac).start();  
    new Thread(toe).start();  
    System.out.println("Hauptprogramm")  
}
```

1. Programmstart
z.B.:

Thread Beispiel
Hauptprogramm
Tac
Tic
Toe

2. Programmstart
z.B.:

Thread Beispiel
Tic
Toe
Tac
Hauptprogramm

- Die Ausgabe ist **nichtdeterministisch**.
 - Das gleiche Programm kann ganz verschiedene Ausgaben produzieren

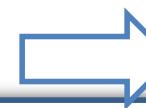
Weitere Methoden der Klasse Thread

- **static void sleep(long millis)**
 - Beim Aufruf dieser Methode wird die aufrufende Methode für *millis* Millisekunden schlafen gelegt.
- **void setPriority(int newPriority)**
 - Hiermit kann die Priorität eines Threads auf *newPriority* gesetzt werden.
 - Ein Thread mit hoher Priorität kommt öfters auf einem Prozessor zur Ausführung als ein Thread mit niedriger Priorität.
- **void join()**
 - An dieser Stelle wird gewartet bis der zuvor gestartete Thread endet.



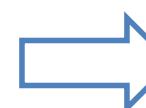
Beispiel

```
public static void main(String args[]) {  
    System.out.println("Thread Beispiel");  
    Runnable tic = () -> System.out.println("Tic ");  
    Runnable tac = () -> System.out.println("Tac ");  
    Runnable toe = () -> System.out.println("Toe ");  
    Thread[] t = {new Thread(tic), new Thread(tac), new Thread(toe)};  
    for (i=0; i < 2; i+=1)  
        t[i].start();  
    for (i=0; i < 2; i+=1)  
        try {  
            t[i].join();  
        }  
        catch (InterruptedException ie ){ /* ToDo */ }  
    System.out.println("Hauptprogramm");  
}
```



Thread Beispiel
Tic
Toe
Tac
Hauptprogramm

- Die Ausgabe ist immer noch nichtdeterministisch.
 - Jedoch ist garantiert, dass stets am Schluss "Hauptprogramm" ausgegeben wird.



Thread Beispiel
Hauptprogramm
Tac
Tic
Toe

17.2 Synchronisation

```
public class SyncProblems {  
    static int count = 0;  
    static void increment() {  
        int countTemp = count;  
        try {  
            Thread.sleep(1);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        count = countTemp + 1;  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread[] threads = new Thread[100];  
        for (int i = 0; i < threads.length; i++) {  
            threads[i] = new Thread(SyncProblems::increment);  
            threads[i].start();  
        }  
        for (Thread thr: threads) {  
            thr.join();  
        }  
        System.out.println(count);      // Was ist das Ergebnis?  
    }  
}
```

17.2 Synchronisation

```
public class SyncProblems {  
    static int count = 0;  
    static void increment() {  
        int countTemp = count;  
        try {  
            Thread.sleep(1);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        count = countTemp + 1;  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread[] threads = new Thread[100];  
        for (int i = 0; i < threads.length; i++) {  
            threads[i] = new Thread(SyncProblems::increment);  
            threads[i].start();  
        }  
        for (Thread thr: threads) {  
            thr.join();  
        }  
        System.out.println(count);      // Was ist das Ergebnis?  
    }  
}
```

Lost Update Problem

Ändern Threads in parallel
eine Variable, können einige
Änderungen verloren gehen.

Schlüsselwort synchronized

- Dieses Problem kann durch Verwendung von synchronized behoben werden.

```
static synchronized void increment() {  
    int countTemp = count;  
    try {  
        Thread.sleep(1);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    count = countTemp + 1;  
}
```

- Damit wird sichergestellt, dass nur ein Thread zu einem Zeitpunkt die Methode ausführen kann.
 - Man kann auch einzelne Code-Blöcke mit dem Schlüsselwort synchronized schützen.
- Der Code wird dann auch als **Thread-safe** bezeichnet.

17.3 Parallele Streams (java.util.stream)

- Eine Pipeline beschreibt
 - **was** berechnet,
 - aber **nicht wie** etwas berechnet werden soll.
- Die Iteration der Elemente wird intern implementiert
 - for-Schleife vs. forEach
 - Optimierungen möglich unter Ausnutzung interner Strukturen
 - einfache Parallelisierung

```
double average = Stream.of(5, 4, 7, 2, 10, 8).parallel()  
    .filter(x -> {  
        System.out.println("Filter: " + x);  
        return x % 2 == 0;  
    }).collect(Collectors.averagingInt(v -> v));  
System.out.println("Durchschnitt: " + average);
```

Filter: 8	Filter: 8
Filter: 7	Filter: 10
Filter: 5	Filter: 7
Filter: 10	Filter: 4
Filter: 4	Filter: 2
Filter: 2	Filter: 5
Durchschnitt: 6.0	Durchschnitt: 6.0

17.4 Executor-Schnittstelle

- Problem bei der Thread-Programmierung ist oft die Strategie und Implementierung für die Ablaufsteuerung.
 - Da die Erzeugung der Thread-Objekte teuer ist, sollten Threads wiederverwendet werden
- Vorgefertigte Strategien werden als Objekte des Interface ExecutorService im Package `java.util.concurrent` zur Verfügung gestellt.
 - Die Objekte werden über statische Methoden aus der Klasse Executors geliefert.

Thread-Erzeugung mittels ExecutorService

- Übergabe der run-Methode an submit-Methode von ExecutorService
- Ein Executor wird zunächst gestartet.

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
executorService.submit(() -> System.out.println("Tic "));
executorService.submit(() -> System.out.println("Tac "));
executorService.submit(() -> System.out.println("Toe "));
executorService.shutdown();
```

Lambdas entsprechen der funktionalen Schnittstelle Runnable.

- Der Executor muss am Ende des Programms noch runtergefahren werden.
 - Aufruf der Methode `executor.shutdown()`

Schnittstelle Callable

- Java bietet inzwischen die Schnittstelle Callable an, um Threads zu erzeugen, die ein **Ergebnis** zurückliefern.
 - Callable-Objekte können ebenfalls über die Methode submit an ein ExecutorService-Objekt übergeben werden.

```
Callable<Integer> task = () -> {  
    try {  
        Thread.sleep(10);  
        return 42;  
    }  
    catch (InterruptedException e) {  
        throw new IllegalStateException("task interrupted", e);  
    }  
};  
ExecutorService executorService = Executors.newSingleThreadExecutor();
```

Callable ist ebenfalls eine funktionale Schnittstelle mit einer ähnlichen Bedeutung wie Runnable. Im Unterschied hierzu hat die Funktion aber einen Rückgabetyp.

- Wie kommt man an das Ergebnis des Threads ran?

Schnittstelle Future

- Hierfür steht die **Schnittstelle Future** zur Verfügung.
 - Beim submit-Aufruf mit einem Callable wird ein Future-Objekt geliefert.
- Das Future-Objekt gibt uns mit der **Methode get** das gewünschte Ergebnis.
 - Falls das Ergebnis noch nicht produziert wurde, wird an dieser Stelle gewartet bis das Ergebnis vorliegt.

```
...
//  
Future<Integer> future = executorService.submit(task);  
System.out.println("future done? " + future.isDone());  
Integer erg = 0;  
try {  
    erg = future.get();  
}  
catch (Exception e) { /* ToDo */ }  
System.out.println("Future fertig? " + future.isDone());  
System.out.print("Resultat: " + erg);
```

Zusammenfassung

- Nebenläufige und parallele Verarbeitung durch Threads
- Erzeugen und Ablaufen von Threads
 - Executor
 - Schnittstelle Runnable und Callable
- Synchronisation von Threads
 - Nutzen gemeinsamer Speicher
- Viele Konzepte aus Java für Threads und Synchronisation wurden in dieser kurzen Übersicht nicht behandelt.
→ Systemsoftware und Rechnerkommunikation