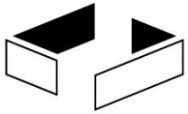
Video

Programmierung 1 - TI

Prof. Dr. Stephan Kleuker
Hochschule Osnabrück

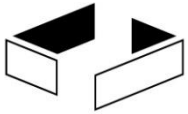
Ich



- Prof. Dr. Stephan Kleuker, geboren 1967, verheiratet, 2 Kinder
- seit 1.9.09 an der HS, Professur für Software-Entwicklung
- vorher 4 Jahre FH Wiesbaden
- davor 3 Jahre an der privaten FH Nordakademie in Elmshorn
- davor 4 ½ Jahre tätig als Systemanalytiker und Systemberater in Wilhelmshaven

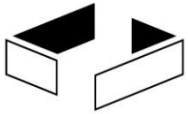
- s.kleuker@hs-osnabrueck.de, kurzfristige Zoom-Termine vereinbar

Studium (meine Sicht) (1/2)



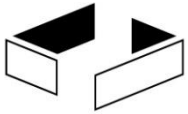
- Sie haben ersten wichtigen Schritt geschafft, sich für ein praxisnahes, qualitativ hochwertiges Studium entschieden
- Sie haben sich für einen recht schweren, sehr interessanten, sehr abwechslungsreichen und prägenden Bildungsweg entschieden
- Studium an der Hochschule: Konzepte, Vorgehensweisen, weniger Visionen, mehr Zusammenhang zur Praxis
- versuchen, frühzeitig mit der Praxis in Kontakt zu kommen

Studium (meine Sicht) (2/2)



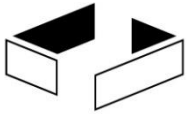
- Die nächsten drei+x Jahre werden prägend sein, nicht so wie die vorherigen Jahre, aber mehr als alle folgenden
- Sie werden viele Hochs und einige Tiefs erleben, lernen Sie aufzustehen , lernen Sie eigene Grenzen zu erkennen
- Lernen Sie, in Gruppen zu arbeiten
- Machen Sie sich nichts vor, seien aber nicht zu selbstkritisch
- Menschen verändern sich
- Versuchen sie, Ratschläge anzunehmen und zu suchen

Ablauf



- zu einem Block gehören eine Mi- und die darauf folgende Mo-Vorlesung
- zu jedem Block gibt es ein Aufgabenblatt, dass am Mo herauskommt, dass zum darauf folgenden Mo zu bearbeiten ist
- Sie stellen Ihre Ergebnisse den Praktikumsleitern vor und diskutieren Probleme im jeweils ersten Praktikumstermin der Woche
- zweiter Praktikumstermin für weitere Abnahmen, Korrekturen und Besprechung aller Probleme bei der Programmierung

Ablauf genauer

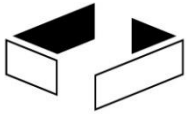


- 4h Vorlesung + 4h Praktikum = 10 CP (Credit Points, Leistungspunkte) d. h. etwa 300 Arbeitsstunden
- Praktikum :
  - Anwesenheit = (Übungsblatt vorliegen + Lösungsversuche zum vorherigen Aufgabenblatt)
  - Übungsblätter mit Punkten ($\Sigma \geq 200$), individuell bearbeitet (im Team diskutieren; jeder arbeitet selbst aus)
  - ein Übungsblatt pro Woche
  - Praktikumsteil mit 170 oder mehr Punkten bestanden
- Prüfung: Klausur recht kurz nach der Vorlesungszeit
- von Studierenden wird hoher Anteil an Eigenarbeit erwartet
- Probleme sofort melden
- wer aussteigt, teilt mit, warum

Einschub : Leistungspunkte (1/2)

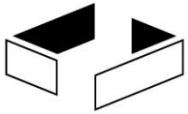
- Veranstaltung hat 10 Leistungspunkte (auch 10 Credit-Points (CP) oder 10 ECTS-Punkte genannt)
- ECTS soll Vergleichbarkeit von Leistungen in Europa ermöglichen; einfachere Anrechnung von Veranstaltungen
- deutsche Kultusminister*innen*konferenz (KMK) hat folgende Vorgaben für Arbeitsbelastung von Studierenden gemacht: *"In der Regel werden pro Studienjahr 60 Leistungspunkte vergeben, d. h. 30 pro Semester. Dabei wird für einen Leistungspunkt eine Arbeitsbelastung der Studierenden im Präsenz- und Selbststudium von 25 bis max. 30 Stunden angenommen, so dass die Arbeitsbelastung im Vollzeitstudium pro Semester in der Vorlesungs- und vorlesungsfreien Zeit insgesamt 750 bis 900 Stunden beträgt"*

Einschub : Leistungspunkte (2/2)



- Programmierung 1 hat 10 CP, d. h. ungefähr 300 Arbeitsstunden für qualifizierten Studierenden
- abhängig von Fähigkeit können 10 CP dann auch 150 Stunden oder beliebig viele Stunden bedeuten
- guter Ansatz:
  - 14 Wochen Kernvorlesungszeit, jede Woche 15 h = 210 h
  - Klausurvorbereitung und Durchführung 40 h
  - Nacharbeit, eigene Studien 50 h
- nicht gleichmäßig verteilt, erhöhter Aufwand bis zu den Klausuren, dann etwas weniger (aber Hausarbeiten später beachten)

Vorlesung



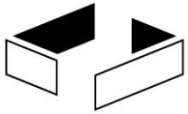
- Sie sind für Ihre Teilnahme alleine verantwortlich
- Vorlesungsstoff muss nachbearbeitet werden
- arbeiten Sie vorgeführte Beispiele nach, machen Sie eigene Beispiele
- tauschen Sie sich bei Fragen mit anderen Studierenden aus, nutzen Sie E-Mail mit Fragen an den Prof

- in Praktika wird Kenntnis der Vorlesungsstoffs erwartet (natürlich sind konkrete Nachfragen erlaubt/ gewünscht!)

Praktikum

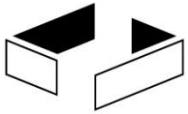
- vorgegebene Aufgaben in einem vorgegebenen Zeitraum bearbeiten
- Aufgaben bereiten Vorlesungsstoff nach, vertiefen ihn, ergänzen ihn und fordern selbstständige Einarbeitung
- im Praktikum werden Lösungen der Aufgaben abgenommen (oder Verbesserungen gefordert)
- im Praktikum sind immer ein oder zwei betreuende Personen anwesend, die man befragen kann
- im Praktikum herrscht Anwesenheits- und Aktivitätspflicht Gesamtaufwand für Praktikumsaufgaben für durchschnittlich guten Studierende zusammen 14 h pro Woche !!

Praktikum - Aufgabenbearbeitung



- Blatt 1 eventuell alleine, danach mindestens 2er-Gruppen
- sinnvoll: Pairprogramming, zwei Personen an einem Rechner
- Ansatz: eigene Tastatur und Maus mitbringen

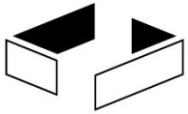
Praktikumsorganisation



- Anmeldung für Vorlesung und die Praktika (ILIAS nur gering genutzt)
- zentrale Webseite http://kleuker.iui.hs-osnabrueck.de/WS25_Prog1/index.html
- Aufgabenblätter online bis Freitag für Montag KW x
- Vorstellung und betreute Bearbeitung in KW x
- Abnahme im Praktikum Montag KW x+1 (zweites Praktikum für restliche Abnahmen und geforderte Nacharbeiten)

Hinweis: Selbst in Praktikumsgruppen eintragen

Hardware / Software



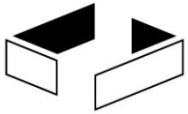
Software

- zu benutzende Entwicklungsumgebung steht als Zip-File von der Veranstaltungsseite zur Verfügung http://kleuker.iui.hs-osnabrueck.de/kleukersSEU/index.html
- enthält Java SE, siehe https://www.azul.com/downloads/zulu-community/
- enthält Entwicklungsumgebung BlueJ (http://www.bluej.org/download/download.html)
- benötigt wird: Programm für Screenshots, z. B. FastStone Capture 5.3

Hardware

- im EDVSZ im SI-Gebäude stehen Rechner zur Verfügung
- **Sie benötigen Rechner mit Windows 10, 64 bit**

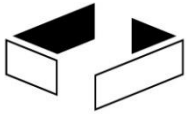
Klausur



- ausgewählte vergleichbare Aufgaben in Praktika behandelt
- Orientierung an Beispielklausur möglich

- selbständiges Erklären von Fachbegriffen
- was passiert in gegebenen Programmfragmenten
- programmieren auf dem Papier

Voraussetzungen

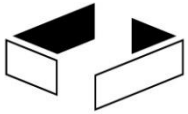


- außer Hochschulreife, keine

beinhaltet

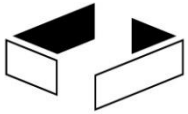
- Fähigkeit, selbständig den Lernprozess zu organisieren
- Fähigkeit zur Gruppenarbeit
- Fähigkeit, Zeiten von konzentrierten Arbeiten und freier Zeitgestaltung zu trennen und zu koordinieren
- Fähigkeit zur kritischen Selbstreflexion zum erreichten Lernstand
- Kopieren ist nicht Kapieren
- Vorlesungen und Praktika als Lernunterstützung zu sehen (Sie sind für Lernerfolg allein verantwortlich)
- Umgang mit Windows-PC

Ziele



- Konzepte der objektorientierten Programmierung verstehen und selbstständig nutzen
- Eigenständig einfache OO-Programme entwickeln, Fehler beseitigen und längerfristig wartbar machen
- Erlernen eines systematischen inkrementellen Prozesses zur Entwicklung größerer Programme
- Erlernen und Anwenden der Fachbegriffe rund um die Programmierung
- Verständnis für die grundsätzlichen Schritte vom Programmcode zur Ausführung auf dem Rechner

Einordnung von Programmierung ins Studium



Abstraktionsgrad, abstraktes Denken

- Erstellung wart- und erweiterbarer SW-Systeme
- Erstellung von Software-Architekturen
- Nutzung von Software-Architekturen
- Nutzung von Design-Pattern
- Modelle von Software
- wart- und erweiterbare Programme
- systematische Programmierung
- laufendes einfaches Programm

5. Semester

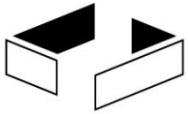
4. Semester

Fachinformatik

1. Semester

gute Schule

Literatur



- Vorlesung orientiert sich nicht genau an einem Buch

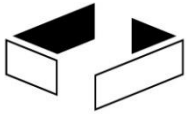
sicherlich sehr hilfreich:

- David J. Barnes , Michael Kölling, Java lernen mit BlueJ: Eine Einführung in die objektorientierte Programmierung, 6. Auflage, Pearson Studium, 2017

erst ab Mitte des Semesters hilfreich (jeweils aktuelle Auflage):

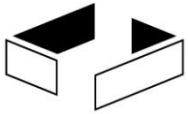
- Cornelia Heinisch, Frank Müller-Hofmann, Joachim Goll, Java als erste Programmiersprache, Vieweg+Teubner
- Dietmar Abts, Grundkurs JAVA: Von den Grundlagen bis zu Datenbank und Netzanwendungen, Vieweg+Teubner
- Christian Ullenboom, Java ist auch eine Insel, Rheinwerk Computing, (auch: http://openbook.rheinwerk-verlag.de/javainsel/)
- Guido Krüger, Thomas Stark, Handbuch der Java-Programmierung, Addison-Wesley, (auch unter http://www.javabuch.de/download.html)
- Sven Eric Panitz, Java will nur spielen, Vieweg+Teubner

Objects First



- Die Vorlesung folgt dem Ansatz, dass man Klassen und Objekte als Basis der objektorientierten Programmierung zuerst lernen sollte
- Viele veraltete Ansätze erklären erst Ablaufstrukturen (; if while for) und dann Objekte
- Mit beiden Ansätzen kann man Programmieren lernen, aber
  - mit dem zweiten Ansatz lernen es weniger auf Anhieb
  - die, die es mit dem zweiten Ansatz lernen, werden häufiger zu schlechten objektorientierten Entwicklern
- Aber auch neue Bücher nutzen den Ansatz nicht
  - Es hat auch gedauert bis sich "Die Erde ist rund" durchgesetzt hat; ohne den Ansatz macht die meiste Astronomie wenig Sinn

Programmiersprache

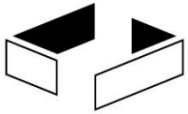


- Programmiersprache ist Hilfsmittel beim programmieren lernen
- hier **kein** Java-Kurs, sondern **Programmierkurs** mit Java
- keine auch nur annähernd vollständige Einführung in Java (für Expertise wichtige Programmkonstrukte fehlen)
- Konzept:
  - erstes Semester grundlegende Ideen
  - zweites Semester, weitere Ideen mit neuen Programmiersprachen (C++ und etwas C)
- Ziel: ab Ende des zweiten Semesters grundsätzlich in der Lage sich in fehlende Sprachkonstrukte und andere ähnlich strukturierte Programmiersprachen einzuarbeiten (5. + 6. Semester: Projekte und Bachelor-Arbeiten auch in C#, PHP, …)

Hintergrund von Programmiersprachen (Ausschnitt)

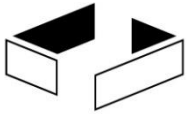
Sprache	seit	basiert auf	Anwendungsgebiet (exemplarisch)
Fortran	1954		math. Berechnungen
Cobol	1959		Banken, Versicherungen
Smalltalk	1972	Simula	Einführung Objektorientierung
C	1972	Algol	Betriebssysteme, sehr HW-nahe Programmierung (E-Technik)
Objective C	1981	C, Smalltalk	jetzt: Apple-Apps
C++	1983	C	C mit OO, Graphik, Bildverarbeitung
Python	1994	Pascal	(durch Zufall): KI-Bibliotheken
PHP	1995	C, Pascal, gehacke	einfache kleine Web-Apps
Java	1995	Smalltalk, C++	große sklalierbare Web-Apps (ERP), Standardprogramme
C#	2001	Java, C	„Java von Microsoft“
Go	2009	C (Java)	Microservice, HW-nah

https://www.tiobe.com/tiobe-index/



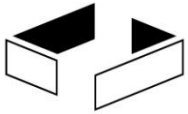
Aug 2025	Aug 2024	Programming Language / OO		Ratings	Change
1	1	Python	+	26.14%	+8.10%
2	2	C++	+	9.18%	-0.86%
3	3	C		9.03%	-0.15%
4	4	Java	+	8.59%	-0.58%
5	5	C#	+	5.52%	-0.87%
6	6	JavaScript	(+)	3.15%	-0.76%
7	8	Visual Basic		2.33%	+0.15%
8	9	Go	(+)	2.11%	+0.08%
9	25	Perl		2.08%	+1.17%

https://www.tiobe.com/tiobe-index/



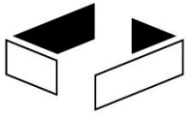
Programming Language	2025	2020	2015	2010	2005	2000
Python	1	3	7	7	7	25
C++	2	4	3	4	3	2
Java	3	1	2	1	2	3
C	4	2	1	2	1	1
C#	5	5	5	6	10	10
JavaScript	6	7	8	9	11	7
Go	7	13	56	183	-	-
Visual Basic	8	12	11	-	-	-
SQL	9	9	-	-	-	

Inhalt (erste Planung) (1/2)



■	Programmierung		
■	Objekt	■	Algorithmus
■	Klasse	■	geschachtelte Schleifen
		■	Iterator
■	Konstruktor	■	for
■	Objektmethoden	■	Klassenvariablen/-methoden
■	Debugger	■	Unit Test
■	Objektweitergabe	■	Vererbung
■	Alternative	■	Überschreiben
■	equals	■	statische Polymorphie
■	Strings - toString	■	dynamische Polymorphie
■	ArrayList - Einstieg	■	casten
■	Schleife	■	Klasse Object
■	switch	■	abstrakte Klasse

Inhalt (erste Planung) (2/2)



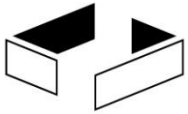
	Interface
	Kommentare
	Mehrfachvererbung
	Array
	Start von Java-Programmen
	Exception
	Collection Framework
	Konstanten
	Aufzählungen
	Pakete
	Entwicklungsumgebungen
	Einige Realitätsbeichten
	Programmieren ohne echte Klassen

	Laden und Speichern
	Frames, Knöpfe und Ereignisbehandlung
	Layout, hierarchischer GUI-Aufbau
	Rundflug über GUI-Bausteine

Video

Programmierung

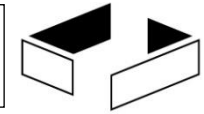
Effekte von Programmen



Beispiel PowerPoint

- Bilder werden über Computer-Hardware-Schnittstelle als Signale an Ausgabegeräte (Monitor, Beamer) geschickt
- Programm reagiert auf Maus- und Tastatureingaben
- Programm wird gestartet (Icon, Dateibrowser, …)
- Programm ist bedienbar (man bestimmt was wann wo gezeichnet wird)
- Programm ist nicht frei bedienbar (man muss Bedienschritte einhalten)
- PowerPoint ist ein Ergebnis von Programmierung
- PowerPoint erlaubt die Gestaltung (Programmierung ?) von Präsentationen

Was steckt dahinter (1/3)

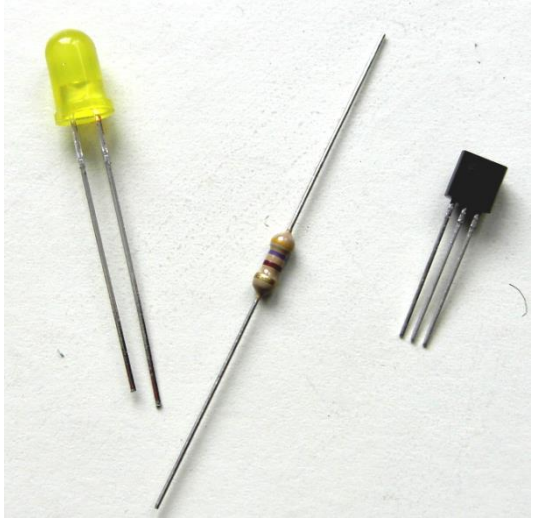


Physik

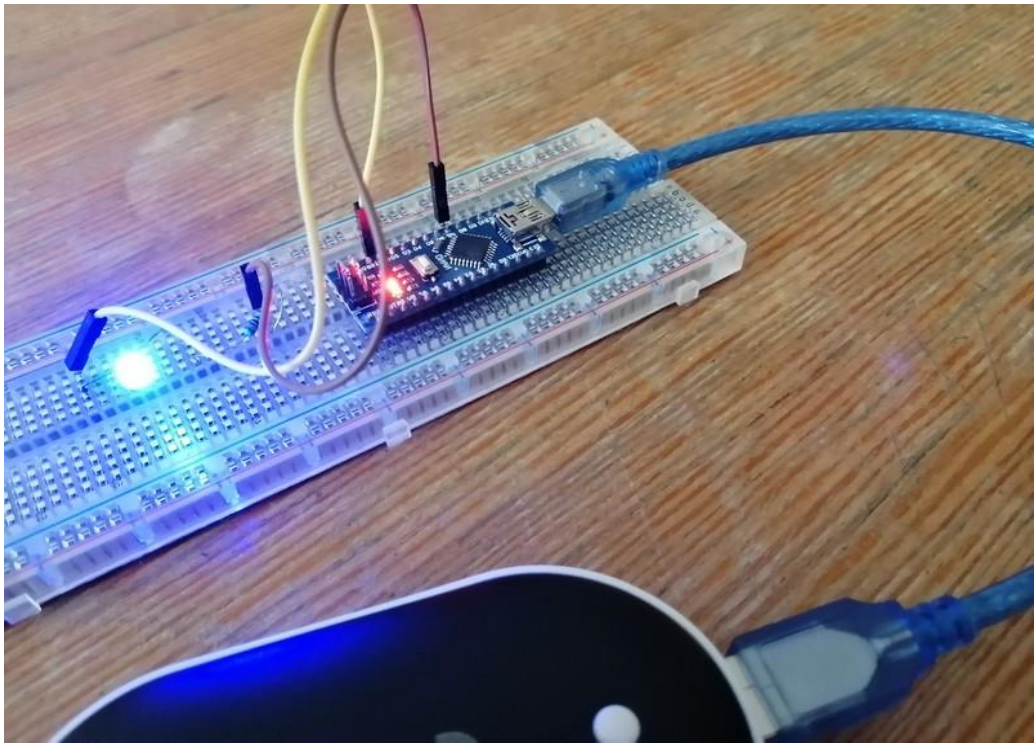
- Strom fließt durch elementare Bauteile
- Physikalische Veränderungen (Zustände)

Elektrotechnik

- Verschaltung von physikalischen Bauteilen
- Zusammenfassung von Bauteilen zu steuerbaren Einheiten



Was steckt dahinter (2/3)



Technische Informatik

- Z. B. Entwurf und Realisierung von Mikroprozessoren (Computer, Auto, Maschine, Waschmaschine)

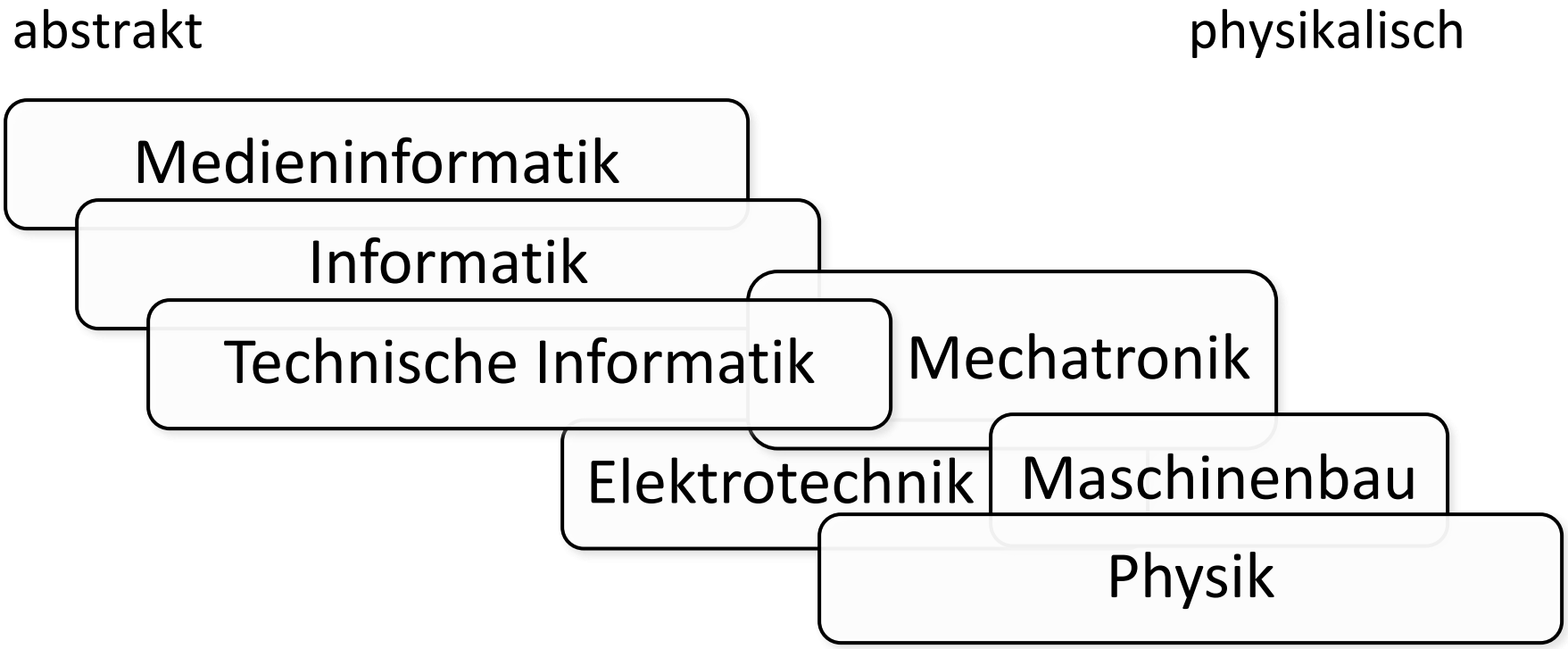
Informatik

- Z. B. Planung, Realisierung und Wartung von Anwendungssoftware (PC, Client-Server, Web, mobil)

Medieninformatik

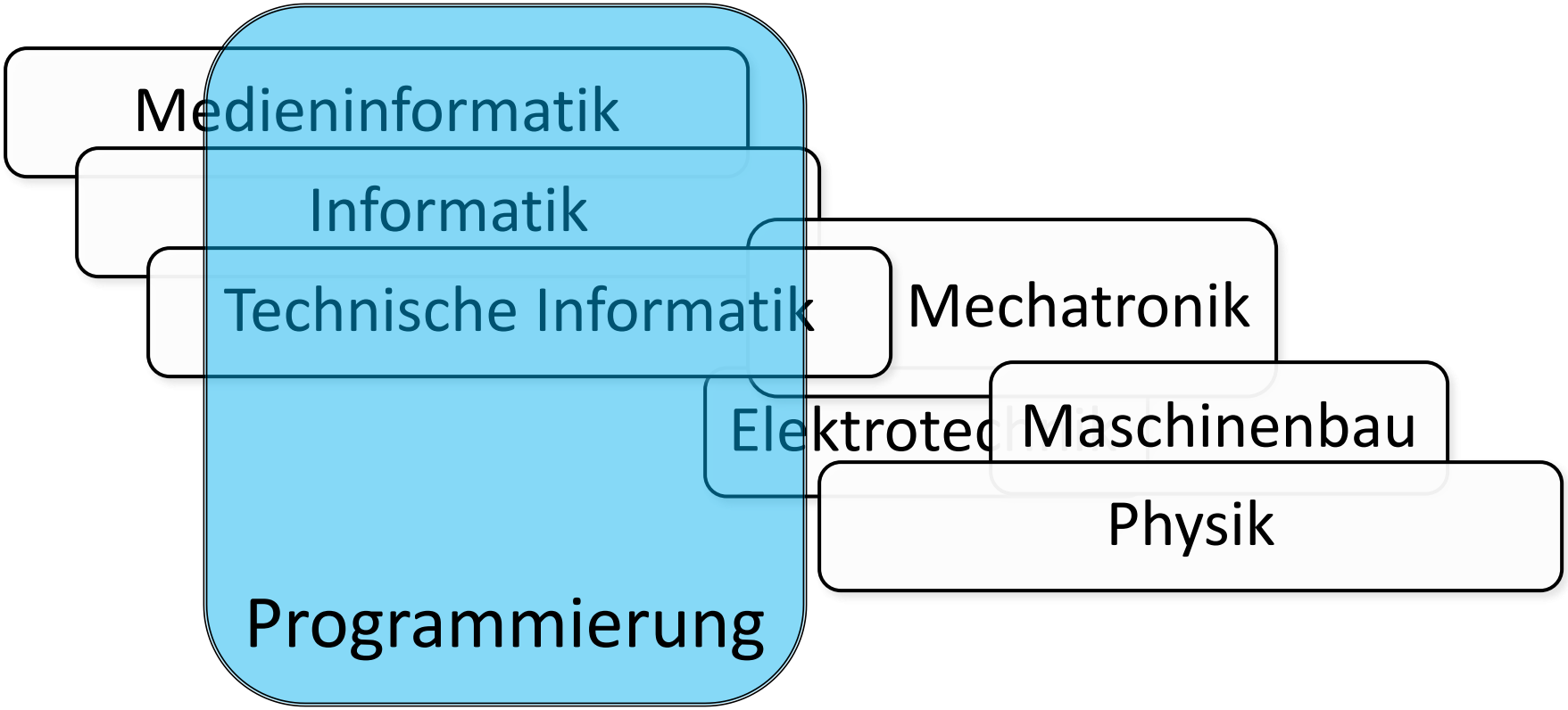
- Z. B. Gestaltung von Oberflächen, Ergonomie, Verknüpfung von Medien (z. B. GUI, Web-Oberflächen)

Was steckt dahinter (3/3)

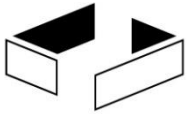


jede Technik nutzt die Ergebnisse der anderen Technik, vereinfacht (abstrahiert) diese, damit sie für komplexe Aufgaben nutzbar wird

Bedeutung der Programmierung



Abstraktionsebenen beim Programmieren



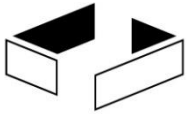
Video

Ebenen der Programmentwicklung

Abstraktion →

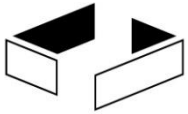
Programmbibliothek	Programme für Teillösungen
Hochsprachen	Programmstruktur, komplexe Schleifen, Unterprogramme
Assembler	elementare Anweisungen auf dem Speicher
Maschinensprache	Steueranweisungen für Hardware (Strom fließt [nicht])

Worum geht es bei der Programmierung



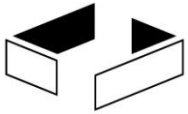
- Menschen haben Maschinen erfunden, um sich das Leben leichter zu machen und Geld zu verdienen
- Computer wurden entwickelt als flexible Maschinen, die Rechenaufgaben lösen und andere Maschinen steuern können
- Computer sind dumm, ihnen muss beigebracht werden, was sie wann zu tun haben
- Dieses „was wann“ heißt Computer-Programm
- Computer-Programme müssen entwickelt werden, dies ist ein Ziel dieser Veranstaltung
- Zentral für die Programmierung ist das Vorgehen zur Entwicklung einer Lösung, eines *Algorithmus*

Worum geht es in Programmen



- Es werden Informationen verwaltet, d. h. angelegt, gesucht und gelesen, verändert und gelöscht
- Es werden neue Ergebnisse berechnet, indem vorhandene Informationen genutzt werden
- Beispiel: Studierende in einer Studierendenverwaltung, Prognose der Studienbeiträge der nächsten Semester
- Informationen müssen in das System, z. B. Studierende in den Rechner
- gesucht ist ein Modell der Realität, so dass gewünschte Berechnungen möglich werden
- man kann relevante Daten des Studierenden-Modells festlegen (Vorname, Nachname, Geburtstag, Studiengang, …)
- man kann Hilfsdaten einführen: Matrikelnummer

Wie finde ich die Programmieraufgabe



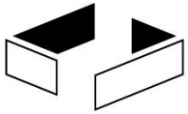
Video

Anforderungsanalyse

- systematische Analyse der Kundenherausforderungen
- Strukturierung der gewünschten Funktionalität in Hauptaufgaben
- Klärung der relevanten Informationen zur Modellierung
- Klärung vieler Randbedingungen
- genauer in der Informatik: siehe Vorlesung Objektorientierte Analyse und Design

- für diese Veranstaltung werden die Aufgabenstellungen als klar gegeben angenommen (große Vereinfachung)

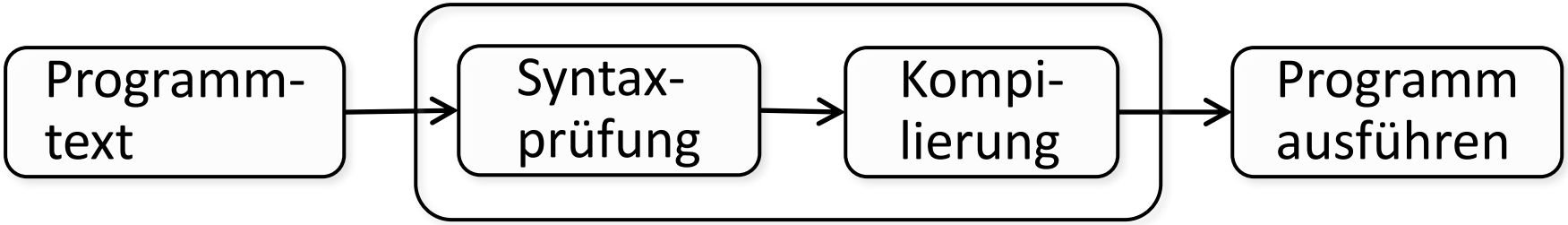
Wie fange ich die Programmierung an



- zunächst klären, ob Aufgabenstellung verstanden
- dann Entwicklungsplanung; typisch inkrementell
  - Aufteilung in Teilaufgaben
  - Für jede Teilaufgabe das erwünschte typische Verhalten realisieren (z. B. Daten eines neuen Studierenden eintragen und speichern)
  - Über alle möglichen Alternativen beim Verhalten nachdenken und diese schrittweise einbauen (z. B. Abbruch der Dateneintragung, Fehler beim Speichern)

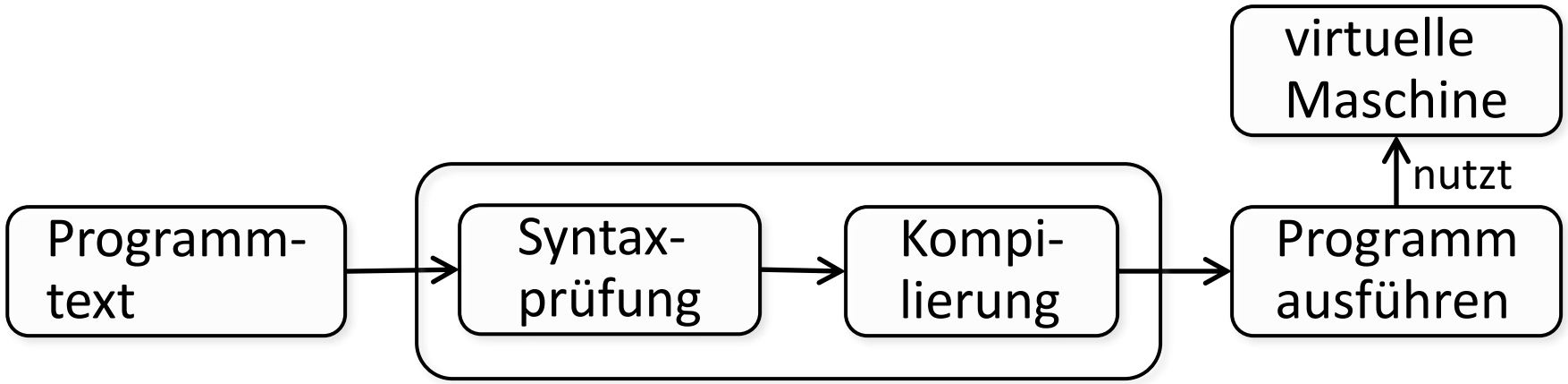
- Grundlage ist dabei ein Modell der gewünschten Daten, die für die Teilaufgabe benötigt werden
- Bei Datenmodellen spricht man oft von Objekten (Individuen mit konkreten Eigenschaften)

Vom Programmtext zum ausführbaren Programm



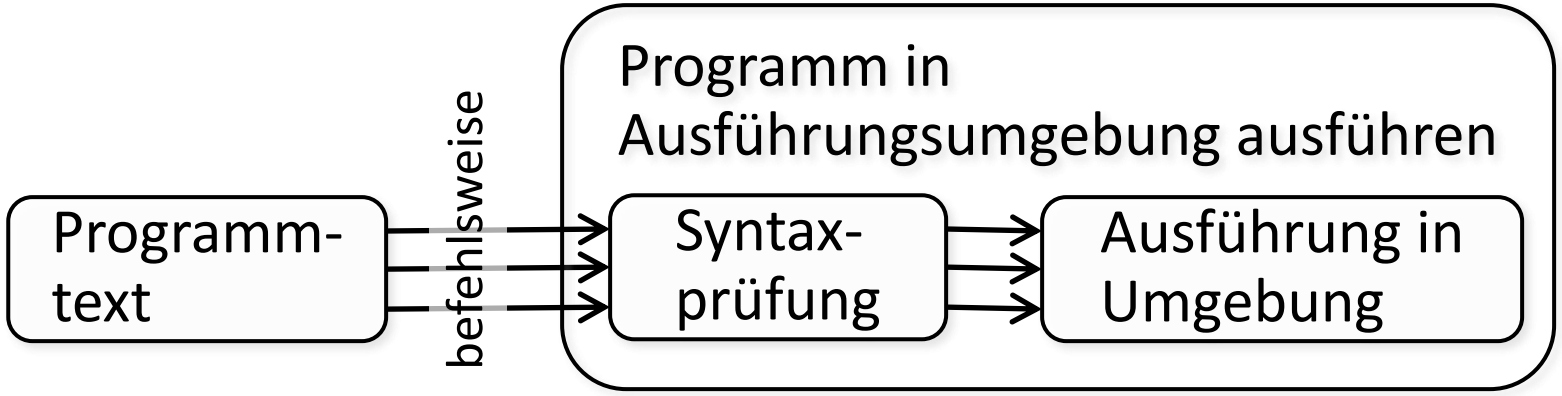
- Programm wird in Programmiersprache geschrieben, typischerweise in Software-Entwicklungsumgebung (SEU)
- Programm wird in vom Computer mit Kompiler in ausführbare Sprache (Maschinencode) übersetzt
- erster Schritt: Syntaxüberprüfung; ist Programm so geschrieben, dass es verarbeitet werden kann
- zweiter Schritt: Kompilierung; Programm wird in Maschinencode übersetzt
- es liegt ein ausführbares Programm vor, dass direkt auf dem Rechner gestartet werden kann

Variante auf dem Weg zum ausführbaren Programm



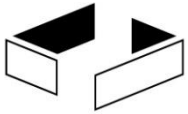
- Java und C# nutzen virtuelle Maschine (JRE, .Net)
- virtuelle Maschine muss auf dem Rechner installiert sein
- bei Kompilierung wird Byte-Code erzeugt, der mit virtueller Maschine ausführbar ist
- großer Vorteil: übersetztes Programm ist auf jedem Rechner mit virtueller Maschine ausführbar
- Vorteil: in virtueller Maschine können Programmiersprachen kombiniert werden, z. B. (Java, Groovy) (C#, Visual Basic)
- kleiner Nachteil: Programmstart langsamer

Variante auf dem Weg zum ausführbaren Programm



- keine Kompilierung, Programm wird zeilenweise ausgeführt, in Ausführungsumgebung ausgeführt
- pro Zeile Syntaxprüfung, dann Ausführung
- Vorteil: schnelle Erstellung und Ausführung
- Nachteile: Fehler werden erst im laufenden Programm sichtbar, langsamere Ausführung
- Beispiele: JavaScript (Umgebung: Browser), PHP, Ruby, Python ; generell Skriptsprachen

Syntax und Semantik von Programmiersprachen



- Syntax legt fest,
  - welche Sprachelemente und -konstrukte es gibt und
  - wie mit ihrer Hilfe korrekte Sätze in der Sprache formuliert werden können
  - Syntax = Menge von Regeln, die die Struktur von Programmen bestimmen
- Semantik einer Programmiersprache
  - legt die Bedeutung syntaktisch korrekter Sätze fest
  - legt fest, welche Wirkung jedes Sprachelement oder -konstrukt im Programmablauf hervorruft
  - Semantik = Menge von Verhaltensregeln, die die Funktionsweise von Programmen bestimmen

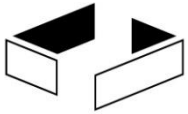
Beispiel

BlueJ

Video

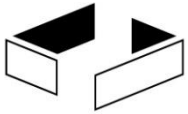
erste Befehle

Programmieraufgabe



- Aufgabe: Bestimme in etwa das Jahr in dem ein Eintritt in die Rente möglich sein könnte
- es werden Informationen benötigt
  - wie alt ist die Person?
  - welches Jahr haben wir?
  - mit welchem Alter Eintritt in die Rente möglich?
- Informationen müssen dem Computer bekannt gemacht werden
- Informationen werden in Variablen gespeichert
- Syntax: Variablen haben einfache Namen (Buchstaben am Anfang, keine Leerzeichen; Java-Regel erstes Zeichen klein)

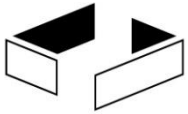
Deklaration einer Variable



```
int alter;
```

- Jede Variable hat in Java (C, C++, C#) einen Typen
- Syntax: <Typ> <Variablenname>;
- Java hat zwei Arten von Typen: elementare Datentypen (z. B. int, double, char, boolean) und Klassen (z. B. String, Integer, alles Selbstgeschriebene)
- durch die Deklaration wird dem Computer bekannt, dass wir eine Variable mit Namen "alter" vom Typ "int" nutzen wollen

Befehl (statement)

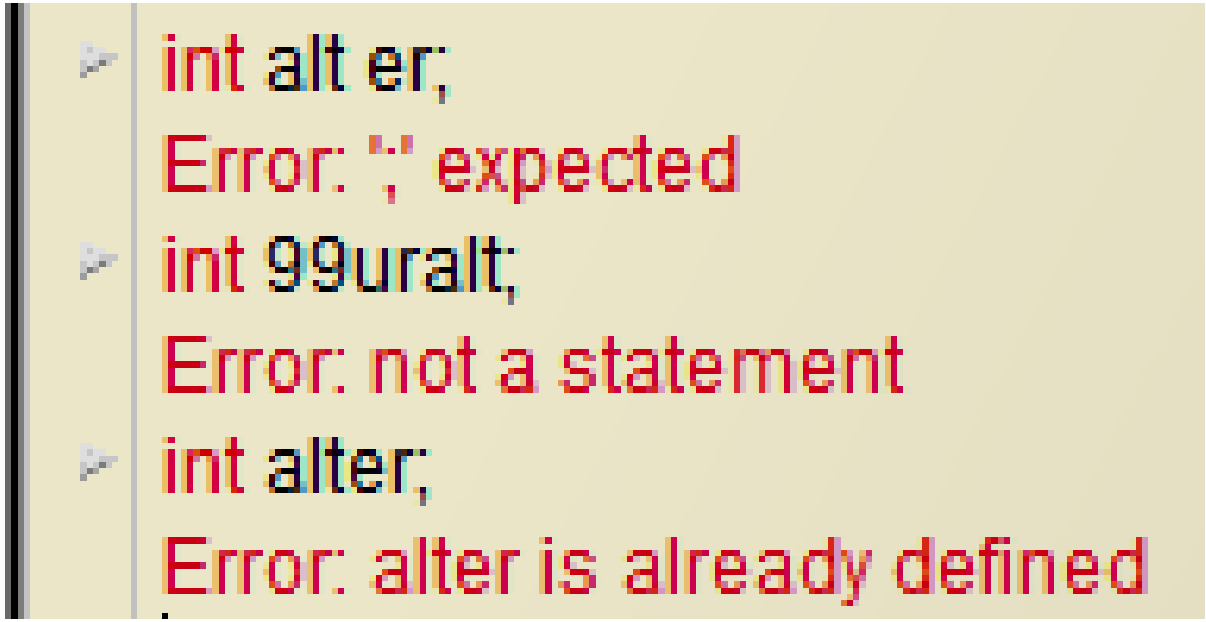


```
int alter;
```

- Syntax: <Programmzeile> gefolgt von Semikolon
- Befehl (oder Anweisung) beschreibt eine Aktion, die der Computer ausführen soll [wann, später!]
- SEU BlueJ erlaubt Skript-Ansatz mit Java in „Code Pad“ / Direkteingabe (Werkzeugnutzung in http://home.edvsz.hs-osnabrueck.de/skleuker/querschnittlich/BlueJUserManual.pdf beschrieben)
- außer Hinweis kein sichtbarer Effekt

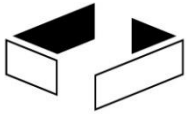
```
int alter;
Note: Codepad variables are automatically initialized
 in the same way as instance fields.
```

Syntax und Semantik beachten



- letzter Fall: jeder Variablenname kann (in einem Block [später!]) nur einmal vergeben werden
- wie bei Skripten üblich, wird nach einem Fehler einfach mit der nächsten Zeile weiter gearbeitet

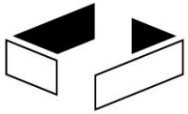
Zuweisung



```
alter = 44;
```

- einer Variablen wird ein (vorher berechneter) Wert zugewiesen; ist ein Befehl
- Syntax <Variable> = <Ausdruck>;
- Variable muss vorher deklariert sein
- Ausdruck [n. Folien] wird zuerst berechnet, muss vom Ergebnis zum Typ der Variablen passen
- Wert des Ausdrucks wird der Variablen zugewiesen; wird diese später genutzt, hat sie diesen Wert
- Abarbeitung: **Zuerst immer rechte Seite berechnen, dann der Variablen auf der linken Seite zuweisen**
- auf der linken Seite steht **immer** genau eine Variable

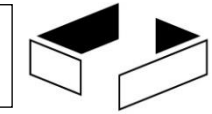
Ausdruck



```
alter + 1
```

- Ausdruck besteht aus Variablen und Konstanten (also konkreten Werten, wie 44)
- Ausdruck kann ausgerechnet werden, dazu werden die Werte der Variablen genutzt
- Ausdruck ist kein Befehl
- Ausdrücke kommen aber in vielen Befehlen an unterschiedlichen Stellen vor (z. B. als Teil einer Zuweisung)
- das Ergebnis eines Ausdrucks hat einen Typen (der für die weitere Bearbeitung interessant ist)

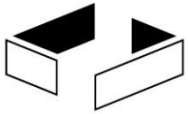
Ausdrücke in Code Pad



- Code Pad zeigt (nur für) Ausdrücke das Ergebnis an

```
alter = 44;
alter
44   (int)
alter+1
45   (int)
alter + alter * 2
132   (int)
(alter + alter) * 2
176   (int)
3+4*5
23   (int)
```

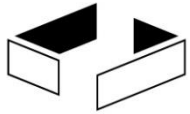
Zusammenfassung von Deklaration und Zuweisung



```
int alter = 44;
```

- zwei Befehle zusammengefasst zu einer Initialisierung
- grundsätzlich gute Idee: jede Variable wird zusammen mit ihrer Deklaration initialisiert
- ohne Initialisierung sind Variablenwerte entweder undefiniert oder erhalten Standardwert (Default-Wert) [abhängig wo die Variable deklariert wird]
- Code Pad: Variablen mit Zahlen-Typen haben am Anfang den Wert 0
- auf der rechten Seite kann wieder ein beliebiger (int-)Ausdruck stehen

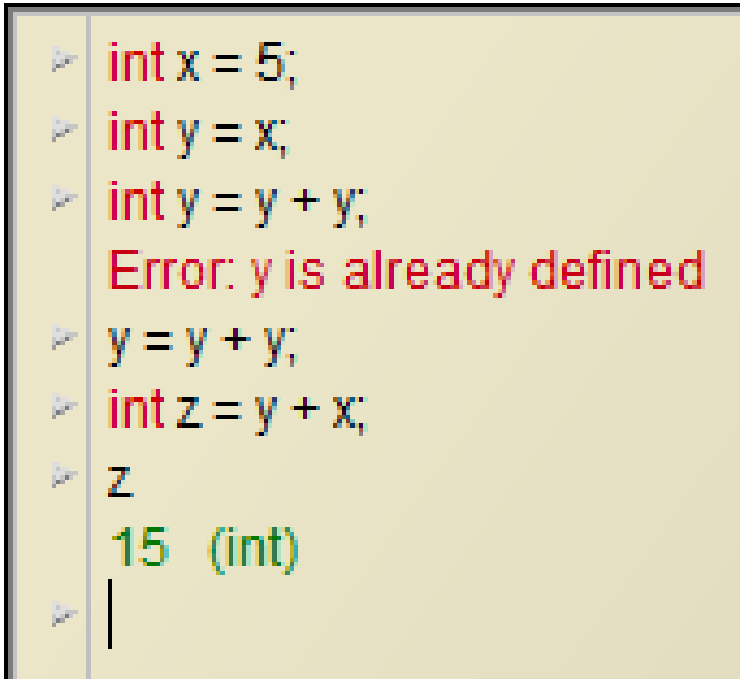
ein erstes Programm (genauer ein Skript)



```
int alter = 20;
int aktuellesJahr = 2018;
int renteneintritt = 67;
int ergebnis = aktuellesJahr + (renteneintritt - alter);
ergebnis
    2065   (int)
```

- Nachteil: nicht einfach wieder nutzbar mit anderem Alter
- auch andere Werte nicht änderbar
- man kann auch hier schon etwas andere Lösungen hinschreiben (Erkenntnis: es kann viele Programme geben, die das gleiche berechnen)

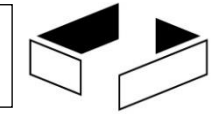
Zweites Programm (Skript) – Analysemöglichkeit



Variable Anweisung	x	y	z
int x = 5;	5		
int y = x;	5	5	
y = y + y;	5	10	
int z = y + x;	5	10	15

- eine leeres Feld bedeutet, dass die Variable noch nicht existiert
- Tabelle zeigt schrittweise Abarbeitung, Einträge nach Abarbeitung der Anweisung

Verallgemeinerung der Analyse



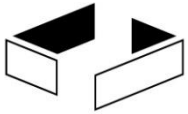
- bisher nur konkrete Werte genutzt
- Frage, was passiert, wenn man z. B. Anfangswert nicht genau kennt, z. B. x habe einen a genannten Wert

```
int x = a;
int y = x;
y = y + y;
int z = y + x;
```

Variable / Anweisung	x	y	z
int x = a;	a		
int y = x;	a	a	
y = y + y;	a	2a	
int z = y + x;	a	2a	3a

- leider so direkt mit nicht mit Code Pad realisierbar
- so gezeigt: z hat am Ende den dreifachen Wert von x nach erstem Schritt

Vom Skript in Code Pad zum Java-Programm (1/8)

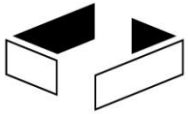


Video

- Hinweis: Folgende Folien sind ein kleiner Ausflug, wie man die ersten Programmideen systematischer verwalten kann
- Hinweis 2: alle in den folgenden Folien genutzten Begriffe werden danach systematisch erläutert
- Hinweis 3: einige Ideen, wie ein gutes Programm aussieht, müssen später erlernt werden
- Umgang mit der SEU wird in Praktika (und im Selbststudium) behandelt

- Ziel 1: Programm nicht immer wieder selbst eintippen
- Ziel 2: Programm flexibel gestalten, wie mit „a“ angedeutet

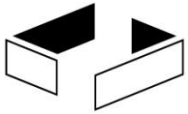
Vom Skript in Code Pad zum Java-Programm (2/8)



- erste Klasse

```
class Anfang {
    int rechne1(){
        int x = 5;
        int y = x;
        y = y + y;
        int z = y + x;
        return z;
    }
}
```

Vom Skript in Code Pad zum Java-Programm (3/8)



Schlüsselwort class

Klassenname (groß, ohne Leerzeichen)

geschweifte Klammern für Blockgrenzen, immer paarweise

bekanntes Programm (Befehle)

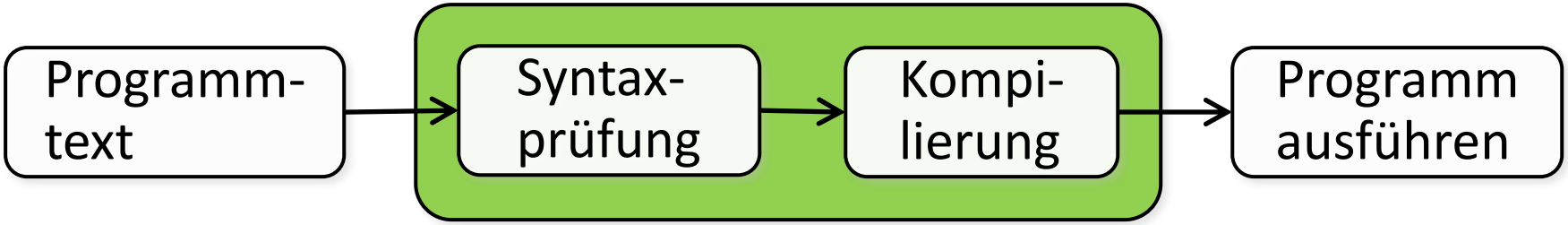
Methode: <Ergebnistyp> <Methodenname> (<Parameterliste>)

Schlüsselwort return, dahinter <Ausdruck> (welcher Wert ist Ergebnis, passt zu <Ergebnistyp>)

```
class Anfang {
    int rechne1(){
        int x = 5;
        int y = x;
        y = y + y;
        int z = y + x;
        return z;
    }
}
```

- Hinweis: eine Klasse selbst ist erstmal nicht ausführbar, sie beinhaltet nur das Programm (den Programm-Code)

Vom Skript in Code Pad zum Java-Programm (4/8)



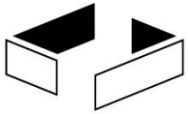
Anfang

Übersetzen | Rückgängig | Ausschneiden | Kopieren

```
class Anfang {
    int rechne1() {
        int x = 5;
        int y = x;
        y = y + y;
        int z = y + x;
        return z;
    }
}
```

Klasse übersetzt - keine Syntaxfehler

Vom Skript in Code Pad zum Java-Programm (5/8)



- Ausführung (eine Möglichkeit in Code Pad)

```
Anfang objekt;
Note: Codepad variables are
 in the same way as instance
objekt = new Anfang();
objekt.rechne1()
15   (int)
int ergebnis = objekt.rechne1();
ergebnis
15   (int)
```

jede Klasse ist neuer Typ
Variable von diesem Typ deklarieren

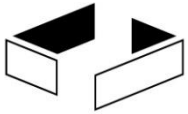
von Klassen werden Objekte mit dem Schlüsselwort new erzeugt

Ausdruck mit Methodenaufruf: <Objekt>.<Methode>

Methodenaufruf als Teil eines Befehls; einer Zuweisung

- Methoden (unsere bisherigen Programme) nur mit Hilfe von Objekten ausführbar

Vom Skript in Code Pad zum Java-Programm (6/8)



- Methode mit Parameter

Kommentar, beliebiger Text umgeben von /* und */

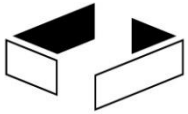
```
/* zweite Methode in der Klasse Anfang */
int rechne2(int a){
    int x = a;
    int y = x;
    y = y + y;
    int z = y + x;
    return z;
}
```

Parameterliste mit einem Parameter a
für Parameter werden immer Typen angegeben

Parameter wird wie Variable genutzt

- Parameter dienen dazu, konkrete Werte zu übergeben
- Variablen können in Methoden gleichen Namen, wie in anderen Methoden haben (keine Beziehung dadurch)

Vom Skript in Code Pad zum Java-Programm (7/8)



```
Anfang objekt = new Anfang();
objekt.rechne2(5)
15   (int)
int tmp = objekt.rechne2(5);
objekt.rechne2(tmp+2*5)
75   (int)
Anfang objekt2 = new Anfang();
objekt.rechne2(objekt2.rechne1())
45   (int)
```

Deklaration mit Initialisierung mit neuem Objekt

Methodenaufruf muss Wert für Parameter enthalten

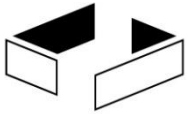
genauer steht hier Ausdruck mit Ergebnis vom Typ int

mehrere Objekte erzeugbar

eine mögliche Kombination von Methodenaufrufen

- Grundregel: Erst Berechnungen in Klammern durchführen

Vom Skript in Code Pad zum Java-Programm (8/8)



- Auswertung einer Methode (Parameter als Variable)

```
/* zweite Methode in der Klasse Anfang */
int rechne2(int a){
int x = a;
int y = x;
y = y + y;
int z = y + x;
return z;
}
```

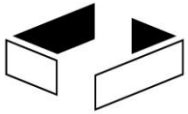
Variable Anweisung	a	x	y	z
rechne2(val)	val			
int x = a;	val	val		
int y= x;	val	val	val	
y = y + y;	val	val	2val	
int z = y + x;	val	val	2val	3val
return z	val	val	2val	3val

Aufgabe

Video

Objekt

Was heißt Objektorientierung

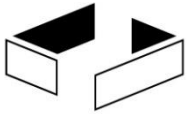


- in Programmen werden Daten verarbeitet und neue berechnet
- Daten gehören zu Objekten
- Objekte sind damit eine Grundstruktur von Programmen

Objekt:

- eindeutig einzeln identifizierbar
- besitzt konkrete Eigenschaften, die es charakterisieren
- !!! Objekte können die gleichen Eigenschaften haben und trotzdem unterschiedlich sein (dasselbe oder das Gleiche, Sie besuchen dieselbe Veranstaltung, Sie sitzen auf gleichen Stühlen)

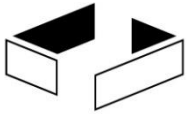
Objekte in der realen Welt mit Eigenschaften



- Ball: Farbe, Größe, Material
- Computer: Form, Prozessor, Graphikkarte, Festplatte, Hersteller, Seriennummer
- Mensch: Name, Geburtsort, Geburtsdatum, Wohnort, Personalausweisnummer
- Konto: besitzende Person, Wohnort der Person, Betrag, Kontoart, Bankleitzahl, Kontonummer

- Informatik nutzt Modelle der realen Welt; ein Informatik-Objekt beinhaltet für Aufgabenstellung relevante Daten

Objekt Adresse

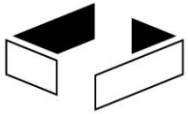


- jedes Haus hat eine eindeutige Adresse

Eigenschaften:	zwei Beispielobjekte:	
• Straße	Barbarastr.	Barbarastr.
• Hausnummer	16	16
• Postleitzahl	49076	79106
• Stadt	Osnabrück	Freiburg
• Bundesland	Niedersachsen	Baden-Württemberg

- es kann optionale Eigenschaften geben, bzw. Eigenschaften, die nur im bestimmten Kontext interessieren
- (Stockwerk)
- (Grundstücksgröße)

Objekt Datum



- Identifiziert einen eindeutigen Tag
- Eigenschaften: zwei Beispielobjekte:

• Tag	31	29
• Monat	12	2
• Jahr	1999	2100

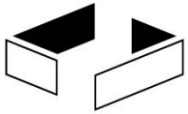
- Man spürt den Wunsch, die Gültigkeit von Objekten zu prüfen (-> später)

Objekt Person

- jede(r) ein Individuum

Eigenschaften:	zwei Beispielobjekte:	
• Vorname	Stephan	Eva
• Nachname	Dr. Kleuker	Mustermann
• Geburtsort	Wilhelmshaven	Wladiwostok
• Geburtsland	Deutschland	Russland
• Adresse		
• Straße	Barbarastr.	Barbarastr.
• Hausnummer	16	16
• Postleitzahl	49076	79106
• Stadt	Osnabrück	Freiburg
• Bundesland	Niedersachsen	Baden-Württemberg

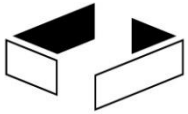
Objekt Studierend



- jede(r) ein Individuum

Eigenschaften:	zwei Beispielobjekte:	
• Vorname	Stephan	Eva
• Nachname	Kleuker	Mustermann
• Geburtsort	Wilhelmshaven	Wladiwostok
• Geburtsland	Deutschland	Russland
• Adresse	…	…
• eingeschrieben am	14.9.1986	29.7.2018
• Studiengang	Informatik	Informatik
• Matrikelnummer	368849	424142

Objekt Praktikum

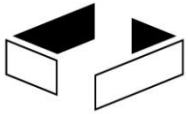


- individuell durch seine Eigenschaften

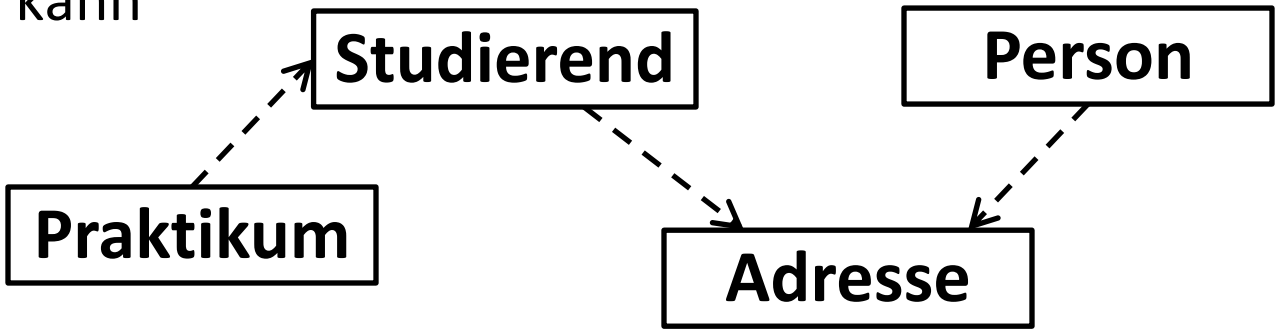
Eigenschaften:	Beispielobjekt:
• Modul	Programmierung 1
• Semester	Wintersemester 2018
• Veranstaltend	Stephan Kleuker
• Mitarbeitend	Ralf Koller
• Termine	[Mo 6:30-8:00, Fr 18:30-20:00]
• Teilnehmende	[Eva Mustermann (424142), Kevin Meier (424345), Jaqueline Schmidt (422323)]

- man erkennt, dass Eigenschaften mehrwertig sein können (Mehrzahl); mehrere einzelne Werte beschreiben Eigenschaft

Erkenntnisse über Objekte



- können große Anzahl von Eigenschaften haben
- relevante Eigenschaften hängen vom Anwendungsbereich ab (Erkennung durch Haarfarbe für Studierende irrelevant)
- Eigenschaften können optional sein
- Eigenschaften können wieder selbst Objekte sein
- kann unterschiedliche Objektarten mit vielen Gemeinsamkeiten geben
- Eigenschaft kann eine Sammlung von Werten sein, die leer sein kann

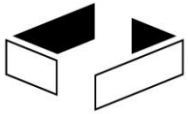


Beispiel

Video

Klasse

Der Begriff der Klasse



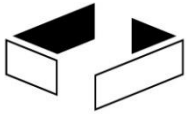
- Objekte haben durch ihre Eigenschaften gleichartige Struktur
- eine Klasse fasst diese Struktur zusammen und benennt die Eigenschaften

Adresse
strasse **hausnummer** **postleitzahl** **stadt** **bundesland**

Praktikum
modul **semester** **veranstaltend** **mitarbeitend** **termine** **teilnehmende**

- strasse ist eine Objektvariable (Name einer Eigenschaft, die für ein konkretes Objekt eine konkreten Wert annimmt)
- Objektvariable = Instanzvariable = Attribut

Wie finde ich die Objektvariablen

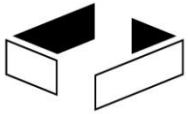


- Zentrale Teilaufgabe ist das Verstehen der Kundenwünsche
- Das Verfahren zum Finden der Wünsche heißt Anforderungsanalyse (Regel: garbage in, garbage out)
- Ansätze zur Anforderungsanalyse werden in der Vorlesung "Objektorientierte Analyse und Design" studiert

- Basierend auf Gesprächen, Texten, Erfahrungen, … werden u.a. die projektrelevanten Objekte und ihre Objektvariablen als ein Analyseergebnis bestimmt

- "Variable", da später veränderbar, z. B. Postleitzahl oder Nachname

Zusammenhang zwischen Klasse und Objekten

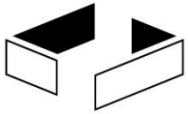


- Objekte haben eine Klasse als Grundlage
- Klassen sind der Rahmen (das Formular), das durch Objekte individuell gefüllt wird
- Objekte weisen den Objektvariablen konkrete Werte zu
- Darstellung <Objektname>:<Klasse>

ich:Adresse
strasse= „Barabarastr.“ **hausnummer= 16** **postleitzahl= 49076** **stadt= „Osnabrück“** **bundesland=„Nds“**

nichtIch:Adresse
strasse= „Barabarastr.“ **hausnummer= 16** **postleitzahl= 79106** **stadt= „Freiburg“** **bundesland=„B-W“**

Typen : int



- Variablen werden in Programmiersprachen oft Typen zugeordnet
- Ein Typ definiert, welche Werte erlaubt sind und wie diese Werte dargestellt werden (korrekte Syntax)
- Beispieltyp: int für ganze Zahlen
  - genauer: Wertebereich -2^{31} bis $2^{31}-1$
  - Folge von einzelnen Ziffern, beginnend mit einer Zahl ungleich 0 [stimmt nicht ganz]
- Man kann Variablen vom Typ deklarieren

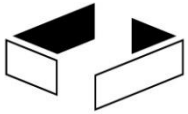
```
int tag
```

- Man kann Variablen erlaubte Werte zuweisen

```
tag = 30
```

- Hinweis: Gibt auch Programmiersprachen, die ohne direkte Angabe von Typen auskommen

Nutzung von Typen

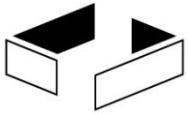


- JEDER Objektvariablen wird ein Typ zugeordnet
- Typen werden in der Klassendefinition festgehalten

```
class Datum {
    int tag;
    int monat;
    int jahr;
}
```

- Typ kann diskutabel sein (möchte man nicht doch lieber den Monatsnamen?)
- Für später: Jede Klasse definiert wieder einen Typen (damit ist Datum wieder ein Typ, der für Variablen genutzt werden kann)

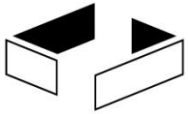
Klassen in Java (Syntax) (1/3)



```
class Datum {
    int tag;
    int monat;
    int jahr;
}
```

- Java hat Schlüsselworte (z. B. class), diese dürfen z. B. nicht als Variablennamen genutzt werden
- viele Programmfragmente stehen in geschweiften oder runden Klammern (immer beide Klammern eintippen, dann füllen)
- geschweifte Klammer am Anfang und am Ende der Klassendefinition
- Befehle enden mit einem Semikolon
- Objektvariable: <Typ> <Name>;

Klassen in Java (Syntax) (2/3)



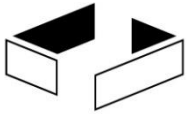
Variablen- und Klassennamen

- beginnen mit einem Groß- oder Kleinbuchstaben (keine Umlaute oder Sonderzeichen, auch wenn theoretisch möglich)
- werden gefolgt von beliebig vielen Groß- und Kleinbuchstaben, Ziffern oder _

keine Syntax, aber Konvention

- Klassennamen groß, typisch Einzahl (Studierend nicht Studierende, Konto nicht Konten)
- Objektvariablen klein, vollständige Worte, "sprechend"
- bei zusammengesetzten Namen werden zweite Worte direkt hinten an gefügt und erster Buchstabe großgeschrieben, z. B. eingeschriebenAm, richtig: starttermin falsch: startTermin
- eine Objektvariable pro Zeile deklarieren

Klassen in Java (Syntax) (3/3)



- Platzierung von Klammern

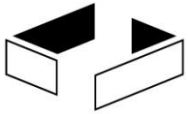
```
class Datum {
    int tag;
    int monat;
    int jahr;
}
```

- oder

```
class Datum
{
    int tag;
    int monat;
    int jahr;
}
```

- auf Folien obige Variante (spart eine Zeile)
- Einrückungen haben keine semantische Bedeutung

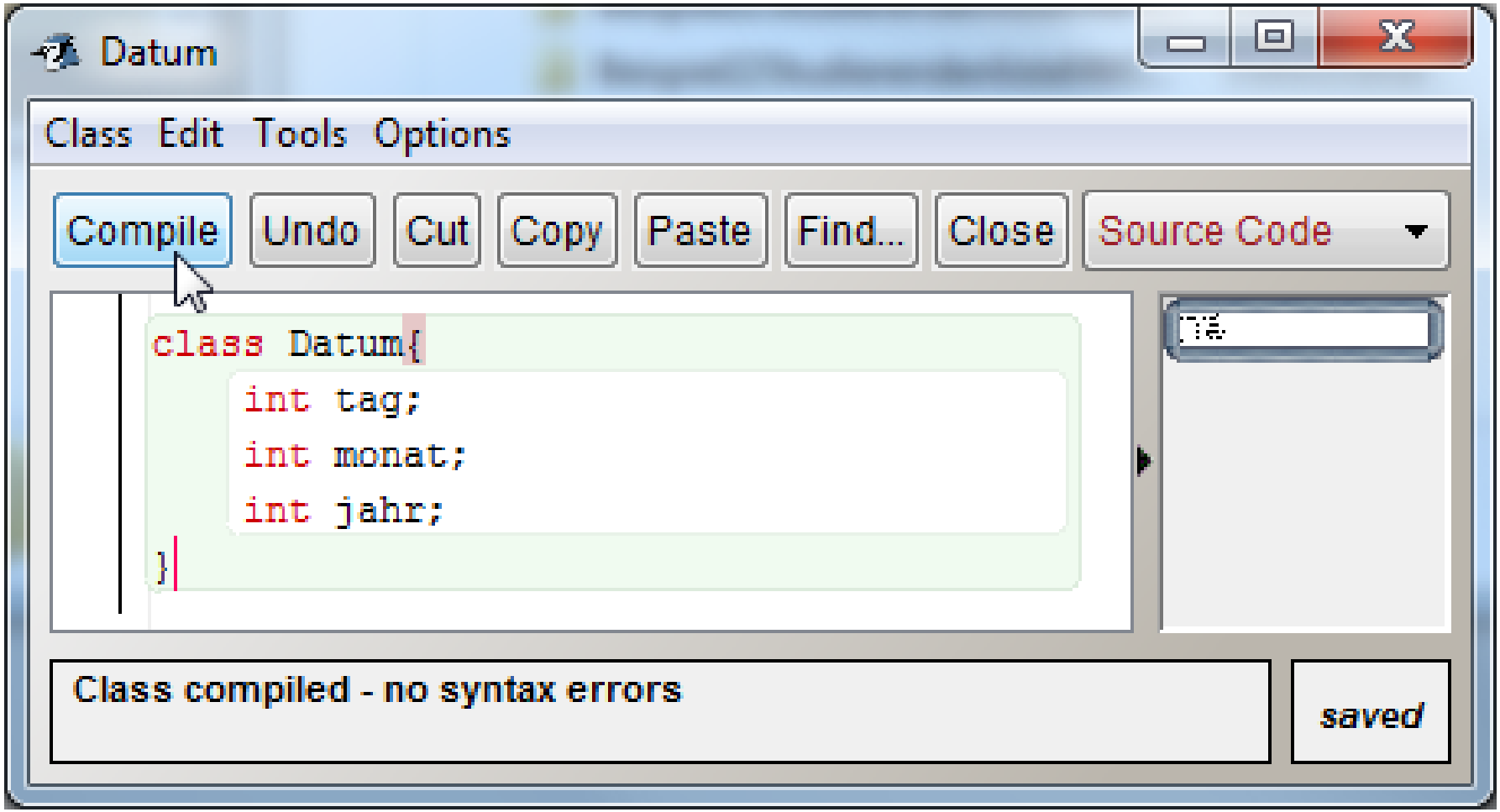
Syntaxprüfung durch Compiler (genauer Parser)



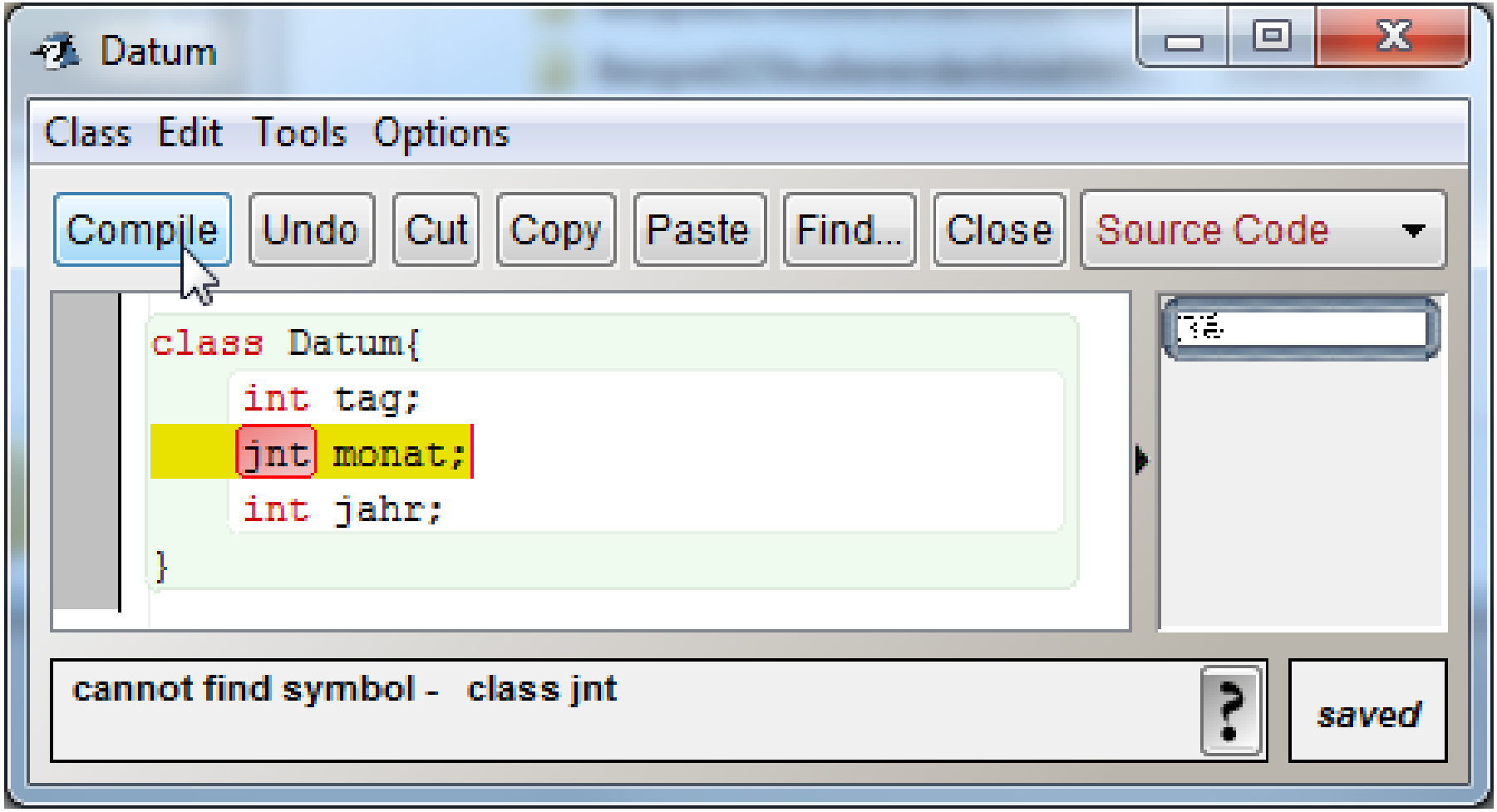
- Programmcode muss so übersetzt werden, dass der Computer Befehle verarbeiten kann (-> genauer später), Prozess heißt kompilieren (übersetzen)
- Vor dem Kompilieren (genauer: erster Schritt dabei), wird die Syntax des Programms geprüft (geparst)
- Bei fehlerhafter Syntax wird mehr oder minder genaue Fehlermeldung ausgegeben

- Kompilierung findet typischerweise in einer Entwicklungsumgebung statt (wir nutzen zunächst BlueJ)

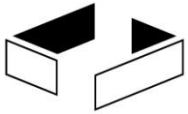
Erfolgreiche Kompilierung in BlueJ



Fehlerhafte Kompilierung in BlueJ

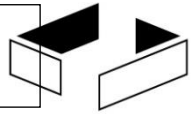


Woher kommen die Klassen und Typen



- Java bietet bereits viele Klassen als Typen an, in der sogenannten Klassenbibliothek
- Das Wissen über existierende Klassen ist Teil der Programmiererfahrung
- In der Programmierausbildung werden teilweise existierende Klassen neu programmiert um Erfahrungen zu sammeln
- Übersicht bietet u. a. Java Dokumentation (lokal herunterladbar) oder auch https://docs.oracle.com/en/java/javase/16/docs/api/index.html
- Woher die Objekte kommen bleibt noch unklar

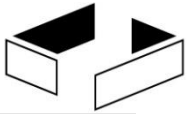
Wichtige elementare Typen und Klassen (Ausschnitt)



elementarer Typ	Nutzung
`int`	ganze Zahlen
`double (float)`	Fließkommazahlen
`boolean`	Wahrheitswerte, false, true
`long`	ganze Zahlen, größerer Zahlenbereich (mehr Speicher benötigt)

Klasse	Nutzung
`String`	Texte, Werte stehen in Hochkommata
`ArrayList<`Klasse`>`	Sammlung von beliebig vielen Objekten des Typs Klasse; nicht einfach hinschreibbar; merkt sich die Reihenfolge
`ArrayList<String>`	beliebig viele Texte, z. B. `["Hai", "Wo", "Da"]`

Einstieg Klasse String



Deklaration einer Variablen

Zuweisung, Nutzung eines Konstruktors

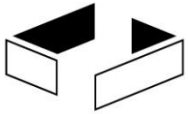
bei Strings erlaubte Kurz-schreibweise

Besonderheit: mit + werden Strings zusammengehängt (Ausdruck)

- * / haben bei Strings keinen Sinn

```
String s;
s = new String("Moin");
s
"Moin"   (String)
s = "Moin";
s
"Moin"   (String)
s + s
"MoinMoin"   (String)
s - s
Error: bad operand types for binary operator '-'
  first type:  java.lang.String
  second type: java.lang.String
s = s + " " + s;
s
"Moin Moin"   (String)
```

Ordnung in der Klassenbibliothek

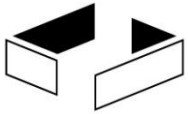


- ähnlich wie große Dateimengen in Ordnern und Unterordnern strukturiert werden, werden auch Klassen in Ordnerstrukturen thematisch abgelegt
- Ordner heißen in Java Pakete
- Beispiel: Die Klasse **`ArrayList`** liegt im Paket **`java.util`**
- Damit Klasse aus Paket genutzt werden kann, muss die Klasse im Quellcode bekannt gemacht werden, vor der Klasse steht dann

```
import java.util.ArrayList
```

- andere Basisklassen wie String liegen in **`java.lang`** (da häufig genutzt, müssen die Klassen nicht importiert werden)

Erstellung der Klasse Adresse (1/2)



ich:Adresse
strasse= „Barabarastr.“ **hausnummer= 16** **postleitzahl= 49076** **stadt= „Osnabrück“** **bundesland=„Nds“**

Beispielobjekt

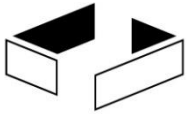
- jeweils genau einen Typ überlegen
- Ist hausnummer vom Typ int?

  Nein, Nummern wie 16a wären syntaktisch nicht korrekt
- Ist Postleitzahl vom Typ int?

  Nein, führende Nullen nicht möglich

  04356 Leipzig
- Fazit: Bild links ist falsch

Erstellung der Klasse Adresse (2/2)



Adresse

Class Edit Tools Options

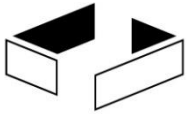
Compile | Undo | Cut | Copy | Paste | Find... | Close | Source Code

```
class Adresse{
    String strasse;
    String hausnummer;
    String postleitzahl;
    String stadt;
    String bundesland;
}
```

Class compiled - no syntax errors

saved

Klasse Studierend



Studierend - BeispielStudierendErsterKonstruktor

Klasse Bearbeiten Werkzeuge Optionen

Studierend

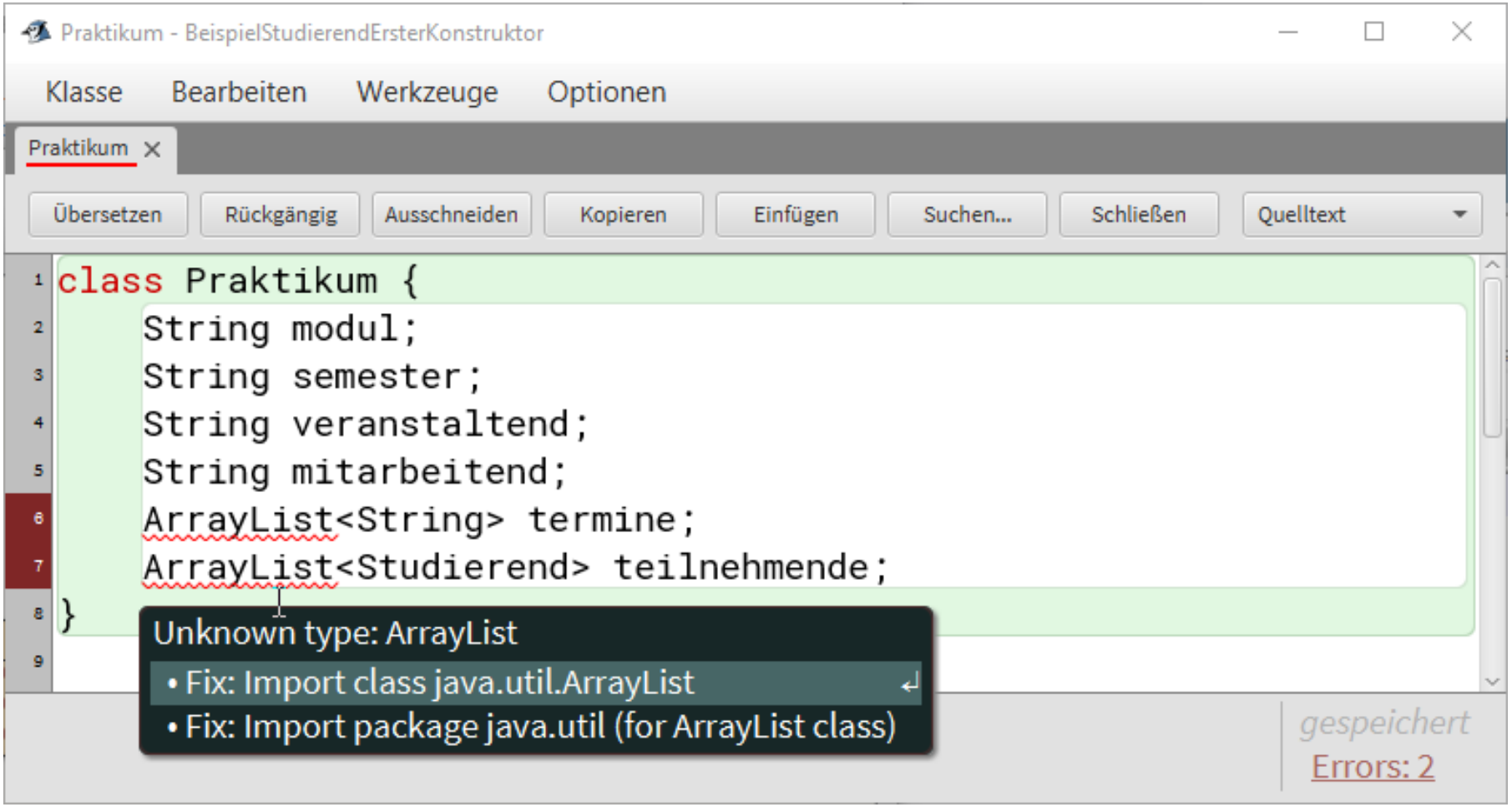
Übersetzen Rückgängig Ausschneiden Kopieren Einfügen Suchen... Schließen Quelltext

```
1  class Studierend{
2      String vorname;
3      String nachname;
4      int geburtsjahr;
5      String geburtsland;
6      Adresse adresse;
7      Datum eingeschriebenAm;
8      String studiengang;
9      int matrikelnummer;
10 }
```

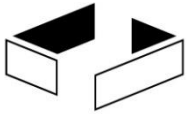
Klasse übersetzt - keine Syntaxfehler

gespeichert

fehlerhafte Klasse Praktikum



korrigierte Klasse Praktikum



Praktikum - BeispielStudierendErsterKonstruktor

Klasse Bearbeiten Werkzeuge Optionen

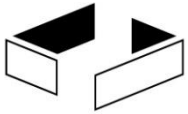
Praktikum

Übersetzen Rückgängig Ausschneiden Kopieren Einfügen Suchen... Schließen Quelltext

```
1  import java.util.ArrayList;
2
3  class Praktikum {
4      String modul;
5      String semester;
6      String veranstaltend;
7      String mitarbeitend;
8      ArrayList<String> termine;
9      ArrayList<Studierend> teilnehmende;
10 }
11
```

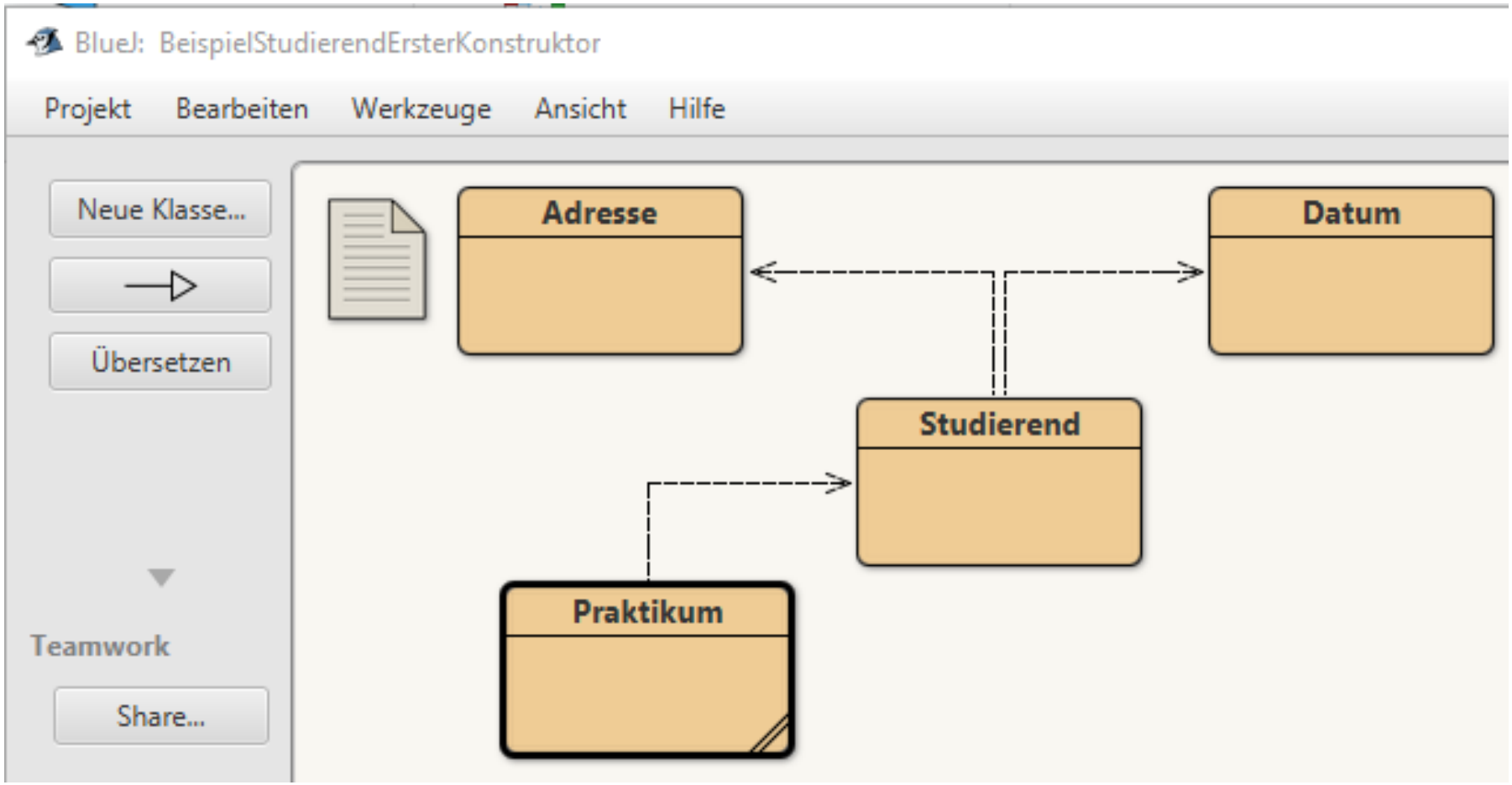
Klasse übersetzt - keine Syntaxfehler gespeichert

Seltener genutzte Variante ohne import

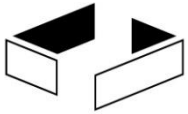


```
class Praktikum {
      String modul;
      String semester;
      String veranstalter;
      String mitarbeiter;
      java.util.ArrayList<String> termine;
      java.util.ArrayList<Studierend> teilnehmer;
}
```

Projektübersicht in BlueJ mit „nutzt“-Beziehung



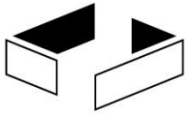
Beispielmodellierung



Video

- Aufgabe: Zu entwickeln ist eine Software mit der man Kassen-Bons verwalten kann. Jeder Bon wird damit ein Objekt sein. Es stellt sich die Frage nach Objektvariablen und Ihren Typen.
- Ansatz: Analysiere einen Bon, bestimme seine Eigenschaften, entweder von außen nach innen (Bon und dann seine Details) oder innen nach außen (erst Details und dann daraus den Bon zusammensetzen)
- Hinweis: Beide Ansätze ok, typischerweise iterativer Prozess

Bon (1/4) – erste Objektvariablen

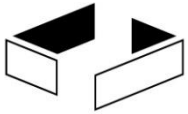


- unternehmen (String)
- adresse (Adresse)
- datum (Datum)
- uhrzeit (String)
- produkte (??? gleich)

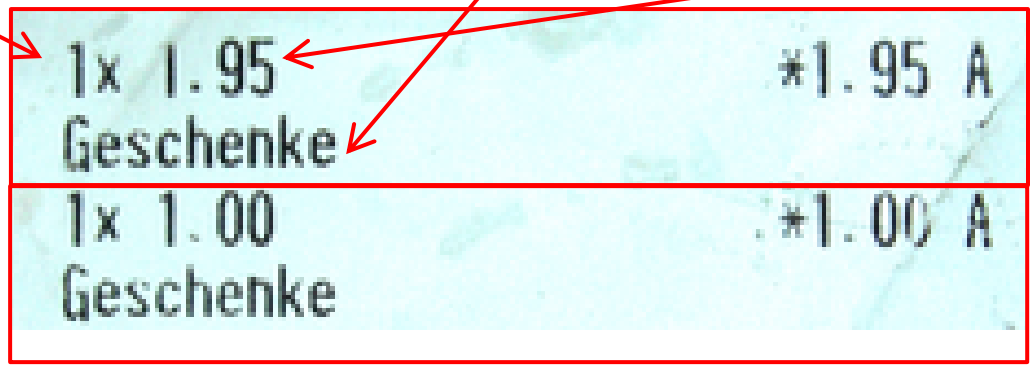
diskutabel

- mehrwertsteuer (gilt für viele Bons)
- summe (da aus produkte berechenbar)
- blabla (String, nicht von Interesse)

Bon (2/4) – Produkte genauer

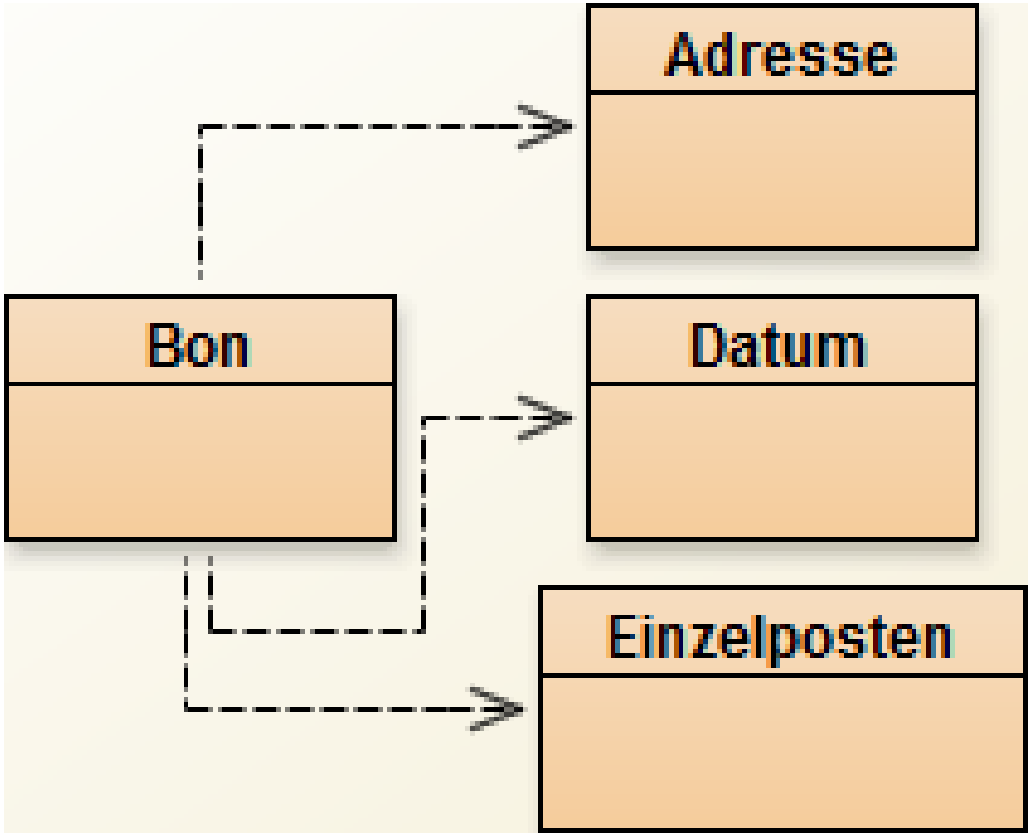


- ist ein Sammlung von zwei Zeilen
- jede Zeile enthält einen Einzelposten bestehend aus
- anzahl (int) produktname (String) einzelpreis (double)

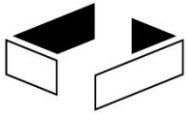


- gesamtpreis (wird aber berechnet)
- Ansatz: Einzelposten wird Klasse, produkte wird Sammlung von Einzelposten

Bon (3/4) – entstandene Klassen



Bon (4/4) – neue Klassen genauer



```
import java.util.ArrayList;
class Bon {
    String unternehmen;
    Adresse adresse;
    Datum datum;
    String uhrzeit;
    ArrayList<Einzelposten> produkte;
}
```

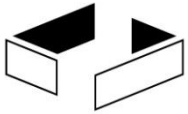
```
class Einzelposten {
    int anzahl;
    String produktname;
    double einzelpreis;
}
```

Beispiel

Video

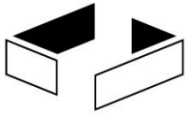
Konstruktor

Wie erstellt man Objekte



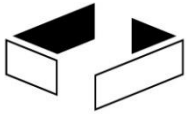
- mit den bisher erstellten Code kann man genau genommen nichts anfangen, da Klassen nur Rahmen sind
- zur Erzeugung eines Objektes muss es die Möglichkeit geben zu sagen: "Erzeuge mir ein Objekt dieser Klasse" (eine Möglichkeit im Code Pad kennen wir)
- gut wäre: zusätzlich mitteilen, welche Werte die Objektvariablen annehmen sollen
- Lösung heißt Konstruktor

Beispiel eines Konstruktors



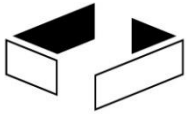
```
class Datum{
    int tag;
    int monat;
    int jahr;

    Datum (int anInt1, int anInt2, int anInt3){
        this.tag = anInt1;
        this.monat = anInt2;
        this.jahr = anInt3;
    }
}
```

Aufbau eines Konstruktors



- hat exakt gleichen Namen wie Klasse
- hat immer Parameterliste in runden Klammern
- Parameterliste besteht aus mit kommaseparierten Parametern
- Parameter hat Aufbau: <Typ> <Parametername>
- im Konstruktorrumpf steht ein kleines Programm mit Zuweisungen
- Zuweisung: <Variable> = <Ausdruck>
- Der Variablen auf der linken Seite wird der Wert der Berechnung des Ausdrucks der rechten Seite zugewiesen
- Zugriff auf Objektvariablen über **`this.`**<Variablenname>
- this ist "dieses Objekt", aus Objektsicht "ich"

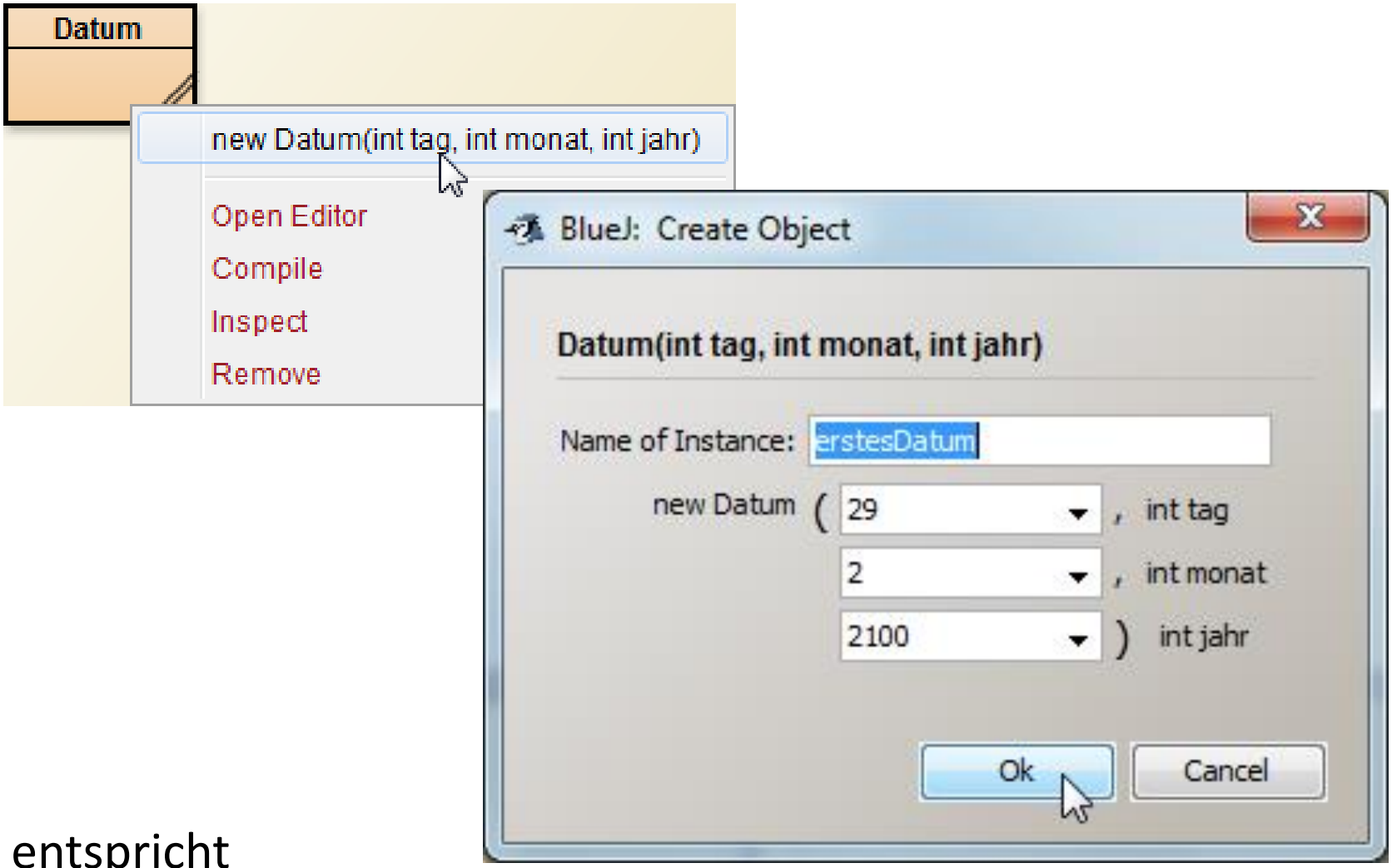
Parameter können beliebige Namen haben



```
Datum (int tag, int monat, int jahr){
    this.tag = tag;
    this.monat = monat;
    this.jahr = jahr;
}
```

- wenn Parameter gleiche Namen wie Objektvariablen haben, ist ihr Sinn aus der Signatur (erste Zeile des Konstruktors) ablesbar
- mit this wird der Bezug auf das gerade betrachtete Objekt deutlich
- Syntaxregeln für Variablennamen wieder beachten

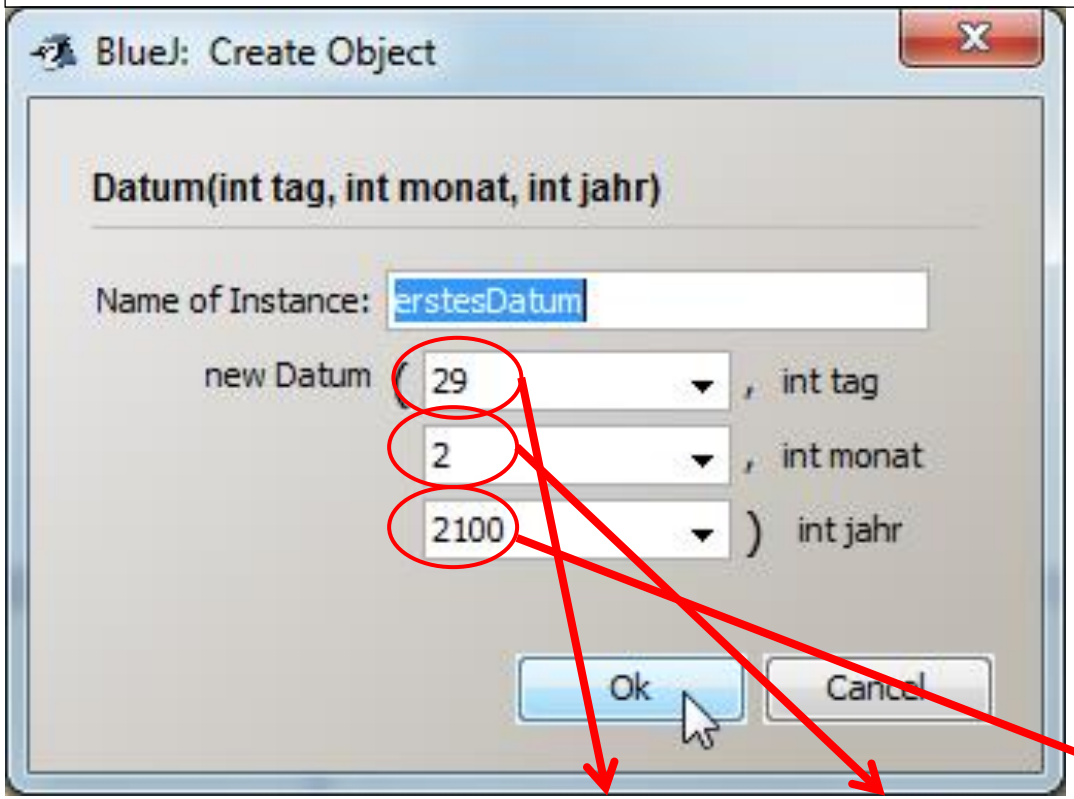
Nutzung eines Konstruktors direkt in BlueJ



entspricht

```
Datum erstesDatum = new Datum(29,2,2100);
```

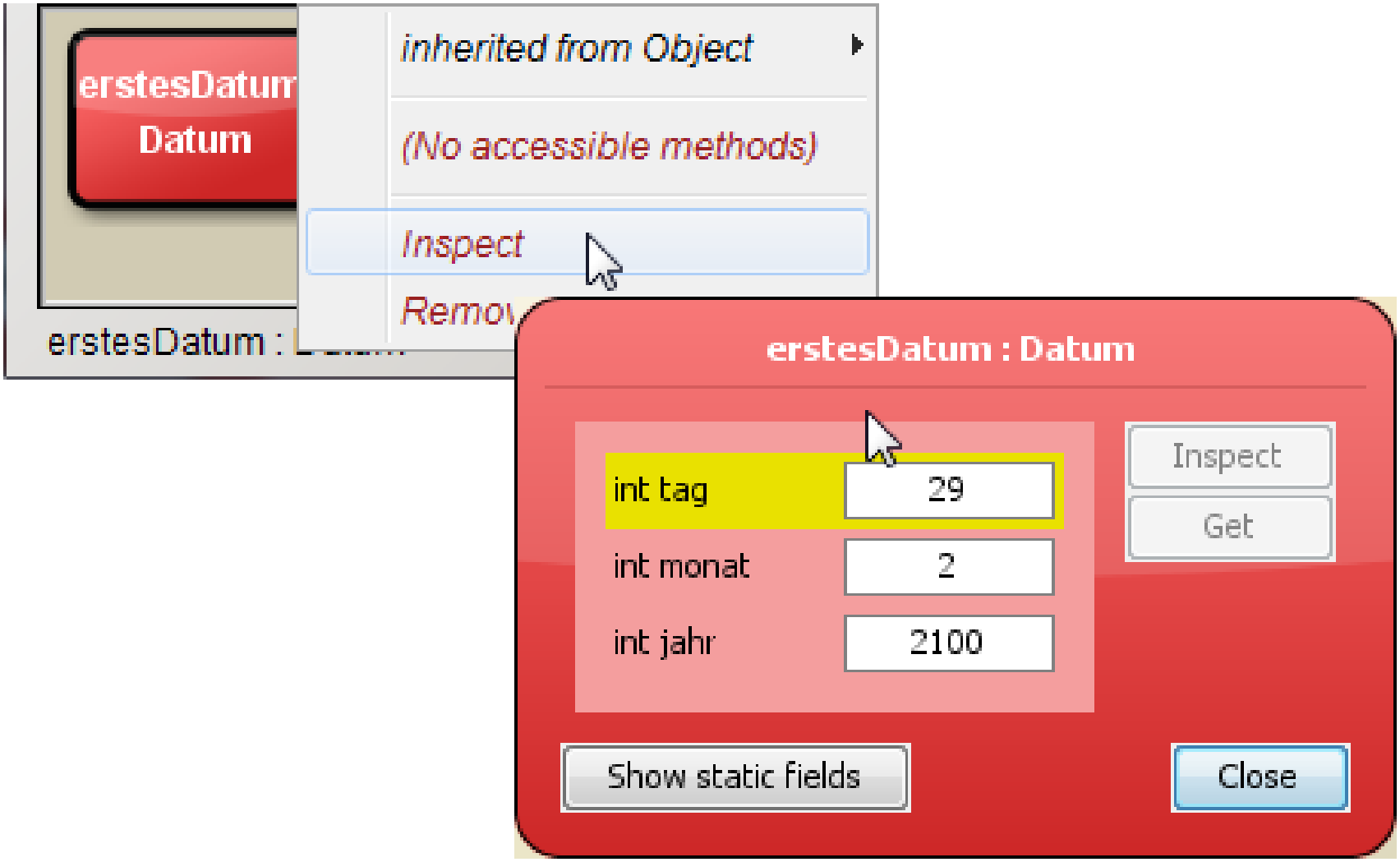
Vorstellung bei der Objektübergabe



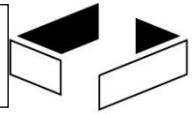
29 2 2100

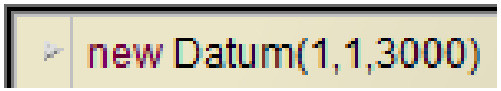
```
Datum (int tag, int monat, int jahr){
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
    }
```

Analyse eines Objekts in BlueJ

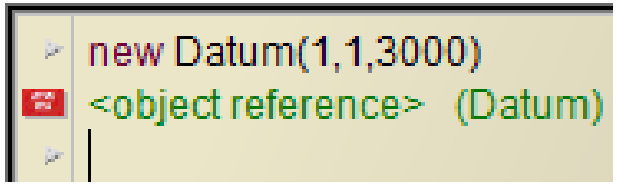


Erstellung eines Objekts im Code Pad – Variante 1/3

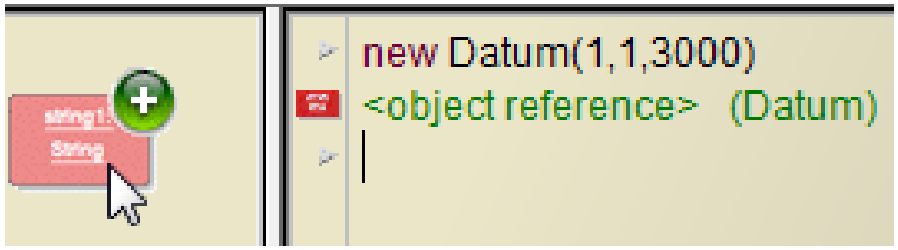




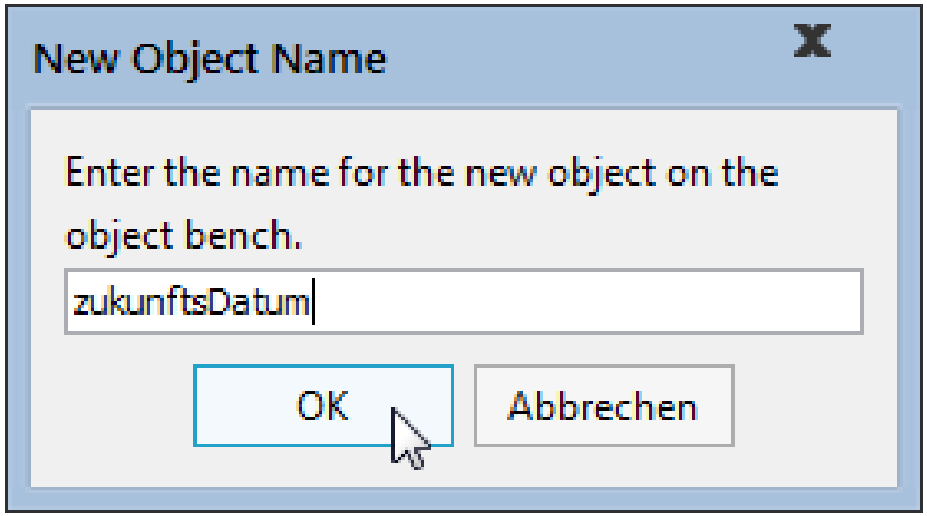
<Return gedrückt>



kleines Objekt am Rand

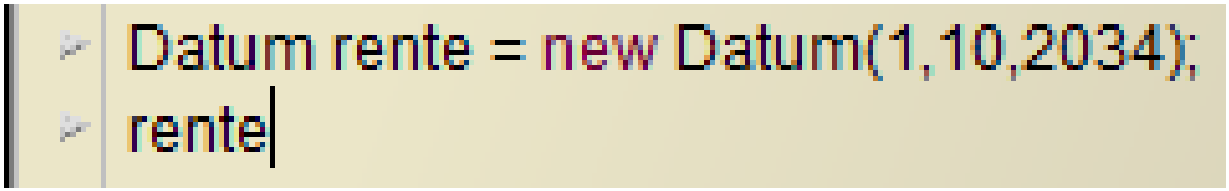


kleines rotes Objekt am Rand auf Objektleiste ziehen

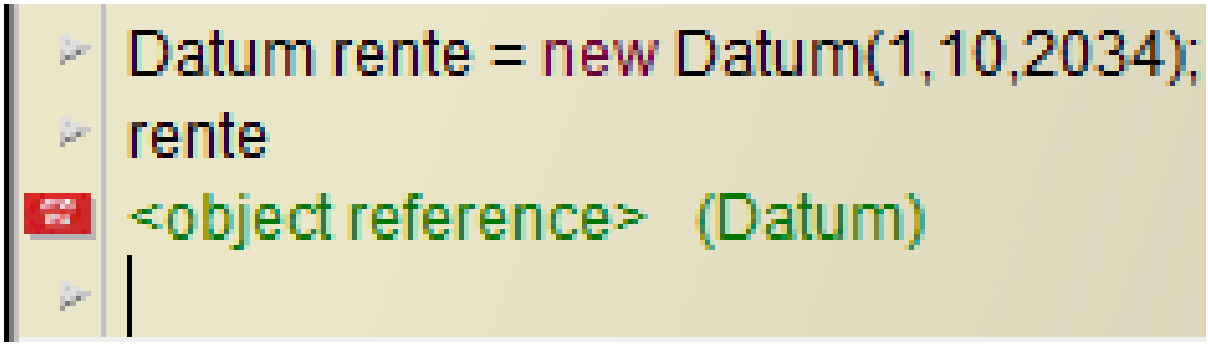


Objekt benennen und wie vorher analysieren

Erstellung eines Objekts im Code Pad – Variante 2/3

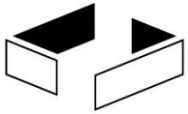


- Anlegen einer lokalen Variable namens rente (wieder mit Typ)
- Zeilenende mit Semikolon, mit Return in nächste Zeile
- Objektname ohne Semikolon als Ausdruck eintippen (Return)



- gefordertes Objekt wird rechts am Rand angezeigt
- Bearbeitung wie vorher

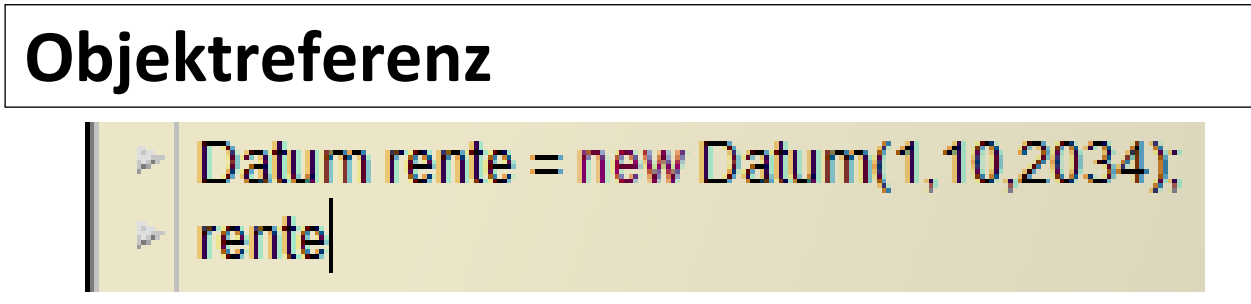
Klasse kann mehrere Konstruktoren haben



```
Datum (int tag, int monat, int jahr){
    this.tag = tag;
    this.monat = monat;
    this.jahr = jahr;
}

Datum (int jahr){
    this.tag = 1;
    this.monat = 1;
    this.jahr = jahr;
}
```

- Konstruktoren müssen nur andere Parameterlisten (d. h. Typen, nur verschiedene Variablennamen reichen nicht) haben

Objektreferenz



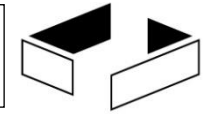
- Eine Variable eines bestimmten Typs enthält eine Referenz auf ein Objekt dieses Typs, das im Speicher liegt
- Objektvariablen sind auch Variablen und referenzieren wieder andere Objekte
- Es gibt Basistypen von Objekten, die anschaulich keine weiteren Referenzen enthalten (z. B. Typen int, String)

Variablen Speicher

rente	Datum

Datum	tag	int	1
	monat	int	10
	jahr	int	2034

Speicherdiagramm genauer



Referenz

rente	Datum

Datum			
	tag	int	1
	monat	int	10
	jahr	int	2034

Dreiteilung

- Name der Variablen

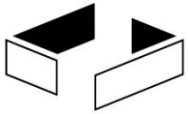
(im Speicher)

- Typ
- Wert oder Referenz

- Klassenname
- Dreiteilung pro Objektvariable

(Name, Typ, Wert)

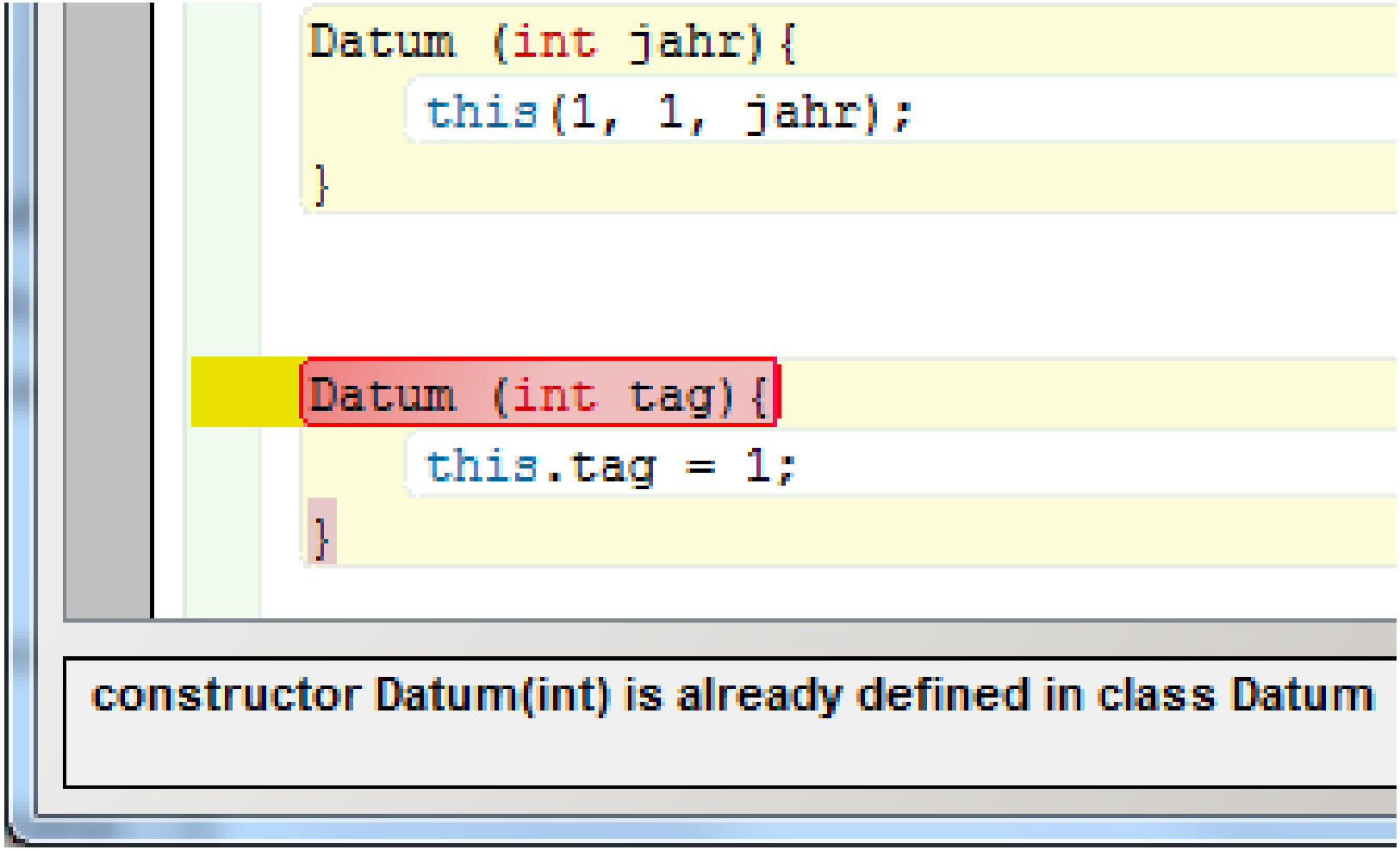
Konstruktoren nutzen Konstruktoren



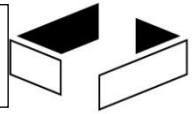
- nulltes Informatikgebot: Schreibe niemals Code doppelt
- Konstruktor kann anderen Konstruktor nur in erster Zeile aufrufen, Variante des vorherigen Datum(int jahr)-Konstruktors:

```
Datum (int tag, int monat, int jahr){
    this.tag = tag;
    this.monat = monat;
    this.jahr = jahr;
}

Datum (int jahr){
    this(1, 1, jahr); // ruft obigen Konstruktor auf
}
```

gleiche Parameter(typ)liste ist Fehler



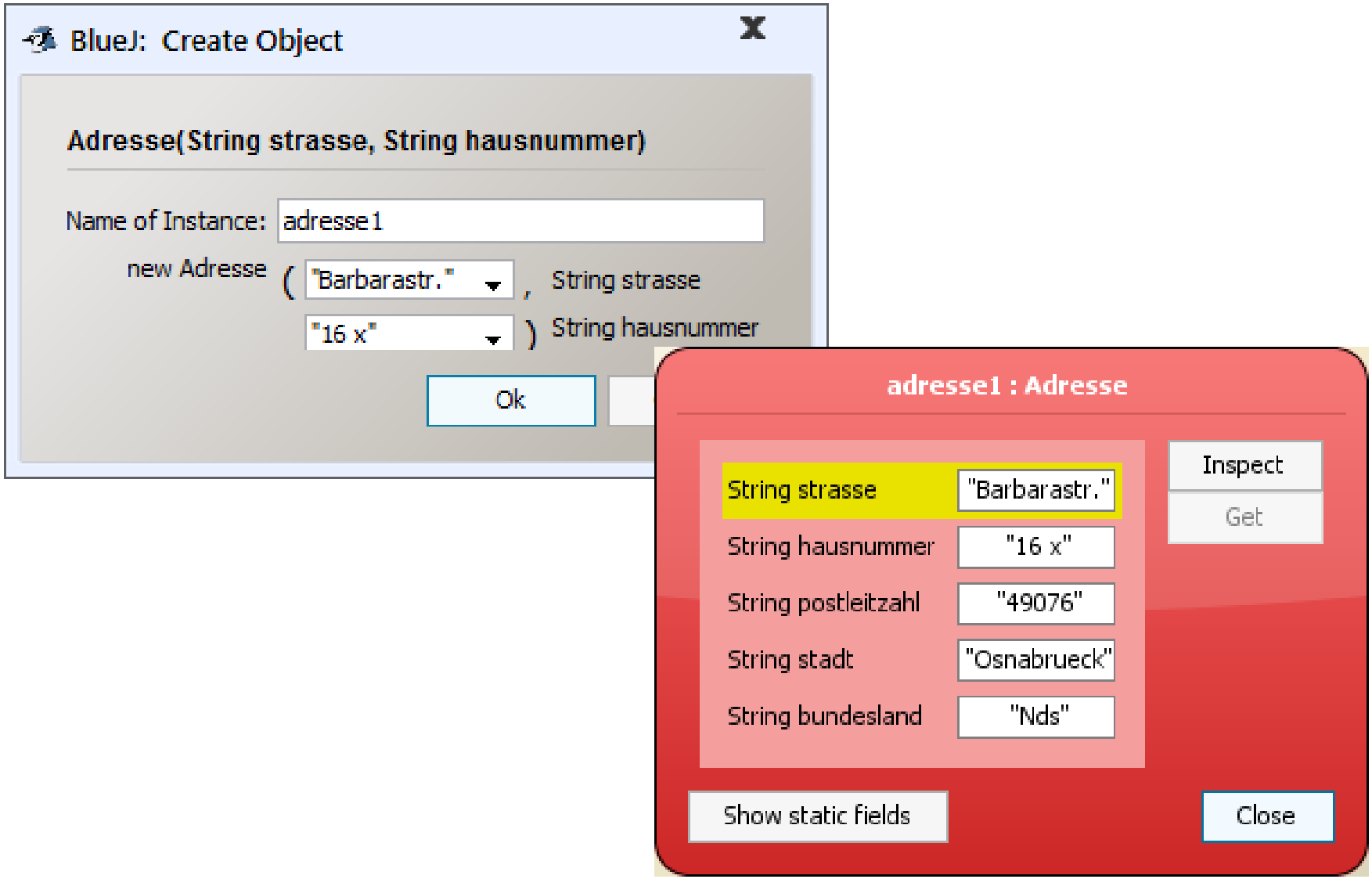
Variante: Startwerte bei Deklaration zuweisen (1/2)



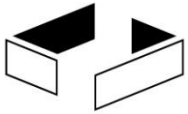
Video

```
class Adresse{
    String strasse;
    String hausnummer;
    String postleitzahl = "49076";
    String stadt = "Osnabrueck";
    String bundesland = "Nds";

    Adresse(String strasse, String hausnummer){
        this.strasse = strasse;
        this.hausnummer = hausnummer;
    }
}
```

Variante: Startwerte bei Deklaration zuweisen (2/2)



Klasse ohne Konstruktor (gibt es nicht)



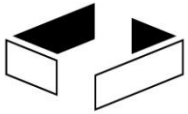
- Wird kein Konstruktor für eine Klasse X angegeben, existiert automatisch (nicht sichtbar) der Default-Konstruktor (Erinnerung an erste Konstruktornutzung)

```
X() {
}
```

- Alle Objektvariablen haben Standardwert oder Wert aus sofortiger Zuweisung bei der Deklaration

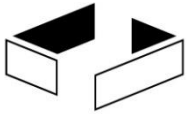
- Grundregel der sauberen Programmierung: geben Sie mindestens einen Konstruktor an (schreiben Sie zumindest den Default-Konstruktor hin)
- Wenn man Konstruktor hinschreibt, gibt es den automatischen Default-Konstruktor nicht!

ein sehr langer Konstruktor (unüblich)



```
Studierend(String vorname, String nachname
        , String geburtsort
        , String geburtsland
        , String strasse, String hausnummer
        , int tag, int monat, int jahr
        , String studiengang, int matrikelnummer) {
    this.vorname = vorname;
    this.nachname = nachname;
    this.geburtsort = geburtsort;
    this.geburtsland = geburtsland;
    this.adresse = new Adresse(strasse, hausnummer);
    this.eingeschriebenAm = new Datum(tag, monat, jahr);
    this.studiengang = studiengang;
    this.matrikelnummer = matrikelnummer;
}
```

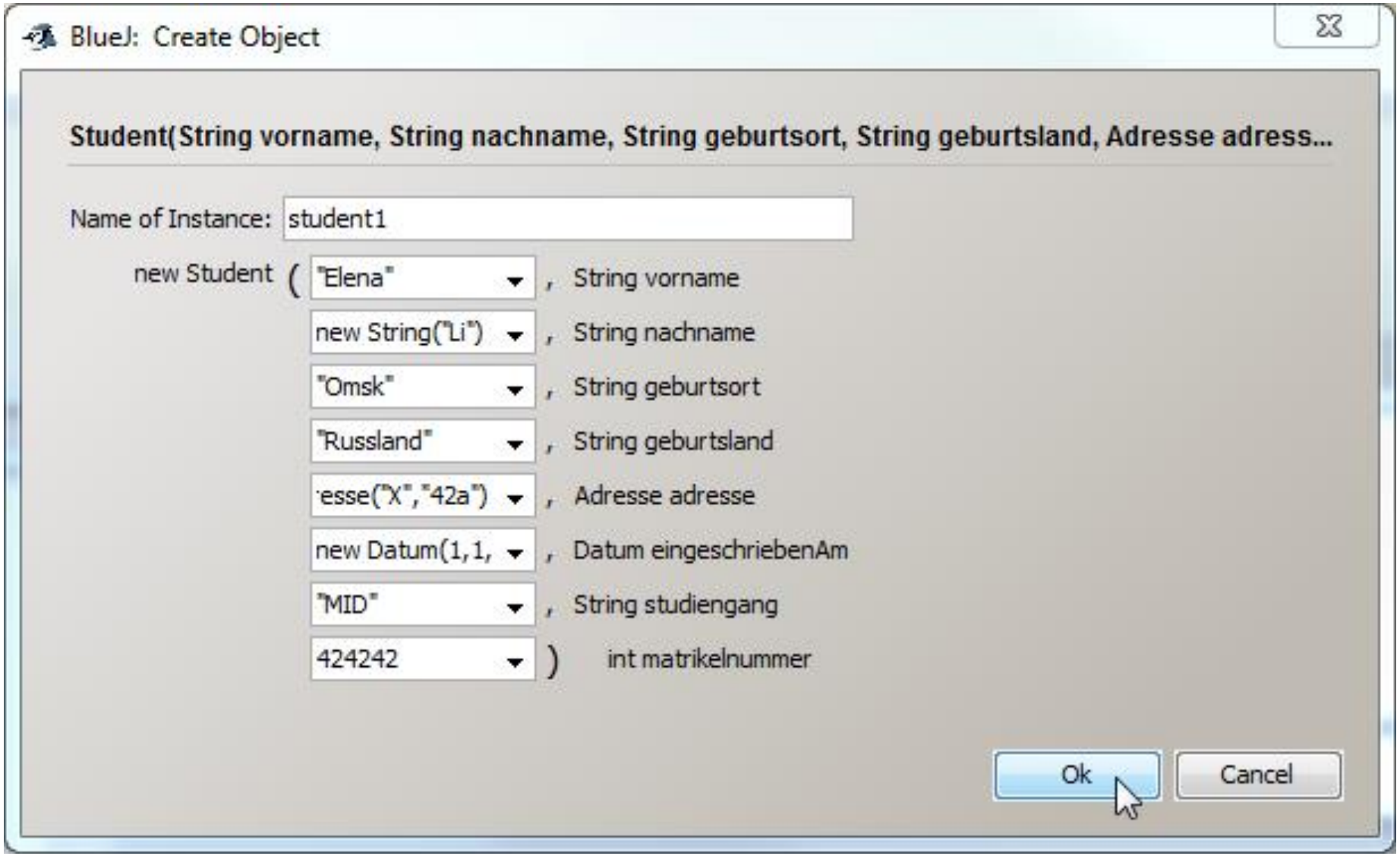
sinnvoll: alle Objektvariablen initialisieren



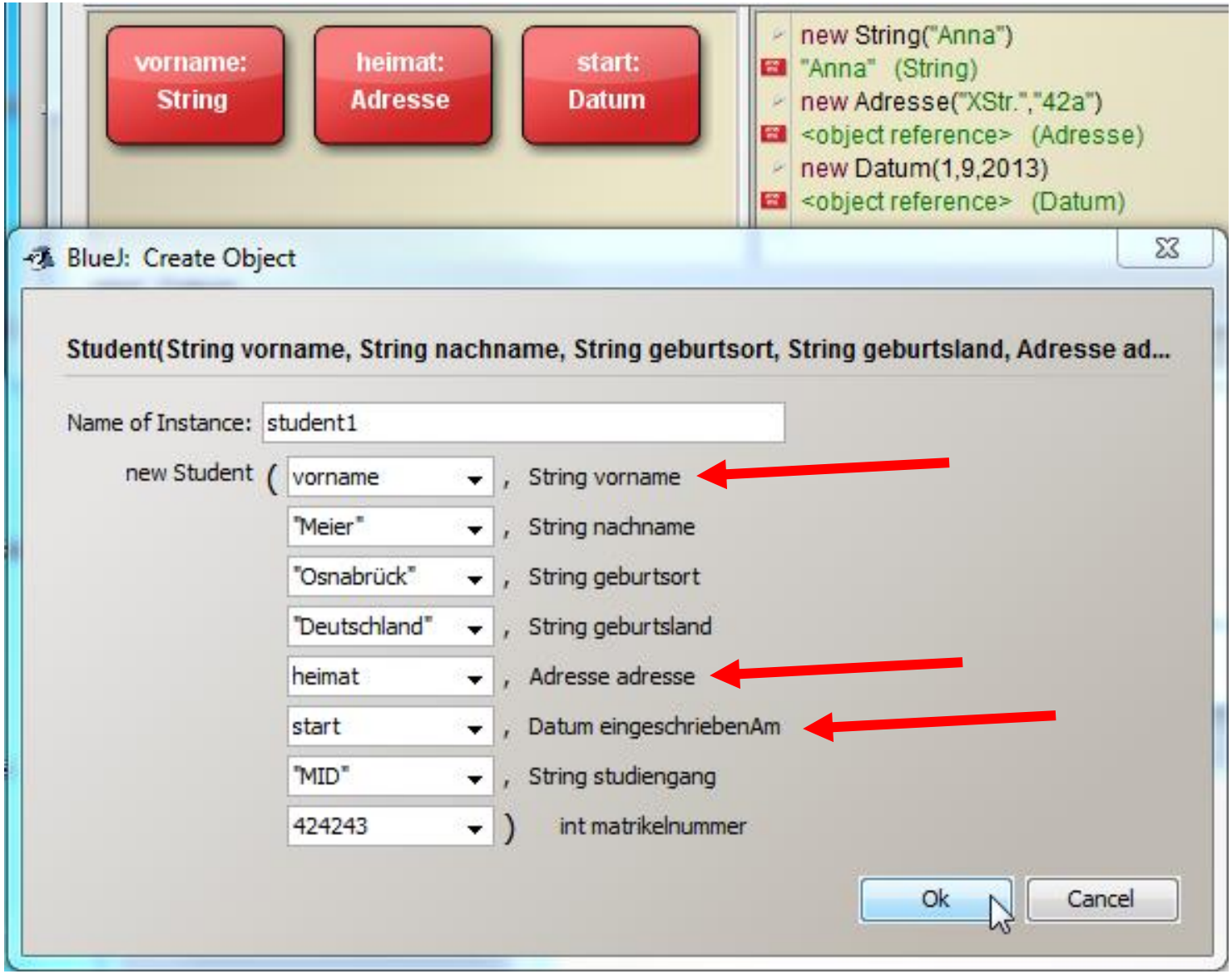
```
Praktikum(String modul, String semester
          , String veranstaltend, String mitarbeitend){
  this.modul = modul;
  this.semester = semester;
  this.veranstaltend = veranstaltend;
  this.mitarbeitend = mitarbeitend;
  this.termine = new ArrayList<String>();
  this.teilnehmende = new ArrayList<Studierend>();
}
```

- Welche Alternativen gibt es?

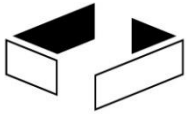
BlueJ: direkte Erzeugung von Objekten



BlueJ: Nutzung vorher erzeugter Objekte



bisherige Konstruktoren kritisch betrachtet



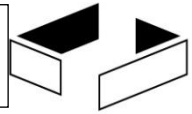
üblich ist der Aufruf mit Parameterliste

```
this.eingeschriebenAm = new Datum(tag, monat, jahr);
```

gibt Klassen mit Ausnahmen, bei denen das Objekt direkt hingeschrieben werden kann, dies sind

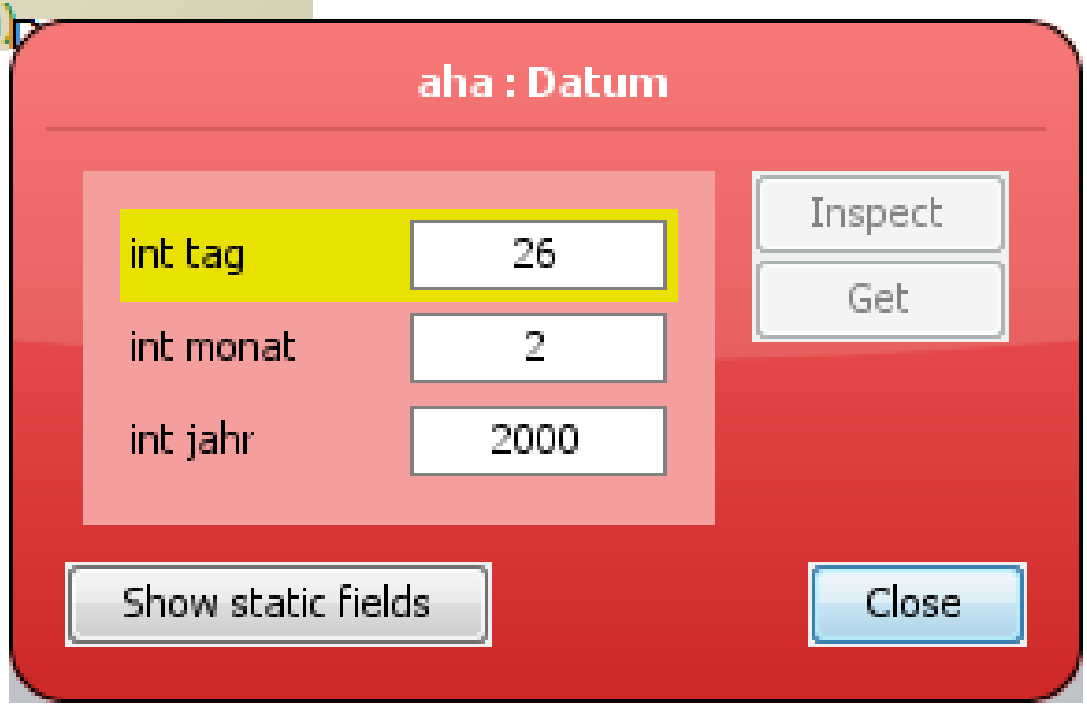
- `String stadt = "Osnabrueck";`
  geht aber auch: `String stadt = new String("Osnabrueck");`
- alle elementaren Datentypen haben keinen Konstruktor
  - `int jahr = 2023;`
  - `double wert = 42.41;`
  - `float wert2 = 42.42f;`
  - `boolean allesToll = true;`

Erstellung eines Objekts im Code Pad – Variante 3/3

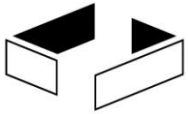


- Nutzung lokaler Variablen

```
int day = 26;
int month = 2;
int year = 2000;
Datum aha = new Datum(day,month,year);
aha
<object reference>   (Datum)
```



Variablen? was passiert denn da (1/5)



- Variablen referenzieren Werte im Speicher

```
int day = 6;
int month = 5;
int year = 1989;
```

(*Objekt)speicherdiagramm*

Variablen	Speicher	
day	int	6
month	int	5
year	int	1989

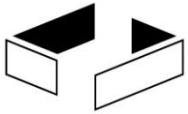
```
Datum aha = new Datum(day,month,year);
```

in der Klasse steht

```
Datum (int anInt1, int anInt2, int anInt3){
      this.tag = anInt1;
      this.monat = anInt2;
      this.jahr = anInt3;
}
```

Ansatz: anInt1 wird lokale Variable in der Ausführung des Konstruktors, erhält Kopie des Wertes von day übergeben

Variablen? was passiert denn da (2/5)



Variablen		Speicher
day	int	6
month	int	5
year	int	1989

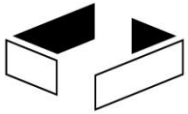
lokale Variablen, leben und sterben mit Konstruktor-ausführung

anInt1	int	6
anInt2	int	5
anInt3	int	1989

- weiter im Konstruktor

```
this.tag = anInt1;
this.monat = anInt2;
this.jahr = anInt3;
```

Variablen? was passiert denn da (3/5)



- Situation kurz vor Ende des Konstruktors

Variablen Speicher

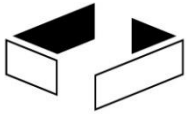
day	int	6
month	int	5
year	int	1989

anInt1	int	6
anInt2	int	5
anInt3	int	1989

Variablen, die zum gerade in Konstruktion befindlichen Objekt gehören

Datum	tag	int	6
	monat	int	5
	jahr	int	1989

Variablen? was passiert denn da (4/5)



- Situation nach der Ausführung von folgendem im Code Pad

```
int day = 6;
int month = 5;
int year = 1989;
Datum aha = new Datum(day,month,year);
```

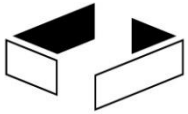
Variablen Speicher

day	int	6
month	int	5
year	int	1989

Datum	tag	int	6
	monat	int	5
	jahr	int	1989

aha	Datum

Variablen? was passiert denn da (5/5)



- neue nächste Anweisung:

```
day = 32;
```

Variablen Speicher

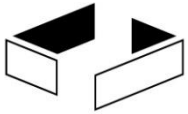
day	int	32
month	int	5
year	int	1989

Datum	tag	int	6
	monat	int	5
	jahr	int	1989

aha	Datum

- Hinweis: Thema wird später noch vertieft

Alternative Analyse



leicht geänderte Ausgangssituation:

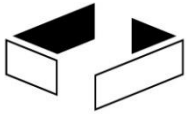
- Programm mit lokalen Variablen

```
int day = 29;
int month = 10;
int year = 1999;
Datum aha = new Datum(day, month, year);
```

- Konstruktor in der Klasse Datum hat jetzt hier die Form

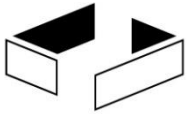
```
Datum (int tag, int monat, int jahr){
    this.tag = tag;
    this.monat = monat;
    this.jahr = jahr;
}
```

Alternative Analyse (2/2)



Variable / Anweisung	day	month	year	aha			tag	monat	jahr
				tag	monat	jahr			
`int day = 29`	29								
`int month = 10`	29	10							
`int year =1999`	29	10	1999						
`Datum aha = new Datum(day ,month,year)`	29	10	1999	0	0	0			
`Datum(tag, monat, jahr)`	29	10	1999	0	0	0	29	10	1999
`this.tag = tag`	29	10	1999	29	0	0	29	10	1999
`this.monat=monat`	29	10	1999	29	10	0	29	10	1999
`this.jahr = jahr`	29	10	1999	29	10	1999	29	10	1999
	29	10	1999	29	10	1999			

Erreichter Stand



- Klassen beschreiben Schablonen zur Objektbeschreibung
- Objektbeschreibung besteht aus Objektvariablen
- jede Objektvariable hat einen eindeutigen Typ
- Objekte werden durch Aufruf eines Konstruktors der Klasse erzeugt, Konstruktor kann Parameterliste zur Übergabe von Werten für die Objektvariablen (Belegung) enthalten
- Objektvariablen ohne Wert enthalten null-Referenz
- jede Klasse hat mindestens einen Konstruktor, wenn keiner geschrieben, dann der Default-Konstruktor
- es ist sehr sinnvoll, alle Konstruktoren explizit hinzuschreiben

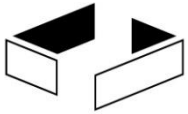
- wir können aber Werte von Objektvariablen nicht verändern

Beispiel

Video

Objektmethoden

Reden mit Objekten



- bei Objekten können Methoden aufgerufen werden
- aufgerufene Methoden können Werte der Objektvariablen verändern
- aufgerufene Methoden können Berechnungen ausführen
- aufgerufene Methoden können ein Ergebnisobjekt als Antwort des Aufrufs liefern

- Syntax

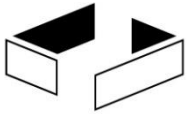
<TypDesErgebnisses> <Methodenname> **(**<Parameterliste>**){**

<Programm>

}

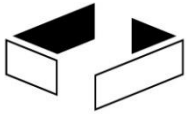
wenn kein Ergebnisobjekt, dann Ergebnistyp **`void`**

Einfache Methoden (1/2)



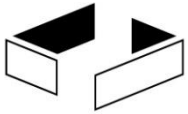
```
class Datum{
    int tag;
    int monat;
    int jahr;

    Datum (int tag, int monat, int jahr){
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
    }

    int getTag() {
        return this.tag;
    }

    void setTag(int tag) {
        this.tag = tag;
    }
```

Einfache Methoden (2/2)



```
    int getMonat() {
        return this.monat;
    }

    void setMonat(int monat) {
        this.monat = monat;
    }

    int getJahr() {
        return this.jahr;
    }

    void setJahr(int jahr) {
        this.jahr = jahr;
    }
}
```

Analyse der Methoden (1/2)

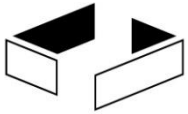


- Methode zum Lesen (Herausgeben) einer Variable var heißt typischerweise getVar (großen Buchstaben beachten)
- Rückgabeergebnis mit **`return`** <Ergebnis>;

```
int getTag() {
    return this.tag;
}
```

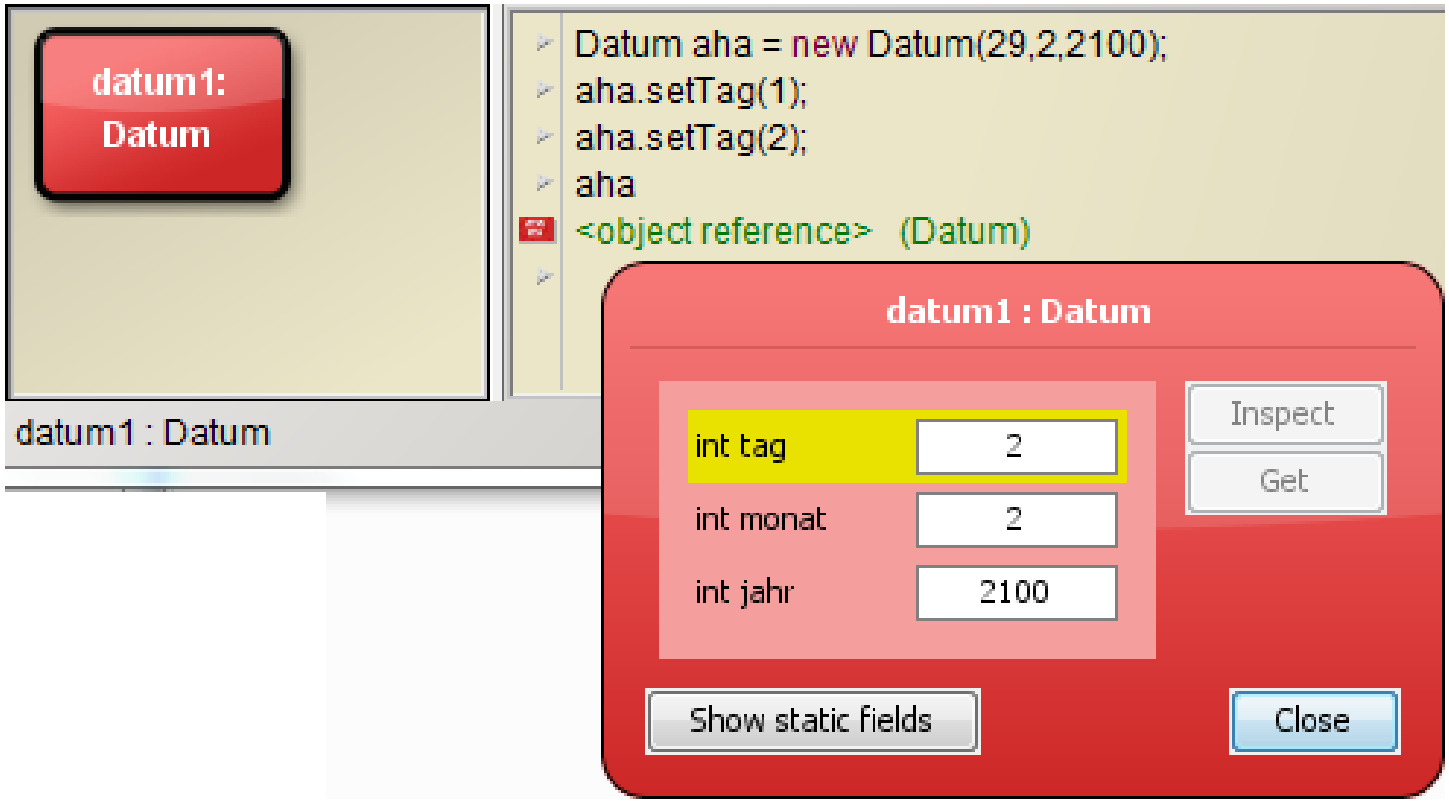
```
Datum aha = new Datum(29,2,2100);
aha.getTag()
29  (int)
aha.getMonat()
2  (int)
aha.getJahr()
2100  (int)
```

Analyse der Methoden (2/2)

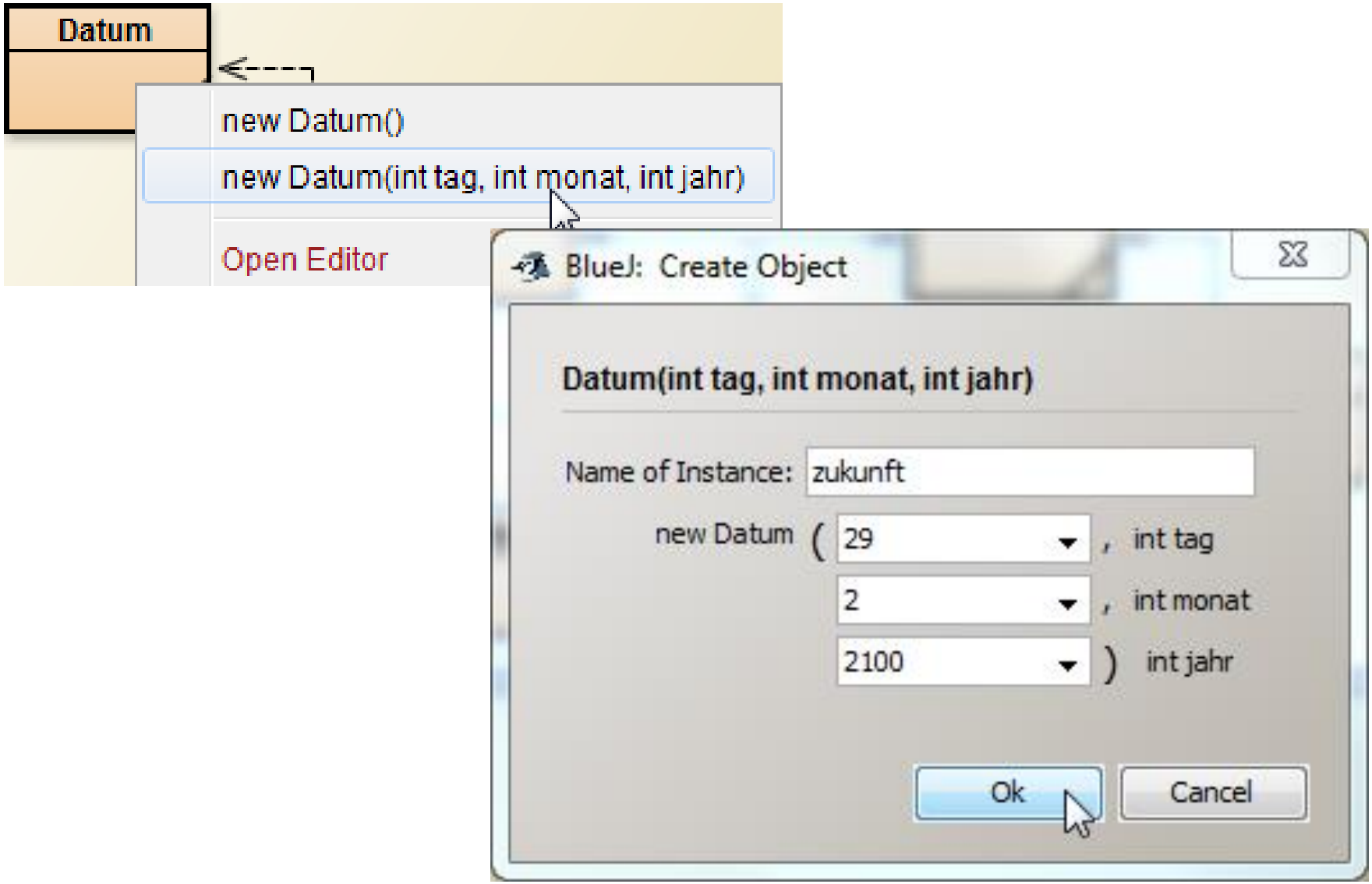


- Methode zum Schreiben (Verändern) einer Variable var heißt typischerweise setVar (großen Buchstaben beachten)

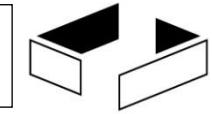
```
void setTag(int tag) {
    this.tag = tag;
}
```



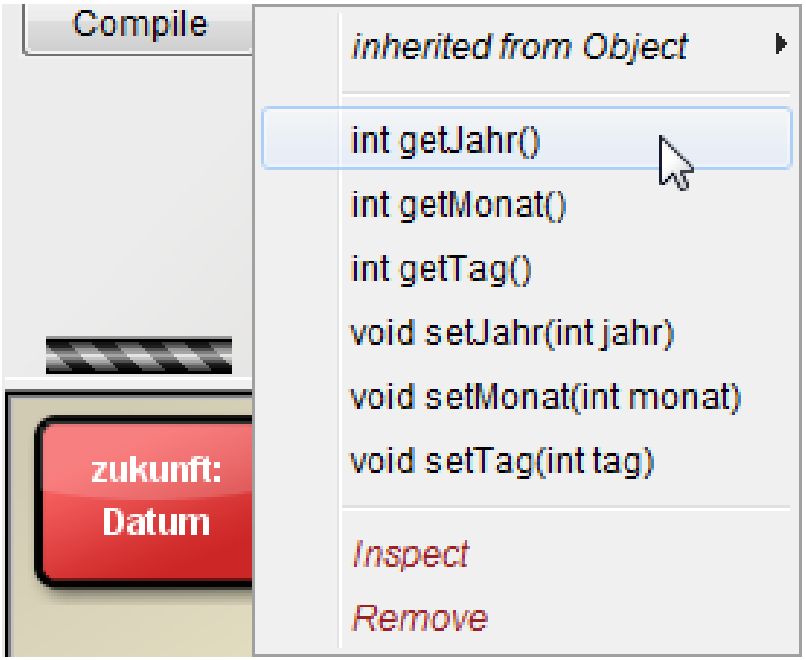
Direktes Ausführen von Methoden (1/3) - Erzeugen

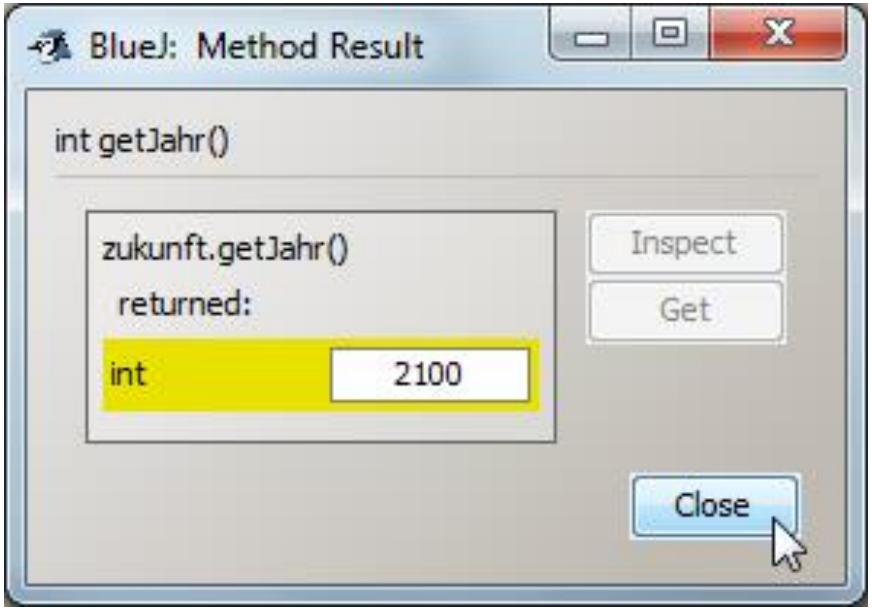


Direktes Ausführen von Methoden (2/3) - aufrufen

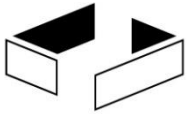


- Methode liefert Ergebnis, dass sofort angezeigt wird

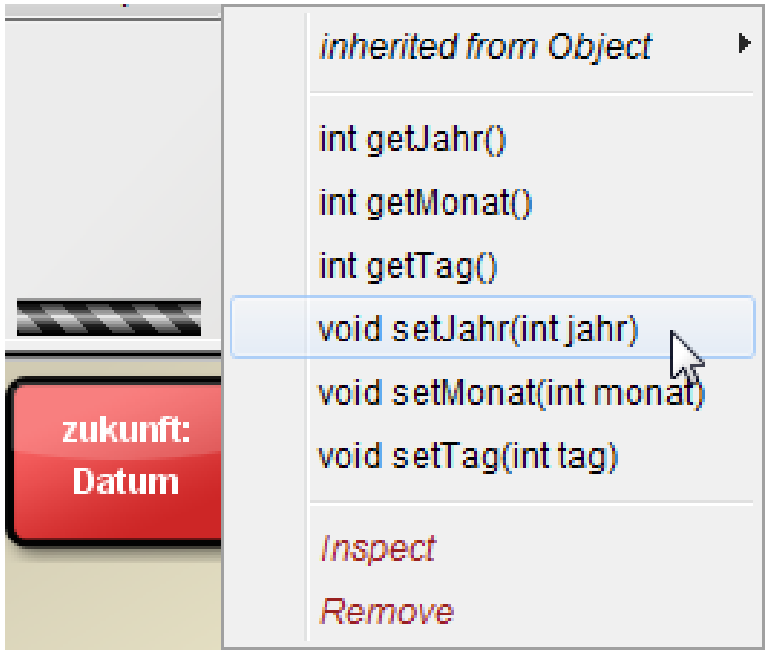


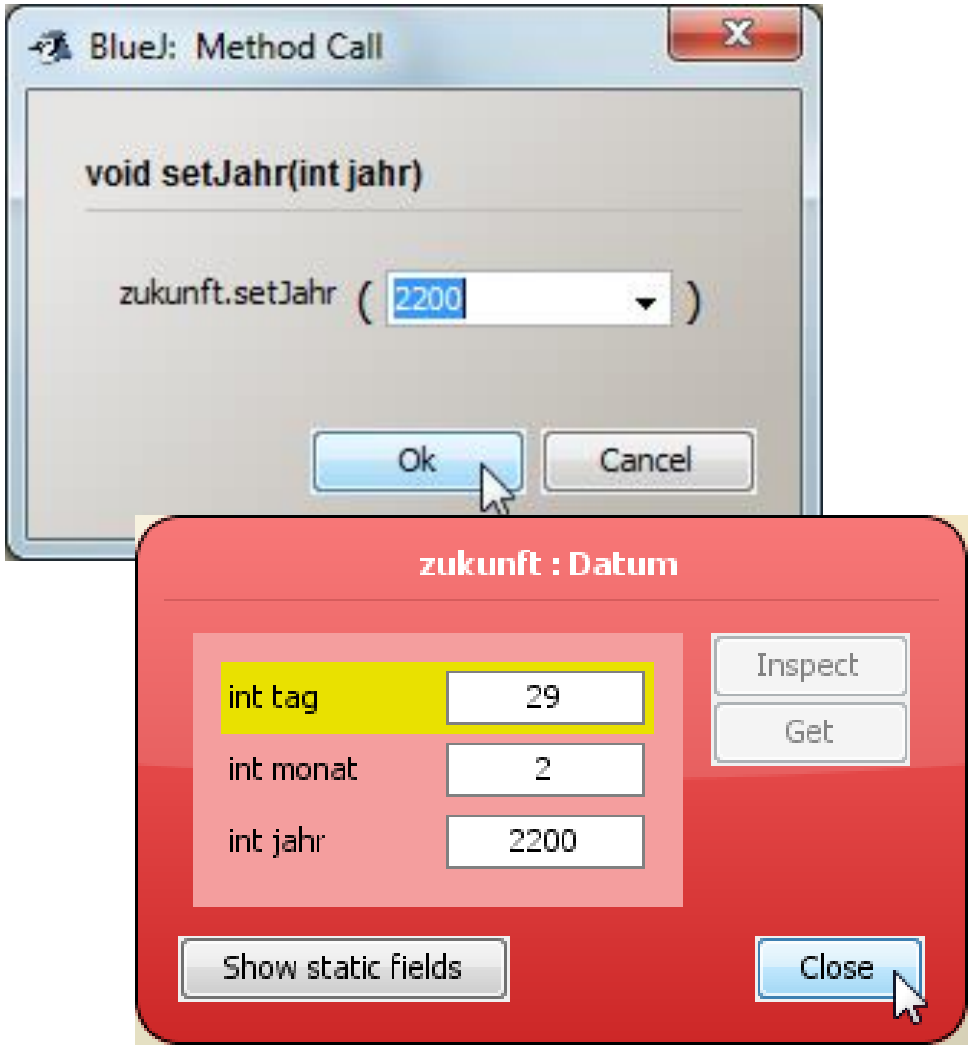


Direktes Ausführen von Methoden (3/3) - aufrufen

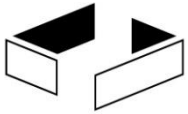


- Methode liefert kein Ergebnis, deshalb kein Effekt; Effekt über „Inspect“ sichtbar





Objekte nutzen Methoden anderer Objekte

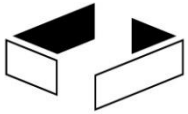


- Code Pad sehr sehr hilfreich beim Ausprobieren
- Bei Code-Änderungen oder Schließen von BlueJ sind alle Experimente gelöscht

```
Datum d = new Datum(29,2,2100);
d.setTag(1);
d.setTag(2);
```

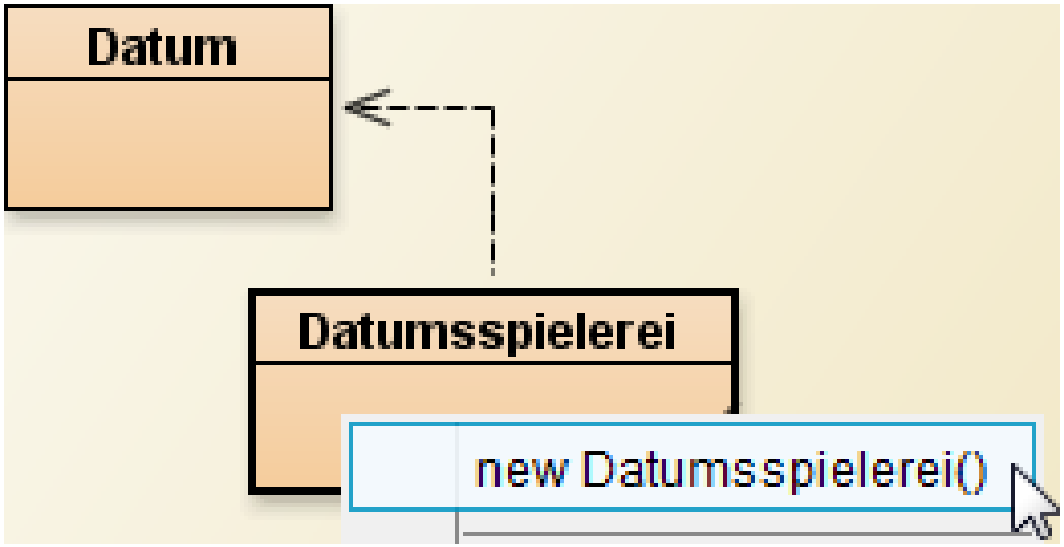
- Idee: Schreibe neue Klasse, die obiges Programmstück in einer Methode enthält, führe dann Methode in BlueJ aus

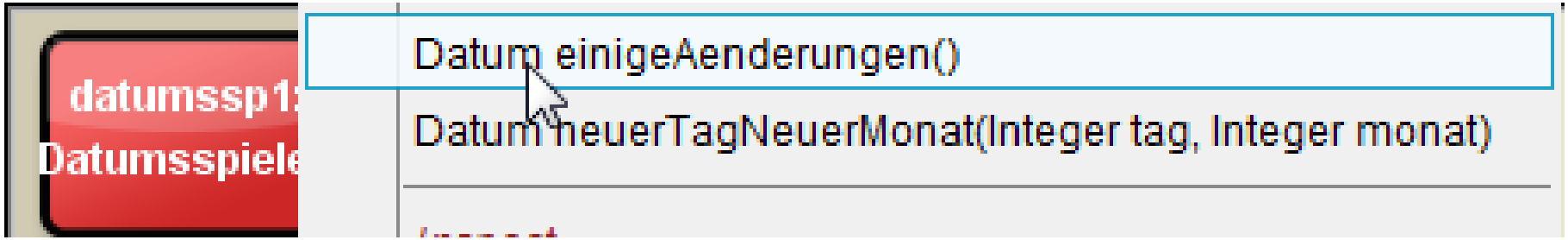
Experimentierklasse



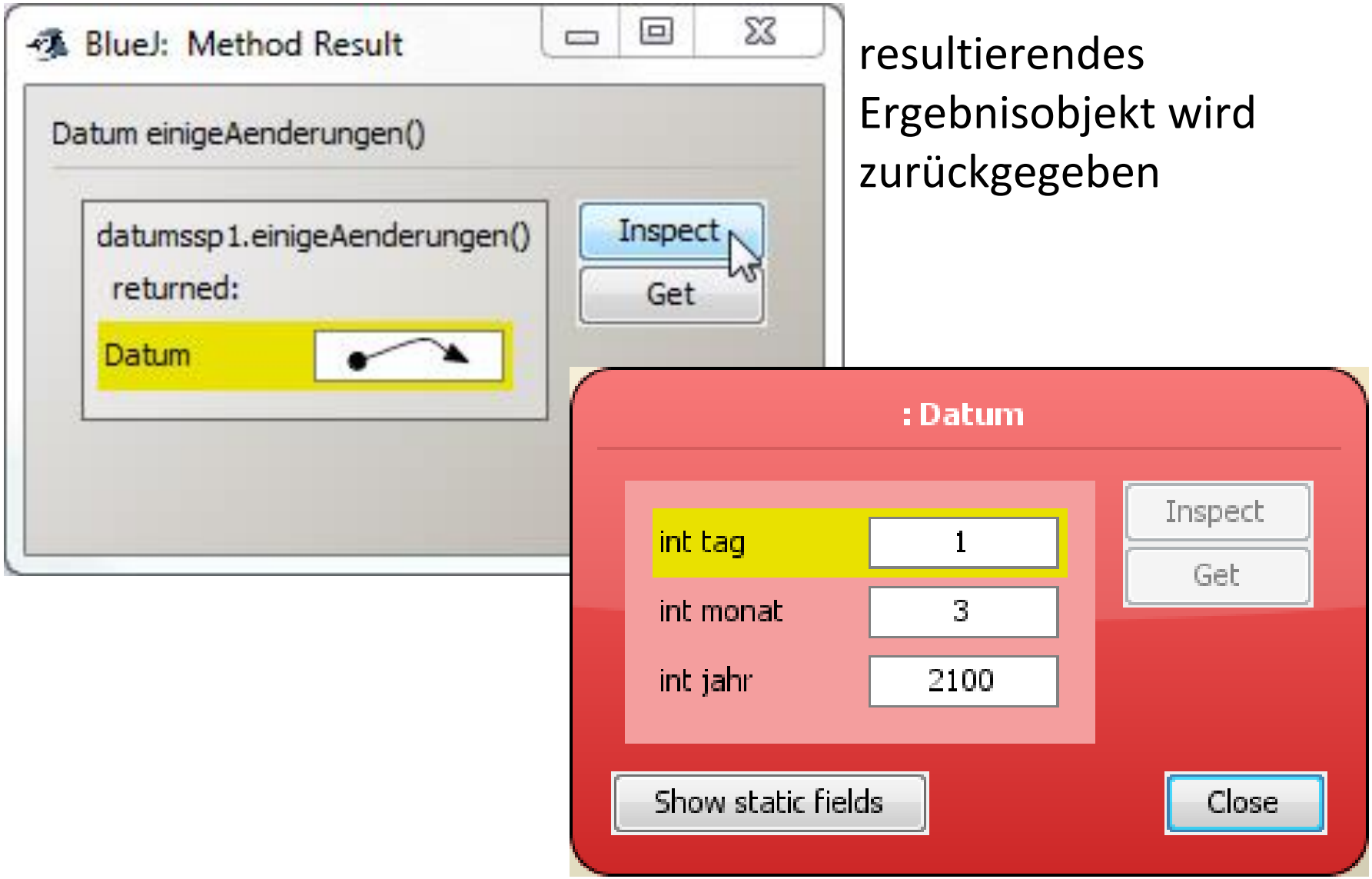
```
class Datumsspielerei{
        Datum datum;
        Datumsspielerei(){
                this.datum = new Datum(29,2,2100);
        }
        Datum einigeAenderungen(){
                this.datum.setTag(1);
                this.datum.setMonat(3);
                return this.datum;
        }
        Datum neuerTagNeuerMonat(int tag, int monat){
                this.datum.setTag(tag);
                this.datum.setMonat(monat);
                return this.datum;
        }
}
```

Experiment in BlueJ (1/2)



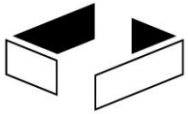


Experiment in BlueJ (2/2)



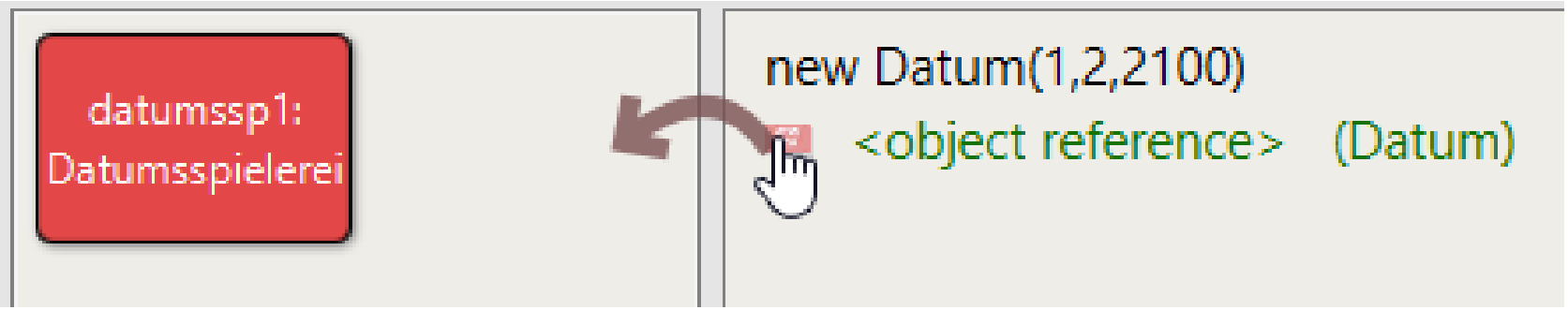
resultierendes Ergebnisobjekt wird zurückgegeben

Spielen mit Objekten in BlueJ (1/2)

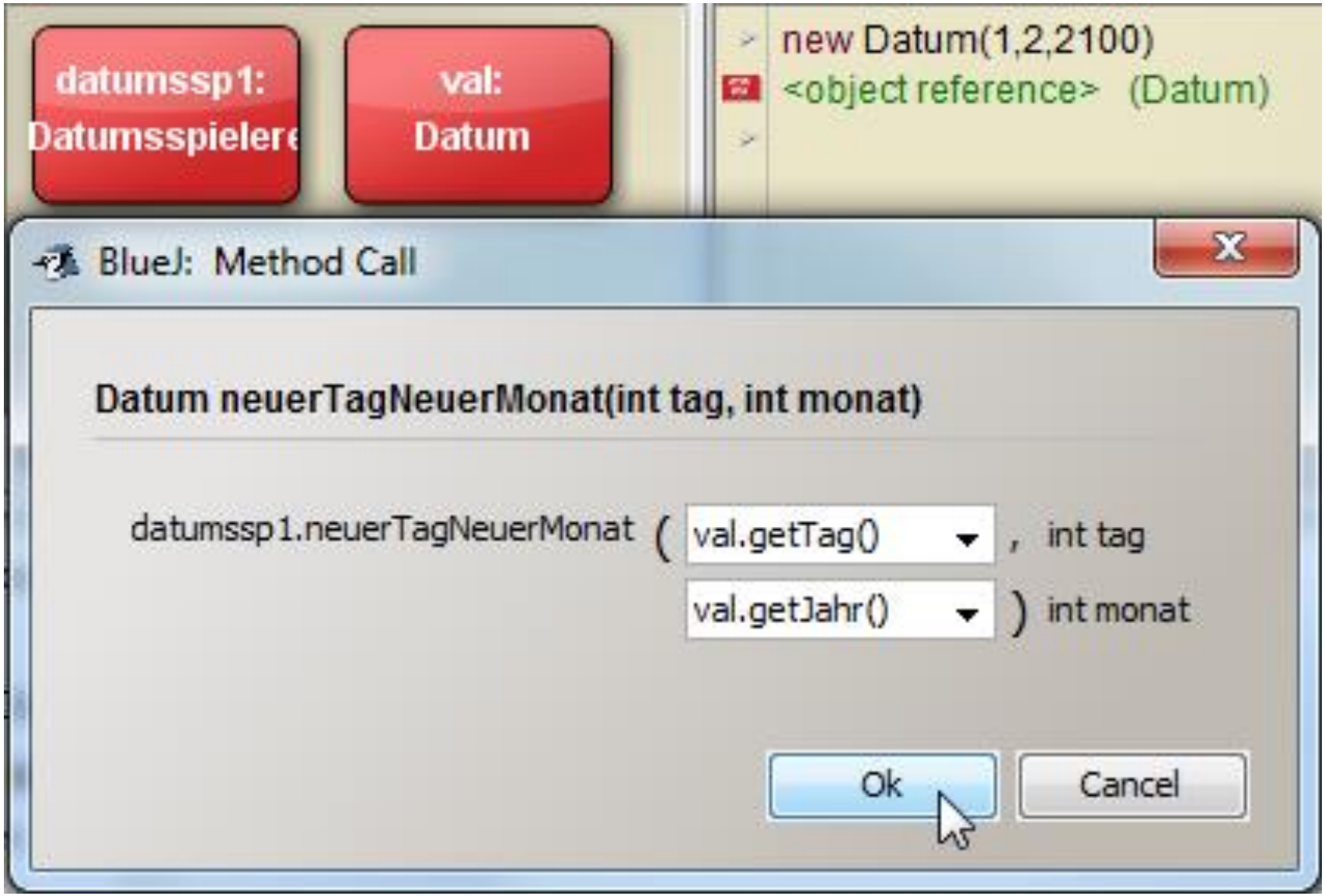


- Generell kann man bei direkten Aufrufen von Konstruktoren und Methoden direkt auf Objekte zugreifen, die in der Objektleiste liegen oder Objekte direkt erstellen

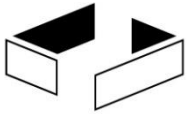
Objekterzeugung



Spielen mit Objekten in BlueJ (2/2)

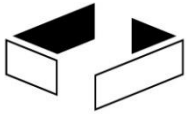


Methoden mit lokalen Variablen



- innerhalb von Methoden können an beliebigen Stellen lokale Variablen deklariert und innerhalb der Methode genutzt werden
- Syntax und Möglichkeit zur Setzung des Startwerts wie bei Objektvariablen
- genauer: lokale Variablen können in Blöcken (Programmfragmente in geschweiften Klammern) deklariert werden; nach dem Verlassen des Blocks ist Variable selbst verloren (ihr Objekt kann aber z. B. als Rückgabewert weiter existieren)

lokale Variablen in Blöcken



```
{
  // Deklaration von x
}
```

x unbekannt, nicht nutzbar

x nutzbar

x „gestorben“, Wert nicht mehr als x nutzbyar

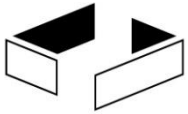
```
{
}
```

x unbekannt, nicht nutzbar

```
{
  // Deklaration von x
}
```

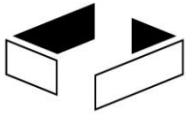
neues x, hat nichts mit obigen x zu tun, kann anderen Typ haben

Beispiel: Nutzung einer lokalen Variable



```
class Datumsspielerei{

        Datumsspielerei(){
        }

        Datum heiraten(int wert){
                Datum ergebnis = new Datum();
                ergebnis.setTag(wert);
                ergebnis.setMonat(wert);
                ergebnis.setJahr(wert);
                return ergebnis;
        }
}
```

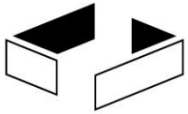
Namensregeln und Sichtbarkeitsbereiche



- In demselben Block darf es nicht zwei Variablen mit gleichem Namen geben (Syntaxfehler)
- Beispiel: Kann nicht eine Objektvariable monat vom Typ int und eine Variable monat vom Typ String geben
- Lokale Variablen können die gleichen Namen wie Objektvariablen haben, verdecken diese aber
- Objektvariablen immer durch **`this.`** zugreifbar

- Grundregel der guten Programmierung: Schaffen Sie niemals Situationen, in denen Sie über verdeckte Variablen nachdenken müssen

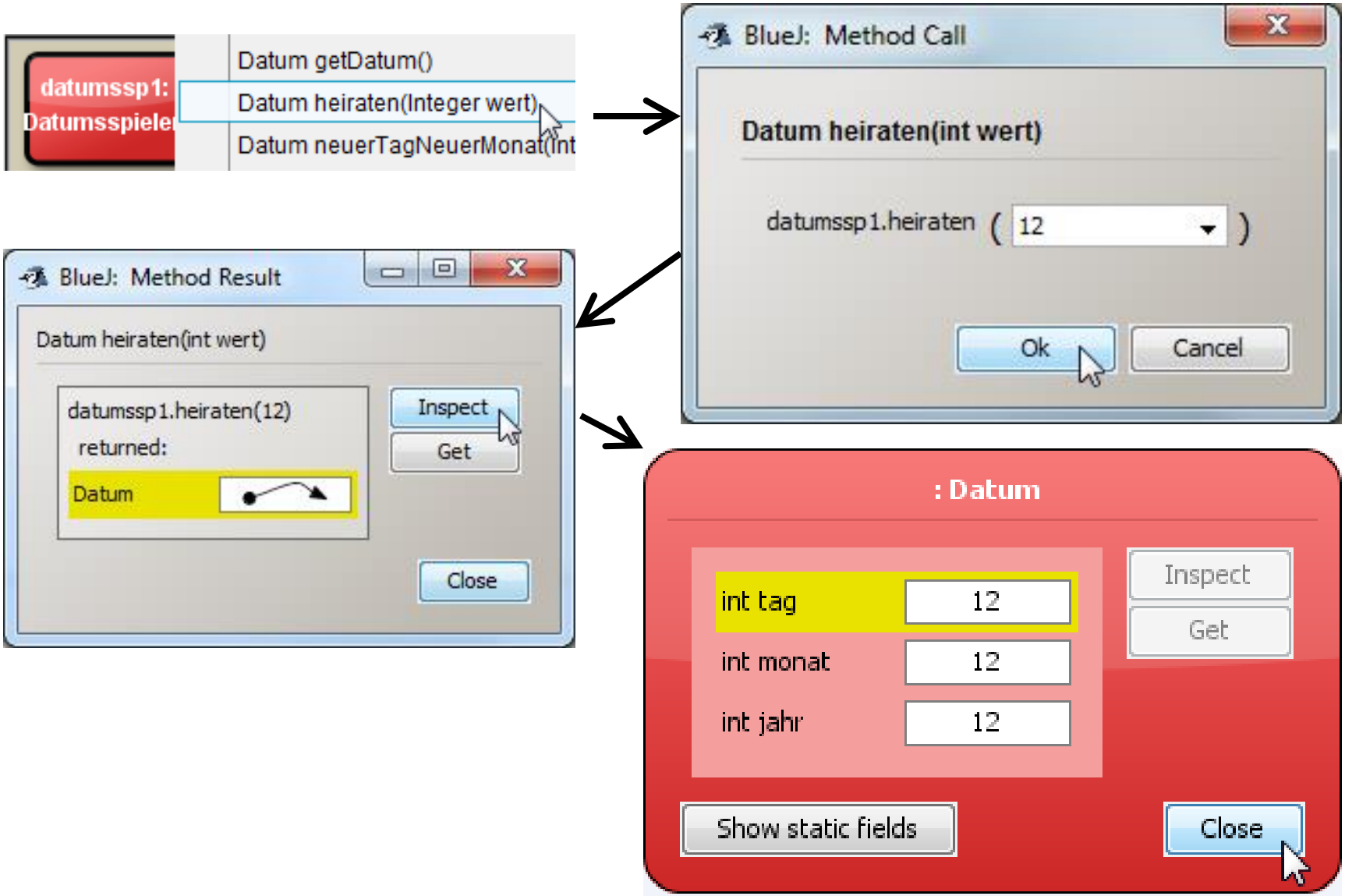
Verdecken einer Objektvariablen (1/3)



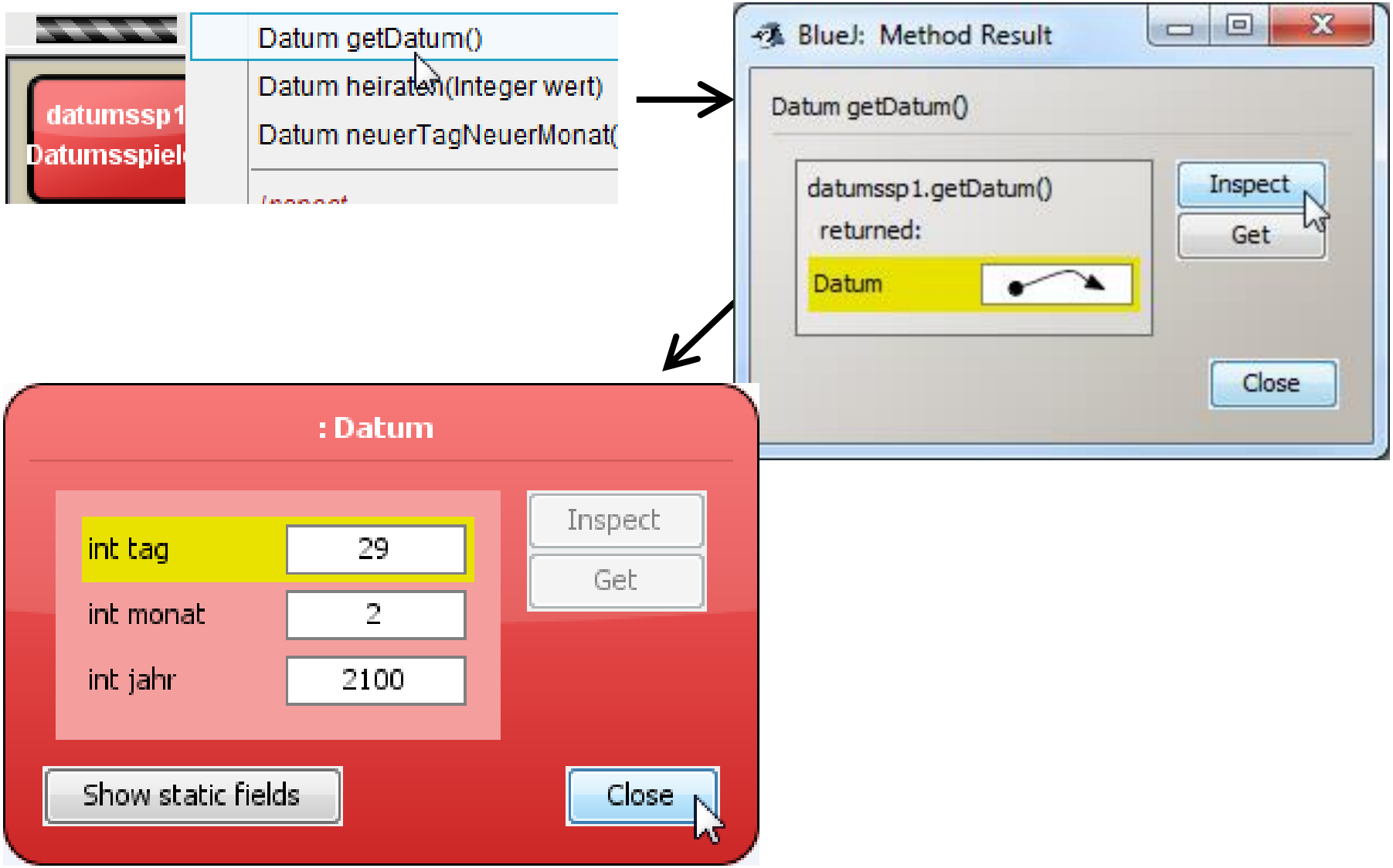
```
class Datumsspielerei{
        Datum datum;
        Datumsspielerei(){
                this.datum = new Datum(29,2,2100);
        }
        Datum getDatum(){
                return this.datum;
        }
        Datum heiraten(int wert){
                Datum datum = new Datum();
                datum.setTag(wert);
                datum.setMonat(wert);
                datum.setJahr(wert);
                return datum;
        }
}
```

hat nichts mit obigen datum zu tun (schlechter Stil)

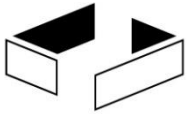
Verdecken einer Objektvariablen (2/3)



Verdecken einer Objektvariablen (3/3)

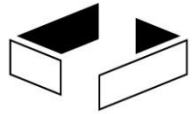


Methodennutzung

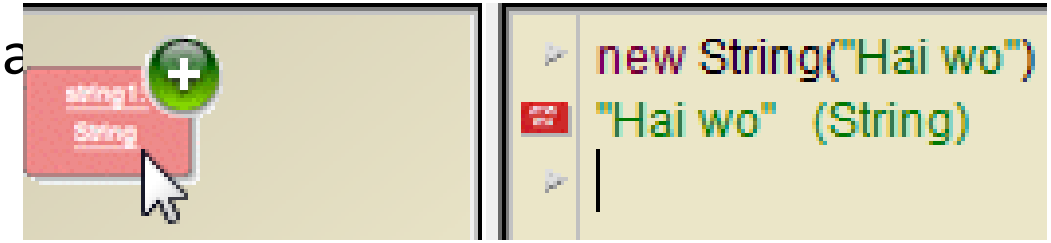


- Neben dem Wissen über existierende Klassen ist das Wissen über die angebotenen Methoden Grundlage der Programmiererfahrung
- In dieser Vorlesung sehen die objektorientierten Programmiergrundlagen im Vordergrund, deshalb Wissen über konkrete Java-Klassen im Hintergrund
- Wichtig: Man muss wissen, wo Informationen über Klassen/Methoden stehen (z. B. Java-Dokumentation) und wie man mit ihnen experimentieren kann (z. B. in BlueJ)

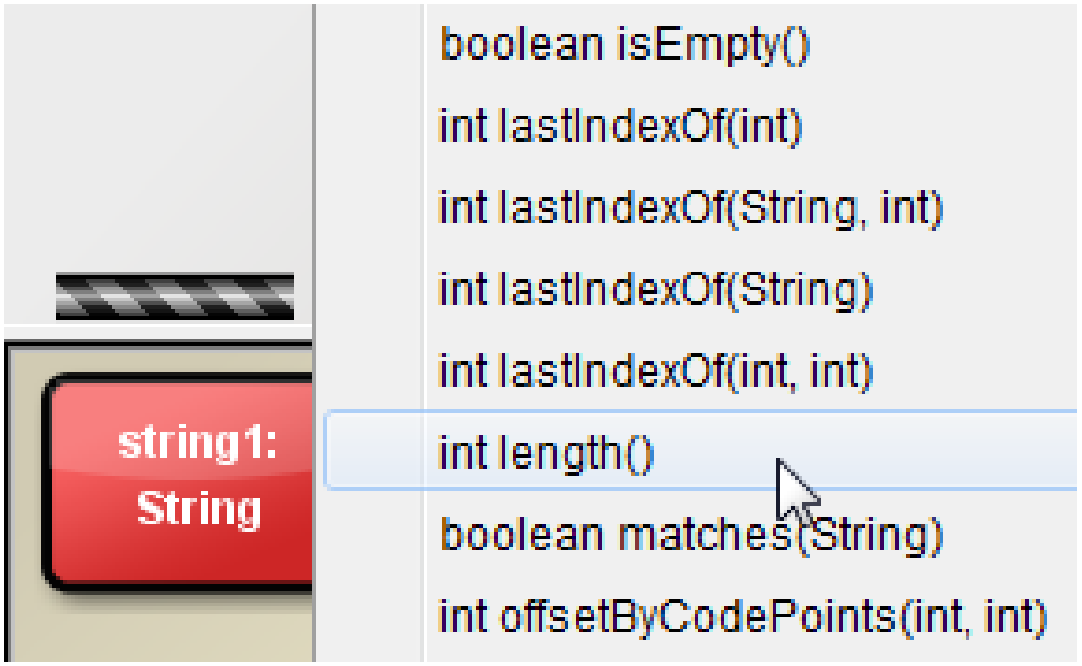
Möglichkeiten mit String-Objekten



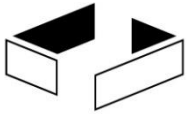
- Erstellung mit Code Pa
  und Objekt in
  Objektleiste ziehen



- Rechtsklick auf dem
  Objekt zeigt nutzbare
  Methoden (zur Zeit nur
  letzte Methode
  halbwegs verständlich)

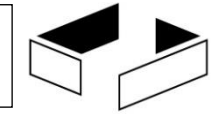


Weiteres Analysebeispiel (1/4) - Klasse

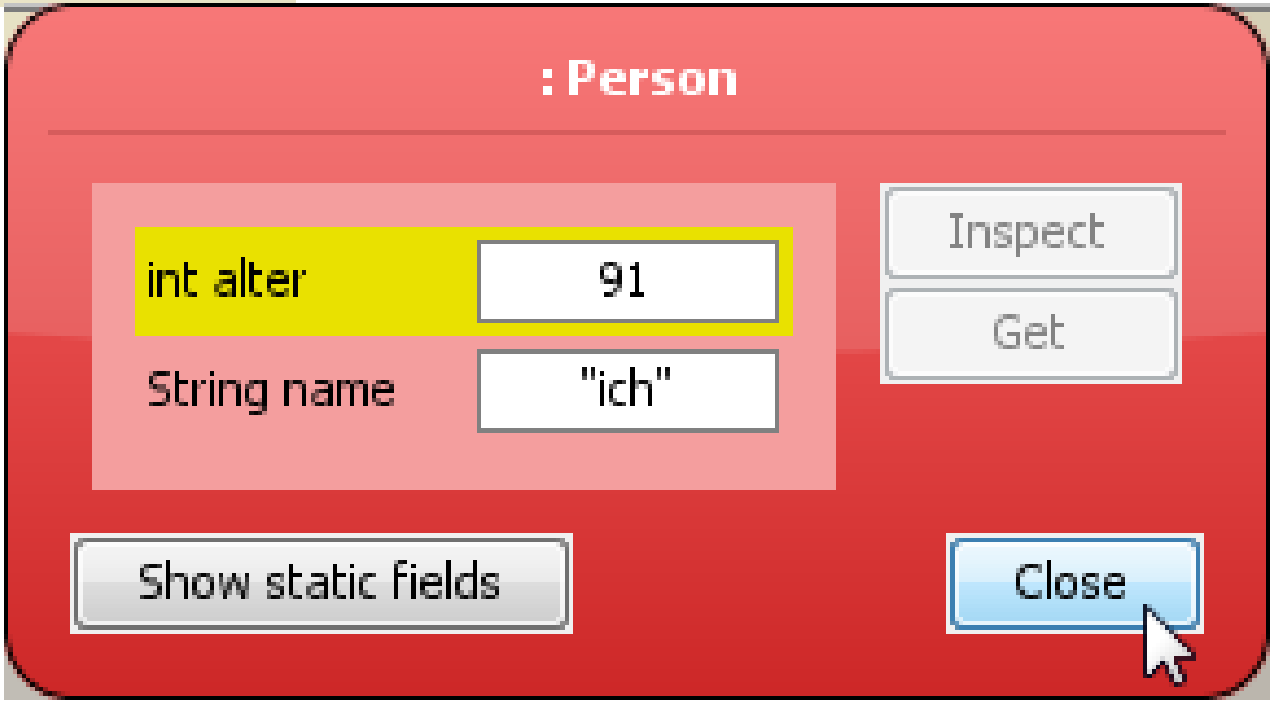


```
class Person{
    int alter = 42;
    String name = "ich";

    int naechstes(){
        int erg = this.alter + 1;
        return erg;
    }

    int merkwuerdig(int wert){
        int erg = this.name.length();
        erg = erg + wert;
        return erg;
    }

    void komisch(int wert, int w2){
        int tmp = wert + w2;
        this.alter = tmp;
    }
}
```

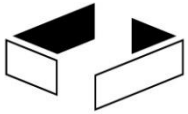
Weiteres Analysebeispiel (2/4) - Beispielprogramm



```
Person p = new Person();
int v1 = p.naechstes();
int v2 = p.merkwuerdig(45);
p.komisch(v1,v2);
p
<object reference>   (Person)
```



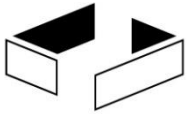
Weiteres Analysebeispiel (3/4) – Analyse



lokale Variablen / Parameter

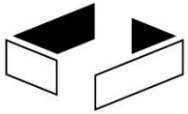
Variable / Anweisung	p		v1	erg	v2	wert	erg	wert	w2	tmp
	alter	name								
`Person p = new Person()`	42	„ich“								
`int v1 = p.naechstes()`	42	„ich“								
`naechstes()`	42	„ich“								
`int erg = this.alter + 1`	42	„ich“		43						
`return erg`	42	„ich“	43	43						
`int v2 = p.merkwuerdig(45)`	42	„ich“	43							

Weiteres Analysebeispiel (4/4) – Analyse



Variable / Anweisung	p		v1	erg	v2	wert	erg	wert	w2	tmp
	alter	name								
`merkwuerdig(int wert)`	42	„ich“	43			45				
`int erg = this.name.length()`	42	„ich“	43			45	3			
`erg = erg + wert`	42	„ich“	43			45	48			
`return erg`	42	„ich“	43		48	45	48			
`p.komisch(v1,v2)`	42	„ich“	43		48					
`komisch(int wert, int w2)`	42	„ich“	43		48			43	48	
`int tmp = wert + w2`	42	„ich“	43		48			43	48	91
`this.alter = tmp`	91	„ich“	43		48			43	48	91
	91	„ich“	43		48					

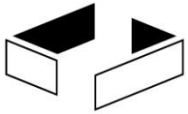
Vereinfachte Klasse Studierend (1/3)



Video

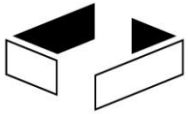
```
class Studierend{
    String vorname ="Eva";
    String nachname ="Mustermann";
    int geburtsjahr = 1990;
    String studiengang = "IMI";
    int matrikelnummer = 232323;

    Studierend(String vorname, String nachname,
                int geburtsjahr, String studiengang,
                int matrikelnummer) {
        this.vorname = vorname;
        this.nachname = nachname;
        this.geburtsjahr = geburtsjahr;
        this.studiengang = studiengang;
        this.matrikelnummer = matrikelnummer;
    }

    Studierend(){
    }
```

Vereinfachte Klasse Studierend (2/3)



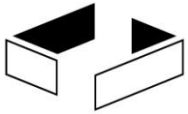
```
String getVorname() {
      return this.vorname;
}

void setVorname(String vorname) {
      this.vorname = vorname;
}

String getNachname() {
      return nachname;
}

void setNachname(String nachname) {
      this.nachname = nachname;
}

int getGeburtsjahr() {
      return this.geburtsjahr;
}
```

Vereinfachte Klasse Studierend (3/3)



```
        void setGeburtsjahr(int geburtsjahr) {
              this.geburtsjahr = geburtsjahr;
        }

        String getStudiengang() {
              return this.studiengang;
        }

        void setStudiengang(String studiengang) {
              this.studiengang = studiengang;
        }

        int getMatrikelnummer() {
              return this.matrikelnummer;
        }

        void setMatrikelnummer(int matrikelnummer) {
              this.matrikelnummer = matrikelnummer;
        }
}
```

Ein- und Ausgabe mit Klasse EinUndAusgabe



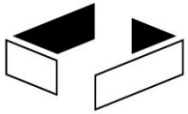
- Ein- und Ausgabe generell etwas trickreich bzw. schmuddelig, deshalb hier objektorientiert saubere Klasse EinUndAusgabe

Methode	Rückgabetyp	Aufgabe	Default
`leseString()`	String	liest Text von Konsole	
`leseInteger()`	int	liest ganze Zahl	-1
`leseDouble()`	double	liest Fließkommazahl	-1.0
`leseBoolean()`	boolean	liest Wahrheitswert	false

- Eingabe endet immer mit dem Drücken von "Return"
- Default-Wert bei falscher Eingabe

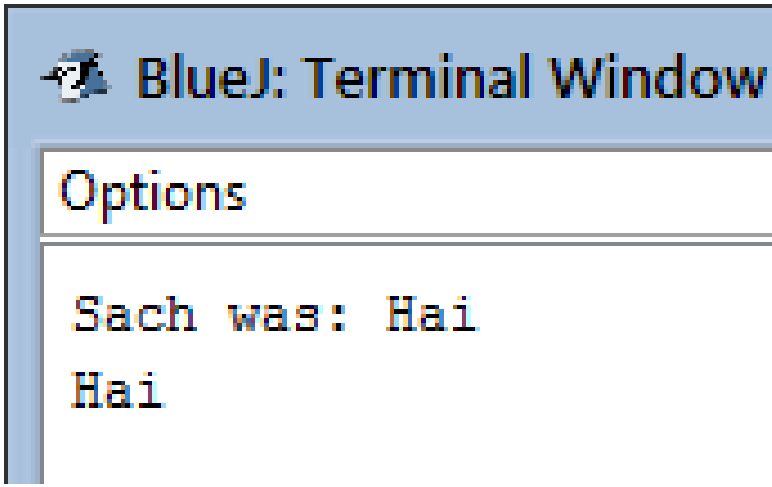
Methode	Parametertyp	Aufgabe
`ausgeben(.)`	String	gibt Text auf Konsole aus

Besonderheit EinUndAusgabe: nur ein Objekt (1/3)

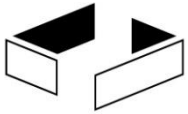


Video

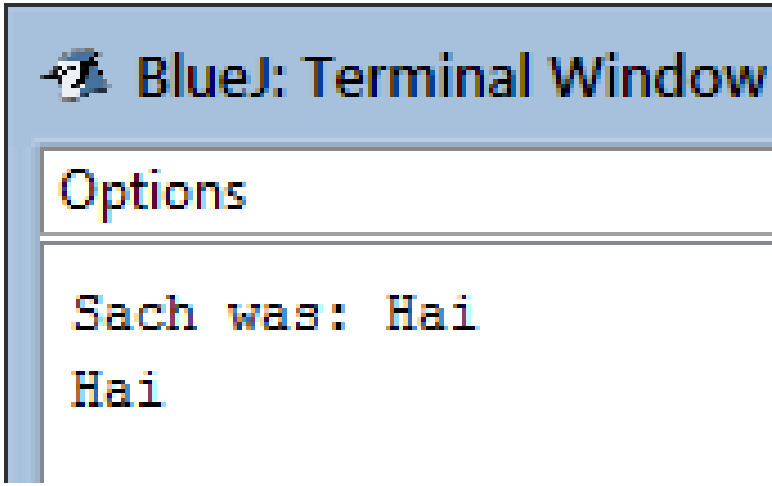
```
class EinUndAusgabeAnalyse{
    EinUndAusgabe io = new EinUndAusgabe();

    String eingeben(String befehl){
        this.io.ausgeben(befehl);
        String ein = this.io.leseString();
        return ein;
    }

    void zeigen(String text){
        this.io.ausgeben(text);
    }

    void beispiel(){
        String s = this.eingeben("Sach was: ");
        this.zeigen(s);
    }
}
```



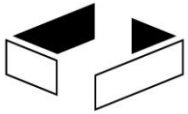
Besonderheit EinUndAusgabe: nur ein Objekt (2/3)



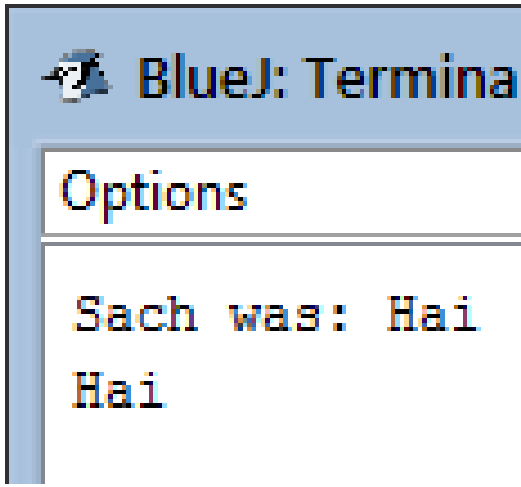
```
class EinUndAusgabeAnalyse{

    String eingeben(String befehl, EinUndAusgabe io){
        io.ausgeben(befehl);
        String ein = io.leseString();
        return ein;
    }

    void zeigen(String text
                , EinUndAusgabe ea){
        ea.ausgeben(text);
    }

    void beispiel(){
        EinUndAusgabe io = new EinUndAusgabe();
        String s = this.eingeben("Sach was: ", io);
        this.zeigen(s, io);
    }
}
```



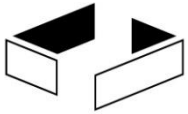
Besonderheit EinUndAusgabe: nur ein Objekt (3/3)



```
class EinUndAusgabeAnalyse{

    String eingeben(String befehl){
        EinUndAusgabe io = new EinUndAusgabe();
        io.ausgeben(befehl);
        String ein = io.leseString();
        return ein;
    }

    void zeigen(String text){
       EinUndAusgabe io = new EinUndAusgabe();
       io.ausgeben(text);
    }

    void beispiel(){
        String s = this.eingeben("Sach was: ");
        this.zeigen(s);
    }
}
```



Beispielnutzung (1/5)



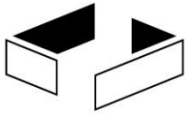
Video

- in Studierend

```
void ausgeben(){
    EinUndAusgabe io = new EinUndAusgabe();
    io.ausgeben("Vorname: ");
    io.ausgeben(this.vorname);
    io.ausgeben("\nNachname: ");
    io.ausgeben(this.nachname);
    io.ausgeben("\nStudiengang: ");
    io.ausgeben(this.studiengang);
    io.ausgeben("\n");
}
```

- \n steht für einen Zeilenumbruch
- Hinweis: später schönere Java-spezifische Lösung

Beispielnutzung (2/5)



- in Klasse StudierendSpielerei

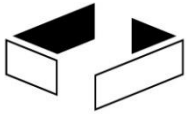
```
void beispielausgabe(){
  Studierend s1 = new Studierend("Mr","X",1990,"IMI",234243);
  Studierend s2 = new Studierend("Ms","Y",1980,"ITI",234244);
  s1.ausgeben();
  s2.ausgeben();
  s1 = s2;
  s2.setVorname("King");
  s1.ausgeben();
}
```

BlueJ: Terminal Window

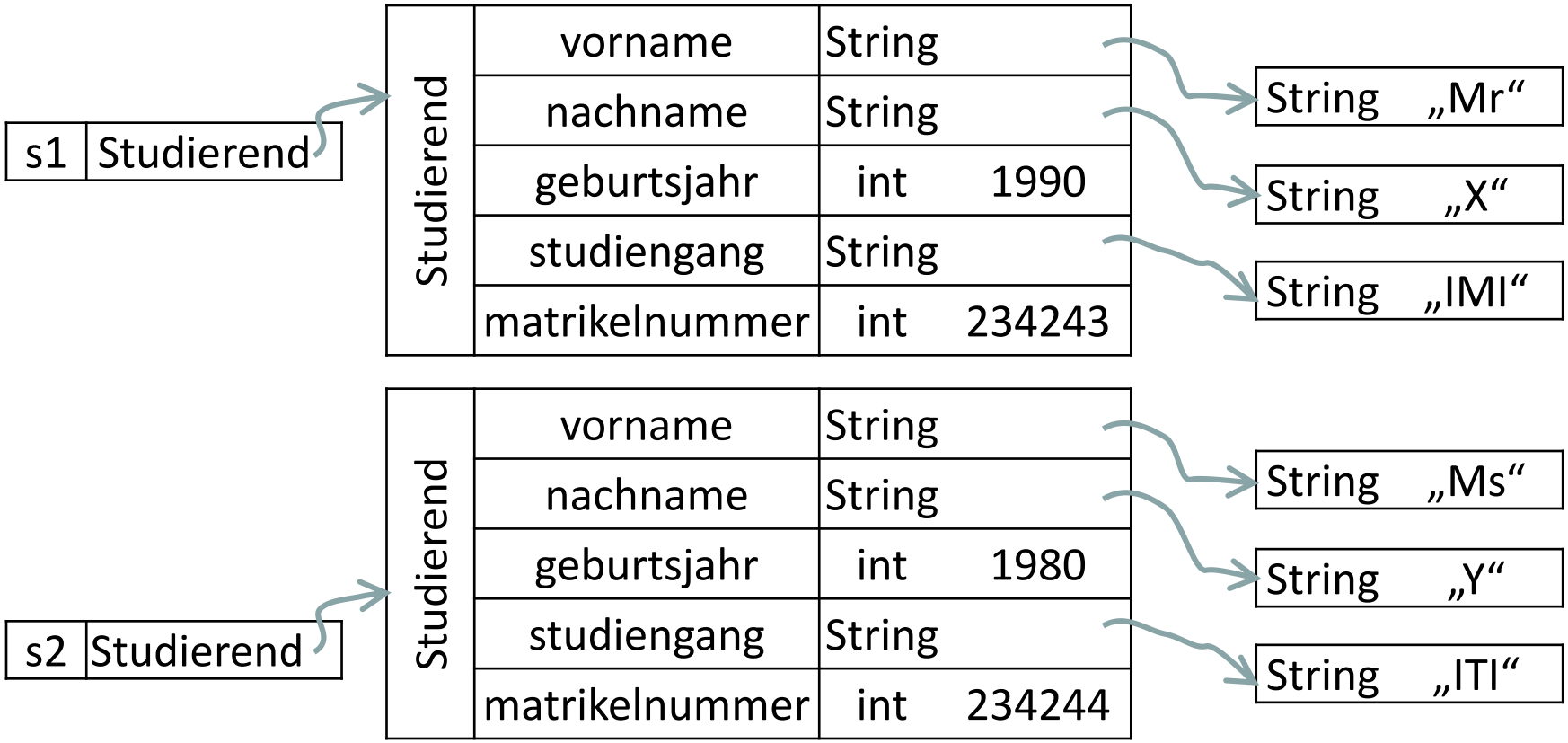
Options

```
Vorname: Mr
Nachname: X
Studiengang: IMI
Vorname: Ms
Nachname: Y
Studiengang: ITI
```

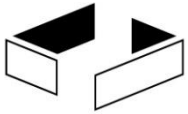
Beispielnutzung (3/5)



```
void beispielausgabe(){
  Studierend s1 = new Studierend("Mr","X",1990,"IMI",234243);
  Studierend s2 = new Studierend("Ms","Y",1980,"ITI",234244);
```

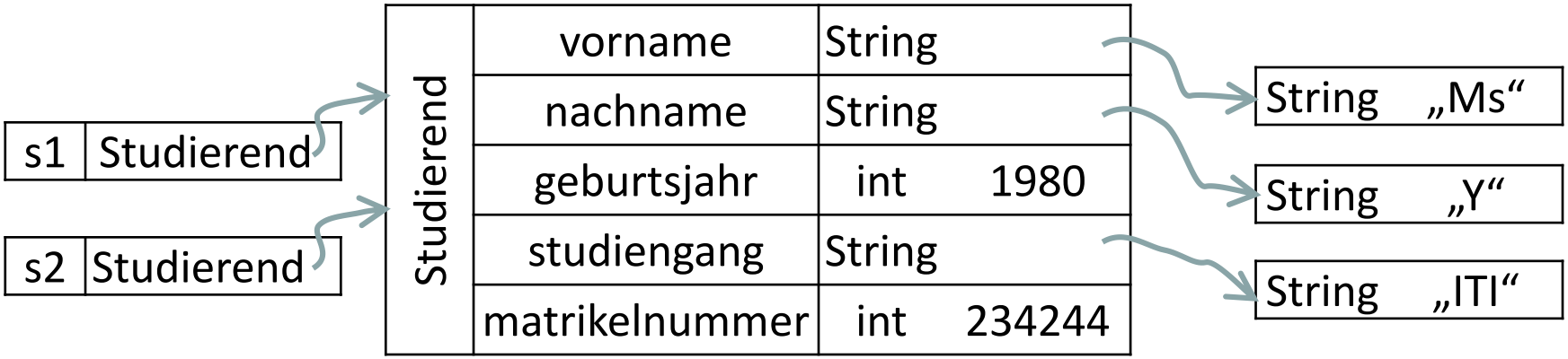


Beispielnutzung (4/5)

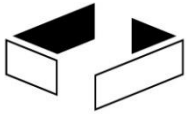


```
void beispielausgabe(){
  Studierend s1 = new Studierend("Mr","X",1990,"IMI",234243);
  Studierend s2 = new Studierend("Ms","Y",1980,"ITI",234244);
  s1.ausgeben();
  s2.ausgeben();
  s1 = s2;
```

Zuweisung von Objekt an Objekt; es sind Referenzen keine Wertekopien wie bei elementaren Typen

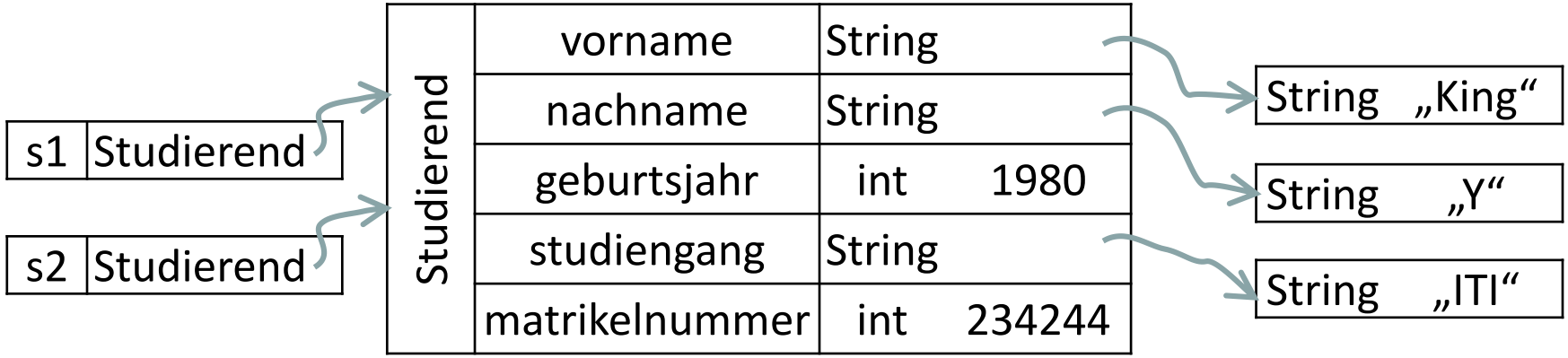


Beispielnutzung (5/5)

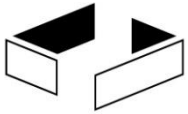


```
void beispielausgabe(){
  Studierend s1 = new Studierend("Mr","X",1990,"IMI",234243);
  Studierend s2 = new Studierend("Ms","Y",1980,"ITI",234244);
  s1.ausgeben();
  s2.ausgeben();
  s1 = s2;
  s2.setVorname("King");
  s1.ausgeben();
 }
```

```
Vorname: King
Nachname: Y
Studiengang: ITI
```

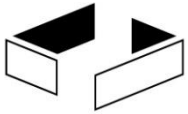


Zusammenfassung



- Objekte sind Grundlagen der objektorientierten Programmierung
- Programme bestehen aus dem
  - Erstellen von Objekten
  - Bearbeiten von Objekten über Methodenaufrufe
  - Weitergeben von Objekten an Methoden über Parameter
- Durch die systematische Bearbeitung aller Objekte entsteht ein Programm
- Es ist Teil der Anforderungsanalyse festzustellen, welche Klassen und Methoden zur Lösung der Aufgabenstellung benötigt werden

Aufgabe (1/5): Druck von Namensschildern



Video

Gegeben Klasse Interaktionsbrett

- **`new Interaktionsbrett()`**

  Konstruktor für neues Objekt mit eigener Ausgabefläche

- **`void neuerText(int x, int y, java.lang.String text)`**

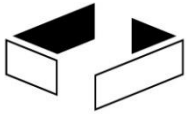
  Methode zur Ausgabe eines Textes, an Position x,y.

- **`void neuesRechteck(int x, int y, int breite`**
  `, int hoehe)`

  Methode zum Zeichnen eines neuen Rechtecks an Pos x,y.

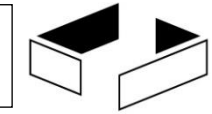
- Aufgabe: Ausgabe von Teilnehmenden bestehend aus Name und Firma untereinander in Ausgabefläche
- Ansatz: Überlege sinnvolle Klassen (Teilnehmend, Druckseite)

Aufgabe (2/5)

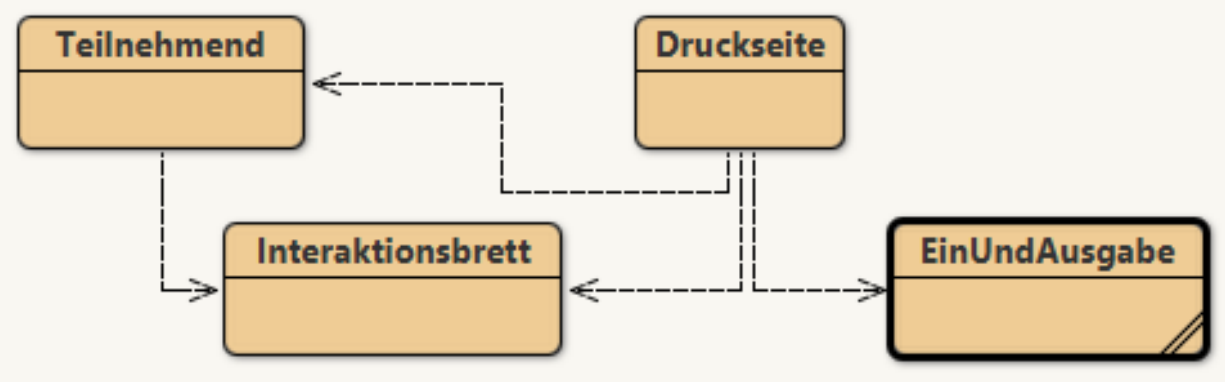


```
class Teilnehmend{
  String name;
  String firma;

  Teilnehmend(String name, String firma){
    this.name = name;
    this.firma = firma;
  }

  void zeichnen(Interaktionsbrett ib, int x, int y){
    ib.neuesRechteck(x, y, 100, 40);
    ib.neuerText(x+2, y+15, this.name);
    ib.neuerText(x+2, y+35, this.firma);
  }
}
```

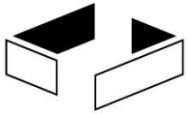
Aufgabe (3/5)



```
class Druckseite{
  Interaktionsbrett ib;
  int yPosition = 5; // Position der nächsten Ausgabe

  Druckseite(){
    this.ib = new Interaktionsbrett();
  }

  void drucken(Teilnehmend t){
    t.zeichnen(this.ib, 5, this.yPosition);
    this.yPosition = this.yPosition + 50;
  }
```

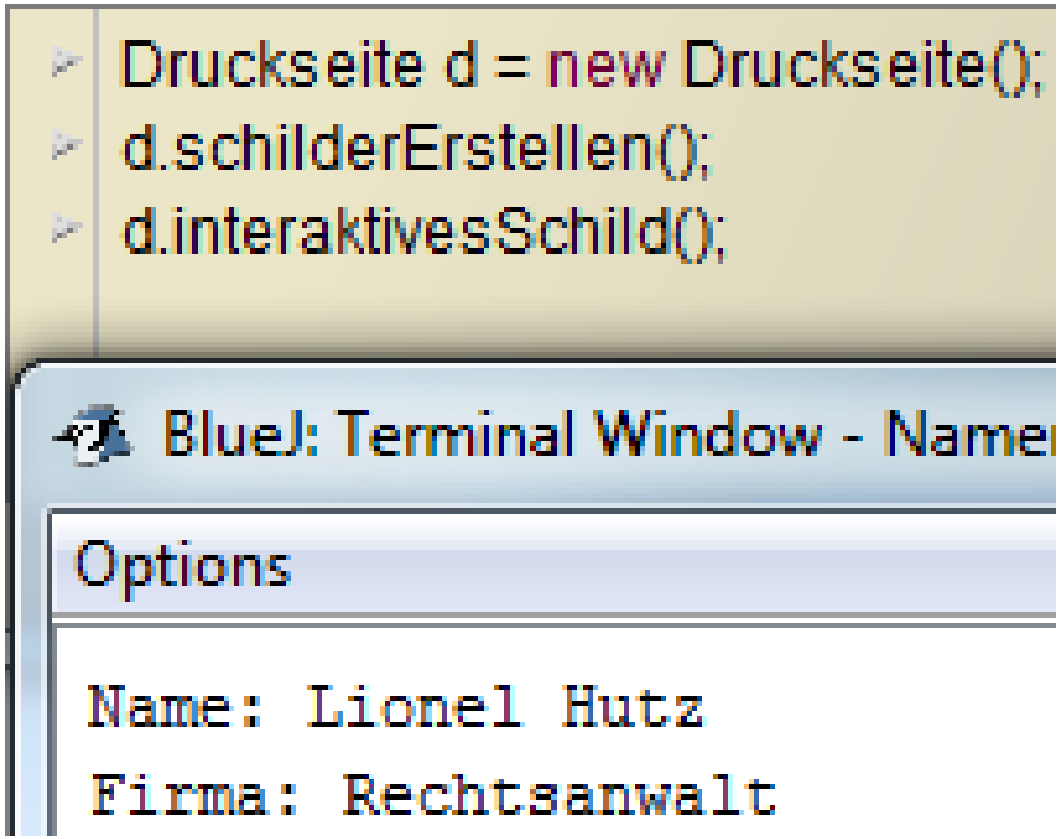


Aufgabe (4/5)



```
  void schilderErstellen(){
    Teilnehmend t = new Teilnehmend("Carl Carlson"
                                  , "Kraftwerk");
    this.drucken(t); // Erinnerung: verkuerzt nur  drucken(t)
    t = new Teilnehmend("Edna Krabbabel","Grundschule");
    this.drucken(t);
    this.drucken(new Teilnehmend("Clancy Wiggum", "Polizei"));
  }

  void interaktivesSchild(){
    EinUndAusgabe io = new EinUndAusgabe();
    io.ausgeben("Name: ");
    String name = io.leseString();
    io.ausgeben("Firma: ");
    String unternehmen = io.leseString();
    Teilnehmend t = new Teilnehmend(name, unternehmen);
    this.drucken(t);
  }
}
```

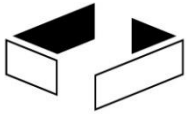
Aufgabe (5/5) - Ergebnis



Diese Folie ist zur Selbstreflexion leer (was kann ich)

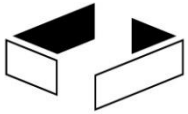
Video

Eingeben, Ausgeben, Übergeben, Zurückgeben



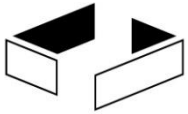
- eingeben: Werte werden bei der Nutzung oder über Eingabegeräte (z. B. Tastatur, Maus, Sensor) eingelesen
- ausgeben: Werte werden auf der Konsole, in einem Display oder einer Datei ausgegeben
- übergeben: Werte, bei Objekten Referenzen, werden zur Nutzung an eine Methode übergeben; die Methode hat dazu eine Parameterliste zur Übernahme
- zurückgeben: Werte bzw. Objekte, werden innerhalb einer Methode berechnet oder ausgewählt und als Ergebnis mit return zurück gegeben, der Typ des Ergebnisses steht vor dem Methodennamen

Klasse Punkt



```
class Punkt {
  int x;
  int y;

  Punkt(int x, int y){
    this.x = x;
    this.y = y;
  }

  int getX() {
    return this.x;
  }

  void setX(int x) {
    this.x = x;
  }

  int getY() {
    return this.y;
  }

  void setY(int y) {
    this.y = y;
  }

  void darstellen(
      Interaktionsbrett ib){
    ib.neuerPunkt(this.x, this.y);
  }
}
```

Eingeben und Zurückgeben eines Punktes



```
// Ausschnitt aus einer Klasse Punktanalyse
  Interaktionsbrett ib = new Interaktionsbrett();
  EinUndAusgabe io = new EinUndAusgabe();

  Punkt punktEingeben(){

      this.io.ausgeben("X-Wert: ");

      int x = this.io.leseInteger();

      this.io.ausgeben("Y-Wert: ");

      int y = this.io.leseInteger();

      return new Punkt(x,y);

  }
```

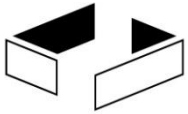
ausgeben

übergeben (String)

eingeben

zurückgeben

Ausgeben und Übergeben eines Punktes



```
void markieren(Punkt pkt){
  this.ib.neuerPunkt(pkt.getX(), pkt.getY());
  this.ib.neuerKreis(pkt.getX()-3, pkt.getY()-3,3);
  this.ib.neuerKreis(pkt.getX()-6, pkt.getY()-6,6);
}

void benutzen(){
  Punkt p1;
  p1 = this.punktEingeben();
  this.markieren(p1);
  Punkt p2 = this.punktEingeben();
  this.markieren(p2);
}
```

ausgeben

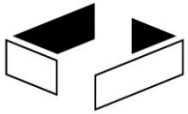
übergeben

```
X-Wert: 10
Y-Wert: 10
X-Wert: 25
Y-Wert: 12
```

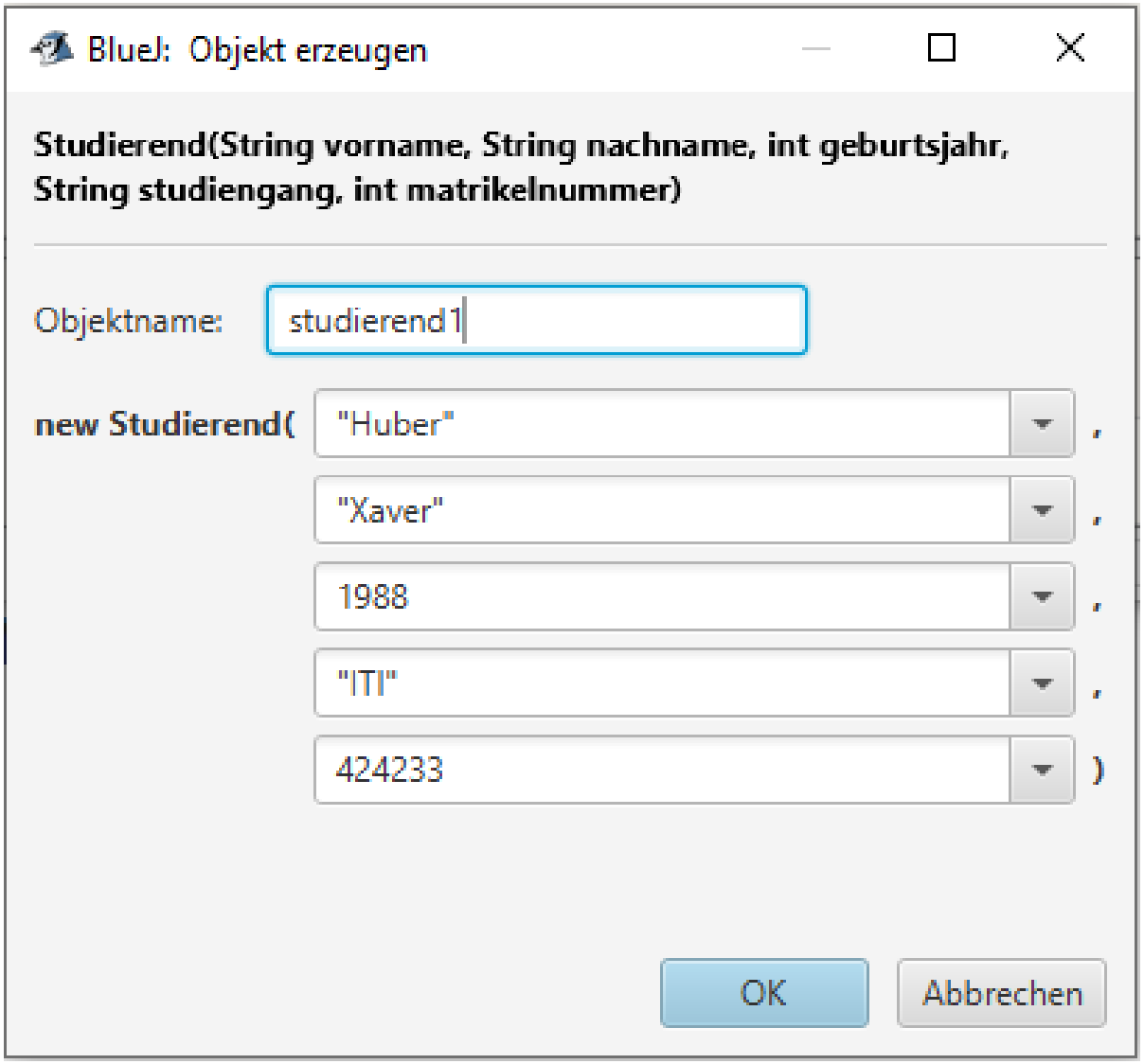
Video

Debugger

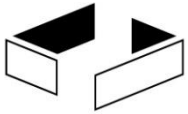
Bayrisches Problem



- Studi hat sich bei Einschreibung mit „Jo mei, I bian der Huber Xaver“ vorgestellt
- Wunsch: Methode zum Vertauschen von Vor- und Nachnamen



(schlechter) Versuch 1 (1/3)



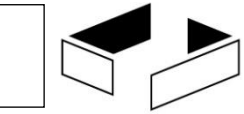
- neue Methode in Studierend

```
void vertauscheNamen(){
   this.vorname = this.nachname;
   this.nachname = this.vorname;
}
```

- neue Klasse zum Ausprobieren

```
class Studierendspielerei{

  Studierend namenstausch(){
    Studierend xaver= new Studierend("Huber", "Xaver", 1988
                                ,"ITI", 424223);
    xaver.vertauscheNamen();
    return xaver;
  }
}
```

(schlechter) Versuch 1 (2/3)



- Ausführung von vertauscheNamen() liefert

BlueJ: Methodenergebnis

Studierend namenstausch()

studiere1.namenstausch() zurückgegeben:

Inspiziere

Studierend

: Studierend

private String vorname	"Xaver"
private String nachname	"Xaver"
private int geburtsjahr	1988
private String studiengang	"ITI"
private int matrikelnummer	424223

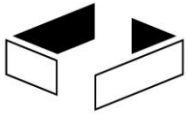
Inspiziere

Hole

Zeige Klassenvariablen

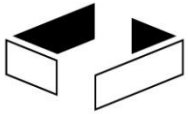
Schließen

(schlechter) Versuch 1 (3/3)



Variable / Anweisung	xaver				
	vorname	nachname	gjahr	stud	matnr
`Studierend xaver= new Studierend( "Huber", "Xaver", 1988, "ITI",424223)`	„Huber“	„Xaver“	1988	„ITI“	424223
`xaver .vertauscheNamen()`					
`vertauscheNamen()`	„Huber“	„Xaver“	1988	„ITI“	424223
`this.vorname = this.nachname`	„Xaver“	„Xaver“	1988	„ITI“	424223
`this.nachname = this.vorname`	„Xaver“	„Xaver“	1988	„ITI“	424223
`return xaver`	„Xaver“	„Xaver“	1988	„ITI“	424223

(guter) Versuch 2 (1/3)

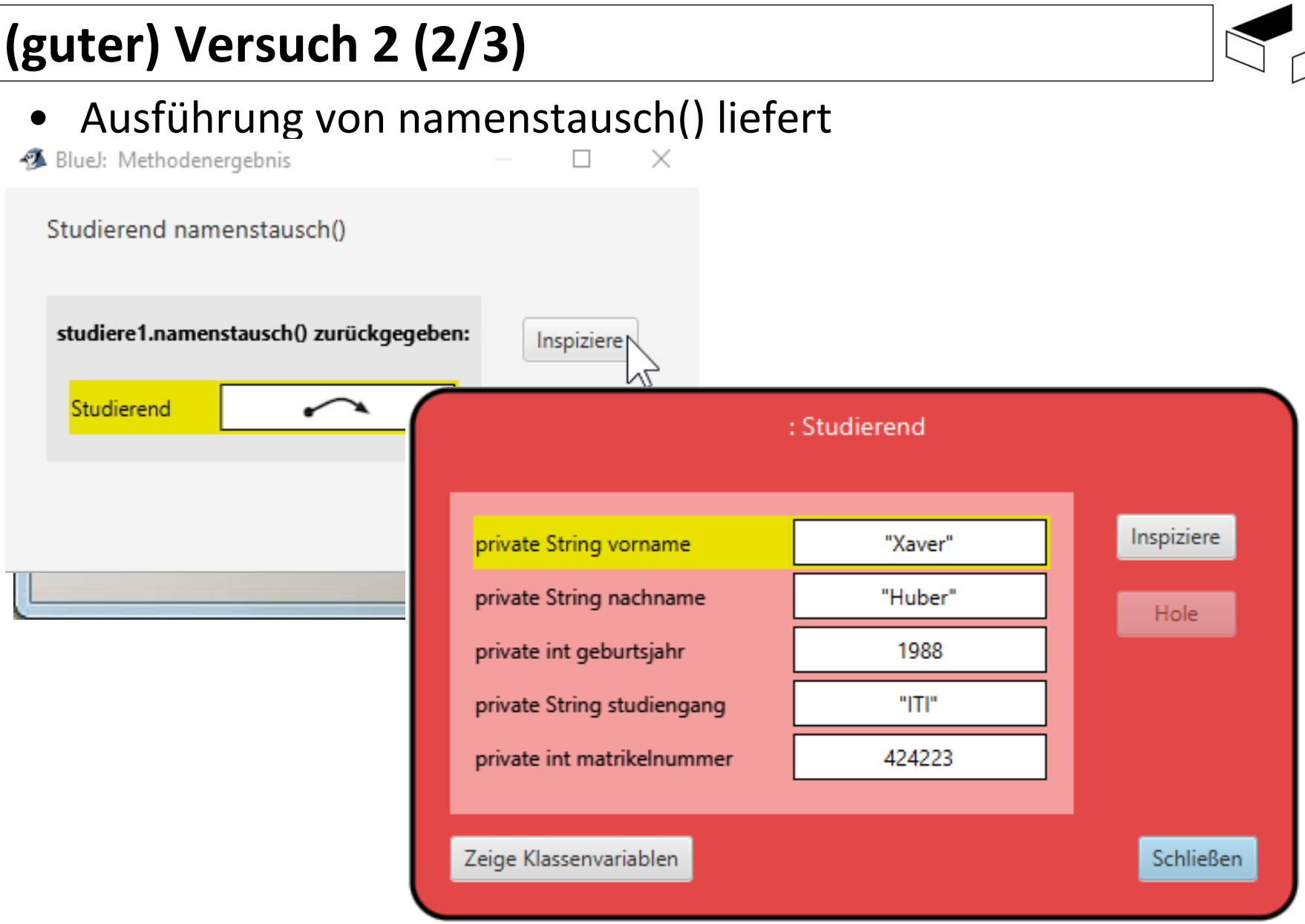


Video

- Nutzung einer lokalen Variablen

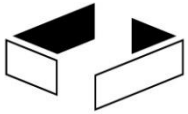
```
void vertauscheNamen(){
        String tmp = this.vorname;
        this.vorname = this.nachname;
        this.nachname = tmp;
}
```

(guter) Versuch 2 (2/3)



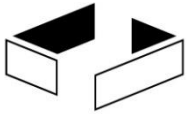
- Ausführung von namenstausch() liefert

(guter) Versuch 2 (3/3)



Variable / Anweisung	xaver					tmp
	vorname	nachname	gjahr	stud	matnr	
`Studierend xaver= new Studierend("Huber", "Xaver", 1988, "ITI", 424223)`	„Huber“	„Xaver“	1988	„ITI“	424223	
`xaver .vertauscheNamen()`	„Huber“	„Xaver“	1988	„ITI“	424223	
`vertauscheNamen()`	„Huber“	„Xaver“	1988	„ITI“	424223	
`String tmp = this.vorname`	„Huber“	„Xaver“	1988	„ITI“	424223	„Huber“
`this.vorname = this.nachname`	„Xaver“	„Xaver“	1988	„ITI“	424223	„Huber“
`this.nachname= tmp`	„Xaver“	„Huber“	1988	„ITI“	424223	„Huber“
`return xaver`	„Xaver“	„Huber“	1988	„ITI“	424223	

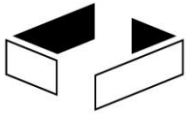
Nutzung des Debuggers (1/6) - Vorbereitung



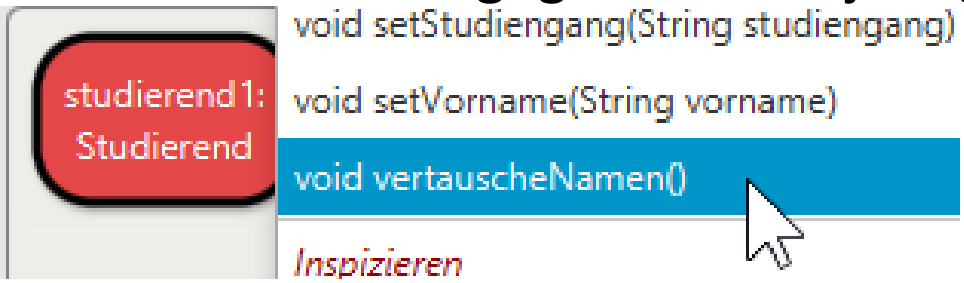
- letzte Folie zeigt, dass genaue Verfolgung der Variablenwerte wichtiges Hilfsmittel sein kann
- Ansatz ist, über Debugger (direkt übersetzt: Entwanzer) die Abarbeitung zu verfolgen
- genauer: Zeilen, ab denen der Debugger die genaue Situation zeigen soll, werden markiert (Break Point, Klick am linken Rand des Editors, nur nach "Compile„ (Übersetzen) sichtbar)

```
29      void vertauscheNamen(){
STOP        String tmp = this.vorname;
31          this.vorname = this.nachname;
32          this.nachname = tmp;
33      }
```

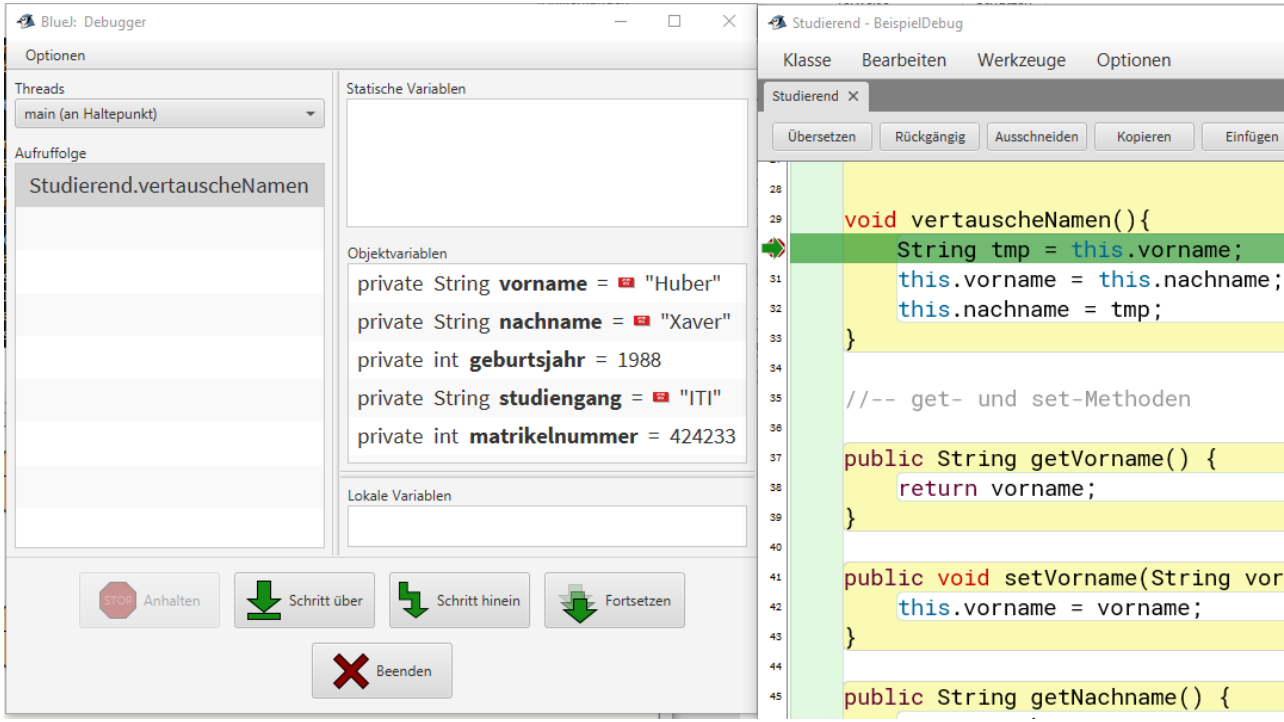
Nutzung des Debuggers (2/6) – Start der Nutzung



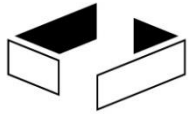
- Methode wird für gegebenes Objekt gestartet



- markierte Zeile wird erreicht; es öffnet sich Debugger und Editor (zeigt immer aktuelle Zeile an)



Nutzung des Debuggers (3/6) – mögliche Aktionen



Typische Bedienmöglichkeiten eines Debuggers:

Step: nächste Anweisung ausführen

Step Into: bei anstehenden Methodenaufrufen kann man in die Methode hineinspringen

Continue: verlasse Debug-Modus und lasse Programm "normal" weiterlaufen, bis nächster Break Point erreicht wird

BlueJ: Debugger

Optionen

Threads

main (an Haltepunkt)

Aufruffolge

Studierend.vertauscheNamen

Statische Variablen

Objektvariablen

private String **vorname** = "Huber"

private String **nachname** = "Xaver"

private int **geburtsjahr** = 1988

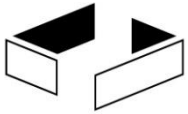
private String **studiengang** = "ITI"

private int **matrikelnummer** = 424233

Lokale Variablen

Anhalten | Schritt über | Schritt hinein | Fortsetzen | Beenden

Nutzung des Debuggers (4/6) - Anzeige



Threads

main (an Haltepunkt)

Aufruffolge

Studierend.vertauscheNamen

Statische V

welche Klassenvariablen (später!) gibt es

welche Methoden laufen zur Zeit (Methoden können ja Methoden aufrufen)

welche Objektvariablen gibt es (detailliertere Inhalte) durch Anklicken

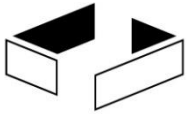
welche lokalen Variablen gibt es

Objektvariablen

```
private String vorname = "Huber"
private String nachname = "Xaver"
private int geburtsjahr = 1988
private String studiengang = "ITI"
private int matrikelnummer = 424233
```

Lokale Variablen

Nutzung des Debuggers (5/6) – Ausführung 1/2



Instance variables

private String **vorname** = "Huber"
private String **nachname** = "Xaver"
private int **geburtsjahr** = 1988
private String **studiengang** = "ITI"
private int **matrikelnummer** = 424233

Local variables

Instance variables

private String **vorname** = "Huber"
private String **nachname** = "Xaver"
private int **geburtsjahr** = 1988
private String **studiengang** = "ITI"
private int **matrikelnummer** = 424233

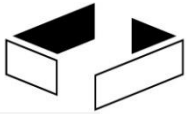
Local variables

String **tmp** = "Huber"

```
29    void vertauscheNamen(){
          String tmp = this.vorname;
31        this.vorname = this.nachname;
32        this.nachname = tmp;
```

```
29    void vertauscheNamen(){
          String tmp = this.vorname;
          this.vorname = this.nachname;
32        this.nachname = tmp;
```

Nutzung des Debuggers (6/6) – Ausführung 2/2



Instance variables

```
private String vorname = "Xaver"
private String nachname = "Xaver"
private int geburtsjahr = 1988
private String studiengang = "ITI"
private int matrikelnummer = 424233
```

Local variables

```
String tmp = "Huber"
```

Instance variables

```
private String vorname = "Xaver"
private String nachname = "Huber"
private int geburtsjahr = 1988
private String studiengang = "ITI"
private int matrikelnummer = 424233
```

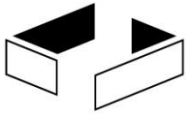
Local variables

```
String tmp = "Huber"
```

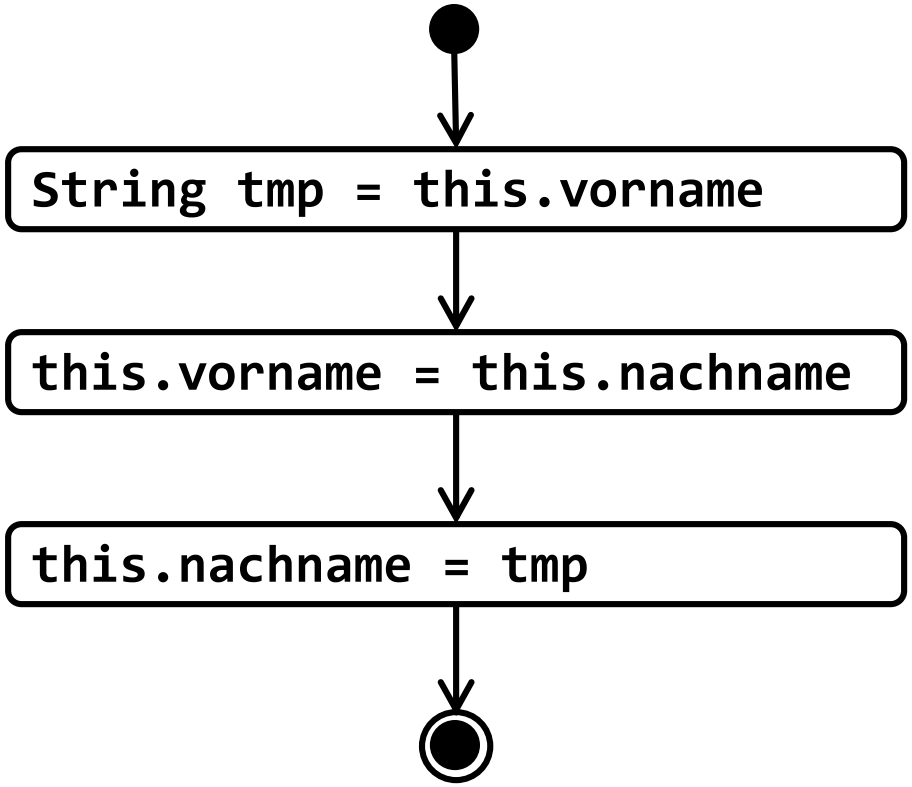
```
29    void vertauscheNamen(){
STOP      String tmp = this.vorname;
31        this.vorname = this.nachname;
          this.nachname = tmp;
33    }
```

```
29    void vertauscheNamen(){
STOP      String tmp = this.vorname;
31        this.vorname = this.nachname;
32        this.nachname = tmp;
      }
```

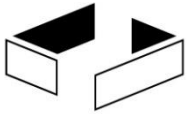
Visualisierung im Aktivitätsdiagramm - Sequenz



- Startknoten (es kann nur einen geben)
- gerichtete Kanten zeigen Ablauf
- abgerundete Rechtecke enthalten Anweisungen (Aktionen)
- Endknoten zur Terminierung (es kann später mehrere geben)



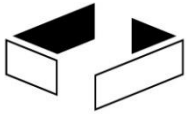
Aufrufstapel (1/2)



- wird innerhalb einer Methode m1 eine andere Methode m2 aufgerufen, wird m2 vollständig abgearbeitet, bis m1 fortgesetzt wird

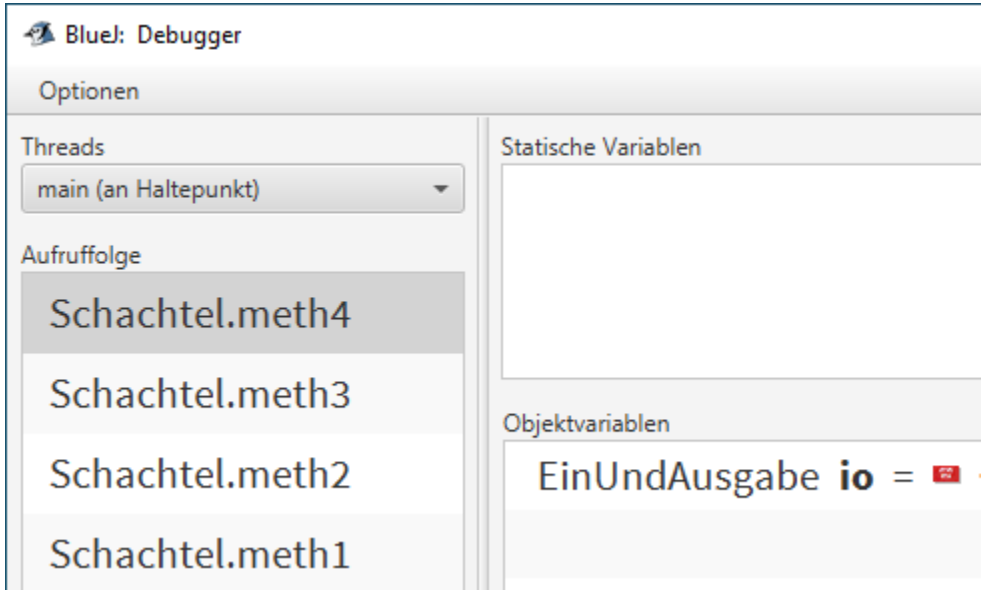
```
class Schachtel{
  EinUndAusgabe io = new EinUndAusgabe();

  void meth1(){
    this.io.ausgeben("start meth1\n");
    this.meth2();
    this.io.ausgeben("ende meth1\n");
  }

  void meth2(){
    this.io.ausgeben("start meth2\n");
    this.meth3();
    this.io.ausgeben("ende meth2\n");
  }
```

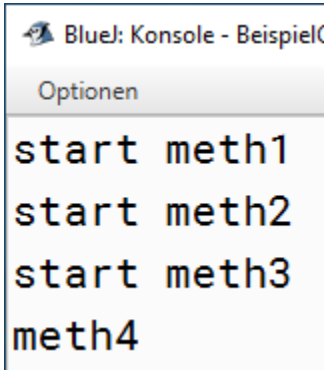
Aufrufstapel (2/2)



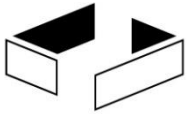
```
  void meth3(){
    this.io.ausgeben("start meth3\n");
    this.meth4();
    this.io.ausgeben("ende meth3\n");
  }

  void meth4(){
    this.io.ausgeben("meth4\n");
  }
}
```

```
void meth4(){
  this.io.ausgeben("meth4\n");
}
```





String

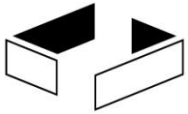


Video

- String ist auch Klasse mit "normalen" Objekten
- genauer "fast normal"; alle Werte sind Konstanten (immutable objects)
- d. h. Objekte können nicht geändert werden
- Beispiel: Klasse String hat zwar viele Methoden, damit wird kein String verändert, sondern immer neues String-Objekt berechnet (anschaulich: keine set-Methoden)

```
String s = "Hallodele";
s.replaceAll("e","au")
"Hallodaulau"   (String)
s
"Hallodele"   (String)
```

Strings unveränderbar

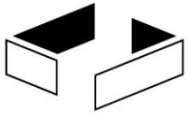


- Mit String-Methoden wird nie String selbst geändert
- wenn, wird neues Objekt zurückgegeben

```
String s ="Hai";
s
"Hai"  (String)
s.replace("ai","allo")
"Hallo"  (String)
s
"Hai"  (String)
String t = s.replace("ai","aha");
t
"Haha"  (String)
s
"Hai"  (String)
```

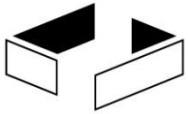
- Anmerkung: gibt verwandte Klassen StringBuffer und StringBuilder, deren Objekte verändert werden können

Datum als Immutable (unveränderbar) - Ausblick



```
class Datum {   // normale get-Methoden (fehlen auf Folie)
  int tag;
  int monat;
  int jahr;

  Datum (int tag, int monat, int jahr){
    this.tag = tag;
    this.monat = monat;
    this.jahr = jahr;
  }

  Datum setTag(int tag) {
    return new Datum (tag, this.monat, this.jahr);
  }

  Datum setMonat(int monat) {
    return new Datum (this.tag, monat, this.jahr);
  }

  Datum setJahr(int jahr) {
    return new Datum (this.tag, this.monat, jahr);
  }
```

Default-Werte für Objektvariablen – null



- bei einer Objektvariablen **`String name;`** ohne Initialisierung hat diese Variable den wert „undefiniert“, der formal null heißt, auch
- **`String name = null;`**
- da null kein Objekt, kann keine Methode oder Berechnung ausgeführt werden (NullPointerException)

```
String name = null;
name.length()
Error: null
name = "Hai";
name.length()
3   (int)
```

BlueJ: Terminal Window - Beispiel07m

Options

```
java.lang.NullPointerException
```

Video

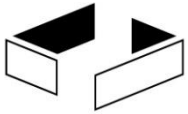
Beispiel

Beispiel

Video

Objektweitergabe

Erinnerung Klasse Studierend

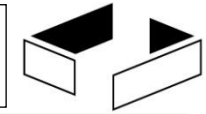


- Anfang

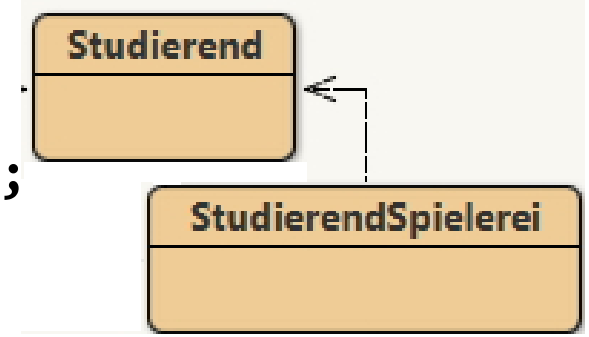
```
class Studierend{
    String vorname ="Eva";
    String nachname ="Mustermann";
    int geburtsjahr = 1990;
    String studiengang = "IMI";
    int matrikelnummer = 232323;

    Studierend(String vorname, String nachname
                ,int geburtsjahr, String studiengang
                ,int matrikelnummer) {...

    Studierend(){
    }...
```

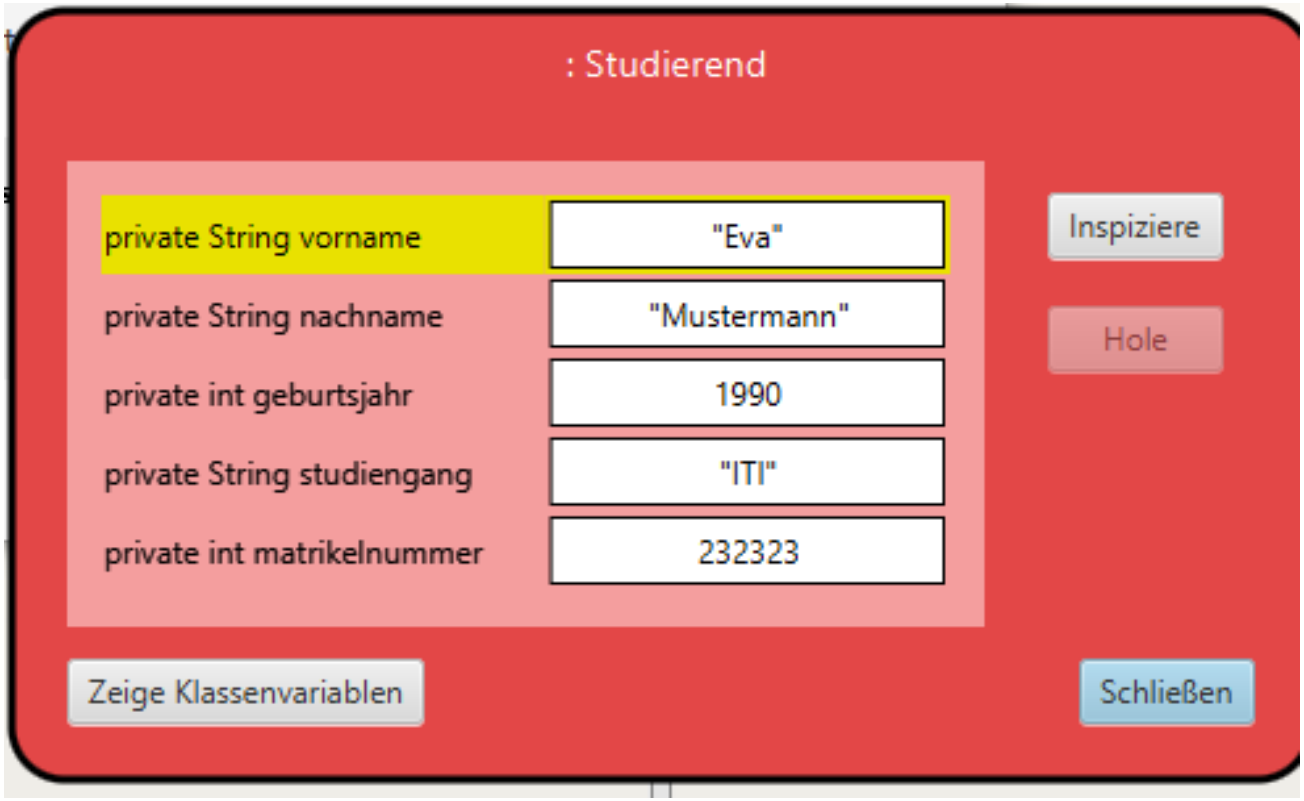
- weiter mit get- und set-Methoden für alle Objektvariablen

Möglichkeit 1: Direkte Erzeugung in anderer Klasse

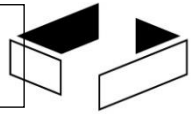


```
class StudierendSpielerei{
  Studierend standardITIStudi(){
    Studierend ergebnis = new Studierend();
    ergebnis.setStudiengang("ITI");
    return ergebnis;
  }
}
```





Möglichkeit 2: Übergabe in Methode/ in Konstruktor

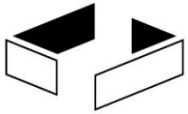


Aufgabe: Daten von Studierenden sollen anonymisiert werden

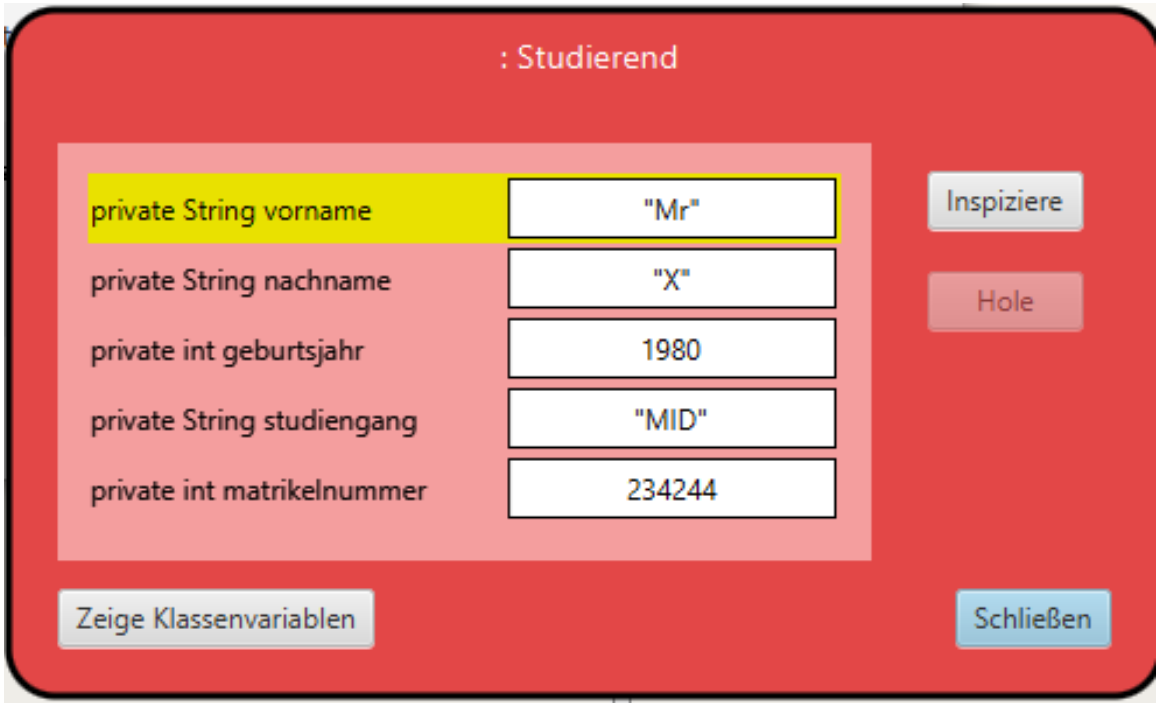
Ansatz: Wähle Dummy-Studierend, dessen Daten in konkrete Studierend-Objekte übertragen werden sollen (hier nur Vor- und Nachname; gäbe natürlich andere Möglichkeiten)

```
class Anonymisierer {
  Studierend dummy;

  Anonymisierer(Studierend dummy){
      this.dummy=dummy;
  }

  void anonymisieren(Studierend zuAendern){
      zuAendern.setVorname(this.dummy.getVorname());
      zuAendern.setNachname(this.dummy.getNachname());
  }
}
```

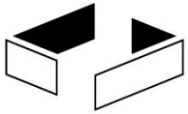
Nutzung in StudierendSpielerei



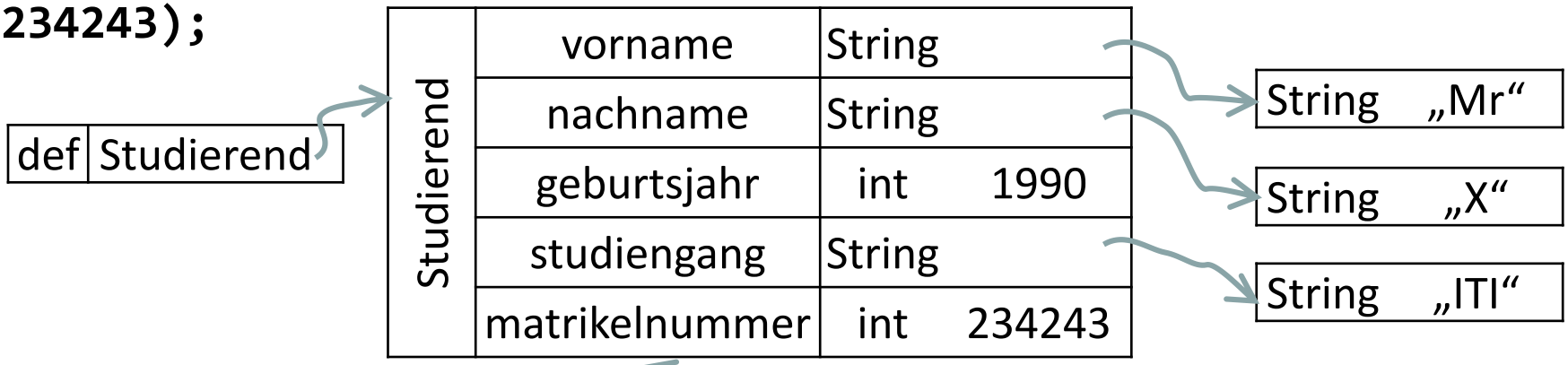
```
class StudierendSpielerei{

  Studierend anonymAusprobieren(){
    Studierend def = new Studierend("Mr","X",1990,"ITI",234243);
    Anonymisierer ano = new Anonymisierer(def);
    Studierend test = new Studierend("Ms","Y",1980,"MID",234244);
    ano.anonymisieren(test);
    return test;
  }
}
```



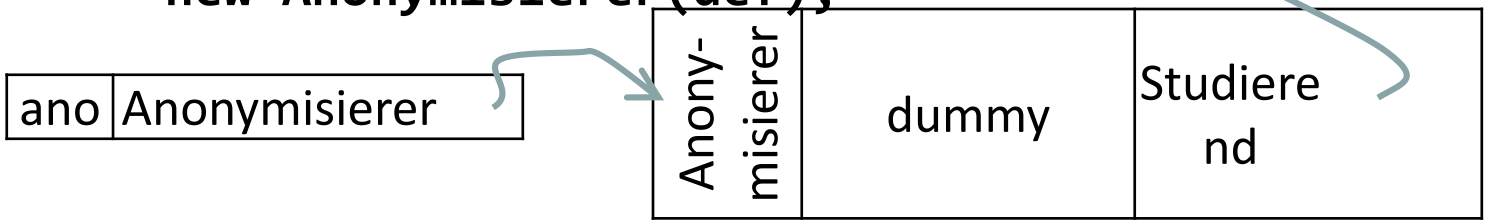
Visualisierung des Ablaufs (1/3)



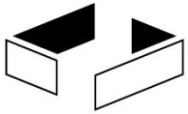
```
Studierend anonymAusprobieren(){
  Studierend def = new Studierend("Mr","X", 1990, "ITI",
234243);
```



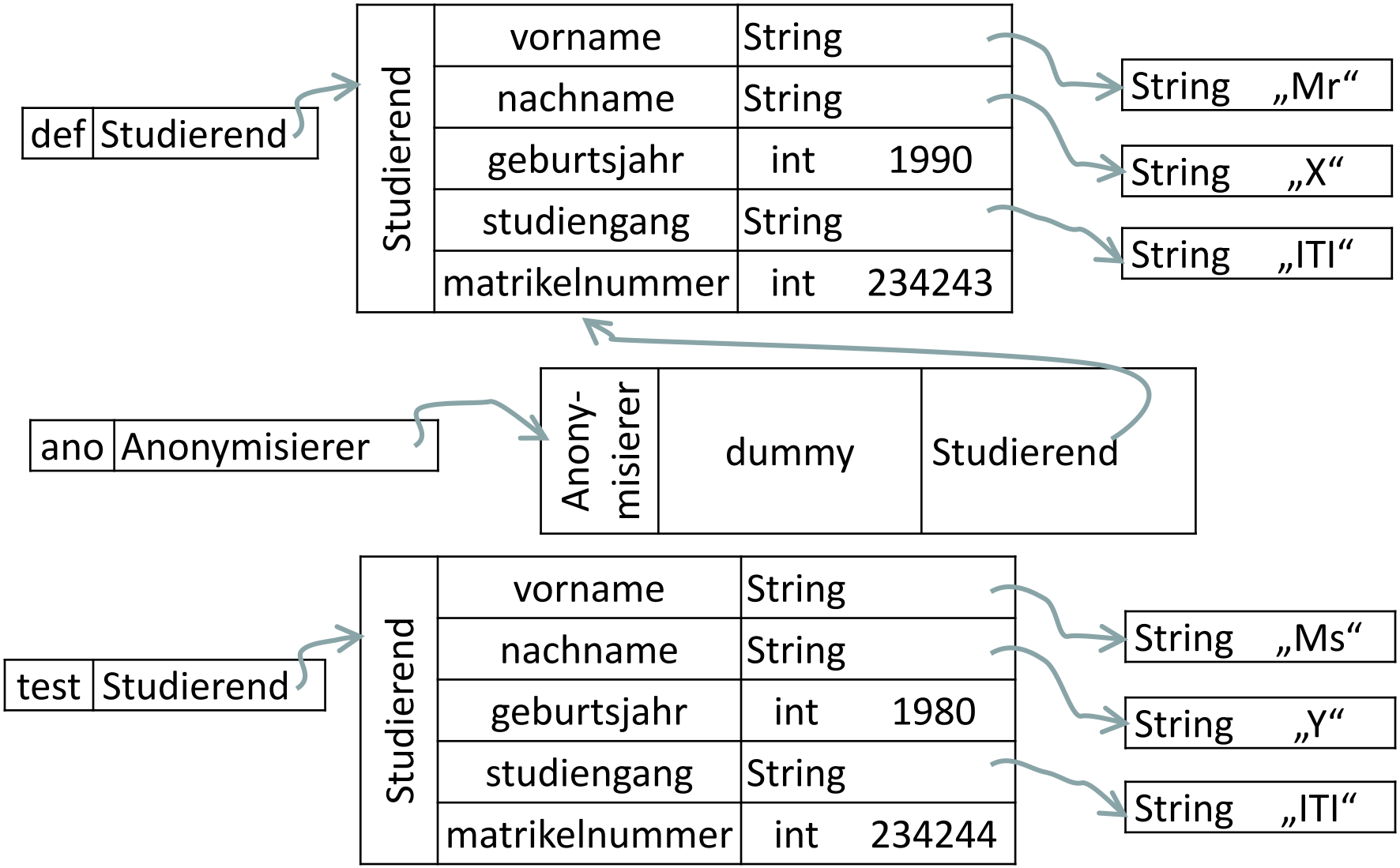

```
  Anonymisierer ano =
      new Anonymisierer(def);
```



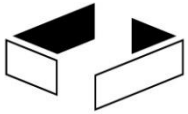
Visualisierung des Ablaufs (2/3)



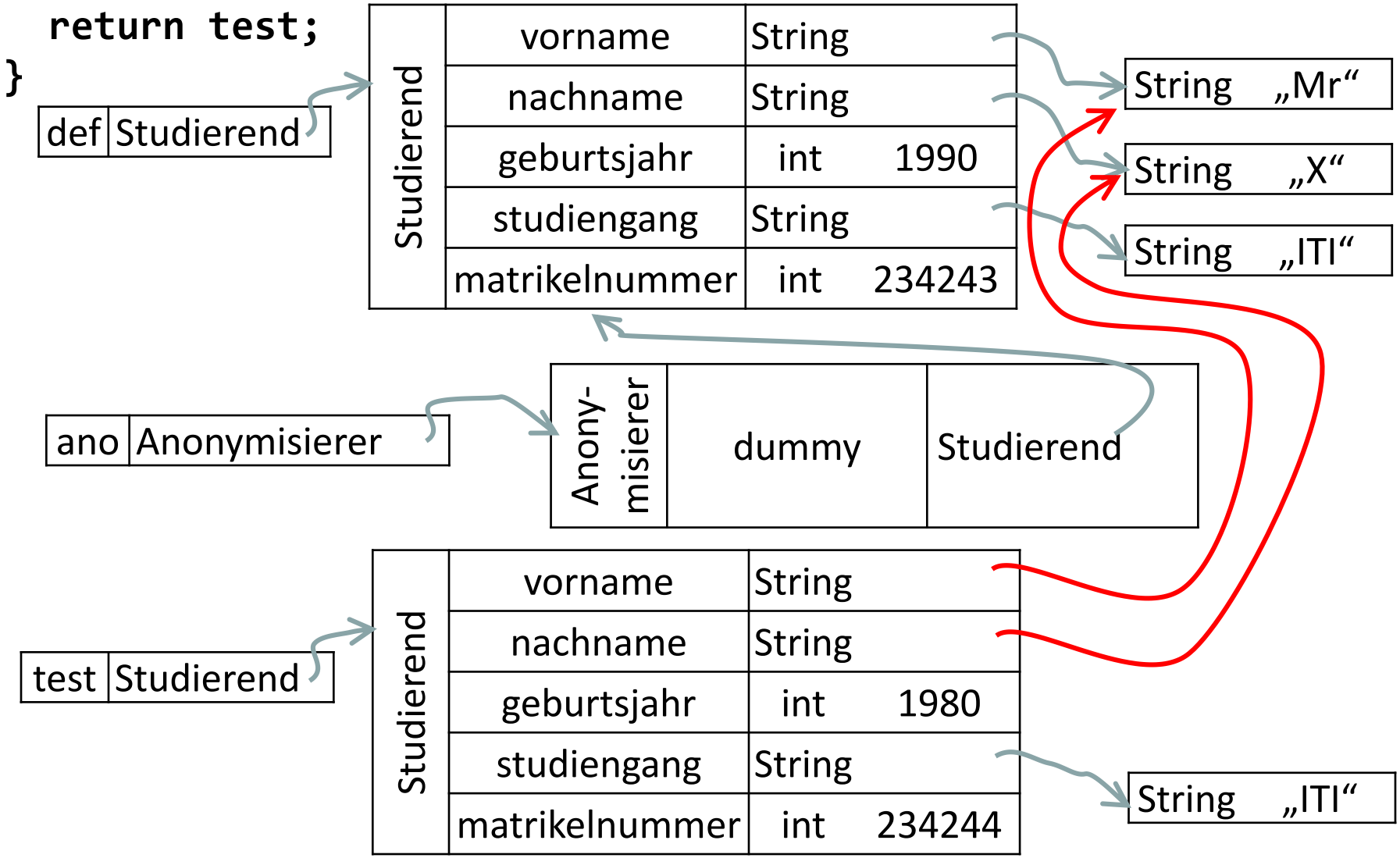
```
Studierend test = new Studierend("Ms","Y",1980,"MID",234244);
```



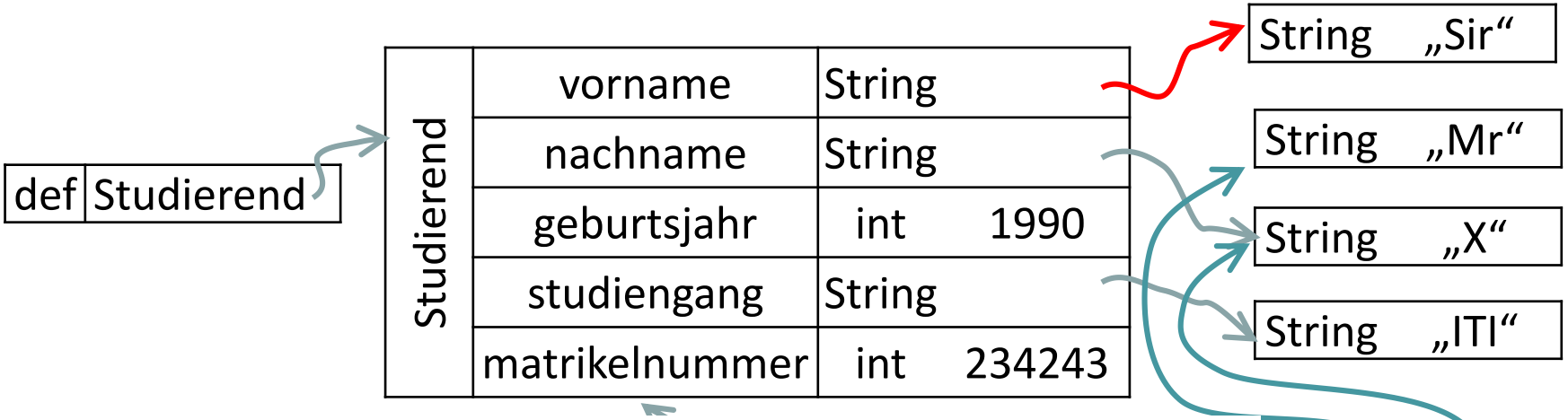
Visualisierung des Ablaufs (3/3)



```
  ano.anonymisieren(test);
  return test;
}
```



Sehr gute Zwischenfrage

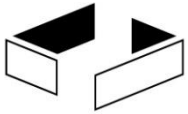


- Wenn mit Referenzen gearbeitet wird und vor return-Befehl def.setVorname("Sir") stehen würde, müsste doch auch der Vorname von test verändert werden, oder?

```
def.setVorname("Sir");
return test;
```

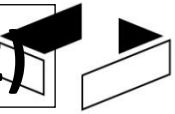
- Antwort nein, da **`"Sir"`** genauer für **`new String("Sir")`** steht und ein neues Objekt entsteht

War die vorherige Frage nicht doch blöd?



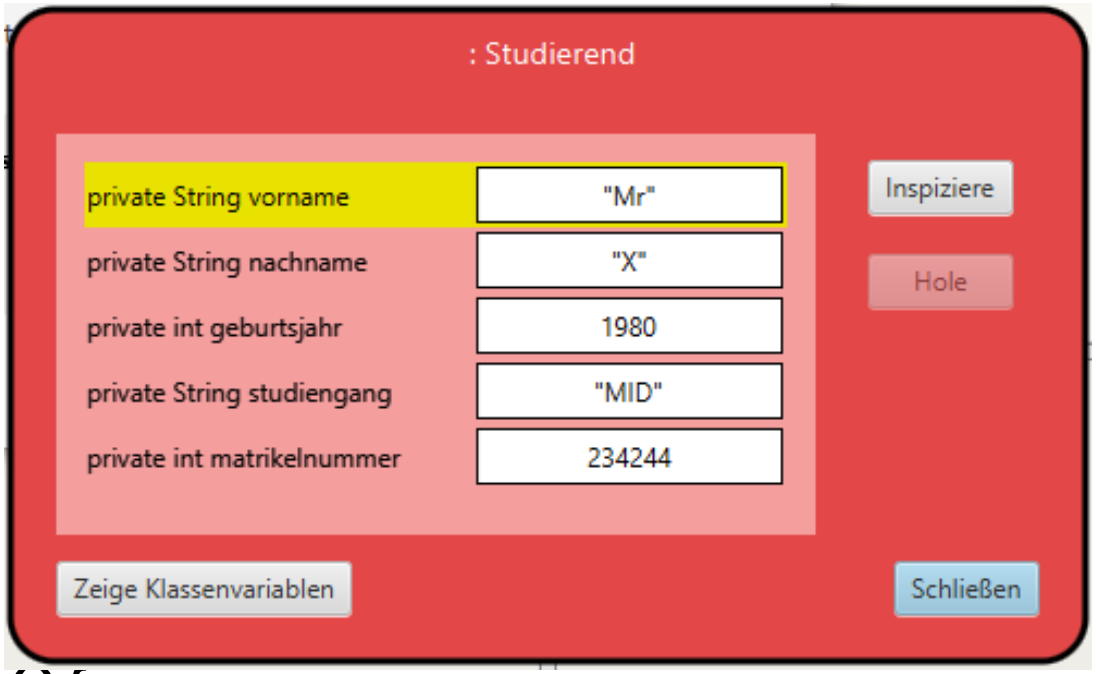
  - Nein, da sie sich auf das sonst übliche Verhalten bei Referenzen bezieht; man aber die implizite Objekterzeugung beachten muss
- Wenn in def der erste Buchstabe des Vornamens auf "C" geändert wird, müsste damit auch der Vorname von test geändert werden, oder?
  - Richtig, allerdings haben Strings keine Methoden, um einen String selbst zu verändern; es sind ähnlich wie int-Werte auch unveränderbare Objekte (immutable objects)
- Oh, schon wieder eine Besonderheit
  - Zunächst nicht; schreibt man eine Klasse z. B. ohne set-Methoden, kann dieses Objekt auch nicht verändert werden

(k)eine Besonderheit: Objekt übergibt sich selbst (1/2)



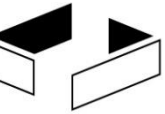
```
class Anonymisierer {
  Studierend dummy;

  Anonymisierer(Studierend dummy){
      this.dummy=dummy;
  }

  void anonymisieren(Studierend zuAendern){
      zuAendern.setVorname(this.dummy.getVorname());
      zuAendern.setNachname(this.dummy.getNachname());
  }

  void leseDummyAus(StudierendSpielerei spielerei){
      this.dummy = spielerei.getStudi();
  }
}
```

(k)eine Besonderheit: Objekt übergibt sich selbst (2/2)

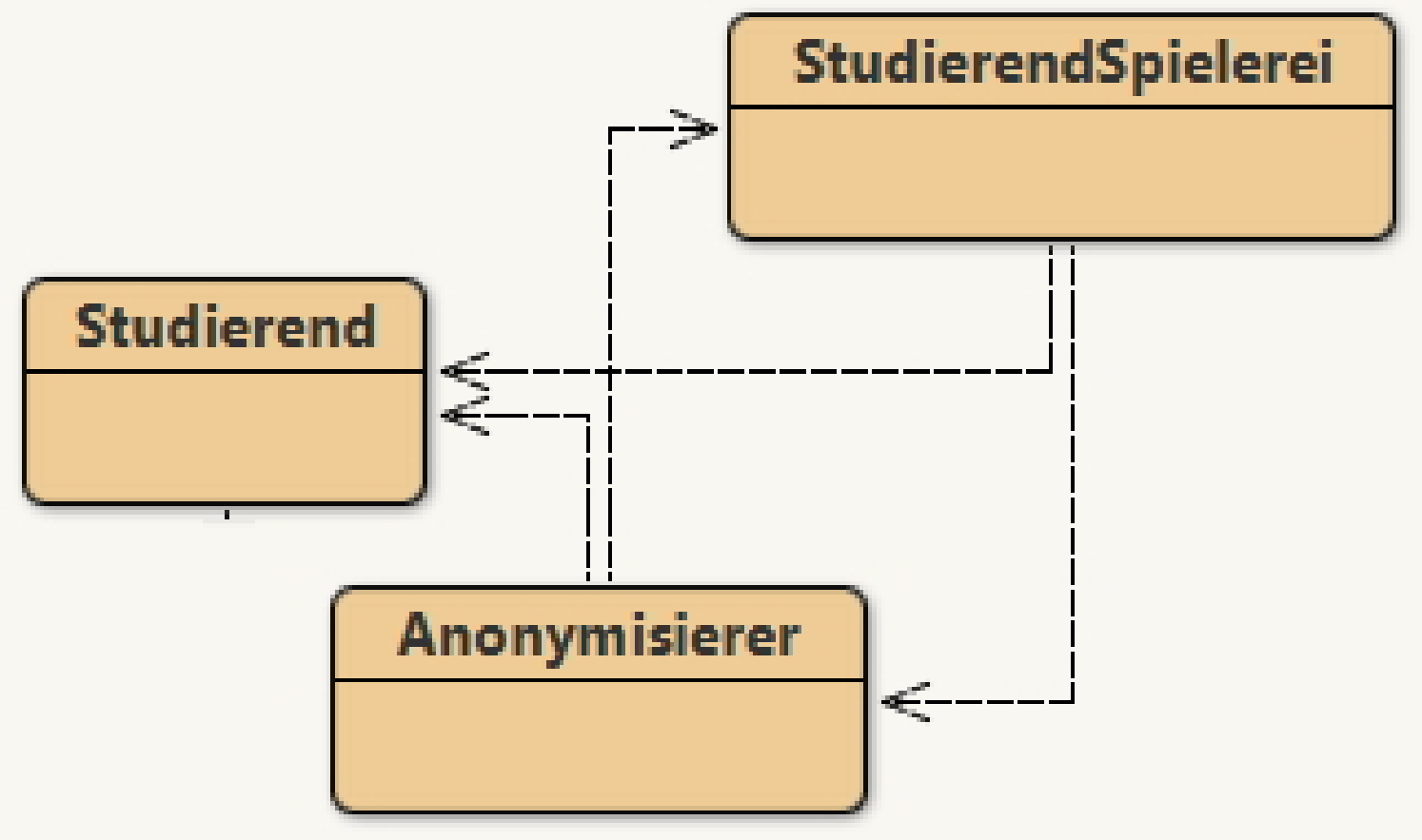



```
class StudierendSpielerei{
  Studierend studi;

  Studierend getStudi(){
      return this.studi;
  }

  Studierend sichUebergeben(){
    this.studi = new Studierend("Mr","X",1990,"IMI",234243);
    Anonymisierer ano = new Anonymisierer(null);
    ano.leseDummyAus(this);
    Studierend test = new Studierend("Ms","Y",1980,"MID",234244);
    ano.anonymisieren(test);
    return test;
  }
```

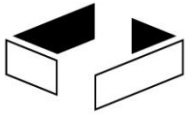
Abhängigkeiten



- Generell wird meist gefordert, dass nicht gleichzeitig eine Klasse A eine Klasse B und eine Klasse B eine Klasse A nutzt



Wertkopien und Referenzen (1/5)



Visualisierung mit Objektspeicherdiagrammen

- elementare Typen: immer Wertkopien

  `int x = 40 + 2;` // rechte Seite auswerten

	int	42

- mit Variable verknüpfen

x	int	42

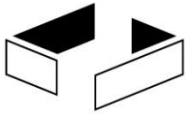
  `int y = x - 1;`// rechte Seite auswerten

x	int	42
	int	41

- mit Variable verknüpfen

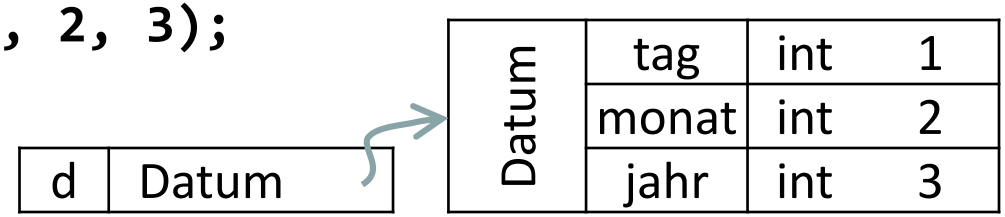
x	int	42
y	int	41

Wertkopien und Referenzen (2/5)

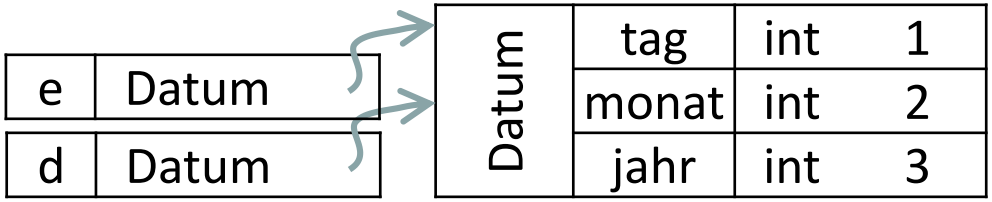


- bei Objekten werden Referenzen genutzt

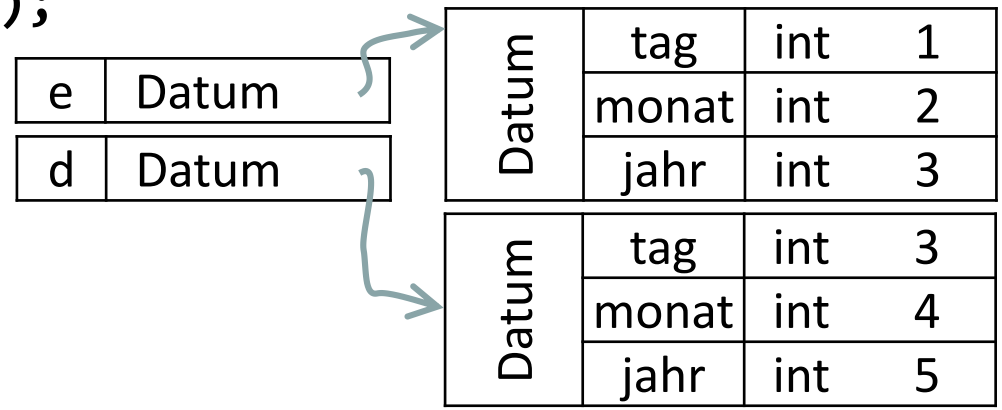
```
Datum d = new Datum(1, 2, 3);
```



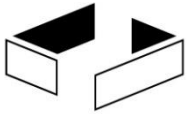

```
Datum e = d;
```



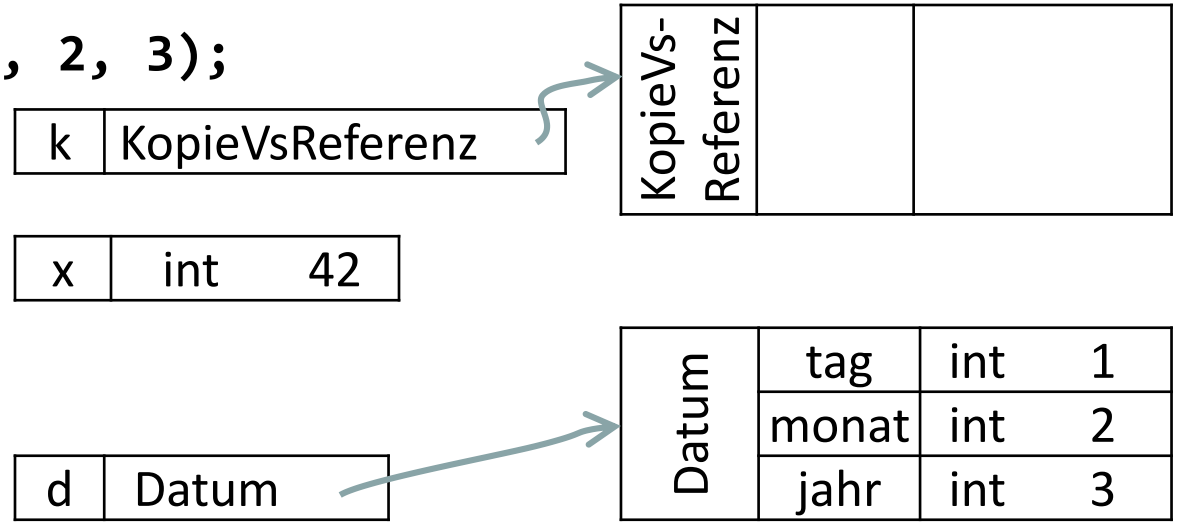

```
d = new Datum(3, 4, 5);
```



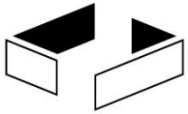
Wertkopien und Referenzen (3/5)



```
class KopieVsReferenz {
  void mach(int w, Datum z) {
    w = 43;
    z.setTag(32);
  }
}

KopieVsReferenz k = new KopieVsReferenz(); //Bsp_nutzung
int x = 42;
Datum d = new Datum(1, 2, 3);
```



Wertkopien und Referenzen (4/5)



```
k.mach(x, d);
```

Wertekopie

Referenz

```
void mach(int w, Datum z) {
  w = 43;
  z.setTag(32);
}
```

Situation nach Start derMethode

k	KopieVsReferenz

x	int	42

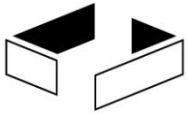
d	Datum
z	Datum

w	int	42

KopieVs-Referenz			

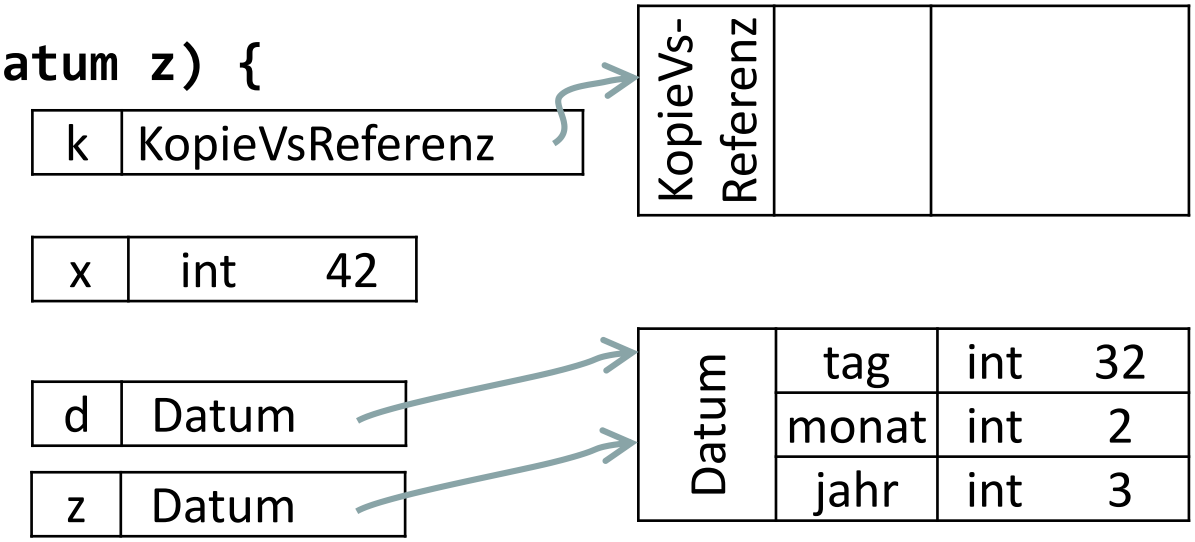
Datum	tag	int 1
	monat	int 2
	jahr	int 3

Wertkopien und Referenzen (5/5)

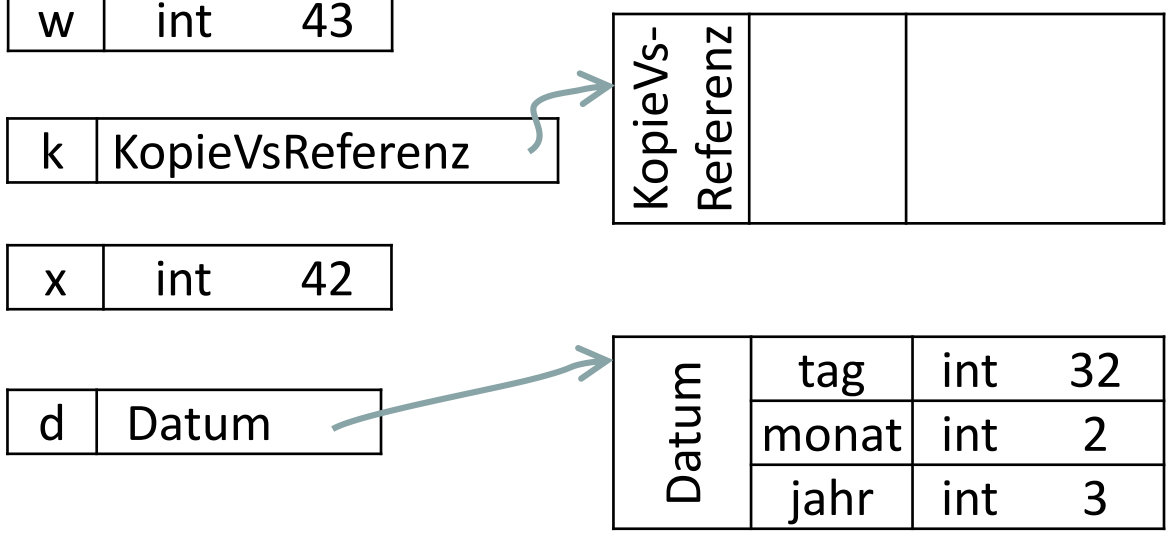


```
void mach(int w, Datum z) {
  w = 43;
  z.setTag(32);
}
```

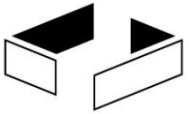
Situation am Ende der Methode



Situation nach Ausführung der Methode (w und z gestorben, Zuweisung an w war damit sinnlos)

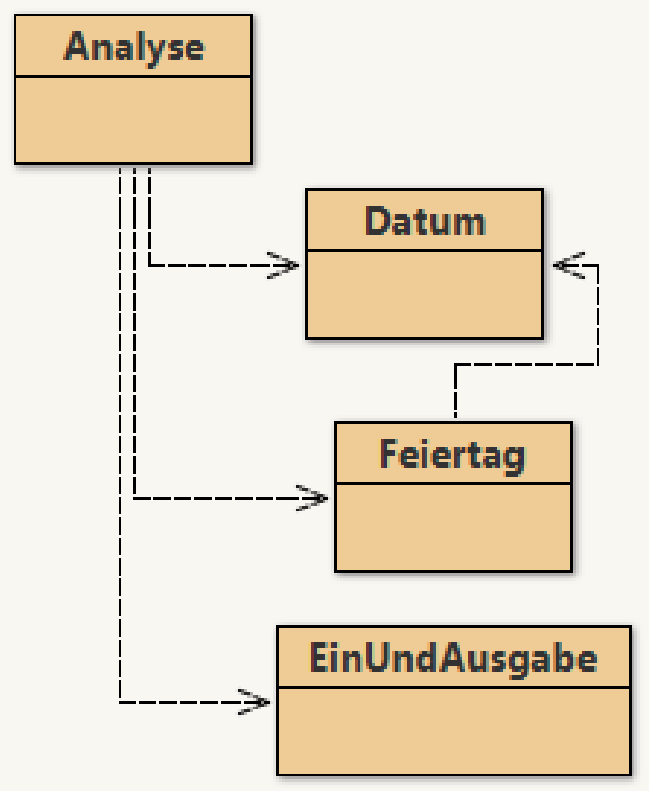


Referenzen nochmals genauer (1/6)

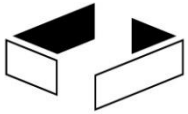


Video

```
class Datum{
    int tag;
    int monat;
    int jahr;

    Datum (int tag, int monat, int jahr){
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
    }

    Datum(){}

    String alsText(){
        return this.tag + "." + this.monat + "." + this.jahr;
    }

  // get- und set-Methoden fuer alle Objektvariablen
```

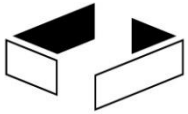


Referenzen nochmals genauer (2/6)



```
class Feiertag {
    int nr; // einfacher Typ
    String name; // unveraenderbare Klasse
    Datum datum; // typische Klasse

    Feiertag(){}

    Feiertag(int nr, String name, Datum datum){
        this.nr = nr;
        this.name = name;
        this.datum = datum;
    }

    String alsText(){
        return this.nr + ": " + this.datum.alsText()
               + " " + this.name;
    }

   // get- und set-Methoden fuer alle Objektvariablen
```

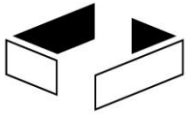
Referenzen nochmals genauer (3/6) - Wertekopien



```
class Analyse {
    EinUndAusgabe io = new EinUndAusgabe();

    void ersteSchritte(){
        Datum datum = new Datum(1, 1, 2020);
        this.io.ausgeben(datum.alsText() + "\n");
        Feiertag f1 = new Feiertag(1, "Neujahr", datum);
        this.io.ausgeben(f1.alsText() + "\n");
        datum.setJahr(2019);
        this.io.ausgeben(f1.alsText() + "\n");
    }
```

```
1.1.2020
1: 1.1.2020 Neujahr
1: 1.1.2019 Neujahr
```

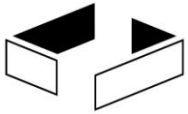
Referenzen nochmals genauer (4/6) - Wertekopien



```
void verzahnteObjekte(){
    Feiertag cn = new Feiertag(1, "Neujahr„
                                , new Datum(25, 1, 2020));
    Feiertag de = new Feiertag(cn.getNr(), cn.getName()
                                , new Datum(1, 1, 2020));
    this.io.ausgeben(de.alsText() + "\n");
    this.io.ausgeben(cn.alsText() + "\n");
    de.setNr(0);
    cn.setName("Chinesisches Neujahr");
    this.io.ausgeben(de.alsText() + "\n");
    this.io.ausgeben(cn.alsText() + "\n");
}
```

```
1: 1.1.2020 Neujahr
1: 25.1.2020 Neujahr
0: 1.1.2020 Neujahr
1: 25.1.2020 Chinesisches Neujahr
```

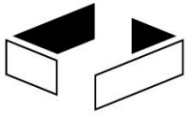
Referenzen nochmals genauer (5/6) - Referenzen



```
void verzahnteObjekte2(){
    Feiertag cn = new Feiertag(1, "Neujahr"
                                , new Datum(25, 1, 2020));
    Feiertag de = new Feiertag(1, cn.getName()
                                , cn.getDatum());
    this.io.ausgeben(cn.alsText() + "\n");
    this.io.ausgeben(de.alsText() + "\n");
    cn.setName("Chinesisches Neujahr");
    de.getDatum().setTag(1);
    this.io.ausgeben(de.alsText() + "\n");
    this.io.ausgeben(cn.alsText() + "\n");
}
```

```
1: 25.1.2020 Neujahr
1: 25.1.2020 Neujahr
1: 1.1.2020 Neujahr
1: 1.1.2020 Chinesisches Neujahr
```

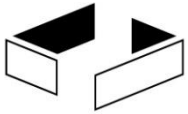
Referenzen nochmals genauer (6/6) - Referenzen



```
void verzahnteObjekte3(){
    Feiertag cn = new Feiertag(1, "Neujahr"
                                , new Datum(25, 1, 2020));
    Feiertag de = cn;
    this.io.ausgeben(cn.alsText() + "\n");
    this.io.ausgeben(de.alsText() + "\n");
    cn.setName("Chinesisches Neujahr");
    de.getDatum().setTag(1);
    this.io.ausgeben(de.alsText() + "\n");
    this.io.ausgeben(cn.alsText() + "\n");
}
```

```
1: 25.1.2020 Neujahr
1: 25.1.2020 Neujahr
1: 1.1.2020 Chinesisches Neujahr
1: 1.1.2020 Chinesisches Neujahr
```

Method Chaining (Aufrufverkettung)



- generell kann auf Objekt jede in der definierenden Klasse angegebene Methode aufgerufen werden
- liefert Methode ein Objekt, kann darauf wieder Methode aufgerufen werden
- lang:

```
Feiertag de;
// …
Datum tmp = de.getDatum();
tmp.setTag(1);
```

- kurz:

```
Feiertag de;
// …
de.getDatum().setTag(1);
```

- Methoden werden von links nach rechts abgearbeitet

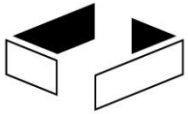
Beispiel

Beispiel

Video

Alternative

Alternativen



- bisher bestehen Programme aus der Nacheinanderausführung von Methodenaufrufen (Sequenz)
- typisch ist die Notwendigkeit der Alternative

  wenn folgendes gilt,

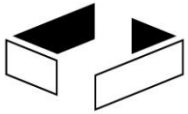
  dann soll folgendes gemacht werden
- oder

  wenn folgendes gilt,

  dann soll folgendes gemacht werden,

  sonst soll folgendes gemacht werden
- wenn = if (dann wird implizit gesagt) sonst = else

Nutzung von if-else

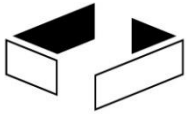


- Anforderung: Basierend auf dem Alter soll eine Persönlichkeitseinstufung erfolgen (z. B. in Klasse Studierend)

```
String persoenlichkeit(){
   String ergebnis;
   if(this.geburtsjahr < 1970){
         ergebnis = "alter Sack";
   } else {
         ergebnis = "Windelhopser";
   }
   return ergebnis;
}
```

- Gibt einige genauso sinnvolle Programmiervarianten (später)

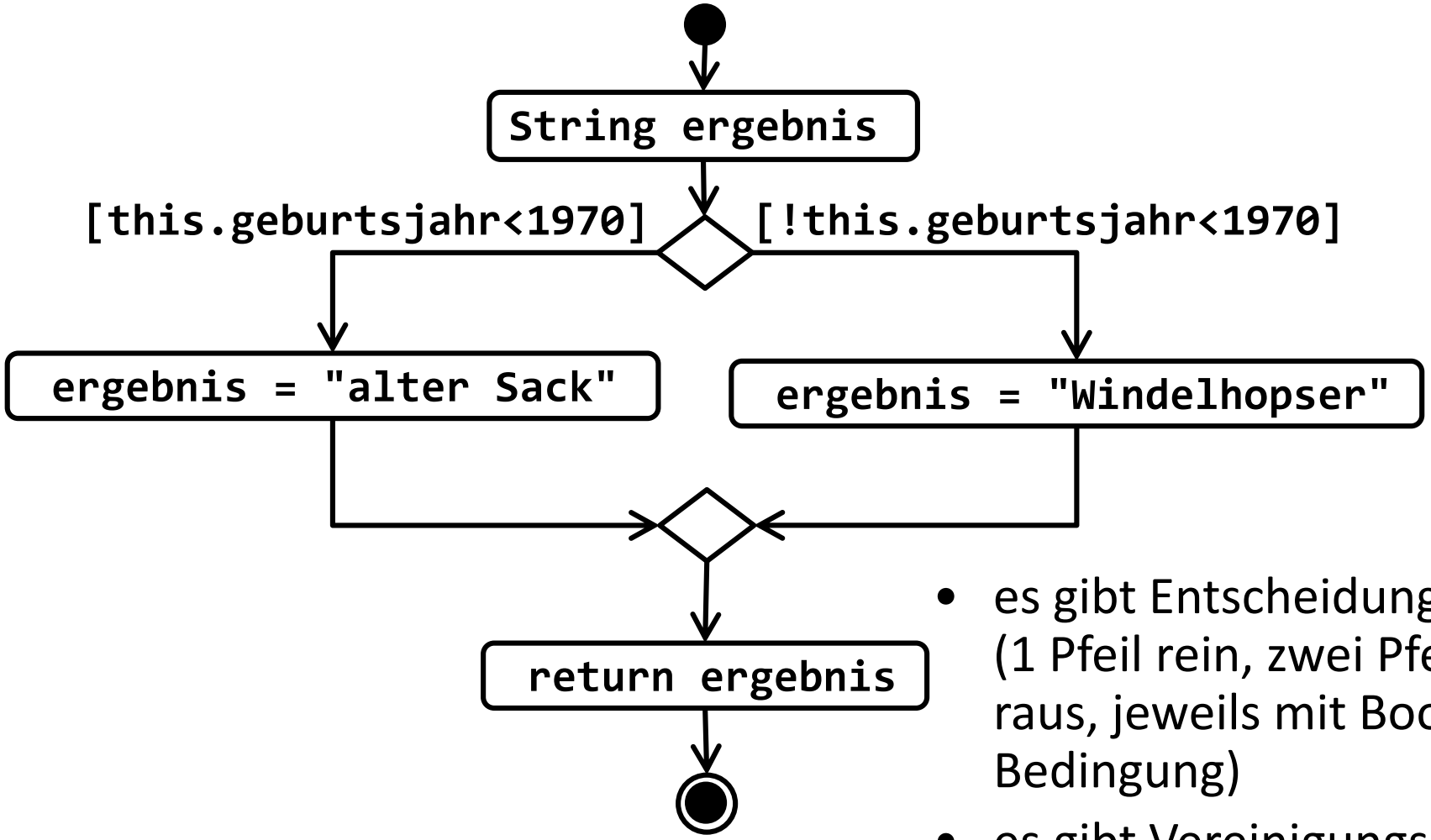
Syntaxanalyse von if -else



```
if( this.geburtsjahr < 1970) {
    ergebnis = "alter Sack";
} else {
    ergebnis = "Windelhopser";
}
```

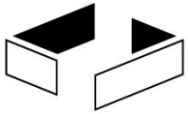
- Boolescher Ausdruck (Bedingung) in runden Klammern, Ausdruck in der Klammer muss nach true oder false ausgewertet werden
- Programmzeilen in geschweiften Klammern; wird ausgeführt, wenn Ausdruck nach true ausgewertet wird
- Programmzeilen in geschweiften Klammern; wird ausgeführt, wenn Ausdruck nach false ausgewertet wird
- generell: Zugriff auf alle sichtbaren Objektvariablen, Methodenparameter und lokale Variablen der umgebenden Blöcke

Visualisierung der Methode mit Aktivitätsdiagramm



- es gibt Entscheidungsraute (1 Pfeil rein, zwei Pfeile raus, jeweils mit Boolescher Bedingung)
- es gibt Vereinigungsraute (n>1 Pfeile hinein, ein Pfeil hinaus)

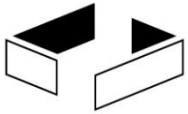
Schachtelung von Programmfragmenten



- Anforderung: Die Persönlichkeitseinstufung soll weiter detailliert werden

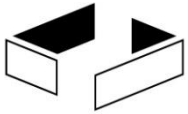
```
String genauerePersoenlichkeit(){
   String ergebnis;
   if(this.geburtsjahr < 1970){
         ergebnis = "alter Sack";
   } else {
         if (this.studiengang.equals("ITI")){
               ergebnis = "Dynamit";
         } else {
               ergebnis = "Windelhopser";
         }
   }
   return ergebnis;
}
```

Analyse s.genauerePersoenlichkeit() – Variante 1



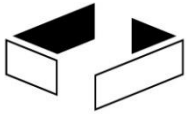
Variable / Anweisung	s		ergebnis
	gebjahr	studiengang	
`s.genauerePersoenlichkeit()`	1967	„ITI“	
`String ergebnis;`	1967	„ITI“	
`if(this.geburtsjahr<1970)`	1967	„ITI“	
`  {ergebnis = "alter Sack";}`	1967	„ITI“	„alter Sack“
`else`			
`  if (this.studiengang .equals("ITI"))`			
`  {ergebnis = "Dynamit";}`			
`  else`			
`  {ergebnis="Windelhopser";}`			
`return ergebnis;`	1967	„ITI“	„alter Sack“

Analyse s.genauerePersoenlichkeit() – Variante 2



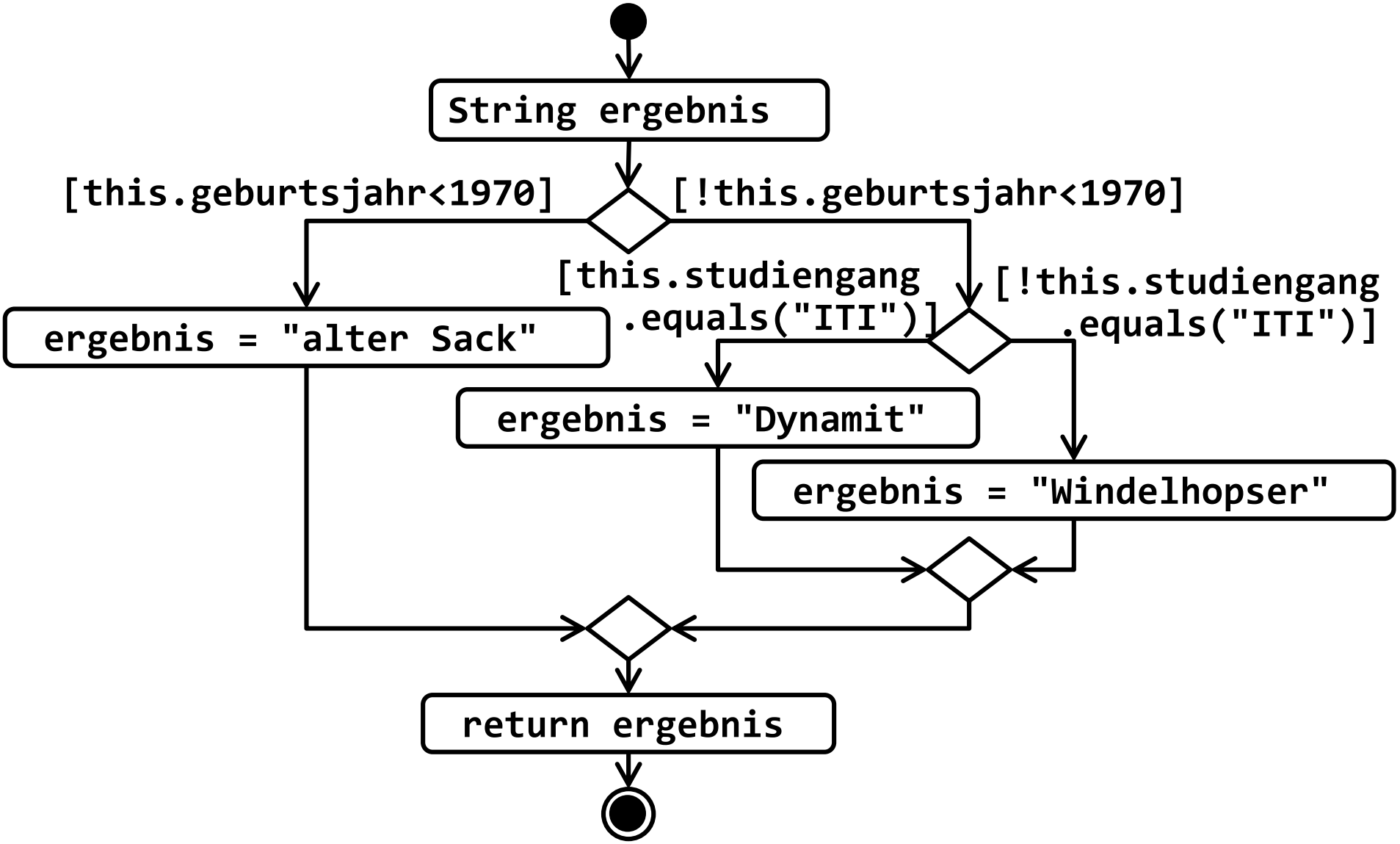
Variable / Anweisung	s		ergebnis
	gebjahr	studiengang	
`s.genauerePersoenlichkeit()`	1988	„ITI“	
`String ergebnis;`	1988	„ITI“	
`if(this.geburtsjahr<1970)`	1988	„ITI“	
`{ergebnis = "alter Sack";}`			
`else`	1988	„ITI“	
`if (this.studiengang .equals("ITI"))`	1988	„ITI“	
`{ergebnis = "Dynamit";}`	1988	„ITI“	„Dynamit“
`else`			
`{ergebnis="Windelhopser";}`			
`return ergebnis;`	1988	„ITI“	„Dynamit“

Analyse s.genauerePersoenlichkeit() – Variante 3

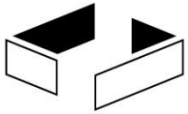


Variable / Anweisung	s		ergebnis
	gebjahr	studiengang	
`s.genauerePersoenlichkeit()`	1988	„IMI“	
`String ergebnis;`	1988	„IMI“	
`if(this.geburtsjahr<1970)`	1988	„IMI“	
`{ergebnis = "alter Sack";}`			
`else`	1988	„IMI“	
`if (this.studiengang .equals("ITI"))`	1988	„IMI“	
`{ergebnis = "Dynamit";}`			
`else`	1988	„IMI“	
`{ergebnis="Windelhopser";}`	1988	„IMI“	„Windelhopser“
`return ergebnis;`	1988	„IMI“	„Windelhopser“

Visualisierung von genauerePersoenlichkeit



Einschub: Vergleichsoperationen



elementare Datentypen, z. B. int x; int y;

- x == y (Prüfung auf Gleichheit),
- x != y (Prüfung auf Ungleichheit), geht auch !(x == y)
- x < y x <= y x > y x >= y

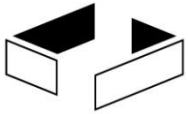
Klassen , z. B. Datum d1 = new Datum(); Datum d2 = new Datum();

- fast immer gewünscht: Prüfung auf inhaltliche Gleichheit
- d1.equals(d2) mit zu schreibenden boolean equals(Datum d)
- ! d1.equals(d2) Ungleichheit, ! für „not“ (nicht)
- bei Strings (de facto) immer falsch; ihnen wird übel

```
String s = ...;
if (s == "Hallo") { ...
```

- gibt == für Objekte (Identität); sehr selten benötigt

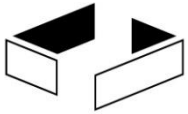
String nie mit == (komplexe Semantik)



- Java: bei inhaltlich gleichen Text darf == true ergeben, muss es aber nicht und tut es nicht immer
- String hat bereits sinnvolle equals-Methode

```
String text1 = "Hu" + "hu";
String text3 = "H" + "uhu";
text1 == text3
    true   (boolean)
String s1 = text1.replace("u","a");
String s2 = text3.replace("u","a");
s1
    "Haha"   (String)
s2
    "Haha"   (String)
s1 == s2
    false   (boolean)
s1.equals(s2)
    true   (boolean)
```

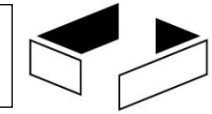
Eine der Implementierungsalternativen



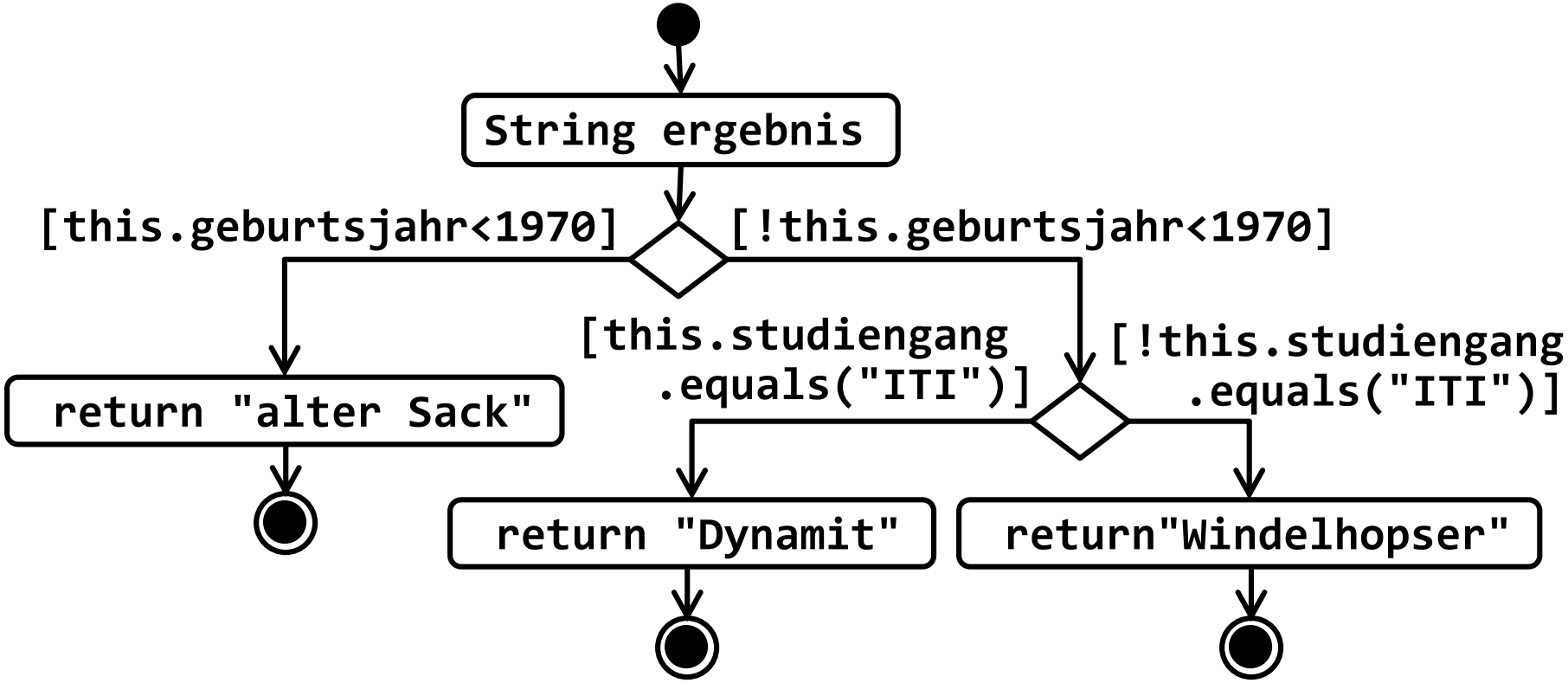
```
String genauerePersoenlichkeit2(){
   if(this.geburtsjahr < 1970){
         return "alter Sack";
   }
   if (this.studiengang.equals("ITI")){
         return "Dynamit";
   }
   return "Windelhopser";
}
```

- Verschachtelung von Anweisungen und komplexe Boolesche Ausrücke machen Programme schwer lesbar

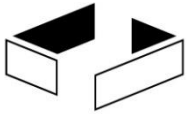
Visualisierung der Alternative



- Hinweis: In Visualisierung nicht sichtbar, ob else genutzt (da mit return Ausführung endet

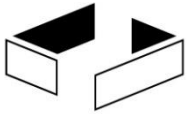


Alternative mit Methodenaufteilung



```
String genauerePersoenlichkeit3(){
   String ergebnis;
   if(this.geburtsjahr < 1970){
         ergebnis = "alter Sack";
   } else {
         ergebnis = this.jungePersoenlichkeit();
   }
   return ergebnis;
}

String jungePersoenlichkeit() {
   String ergebnis;
   if (this.studiengang.equals("ITI")){
         ergebnis = "Dynamit";
   } else {
         ergebnis = "Windelhopser";
   }
   return ergebnis;
}
```

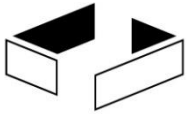
Hinweise zur Methodenaufteilung



- sehr sinnvoll: einfache kurze Methoden mit sinnvollen Methodennamen
- Grundregel: wenn es zu komplex wird, neue Methode auslagern

- Die hier gezeigte (für die Veranstaltung relevante) Methodenaufteilung wird in älteren Programmiersprachen kontrovers diskutiert
- Pro: bessere Lesbarkeit, klarere Struktur (allerdings mehr Methoden)
- Contra: zusätzliche Methodenaufrufe benötigen Zeit (in Java minimal, teilweise durch Compiler wegoptimiert)

Ausgabe des Studiengangsnamens

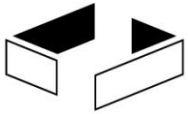


Video

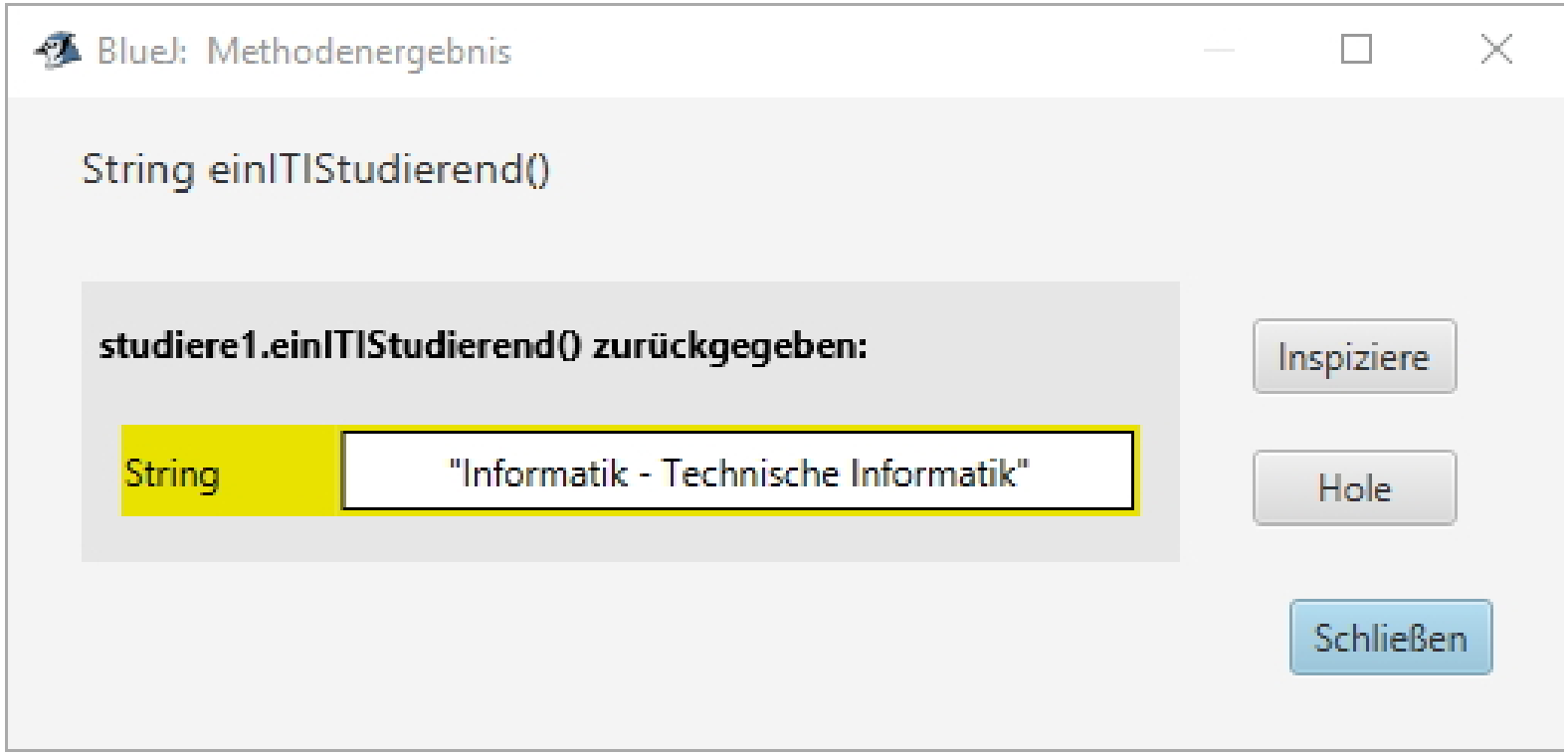
- Die Methode soll einen Langnamen für den Studiengang als Ergebnis liefern, wenn der Name "ITI" ist, soll "Informatik – Technische Informatik" sonst der im Objekt eingetragene Studiengangsname das Ergebnis sein
- Realisierung in der Klasse Studierend

```
String langnameBerechnen(){
    String ergebnis = this.studiengang;
    if(this.studiengang.equals("ITI")){
        ergebnis = "Informatik – Technische Informatik";
    }
    return ergebnis;
}
```

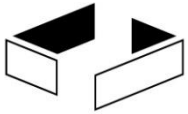
Ausprobieren (in Studierendspielerei) (1/2)



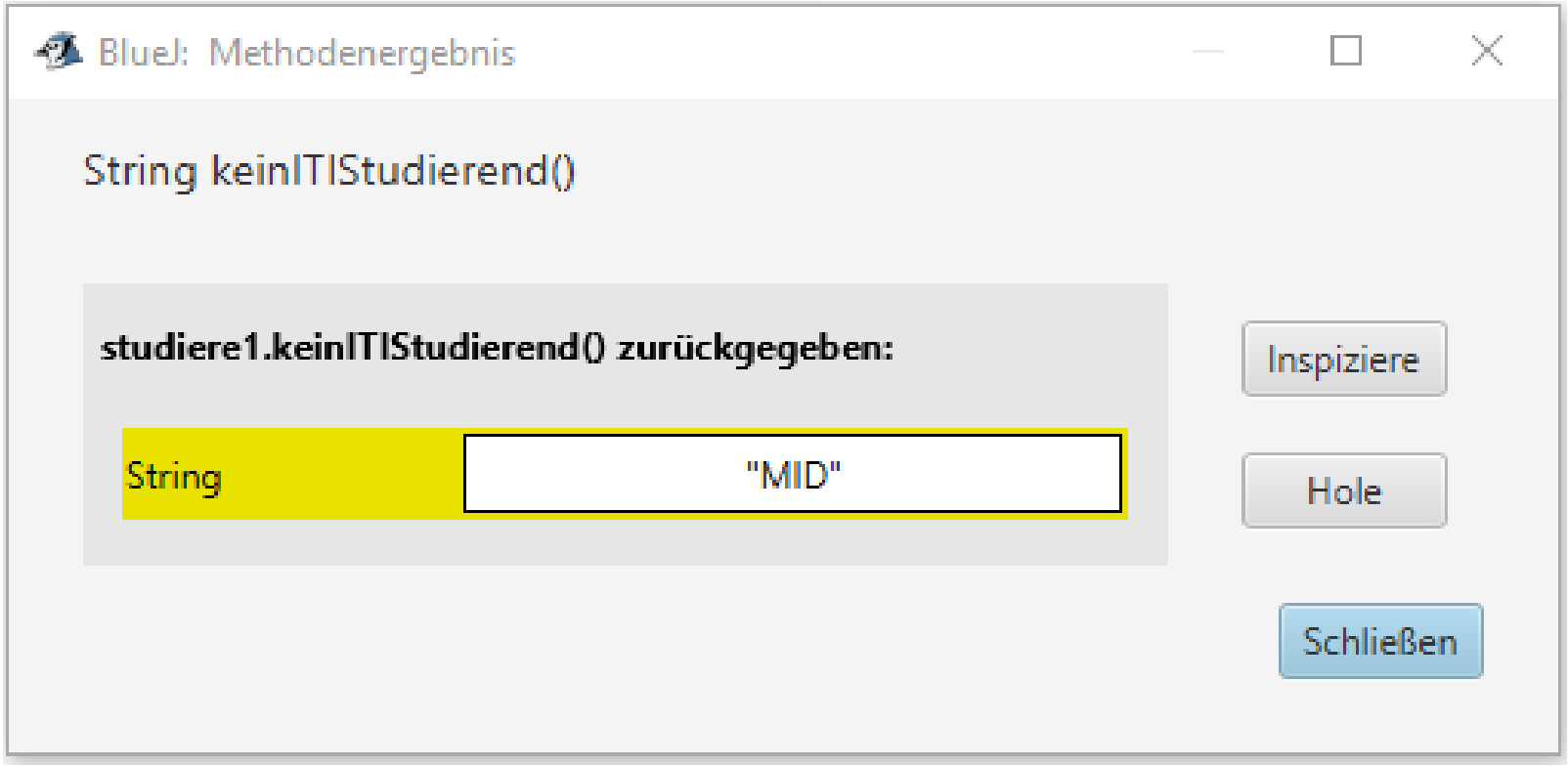
```
String einITIStudierend(){
  Studierend eva= new Studierend("Eva", "Li", 1988, "ITI", 13);
  String ergebnis = eva.langnameBerechnen();
  return ergebnis;
}
```



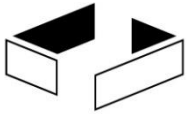
Ausprobieren (in Studierendspielerei) (2/2)



```
String keinITIStudierend(){
  Studierend eva= new Studierend("Udo", "Li", 1987, "MID", 14);
  String ergebnis = eva.langnameBerechnen();
  return ergebnis;
}
```



Boolescher Ausdruck



- benannt nach George Boole (1815-1864), Mathematiker
- wird nach false (falsch) oder true (wahr) ausgewertet
- kann Methode mit Booleschem Ergebnis sein
- kann einfacher Vergleich von Zahlen sein

```
geburtsjahr < 1900
1900 > geburtsjahr
geburtsjahr.equals(matrikelnummer)
```
für Gleichheit

- Operator == prüft Identität, nicht zum inhaltlichen Vergleich nutzbar
- auch `<=` `>=`
- negierbar, Negation mit Ausrufungszeichen

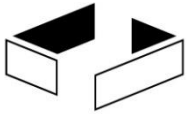
```
! (geburtsjahr < 1900)
```

- Boolesche Variable `boolean jaja = false;`

```
! jaja
```

```
int y = 500;
int z = 500;
y == z
    true   (boolean)
y ! = z
    Error: not a statement
y != z
    false   (boolean)
```

Zusammengesetzter Boolescher Ausdruck (1/2)



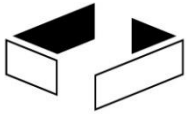
- Boolesche Ausdrücke können mit den Booleschen Operatoren "und" (**&&**) sowie "oder" (||) verknüpft werden
- Boolesche Ausdrücke können in runden Klammern stehen
- Auswertungsreihenfolge: Klammern binden mehr als Negation, diese bindet mehr als Und, das bindet mehr als Oder (einfacher Ansatz: immer Klammern setzen)
- Wahrheitstafeln

&&	true	false
true	true	false
false	false	false

||	true	false
true	true	true
false	true	false

!	
true	false
false	true

Zusammengesetzter Boolescher Ausdruck (2/2)



- Beispiele mit Variablen `int alter = 23` und `int iq = 42`

`alter > 20 && iq > alter` → true && true ergibt true

`alter > 20 || iq < alter` → true || false ergibt true

`alter > 20 || iq < 10 && iq >99` → true || false && false ergibt true || false, ergibt true

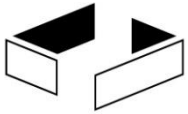
`(alter > 20 || iq < 10) && iq >99` → (true || false) && false ergibt true && false, ergibt false

`!(alter>23) || alter<24` → (!false || true) ergibt (true || true), ergibt true

`!(alter>23 || alter<24)` → !(false || true) ergibt !(true), ergibt false

`! alter>23` gibt Fehler, da int-Wert nicht negiert werden kann

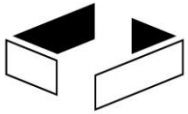
Analyse von Booleschen Ausdrücken in Code Pad



- zur Auswertung kein Semikolon am Zeilenende, Boolescher Ausdruck

```
int alter = 23;
int iq = 42;
alter > iq
    false   (boolean)
alter > 20 || iq < 10 && iq > 99
    true   (boolean)
(alter > 20 || iq < 10) && iq > 99
    false   (boolean)
```

Wahrheitstabellen

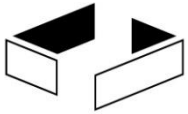


Video

```
boolean a;
boolean b;
boolean c;
```

a	b	c	a || b && c	(a || b) && c
false	false	false	false	false
false	false	true	false	false
false	true	false	false	false
false	true	true	true	true
true	false	false	true	false
true	false	true	true	true
true	true	false	true	false
true	true	true	true	true

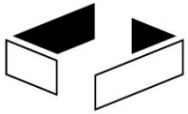
Rechenregeln für Boolesche Ausdrücke



- nutzt Aussagenlogik
- ≡ heißt "logisch äquivalent", bedeutet dass beide Ausdrücke immer den gleichen Wahrheitswert liefern

- a && (b || c) ≡ (a && b) || (a && c)
- a || (b && c) ≡ (a || b) && (a || c)
- ! (a && b) ≡ !a || !b a && b ≡ b && a
- ! (a || b) ≡ !a && !b a || b ≡ b || a
- a && !a ≡ false a && a ≡ a
- a || !a ≡ true a || a ≡ a
- a && true ≡ a a && false ≡ false
- a || false ≡ a a || true ≡ true

Einsatz von Rechenregeln

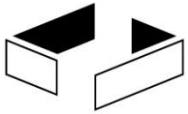


- Umformung Boolescher Ausdrücke mit dem Ziel der Vereinfachung (hier int-Variablen)

```
if (iq<99 && bmi>25 || iq>98 && bmi>25 || iq<99 && bmi<=25
                    || iq>98 && bmi<=25){
    <Teilprogramm>
}
```

```
  iq<99 && bmi>25 || iq>98 && bmi>25 || iq<99 && bmi<=25
                  || iq>98 && bmi<=25
≡    iq<99 && bmi>25 || iq<99 && bmi<=25    (nur vertauschen)
  || iq>98 && bmi>25 || iq>98 && bmi<=25
≡    iq<99 && (bmi>25 || bmi<=25)
  || iq>98 && (bmi>25 || bmi<=25)
≡ iq<99 && (true)   ||   iq>98 && (true)
≡ iq<99 || iq>98  ≡  true   (if ist überflüssig)
```

Ausdrücke mit ganzen Zahlen



Video

int-Variablen a, b, und c, es gibt folgende Operatoren

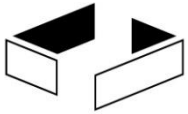
- a + b (Addition)
- a – b (Subtraktion)
- a * b (Multiplikation)
- a / b (ganzzahlige Division 7 / 4 == 1)
- a % b (ganzzahliger Rest der Division (modulo) 7 % 4 == 3)
- Boolesche Ausdrücke der Form a + b < a * b möglich
- Ausdrücke können Variablen auf der linken Seite zugewiesen werden

```
int oha = (7 / 4) * 4 + (7 % 4);
```

- Erinnerung: unglückliche Nutzung des Gleichheitszeichens

```
oha = oha + 1;
```

Analyse von ganzzahligen Ausdrücken in Code Pad

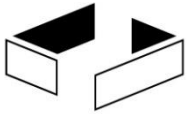


```
7+4*3
19   (int)
(7+4)*3
33   (int)
7/4
1   (int)
7%4
3   (int)
7%(1-7/4)
Error: / by zero
0%2
0   (int)
7+4>5+6
false   (boolean)
```

- es gilt: erst Klammern, dann *, / und %, dann + und –
- mathematische Operatoren binden stärker (werden zuerst ausgewertet) als Boolesche Operatoren
- Division durch Null nicht erlaubt, man erhält in weiterem Fenster:

```
java.lang.ArithmeticException: / by zero
```

Ausdrücke mit Fließkomma-Zahlen



double-Variablen a, b, und c, es gibt folgende Operatoren

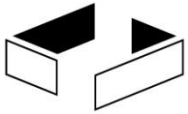
- a + b (Addition)
- a – b (Subtraktion)
- a * b (Multiplikation)
- a / b (Division 7.0 / 4.0 == 1.75)
- a % b (Rest nach ganzzahliger Division 7.0 % 3.4 == 0.2)
- Boolesche Ausdrücke der Form a + b < a * b möglich
- Ausdrücke können Variablen auf der linken Seite zugewiesen werden

```
double oha = (7.0 / 4.0) * 4.0;
```

- man beachte unglückliche Nutzung des Gleichheitszeichens

```
oha = oha + 1;
```

Analyse von Fließkommazahl-Ausdrücken



```
7.3+5.4*3.8
27.82  (double)
(7.3+5.4)*3.8
48.26  (double)
7.3>5.3+2.4 || 5.5+4.4<3.3+6.6
false  (boolean)
7.3/0
Infinity  (double)
-7.3/0
-Infinity  (double)
0.0/0.0
NaN  (double)
7.0%3.4
0.20000000000000018  (double)
```

es gilt, erst Klammern, dann * und /, dann + und –

mathematische Operatoren binden stärker (werden zuerst ausgewertet) als Boolesche Operatoren

Division durch Null führt zu neuen double-Werten (später mehr):

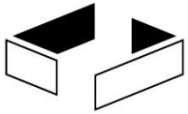
-Infinity

Infinity

NaN

- Rechengenauigkeit beachten

Programmiervarianten (1/2)



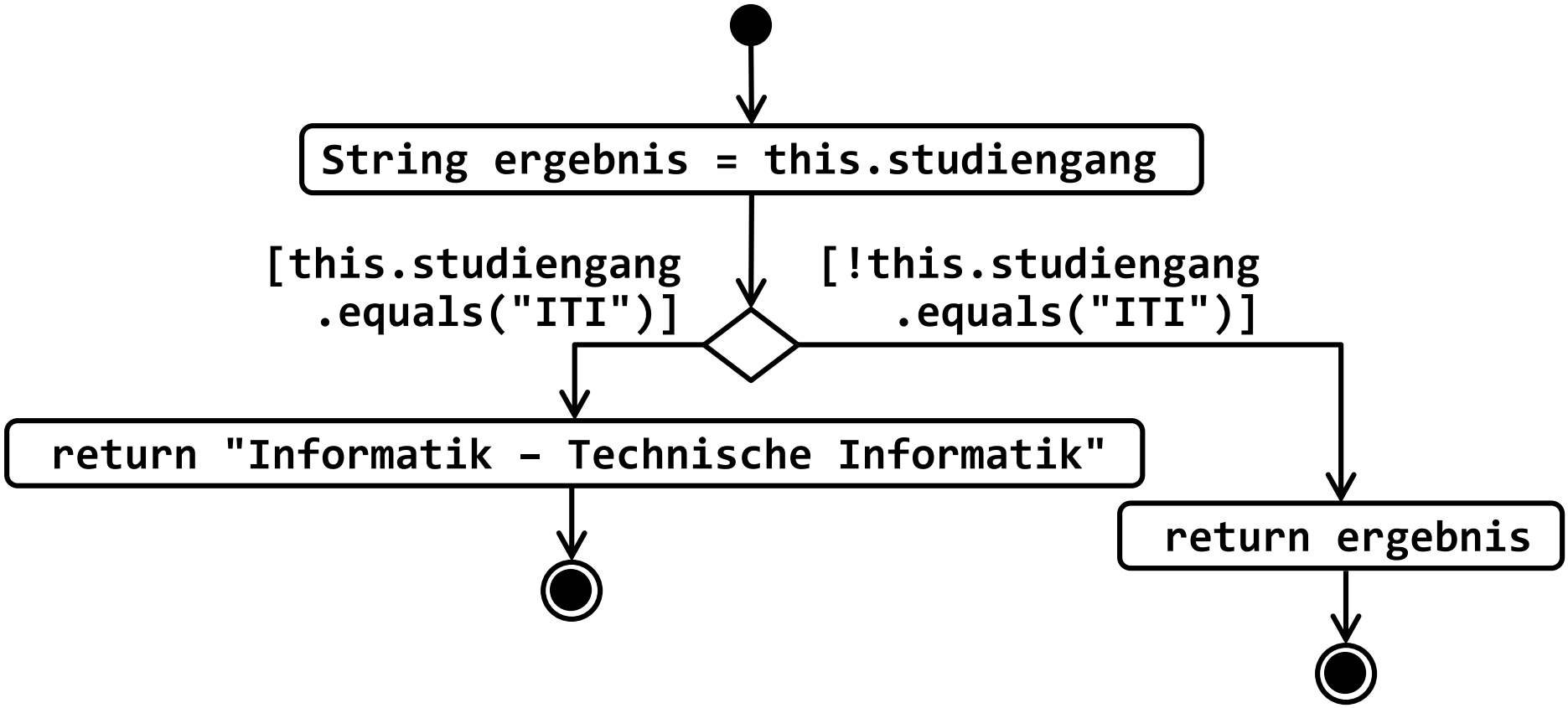
- Nutzung einer Ergebnisvariablen (Name beliebig):

```
String langnameBerechnen(){
    String ergebnis = this.studiengang;
    if(this.studiengang.equals("ITI")){
        ergebnis = "Informatik – Technische Informatik";
    }
    return ergebnis;
}
```

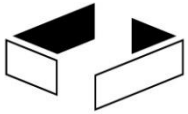
- Variante: direkter Rücksprung (evtl. kürzere Berechnung aber schwerer zu Lesen, da es mehr als ein Ende gibt)

```
String langnameBerechnen2(){
    String ergebnis = this.studiengang;
    if(this.studiengang.equals("ITI")){
        return "Informatik – Technische Informatik";
    }
    return ergebnis;
}
```

Visualisierung der Variante



Programmiervarianten (2/2)



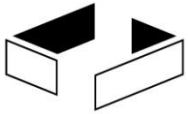
- kürzer; zumindest für Anfänger ist es aber sinnvoll, eine ergebnis-Variable einzuführen und zurückzugeben

```
String langnameBerechnen3(){
    if(this.studiengang.equals("ITI")){
        return "Informatik – Technische Informatik";
    }
    return this.studiengang;
}
```

- Nutzung eines aus anderen Programmiersprachen übernommenen Operators

```
String langnameBerechnen4(){
   return this.studiengang.equals("ITI")
          ? "Informatik – Technische Informatik"
          : this.studiengang;
}
```

Operator ? :



- Operator hat folgende Syntax
- <Boolescher Ausdruck> ? <Ausdruck1> : <Ausdruck2>
- Wenn Boolescher Ausdruck nach wahr ausgewertet wird, dann ist das Ergebnis der Wert des Ausdrucks1 sonst des Ausdrucks2
- Ausdruck1 und Ausdruck2 müssen selben Typ haben

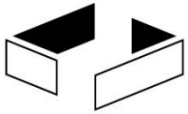
```
int mo = name.equals("Li") ? 42 : 23;
```

- äquivalent zu

```
int mo;
if(name.equals("Li")){
    mo = 42;
} else {
    mo = 23;
}
```

- Wunsch: verzichten Sie auf diesen coolen Operator

Grundregel für Alternativen

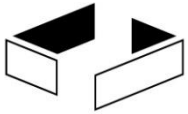


Müssen zur Ausführung mehrere Bedingungen berücksichtigt werden

- müssen die Booleschen Ausdrücke zunächst einzeln formuliert werden
- muss überlegt werden, ob ein Ausdruck wichtiger als ein anderer ist; falls es den gibt: if(<Bed.>) {… return…}
- müssen die zusammengesetzten Booleschen Ausdrücke formuliert und in eine sinnvolle
  - Sequenz von if-else-Befehlen, oder
  - Verschachtelung von if-else-Befehlen

  gebracht werden

Kurzschreibweise (1/3)

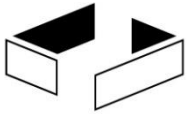


- steht nur ein Befehl bei if oder else, können geschweifte Klammern weggelassen werden

```
int mo;
if(name.equals("Li"))
    mo = 42;
else
    mo = 23;
```

- das ist sehr schlechter Programmierstil, da fehleranfällig und nicht leicht erweiterbar (nächste Folien)
- z. B.: [LMS14] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, D. Svoboda, Java Coding Guidelines – 75 Recommendations for Reliable and Secure Programs, Addison Wesley, Upper Saddle River (USA), 2014

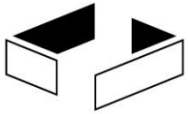
Kurzschreibweise (2/3) [LMS14]



```
public String gruppe(String nutzend, String passwort){
        String ergebnis = "keine Rechte";
        if (! this.istLoginKorrekt(nutzend, passwort))
           if (this.alsGastErlaubt(nutzend, passwort))
             ergebnis = "Gast";
        else
             ergebnis = "Admin";
        return ergebnis;
    }
```

- Annahme: wenn Login korrekt, dann ist Ergebnis „Admin“

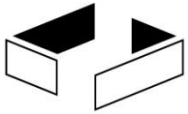
Kurzschreibweise (3/3)



- wahrscheinlich gewünschtes, korrektes Programm

```
public String gruppeOK(String nutzend, String passwort){
    String ergebnis = "keine Rechte";
    if (! this.istLoginKorrekt(nutzend, passwort)) {
        if (this.alsGastErlaubt(nutzend, passwort)) {
            ergebnis = "Gast";
        }
    } else {
            ergebnis = "Admin";
    }
    return ergebnis;
 }
```

Sichtbarkeiten - Einstieg



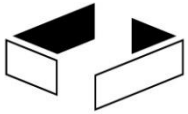
Video

- beim Auslagern von Methoden gab es folgendes Ergebnis:

```
String genauerePersoenlichkeit3(){ ... }
String jungePersoenlichkeit() { ...}
```

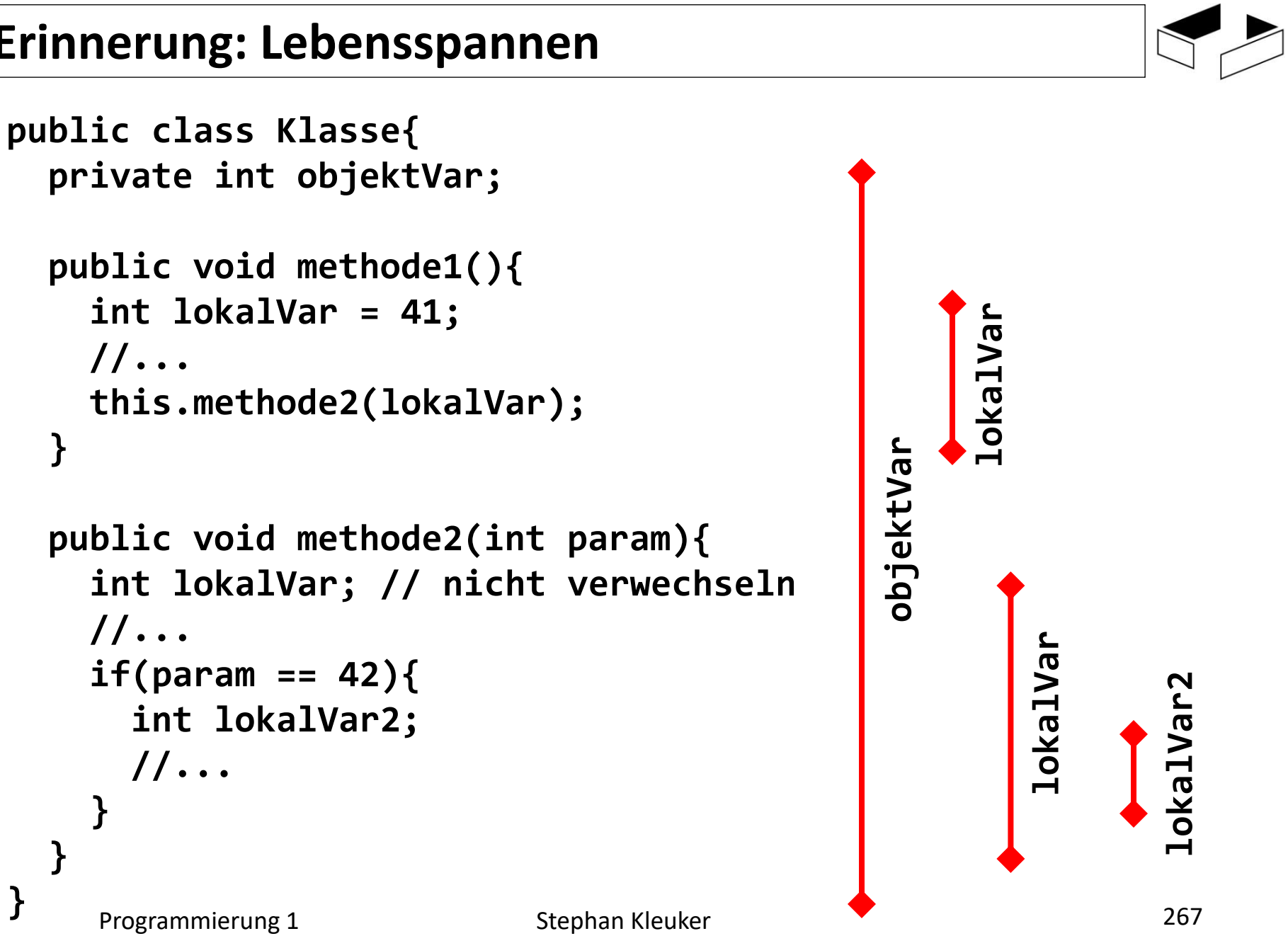
- Problem: Objekt-Nutzende können auch Methode jungePersoenlichkeit() aufrufen, obwohl dies vielleicht nicht gewünscht ist
- Lösung: Klassen, Objektvariablen und Methoden können mit Sichtbarkeiten markiert werden, hier zunächst
  - public: alle können darauf zugreifen
  - private: nur innerhalb des Objektes kann darauf zugegriffen werden
- Grundregel der Objektorientierung: Information-Hiding; nur über Methoden kann auf Objektvariablen zugegriffen werden

Sichtbarkeiten in der Klasse Studierend



```
public class Studierend{
    private String vorname ="Eva";
    private String nachname ="Mustermann";
    private int geburtsjahr = 1990;
    private String studiengang = "IMI";
    private int matrikelnummer = 232323;

    public Studierend(String vorname, …
    public Studierend(){ …
    public void vertauscheNamen(){…
    public String genauerePersoenlichkeit3(){…
    private String jungePersoenlichkeit() {…
    public boolean equals(Studierend other){…
    public String getVorname() {…
    public String getNachname() {…
    public void setVorname(String vorname) {…
    public void setNachname(String nachname) {…
…
```

Erinnerung: Lebensspannen

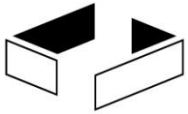


Beispiel

Video

equals

Was ist gleich (1/3)



- Erinnerung: Java nutzt elementare Datentypen und Klassen
- Vergleich von elementaren Datentypen mit ==

```
EinUndAusgabe io = new EinUndAusgabe();
int x = 42;
if (x == 42) {
Error: reached end of file while parsing
if (x == 42) {
  io.ausgeben("x ist 42 \n");
}
```

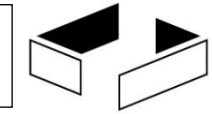
BlueJ: Terminal

Options

```
x ist 42
```

- auch in Code Pad ausprobierbar (Eingabe von Befehlen über mehrere Zeilen mit Shift+Return)

Was ist gleich (2/3)



- Objekte werden mit == (dasselbe) oder equals (das Gleiche) verglichen

```
EinUndAusgabe io = new EinUndAusgabe();
String s1 = new String("Hallo");
String s2 = "Hallo";
String s3 = s2;
if(s1 == s2){
  io.ausgeben("identisch");
}
if(s1.equals(s2)){
  io.ausgeben("gleich\n");
}
if(s3 == s2){
  io.ausgeben("identisch\n");
}
```

BlueJ: Terminal

Options

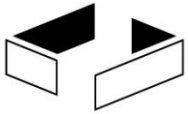
```
gleich
identisch
```

s1	String
s2	String
s3	String

String „Hallo“

String „Hallo“

Was ist gleich (3/3)



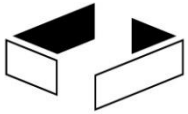
- Objekte: inhaltlicher Vergleich immer mit Methode equals
- Klassen aus der Java-Klassenbibliothek haben immer eine sinnvoll programmierte equals-Methode
- bei selbstgeschriebenen Klassen muss die Methode equals selbst geschrieben werden, um sinnvolle Realisierung zu bekommen

```
class MeineKlasse { …

  public boolean equals (MeineKlasse obj) { …
```

- später:

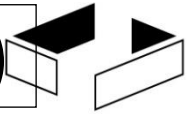
```
  public boolean equals (Object obj) { …
```

Inhaltliche Objektgleichheit mit equals



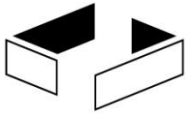
- Vergleich von Strings auf gleichen Inhalt mit equals
- genauer: jede vorhandene Java-Klasse hat equals-Methode
- Folgerung: wir müssen auch equals-Methode schreiben
- [Hinweis: ganz sauber etwas später nach Vererbung]
- wir bestimmen wann etwas gleich sein soll:
  - typisch: alle Werte der Objektvariablen sollen übereinstimmen
  - Variante: Objekt hat eindeutigen Identifikationswert, z. B. Matrikelnummer; restliche Objektwerte müssen nicht übereinstimmen (Tippfehler, Heirat)

Methode equals für Klasse Studierend (erster Versuch)



```
public boolean equals(Studierend other){
  if (other == null){
    return false;
  }
  if (this.vorname.equals(other.getVorname())
      && this.nachname.equals(other.getNachname())
      && this.studiengang.equals(other.getStudiengang())
      && other.getGeburtsjahr() == this.geburtsjahr
      && other.getMatrikelnummer() == this.matrikelnummer){
    return true;
  }
  return false;
}
```

Erinnerung: null-Referenz



- Methoden können nur auf echten Objekten, nicht der null-Referenz ausgeführt werden
- Methodenaufrufe auf der null-Referenz führen zu gefürchteten NullPointerExceptions
- defensive Programmierung: zuerst auf null testen

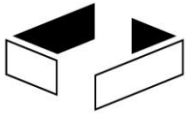
```
if (other == null){
    return false;
}
```

- sonst:

```
Studierend s1 = null;
s1.getMatrikelnummer();
    Exception: java.lang.NullPointerException (null)
```

```
running
java.lang.NullPointerException: Cannot invoke "Studierend.getMatrikelnummer()" because "s1" is null
```

Analyse des equals für Studierend



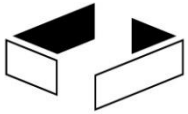
- Parameter sauber auf null überprüft, aber bei

  ```
  this.nachname.equals(other.getNachname())
  ```

  könnte this.nachname eine null-Referenz sein
- Antwort: Nein, da beide Konstruktoren mit Default-Werten garantieren, dass Objekte gesetzt werden (Eva Mustermann)
- Kommentar: Halt! Mit Methode setNachname kann nachträglich null-Referenz übergeben werden
- Also entweder set-Methoden absichern oder equals umschreiben
- Ansatz: prüfe für jeden Parameter ob er null-Referenz ist und mache dann inhaltliche Prüfung

  gebe bei erster gefundener Ungleichheit false und nur ganz am Ende nach allen Überprüfungen true zurück

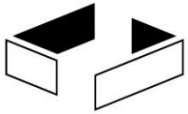
Methode equals ohne NullPointerException (1/2)



```
boolean equals(Studierend other){
    if (other == null){
      return false;
    }
    if (this.nachname == null) {
      if (other.getNachname() != null) {
        return false;
      }
    } else {
      if (!this.nachname.equals(other.getNachname())) {
        return false;
      }
    }
    if (this.studiengang == null) {
      if (other.getStudiengang() != null) {
        return false;
      }
    } else {
```

Ansatz: Prüfe Objektvariablen nacheinander, bei Ungleichheit „false“

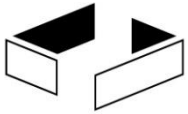
Methode equals ohne NullPointerException (2/2)



```
  } else {  // wiederholt Zeile der letzten Folie
    if (!this.studiengang.equals(other.getStudiengang())) {
      return false;
    }
  }
  if (this.vorname == null) {
    if (other.getVorname() != null) {
      return false;
    }
  } else {
    if (!this.vorname.equals(other.getVorname())) {
      return false;
    }
  }
  return other.getGeburtsjahr() == this.geburtsjahr
     && other.getMatrikelnummer() == this.matrikelnummer;
}
```

abschließender Ansatz: bei elementaren Typen nur Gleichheit prüfen

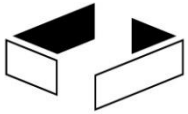
Nutzung von equals (in Klasse Studierendspielerei)



```
public void gleichheitsanalyse(){
  EinUndAusgabe io = new EinUndAusgabe();
  Studierend eva1 = new Studierend("Eva", null, 1988
                                  , "ITI", 13);
  Studierend eva2 = eva1;
  Studierend udo1 = new Studierend("Udo", "Li", 1987, null, 14);
  Studierend udo2 = new Studierend("Udo", "Li", 1987, null, 14);
  io.ausgeben("eva1 == eva2   : " + (eva1 == eva2) + "\n");
  io.ausgeben("eva1 eq eva2   : " + (eva1.equals(eva2)) + "\n");
  io.ausgeben("udo1 == udo2   : " + (udo1 == udo2) + "\n");
  io.ausgeben("udo1 eq udo2   : " + (udo1.equals(udo2)) + "\n");
}
```

```
eva1 == eva2    : true
eva1 eq eva2    : true
udo1 == udo2    : false
udo1 eq udo2    : true
```

Gleichheit in Objektspeicherdiagrammen



elementare Typen

- int x = 42;
- int y = 42;
- x == y ergibt true

x	int	42
y	int	42

Prüfung auf inhaltliche Gleichheit

Objekte

- Studierend eva1 = new Studierend(...);
- Studierend eva2 = eva1
- Studierend eva3 = new Studierend(...);
- eva1 == eva2 ergibt true
- eva2 == eva3 ergibt immer false

eva1	Studierend
eva2	Studierend

gdw. gilt ==

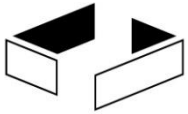
Studierend	vor...	
	...	

eva3	Studier end

Studierend	vor...	
	...	

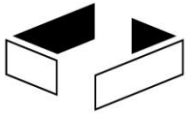
- mit equals() legt man eigene Regeln für Gleichheit fest

Kommentare - Einstieg



- Kommentare sollen das Lesen von Programmen für Andere und sich selbst (!!!) einfacher machen
- zwei Kommentar-Arten
- Kurzkommentar nach // , Kommentar endet mit Zeile
- Langkommentar, beginnt mit /* und endet mit */
- Anfänger sollten intensiver kommentieren
- typische Kommentare
- Kurzbeschreibung: wozu gibt es Objektvariable
- Methode: wozu ist sie da, welche Parameter mit welchem Sinn gibt es, wie sieht das berechnete Ergebnis aus
- meist wenig/keine Kommentare in Methoden, da durch Namen, ihre Kürze und sprechende Variablen selbsterklärend
- systematische Kommentierung (→ JavaDoc) später

Kommentare: Beispiel

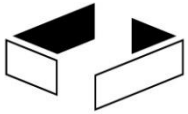


```
/** Klasse zur Verwaltung des aktuell leitenden Person. */
public class Leitend {
  private String name;                    // Name Leitend
  private int level = 23;                 // erreichte Stufe
  /* Konstruktor fuer neues Leitend-Objekt
   * Parameter name: neuer Name der leitenden Person
   */
  public Leitend(String name){
    this.name = name;
  }
  /* Methode zur Pruefung, ob Leitend Andrea heisst
     Ergebnis: heisst Leitend Andrea  */
  public boolean issesAndrea(){
    if("Andrea".equals(this.name)){ // geht auch so
      return true;
    }
    return false;
  }
}
```

Video

Strings - toString

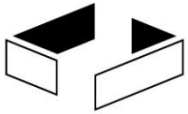
Erinnerung: Strings als Ausdrücke



- neben den Objektmethoden gibt es Möglichkeit, Strings mit "+" zu verknüpfen (konkatenieren)
- weiterhin können Strings mit beliebigen Objekten mit "+" verknüpft werden

```
"Hallo" + "dele"
  "Hallodele"   (String)
"Zeich ma Zahl " + 42
  "Zeich ma Zahl 42"   (String)
Studierend eva = new Studierend();
eva + " studiert im " + (3 - 2) + ".ten Semester"
  "Studierend@202dc779 studiert im 1.ten Semester"   (String)
```

Methode toString()



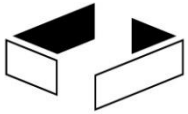
- neue Klassen werden unbefriedigend in Strings umgesetzt
- Lösung: Klasse erhält Methode public String toString()

```
public String toString(){
    return this.vorname+" "+this.nachname+" ("
        + this.matrikelnummer+"):"+this.studiengang
}
```

```
Studierend eva = new Studierend();
"Da: " + eva
"Da: Eva Mustermann (232323):IMI"   (String)
```

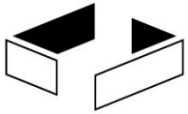
- Methode wird implizit bei String-Bearbeitung angestoßen
- man kann toString als normale Methode nutzen **`eva.toString()`**

Erinnerung mit Erweiterung (1/2)



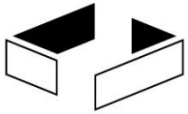
```
public class Datum { // Standardaufbau, schlecht formatiert
    private int tag;
    private int monat;
    private int jahr;

    public Datum (int tag, int monat, int jahr){
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
    }

    public Datum(){
    }

    public int getTag() { return this.tag; }

    public void setTag(int tag) { this.tag = tag; }

    public int getMonat() { return this.monat; }

    public void setMonat(int monat) { this.monat = monat; }
```

Erinnerung mit Erweiterung (2/2)



```
    public int getJahr() {
        return this.jahr;
    }
    public void setJahr(int jahr) {
        this.jahr = jahr;
    }

    public boolean equals(Datum other){
        return (other != null) && this.tag == other.getTag()
            && this.jahr == other.getJahr()
            && this.monat == other.getMonat();
    }

    public String toString(){
        return this.tag + "." + this.monat + "." + this.jahr;
    }
}
```

Besondere Zeichen (1/2)

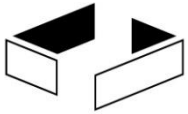


- Ausgabe wird zur Zeit als unendliche Zeile ausgegeben
- Zeilenumbruch mit Zeichen "\n"
- Backslash leitet Sonderzeichen ein
- \u leitet Unicode-Zeichen ein (genauer UTF-16)

Text	Ausgabe
\n	Zeilenumbruch
\t	Tabulator
\\	\
\"	"
\u00c4	Ä
\u00e4	ä

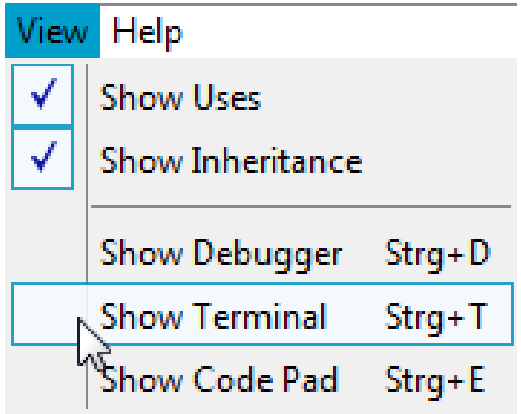
Text	Ausgabe
\u00d6	Ö
\u00f6	ö
\u00dc	Ü
\u00fc	ü
\u00df	ß

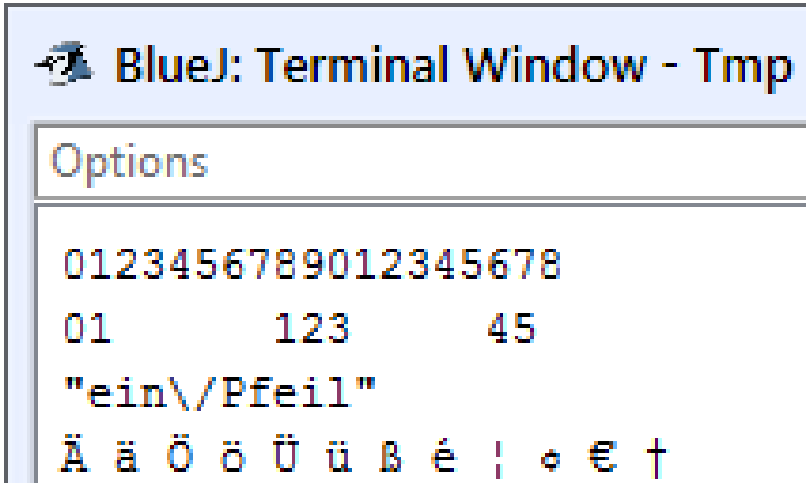
Besondere Zeichen (2/2)



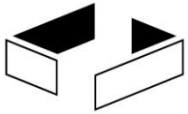
- Ein- und Ausgabe erfolgt in BlueJ in "Terminal"
- Beispielmethode:

```
public void sonderzeichen(){
  EinUndAusgabe io = new EinUndAusgabe();
  io.ausgeben("0123456789012345678\n");
  io.ausgeben("01\t123\t45\n");
  io.ausgeben("\"ein\\/Pfeil\"\n");
  io.ausgeben("\u00c4 \u00e4 \u00d6 \u00f6 \u00dc \u00fc "
      + " \u00df \u00E9 \u00A6 \u00A2 \u20AC \u2020 ");
}
```





Vision: alle Zeichen sichtbar mit Unicode



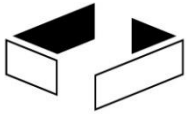
- Unicode [ˈjuːnɪkoʊd] ist ein internationaler Standard, in dem langfristig für jedes sinntragende Schriftzeichen oder Textelement aller bekannten Schriftkulturen und Zeichensysteme ein digitaler Code festgelegt wird. Ziel ist es, die Verwendung unterschiedlicher und inkompatibler Kodierungen in verschiedenen Ländern oder Kulturkreisen zu beseitigen. Unicode wird ständig um Zeichen weiterer Schriftsysteme ergänzt. (wikipedia.de)
- recht mächtig und häufig genutzt: ISO 8859-1 (Latin1, z. B. auch griechisch und hebräisch)
- <?xml version="1.0" encoding="UTF-8"?>
- <?xml version="1.0" encoding="ISO-8859-1"?>
- http://www.unicode.org

Beispiel

Video

ArrayList - Einstieg

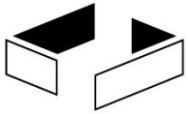
Sammlungen



- oftmals werden mehrere Werte gleichen Typs benötigt
- Praktikum mit allen Studierend-Objekten
- man spricht dann von Sammlungen (Collections)
- generell gibt es mehrere sinnvolle Varianten von Collections; hier wird zunächst nur eine betrachtet
- ArrayList<Klasse> liste = new ArrayList<Klasse>();
- kürzer: ArrayList<Klasse> liste = new ArrayList<>();
- liste kann beliebig viele Objekte der Klasse Klasse aufnehmen

- Problem: elementare Datentypen (z. B. int) sind keine Klassen; ArrayList<int> in Java nicht erlaubt ☹

Java-Lösung für Collections mit elementaren Typen

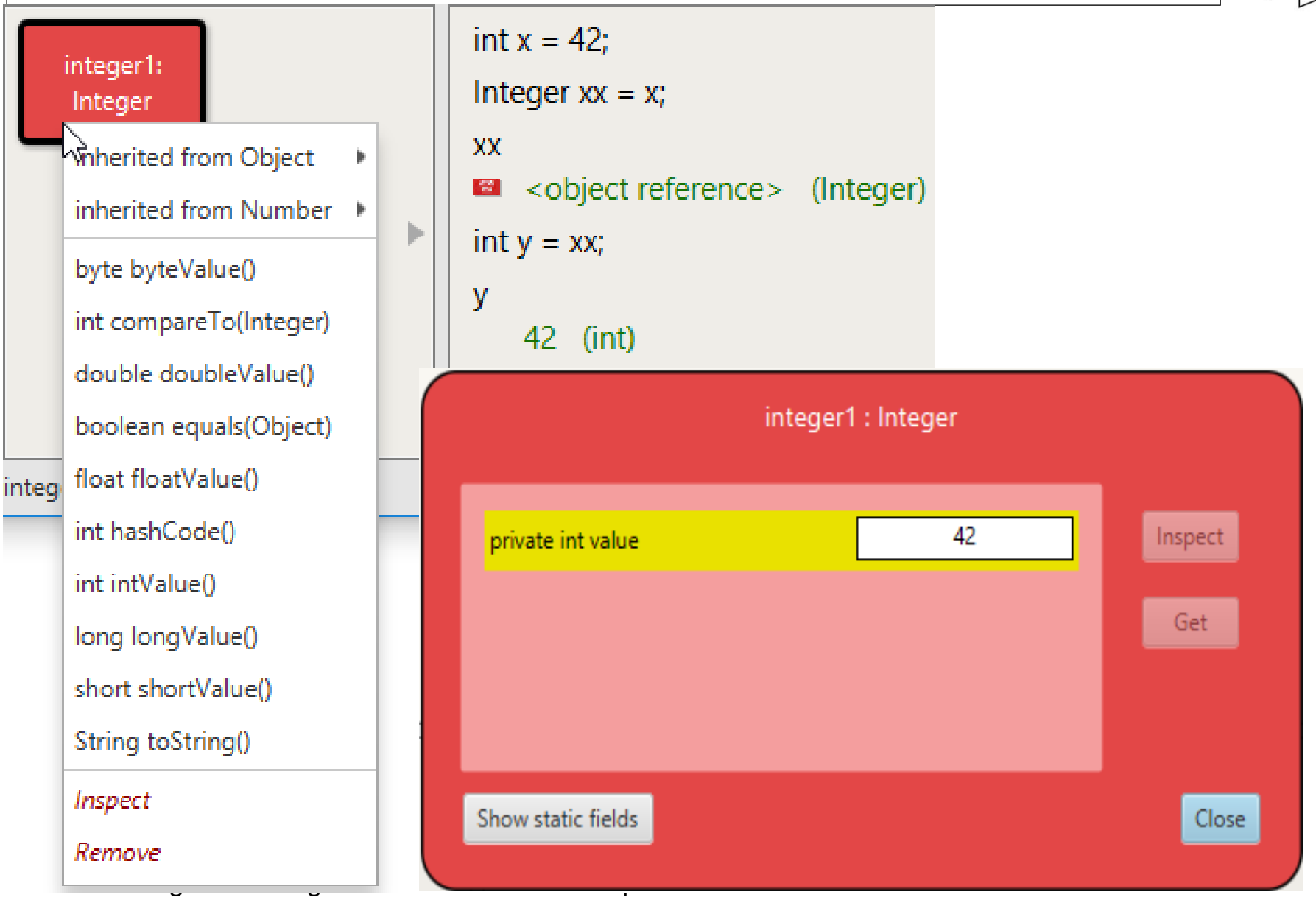


- Java-Lösung: Wrapper Typen, z. B.

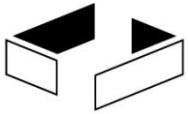
```
public class Integer {
  private int value;
  public Integer(int value){
    this.value = value;
  }
  …
}
```

- z. B. Integer x = new Integer(42);
- geht, ist aufwändig zu schreiben, deshalb automatische Umwandlung in beide Richtungen (boxing, unboxing)
- Integer y = 42; // automatische Wandlung in Objekt
- int z = y; // automatische Rückwandlung von Objekt in Wert
- Wrapper: Byte, Short, Integer, Long, Float, Double, Boolean, Character

Wrappen- Boxing und Unboxing

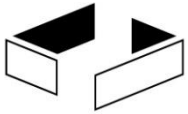


Einige Methoden der Klasse ArrayList<Klas>



Metho-denname	Parame-tertyp	Ergebnis-typ	Beschreibung
add	Klas		fügt übergebenes Element hinten an
add	int, Klas		fügt übergebenes Element an angegebener Position ein
get	int	Klas	gibt Element an gefragter Position zurück
remove	int	Klas	entfernt Element an gegebener Position (ist auch Ergebnis)
set	int, Klas	Klas	ersetzt Objekt an angegebener Position, gibt altes Objekt zurück
size		int	gibt Anzahl der Elemente

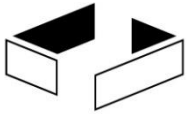
Spielerei mit ArrayList



```
import java.util.ArrayList;

public class ListenSpielerei {
   public void spielen(){
    ArrayList<Integer> zahlen = new ArrayList<Integer>();
    EinUndAusgabe io = new EinUndAusgabe();
    io.ausgeben("1: " + zahlen + "\n");
    zahlen.add(42);
    zahlen.add(41);
    io.ausgeben("2: " + zahlen + "\n");
    zahlen.add(1,23);
    io.ausgeben("3: " + zahlen + "\n");
    io.ausgeben("4: " + zahlen.get(2) +"\n");
    zahlen.set(0, 67);
    io.ausgeben("5: " + zahlen + "\n");
    zahlen.remove(0);
    io.ausgeben("6: " + zahlen + "\n");
    io.ausgeben("7: " + zahlen.size() + "\n");
  }
}
```

```
1: []
2: [42, 41]
3: [42, 23, 41]
4: 41
5: [67, 23, 41]
6: [23, 41]
7: 2
```

ArrayList genauer



```
ArrayList<Integer> zahlen = new ArrayList<Integer>();
zahlen.add(42);
zahlen.add(41);
```

Position:	0	1
Inhalt:	42	41

```
zahlen.add(1,23);
```

Position:	0	1	2
Inhalt:	42	23	41

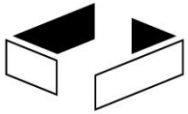
```
zahlen.set(0, 67);
```

Position:	0	1	2
Inhalt:	67	23	41

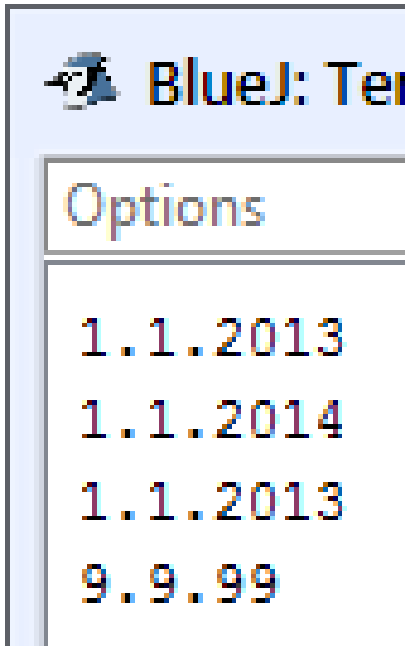
```
zahlen.remove(0);
```

Position:	0	1
Inhalt:	23	41

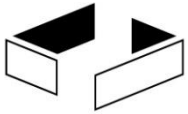
ArrayList mit Referenzen (wie üblich) (1/2)



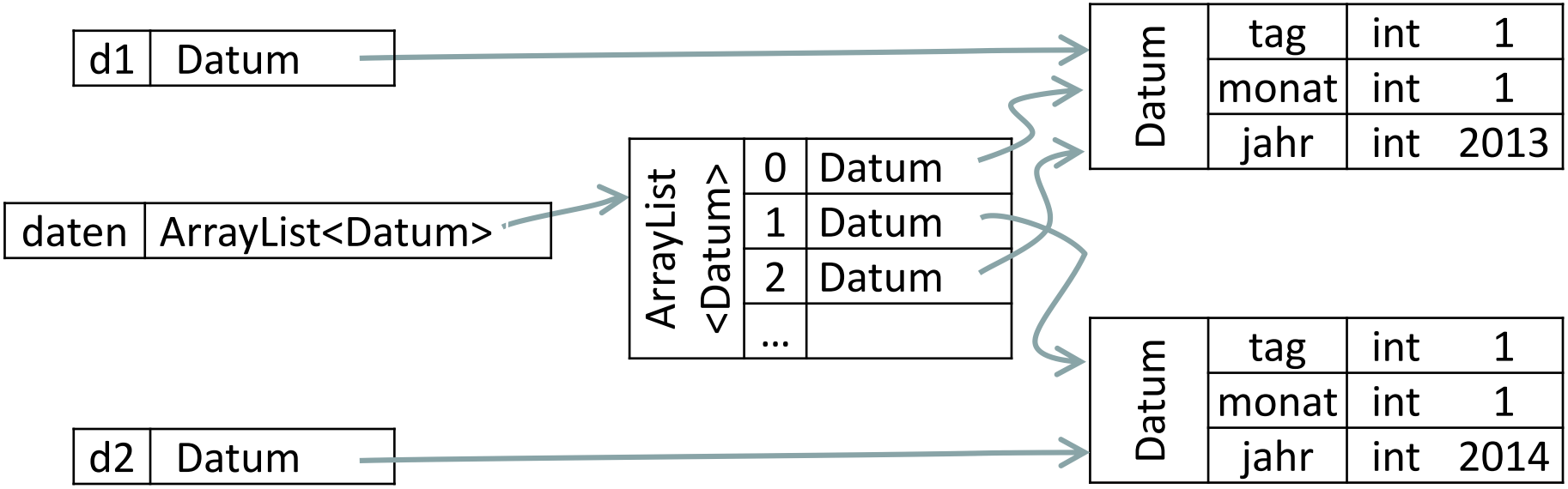
```
import java.util.ArrayList;

public class DatumsListe{
  public void analyse(){
    EinUndAusgabe io = new EinUndAusgabe();
    ArrayList<Datum> daten = new ArrayList<Datum>();
    Datum d1 = new Datum(1, 1, 2013);
    Datum d2 = new Datum(1, 1, 2014);
    daten.add(d1);
    daten.add(d2);
    daten.add(d1);
    Datum tmp = daten.get(0);
    io.ausgeben(tmp.toString() + "\n");
    Datum tmp2 = daten.get(1);
    io.ausgeben(tmp2 + "\n");
    io.ausgeben(daten.get(2) + "\n");
    d1.setTag(9);
    tmp.setMonat(9);
    daten.get(0).setJahr(99);
    io.ausgeben(daten.get(2) + "\n");
  }
}
```



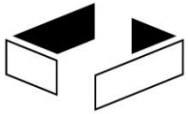
ArrayList mit Referenzen (wie üblich) (2/2)



```
ArrayList<Datum> daten = new ArrayList<Datum>();
Datum d1 = new Datum(1, 1, 2013);
Datum d2 = new Datum(1, 1, 2014);
daten.add(d1);
daten.add(d2);
daten.add(d1);
```



Einschub: nur Referenzen sind wichtig (1/2)



- grausame Methode in der Klasse Datum

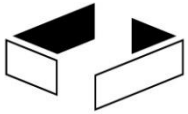
```
public void startDerZeit(Datum d){
    EinUndAusgabe io = new EinUndAusgabe();
    io.ausgeben("vorher: " + d + "\n");
    d = new Datum(0, 0, 0);
    io.ausgeben("nachher: "+ d +"\n");
}
```

- Ausprobieren in DatumsListe

```
public void gehtNicht(){
    EinUndAusgabe io = new EinUndAusgabe();
    Datum d1 = new Datum(1, 1, 2013);
    d1.startDerZeit(d1);
    io.ausgeben(d1 + "\n");
}
```

```
vorher: 1.1.2013
nachher: 0.0.0
1.1.2013
```

Einschub: nur Referenzen sind wichtig (2/2)



```
Datum d1 = new Datum(1, 1, 2013);
d1.startDerZeit(d1);
```

d1	Datum

Datum			
	tag	int	1
	monat	int	1
	jahr	int	2013

```
    startDerZeit(Datum d){
```

d1	Datum
d	Datum

Datum			
	tag	int	1
	monat	int	1
	jahr	int	2013

```
        d = new Datum(0, 0, 0);
```

Datum			
	tag	int	0
	monat	int	0
	jahr	int	0

d1	Datum
d	Datum

Datum			
	tag	int	1
	monat	int	1
	jahr	int	2013

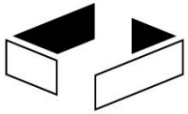
```
    }
```

Datum			
	tag	int	0
	monat	int	0
	jahr	int	0

d1	Datum

Datum			
	tag	int	1
	monat	int	1
	jahr	int	2013

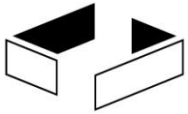
ArrayList mit zwei remove-Methoden (1/2)



- remove(int): lösche an Position
- remove(Objekt): lösche erstes Objektvorkommen
- Problem: nur wenn Objekt Typ Integer hat, Lösung:

```
import java.util.ArrayList;

public class IntegerListe{
     void problem(){
        EinUndAusgabe io = new EinUndAusgabe();
        ArrayList<Integer> arr = new ArrayList<Integer>();
        arr.add(2);
        arr.add(3);
        arr.add(4);
        Integer big = 42;
        arr.add(big);
```

ArrayList mit zwei remove-Methoden (2/2)



```
    io.ausgeben(arr + "\n");
    arr.remove(2);
    io.ausgeben(arr + "\n");
    arr.remove(new Integer(2));
    io.ausgeben(arr + "\n");
    arr.remove(new Integer(43));
    io.ausgeben(arr + "\n");
    arr.remove(big);
    io.ausgeben(arr + "\n");
    arr.remove(3);
  }
}
```

BlueJ: Terminal Window - ArrayL

Options

```
[2, 3, 4, 42]
[2, 3, 42]
[3, 42]
[3, 42]
[3]
```

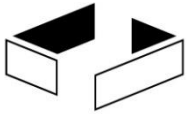
```
java.lang.IndexOutOfBoundsException:
Index: 3, Size: 1 (in java.util.ArrayList)
```

Beispiel

Video

Schleife

Idee - Schleife



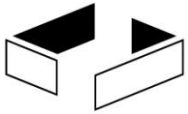
- oftmals sollen gleiche Programmzeilen wiederholt ausgeführt werden
- wiederholtes Hinschreiben keine gute Idee
- weiterhin muss festgelegt werden können, wie oft Wiederholung stattfindet

- Neues Sprachkonstrukt: Schleife

  Solange folgende Bedingung gilt,

  wiederhole folgende Schritte

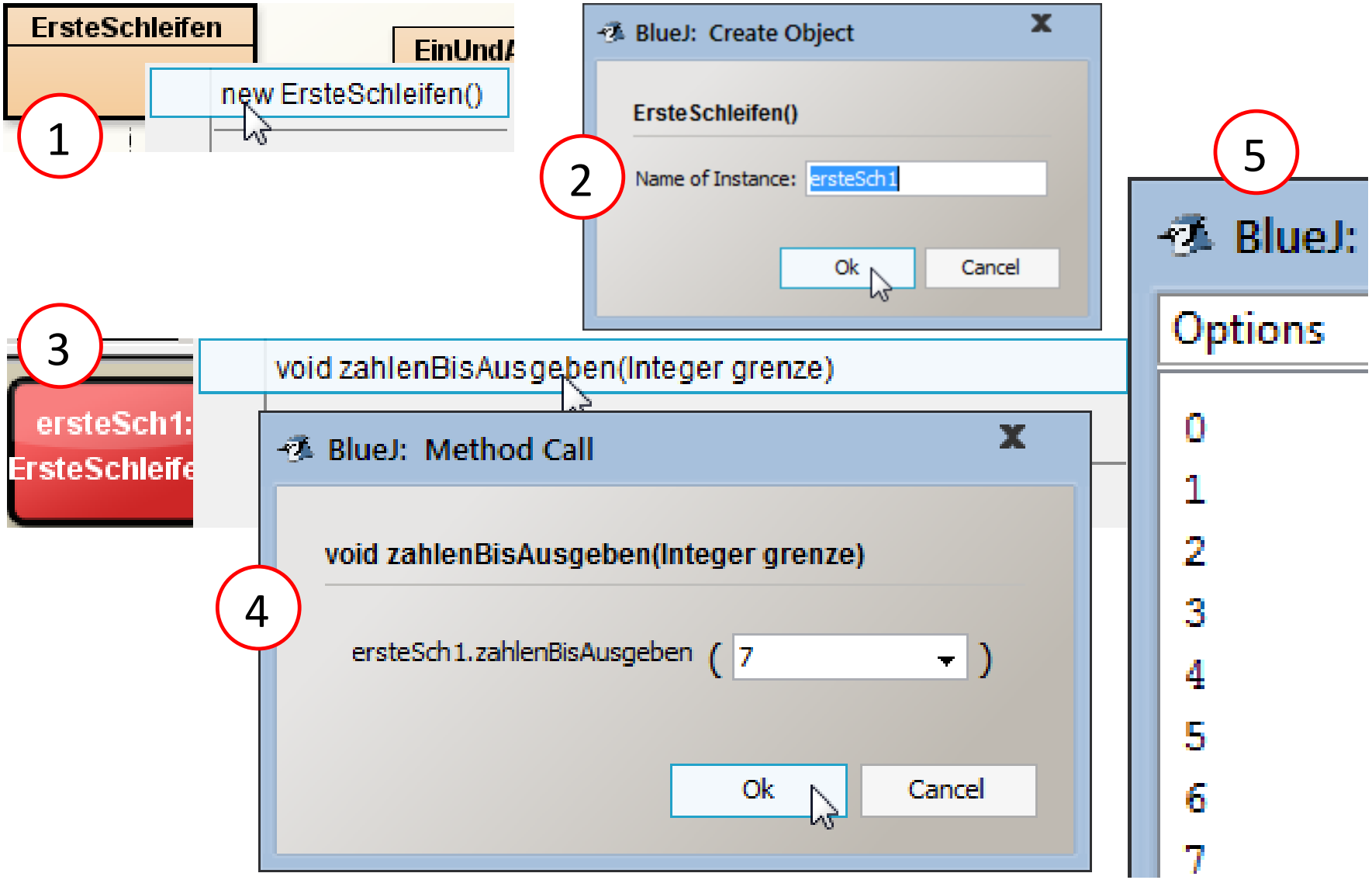
Erste Schleifenklasse



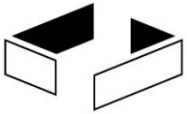
- Aufgabe: Alle Zahlen von 0 bis einschließlich einem Grenzwert ausgeben

```
import java.util.ArrayList;
public class ErsteSchleifen{

    public void zahlenBisAusgeben(int grenze){
        EinUndAusgabe io = new EinUndAusgabe();
        int zaehler = 0;
        while(zaehler <= grenze){
            io.ausgeben(zaehler + "\n");
            zaehler = zaehler + 1;
        }
    }
```

Ausführung

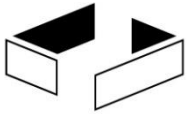

ErsteSchleifen
EinUndA
new ErsteSchleifen()
1
BlueJ: Create Object
ErsteSchleifen()
Name of Instance: ersteSch1
Ok
Cancel
2
5
BlueJ:
Options
0
1
2
3
4
5
6
7
3
ersteSch1:
ErsteSchleife
void zahlenBisAusgeben(Integer grenze)
BlueJ: Method Call
void zahlenBisAusgeben(Integer grenze)
ersteSch1.zahlenBisAusgeben (7)
Ok
Cancel
4

Analyse



Variable Anweisung	grenze	io	zaehler
`zahlenBisAusgeben(1)`	1		
`EinUndAusgabe io =` `new EinUndAusgabe()`	1	<obj>	
`int zaehler = 0`	1	<obj>	0
`while(zaehler <= grenze)`	1	<obj>	0
`io.ausgeben(zaehler+"\n")`	1	<obj>	0
`zaehler = zaehler + 1;`	1	<obj>	1
`while(zaehler <= grenze)`	1	<obj>	1
`io.ausgeben(zaehler+"\n")`	1	<obj>	1
`zaehler = zaehler + 1;`	1	<obj>	2
`while(zaehler <= grenze)`	1	<obj>	2
`}`			

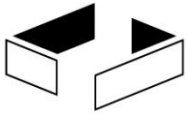
Syntaxanalyse von while



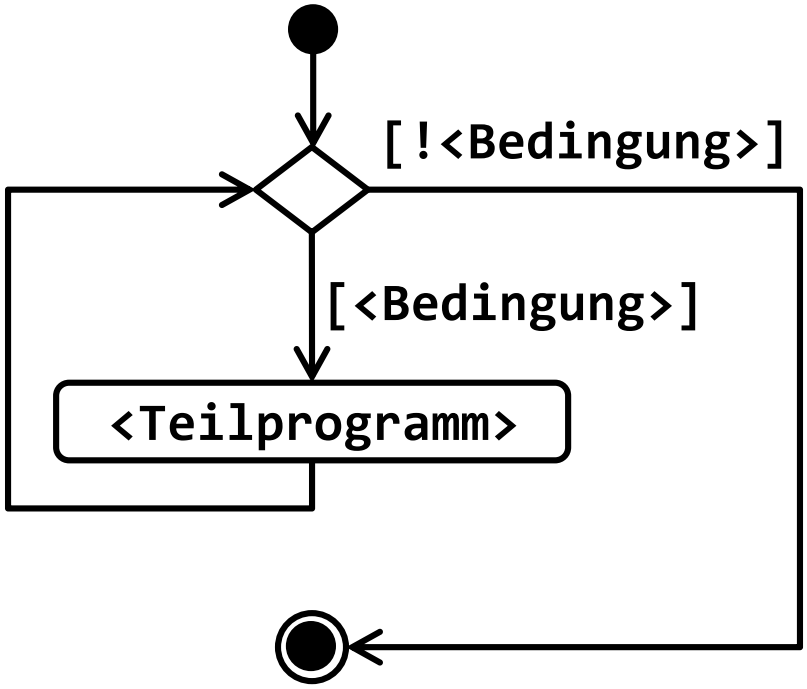
```
while(zaehler <= grenze){
    io.ausgeben(zaehler + "\n");
      zaehler = zaehler + 1;
}
```

- Boolescher Ausdruck (Bedingung) in runden Klammern, Ausdruck in der Klammer muss nach true oder false ausgewertet werden
- Block(Programmfragment) in geschweiften Klammern; wird ausgeführt, wenn Ausdruck nach true ausgewertet wird
- nach Programmfragment-ausführung wird Boolescher Ausdruck erneut ausgewertet

Visualisierung im Aktivitätsdiagramm - generell

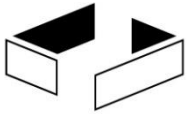


```
while ( <Bedingung> ) {
    <Teilprogramm>
}
```



- Raute am Anfang, in der sich zwei Abläufe (Start und Wiederholung) vereinigen (auch als zwei Rauten darstellbar)
- generell soll nur ein Pfeil in eine Aktion laufen

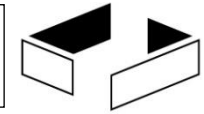
Variante: zurück statt ausgeben



- Aufgabe: Alle Zahlen von 0 bis einschließlich einem Grenzwert zurückgeben

```
public ArrayList<Integer> zahlenBisZurueckgeben(int grenze){
    ArrayList<Integer> ergebnis = new ArrayList<Integer>();
    int zaehler = 0;
    while(zaehler <= grenze){
        ergebnis.add(zaehler);
        zaehler = zaehler + 1;
    }
    return ergebnis;
}
```

Ausführung (z. B. in Code Pad)



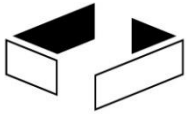
```
EinUndAusgabe io = new EinUndAusgabe();
ErsteSchleifen es = new ErsteSchleifen();
io.ausgeben(es.zahlenBisZurueckgeben(7));
```

BlueJ: Terminal Window - ErsteSchleifen

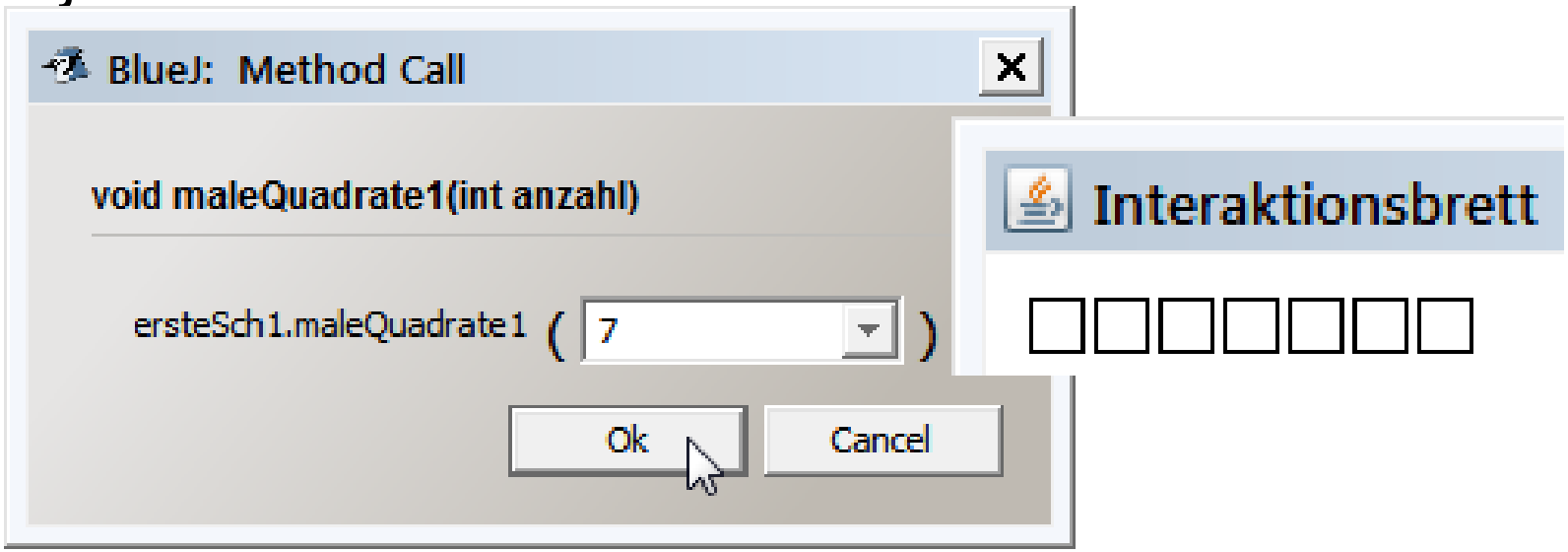
Options

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

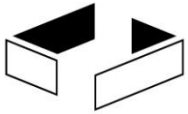
Male anzahl Quadrate (1/2)



```
public void maleQuadrate1(int anzahl){
    Interaktionsbrett ib = new Interaktionsbrett();
    int zaehler = 0;
    while (zaehler < anzahl) {
        ib.neuesRechteck(10 + zaehler * 15, 10, 12, 12);
        zaehler = zaehler + 1;
    }
}
```

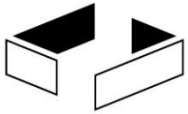


Male anzahl Quadrate (2/2)



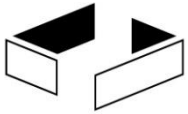
```
// etwas schlechtere Variante
public void maleQuadrate2(int anzahl){
    Interaktionsbrett ib = new Interaktionsbrett();
    int zaehler = 0;
    int xPosition = 10;
    while (zaehler < anzahl) {
        ib.neuesRechteck(xPosition, 10, 12, 12);
        zaehler = zaehler + 1;
        xPosition = xPosition + 15;
    }
}
```

Eingabedialog und Schleife



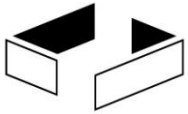
- Studierend-Objekt soll durch Tastatureingabe im Dialog erstellt werden
- Erstellung soll in neuer Klasse Studierendenverwaltung liegen
- Bei der Eingabe sollen Randbedingungen überprüft werden, wie: der eingegebene String ist nicht leer
- Informeller Ansatz: Solange ein ungewünschter Wert eingegeben wird, soll die Eingabe wiederholt werden

Korrekte Eingabe des Vornamens



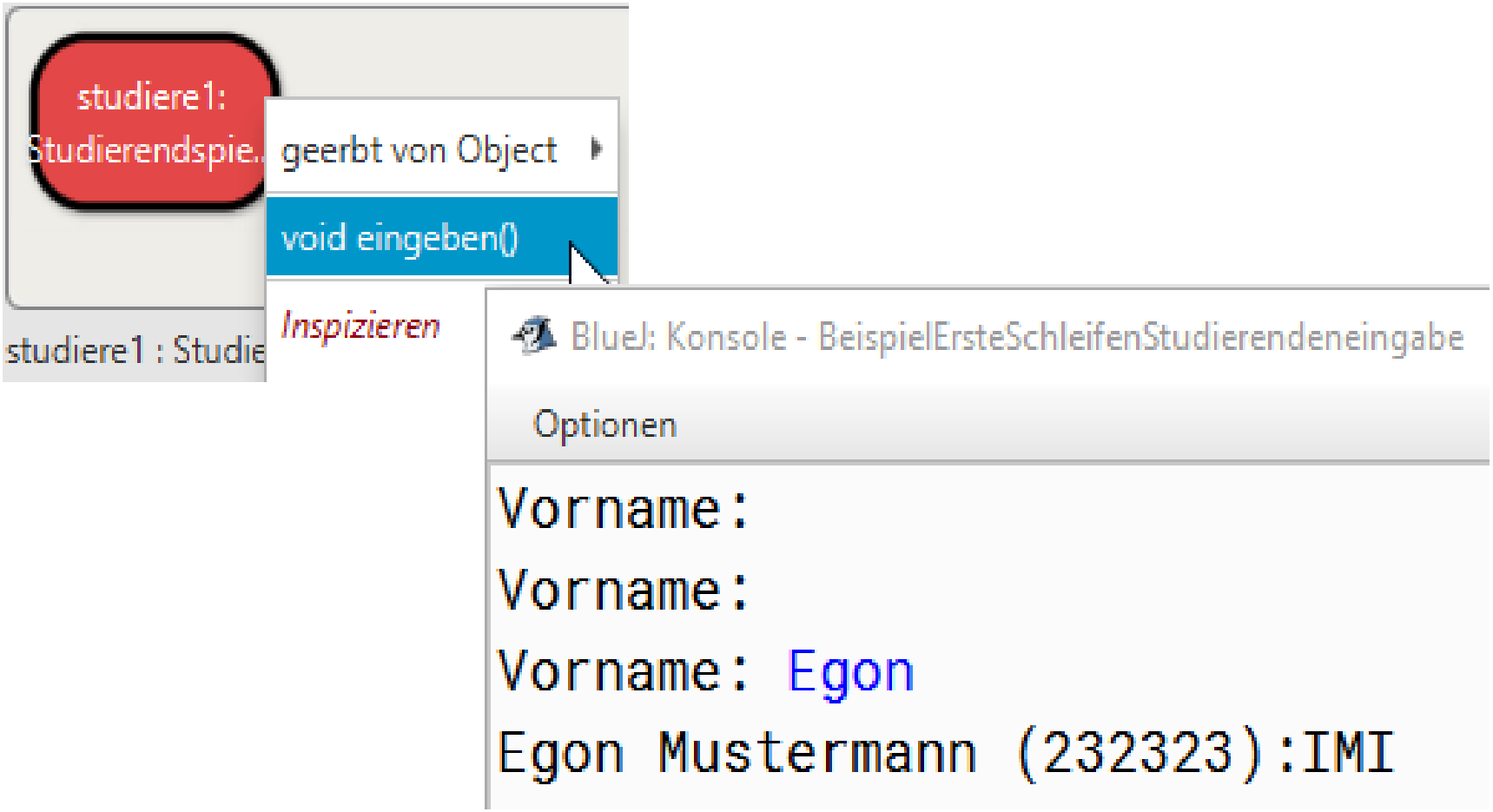
```
public class Studierendenverwaltung{

  public Studierend studierendEingeben(){
    Studierend ergebnis = new Studierend();
    EinUndAusgabe io = new EinUndAusgabe();
    String tmpVorname = "";
    while(tmpVorname.equals("")){
      io.ausgeben("Vorname: ");
      tmpVorname = io.leseString();
    }
    ergebnis.setVorname(tmpVorname);
    return ergebnis;
  }
}
```

Ausprobieren in Studierendspielerei (1/2)

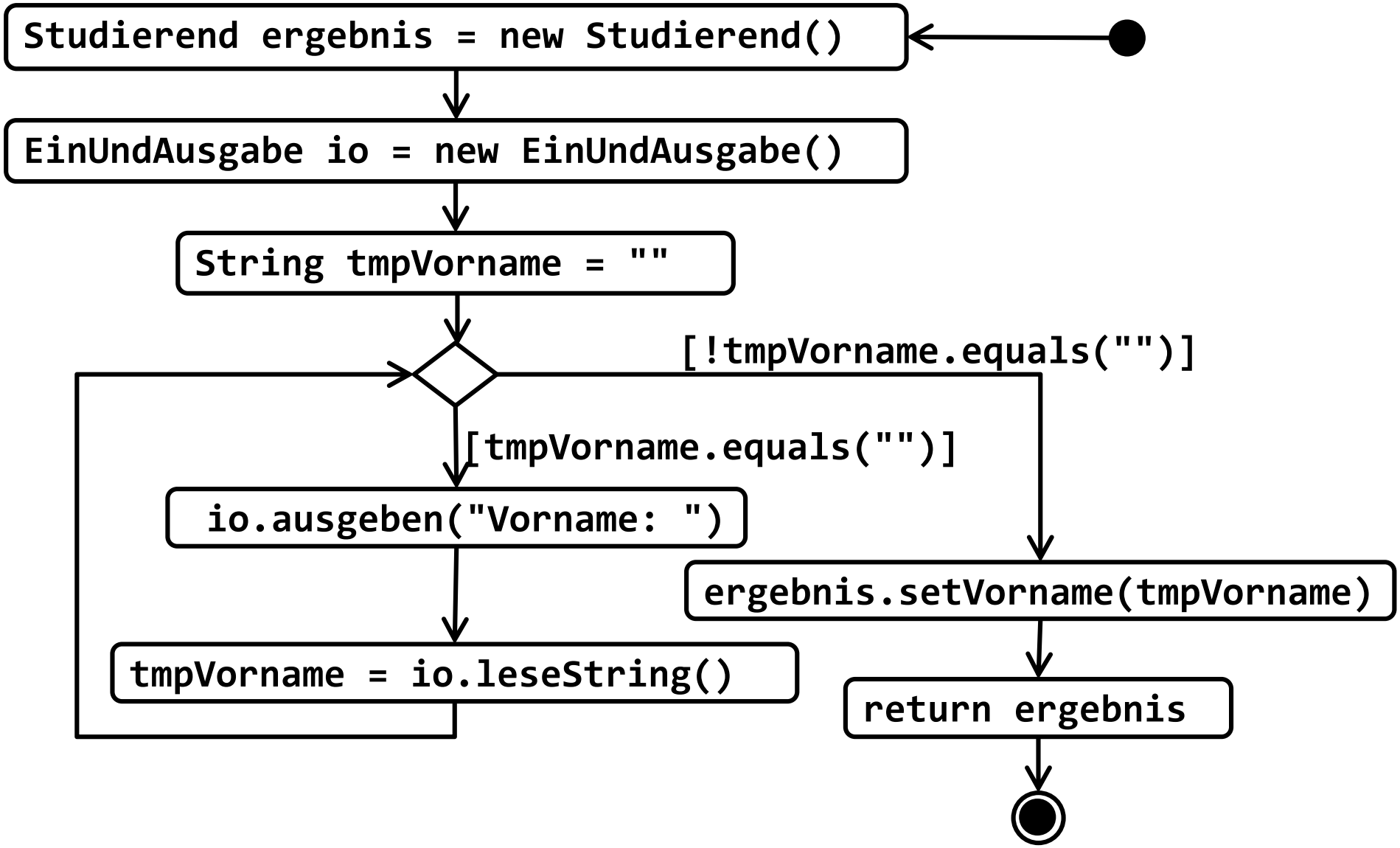


```
public class Studierendspielerei{

  public void eingeben(){
    Studierendenverwaltung sv = new Studierendenverwaltung();
    Studierend ein = sv.studierendEingeben();
    EinUndAusgabe io = new EinUndAusgabe();
    io.ausgeben("" + ein);
  }
}
```

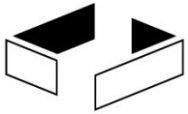
Ausprobieren in Studierendspielerei (2/2)



Visualisierung der Methode



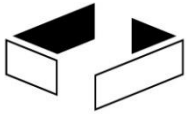
Vervollständigung der Eingabe (1/4)



- man kann jetzt ein vergleichbares Programmfragment für alle Objektvariablen schreiben
- aber Grundregel erinnern: kein Code wiederholen
- Ansatz schreibe Hilfsmethoden
  - zum Einlesen eines nicht leeren Textes
  - zum Einlesen einer ganzen Zahl aus einem Intervall [untereGrenze, obereGrenze]

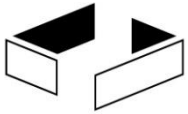
- Hinweis 1: Software-Ergonomie nicht berücksichtigt
- Hinweis 2: Eingabe mehrerer Leerzeichen wird als korrekter Text erkannt; String hat Methode trim(), die einen String mit gleichem Inhalt ohne alle umgebenden Leerzeichen zurück gibt

Vervollständigung der Eingabe (2/4)



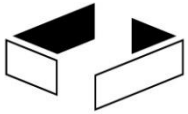
```
private String nichtLeererText(String aufforderung){
  String ergebnis = "";
  EinUndAusgabe io = new EinUndAusgabe();
  while(ergebnis.equals("")){
    io.ausgeben(aufforderung + ": ");
    ergebnis = io.leseString();
    ergebnis = ergebnis.trim();
  }
  return ergebnis;
}
```

Vervollständigung der Eingabe (3/4)



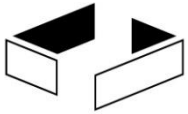
```
private int zahlZwischen(String aufforderung
                , int untereGrenze, int obereGrenze ){
  int ergebnis = untereGrenze - 1;
  EinUndAusgabe io = new EinUndAusgabe();
  while(ergebnis < untereGrenze || ergebnis > obereGrenze){
    io.ausgeben(aufforderung + ": ");
    ergebnis = io.leseInteger();
  }
  return ergebnis;
}
```

Vervollständigung der Eingabe (4/4)



```
public Studierend studierendEingeben(){
  Studierend ergebnis = new Studierend();
  ergebnis.setVorname(this.nichtLeererText("Vorname"));
  ergebnis.setNachname(this.nichtLeererText("Nachname"));
  ergebnis.setStudiengang(
        this.nichtLeererText("Studiengang"));
  ergebnis.setGeburtsjahr(
        this.zahlZwischen("Geburtsjahr",1900,2000));
  ergebnis.setMatrikelnummer(
        this.zahlZwischen("Matrikelnummer",99999,1000000));
  return ergebnis;
}
```

Beispiel für einen Nutzungsdialog

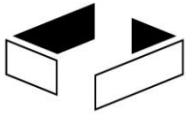


BlueJ: Konsole - BeispielErsteSchleifenStudierendeneingabe

Optionen

```
Vorname: Ernesto
Nachname:
Nachname: Guevara
Studiengang: ITI
Geburtsjahr: 1899
Geburtsjahr: 1999
Matrikelnummer: 12345
Matrikelnummer: 123456
Ernesto Guevara (123456):ITI
```

Analyse der Methodenschachtelung (1/2)



- Möglichkeit 1: Ergebnisobjekt wird Parameter eines neuen Methodenaufrufs

```
ergebnis.setVorname(this.nichtLeererText("Vorname"));
```

alternativ:

```
String tmp = this.nichtLeererText("Vorname");
ergebnis.setVorname(tmp);
```

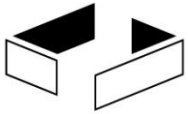
- Möglichkeit 2: auf berechnetem Ergebnisobjekt wird direkt nächste Methode aufgerufen

```
ergebnis = io.leseString().trim();
```

alternativ:

```
String tmp = io.leseString();
ergebnis = tmp.trim();
```

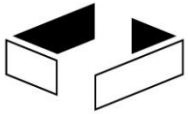
Analyse der Methodenschachtelung (2/2)



- Ist Methodenschachtelung besser als schrittweise Berechnung?
- Vorteil: Programme werden kürzer
- Vorteil: keine zusätzlichen Variablen (kein echter Vorteil, da Kompilierer den Zwischenschritt wegoptimiert)
- Nachteil: wenn Hilfsvariable sinnvollen Namen hat, würde die Lesbarkeit steigen

- Fazit: Gerade bei Anfängern sind Schachtelungen eher nicht notwendig
- Grundregel: Programme müssen so geschrieben werden, dass die unerfahrenste entwickelnde Person im Projekt diese verstehen und bearbeiten kann

Erinnerung: Weitere Kürzungsmöglichkeit



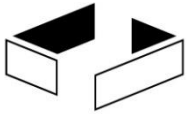
- statt

```
public boolean issesAndrea(){
  if("Andrea".equals(this.name)){
    return true;
  }
  return false;
}
```

- auch

```
public boolean issesAndrea(){
  return "Andrea".equals(this.name);
}
```

Gefahr der Endlosschleifen

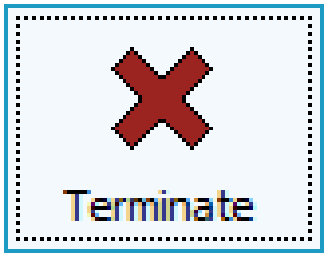


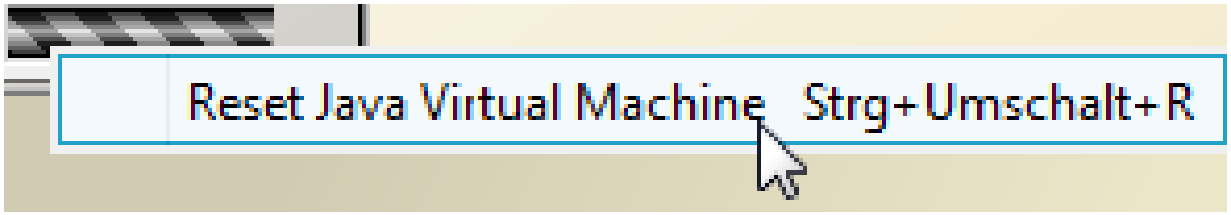
Beispiel Video

- Schleifen haben immer die Gefahr, nicht zu terminieren

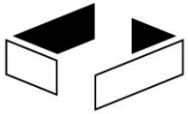
```
while(ergebnis < untereGrenze || ergebnis > obereGrenze){
```

- z. B. jemand "weigert sich" sinnvollen Wert einzugeben
- aber auch wenn z. B. untereGrenze = 1 und obereGrenze = 0 keine Chance für Eingabe, um Schleife zu beenden
- Programme benötigen Abbruchmöglichkeiten
- häufig: Betriebssystemprozess terminieren
- Entwicklungsumgebungen bieten immer Abbruchmöglichkeit (in BlueJ Debugger aufrufen und "Terminieren" wählen oder Rechtsklick auf Balken für Reset)





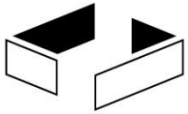
Hölzerne Programmierung



```
private String nichtLeererText(String aufforderung){
   String ergebnis = "";
   while(ergebnis.equals("")){
     EinUndAusgabe io = new EinUndAusgabe();  // aua
     io.ausgeben(aufforderung + ": ");
     ergebnis = io.leseString();
     ergebnis = ergebnis.trim();
   }
   return ergebnis;
 }
```

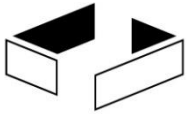
- so muss immer wieder neues Objekt für io erzeugt werden (langsam; Java-Compiler optimiert zum Glück)

Regeln zur Nutzung lokaler Variablen



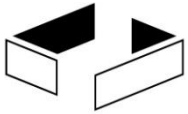
- lokale Variablen möglichst genau da deklarieren, wo sie benötigt werden
- wenn möglich, sollten lokale Variablen nicht lange leben (nicht mal genutzt und viele Zeilen später wieder genutzt werden)
- lokale Variablen sollen sprechende Namen haben (irgendwas mit tmp vielleicht akzeptabel, wenn Variable drei Zeilen später stirbt)
- lokale Variablen nur innerhalb von Schleifen deklarieren, wenn es sich um wahrscheinlich unterschiedlich genutzte Objekte handelt

Polygonzug – durch Linien verbundene Punkte (1/5)



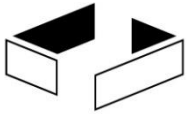
```
public class Punkt {
    private int x;
    private int y;

    public Punkt(int x, int y){
        this.x = x;
        this.y = y;
    }

    public int getX() {return this.x;}
    public void setX(int x) {this.x = x;}
    public int getY() {return this.y;}
    public void setY(int y) {this.y = y;}

    public void verschieben(int dx, int dy){     // neu!
        this.x = this.x + dx;
        this.y = this.y + dy;
    }
}
```

Polygonzug – durch Linien verbundene Punkte (2/5)



```
import java.util.ArrayList;

public class Polygonzug{
    private ArrayList<Punkt> punkte;
    private boolean geschlossen = false;

    public Polygonzug(){
        this.punkte = new ArrayList<Punkt>();
    }

    public void setGeschlossen(boolean neu){
        this.geschlossen = neu;
    }

    public void hinzufuegen(Punkt p){
        this.punkte.add(p);
    }
```

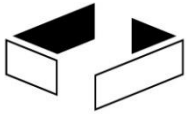
Polygonzug – durch Linien verbundene Punkte (3/5)



```
public void anzeigen(Interaktionsbrett ib){
    int anzahl = this.punkte.size();
    int zaehler = 0;
    while(zaehler < anzahl - 1){
        Punkt start = this.punkte.get(zaehler);
        Punkt ende = this.punkte.get(zaehler+1);
        ib.neueLinie(start.getX(), start.getY()
                    ,ende.getX(), ende.getY());
        zaehler = zaehler + 1;
    }
    if(geschlossen == true && anzahl > 0){
        Punkt anfang = this.punkte.get(0);
        Punkt ende = this.punkte.get(this.punkte.size()-1);
        ib.neueLinie(anfang.getX(), anfang.getY()
                    ,ende.getX(), ende.getY());
    }
}
```

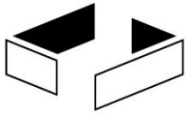
Finden Sie zwei kleine Verbesserungsmöglichkeiten!

Polygonzug – durch Linien verbundene Punkte (4/5)

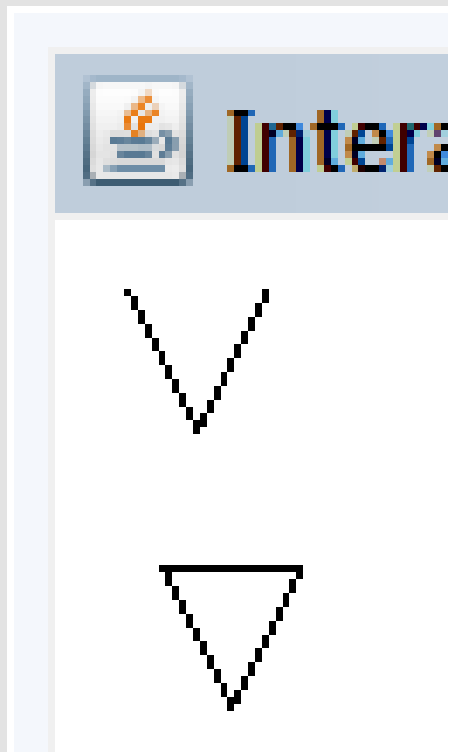


```
    public void loeschenAnPosition(int pos){
        this.punkte.remove(pos);
    }

    public void verschieben(int deltaX, int deltaY){
        int zaehler = 0;
        while(zaehler < this.punkte.size()){
            Punkt p = this.punkte.get(zaehler);
            p.verschieben(deltaX, deltaY);
            zaehler = zaehler + 1;
        }
    }
}
```

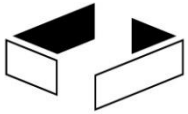
Polygonzug – durch Linien verbundene Punkte (5/5)



```
public class Polygonzugnutzung{

    public void beispiel(){
        Interaktionsbrett ib = new Interaktionsbrett();
        Polygonzug pg = new Polygonzug();
        pg.hinzufuegen(new Punkt(10,10));
        pg.hinzufuegen(new Punkt(20,30));
        pg.hinzufuegen(new Punkt(30,10));
        pg.anzeigen(ib);
        pg.verschieben(5,40);
        pg.anzeigen(ib);
        pg.loeschenAnPosition(1);
        pg.anzeigen(ib);
    }
}
```



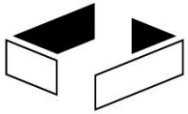
Erweiterung: Polygonzug erstellen (1/2)



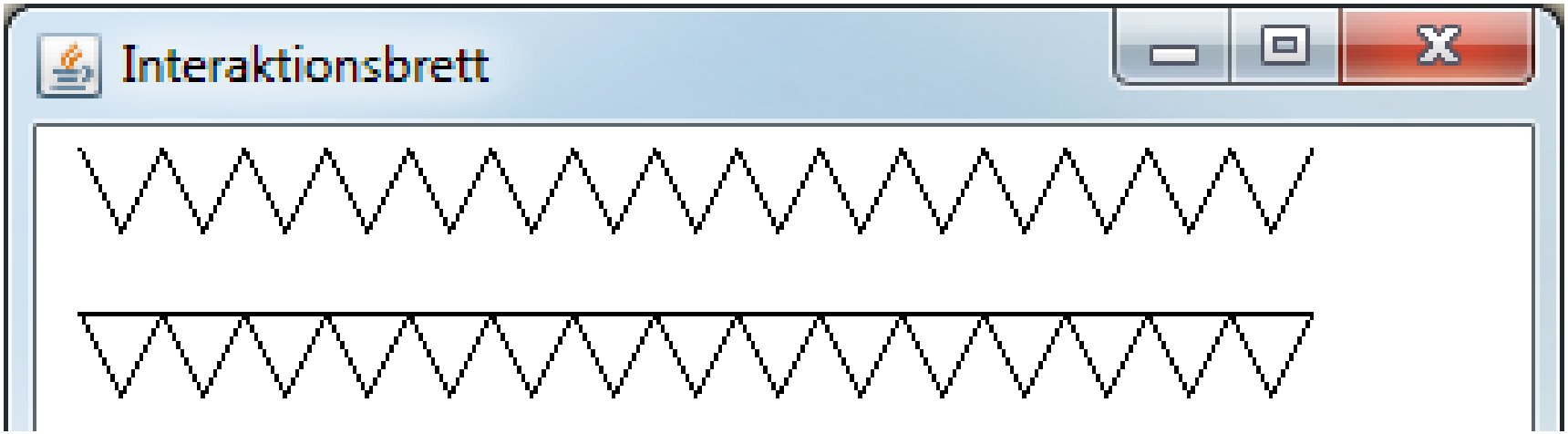
- in Polygonzugnutzung:

```
public Polygonzug zickzack(int anzahl){
  Polygonzug ergebnis = new Polygonzug();
  int zaehler = 0;
  boolean zick = true;
  while (zaehler < anzahl){
    if(zick == true){
      ergebnis.hinzufuegen(new Punkt(10+zaehler*10,5));
    } else {
      ergebnis.hinzufuegen(new Punkt(10+zaehler*10,25));
    }
    zick = !zick;
    zaehler = zaehler+1;
  }
  return ergebnis;
}
```

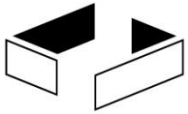
Erweiterung: Polygonzug erstellen (2/2)



```
public void beispiel2(){
    Interaktionsbrett ib = new Interaktionsbrett();
    Polygonzug pg = this.zickzack(31);
    pg.anzeigen(ib);
    pg.verschieben(0,40);
    pg.setGeschlossen(true);
    pg.anzeigen(ib);
}
```



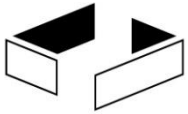
Erweiterung: Punkte auswählen (1/2)



- in Polygonzug: neuer Polygonzug mit Punkten an 0-ter, n-ter, 2*n-ter, … Position

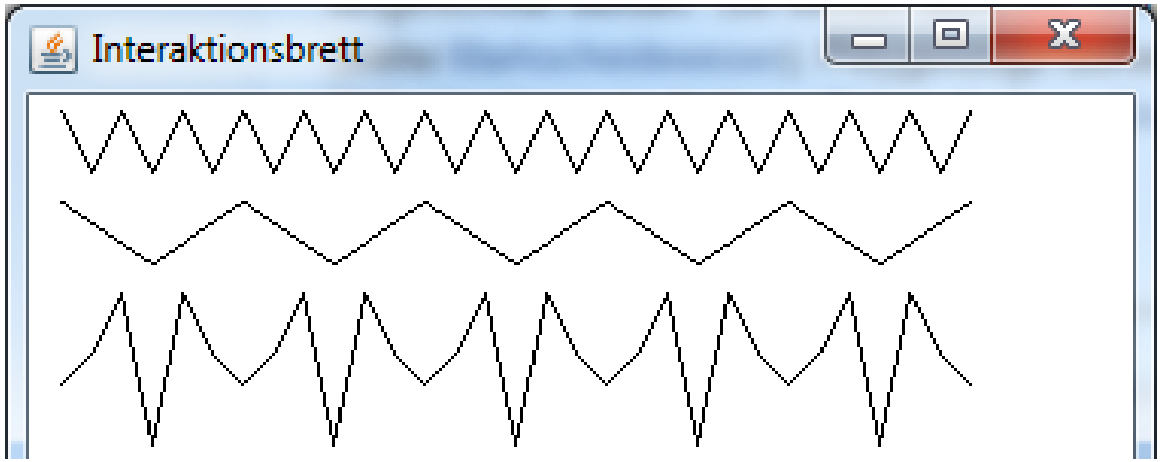
```
public Polygonzug nurNtePunkte(int n){
    Polygonzug ergebnis = new Polygonzug();
    int zaehler = 0;
    while(zaehler < this.punkte.size()){
        Punkt p = this.punkte.get(zaehler);
        ergebnis.hinzufuegen(p);
        zaehler = zaehler + n;
    }
    return ergebnis;
}
```

Erweiterung: Punkte auswählen (2/2)

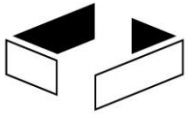


- in Polygonzugnutzung:

```
public void beispiel3(){
        Interaktionsbrett ib = new Interaktionsbrett();
        Polygonzug pg = this.zickzack(31);
        pg.anzeigen(ib);
        Polygonzug pg2 = pg.nurNtePunkte(3);
        pg2.verschieben(0,30);
        pg2.anzeigen(ib);
        pg.verschieben(0,60);
        pg.anzeigen(ib);
    }
```



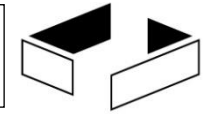
Schleifen-Variante - Beispiel



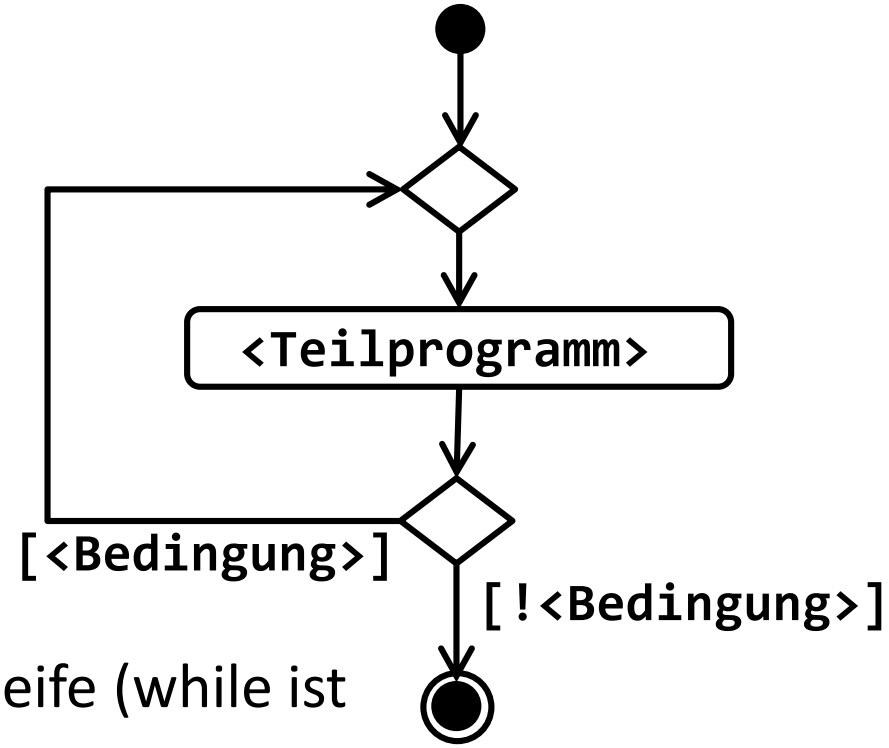
Video

```
private String nichtLeererText(String aufforderung){
  String ergebnis;
  EinUndAusgabe io = new EinUndAusgabe();
  do {
    io.ausgeben(aufforderung+": ");
    ergebnis = io.leseString();
    ergebnis = ergebnis.trim();
  } while(ergebnis.equals(""));
  return ergebnis;
}
```

Analyse der do-Schleife

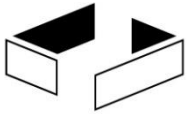


```
do {
    <Teilprogramm>
} while (<Bedingung>)
```



- wird mindestens einmal durchlaufen
- Schleife wird wiederholt solange Bedingung wahr ist
  - heißt auch fußgesteuerte Schleife (while ist kopfgesteuerte Schleife)
  - obere Raute: Vereinigung von zwei Pfaden
  - jede while-Schleife ist in do-Schleife verwandelbar und umgekehrt
  - viele entwickelnde Personen verzichten auf do-Schleife

Weitere Schleifenbeispiele



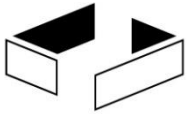
Beispiel Video

- Innerhalb von Methoden "spürt" man teilweise wenig von Objektorientierung (hauptsächlich Aufruf von Methoden)
- man greift auf Objektvariablen zu und deklariert lokale Hilfsvariablen
- folgende Beispiele fokussieren Zahlen und Sammlungen von Zahlen (genauer ArrayList)
- Methoden liegen in der Klasse

```
public class Zahlenanalyse {
  private ArrayList<Integer> zahlen;
  private String name = "default";

  public Zahlenanalyse(ArrayList<Integer> zahlen) {
    this.zahlen = zahlen;
  } …
```

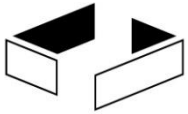
- name kann z. B. Messwerte oder Lottozahlen sein

weiterer Konstruktor, get – und set – Methoden



```
public Zahlenanalyse(ArrayList<Integer> zahlen, String name) {
  this(zahlen);
  this.name = name;
}

public void hinzufuegen(int wert){
      this.zahlen.add(wert);
}

public ArrayList<Integer> getZahlen() {
  return this.zahlen;
}

public void setZahlen(ArrayList<Integer> zahlen) {
  this.zahlen = zahlen;
}

// getName, setName, toString
```

Zeige Gerade Zahlen in einem Intervall

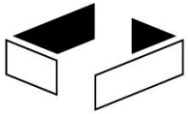


- typisch: Innerhalb der Schleife wird jedes Element analysiert, ob es bestimmte Eigenschaft hat

```
public void zeigeGeradeZahlen(int start, int ende){
  int zaehler = start;
  EinUndAusgabe io = new EinUndAusgabe();
  while(zaehler <= ende){
    if(zaehler % 2 == 0){
      io.ausgeben(" "+zaehler);
    }
    zaehler = zaehler + 1;
  }
}
```

- untypisch: Berechnung und Ausgabe innerhalb einer Methode, besser: eine Methode für Berechnungen, eine Methode für Ausgaben (u. a. Ausgabe leicht änderbar)

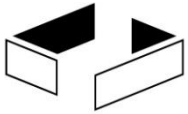
Aufteilung von Berechnung und Ausgabe (1/3)



```
public void geradeZahlenAus(int start, int ende){
  if (this.zahlen == null){
    this.zahlen = new ArrayList<Integer>();
  }
  int zaehler = start;
  while(zaehler <= ende){
    if(zaehler % 2 == 0){
      this.zahlen.add(zaehler);
    }
    zaehler = zaehler + 1;
  }
}
```

- Hier Nutzung der Objektvariable zahlen
- generell erst prüfen, ob zahlen==null, sonst Fehler
- was passiert bei mehrmaligem Aufruf?

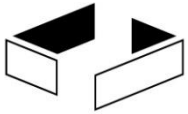
Aufteilung von Berechnung und Ausgabe (2/3)



- einfaches Iterieren über eine Liste mit get und size (etwas hölzern, siehe später)

```
public void listeUntereinanderAusgeben(){
  EinUndAusgabe io = new EinUndAusgabe();
  int zaehler = 0;
  while(zaehler < this.zahlen.size()){
    io.ausgeben("\t"+this.zahlen.get(zaehler)+"\n");
    zaehler = zaehler + 1;
  }
}
```

Aufteilung von Berechnung und Ausgabe (3/3)



- Ausprobieren in nutzender Hilfsklasse

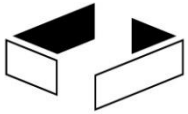
```
import java.util.ArrayList;

public class ZahlenanalyseSpielerei {
  public void zeigeGeradeZahlen(){
    EinUndAusgabe io = new EinUndAusgabe();
    Zahlenanalyse zana = new Zahlenanalyse(null);
    io.ausgeben("Startwert: ");
    int start = io.leseInteger();
    io.ausgeben("Endwert: ");
    int ende = io.leseInteger();
    zana.geradeZahlenAus(start, ende);
    zana.listeUntereinanderAusgeben();
  }
}
```

```
Startwert: -3
Endwert: 3
        -2
        0
        2
```

Video

switch

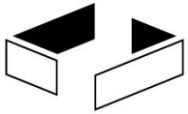
Einfacher Nutzungsdialog mit switch (1/3)



- Typische Dialogstruktur mit Auswahlalternativen
- würde langes verschachteltes if benötigen

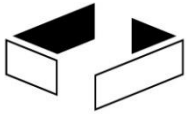
```
public class Minidialog {
    public void dialog(){
    EinUndAusgabe io = new EinUndAusgabe();
    int ein = -1;
    while(ein != 0){
      io.ausgeben("(0) Programmende\n"
          + "(1) waschen, f\u00f6hnen, schneiden\n"
          + "(2) schneiden\n"
          + "(3) rasieren\n"
          + " Ihre Auswahl: ");
      ein = io.leseInteger();
```

Einfacher Nutzungsdialog mit switch (2/3)



```
        switch(ein){
          case 0: {
            break;
          }
          case 1: {
            io.ausgeben("wash and dry\n");
          }
          case 2: {
            io.ausgeben("cut\n");
            break;
          }
          case 3: {
            io.ausgeben("shave\n");
            break;
          }
          default: {
            io.ausgeben("Zahl zwischen 0 und 3\n");
            break;
          }
        }
      }
    }
  }
```

Einfache Nutzungsdialog mit switch (3/3)

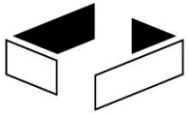


- Beispieldialog

```
(0) Programmende
(1) waschen, föhnen, schneiden
(2) schneiden
(3) rasieren
 Ihre Auswahl: 4
Zahl zwischen 0 und 3
(0) Programmende
(1) waschen, föhnen, schneiden
(2) schneiden
(3) rasieren
 Ihre Auswahl: 3
shave
(0) Programmende
(1) waschen, föhnen, schneiden
(2) schneiden
(3) rasieren
 Ihre Auswahl: 2
cut
```

```
(0) Programmende
(1) waschen, föhnen, schneiden
(2) schneiden
(3) rasieren
 Ihre Auswahl: 1
wash and dry
cut
(0) Programmende
(1) waschen, föhnen, schneiden
(2) schneiden
(3) rasieren
 Ihre Auswahl: 0
```

Syntaxanalyse von switch



```
switch(ein){
  case 0: {
    break;
  }
  case 1: {
    io.ausgeben("wash");
  }
  case 2: {
    io.ausgeben("cut\n");
    break;
  }
  default: {
    io.ausgeben("Fehler");
    break;
  }
}
```

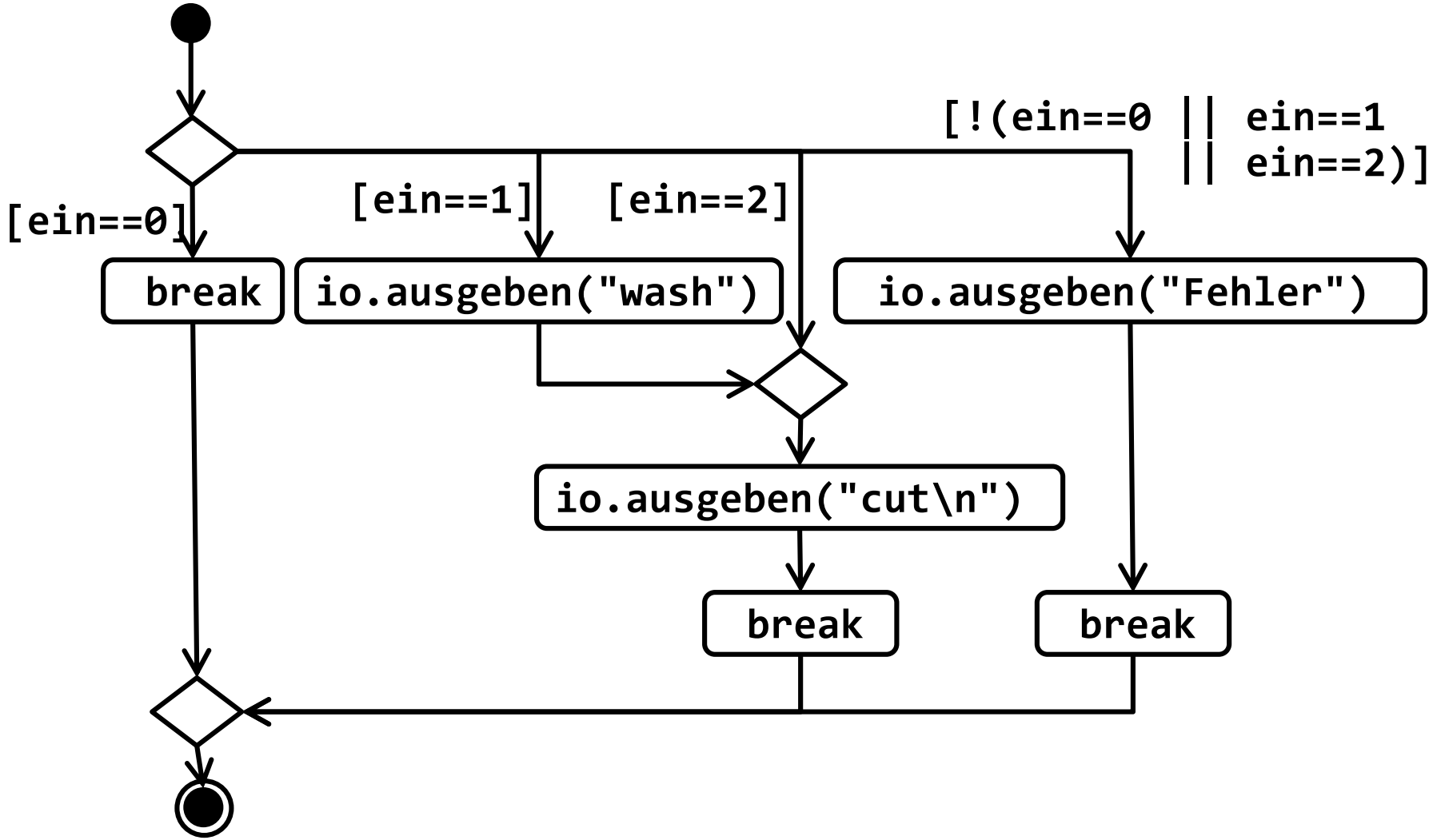
Ausdruck, der zu einer Zahl oder einem Buchstaben (ab Java 7 auch String) ausgewertet wird

Konstante, die zum Typ des Ausdruck passt; stimmen die Werte überein, wird zugehöriges Teilprogramm ausgeführt

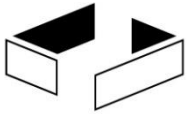
break zum Verlassen des Blocks, sonst werden Anweisungen des Folgeblocks ausgeführt

default ist optional; genutzt, wenn kein anderer Fall zutrifft

Aktivitätsdiagramm für switch



Programmfragment mit if statt switch

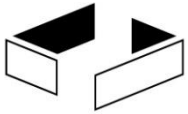


```
        if (ein == 0) {
          ;
        } else {
            if (ein == 1 || ein == 2) {
              if (ein == 1) {
                io.ausgeben("wash and dry\n");
              }
              io.ausgeben("cut\n");
            } else {
                if (ein == 3) {
                  io.ausgeben("shave\n");
                } else {
                  io.ausgeben("Zahl zwischen 0 und 3\n");
                }
            }
        }
    }
  }
```

Video

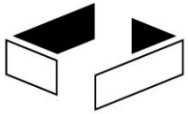
Algorithmus

Primzahlanalyse – Aufgabenstellung



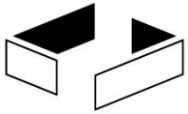
- Primzahl: Natürliche Zahl n mit zwei Teilern: 1 und n
- konkret: 2, 3 ,5 ,7, …
- erste Überlegung zum Berechnungsverfahren (Algorithmus):
  - Eingabe soll ein int-Wert n sein
  - nimm zunächst an, dass es eine Primzahl ist
  - prüfe von zwei aufsteigend bis n, ob n durch diese Zahl teilbar ist; ist es so, dann ist es keine Primzahl
  - gib Ergebnis zurück
  - da ganze Zahl übergeben wird, muss für nicht-natürliche Zahlen die Antwort false sein (da Primzahlen erst bei zwei beginnen, gilt das Ergebnis für alle n < 2)

Primzahlanalyse – Version 0 (nicht schlecht)



```
public boolean istPrimzahl0(int zahl){
  if (zahl < 2){
    return false;
  }
  boolean ergebnis = true;
  int testwert = 2;
  while(testwert < zahl){
    if(zahl % testwert == 0){
      ergebnis = false;
    }
    testwert = testwert + 1;
  }
  return ergebnis;
}
```

Primzahlanalyse – Version 1 (optimiert)

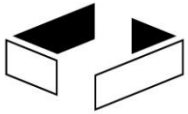


- gerade Zahlen ausschließen
- Schleife beenden, wenn es keine Primzahl ist

```
public boolean istPrimzahl(int zahl){
  if (zahl < 2){
    return false;
  }
  boolean ergebnis = ((zahl % 2 != 0) || (zahl == 2));
  int testwert = 3;
  while(testwert < zahl && ergebnis){
    if(zahl % testwert == 0){
      ergebnis = false;
    }
    testwert = testwert + 2;
  }
  return ergebnis;
}
```

- fehlt: Schleife nur bis Wurzel aus n laufen lassen

Der Begriff des Algorithmus (1/2)

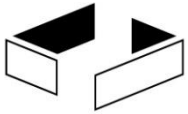


- Ein Algorithmus ist ein Verfahren (genauer die Beschreibung eines Verfahrens) zur Lösung einer gegebenen Aufgabe
- “Algorithmus” ist einer der zentralen Begriffe der Informatik, aber auch bekannt in vielen anderen Zusammenhängen, z. B.
  - Kochrezept
  - Anweisung zum Zusammenbau eines Geräts, Möbelstücks, . . .
  - Gebrauchsanweisung
  - Musik-Noten
- Die Beschreibung eines Algorithmus kann in sehr unterschiedlicher Form (Sprache) gegeben werden, z. B. Programmiersprache

Der Begriff des Algorithmus (2/2)

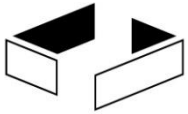
- Der Begriff leitet sich ab von der latinisierten Form des Namens Muhammad ibn Musa Al-Khwarizmi
- (Abu Ja'far) Muhammad ibn Musa Al-Khwarizmi:
  - bedeutender Mathematiker und Astronom,
  - lebte ca. 780 – 840, hauptsächlich in Bagdad
  - führte Null und Dezimalsystem in westliche Welt ein
  - Sein Text Hisab al-jabr w'al-muqabala ist erstes Lehrbuch zur Algebra,
  - beschreibt insbesondere das Lösen von Gleichungen
  - der Begriff "Algebra" ist vom Titel abgeleitet

Eigenschaften von Algorithmen



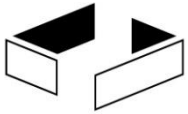
Eigenschaft	Bedeutung
Komplexität	charakterisiert den für die Ausführung erforderlichen Aufwand an Rechenzeit und Speicherplatz
Effizienz	der Algorithmus kommt mit möglichst wenig Zeit- und Speicheraufwand aus
Endlichkeit (statisch)	Beschreibung des Algorithmus ist von endlicher Länge
Endlichkeit (dynamisch)	für Ausführung benötigten Ressourcen (Speicherplatz, Rechenzeit) sind jederzeit endlich
Terminiert-heit	für jede zulässige Eingabe liefert der Algorithmus nach endlich vielen Schritten ein Ergebnis
Determi-nismus	Zu jeder Aktion existiert in einem Algorithmus höchstens eine Folgeaktion und der Berechnungsablauf ist dadurch eindeutig bestimmt
Determi-niertheit	Ausgabewerte eindeutig durch die Eingabewerte bestimmt (Determinismus impliziert Determiniertheit)

Eigenschaften von `istPrimzahl(int zahl)`



Eigenschaft	Bedeutung
Komplexität	Schrittanzahl klar von Eingabe abhängig, benötigte Variablen einfach ablesbar
Effizienz	Version 1 ist schneller als Version 0 (weniger Schritte)
Endlichkeit (statisch)	ca. 14 Zeilen
Endlichkeit (dynamisch)	Speicher nur für Variablen ergebnis und testwert, Rechenzeit begrenzt durch zahl*x Schritte
Terminiert-heit	klare, für bestimmte Zahl immer gleiche Antwort: true oder false
Determi-nismus	Sequenz, Schleife mit ja oder nein, if mit einem Fall
Determi-niertheit	für jeden int-Wert jederzeit das gleiche Ergebnis

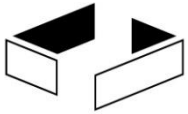
Zentrale Aufgabe: Algorithmen optimieren



- 0. Schritt: Klare Definition der Aufgabenstellung
- 1. Schritt: Entwicklung einer Lösung
- 2. Schritt: Optimierung der Lösung
- Abhängig von der Relevanz der Aufgabenstellung steigt oder fällt die Optimierungsbedeutung
- Bremsen im Auto, Turbulenzen beim Fliegen, andere Umwelteinflüsse beim Raketenstart, auf einen zufliegender Flugkörper, minimale Schwankung des Aktienpreises: Effizienz bekommt enorme Bedeutung!

- → Vorlesung "Algorithmen und Datenstrukturen" (nicht nur Verfahren, auch Schulung der Denkweise)

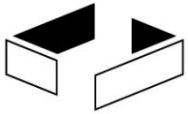
Bestimme Primzahl in einem Zahlenbereich



- Aufgabe: Fülle Liste in Zahlenanalyse mit Primzahlen aus einem vorgegebenen Bereich

```
public void zahlbereichMitPrim(int start, int ende) {
  if (this.zahlen == null){
    this.zahlen = new ArrayList<Integer>();
  }
  int zaehler = start;
  while (zaehler <= ende) {
    if (this.istPrimzahl(zaehler)) {
      this.zahlen.add(zaehler);
    }
    zaehler = zaehler + 1;
  }
}
```

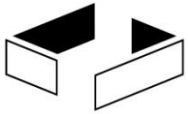
Listenauswahl



- Suche alle Zahlen, die mit der Ziffer 3 enden und füge sie in ein neues Objekt der Klasse Zahlenanalyse

```
public Zahlenanalyse zahlenMitDreiAmEnde() {
  ArrayList<Integer> ergebnis = new ArrayList<Integer>();
  int zaehler = 0;
  if (this.zahlen != null) { // kann man weglassen
    while (zaehler < this.zahlen.size()) {
      int element = this.zahlen.get(zaehler);
      if (element % 10 == 3) {
        ergebnis.add(element);
      }
      zaehler = zaehler + 1;
    }
  }
  return new Zahlenanalyse(ergebnis);
}
```

Listenauswahl



- Suche alle Zahlen, die mit der Ziffer 3 enden und füge sie in ein neues Objekt der Klasse Zahlenanalyse

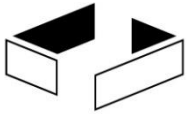
```
public Zahlenanalyse zahlenMitDreiAmEnde() {
  ArrayList<Integer> ergebnis = new ArrayList<Integer>();
  int zaehler = 0;
  if (this.zahlen != null) { // kann man weglassen
    while (zaehler < this.zahlen.size()) {
      int element = this.zahlen.get(zaehler);
      if (element % 10 == 3) {
        ergebnis.add(element);
      }
      zaehler = zaehler + 1;
    }
  }
  return new Zahlenanalyse(ergebnis);
}
```

Beispiel

Video

geschachtelte Schleifen

Ineinander geschachtelte Schleifen

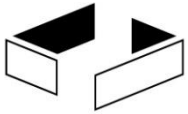


- die innere Schleife wird für jedes Element der äußeren Schleife durchlaufen !!!

```
public void schleifeInSchleife(){
  EinUndAusgabe io = new EinUndAusgabe();
  int zaehler = 0;
  while (zaehler < 5){
    int zaehler2 = 20;
    while (zaehler2 < 25){
      io.ausgeben(zaehler+":"+zaehler2+" ");
      zaehler2 = zaehler2 + 1;
    }
    zaehler = zaehler + 1;
    io.ausgeben("\n");
  }
}
```

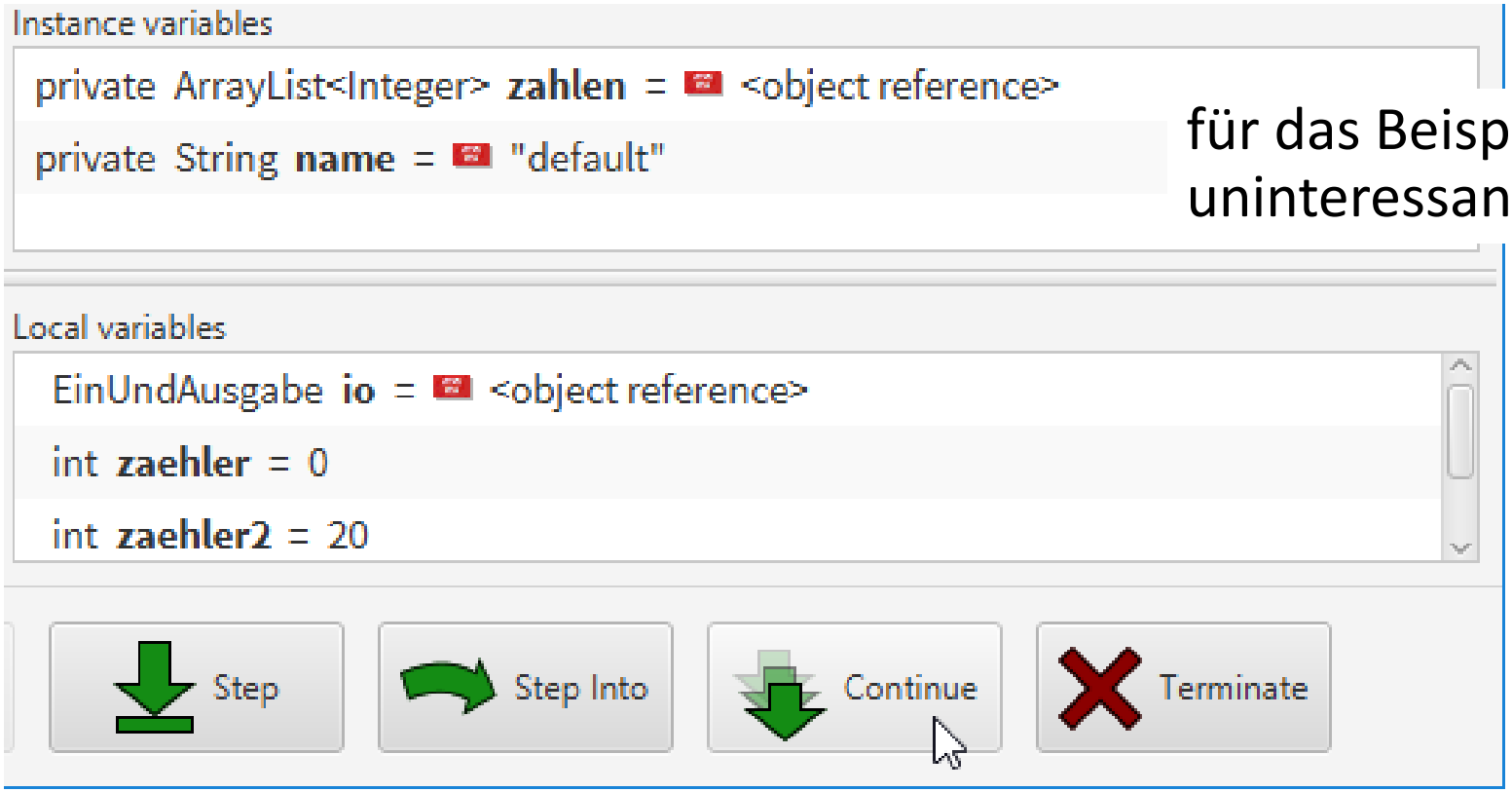
```
0:20 0:21 0:22 0:23 0:24
1:20 1:21 1:22 1:23 1:24
2:20 2:21 2:22 2:23 2:24
3:20 3:21 3:22 3:23 3:24
4:20 4:21 4:22 4:23 4:24
```

Mit Break Point (1/4)



```
152     public void schleifeInSchleife() {
153         EinUndAusgabe io = new EinUndAusgabe();
154         Integer zaehler = 0;
155         while (zaehler < 5) {
156             Integer zaehler2 = 20;
157             while (zaehler2 < 25) {
158                 io.ausgeben(zaehler + ":" + zaehler2 + " ");
159                 zaehler2 = zaehler2 + 1;
160             }
161             zaehler = zaehler + 1;
162             io.ausgeben("\n");
163         }
164     }
```

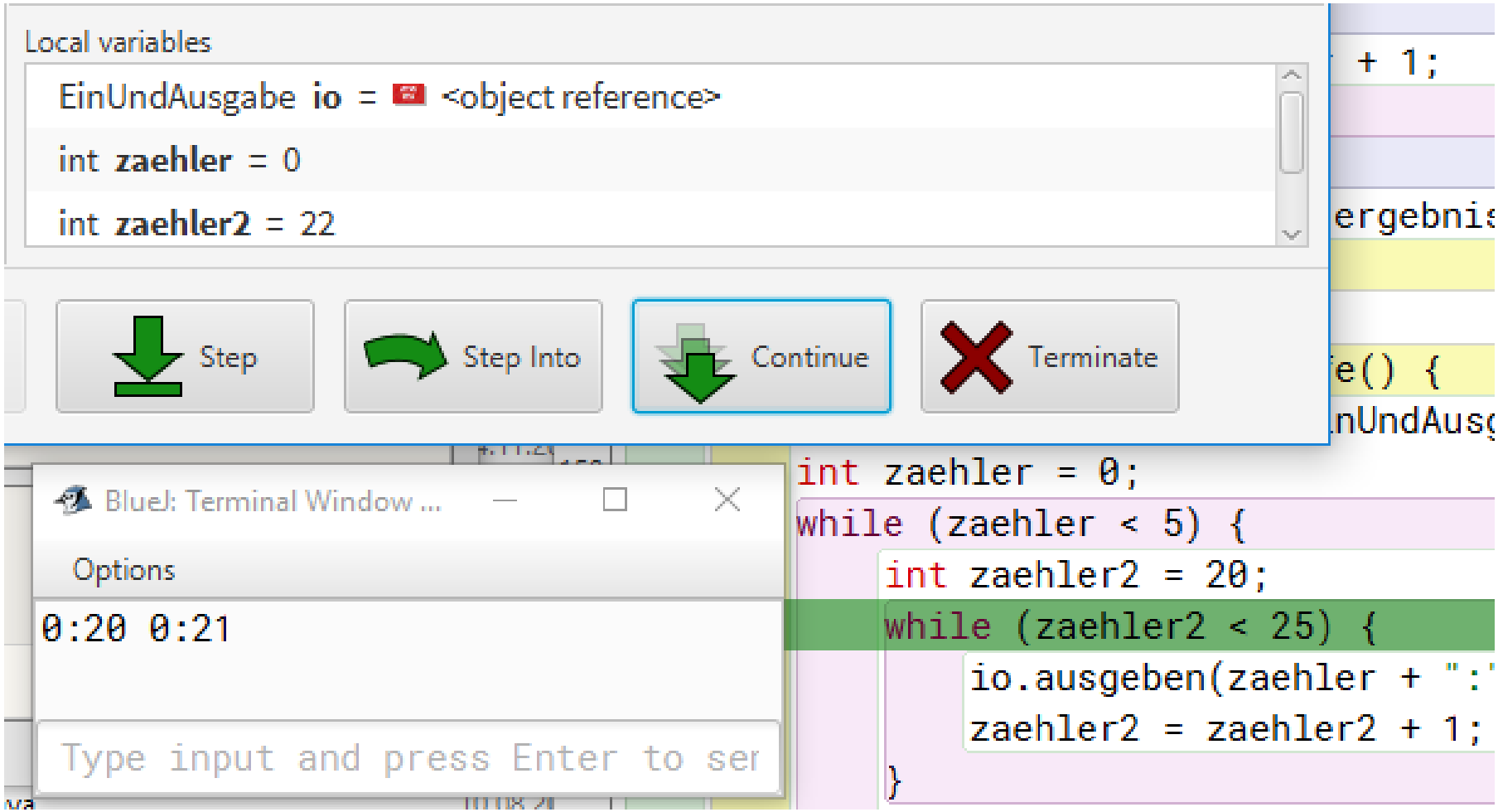
Mit Break Point (2/4)



für das Beispiel uninteressant

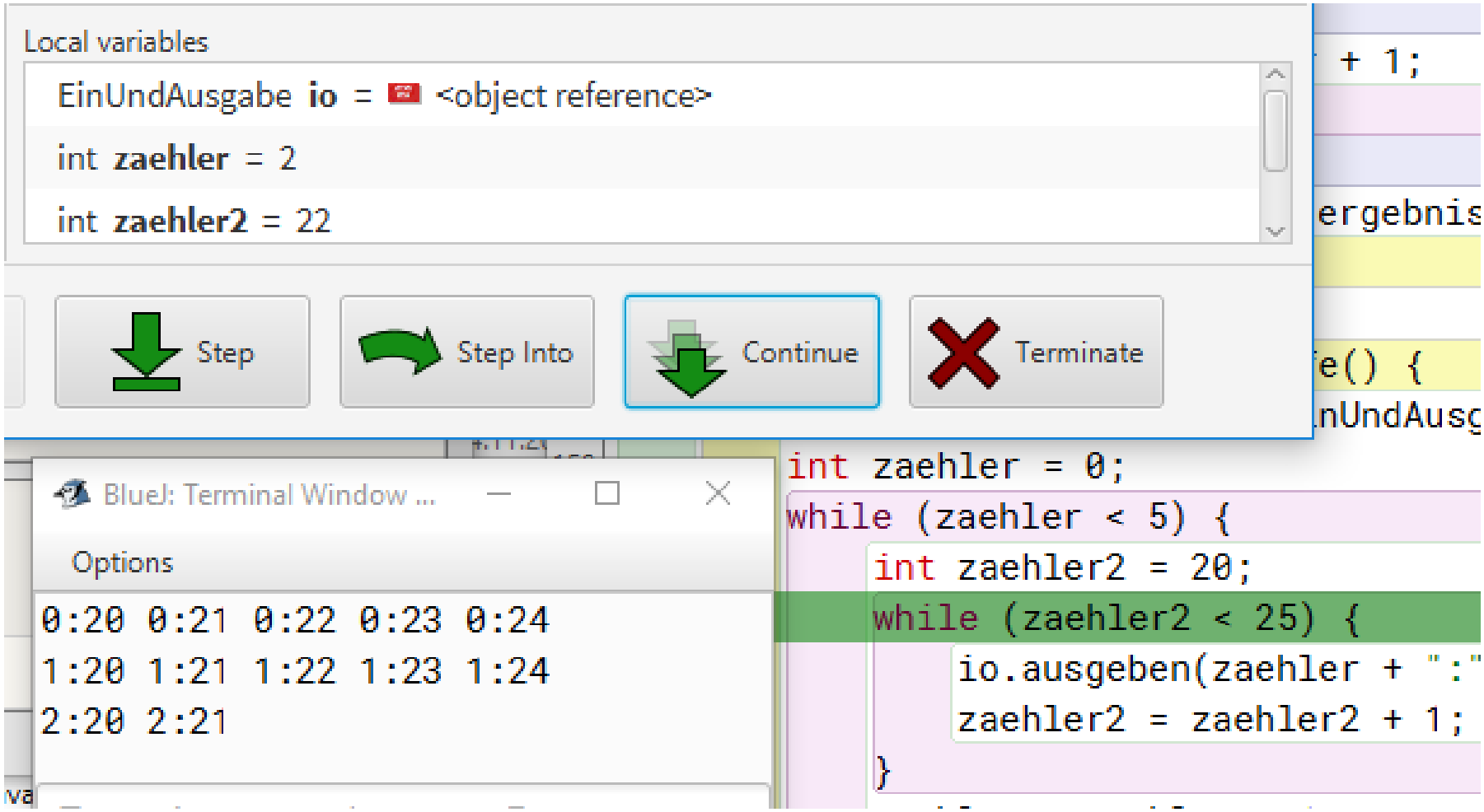
- Ausgangssituation
- mit Continue wird zum nächsten Break Point gelaufen (also ein Schleifendurchlauf) [wird zweimal gemacht]

Mit Break Point (3/4)

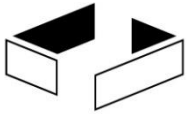


- [12 mal "Continue" drücken]

Mit Break Point (4/4)



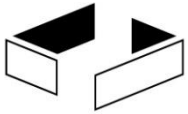
Strukturierte Ausgabe der Zahlenanalyse



- in jeder Zeile genau anzahl Elemente (Ausnahme letzte Zeile)

```
public void strukturiertAusgeben(int anzahl) {
  if (this.zahlen == null){
    return;
  }
  EinUndAusgabe io = new EinUndAusgabe();
  int zaehler = 0;
  while (zaehler < this.zahlen.size()) {
    io.ausgeben("" + this.zahlen.get(zaehler) + "\t");
    zaehler = zaehler + 1;
    if(zaehler % anzahl == 0){
      io.ausgeben("\n");
    }
  }
}
```

Anwendung bisheriger Methoden



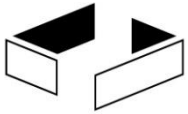
```
public void primAnalysieren(){ // in ZahlenAnalyseSpielerei
  Zahlenanalyse zana = new Zahlenanalyse(null);
  zana.zahlbereichMitPrim(100, 470);
  Zahlenanalyse tmp = zana.zahlenMitDreiAmEnde();
  tmp.strukturiertAusgeben(5);
}
```

```
103    113    163    173    193
223    233    263    283    293
313    353    373    383    433
443    463
```

letzten beiden Zeilen alternativ zusammengefasst als:

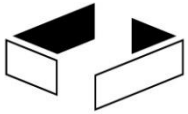
```
zana.zahlenMitDreiAmEnde().strukturiertAusgeben(5)
```

Gemeinsamkeit (1/3)



- Aufgabe: Gibt es in zwei Listen gemeinsame Elemente?
- Ansatz: Annahme, es gibt keine gemeinsamen Elemente, bis eines gefunden ist
- Nimm erstes Element der ersten Liste und überprüfe mit jedem Element der zweiten Liste ob diese gleich sind, wenn true, gibt Antwort aus
- …
- Nimm letztes Element der ersten Liste und überprüfe mit jedem Element der zweiten Liste ob diese gleich sind, wenn true, gibt Antwort aus
- Gib false als Antwort aus

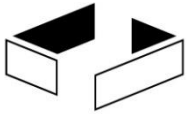
Gemeinsamkeit (2/3)



```
public boolean gibtGleiche(Zahlenanalyse other) {
  ArrayList<Integer> liste = other.getZahlen();
  int zaehler1 = 0;
  while (zaehler1 < this.zahlen.size()) {
    int zaehler2 = 0;
    while (zaehler2 < liste.size()) {
      if(this.zahlen.get(zaehler1) == liste.get(zaehler2)){
        return true;
      }
      zaehler2 = zaehler2 + 1;
    }
    zaehler1 = zaehler1 + 1;
  }
  return false;
}
```

Achtung, hier fehlt was: Es fehlen die Prüfungen, ob eine der Listen null ist (Fehler werden wir später finden)

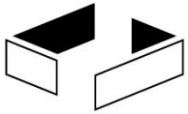
Gemeinsamkeit (3/3)



```
public void gleicheAnalysieren() {
  Zahlenanalyse tmp1 = new Zahlenanalyse(null);
  tmp1.geradeZahlenAus(1, 23);
  Zahlenanalyse tmp2 = new Zahlenanalyse(null);
  tmp2.geradeZahlenAus(23, 42);
  Zahlenanalyse tmp3 = new Zahlenanalyse(null);
  tmp3.geradeZahlenAus(22, 32);
  EinUndAusgabe io = new EinUndAusgabe();
  io.ausgeben("tmp1 und tmp2: "+tmp1.gibtGleiche(tmp2)+ "\n");
  io.ausgeben("tmp2 und tmp1: "+tmp2.gibtGleiche(tmp1)+ "\n");
  io.ausgeben("tmp1 und tmp3: "+tmp1.gibtGleiche(tmp3)+ "\n");
  io.ausgeben("tmp3 und tmp1: "+tmp3.gibtGleiche(tmp1)+ "\n");
}
```

```
tmp1 und tmp2: false
tmp2 und tmp1: false
tmp1 und tmp3: true
tmp3 und tmp1: true
```

Visualisierung (1/3)

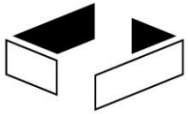


```
other.getZahlen()
zaehler2:  0   1   2   3   4
```

	0	1	2	3	4
this.zahlen	1	9	34	42	8
zaehler1	≠	≠	≠	≠	≠

0	17
1	3
2	34
3	2

Visualisierung (2/3)



```
other.getZahlen()
zaehler2:  0   1   2   3   4
```

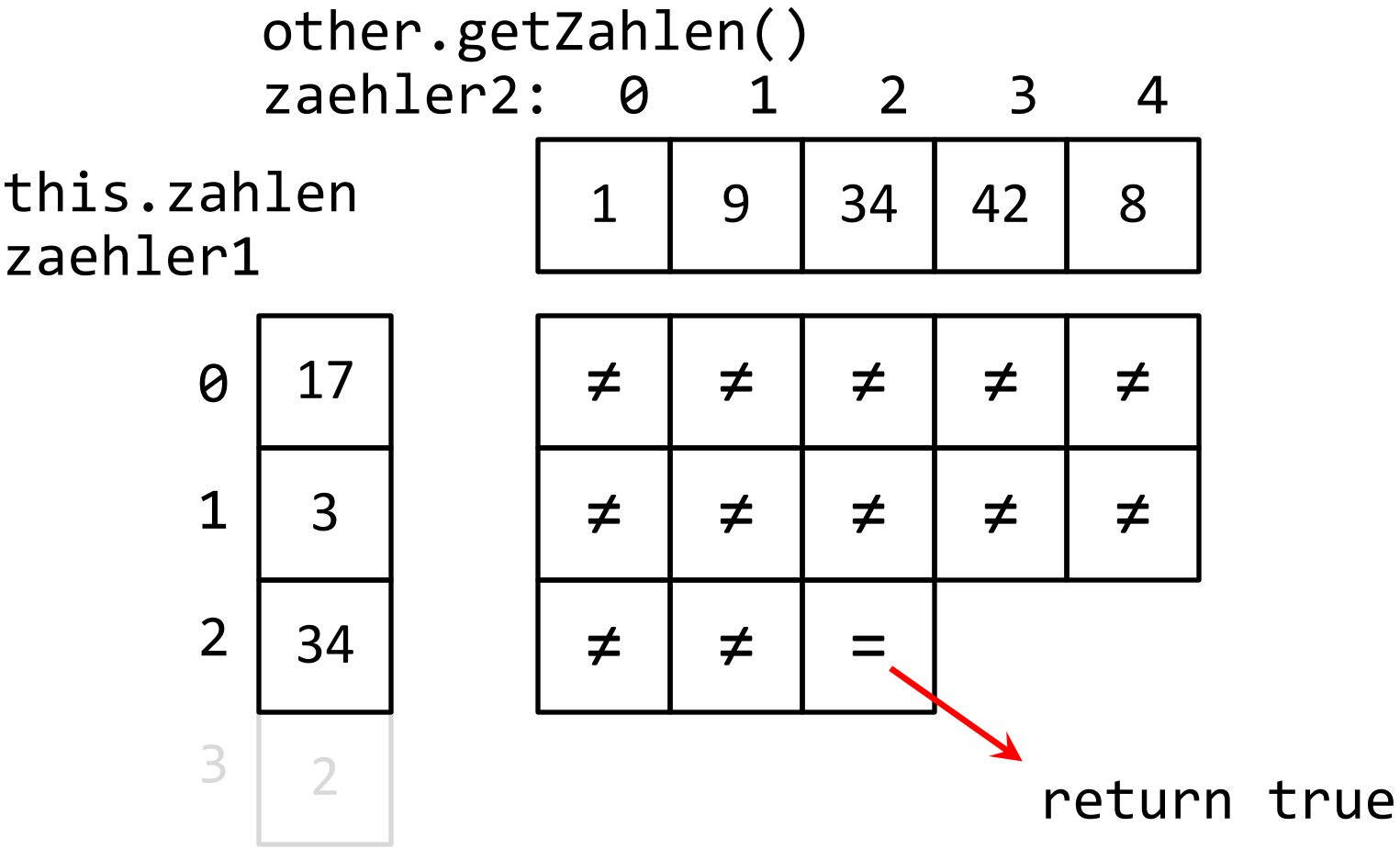
1	9	34	42	8

```
this.zahlen
zaehler1
```

0	17
1	3
2	34
3	2

≠	≠	≠	≠	≠
≠	≠	≠	≠	≠

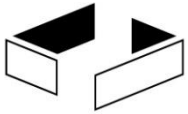
Visualisierung (3/3)



Video

Iterator

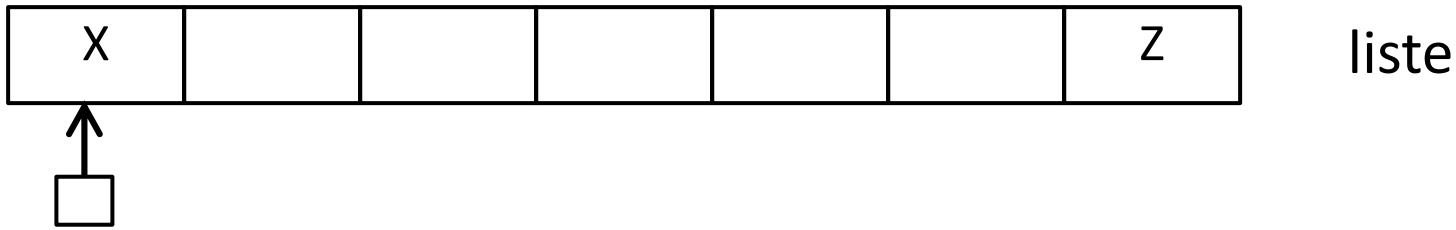
Effizientere Listenbearbeitung - Iterator



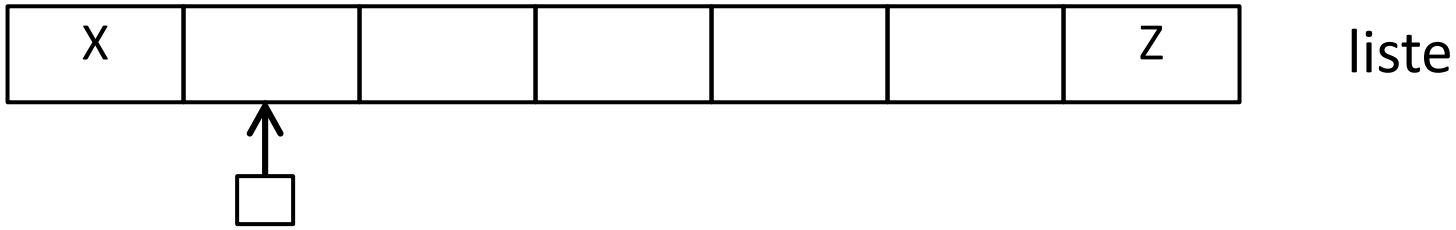
- Problem: Der Zugriff mit get(i) ist relativ langsam, man kann schneller von i zum i+1-Element manövrieren
- Zur schrittweisen Bearbeitung von Listen gibt es Iterator-Objekte; Objekt kennt aktuell betrachtetes Element
- Listen haben Methode iterator(); gibt Iteratorobjekt
- Auszug aus Methoden von Iterator<Typ>

Metho-denname	Parameter	Ergebnis	Beschreibung
hasNext		boolean	gibt es für Iterator nächstes Element?
next		Typ	gibt aktuelles Element des Iterators zurück und setzt ihn ein Element weiter

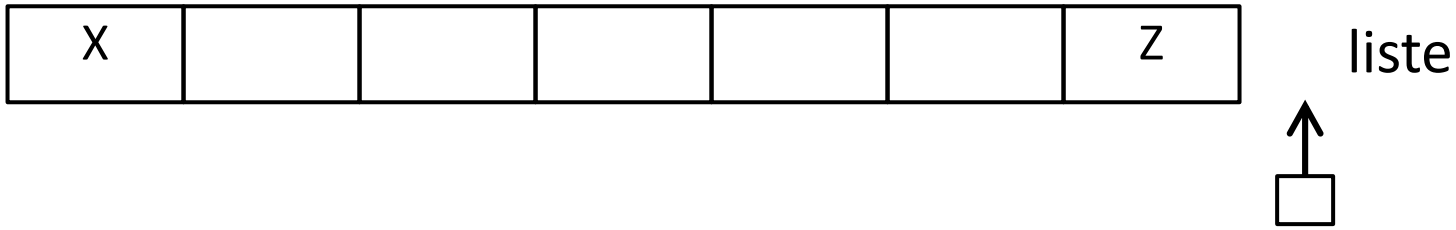
Iteratornutzung - allgemein



Iterator<Integer> i = liste.iterator()
i.hasNext() ergibt true

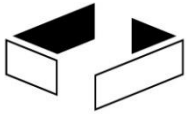


i.next() ergibt X und Iterator wird Feld weiter gesetzt
i.hasNext() ergibt true



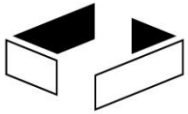
i.next() ergibt Z und Iterator wird Feld weiter gesetzt
i.hasNext() ergibt false

Methodenvergleich



```
public void listeUntereinanderAusgebenIterator() {
  EinUndAusgabe io = new EinUndAusgabe();
  Iterator<Integer> it = this.zahlen.iterator();
  while (it.hasNext()) {
    io.ausgeben("\t" + it.next() + "\n");
  }
}

public void listeUntereinanderAusgebenAlt() {
  EinUndAusgabe io = new EinUndAusgabe();
  int zaehler = 0;
  while (zaehler < this.zahlen.size()) {
    io.ausgeben("\t" + this.zahlen.get(zaehler) + "\n");
    zaehler = zaehler + 1;
  }
}
```

Analyse vom Iterator



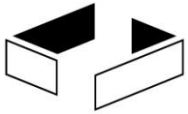
- sehr einfacher Durchlauf durch die Liste
- (meist) etwas schnellerer Durchlauf durch die Liste
- sicherer Durchlauf, da man nicht einfach auf nicht existente Elemente zugreifen kann

- es ist allerdings nicht möglich, die Nummer (Position) des betrachteten Elements auszugeben (mit mitlaufender Zählvariable schon)

Video

for

for-Schleife



- häufig sehr ähnliche Struktur einer Schleife

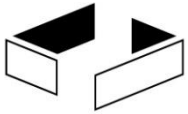
```
int zaehler = 0;
while (zaehler …) {
  // do something
  zaehler = zaehler + 1;
}
```

- für Schleifen mit festem Start- und Endwert gibt es Variante

```
for(int zaehler = 0; zaehler…; zaehler = zaehler+1)
```

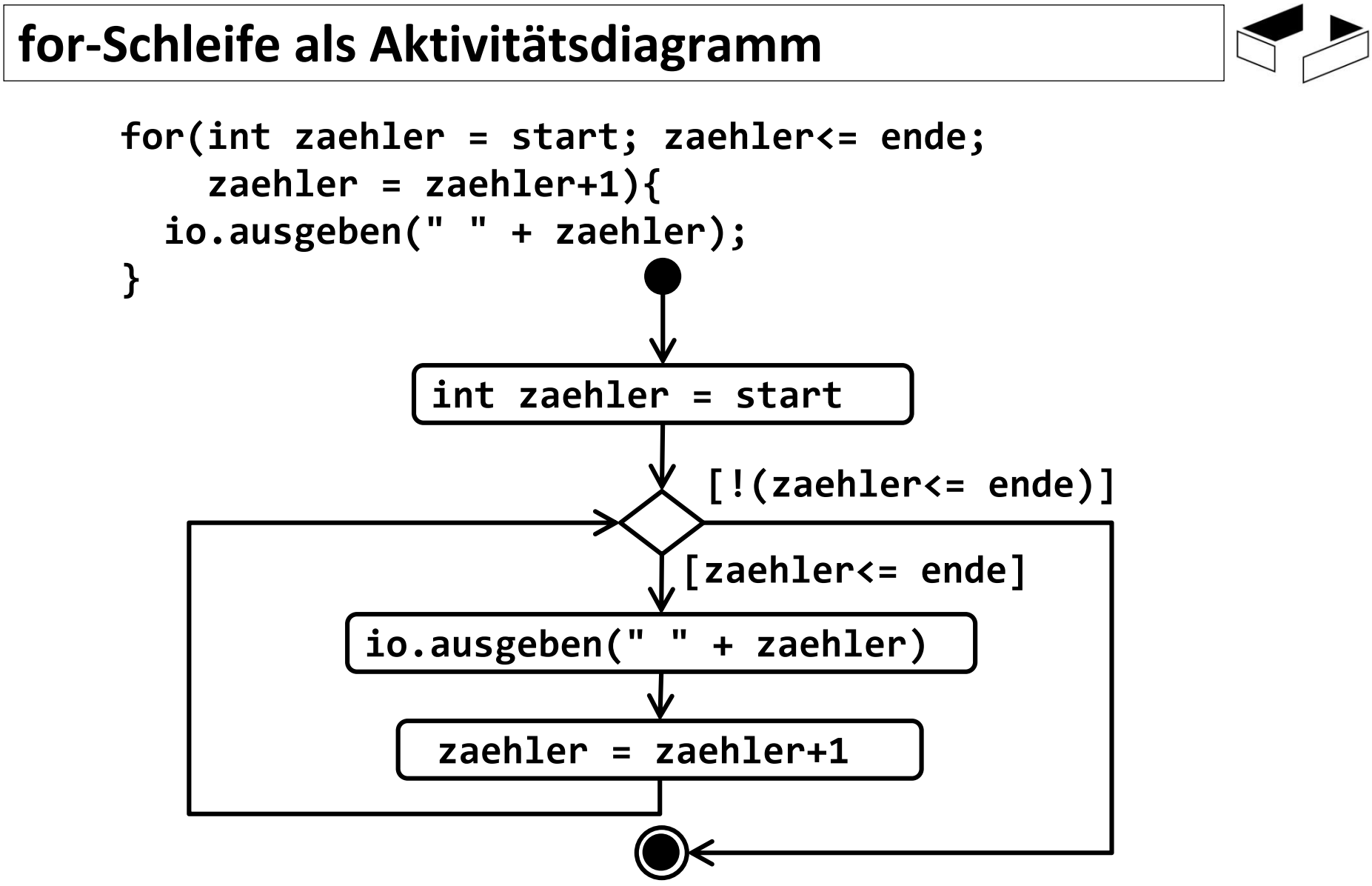
```
for( <Startanweisung>; <Abbruchbedingung>;
                       <AnweisungNachTeilprogramm>) {
    // do something
}
```

Schleifenvergleich



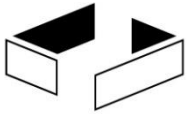
```
public void zeigeZahlen(int start, int ende) {
  int zaehler = start;
  EinUndAusgabe io = new EinUndAusgabe();
  while (zaehler <= ende) {
    io.ausgeben(" " + zaehler);
    zaehler = zaehler + 1;
  }
}

public void zeigeZahlenFor(int start, int ende) {
  EinUndAusgabe io = new EinUndAusgabe();
  for(int zaehler = start; zaehler<= ende; zaehler = zaehler+1){
    io.ausgeben(" " + zaehler);
  }
}
```

for-Schleife als Aktivitätsdiagramm



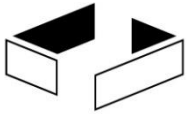
```
for(int zaehler = start; zaehler<= ende;
    zaehler = zaehler+1){
  io.ausgeben(" " + zaehler);
}
```

for-Schleife mit Iterator



```
public void listeUntereinanderAusgebenIterator() {
  EinUndAusgabe io = new EinUndAusgabe();
  Iterator<Integer> it = this.zahlen.iterator();
  while (it.hasNext()) {
    io.ausgeben("\t" + it.next() + "\n");
  }
}

public void listeUntereinanderAusgebenIteratorFor() {
  EinUndAusgabe io = new EinUndAusgabe();
  for(Iterator<Integer> it = this.zahlen.iterator();
        it.hasNext();) {    // Semikolon und dann leer
    io.ausgeben("\t" + it.next() + "\n");
  }
}
```

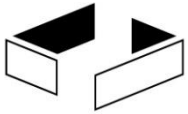
Regeln für for-Schleifen



```
for( <Startanweisung>; <Abbruchbedingung>;
                       <AnweisungNachTeilprogramm>)
```

- erfahrene entwickelnde Personen erwartet beim Lesen einer For-Schleife, dass beim Start der Schleife bekannt ist, wie oft sie durchlaufen wird (Abbruchbedingung nicht im Schleifenrumpf verändern)
- <Startanweisung> kann leer gelassen werden
- <AnweisungNachTeilprogramm> kann weggelassen werden

for-Schleife mit impliziten Iterator



- Listendurchlauf sieht häufig so aus:

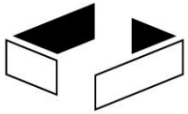
```
for(Iterator<Integer> it = liste.iterator(); it.hasNext();)
```

- Ab Java 5: leichte Abkürzung als „forEach“-Schleife

```
public void listeUntereinanderAusgebenIteratorForEach() {
  EinUndAusgabe io = new EinUndAusgabe();
  for(int el: this.zahlen) {
    io.ausgeben("\t" + el + "\n");
  }
}
```

- Achtung: Nutzung des Iterator-Objekts ist wichtige, sich in vielen Programmierbereichen wiederholende, Idee zur Abarbeitung von Sammlungen (Listen, Mengen, Einlesen von Dateien, Einlesen von Daten aus Datenbank …)

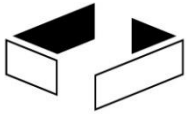
Einfaches Durchlaufen von Sammlungen



```
for( int el: this.zahlen) {
  io.ausgeben("\t" + el + "\n");
}
```

- Sammlung , die durchlaufen werden soll
- Typ der Elemente der Sammlung
- lokale Variable vom Typ der Elemente der Sammlung; Variable nimmt nacheinander alle Werte der Sammlung an
- Programm, dass für alle Elemente der Sammlung ausgeführt wird

Zahlenanalyse –Sortieren (1/8)



Video

Sortieren Sie mit einer Methode sortieren die Zahlenliste

1. Kläre den Typ des Ergebnisses

   hier: nur innere Veränderung des Objekts, also void

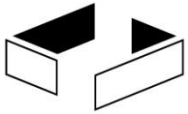
2. Kläre die Parameter der Methode

   hier: nur bearbeitetes Objekt selbst betroffen, also leer

   ```
   public void sortieren()
   ```

3. veranschauliche durch Beispiele, ob klar ist, wann welches Ergebnis herauskommt

Zahlenanalyse – Sortieren (2/8)



z1=

name	text1				
zahlen	9	1	7	1	0

`z1.sortieren()`

z1=

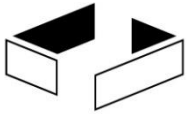
name	text1				
zahlen	0	1	1	7	9

4. Überlege Lösungsverfahren (Algorithmus)

Ansatz: Nutze neue Ergebnisliste, suche in `zahlen` kleinstes Element, lösche es in `zahlen` und hänge es an Ergebnisliste an. Mach das, bis `zahlen` leer ist und ersetze dann `zahlen` durch Ergebnisliste

5. Denke über Visualisieren nach oder/und probiere auf Papier

Zahlenanalyse – sortieren (3/8)



	0	1	2	3	4
`zahlen`	9	1	7	1	0

`ergebnis`

finde Minimum, gefunden an Position 4, schreibe Wert von Position 4 ans Ende von Ergebnis, lösche Wert an Position 4 in `zahlen`

	0	1	2	3
`zahlen`	9	1	7	1

`ergebnis`

0

finde Minimum, gefunden an Position 1, schreibe Wert von Position 1 ans Ende von Ergebnis, lösche Wert an Position 1 in `zahlen`

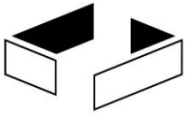
	0	1	2
`zahlen`	9	7	1

`ergebnis`

0	1

finde Minimum, gefunden an Position 2, schreibe Wert von Position 2 ans Ende von Ergebnis, lösche Wert an Position 2 in `zahlen`

Zahlenanalyse – sortieren (4/8)



	0	1
zahlen	9	7

ergebnis	0	1	1

finde Minimum, gefunden an Position 1, schreibe Wert von Position 1 ans Ende von Ergebnis, lösche Wert an Position 1 in `zahlen`

	0
zahlen	9

ergebnis	0	1	1	7

finde Minimum, gefunden an Position 0, schreibe Wert von Position 0 ans Ende von Ergebnis, lösche Wert an Position 0 in `zahlen`

zahlen

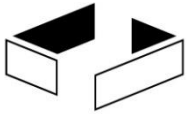
ergebnis	0	1	1	7	9

erkenne, dass zahlen leer ist und setze zahlen auf Ergebnis

zahlen	0	1	1	7	9

ergebnis	0	1	1	7	9

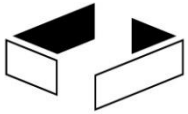
Zahlenanalyse – sortieren (5/8)



6. Programmieren

```
public void sortieren(){
   ArrayList<Integer> ergebnis = new ArrayList<Integer>();
   while (this.zahlen.size() > 0) {
     int minimum = this.zahlen.get(0);
     int pos = 0;
     for (int zaehler = 0; zaehler < this.zahlen.size()
          ; zaehler++){
       if (minimum > this.zahlen.get(zaehler)){
         minimum = this.zahlen.get(zaehler);
         pos = zaehler;
       }
     }
     ergebnis.add(minimum);
     this.zahlen.remove(pos);
   }
   this.zahlen = ergebnis;
 }
```

Zahlenanalyse – sortieren (6/8)



7. Testen

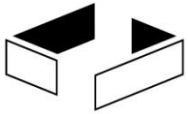
```
public void sortierbeispiele(){
   Zahlenanalyse z = new Zahlenanalyse();
   z.hinzufuegen(9);
   z.hinzufuegen(1);
   z.hinzufuegen(7);
   z.hinzufuegen(1);
   z.hinzufuegen(0);
   EinUndAusgabe io = new EinUndAusgabe();
   io.ausgeben("1: "+z+"\n");
   z.sortieren();
   io.ausgeben("2: "+z+"\n");
   z.hinzufuegen(5);
   z.sortieren();
   io.ausgeben("3: "+z+"\n");
   z.hinzufuegen(11);
   z.sortieren();
   io.ausgeben("4: "+z+"\n");
   z.hinzufuegen(-1);
   z.sortieren();
   io.ausgeben("5: "+z+"\n");
}
```

BlueJ: Terminal Window - BeispielZa...

Options

```
1: default: [9, 1, 7, 1, 0]
2: default: [0, 1, 1, 7, 9]
3: default: [0, 1, 1, 5, 7, 9]
4: default: [0, 1, 1, 5, 7, 9, 11]
5: default: [-1, 0, 1, 1, 5, 7, 9, 11]
```

Zahlenanalyse – sortieren (7/8)



8. Fehler finden, Extremfälle beachten

```
public void extremfaelle(){
        Zahlenanalyse z = new Zahlenanalyse();
        EinUndAusgabe io = new EinUndAusgabe();
        io.ausgeben("1: "+ z +"\n");
        z.sortieren();
        io.ausgeben("2: "+ z +"\n");
        z.setZahlen(null);
        io.ausgeben("3: "+ z +"\n");
        z.sortieren();
        io.ausgeben("4: "+ z + "\n");
}
```

BlueJ: Terminal Window - BeispielZahlenlistenSchleife2

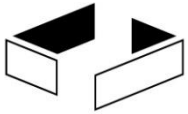
Options

```
1: default: []
2: default: []
3: default: null
```

Can only enter input while your programming is running

```
java.lang.NullPointerException
        at Zahlenanalyse.sortieren(Zahlenanalyse.java:256)
```

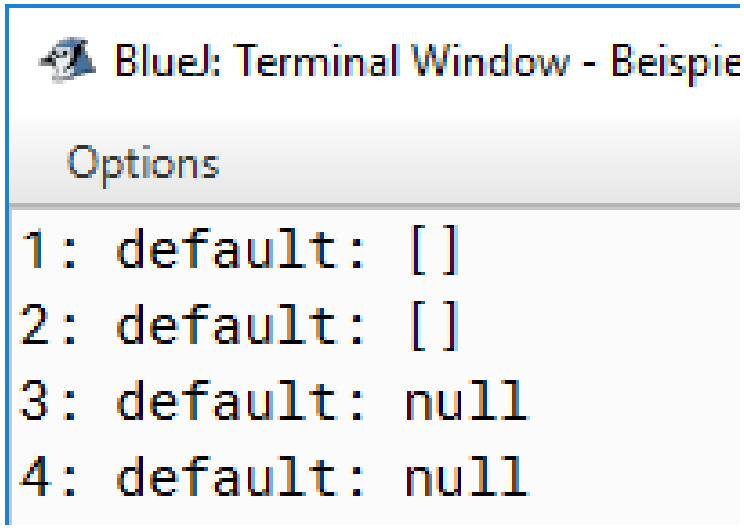
Zahlenanalyse – sortieren (8/8)



9. Korrigieren

```
public void sortieren(){
        if (this.zahlen == null) {
            return;
        }
        // wie vorher
```

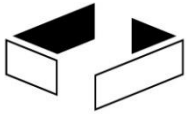
10. Testen, solange 8.-10. bis fertig



Video

Klassenvariablen und Klassenmethoden

Erinnerung: zentrale Idee Objektorientierung



Klasse

```
public class Studierend {
  private int mat;
  private String name;
  …
```

new … new … new …

unabhängige Objekte

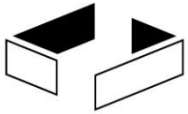
erzeugt über

Konstruktor

s1:Studierend	s2:Studierend	s3:Studierend
mat = 42	mat = 43	mat = 44
name =„X“	name =„U“	name =„Ω“
…	…	…

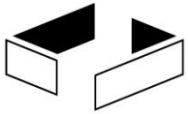
Objekte haben nichts miteinander zu tun; können natürlich Methoden in anderen Objekten aufrufen, z. B. s1.equals(s2)

Objekte und Klassen



- Bisher: Wir programmieren das Verhalten von Objekten
- Objekte können über Objektmethoden
  - verändert werden
  - Ergebnisse, d.h. neue Objekte, berechnen
- Klassen beschreiben nur Objekte und erlauben über Konstruktoren die Erstellung beliebig vieler Objekte
- außer Konstruktoren bieten Klassen selbst bisher nichts an
- bisher nur recht aufwändig machbar: Vergabe eindeutiger Matrikelnummern
- "letzte vergebene Matrikelnummer" ist keine Objekt-Eigenschaft, sondern Eigenschaft der Klasse

Klassenvariablen

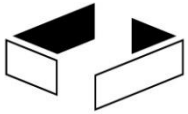


- Klassenvariable ist Eigenschaft einer Klasse, z. B. ein Zähler wieviele Objekte existieren, Schlüsselwort static

```
private static int letzteNummer;
```

- Klassenvariablen gehören zur Klasse und können damit ohne Objekte existieren
- Klassenvariablen können in Objektmethoden genutzt werden
- d. h. Objektvariablen kann der Wert von Klassenvariablen zugeordnet werden
- einer Klassenvariable kann nicht ein zu einer Objektvariablen gehöriges Objekt zugewiesen werden (da Objekte gelöscht werden können)

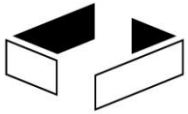
Klassenmethoden



- Klassenmethoden sind Methoden, die nur zur Klasse gehören und auch ohne Objekte ausgeführt werden können
- In Klassenmethoden kann auf Klassenvariablen lesend und schreibend, andere Klassenmethoden und Parameter (auch Objekte) zugegriffen werden
- Um deutlich zu machen, dass eine Klassenmethode aufgerufen wird, kann der Name der Klasse vor dem Methodennamen mit einem Punkt getrennt stehen

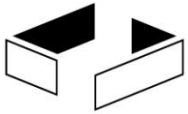
  <Klassenname>.<Klassenmethodenname>()
- gleiche Schreibweise auch bei Klassenvariable möglich
- wird Klassenmethode in anderer Klasse oder Objekt einer anderen Klasse genutzt, muss diese Schreibweise genutzt werden

Nutzung von Klassenvariablen und -methoden (1/2)



```
public class Studierend{
    private static int letzteNummer = 99999;

    protected String vorname ="Eva";
    protected String nachname ="Mustermann";
    ...

    public Studierend(){
      Studierend.letzteNummer = Studierend.letzteNummer + 1;
      this.matrikelnummer = Studierend.letzteNummer;
    }

    public static int leseLetzteNummer(){
      return Studierend.letzteNummer;
    }

    public static void schreibeLetzteNummer(int m){
      Studierend.letzteNummer = m;
    }
```

Nutzung von Klassenvariablen und -methoden (2/2)



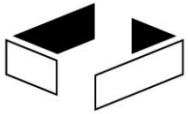
```
// in Klasse Studierendspielerei
  public void klassenvariableAnalysieren() {
    EinUndAusgabe io = new EinUndAusgabe();
    Studierend s1 = new Studierend();
    Studierend s2 = new Studierend();
    io.ausgeben("s1: " + s1 + "\n");
    io.ausgeben("s2: " + s2 + "\n");
    io.ausgeben("lN: " + Studierend.leseLetzteNummer() + "\n");
    Studierend.schreibeLetzteNummer(42);
    Studierend s3 = new Studierend();
    io.ausgeben("s3: " + s3 + "\n");
  }
```

BlueJ: Konsole - BeispielStudierendenKlassenvariable

Optionen

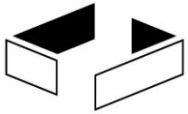
```
s1: Eva Mustermann (100000):ITI
s2: Eva Mustermann (100001):ITI
lN: 100001
s3: Eva Mustermann (43):ITI
```

Nutzung einer Klassenmethode ohne Objekt



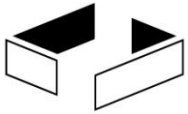
```
Studierend.leseLetzteNummer()
    43   (int)
```

Wann sind Klassenmethoden sinnvoll?



- Klassenmethoden werden dann eingesetzt, wenn ausschließlich Klassenvariablen und andere Klassenmethoden betroffen sind
- Klassenmethoden sind typischerweise Hilfsmethoden, die mit dem Konzept der Objektorientierung unmittelbar wenig zu tun haben
- Grundsätzlich muss ein Entwickler in Objekten und ihrer Bearbeitung mit Objektmethoden denken
- Erst wenn man feststellt, dass keine Objekte benötigt werden, kann man über die Erstellung von Klassenmethoden nachdenken
- typisches Beispiel für Klasse mit ausschließlich Klassenmethoden ist java.lang.Math

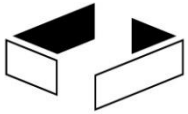
Ausschnitt der Klassenmethoden von Math



Modifier and Type	Method and Description
static double	abs(double a) Returns the absolute value of a double value.
static double	cos(double a) Returns the trigonometric cosine of an angle.
static double	exp(double a) Returns Euler's number *e* raised to the power of a double value.
static double	log(double a) Returns the natural logarithm (base *e*) of a double value.
static double	pow(double a, double b) Returns the value of the first argument raised to the power of the second argument.
static double	random() Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
static long	round(double a) Returns the closest long to the argument, with ties rounding up.
static double	sqrt(double a) Returns the correctly rounded positive square root of a double value.
static double	tan(double a) Returns the trigonometric tangent of an angle.

Quelle: https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Math.html

Schlechte und gute Ideen zu Klassenmethoden



- gute Idee: EinUndAusgabe-Klasse könnte nur aus Klassenmethoden bestehen

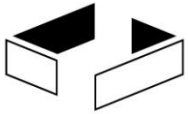
- schlechte Idee: wahrscheinlich brauche ich nur ein Objekt der Klasse, also nutze ich nur Klassenvariablen und Methoden
  - nein !!! ausschließliche Nutzung nur für Klasseneigenschaften und Hilfsmethoden
  - wenn nur ein Objekt, dann gibt es bessere, objektorientiertere Wege

Beispiel

Video

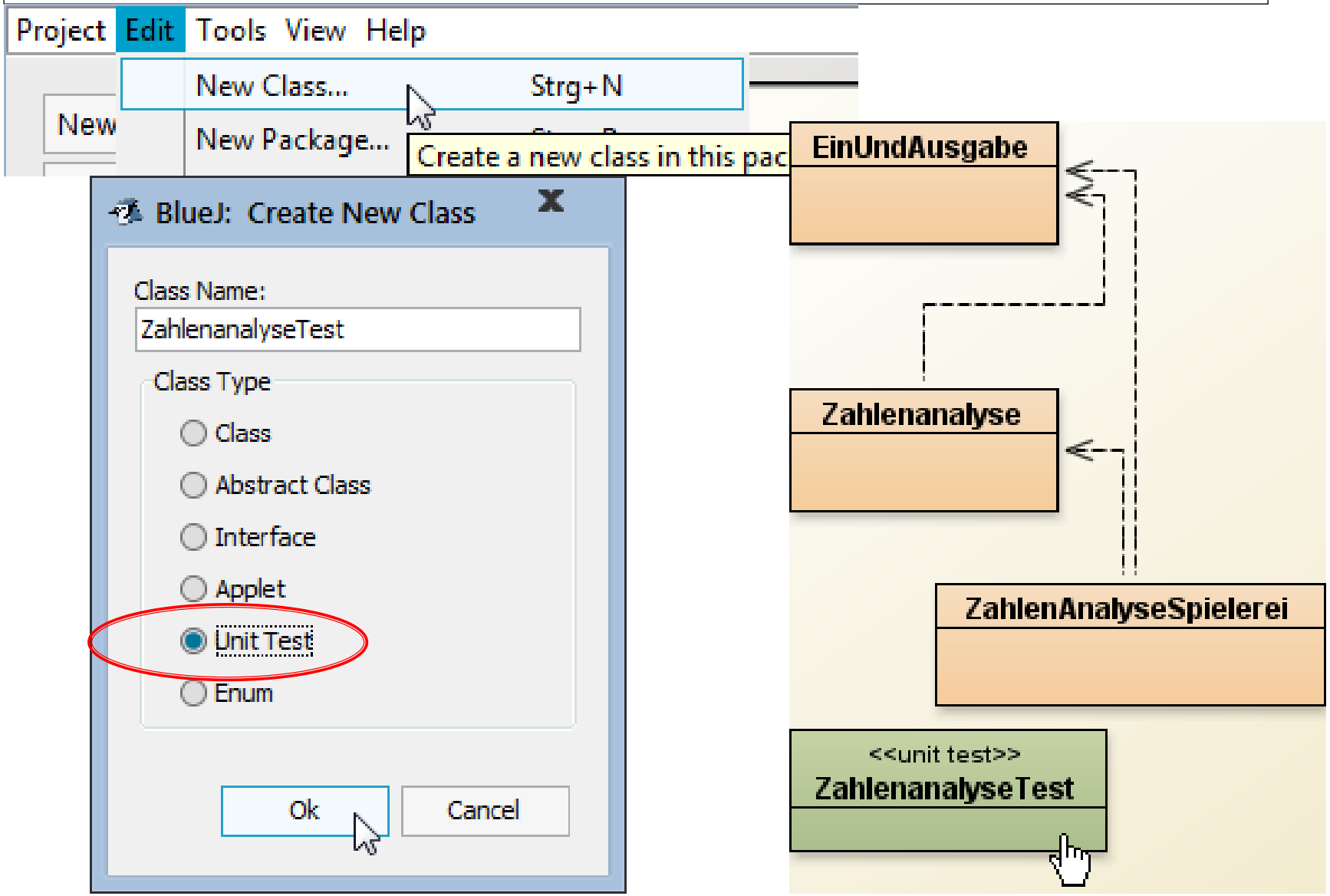
Unit Test

Ausprobieren oder systematisch Testen

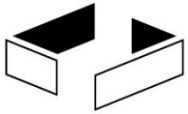


- bisher werden Methoden zum Ausprobieren direkt in BlueJ aufgerufen und das Ergebnis manuell überprüft
- um Schritte zusammenzufassen, wurden Hilfsklassen wie **`ZahlenAnalyseSpielerei`** geschrieben
- dieser Ansatz wird durch die Nutzung spezieller Testklassen vereinfacht:
  - Nutzung von JUnit (hier Version 5)
  - Tests werden in neuer Klasse (ähnlich zu Spielerei) in Java geschrieben
  - Man kann für Tests gemeinsame Ausgangssituation schaffen
  - Jeder Test läuft unabhängig von anderen Tests
  - es gibt spezielle Zusicherungsmethoden zum Überprüfen; dies sind Klassenmethoden der Klasse Assertions

Testerstellung in BlueJ

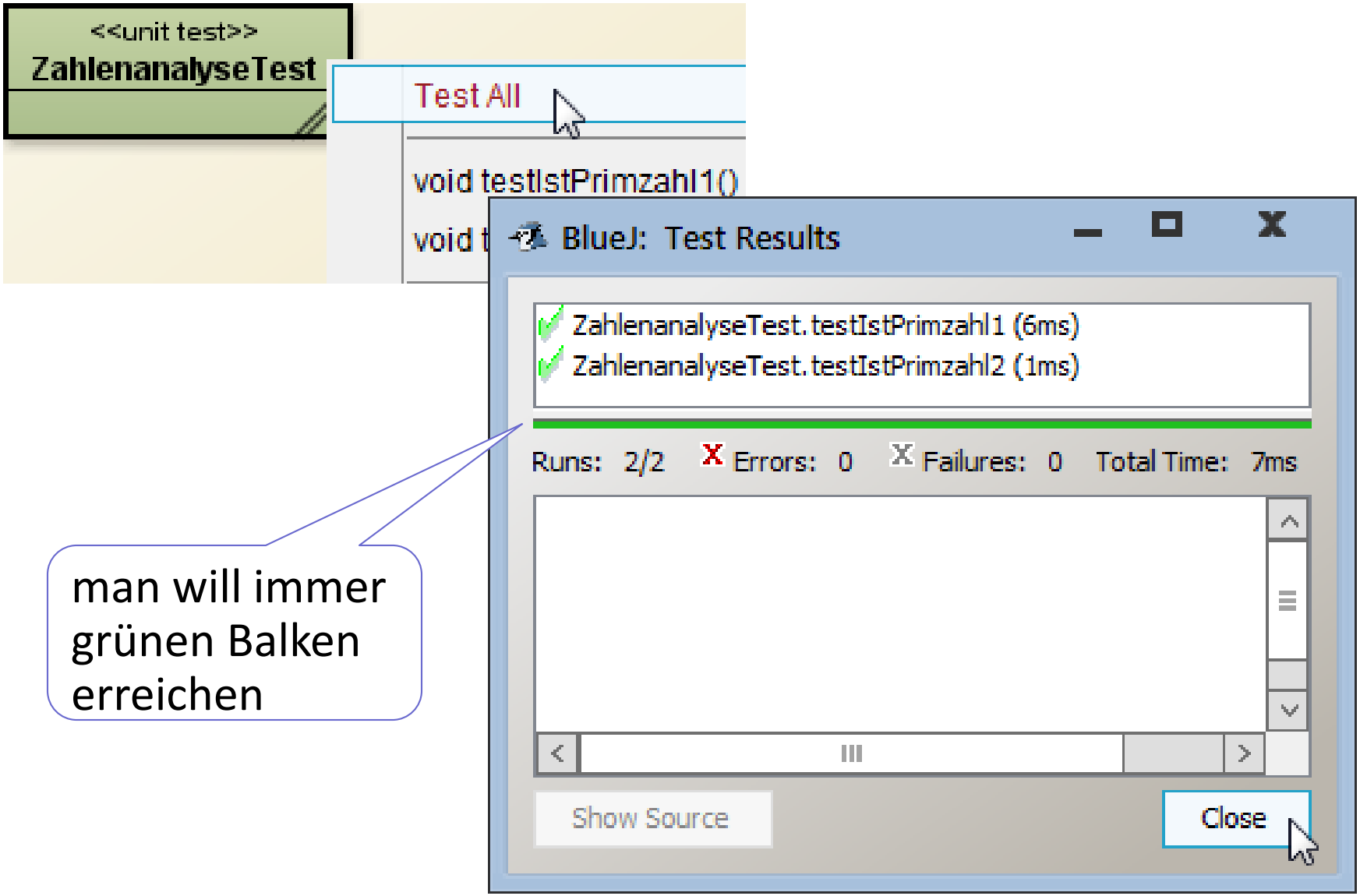


Erste Testfälle (1/2)



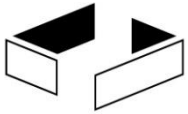
```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class ZahlenanalyseTest{

  @Test
  public void testIstPrimzahl1() {
    Zahlenanalyse zana = new Zahlenanalyse(null);
    boolean nicht42 = zana.istPrimzahl(42);
    Assertions.assertTrue(!nicht42, "42 ist Primzahl");
  }

  @Test
  public void testIstPrimzahl2() {
    Zahlenanalyse zana = new Zahlenanalyse(null);
    boolean doch41 = zana.istPrimzahl(41);
    Assertions.assertTrue(doch41, "41 ist keine Primzahl");
  }
}
```

Erste Testfälle (2/2)



man will immer grünen Balken erreichen

Aufbau der Testklasse



Klassenname beliebig, üblich: Name der zu testenden Klasse mit "Test" am Ende

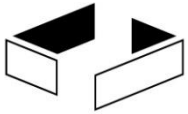
jeder Test in eigener Methode, sollte mit "public void test…" beginnen; Testname sollte sich auf getestete Methode beziehen

diese Annotation (?!?) sorgt für die Ausführung als Test[genauer später]

```
public class ZahlenanalyseTest {
  @Test
  public void testIstPrimzahl1() {
    Zahlenanalyse zana = new Zahlenanalyse(null);
    boolean nicht42 = zana.istPrimzahl(42);
    Assertions.assertTrue(!nicht42, "42 ist Primzahl");
  }
```

es gibt verschiedene Überprüfungs(klassen)methoden (assertTrue reicht erstmal); erster Parameter ist Meldung im Fehlerfall, zweiter Parameter ist Boolescher Ausdruck (erster Parameter optional)

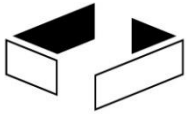
Test-Fixture (Test-Ausgangskonfiguration)



- Testfall sieht in der Regel so aus, dass eine bestimmte Konfiguration von Objekten aufgebaut wird, gegen die der Test läuft
- Menge von Testobjekten wird als Test-Fixture bezeichnet
- Damit fehlerhafte Testfälle nicht andere Testfälle beeinflussen können, wird die Test-Fixture für jeden Testfall neu initialisiert

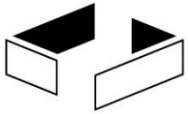
- In der mit **`@BeforeEach`** annotierten Methode **`public void setUp()`** werden Objektvariablen initialisiert
- In der mit **`@AfterEach`** annotierten Methode **`public void tearDown()`** werden wertvolle Testressourcen wie zum Beispiel Datenbank- oder Netzwerkverbindungen wieder freigegeben

Testklasse leicht umgeschrieben



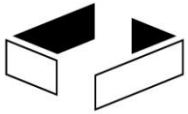
```
public class ZahlenanalyseTest {
  private Zahlenanalyse zana; // zu testendes Objekt
  @BeforeEach
  public void setUp() {
    zana = new Zahlenanalyse(null);
  }
  @Test
  public void testIstPrimzahl1() {
    boolean nicht42 = this.zana.istPrimzahl(42);
    Assertions.assertTrue(!nicht42, "42 ist Primzahl");
  }
  @Test
  public void testIstPrimzahl2() {
    boolean doch41 = this.zana.istPrimzahl(41);
    Assertions.assertTrue(doch41, "41 ist keine Primzahl");
  }
}
```

Prinzipieller Ablauf von JUnit



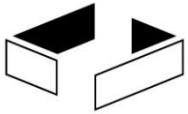
- Suche in der Testklasse alle Methoden, die mit @Test annotiert sind,
- führe für jede der gefundenen Methoden folgende Schritte aus:
  - führe setUp (mit @BeforeEach annotierte Methode) aus
  - führe den Test aus und protokolliere das Ergebnis
  - führe tearDown (mit @AfterEach annotierte Methode) aus
- im Beispiel: setUp(), testIstPrimzahl1(), setUp(), testIstPrimzahl2()
- sollte ein Test scheitern, wird dieser sofort beendet, die anderen Tests unabhängig davon ausgeführt
- Tests sollten nach Zusicherungen (assert…) keinen anderen Code haben (asserts am Ende)

Wie erstellt man Testfälle



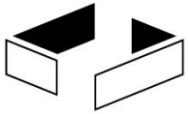
- Grundsätzlich liest man intensiv die Aufgabenstellung und beantwortet folgende Fragen:
- was sind die typischen Abläufe bzw. typischen Ergebnisse; für jeden der gefundenen Fälle wird ein Test geschrieben
- was sind die besonderen Randfälle, die auftreten können; für jeden Randfall wird ein getrennter Testfall geschrieben
- Beispiel: ein Parameter liste vom Typ ArrayList<Integer>
  - was passiert, wenn liste == null ist
  - was passiert bei einer leeren Liste
  - was passiert bei besonderen Listen, die z. B. nur ein Element, dies aber mehrfach enthalten
- [Kle19] S. Kleuker, Qualitätssicherung durch Softwaretests, 2. Auflage, Springer Vieweg, Wiesbaden, 2019

Weiteres Testbeispiel (1/4)



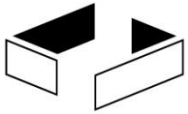
```
// Imports fehlen

public class ZahlenanalyseTest {

  private Zahlenanalyse zana42und23;
  private Zahlenanalyse zanaNull;
  private Zahlenanalyse zana23;

  @BeforeEach
  public void setUp() {
    ArrayList<Integer> l1 = new ArrayList<Integer>();
    l1.add(42);
    l1.add(23);
    this.zana42und23 = new Zahlenanalyse(l1);
    this.zanaNull = new Zahlenanalyse(null);
    ArrayList<Integer>l2 = new ArrayList<Integer>();
    l2.add(23);
    this.zana23 = new Zahlenanalyse(l2);
  }
```

Weiteres Testbeispiel (2/4)



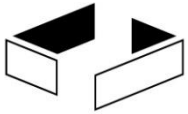
```
@Test
public void testZahlenMitDreiAmEnde1(){
  Zahlenanalyse tmp = this.zana42und23.zahlenMitDreiAmEnde();
  Assertions.assertTrue(tmp.getZahlen().size() == 1);
  Assertions.assertTrue(tmp.getZahlen().get(0) == 23);
}

@Test
public void testZahlenMitDreiAmEnde2(){
  Zahlenanalyse tmp = this.zanaNull.zahlenMitDreiAmEnde();
  Assertions.assertTrue(tmp.getZahlen().size() == 0);
}

@Test
public void testZahlenMitDreiAmEnde3(){
  Zahlenanalyse tmp = this.zana23.zahlenMitDreiAmEnde();
  Assertions.assertTrue(tmp.getZahlen().size() == 1);
  Assertions.assertTrue(tmp.getZahlen().get(0) == 23);
}
```

Weiteres Testbeispiel (3/4)



```
@Test
public void testgibtGleiche1(){
  boolean tmp = this.zana42und23.gibtGleiche(zanaNull);
  Assertions.assertTrue(!tmp);
}

@Test
public void testgibtGleiche2(){
  boolean tmp = this.zana42und23.gibtGleiche(zana23);
  Assertions.assertTrue(tmp);
}

@Test
public void testgibtGleiche3(){
  boolean tmp = this.zana23.gibtGleiche(zana42und23);
  Assertions.assertTrue(!tmp); //?!?
}
```

Weiteres Testbeispiel (4/4)



BlueJ: Test Results

- ZahlenanalyseTest.testZahlenMitDreiAmEnde1 (8ms)
- ZahlenanalyseTest.testZahlenMitDreiAmEnde2 (1ms)
- ZahlenanalyseTest.testZahlenMitDreiAmEnde3 (1ms)
- ZahlenanalyseTest.testgibtGleiche1 (3ms)
- ZahlenanalyseTest.testgibtGleiche2 (0ms)
- ZahlenanalyseTest.testgibtGleiche3 (1ms)
- ZahlenanalyseTest.testIstPrimzahl1 (0ms)
- ZahlenanalyseTest.testIstPrimzahl2 (0ms)

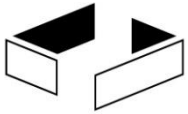
ein Test zeigt, dass null-Referenzen nicht berücksichtigt werden, anderer Test zeigt falschen Testfall

Runs: 8/8 Errors: 1 Failures: 1 Total Time: 14ms

```
null
---
junit.framework.AssertionFailedError: null
        at ZahlenanalyseTest.testgibtGleiche3(ZahlenanalyseTest.java:50)
        at org.junit.internal.runners.JUnit38ClassRunner.run(JUnit38Clas
```

Show Source Close

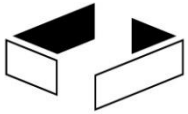
Systematische Entwicklung



- Neben echten Denkfehlern befinden sich häufig viele Flüchtigkeitsfehler in Programmen
- Sinnvoll: Inkrementelle Entwicklung, d. h. nachdem eine Methode implementiert wurde, werden sofort zugehörige Testfälle geschrieben
- Variante ist "Test first"; nachdem man die Aufgabenstellung verstanden hat, programmiert man zunächst die Testfälle und schreibt dann schrittweise immer mehr Programmteile, so dass immer mehr Testfälle keine Fehler melden

- man beachte, dass Entwickler*innen betriebsblind gegenüber ihren eigenen Fehlern werden

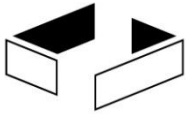
Hinweise zu JUnit



- JUnit kann auch zur Organisation von Tests (Gruppierungen) genutzt werden
- JUnit oder TestNG Standard bei den meisten Entwicklungen
- JUnit kann so nicht (nicht einfach) Eingaben in der Konsole simulieren
- es gibt viele weitere Testwerkzeuge, einige auf Basis von JUnit, die viele andere Tests automatisiert durchführen können

- Man muss Programme aber auch in Richtung Testbarkeit entwickeln:
  - get- und set-Methoden für alle Objektvariablen
  - evtl. private-Methoden public machen

Systematische Objektbearbeitung (1/2)

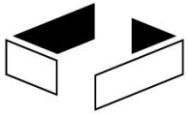


- hat ein Objekt eine Collection von Objekten (z. B. ArrayList) in einer Objektvariablen, so gilt folgende Regel zur Bearbeitung:
- grundsätzlich stellt das Objekt selbst Methoden zur Verfügung, um die Collection zu bearbeiten, d. h. Elemente zu suchen
- also schlecht:

```
Zahlenanalyse tmp = zanaNull.zahlenMitDreiAmEnde();
if(tmp.getZahlen().size() == 0)) ...
```

- besser: Zahlenanalyse bietet Methoden wie:
  - berechneListengroesse()
  - hinzufuegen(int)

Systematische Objektbearbeitung (2/2)



- Veranschaulichung als schlechte Idee: Organe herausoperieren, einschicken, bearbeiten, zurückschicken und wieder einsetzen

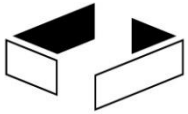
- also keine getZahlen()-Methode? diskutabel
- wir benötigen Methode zum effizienten Testen
- in Tests darf gegen gewisse Regeln verstoßen werden (muss aber nicht)
- Klassen müssen ab und zu Methoden enthalten, die das Testen erleichtern bzw. erst ermöglichen

Beispiel

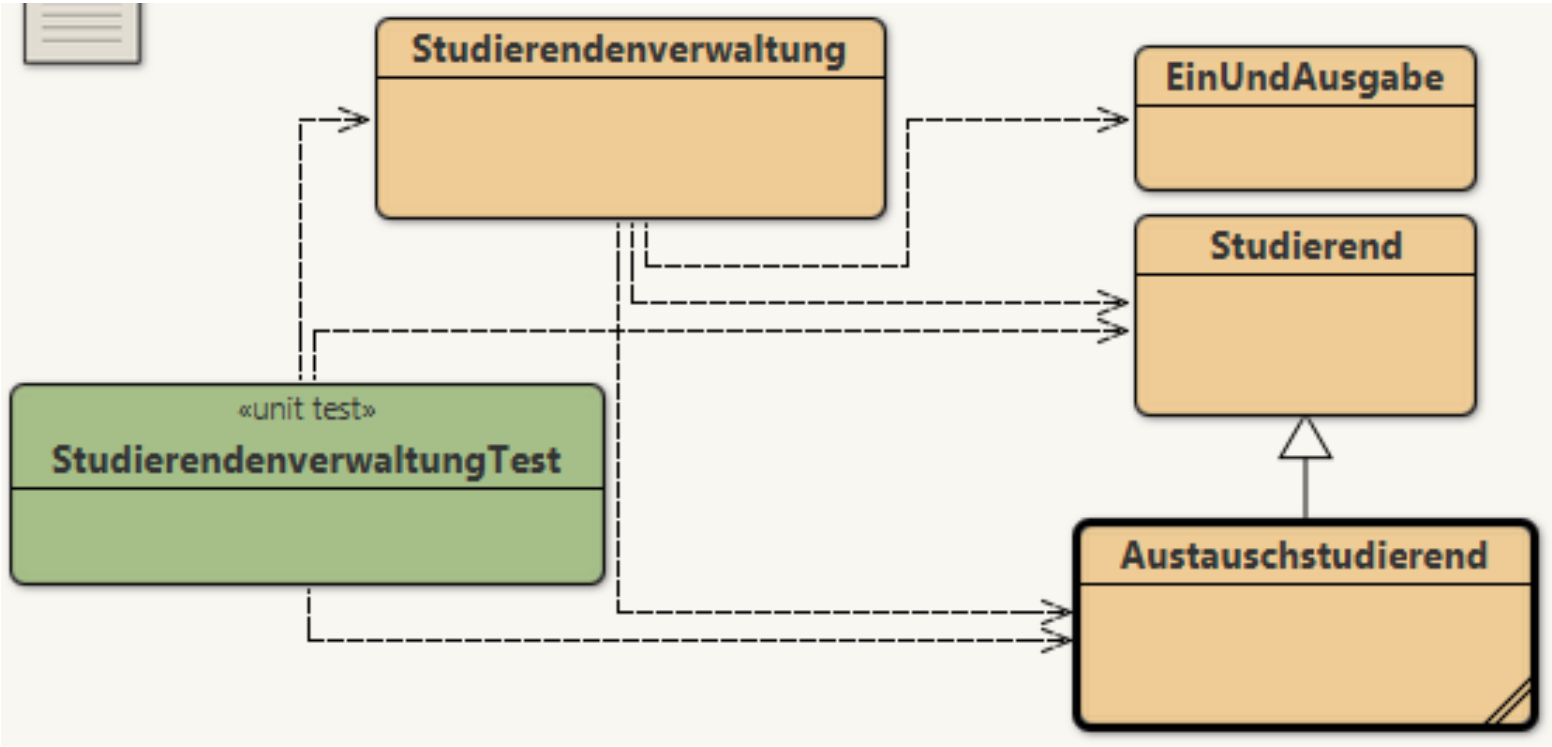
Video

Vererbung

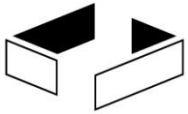
Erweiterung: Austauschstudierende - Vererbung



- objektorientierte Antwort heißt: Erweiterung einer existierenden Klasse (meist Vererbung genannt)
- Der Zugriff auf Teile der erweiterten Klasse erfolgt mit super statt this

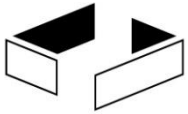


Vererbung mit Konstruktornutzung



```
public class AustauschStudierend extends Studierend{
  private String land = null;

  public AustauschStudierend(String land, String vorname
        , String nachname, int geburtsjahr
        , String studiengang, int matrikelnummer){
    super(vorname, nachname, geburtsjahr
          , studiengang, matrikelnummer);
    this.land = land;
  }

  // geht auch ohne super, da Studierend() existiert,besser mit
  public AustauschStudierend(){
  }

  public String getLand() {
    return this.land;
  }

  public void setLand(String land) {
    this.land = land;
  }
}
```

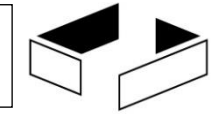
Vererbung und Konstruktoren genauer

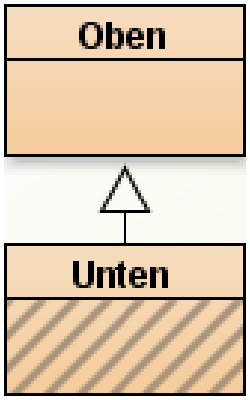


- Jeder Konstruktor in einer erbenden (= erweiternden) Klasse ruft Konstruktor der beerbten (= erweiterten) Klasse auf
- dieser Konstruktoraufruf muss die erste Zeile in Konstruktoren der erbenden Klasse sein
- wird kein Konstruktoraufruf in der erbenden Klasse angegeben, wird automatisch der Default-Konstruktor (= parameterloser Konstruktor) der beerbten Klasse zuerst aufgerufen
- hat dann beerbte Klasse keinen Default-Konstruktor wird ein Fehler ausgegeben
- Grundregel der sauberen Programmierung: In Konstruktoren von erbenden Klassen wird als erstes immer ein Konstruktor der Oberklasse aufgerufen; also:

```
public AustauschStudierend(){
    super();
}
```

Vererbung, was geht nicht



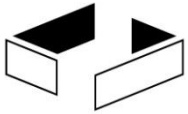



```java
public class Oben{
  private Integer wert;

  public Oben(Integer wert){
      this.wert = wert;
  }
}
```

```java
public class Unten extends Oben{
  private String name;

  public Unten(Integer wert, String name){
      super(wert);
      this.name = name;
  }

  public Unten(String name){
      this.name = name;
  }
}
```

cannot find symbol - constructor Oben()

Vererbung – Nutzung von Gemeinsamkeiten



- Grundsätzlich kann ein Objekt der erbenden Klasse an jeder Stelle genutzt werden, an der auch ein Objekt der beerbten Klasse stehen kann
- damit sind alle public-Methoden aus der beerbten Klasse in der erbenden Klasse nutzbar!
- Zuweisungen an Variablen vom Typ der beerbten Klasse sind erlaubt

```
Studierend s = new AustauschStudierend();
AustauschStudierend aus = new Studierend(); // geht nie !!!
```

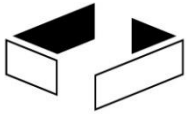
- Sammlungen mit Objekten des Typs der beerbten Klasse können Objekte der erbenden Klasse aufnehmen

```
ArrayList<Studierend> studis = new ArrayList<Studierend>();
studis.add(new AustauschStudierend());
```

Video

Überschreiben

Vererbung: Möglichkeiten zur Individualisierung



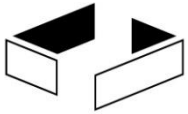
- Ab und zu möchte man, dass sich eine Methode bei einer erbenden Klasse etwas anders als bei der beerbten Klasse verhält; hierzu können Methoden überschrieben werden
- aktuell wird für AustauschStudierend und Studierend die gleiche toString-Methode genutzt

```
public void austauschStudierendAusgeben(){ // Klasse Analyse
  AustauschStudierend aus = new AustauschStudierend(
                            "USA","Mo","Jo",1989,"ITI",424243);
  EinUndAusgabe io = new EinUndAusgabe();
  io.ausgeben("Ausgabe: "+aus);
}
```

```
Ausgabe: Mo Jo (424243):ITI
```

- Nun soll bei AustauschStudierend nur Vorname, Name, Land und Studiengang ausgegeben werden

Methoden überschreiben

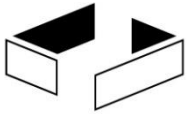


```
@Override
public String toString(){
  return super.getVorname() + " " + super.getNachname()
      +" (" + this.land +"): " + super.getStudiengang();
}
```

```
Ausgabe: Mo Jo (USA):ITI
```

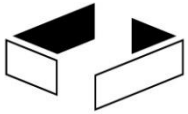
- Zugriff auf Methoden der beerbten Klasse über super.<Methode>

zu @Override



- @Override ist eine Annotation (Erinnerung an @Test)
- Annotationen werden zur deklarativen Programmierung genutzt; d. h. es wird beschrieben, was gemacht werden soll, aber nicht wie
- hier: Der Compiler soll prüfen, dass eine Methode der beerbten Klasse überschrieben wird
- Annotationen können beliebige Sprachkonstrukte (z. B. Klassen, Objektvariablen, Methoden, lokale Variablen) annotieren
- Annotationen spielen in C# und Java bei fortgeschrittenen Programmierkonzepten häufig eine zentrale Rolle; wird in der Anfängerveranstaltung aber nicht betrachtet

Direkter Zugriff auf geerbte Objektvariablen



- generell können mit get- und set-Methoden die Objektvariablen der beerbten Klasse bearbeitet werden
- einfacher wird es, wenn man Sichtbarkeit protected nutzt

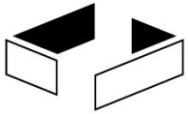
```
public class Studierend{
    protected String vorname = "Eva";
    protected String nachname = "Mustermann";
    protected int geburtsjahr = 1990;
    protected String studiengang = "IMI";
    protected int matrikelnummer = 232323;
```

- in AustauschStudierend:

```
    @Override
    public String toString(){
      return super.vorname + " " + super.nachname + " ("
                 + this.land + "): " + super.studiengang;
    }
```

- direkter Zugriff auf Objektvariablen der beerbten Klasse durch super.<Objektvariable>

Welche Ausgabemethode wird genutzt?



- Methode in Klasse Analyse

```
public void wasWirdGezeigt(){
  ArrayList<Studierend> list
      = new ArrayList<Studierend>();
  list.add(new AustauschStudierend("USA"
                                , "Mo","Jo",1989,"ITI",424243));
  list.add(new Studierend("Mo","Jo",1989,"ITI",424243));
  EinUndAusgabe io = new EinUndAusgabe();
  for(Studierend s:list){
    io.ausgeben(s + "\n");
  }
}
```

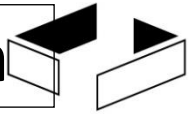
```
Mo Jo (USA): ITI
Mo Jo (424243):ITI
```

Beispiel

Video

statische Polymorphie

Erinnerung: statische Polymorphie bei Konstruktoren

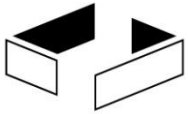


- statische Polymorphie: Vielgestaltigkeit, konkret mehrere Konstruktoren angebbar
- „statisch“ da bei der Übersetzung des Programms (Compile-Zeit) klar ist, welcher Konstruktor genutzt wird

```
public class Datum{
    private int tag;
    private int monat;
    private int jahr;
    public Datum (){ …
    public Datum (int anInt1){ …
    public Datum (int anInt1, int anInt2){ …
    public Datum (int anInt1, int anInt2, int anInt3){ …
}
```

- Erinnerung: Konstruktoren müssen sich unterscheiden: Anzahl der Parameter oder/und unterschiedliche Parametertypen an gleichen Positionen

Überladen von Methoden - statische Polymorphie

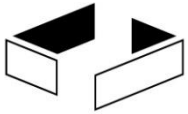


- Java bietet die Möglichkeit, Methoden zu überladen, d.h.
    - gleicher Methodenname
  - aber
    - unterschiedliche Parameteranzahl bzw. Parametertypen

- Rückgabewerte spielen keine Rolle bei der Methodenzuordnung

- generelles Prinzip wie bei Konstruktoren

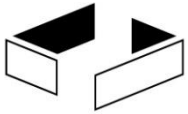
Auswahl bei überladenen Methoden



- Ist eine Zuordnung der aufrufenden Methode eindeutig, so wird die Methode ausgewählt, welche gleiche Parameteranzahl mit den passenden Parametertypen hat
- Ist die Zuordnung eines Methodenaufrufs nicht eindeutig, wird die Methode ausgewählt, die zu allen in Frage kommenden Methoden die speziellste ist

- Eine Methode A ist spezieller als eine Methode B, falls die Parametertypen von A aus den Parametertypen von B abgeleitet werden können

statische Polymorphie: was geht nicht



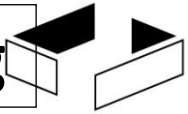
StatischePolymorphie1

Compile | Undo | Cut | Copy | Paste | Find... | Close

```
1  public class StatischePolymorphie1{
2
3      public int mach(int x, int y) {
4          return x + y;
5      }
6
7      public int mach(int a, int b) {
8          return a
9      }
10 }
```

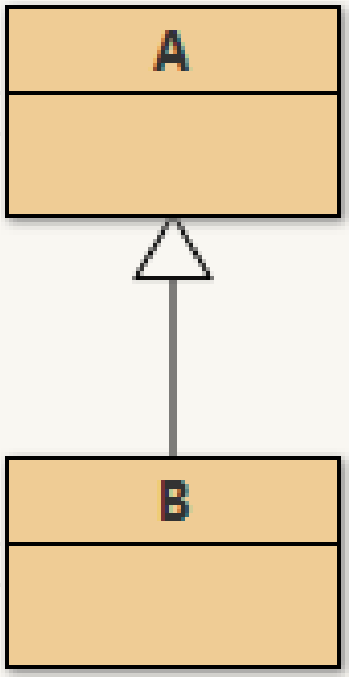
method mach(int,int) is already defined in class StatischePolymorphie1

statische Polymorphie: was geht (1/3) - Vorbereitung

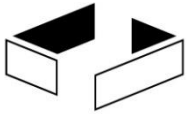


```
public class A {
    protected int wert;

    public A(int wert) {
        this.wert = wert;
    }

    public int getWert() {
        return this.wert;
    }
}
```

```
public class B extends A{

    public B(int wert) {
        super(wert);
    }
}
```

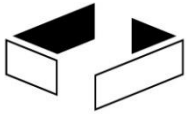


statische Polymorphie: was geht (2/3)



```
public class StatischePolymorphie1{
    EinUndAusgabe io = new EinUndAusgabe();

    public int mach(A x, A y) {
        this.io.ausgeben ("A, A \n");
        return x.getWert() + y.getWert();
    }

    public int mach(A x, B y){
        this.io.ausgeben ("A, B \n");
        return x.getWert() + y.getWert();
    }

    public int mach(B x, B y){
        this.io.ausgeben ("B, B \n");
        return x.getWert() + y.getWert();
    }
```

statische Polymorphie: was geht (3/3)

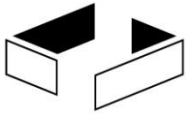


```
public class Analyse {
    public void ausfuehrenAB() {
        A a = new A(42);
        B b = new B(43);
        StatischePolymorphie1 s = new StatischePolymorphie1();
        s.mach(a, a);
        s.mach(a, b);
        s.mach(b, a);
        s.mach(b, b);
        A bb = b;
        s.mach(a, bb);
    }
```

```
A, A
A, B
A, A
B, B
A, A
```

statische Polymorphie generell sinnvoll, z. B. wenn nicht immer alle Parameter benötigt; wenn Verwirrung möglich, dann vermeiden

statische Polymorphie: was geht nicht



```
public void mach(String text){ // in StatischePolymorphie1
}

public void mach(ArrayList<Integer> list){
}
```

```
public void ausfuehrenMitNull() { // in Klasse Analyse
    StatischePolymorphie1 s = new StatischePolymorphie1();
    s.mach("Hai");
    s.mach(new ArrayList<Integer>());
    String str = null;
    s.mach(str);  // ok
    s.mach(null);
}
```

```
reference to mach is ambiguous
  both method mach(java.lang.String) in StatischePolymorphie1 and
method mach(java.util.ArrayList<java.lang.Integer>) in
StatischePolymorphie1 match
```

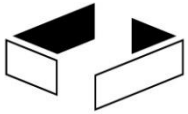
Beispiel

Beispiel

Video

dynamische Polymorphie

Grundregel der Vererbung

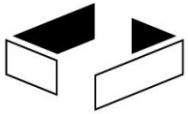


- Jedes Objekt einer erbenden Klasse kann jedes Objekt ihrer Beerbten ersetzen, es folgt:
  - überschreibende Methoden dürfen in der abgeleiteten Klasse nur schwächere Vorbedingungen haben
  - überschreibende Methoden dürfen in der abgeleiteten Klasse nur stärkere Nachbedingungen haben

Beispiele:

- man kann statt eines Studierend-Objektes auch ein AustauschStudierend-Objekt nutzen, da sich das Verhalten durch überschriebene Methoden nur im Detail verändert
- Methode zum Speichern von Daten wird nicht mit einer Methode zum Formatieren der Festplatte überschrieben

Idee der dynamischen Polymorphie



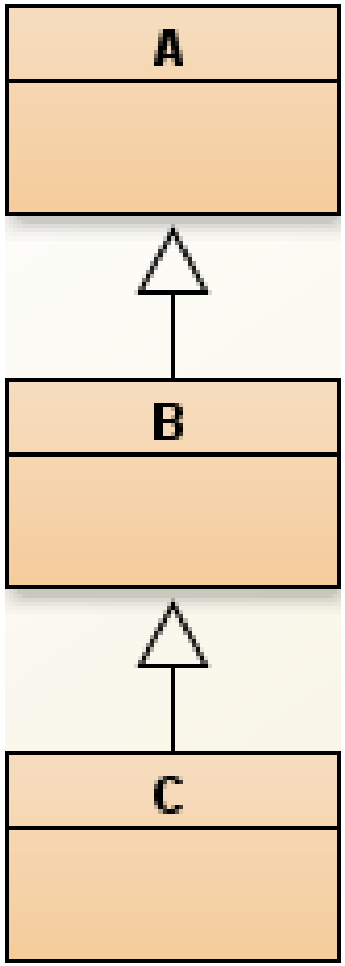
- Beim Aufruf einer Methode wird zur Laufzeit geprüft, zu welcher Klasse (genauer am weitesten abgeleiteten Klasse) das Objekt gehört
- dann wird in dieser Klasse nach der Methode gesucht und wenn gefunden, ausgeführt
- danach wird schrittweise jede beerbte Klasse nach und nach geprüft, ob sie die Methode realisiert und dann ausgeführt

```
Studierend studi = x;
studi.mach();
```

bedeutet nicht unbedingt, dass Methode mach() in Studierend ausgeführt wird, wenn x zu einer direkt oder indirekt beerbten Klasse von Studierend gehört

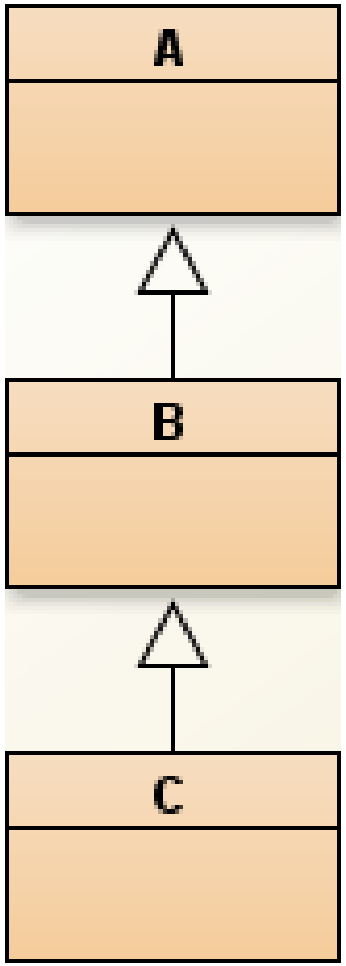
- **Hinweis: nicht einfach, macht aber die Entwicklung oft sehr viel einfacher und eleganter; zentrales Konzept**

Beispiel zur dynamischen Polymorphie (1/6)



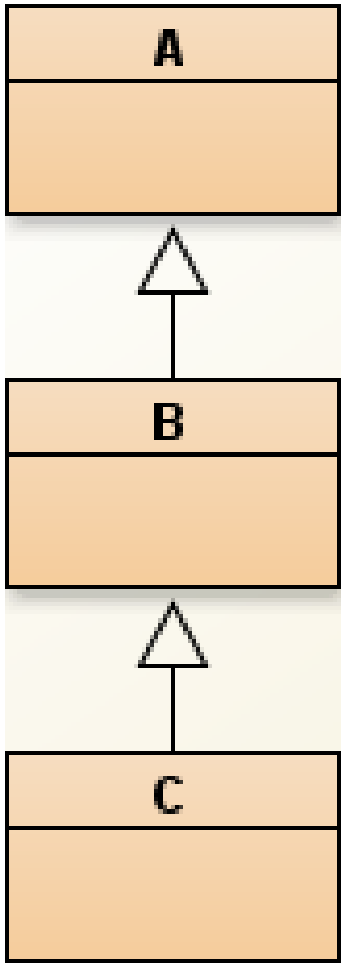

```
public class A{
protected EinUndAusgabe io = new EinUndAusgabe();
  public void meth1(){
    this.io.ausgeben("A1\n");
  }

  public void meth2(){
    this.io.ausgeben("A:meth2 ");
    meth3();
  }

  public void meth3(){
    this.io.ausgeben("A3\n");
  }
}
```

Beispiel zur dynamischen Polymorphie (2/6)



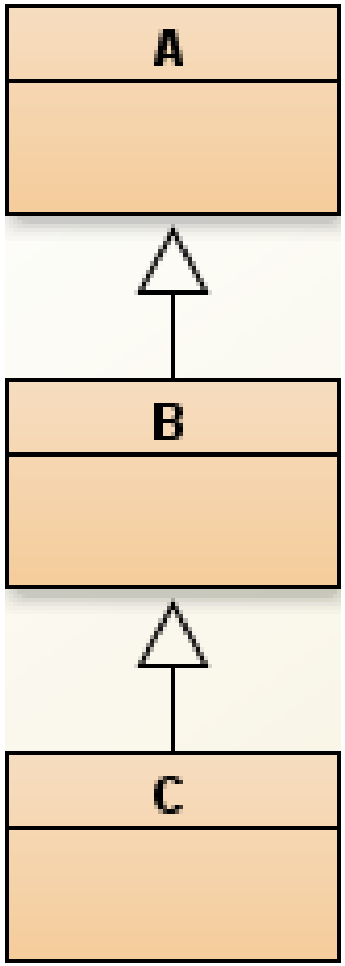

```
public class B extends A{

  @Override
  public void meth1(){
    super.io.ausgeben("B1\n");
  }

  @Override
  public void meth3(){
    super.io.ausgeben("B3\n");
  }
}
```

Beispiel zur dynamischen Polymorphie (3/6)




```
public class C extends B{
  @Override
  public void meth1(){
    super.io.ausgeben("C1\n");
  }
  @Override
  public void meth2(){
    super.io.ausgeben("C:meth2 ");
    super.meth2();
  }
  @Override
  public void meth3(){
    super.io.ausgeben("C3\n");
  }
}
```

Beispiel zur dynamischen Polymorphie (4/6)




```
meth1: ausgeben(A1)
meth2: meth3()
meth3: ausgeben(A3)
```

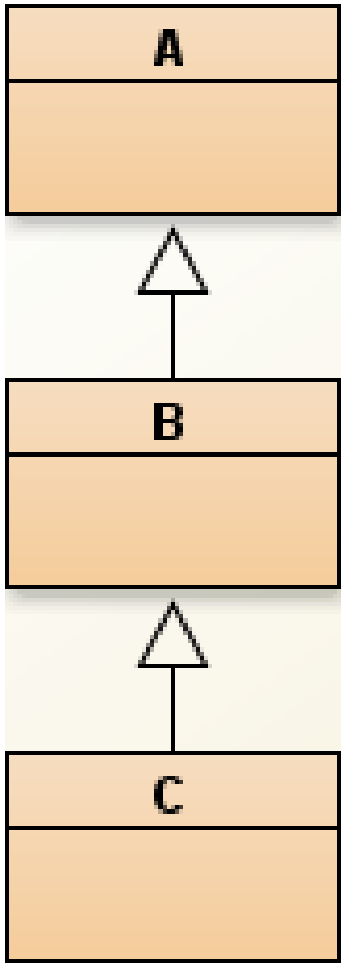
```
meth1: ausgeben(B1)
meth3: ausgeben(B3)
```

```
meth1: ausgeben(C1)
meth2: super.meth2()
meth3: ausgeben(C3)
```

```
public class Analyse{

  public void analyseA(){
    A a = new A();
    a.meth1();
    a.meth2();
  }
```

```
A1
A:meth2 A3
```

Beispiel zur dynamischen Polymorphie (5/6)




```
meth1: aus(A1)
meth2: meth3()
meth3: aus(A3)
```

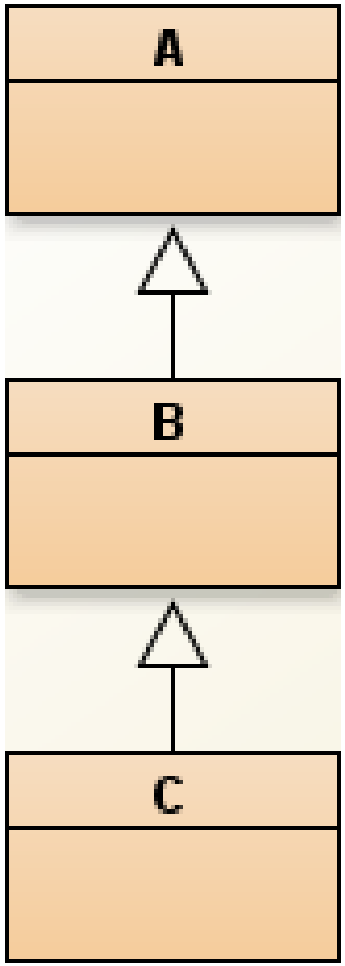
```
meth1: aus(B1)
meth3: aus(B3)
```

```
meth1: aus(C1)
meth2: super.meth2()
meth3: aus(C3)
```

```
public void analyseB(){
  A b = new B();
  b.meth1();
  b.meth2();
}
```

```
B1
A:meth2 B3
```

Beispiel zur dynamischen Polymorphie (6/6)




```
meth1: aus(A1)
meth2: meth3()
meth3: aus(A3)
```

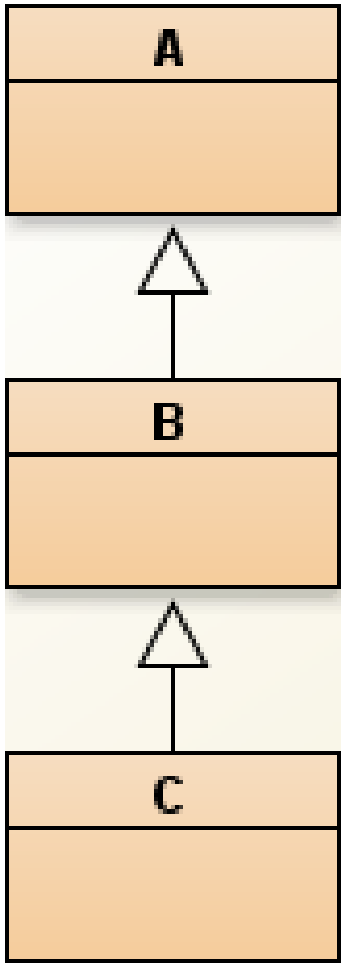
```
meth1: aus(B1)
meth3: aus(B3)
```

```
meth1: aus(C1)
meth2: super.meth2()
meth3: aus(C3)
```

```
public void analyseC(){
  A c = new C();
  c.meth1();
  c.meth2();
}
```

```
C1
C:meth2 A:meth2 C3
```

Wo wird Methode gesucht?



- unabhängig vom Typ der Variablen, wird der Typ der Erzeugung im Objekt gemerkt

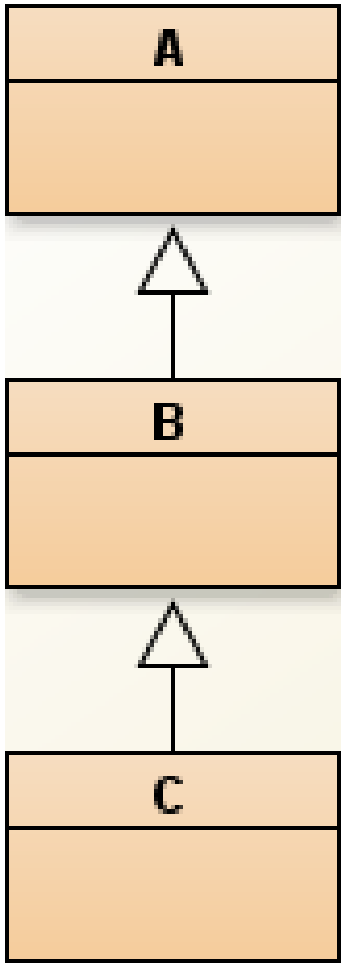
```
A a = new C();
```

- a hat Typ A, es können nur in A definierte Methoden ausgeführt werden
- beim Aufruf einer Methode wird immer in der Erzeugungsklasse nach dieser Methode gesucht und dann schrittweise in den beerbten Klassen

```
a.methode();
```

- wenn methode() in C, dann ausführen
- wenn nicht in C, dann, wenn in B, dann ausführen
- wenn nicht in B, dann, wenn in A, dann ausführen
- wenn nicht in A, dann Fehlermeldung

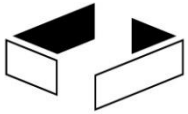
Besonderheit: Hochhangeln mit super




```
A a = new C();
a.methode();
```

- Erinnerung: methode() zunächst in C gesucht
- wenn in methode() [egal ob in A, B, oder C] dann methode2() aufgerufen wird, beginnt die Suche wieder in C
- wenn in methode() in C dann super.methode2() aufgerufen wird, beginnt die Suche in B
- wenn in methode2() in B dann super.methode3() aufgerufen wird, beginnt die Suche in A (!!!)
- ohne „super“ wird Methode immer in C gesucht
- mit „super“ wird ausgehend von der aktuell genutzten Klasse gesucht

Beispiel mit super



```
public class A {
  protected EinUndAusgabe io = new EinUndAusgabe();
  public void meth1(){
    this.io.ausgeben("in A\n");
  }
}
```

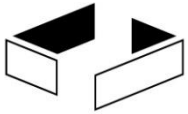
```
public class B extends A{
  @Override public void meth1(){
    super.io.ausgeben("in B ");
    super.meth1();
  }
}
```

```
public class C extends B{
  @Override public void meth1(){
    super.io.ausgeben("in C ");
    super.meth1();
  }
}
```

```
// in Klasse Analyse
public void analyseC(){
  A c = new C();
  c.meth1();
}
```

```
in C in B in A
```

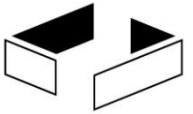
Einfache Erweiterung (1/2)



- In erbenden Klassen können Methoden ergänzt werden, die dann von Objekten dieser Klasse (und davon erbenden Klassen) nutzbar sind
- in AustauschStudierend:

```
public String gruss(){
  if(this.land.equals("USA")){
    return "Howdy " + super.vorname;
  }
  if(this.land.equals("OSF")){
    return "Moin " + super.vorname;
  }
  return "Hello "+ super.vorname;
}
```

Einfache Erweiterung (2/2)



- Erweiterungen in der beerbten Klasse und für Variablen vom Typ der beerbten Klasse nicht nutzbar
- in Analyse:

```
public void gruessen(){
  AustauschStudierend aus = new AustauschStudierend(
                    "USA","Mo","Jo",1989,"ITI",424243);
  EinUndAusgabe io = new EinUndAusgabe();
  io.ausgeben(aus.gruss());
}
```

```
Howdy Mo
```

```
public void gruessen2(){
  Studierend s = new Studierend("Mo","Jo",1989,"ITI",424243);
  EinUndAusgabe io = new EinUndAusgabe();
  io.ausgeben(s.gruss());
}
```

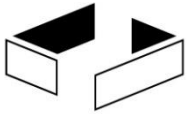
Undeclared method: gruss(...)

Beispiel

Video

casten

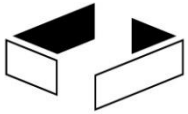
Neues Inkrement



- Die Verwaltung soll um eine Möglichkeit ergänzt werden, die Anzahl der AustauschStudierenden zu berechnen
- Problem: ArrayList<Studierend> kennt nur Objekte der Klasse Studierend und man kann mit bisherigem Wissen nicht feststellen, ob Objekt auch zu einer erbenden Klasse gehört
- Gibt allerdings schmuddeligen (wieso? später!) Ansatz
  - Boolescher Operator instanceof, genauer

    <Objekt> instanceof <Klassenname>
  - gibt true, wenn Objekt auch zu dieser Klasse gehört; Objekt gehört zu der Klasse, von der es instanziiert wurde (new <Klassenname>) und zu allen Klassen, die diese Klasse beerbt hat (direkt oder indirekt)

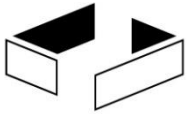
Beispiel für instanceof



```
public void instanceofSpielerei(){ // in Klasse Analyse
  AustauschStudierend aus = new AustauschStudierend("USA",
                                "Mo","Jo",1989,"ITI",424243);
  Studierend s0 = new AustauschStudierend("USA",
                                "Mo","Jo",1989,"ITI",424243);
  Studierend s1 = new Studierend("Mo","Jo",1989,"ITI",424243);
  EinUndAusgabe io = new EinUndAusgabe();
  io.ausgeben("aus: "
            + (aus instanceof AustauschStudierend) + "\n");
  io.ausgeben("aus: "+(aus instanceof Studierend)+"\n");
  io.ausgeben("s0: "+(s0 instanceof AustauschStudierend)+"\n");
  io.ausgeben("s0: "+(s0 instanceof Studierend)+"\n");
  io.ausgeben("s1: "+(s1 instanceof AustauschStudierend)+"\n");
  io.ausgeben("s1: "+(s1 instanceof Studierend)+"\n");
}
```

```
aus: true
aus: true
s0: true
s0: true
s1: false
s1: true
```

Casten

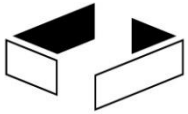


- Wenn man weiß, dass ein Objekt eines Typs auch zu einer anderen Klasse gehört, kann man das Objekt in ein Objekt der anderen Klasse umwandeln (casten):

  (<NameDerErbendenKlasse>) Objektvariable

- Sollte man in eine falsche Klasse casten, erhält man einen Fehler (Exception)
- Entwickler dafür verantwortlich, dass hier kein Fehler auftritt, da dies erst zur Laufzeit, nicht zur Compile-Zeit, erkannt wird
- Auch casten sollte man mit Bedacht einsetzen

Beispiel für Casten



```
public void castenSpielerei(){ // in Klasse Analyse
  Studierend s0 = new AustauschStudierend("USA"
                                , "Mo","Jo",1989,"ITI",424243);
  AustauschStudierend aus = (AustauschStudierend) s0;
  EinUndAusgabe io = new EinUndAusgabe();
  io.ausgeben(aus.gruss());
  s0 = new Studierend("Mo","Jo",1989,"ITI",424243);
  aus = (AustauschStudierend) s0;
  io.ausgeben(aus.gruss());
}
```

```
53        aus = (Austauschstudierend) s0;
```

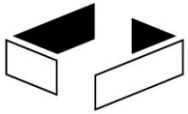
```
java.lang.ClassCastException:
class Studierend cannot be cast to class Austauschstudierend (Studierend and
Austauschstudierend are in unnamed module of loader java.net.URLClassLoader
@2fd8bb16)
```

BlueJ: Konsole - BeispielStudierendenlisteMitVererbung

Optionen

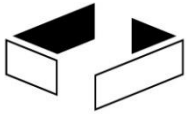
```
Howdy  Mo
```

instanceof und casten nur im "Notfall"



- Casten sowie instanceof sollte man nur mit allergrößter Vorsicht nutzen, da
  - der Programmcode viele if-Konstrukte enthält
  - bei Erweiterungen mit weiteren erbenden Klassen weitere Alternativen hinzukommen
  - wenn man instanceof und casten häufig benötigt, hat man beim Programmdesign einen großen Fehler gemacht
    - z. B. eine ArrayList<Studierend> und man immer wieder Alternativen für AustauschStudierenden benötigt
- Grundsätzlich deutet Nutzung auf schlechte Programmierung
- Merksatz: "Ca(r)sten ist böse"
- Casten in Standardprogrammen selten sinnvoll, sinnvoll/notwendig bei Frameworks, bei denen Software für andere Software erstellt wird

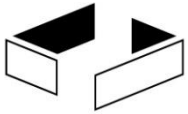
Anzahl AustauschStudierenden: schwächere Lösung



```
private int anzahlAustauschStudierendeNichtSoToll( // in Analyse
                                ArrayList<Studierend> Studierenden){
  int ergebnis = 0;
  for(Studierend s: this.Studierenden){
    if(s instanceof AustauschStudierend){ // hölzern
      ergebnis = ergebnis + 1;
    }
  }
  return ergebnis;
}

public void beispielAustauschstudierendeZaehlen0(){  // in Analyse
  ArrayList<Studierend> tmp = new ArrayList<Studierend>();
  tmp.add(new AustauschStudierend("USA","Mo","Jo",1989,"ITI",4243));
  tmp.add(new AustauschStudierend("USA","Mo","Jo",1989,"ITI",4242));
  tmp.add(new Studierend("Mo","Jo",1989,"ITI",4244));
  new EinUndAusgabe().ausgeben( "Anzahl: "
                 + this.anzahlAustauschStudierendenNichtSoToll(tmp));
}
```

```
Anzahl: 2
```

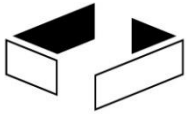
Anzahl AustauschStudierenden: bessere Alternative



- Nutzung von dynamischer Polymorphie
- Einführung in Studierend einer neuen Methode zaehlenAlsAustauschStudierend
- Methode gibt den Wert 0 für Studierend zurück
- Methode wird in AustauschStudierend überschrieben und gibt Wert 1 zurück
- Werte aller Studierenden werden summiert

- Hinweis: Ok, ist kleiner Trick, aber die Behandlung von Alternativen wurde in Polymorphie verschoben

Realisierung (1/2)



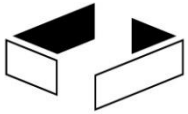
- in Studierend:

```
public int zaehlenAlsAustauschStudierend(){
  return 0;
}
```

- in AustauschStudierend:

```
@Override
public int zaehlenAlsAustauschStudierend(){
  return 1;
}
```

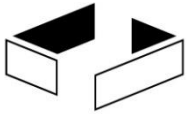
Realisierung (2/2)



```
public int anzahlAustauschStudierende(){// in Analyse
                                ArrayList<Studierend> Studierenden){
  int ergebnis = 0;
  for(Studierend s:this.studierende){
    ergebnis = ergebnis + s.zaehlenAlsAustauschStudierend();
  }
  return ergebnis;
}

public void beispielAustauschstudisZaehlen0(){  // in Analyse
  ArrayList<Studierend> tmp = new ArrayList<Studierend>();
  tmp.add(new AustauschStudierend("USA","Mo","Jo",1989,"ITI",4243));
  tmp.add(new AustauschStudierend("USA","Mo","Jo",1989,"ITI",4242));
  tmp.add(new Studierend("Mo","Jo",1989,"ITI",4244));
  new EinUndAusgabe().ausgeben( "Anzahl: "
                + this.anzahlAustauschStudierende(tmp));
}
```

```
Anzahl: 2
```

Überblick: Wiederverwendung durch OO



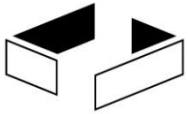
- Im Laufe der Entwicklung der für eine Software benötigten Klassen kann es auffallen, dass man einige „ähnliche“ Klassen findet
- Ähnlich bedeutet eine große Gemeinsamkeit bei gewissen Objektvariablen und Methoden und größere Unterschiede bei anderen Objektvariablen und Methoden
- Die Grundidee bei der Objektorientierung ist es, die großen Gemeinsamkeiten in eine Klasse zu packen und die Spezialfälle in andere Klassen, die die gemeinsamen Eigenschaften der anderen Klasse *erben* können
- Kleine Gemeinsamkeiten nicht „auf Krampf“ in Vererbung umformen; Vererbung ist Hilfsmittel nicht Ziel; Ziel bleibt immer Wart- und Erweiterbarkeit

Beispiel

Video

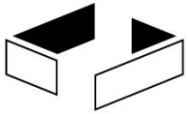
Klasse Object

Klasse Object



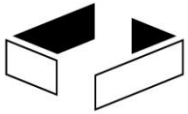
- In Java erben alle Klassen automatisch von der Klasse Object, dies ist nicht änderbar
- Ansatz erlaubt es, über dynamische Polymorphie Gemeinsamkeiten von Objekten zu nutzen, bekannt:
  - toString(): String-Repräsentation des Objekt
  - equals(): Überprüfung auf Gleichheit [gleich genauer]
- weiterer Ansatz: ArrayList<Object> kann beliebige Objekte aufnehmen (Rückumwandlung nur mit casten); entspricht der Klasse ArrayList, die so nicht mehr verwendet wird
- Zentrale Idee ist, diese Methoden (wenn benötigt) zu überschreiben
- Überschreiben von toString() und equals() immer, hashCode() fast immer sinnvoll [später genauer]

jetzt variierte Beispielklasse



```
public class Punkt{
  private int x;
  private int y;

  public Punkt(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() { return this.x; } // schlecht formatiert

  public int getY() { return this.y; }

  public void setX(int x) {
    this.x = x;
  }

  public void setY(int y) {
    this.y = y;
  }
}
```

Direkte Ausgabe

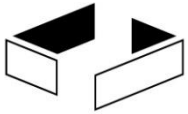


```
import java.util.ArrayList;

public class Analyse{

  public void ausgeben(){
    EinUndAusgabe io = new EinUndAusgabe();
    Punkt p1 = new Punkt(0, 0);
    Punkt p2 = new Punkt(0, 0);
    io.ausgeben("p1: " + p1 + "\n");
    io.ausgeben("p2: " + p2 + "\n");
  }
}
```

```
p1: Punkt@42e816
p2: Punkt@190d11
```

- Hier wurde toString()-Methode der Klasse Object genutzt

Nutzung von toString



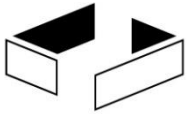
- in Punkt:

```
@Override
public String toString(){
  return "[" + this.x + "," + this.y + "]";
}
```

- Aufruf von ausgeben() in Analyse liefert

```
p1: [0,0]
p2: [0,0]
```

Identität und Gleichheit

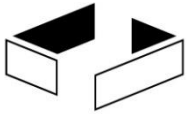


- bisher verboten, Objekte mit == zu prüfen
- == prüft auf Identität, handelt es sich um dieselben Objekte?
- mit equals soll inhaltliche Gleichheit geprüft werden, also ob es sich um das gleiche Objekt handelt (fast immer das, was man wissen will)
- Standardimplementierung von equals() in Object unterscheidet dasselbe und das gleiche Objekt nicht

```
public boolean equals (Object other){
  return this == other;
}
```

- Jede Java-Klasse implementiert equals() selbst, ansonsten werden inhaltliche Gleichheit und Identität nicht unterschieden
- ***wir müssen*** `public boolean equals (Object other)` ***überschreiben!***

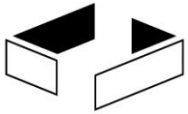
Identität bei Punkt-Objekten



```
// equals in Punkt nicht überschrieben
public void gleichUndIdentisch(){ // in Klasse Analyse
  EinUndAusgabe io = new EinUndAusgabe();
  Punkt p1 = new Punkt(0, 0);
  Punkt p2 = new Punkt(0, 0);
  Punkt p3 = p1;
  io.ausgeben("p1==p2: " + (p1 == p2) + "\n");
  io.ausgeben("p1==p3: " + (p1 == p3) + "\n");
  io.ausgeben("p1.equals(p2): " + (p1.equals(p2)) + "\n");
  io.ausgeben("p1.equals(p3): " + (p1.equals(p3)) + "\n");
}
```

```
p1==p2: false
p1==p3: true
p1.equals(p2): false
p1.equals(p3): true
```

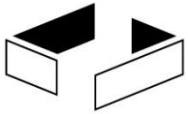
Erinnerung: unvollständiges equals()



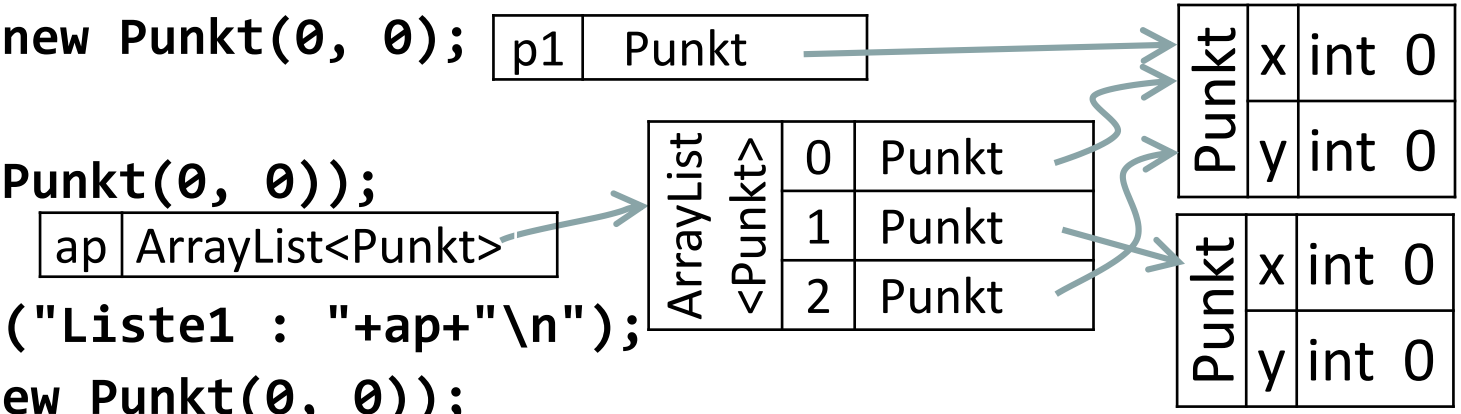
```
public boolean equals(Punkt other) {
    EinUndAusgabe io = new EinUndAusgabe();
    io.ausgeben("in equals(Punkt)\n");
    if (other == null) {
        return false;
    }
    return this.x == other.getX() && this.y == other.getY();
}
// Ausgabe von gleichUndIdentisch() von vorheriger Folie
// ist erstmal ok
```

```
p1==p2: false
p1==p3: true
in equals(Punkt)
p1.equals(p2): true
in equals(Punkt)
p1.equals(p3): true
```

Frust mit remove() und Liste (1/4)

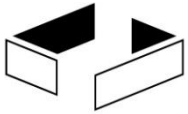


```
public void objektInListeLoeschen(){
  EinUndAusgabe io = new EinUndAusgabe();
  ArrayList<Punkt> ap = new ArrayList<Punkt>();
  Punkt p1 = new Punkt(0, 0);
  ap.add(p1);
  ap.add(new Punkt(0, 0));
  ap.add(p1);
  io.ausgeben("Liste1 : "+ap+"\n");
  ap.remove(new Punkt(0, 0));
  io.ausgeben("Liste2 : "+ap+"\n");
  ap.remove(p1);
  io.ausgeben("Liste3 : "+ap+"\n");
}
```




```
Liste1 : [[0,0], [0,0], [0,0]]
Liste2 : [[0,0], [0,0], [0,0]]
Liste3 : [[0,0], [0,0]]
```

Frust mit remove() und Liste (2/4)

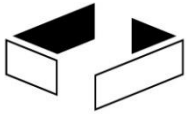


- eigentlich erwartet: gleiche Objekte werden gelöscht
- Problem: remove() nutzt intern die equals(Object)-Methode von Object; Punkt erbt zwar von Object überschreibt aber diese Methode nicht (nur Punkt mit Punkt verglichen)
- intern wird in ArrayList nur mit Klasse Object gearbeitet

```
public void klasseObjectNutzen(){ // in Klasse Analyse
  EinUndAusgabe io = new EinUndAusgabe();
  Punkt p1 = new Punkt(1,2);
  Object o1 = p1;
  Object o2 = new Punkt(1,2);
  io.ausgeben("p1==o1: "+(p1==o1)+"\n");
  io.ausgeben("o1==p1: "+(o1==p1)+"\n");
  io.ausgeben("o1.equals(p1): "+(o1.equals(p1))+"\n");
  io.ausgeben("o1.equals(o2): "+(o1.equals(o2))+"\n");
}
```

```
p1==o1: true
o1==p1: true
o1.equals(p1): true
o1.equals(o2):
false
```

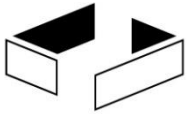
(kein) Frust mit remove() und Liste (3/4)



- vollständiges equals:

```
@Override
public boolean equals(Object obj) {
    EinUndAusgabe io = new EinUndAusgabe();
    io.ausgeben("in equals(Object)\n");
    if (obj == null || (this.getClass() != obj.getClass())) {
        return false;
    }
    Punkt other = (Punkt) obj;
    return this.equals(other); // unser altes equals
}
```

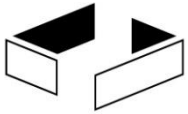
(kein) Frust mit remove() und Liste (4/4)



- neue Ausgabe von objektInListeLoeschen():

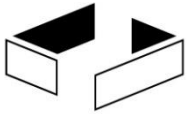
```
Liste1 : [[0,0], [0,0], [0,0]]
in equals(Object)
in equals(Punkt)
Liste2 : [[0,0], [0,0]]
in equals(Object)
in equals(Punkt)
Liste3 : [[0,0]]
```

Zusammenfassung: Vollständiges equals() (1/2)



```
@Override // so sicherstellen, dass Methode ueberschrieben
public boolean equals(Object obj) {
  if (obj == null || (this.getClass() != obj.getClass())) {
    return false;
  }
  AktuelleKlasse other = (AktuelleKlasse) obj;

  „für jede Objektvariable obv, die eine Klasse als Typ hat:“
  if(this.obv == null) {
    if (other.getObv() != null) {
      return false;
    }
  } else {
    if (! this.obv.equals(other,getObv())) {
      return false;
  } // weiter nächster Folie
```

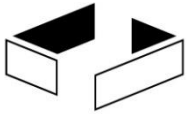
Zusammenfassung: Vollständiges equals() (2/2)



```
    „für jede Objektvariable obv, die einen elementaren Typ
     als Typ hat:“
    if (this.obv != other.getObv()) {
      return false;
    }
    return true;
  }
```

- genereller Ansatz: Suche schrittweise einen Verstoß gegen Gleichheit, wenn gefunden sofort „return false“
- nur wenn kein Verstoß gefunden, final „return true“
- letzter Teil für elementare Typen einfach mit || verknüpfbar
- `(this.getClass() != obj.getClass());` zu jeder Klasse existiert genau ein Class-Objekt

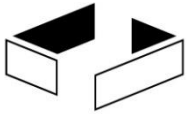
Ausflug: getClass() (1/2)



- Zu jedem Java-Typen gibt es genau ein Class-Objekt
- soll der Typ von zwei Objekten gleich sein, wird die Identität des zugehörigen Class-Objekts abgefragt
- später: Class-Objekte erlauben Einstieg in das Thema Reflection, z. B. führe Methode aus, deren Name erst zur Laufzeit bekannt wird

```
Punkt p = new Punkt(41,42);
Class cp = p.getClass();
Punkt q = new Punkt(1,2);
Class cq = q.getClass();
cp == cq
    true   (boolean)
```

Ausflug: getClass() (2/2)

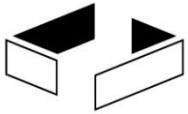


```
import java.lang.reflect.Method;

public class Reflexion {
    public void beispiel() throws Exception{
        EinUndAusgabe io = new EinUndAusgabe();
        Punkt p = new Punkt(0,42);
        io.ausgeben(p);
        Class cl = p.getClass();
        Method m = cl.getMethod("setX", new Class[]{int.class});
        m.invoke(p, new Integer[]{41});
        io.ausgeben(p);
    }
}
```

```
[0,42][41,42]
```

- nur kleine Java-Show zum Posen, muss jetzt nicht verstanden werden

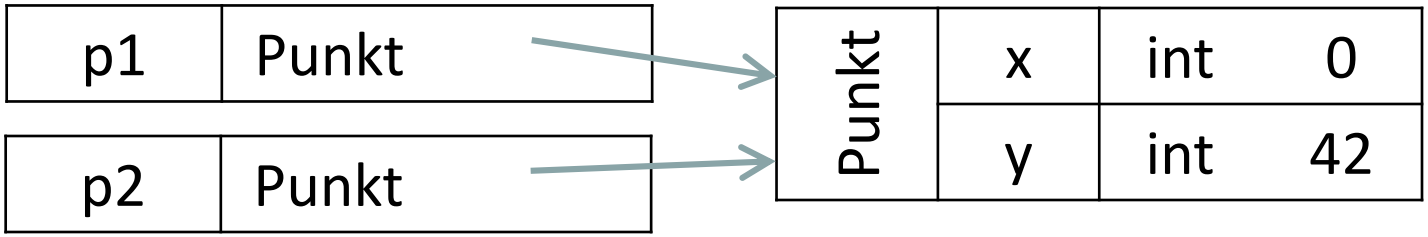
Erinnerung: Java arbeitet mit Objektreferenzen 1



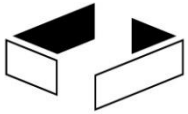
Video

```
public void zuweisen(){
  EinUndAusgabe io = new EinUndAusgabe();
  Punkt p1 = new Punkt(0,0);
  Punkt p2 = p1;
  p2.setY(42);
  io.ausgeben("p1: " + p1 + "\n");
  io.ausgeben("p2: " + p2 + "\n");
}
```

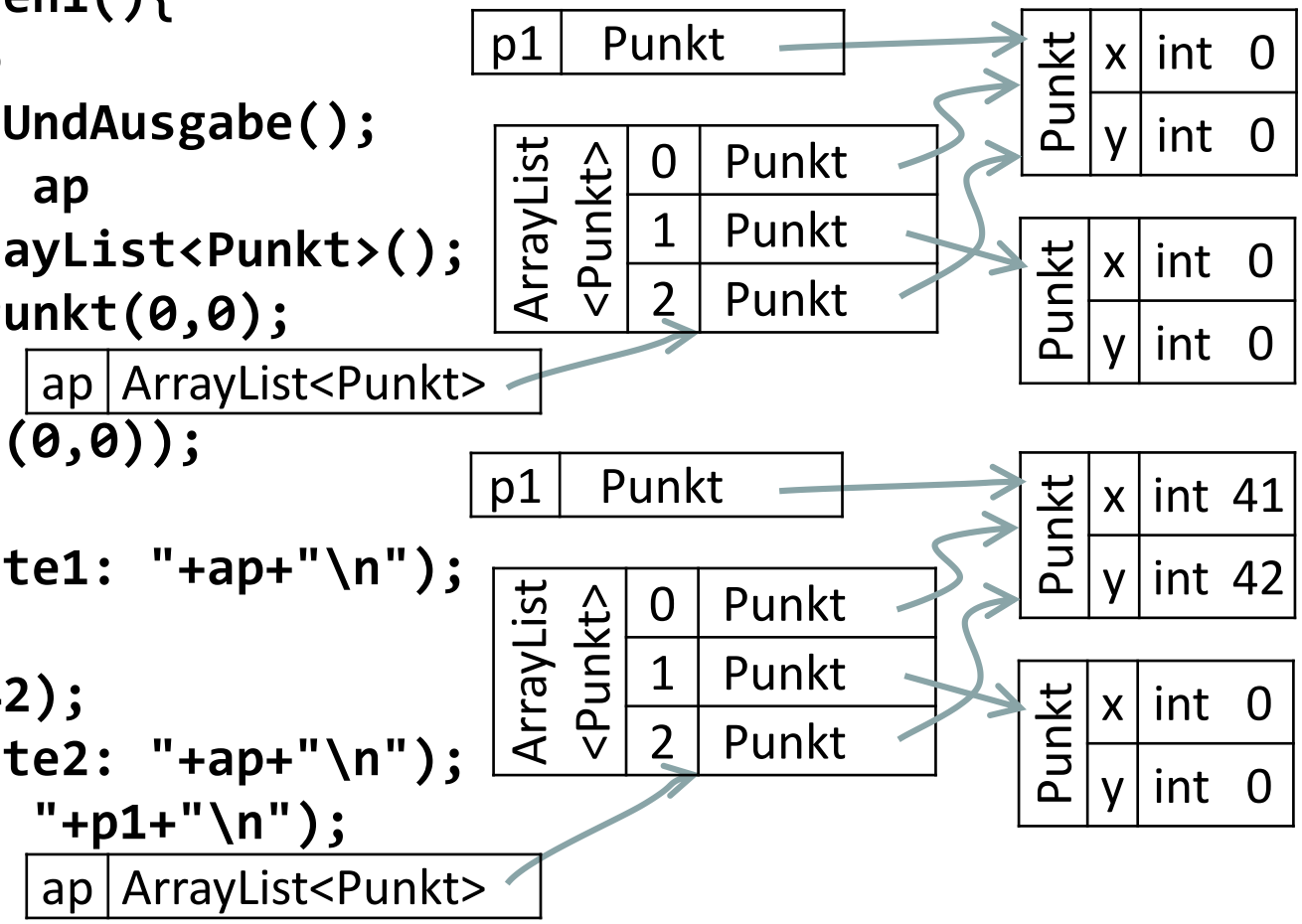
```
p1: [0,42]
p2: [0,42]
```



Erinnerung: Java arbeitet mit Objektreferenzen 2

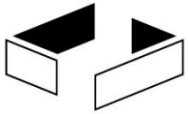


```
public void zuweisen1(){
  EinUndAusgabe io
        = new EinUndAusgabe();
  ArrayList<Punkt> ap
        = new ArrayList<Punkt>();
  Punkt p1 = new Punkt(0,0);
  ap.add(p1);
  ap.add(new Punkt(0,0));
  ap.add(p1);
  io.ausgeben("Liste1: "+ap+"\n");
  p1.setX(41);
  ap.get(0).setY(42);
  io.ausgeben("Liste2: "+ap+"\n");
  io.ausgeben("p1: "+p1+"\n");
}
```



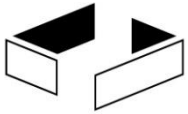

```
Liste1: [[0,0], [0,0], [0,0]]
Liste2: [[41,42], [0,0], [41,42]]
p1: [41,42]
```

wenn keine Referenzen erwünscht



- in der Praxis muss fast nur mit Objektreferenzen gearbeitet werden; trotzdem soll es möglich sein, Ausgangsobjekte nicht zu verändern
- Wunsch: Erstellung echter Objektkopien ohne gemeinsame Referenzen
- Ansatz: Realisierung einer Methode clone()
- In Java bereits durch Klasse Object etwas unterstützt, gibt Methode protected Object clone()
- protected nicht direkt nutzbar, aber überschreibbar
- [später: Klasse soll dann Interface Cloneable realisieren]
- Beim Kopieren muss der clone()-Befehl an jede Objektvariable weitergegeben werden; diesen wieder weitergeben

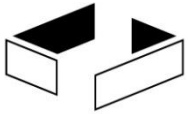
Beispiel: Realisierung von Clone



- Anmerkung: für elementare Datentypen und Klassen die Konstanten nutzen (int), wird kein clone() benötigt
- Beobachtung: Integer hat keine set-Methode für den Wert

```
public class Punkt implements Cloneable{
    //..

    @Override
    public Punkt clone(){
          return new Punkt(this.x, this.y);
    }
```

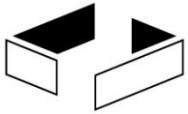
Nutzung von clone()



```
public void zuweisen2(){ // in Analyse
  EinUndAusgabe io = new EinUndAusgabe();
  ArrayList<Punkt> ap = new ArrayList<Punkt>();
  Punkt p1 = new Punkt(0, 0);
  ap.add(p1.clone());
  ap.add(new Punkt(0, 0));
  ap.add(p1.clone());
  io.ausgeben("Liste1: " + ap + "\n");
  p1.setX(41);
  ap.get(0).setY(42);
  io.ausgeben("Liste2: " + ap + "\n");
  io.ausgeben("p1: " + p1 + "\n");
}
```

```
Liste1: [[0,0], [0,0], [0,0]]
Liste2: [[0,42], [0,0], [0,0]]
p1: [41,0]
```

Clone-Variante: Kopierkonstruktor



- Wenn häufiger Kopien gewünscht, dann auch als Konstruktor realisierbar, der zu clonendes Objekt erhält

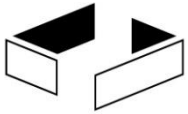
```
public Punkt(Punkt other){
  this.x = other.getX();
  this.y = other.getY();
}
```

- in Analyse, zuweisen2():

```
ap.add(new Punkt(p1));
ap.add(new Punkt(0, 0));
ap.add(new Punkt(p1));
```

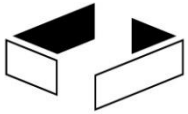
```
Liste1: [[0,0], [0,0], [0,0]]
Liste2: [[0,42], [0,0], [0,0]]
p1: [41,0]
```

Beispiel für (fast) vollständige Klasse (1/6)



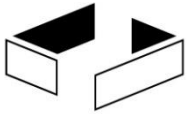
```
public class Linie { // implements Cloneable{
  private Punkt start;
  private Punkt ende;

  public Linie(Punkt start, Punkt ende) {
    this.start = start;
    this.ende = ende;
  }

  public Punkt getStart() {
    return start;
  }

  public void setStart(Punkt start) {
    this.start = start;
  }
```

Beispiel für (fast) vollständige Klasse (2/6)



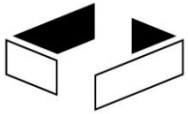
```
public Punkt getEnde() {
  return ende;
}

public void setEnde(Punkt ende) {
  this.ende = ende;
}

@Override
public String toString(){
  return "[" + this.start + "->" + this.ende + "]";
}
```

Beispiel für (fast) vollständige Klasse (3/6)



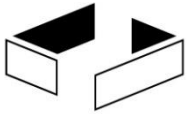
```
@Override
public Linie clone(){
  Punkt tmp1 = null;
  Punkt tmp2 = null;
  if (this.start != null){
    tmp1 = this.start.clone();
  }
  if (this.ende != null){
    tmp2 = this.ende.clone();
  }
  return new Linie(tmp1, tmp2);
}
```

Beispiel für (fast) vollständige Klasse (4/6)



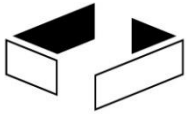
```
@Override
public boolean equals(Object obj) {
    if (obj == null || (this.getClass() != obj.getClass())) {
        return false;
    }
    Linie other = (Linie) obj;
    if (this.ende == null) {
        if (other.getEnde() != null) {
            return false;
        }
    } else {
        if (!this.ende.equals(other.getEnde())) {
            return false;
        }
    } // weiter naechste Folie
```

Beispiel für (fast) vollständige Klasse (5/6)



```
        if (this.start == null) {
            if (other.getStart() != null) {
                return false;
            }
        } else {
            if (!this.start.equals(other.getStart())) {
                return false;
            }
        }
        return true;
    }

// warum fast vollständig? fehlt Methode public int hashCode()
// (spaeter)
```

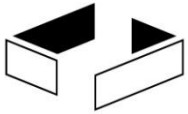
Beispiel für (fast) vollständige Klasse (6/6)



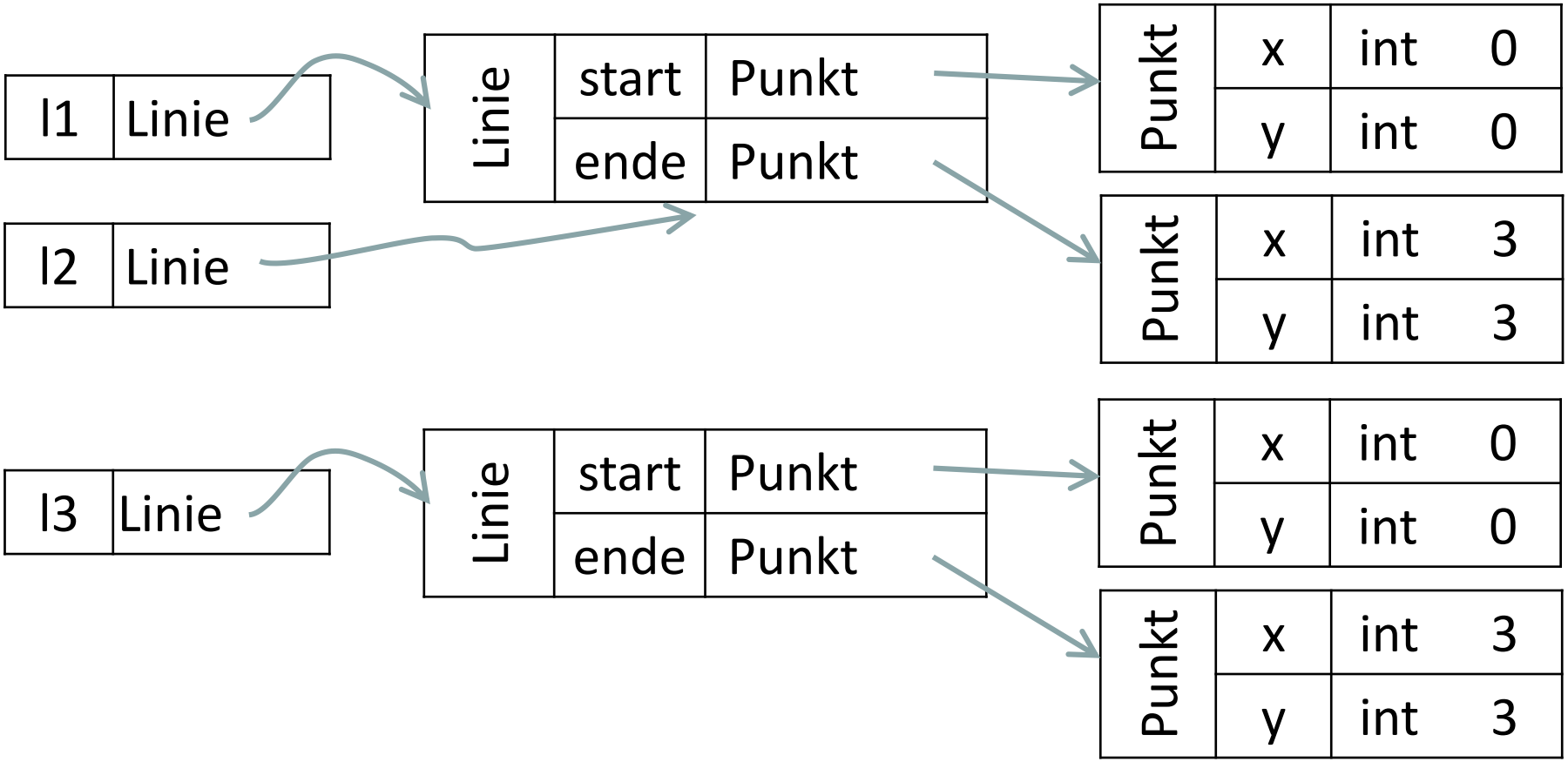
```
public void linienspielerei(){
  EinUndAusgabe io = new EinUndAusgabe();
  Linie l1 = new Linie(new Punkt(0, 0),new Punkt(3, 3));
  Linie l2 = l1;
  Linie l3 = l2.clone();
  io.ausgeben("l1==l2: "+ (l1==l2) +"\n");
  io.ausgeben("l1==l3: "+ (l1==l3) +"\n");
  io.ausgeben("l1.equals(l2): "+ (l1.equals(l2)) +"\n");
  io.ausgeben("l1.equals(l3): "+ (l1.equals(l3)) +"\n");
  l2.setStart(new Punkt(4,4));
  io.ausgeben("l1: "+ l1 +"\n");
  io.ausgeben("l2: "+ l2 +"\n");
  io.ausgeben("l3: "+ l3 +"\n");
  l3.setEnde(l2.getStart());
  l1.getStart().setY(42);
  io.ausgeben("l1: "+ l1 +"\n");
  io.ausgeben("l2: "+ l2 +"\n");
  io.ausgeben("l3: "+ l3 +"\n");
}
```

```
l1==l2: true
l1==l3: false
l1.equals(l2): true
l1.equals(l3): true
l1: [[4,4]->[3,3]]
l2: [[4,4]->[3,3]]
l3: [[0,0]->[3,3]]
l1: [[4,42]->[3,3]]
l2: [[4,42]->[3,3]]
l3: [[0,0]->[4,42]]
```

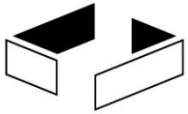
Analyse mit Speicherbildern (1/2)



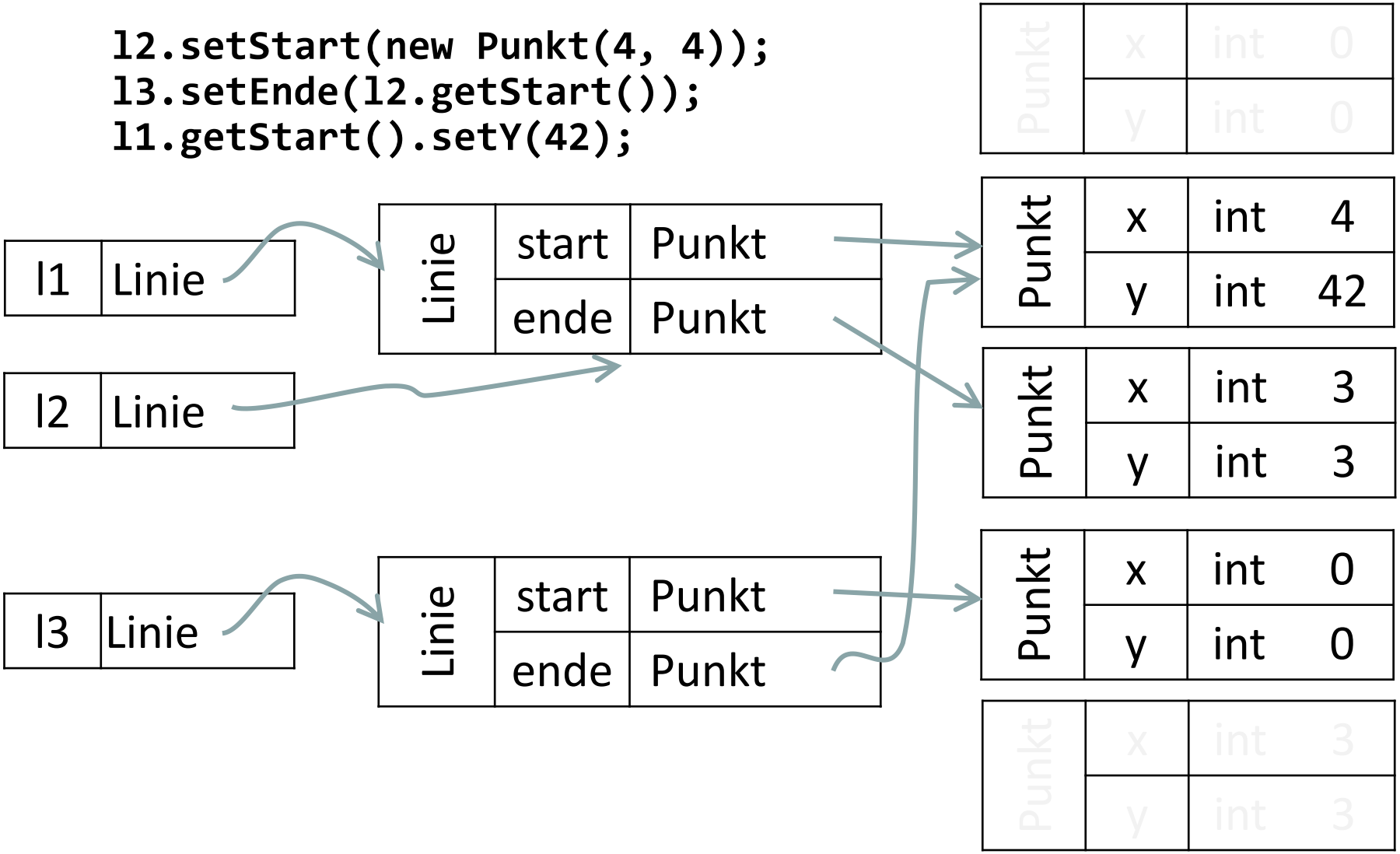
```
Linie l1 = new Linie(new Punkt(0, 0),new Punkt(3, 3));
Linie l2 = l1;
Linie l3 = l2.clone();
```



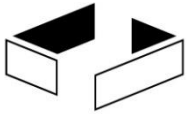
Analyse mit Speicherbildern (2/2)



```
l2.setStart(new Punkt(4, 4));
l3.setEnde(l2.getStart());
l1.getStart().setY(42);
```



Tiefe Kopie vs. flache Kopie (1/4): tief 1/2

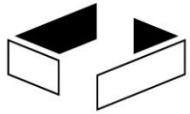


- Kopierkonstruktor (tief: jede Objektvariable wird gecloned)

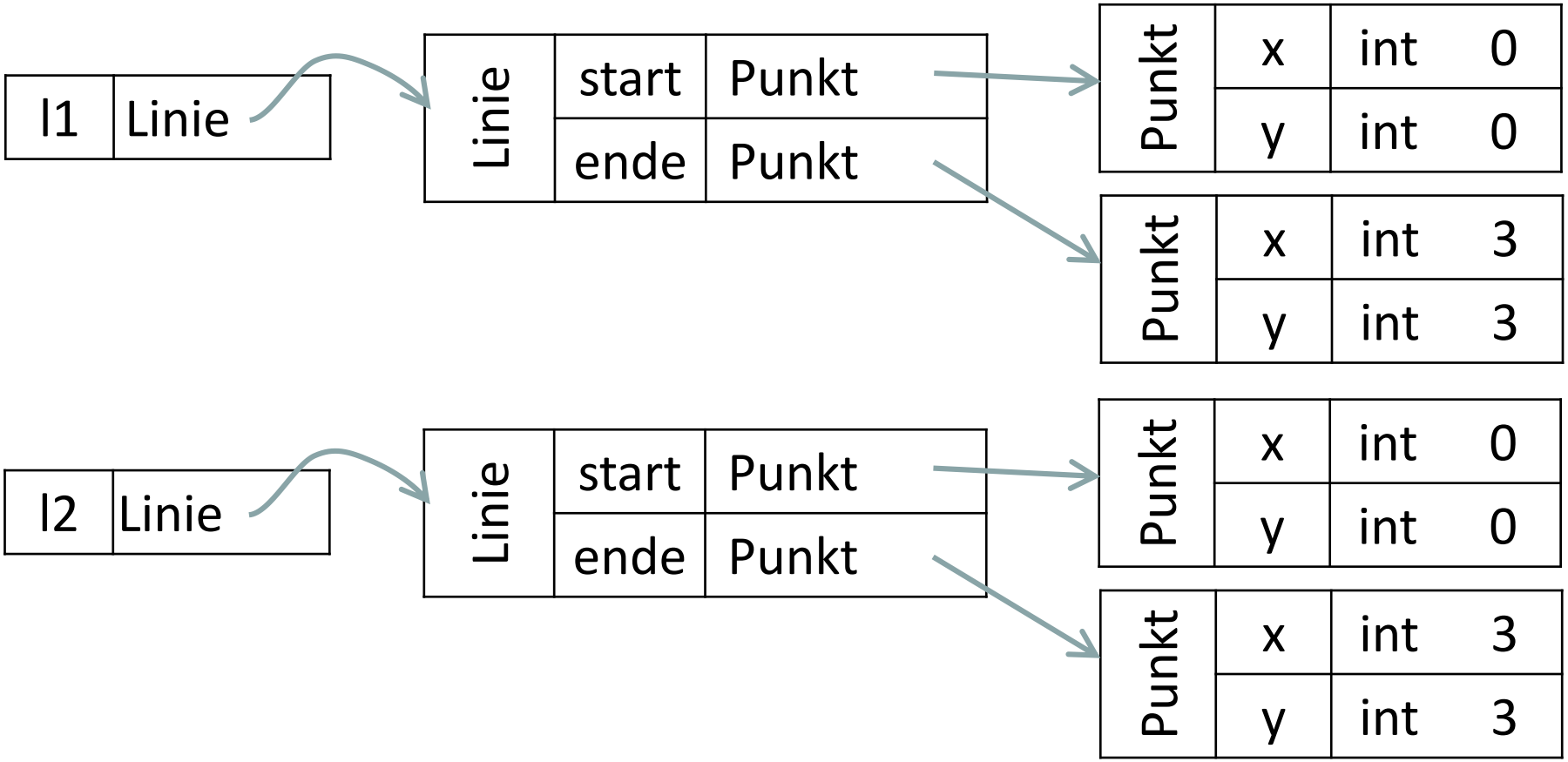
```
public Linie(Linie other) {
  if (other.getStart() != null) {
    this.start = other.getStart().clone();
  }
  if (other.getEnde() != null) {
    this.ende = other.getEnde().clone();
  }
}

public void kopieranalyse(){ // in Analyse
  EinUndAusgabe io = new EinUndAusgabe();
  Linie l1 = new Linie(new Punkt(0,0), new Punkt(3,3));
  Linie l2 = new Linie(l1);
  io. ausgeben("l1 == l2: " + (l1 == l2) + "\n");
  io. ausgeben("l1.start == l2.start: "
       + (l1.getStart() == l2.getStart()) + "\n");
}
```

```
l1 == l2: false
l1.start == l2.start: false
```

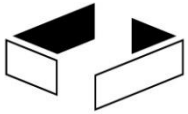
Tiefe Kopie vs. flache Kopie (2/4): tief 2/2



- Kopierkonstruktor (tief: jede Objektvariable wird gecloned)



Tiefe Kopie vs. flache Kopie (3/4): flach 1/2



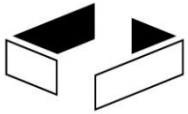
- Kopierkonstruktor (flach: neues Objekt, gemeinsame Referenzen)

```
public Linie(Linie other) { // flache Kopie
  this.start = other.getStart();
  this.ende = other.getEnde();
}

public void kopieranalyse(){ // in Analyse
  EinUndAusgabe io = new EinUndAusgabe();
  Linie l1 = new Linie(new Punkt(0,0), new Punkt(3,3));
  Linie l2 = new Linie(l1);
  io. ausgeben("l1 == l2: " + (l1 == l2) + "\n");
  io. ausgeben("l1.start == l2.start: "
       + (l1.getStart() == l2.getStart()) + "\n");
}
```

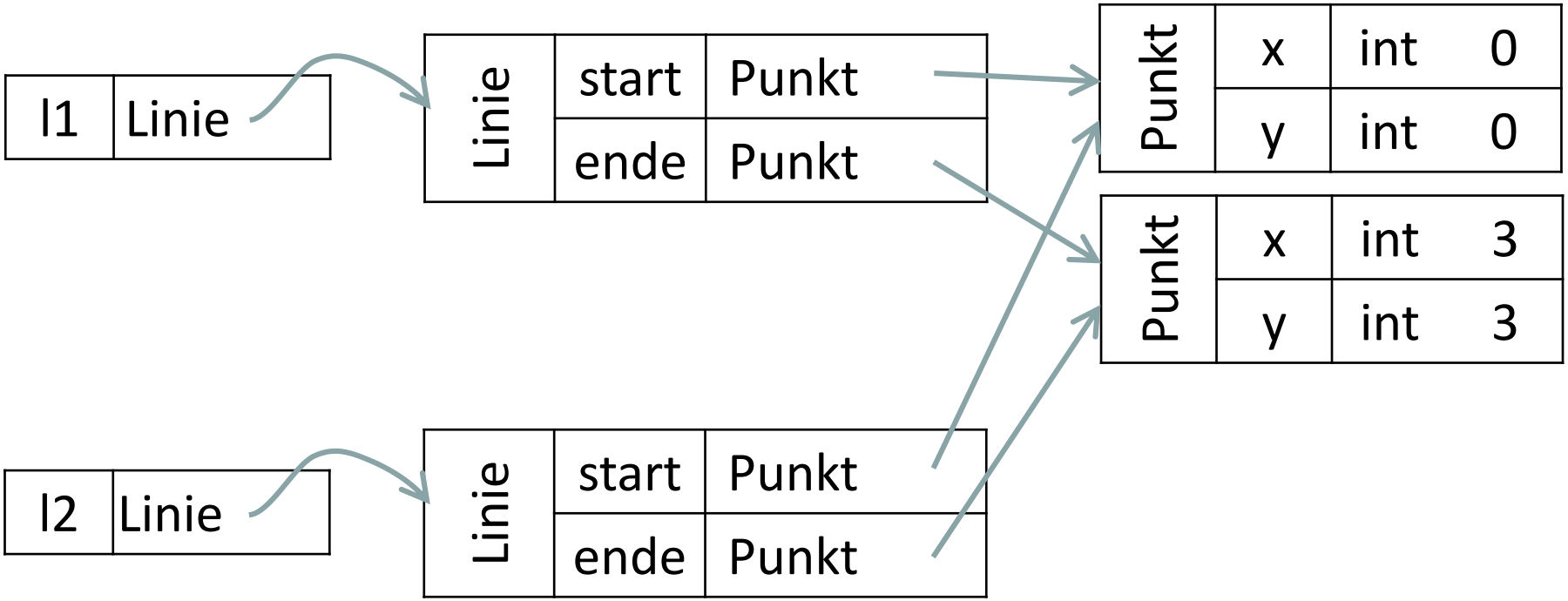
```
l1 == l2: false
l1.start == l2.start: true
```

```
// wenn in Objektvariablen identische Objekte referenziert
// werden, wird der Ansatz noch komplexer
```

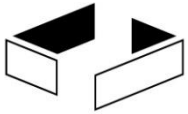
Tiefe Kopie vs. flache Kopie (4/4): flach 2/2



- Kopierkonstruktor (flach: neues Objekt, gemeinsame Referenzen)



List und clone()



```
public void listKopieanalyse(){
  EinUndAusgabe io = new EinUndAusgabe();
  ArrayList<Linie> l1 = new ArrayList<Linie>();
  l1.add(new Linie(new Punkt(0,0), new Punkt(3,3)));
  ArrayList<Linie> l2 = (ArrayList<Linie>)l1.clone();
  io.ausgeben("l1 == l2: " + (l1 == l2) + "\n");
  io.ausgeben("l1(0) == l2(0): "
              + (l1.get(0) == l2.get(0)) + "\n");
}
```

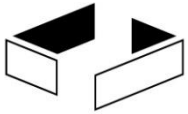
```
l1 == l2: false
l1(0) == l2(0): true
```

Beispiel

Video

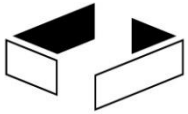
abstrakte Klasse

Fallstudie: Arenenkampf



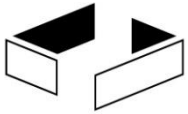
- In einer Arena sollen verschiedene kämpfende Personen verschiedener Clans gegeneinander antreten können. Dabei soll die spielende Person sich eine Person wählen und gegen eine Gruppe anderer kämpfenden Personen antreten. Jede kämpfende Person wird durch Namen, Geschick und Stärke charakterisiert. Ein Kampf findet jeweils Eins-gegen-Eins statt, wobei sich beide (nacheinander) gegenseitig attackieren und versuchen die Attacke abzuwehren. Jede kämpfende Person hat eine Anzahl von Gesundheitspunkten die, nach einer Attacke sinken kann. Ist die Punktzahl nicht mehr positiv, scheidet die kämpfende Person aus. Die spielende Person kämpft nacheinander gegen die Elemente der generischen Gruppe. Ziel ist es, alle anderen kämpfenden Personen zu besiegen.

Kaempfend (1/3) - Modellierung



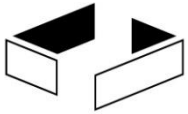
```
public class Kaempfend {
  protected String name;
  protected int gesundheit;
  protected int staerke;
  protected int geschick;

  public Kaempfend(){
  }

  public Kaempfend(String name, int gesundheit
                      , int staerke, int geschick) {
    this.name = name;
    this.gesundheit = gesundheit;
    this.staerke = staerke;
    this.geschick = geschick;
  }
```

Kaempfend (2/3) – übliche get- und set-Methoden



```
public int getGesundheit() { return this.gesundheit;}
public void setGesundheit(int gesundheit) {
  this.gesundheit = gesundheit;
}
public int getStaerke() { return this.staerke;}
public void setStaerke(int staerke) {
  this.staerke = staerke;
}
public int getGeschick() { return this.geschick; }
public void setGeschick(int geschick) {
  this.geschick = geschick;
}
public String getName() { return this.name;}
public void setName(String name) {
  this.name = name;
}
```

Kaempfend (3/3) – Kernfunktionalität



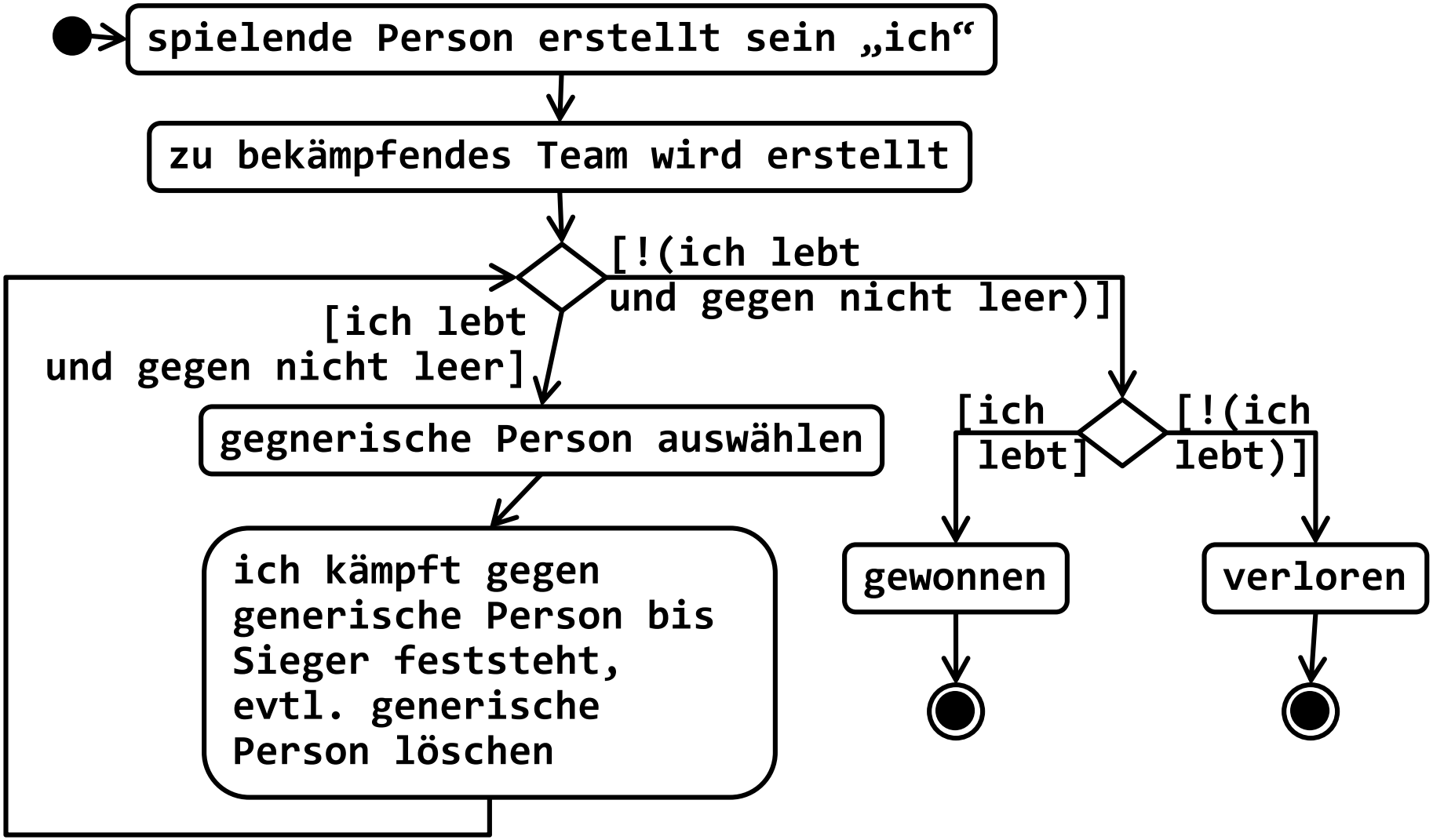
- Angreifen: zufälliger Schadenswert wird berechnet

```
public int angreifen(){
  EinUndAusgabe io = new EinUndAusgabe();
  return   io.zufall(2, this.staerke)
         * io.zufall(2, this.geschick);
}
```

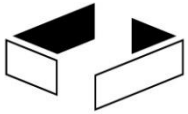
- Angegriffen werden: Angriff erfolgt mit einem Schadenswert, der noch zufällig abgemildert werden kann

```
public int einstecken(int haue){
  EinUndAusgabe io = new EinUndAusgabe();
  int ergebnis = io.zufall(0,haue);
  this.gesundheit = this.gesundheit - ergebnis;
  return ergebnis;
}
}
```

Arena - Konzept



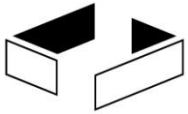
Arena – Realisierung (1/4)



```
import java.util.ArrayList;

public class Arena {
  private ArrayList<Kaempfend> gegen
                               = new ArrayList<Kaempfend>();
  private Kaempfend ich;

  public Kaempfend kaempfendErstellen(){
    int punkte = 30;
    EinUndAusgabe io = new EinUndAusgabe();
    io.ausgeben("Teilen Sie " + punkte
        + " in Kraft und Geschick auf:\n"
        + "Kraft (1-"+ (punkte - 1) + "): ");
    int kraft = 0;
    while (kraft < 1 || kraft > punkte - 1){
      kraft = io.leseInteger();
    }
    return new Kaempfend("Isch", 500, kraft, 30 - kraft);
  }
```

spielende Person erstellt sein „ich“

Arena – Realisierung (2/4)



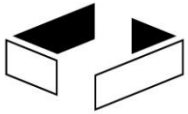
```
public void gegenErstellen() {
  for (int i = 1; i <= 4; i = i + 1) {
    this.gegen.add(new Kaempfend("Andy"+i
                                 , i * 100, i*3, i*3));
  }
}
```

zu bekaempfendes Team wird erstellt

```
public Kaempfend kaempfen(Kaempfend k1, Kaempfend k2) {
  while (k1.getGesundheit() > 0 && k2.getGesundheit() > 0) {
    this.einsHautZwei(k1, k2);
    if (k2.getGesundheit() > 0) {
      this.einsHautZwei(k2, k1);
    }
  }
  if (k1.getGesundheit() >= 0){
    return k1;
  }
  return k2;
}
```

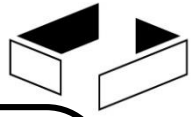
Kaempfend1 kämpft gegen Kaempfend2 bis Sieger feststeht

Arena – Realisierung (3/4)



```
private void einsHautZwei(Kaempfend k1, Kaempfend k2) {
  EinUndAusgabe io = new EinUndAusgabe();
  io.ausgeben(k1.getName() + " haut zu.\n");
  int schaden = k2.einstecken(k1.angreifen());
  io.ausgeben(k2.getName() + " verliert " + schaden
              + " Punkte.\n");
  io.ausgeben(k1.getName() + " hat " + k1.getGesundheit()
              + " Punkte, "
              + k2.getName() + " hat " + k2.getGesundheit()
              + " Punkte.\n*****\n");
}
```

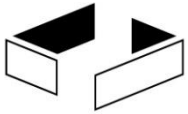
Arena – Realisierung (4/4)



```
public void kampfOrganisieren() {
  EinUndAusgabe io = new EinUndAusgabe();
  this.ich = this.kaempfendErstellen();
  this.gegenErstellen();
  while (this.ich.getGesundheit() > 0
        && this.gegen.size() > 0) {
    Kaempfend aktuell = this.gegen.get(0);
    Kaempfend sieger = this.kaempfen(this.ich, aktuell);
    if (this.ich.equals(sieger)) {
      this.gegen.remove(0);
    }
  }
  if (this.ich.getGesundheit() > 0) {
    io.ausgeben("Sie haben glorreich gewonnen.\n");
  } else {
    io.ausgeben("Sie haben schmachvoll verloren.\n");
  }
}
```

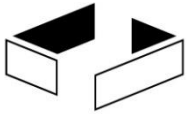
ich kämpft gegen generische Person bis Sieger feststeht, evtl. generische Person löschen

Ausschnitt aus Kampfprotokoll

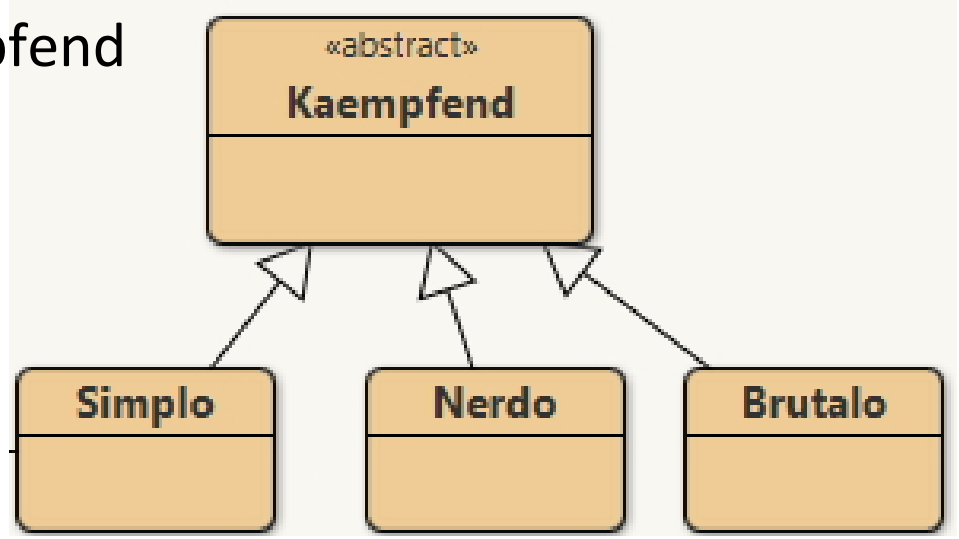


```
Teilen Sie 30 in Kraft und Geschick auf:
Kraft (1-29): 13
Isch haut zu.
Andy1 verliert 154 Punkte.
Isch hat 500 Punkte, Andy1 hat -54 Punkte.
*****
Isch haut zu.
Andy2 verliert 25 Punkte.
Isch hat 500 Punkte, Andy2 hat 175 Punkte.
*****
...
*****
Andy4 haut zu.
Isch verliert 21 Punkte.
Andy4 hat 19 Punkte, Isch hat -12 Punkte.
*****
Sie haben schmachvoll verloren.
```

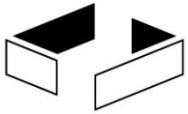
Inkrement: neue Kaempfendarten (Clans)



- Erweiterung: Es sollen unterschiedliche kämpfende Personen mit verschiedenen Angriffs- und Abwehrverhalten spezifiziert werden können.
- Ansatz 1 (machbar): neue Kaempfend-Klassen erben von Kaempfend und überschreiben angreifen() und einstecken()
- logisches Problem: Die aktuell vorhandenen Implementierungen der beiden Methoden stehen auch nur für eine spezielle Art von Kaempfend
- Lösung: Abstrakte Klasse

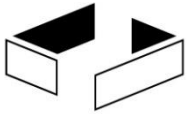


Idee: Abstrakte Klasse



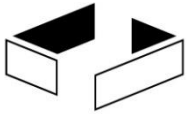
- abstrakte Klassen sind dazu konzipiert, dass von ihnen geerbt werden muss
- abstrakte Klassen enthalten Teilimplementierungen, die vererbt werden sollen (trotzdem überschreibbar)
- Methoden, die eine erbende Klasse realisieren muss (damit wird dynamische Polymorphie möglich) werden als **`abstract`** markiert und haben keine Implementierung
- Klasse selbst wird auch als **`abstract`** markiert
- von abstrakter Klasse erbende Klasse realisiert entweder alle abstrakten Methoden der Oberklasse oder ist selber abstrakt
- von abstrakten Klassen können *keine Objekte erzeugt werden*
- abstrakte Klasse aber als Typ für Variablen verwendbar

Abstrakt Kaempfend



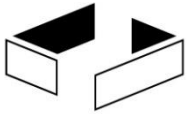
```
public abstract class Kaempfend {
  protected String name;
  protected int gesundheit;
  protected int staerke;
  protected int geschick;

  public Kaempfend(){}

  public Kaempfend(String name, int gesundheit
                      , int staerke, int geschick) {
    this.name = name;
    this.gesundheit = gesundheit;
    this.staerke = staerke;
    this.geschick = geschick;
  }

  public abstract int angreifen();

  public abstract int einstecken(int haue);

  // folgen normale get- und set-Implementierungen
}
```

Beispielrealisierung 1 – von vorher übernommen



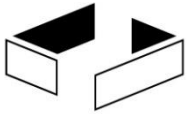
```
public class Simplo extends Kaempfend { // aus alter Klasse
  public Simplo() {}

  public Simplo(String name, int gesundheit
              , int staerke, int geschick) {
    super(name, gesundheit, staerke, geschick);
  }

  @Override
  public int angreifen(){
    EinUndAusgabe io = new EinUndAusgabe();
    return io.zufall(2, super.staerke)
         * io.zufall(2, super.geschick);
  }

  @Override
  public int einstecken(int haue){
    EinUndAusgabe io = new EinUndAusgabe();
    int ergebnis = io.zufall(0, haue);
    super.gesundheit = super.gesundheit - ergebnis;
    return ergebnis;
  }
}
```

Beispielrealisierung 2 – schwach, kann einstecken



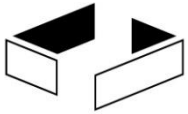
```
public class Nerdo extends Kaempfend {
  public Nerdo() {}

  public Nerdo(String name, int gesundheit,
               int staerke, int geschick) {
    super(name, gesundheit, staerke, geschick);
  }

  @Override
  public int angreifen(){
    EinUndAusgabe io = new EinUndAusgabe();
    return io.zufall(2, super.staerke)
         * io.zufall(2, super.geschick)/3;
  }

  @Override
  public int einstecken(int haue){
    EinUndAusgabe io = new EinUndAusgabe();
    int ergebnis = io.zufall(0,haue/3);
    super.gesundheit = super.gesundheit - ergebnis;
    return ergebnis;
  }
}
```

Beispielrealisierung 3 – stark, steckt nicht gut ein



```
public class Brutalo extends Kaempfend {
  public Brutalo() {}

  public Brutalo(String name, int gesundheit,
                 int staerke, int geschick) {
    super(name, gesundheit, staerke, geschick);
  }

  @Override
  public int angreifen(){
    EinUndAusgabe io = new EinUndAusgabe();
    return super.staerke * io.zufall(3, super.geschick);
  }

  @Override
  public int einstecken(int haue){
    EinUndAusgabe io = new EinUndAusgabe();
    int ergebnis = io.zufall(haue/2,haue);
    super.gesundheit = super.gesundheit - ergebnis;
    return ergebnis;
  }

}
```

Notwendige Änderungen in Arena



- Erinnerung: new nicht für abstrakte Klassen erlaubt

```
return new Kaempfend("Isch", 500, kraft, 30 - kraft);
}
```

Kaempfend is abstract; cannot be instantiated

```
return new Simplo("Isch", 500, kraft, 30-kraft);
```

bleibt: ArrayList<Kaempfend> gegen, aber:

```
public void gegenErstellen(){
  this.gegen.add(new Nerdo("Ulf", 100, 3, 3));
  this.gegen.add(new Simplo("Uta", 200, 6, 6));
  this.gegen.add(new Simplo("Urs", 300, 9, 9));
  this.gegen.add(new Brutalo("Ute", 400, 12, 12));
}
```

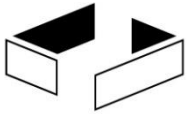
Beispiel

Beispiel

Video

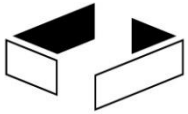
Interface

Völlig Abstrakte Klassen



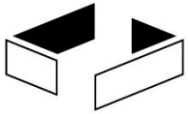
- wichtig bei Nutzung der dynamischen Polymorphie sind die "gemeinsame Herkunft" (d. h. beerbte Klasse) und die in allen Klassen nutzbaren Methoden
- nächster Schritt ist die vollständig abstrakte Klasse ohne Objektvariablen und mit nur abstrakten Methoden
- dieses wird in Java auch Interface genannt
- ein Interface I wird durch **`implements I`** (statt **`extends I`**) realisiert
- Interfaces können andere Interfaces beerben (extends)

Beispiel: Interface

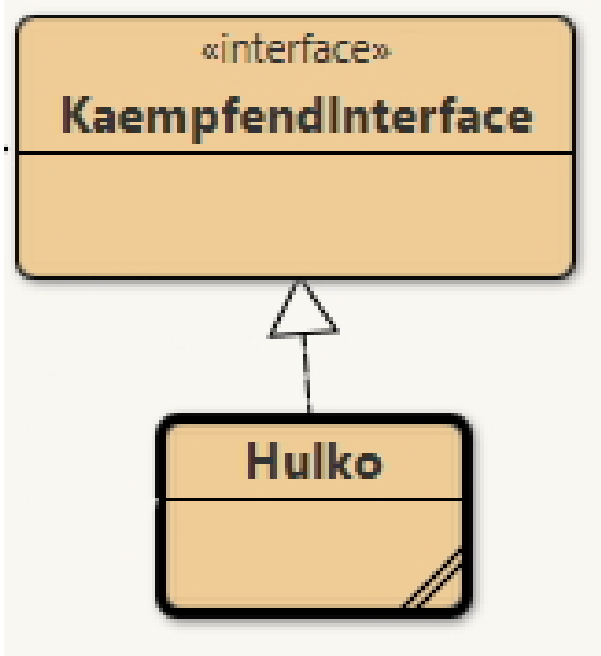


```
public interface KaempfendInterface{
  public int angreifen();
  public int einstecken(int haue);
  public int getGesundheit();
  public String getName();
}
// public bei Methoden koennte weggelassen werden,
// Semantik aendert sich dadurch nicht
```

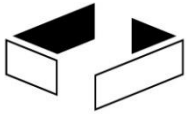
Beispiel: Realisierung eines Interface (1/2)



```
public class Hulko implements KaempfendInterface {

  private String name;
  private int gesundheit;

  public Hulko(String name, int gesundheit) {
    this.name = name;
    this.gesundheit = gesundheit;
  }

  @Override
  public int angreifen() {
    return 100;
  }
```



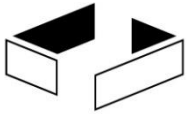
Beispiel: Realisierung eines Interface (2/2)



```
  @Override
  public int einstecken(int haue) {
    this.gesundheit = this.gesundheit - haue/10;
    return haue/10;
  }

  @Override
  public int getGesundheit() {
    return this.gesundheit;
  }

  @Override
  public String getName() {
    return this.name;
  }
}
```

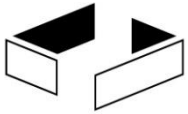
- zur Nutzung in Arena Kaempfend durch KaempfendInterface ersetzen; oder auch Kaempfend implements Interface

Design by Contract



- Interfaces zentrale Idee zur Realisierung von Software in Teams
- Ansatz: Zunächst Modellierung der Klassen und dann werden Interfaces für diese Klassen definiert
- Vertrag zwischen Nutzenden des Interfaces und entwickelnden Personen
- Nutzende Person weiß, dass es eine Klasse gibt, die das Interface realisiert
- Entwickelnde Person verpflichtet sich, das Interface zu realisieren; wie es gemacht wird, geht nutzende Person nichts an
- zum Testen kann nutzende Person des Interfaces minimale Implementierung des Interfaces selbst vornehmen
- grundsätzlich gilt: es reichen nicht nur Methodennamen; Vor- und Nachbedingungen sind zu definieren

Vervollständigtes Interface als Vertrag

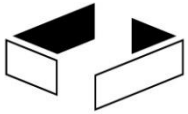


```
public interface KaempfendInterface{
  /** Die Methode gibt einen Angriffswert des Objekts
   * zurück, der abhaengig von der momentanen Staerke ist.
   * @return ein nicht-negativer Wert kleiner-gleich 250 */
  public int angreifen();
  /** Methode berechnet den Effekt eines erhaltenen
   * Angriffs der Staerke haue. Es wird erwartet, dass der
   * Gesundheitswert negativ beeinflusst wird.
   * @param haue nicht negativer Wert des empfangenen Schlags
   * @return Staerke der Auswirkung auf das eigene Objekt,
   *         Wert liegt zwischen 0 und haue */
  public int einstecken(int haue);
  /** Gibt aktuellen Gesundheitswert zurueck.
   * @return aktueller Gesundheitswert */
  public int getGesundheit();
  /** Gibt Namen der kaempfenden Person zurueck.
   * @return Name der kaempfenden Person  */
  public String getName();
}
```

Video

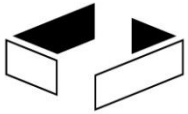
Kommentare

Einschub Dokumentation (1/5)



- vorherige Dokumentationsform nicht zufällig
- erlaubt Programmdokumentation aus dem Quellcode zu generieren
- Generierung erfolgt durch Java-Hilfsprogramm javadoc
- javadoc hat viele Parameter mit denen z. B. das Layout oder der Detaillierungsgrad der Information (sollen z. B. Objektvariablen in der Dokumentation sichtbar sein?) eingestellt werden kann
- generierte Dokumentation besteht typischerweise aus einem Übersichtsteil und der Detaildokumentation
- hier wird nur Grundprinzip erklärt
- Ähnliches Konzept für weitere Sprachen: Doxygen http://www.doxygen.nl/

Einschub Dokumentation (2/5)



KaempfendInterface ×

Übersetzen | Rückgängig | Ausschneiden | Kopieren | Einfügen | Suchen... | Schließen | Dokumentation

Interface KaempfendInterface

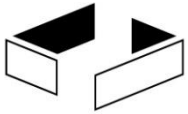
```
public interface KaempfendInterface
```

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method	Description
int	angreifen()	Die Methode gibt einen Angriffswert des Objekts zurueck, der Abhaengig von der momantanen Staerke ist.
int	einstecken(int haue)	Methode berechnet den Effekte eines erhaltenen Angriffs der Staerke haue.
int	getGesundheit()	Gibt aktuellen Gesundheitswert zurueck.
String	getName()	Gibt Namen der kaempfenden Person zurueck.

Einschub Dokumentation (3/5)



KaempfendInterface ×

Übersetzen | Rückgängig | Ausschneiden | Kopieren | Einfügen | Suchen... | Schließen | Dokumentation

Method Details

angreifen

```
int angreifen()
```

Die Methode gibt einen Angriffswert des Objekts zurueck, der Abhaengig von der momantanen Staerke ist.

Returns:

ein nicht-negativer Wert kleiner-gleich 250

einstecken

```
int einstecken(int haue)
```

Methode berechnet den Effekte eines erhaltenen Angriffs der Staerke haue. Es wird erwartet, dass der Gesundheitswert negativ beeinflusst wird.

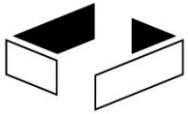
Parameters:

`haue` - nicht negativer Wert des empfangenen Schlags

Returns:

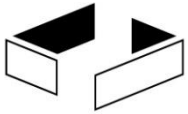
Staerke der Auswirkung auf das eigene Objekt, Wert liegt zwischen 0 und haue

Einschub Dokumentation (4/5)



- Grundsätzlich ist jedes Sprachelement (Klasse, Objektvariable, Methode) vor ihrer Deklaration zu dokumentieren
- Javadoc-Kommentar beginnt mit /**
- Text bis zum ersten Punkt wird in die Zusammenfassung aufgenommen, Rest steht bei Details
- Jeder Übergabeparameter ist mit @param zu dokumentieren
- Gibt es Methodenrückgabe, ist diese mit @return zu dokumentieren
- In der Dokumentation können HTML4-Tags genutzt werden

Einschub Dokumentation (5/5)



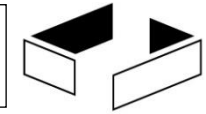
- sorgfältige Dokumentation ist zentrale Aufgabe der Entwicklung
- man schreibt so, dass ein anderer entwicklende Personen das Programm nutzen, aber auch weiterentwickeln können
- Textfragmente können durchaus redundante Informationen enthalten, da sie oft nicht zusammenhängend gelesen werden
- wann entsteht Dokumentation (keine klare Antwort):
  - nach der Programmerstellung (keine aufwändige Aktualisierung; nicht mehr alle Details im Kopf)
  - nach Methodenerstellung (alle Details der Erstellung noch bekannt; muss bei Änderungen angepasst werden)
  - vor der Methodenerstellung (man kann sich so eine Aufgabenstellung definieren; Änderungen sind aufwändig)

Beispiel

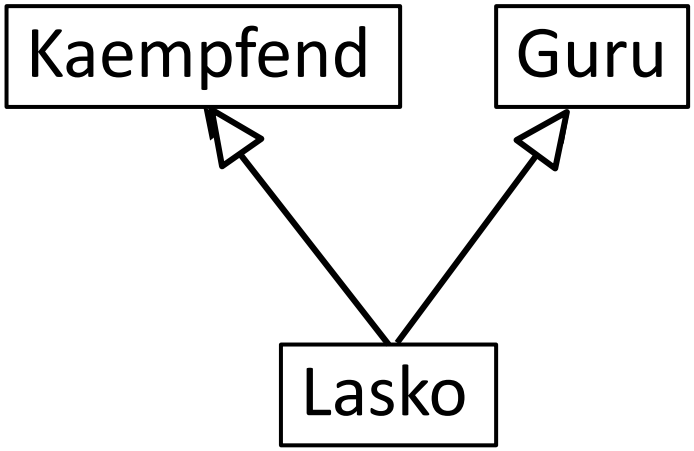
Video

Mehrfachvererbung

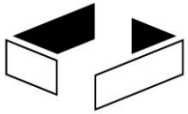
Mehrfachvererbung



- einige objektorientierte Sprachen unterstützen Mehrfachvererbung
- Problem: wenn Methode mit gleicher Signatur von verschiedenen Klassen geerbt (Notlösung: man muss Klasse angeben, deren Methode genutzt werden soll)
- Auflösung dynamischer Polymorphie machbar, aber aufwändig

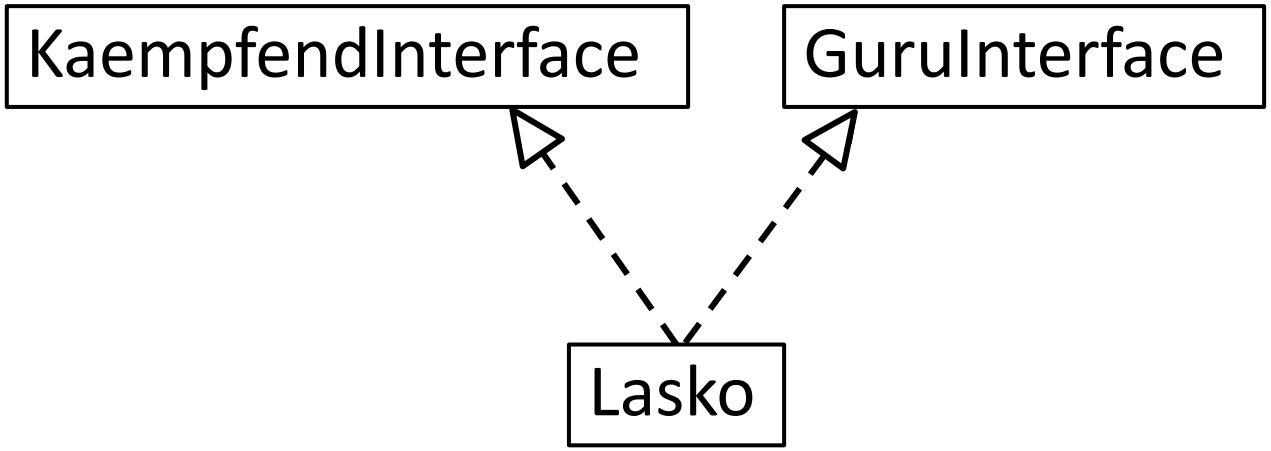


Mehrfachvererbung mit Interfaces

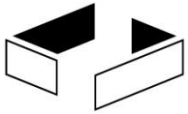


- Java erlaubt es, dass eine Klasse mehrere Interfaces realisiert

```
public interface GuruInterface {
  public String weisheit(int nr);
}
```

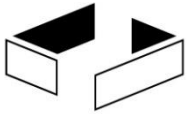


Beispiel für Realisierung mehrerer Interfaces



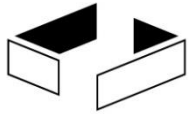
```
public class Lasko implements KaempfendInterface
                              , GuruInterface {
  @Override
  public String weisheit(int nr) {
    if(nr == 42){
      return "Das ist die Antwort";
    }
    return "Suche die Frage";
  }
  @Override
  public int angreifen() {return 0;}
  @Override
  public int einstecken(int haue) {return 0;}
  @Override
  public int getGesundheit() {return 1;}
  @Override
  public String getName() {return "Lasko";}
}
```

Anmerkungen zur Vererbung

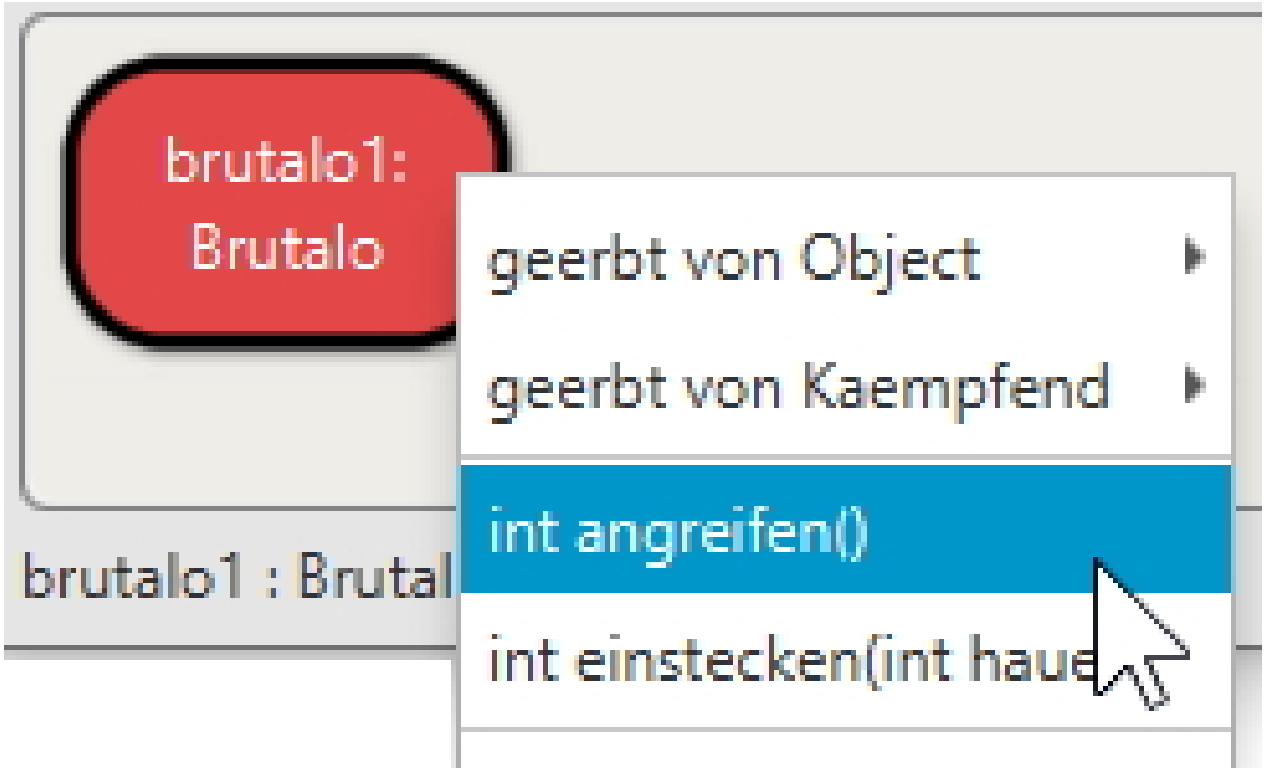


- Vererbung ist ein wichtiger Ansatz, damit ähnlicher Code nicht doppelt geschrieben werden muss
- Vererbung spielt beim Aufbau von Programmbibliotheken eine wichtige Rolle (Übergang vom generellen Konzept bis zur Speziallösung)
- Aber, Vererbung ist kein „Muss“, sie ergibt sich in der Programmentwicklung als sinnvolle Vereinfachung
- OO-Anfänger*innen suchen oft krampfhaft nach Möglichkeiten zur Nutzung von Vererbung, das Ergebnis sind häufig schlecht strukturierte Programme

Doch Mehrfachvererbung? nur immer von Object



- Bei der Ausführung von Methoden fällt frühzeitig auf, dass jede Klasse eine Vererbung nutzt, erbende Klassen sogar zwei



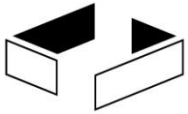
Beispiel

Beispiel

Video

Array

Idee von Arrays

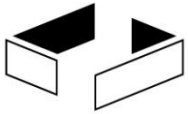


- Nutzung der bisher bekannten Sammlungen kann recht aufwändig sein, da z. B. beim Hinzufügen immer neuer Platz im Speicher gesucht werden soll
- Effizienter sind Objekte, die einmal gespeichert werden und ihre Größe nicht mehr ändern
- ***praktisch einsetzbar, wenn man die maximale Anzahl von Objekten kennt, die alle den gleichen Typen haben und deren Anzahl nicht allzu groß ist***
- Zugehöriger Datentyp wird Array (Feld, Reihung) genannt
- besondere Form der Typangabe

  <Typ>[] <Variablenname> ; oder äquivalent

  <Typ> <Variablenname> [];
- Arrays können wie Sammlungen iteriert werden, Länge allerdings durch <Variablenname>.length bestimmbar

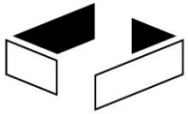
Varianten der Initialisierung



```
public class Analyse {

  private String[] wo1 = {"Mo","Di","Mi","Do","Fr","Sa","So"};
  private String[] wo2 = new String[7]; //null-Werte
  private String[] wo3;

  public void initialisieren(){
    //this.wo3 = {"42"}  geht nicht
    this.wo3 = this.wo2;
    EinUndAusgabe io = new EinUndAusgabe();
    for(int i = 0; i < this.wo1.length; i = i + 1){
      this.wo2[i] = this.wo1[i];
    }
    for(int i = 0; i < this.wo3.length; i = i + 1){
      io.ausgeben(this.wo3[i]);
    }
    io.ausgeben("\n");
  }
}
```

```
MoDiMiDoFrSaSo
```

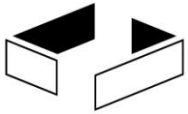
Beispiel: Klasse für einen Lotto-Tipp (1/4)



- sinnvoll, Arrays mit eigener Bedeutung in Klassen zu kapseln

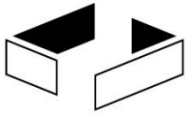
```
public class Tipp {

  private int[] werte = new int[6];

  public Tipp() {
    EinUndAusgabe io = new EinUndAusgabe();
    int i = 0;
    while (i < this.werte.length) {
      int kugel = io.zufall(1, 49);
      if (this.kommtNichtVor(kugel, i)) {
        this.werte[i] = kugel;
        i = i + 1;
      }
    }
  }
```

Beispiel: Klasse für einen Lotto-Tipp (2/4)



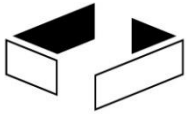
```
public boolean kommtNichtVor(int wert, int position){
  for (int i = 0; i < position; i = i + 1) {
    if (this.werte[i] == wert) {
      return false;
    }
  }
  return true;
}

public int[] getWerte() {
  return this.werte;
}

public void setWerte(int[] werte) {
  this.werte = werte;
}
```

Beispiel: Klasse für einen Lotto-Tipp (3/4)



```
public boolean enthaelt(int wert) {
  for (int i : this.werte) {
    if (i == wert) {
      return true;
    }
  }
  return false;
}

@Override
public String toString() {
  String ergebnis = "[ ";
  for (int i : this.werte) {
    ergebnis = ergebnis + i + " ";
  }
  return ergebnis + "]";
}
```

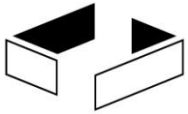
Beispiel: Klasse für einen Lotto-Tipp (4/4)



- Analyse der Übereinstimmungen von zwei Tipps
- Ansatz: Jedes Element des ersten Tipps wird mit jedem Element des zweiten Tipps verglichen (gibt keine Doppelten)

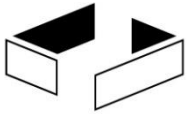
```
  public int gemeinsam(Tipp tip) {
    int ergebnis = 0;
    for (int i : this.werte) {
      for (int j : tip.getWerte()) {
        if (i == j) {
          ergebnis = ergebnis + 1;
        }
      }
    }
    return ergebnis;
  }
}
```

Testen von Tipp (1/2)



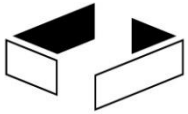
```
public class TippTest {
  private Tipp t1 = new Tipp();
  private Tipp t2 = new Tipp();

  @Test
  public void testUnterschiedlicheWerte(){
    for(int i = 0; i < 100; i = i + 1){
      this.unterschiedlich(new Tipp());
    }
  }

  private void unterschiedlich(Tipp tip){
    int inhalt[] = tip.getWerte();
    for(int i = 0; i < inhalt.length - 1; i = i + 1){
      Assertions.assertTrue(inhalt[i] > 0 && inhalt[i] < 50);
      for(int j = i + 1; j < inhalt.length; j = j + 1){
        Assertions.assertTrue(!(inhalt[i] == inhalt[j]));
      }
    }
  }
```

Testen von Tipp (2/2) - Ausschnitt



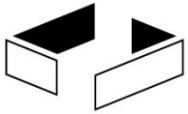
```
  @Test
  public void testNuller(){
    this.t1.setWerte(new int[]{1,2,3,4,5,6});
    this.t2.setWerte(new int[]{11,12,13,14,15,16});
    Assertions.assertTrue(this.t1.gemeinsam(this.t2) == 0);
  }

  @Test
  public void testFuenfer(){
    this.t1.setWerte(new int[]{4,12,43,24,5,36});
    this.t2.setWerte(new int[]{31,12,43,4,24,36});
    Assertions.assertTrue(this.t1.gemeinsam(this.t2) == 5);
  }

  @Test
  public void testSechser(){
    this.t1.setWerte(new int[]{4,12,43,24,36,31});
    this.t2.setWerte(new int[]{31,12,43,4,24,36});
    Assertions.assertTrue(this.t1.gemeinsam(this.t2) == 6);
  }
}
```

Analyse von Häufigkeiten (1/3)



```
public void haeufigkeiten(int anzahl){ // in Analyse
  Tipp mein = new Tipp();
  int[] ergebnis = { 0, 0, 0, 0, 0, 0, 0 }; // wie oft 0-6er
  int[] gezogen = new int[49]; // einzelne Zahl wie oft
  for (int i = 0; i < gezogen.length; i = i + 1) {
    gezogen[i] = 0;
  }
  this.losenUndAuswerten(mein, ergebnis, gezogen, anzahl);
  EinUndAusgabe io = new EinUndAusgabe();
  for(int i = 0; i < ergebnis.length; i = i + 1){
    io.ausgeben(i + "er: " + ergebnis[i] + "\n");
  }
  for(int i = 0; i < gezogen.length; i = i + 1){
    io.ausgeben((i + 1) + ":" + gezogen[i] + "  ");
    if((i + 1) % 6 == 0){
      io.ausgeben("\n");
    }
  }
}
```

Analyse von Häufigkeiten (2/3)



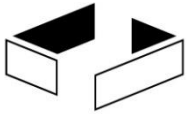
```
public void losenUndAuswerten(Tipp mein, int[] ergebnis
          , int[] gezogen, int anzahl) {
  for (int i = 0; i < anzahl; i = i + 1) {
    Tipp tmp = new Tipp();
    int treffer = mein.gemeinsam(tmp);
    ergebnis[treffer] = ergebnis[treffer] + 1;
    for (int j = 0; j < 49; j = j + 1) { // gibt Alternativen
      if (tmp.enthaelt(j + 1)) {
        gezogen[j] = gezogen[j] + 1;
      }
    }
  }
}
```

Analyse von Häufigkeiten (3/3) – 2000000 Versuche

- Dauer: 14 Sekunden

```
0er: 870573
1er: 828056
2er: 263832
3er: 35540
4er: 1967
5er: 32
6er: 0
1:244608  2:244572  3:245968  4:245208  5:244385  6:245379
7:244733  8:244578  9:243898  10:244732  11:245054  12:244219
13:244770  14:245695  15:244858  16:244865  17:244949  18:244964
19:244590  20:245296  21:244815  22:244602  23:244568  24:244466
25:245505  26:244264  27:244369  28:245617  29:245516  30:245340
31:245063  32:244694  33:245223  34:244721  35:245899  36:244103
37:244962  38:245284  39:245049  40:245308  41:244512  42:244984
43:244649  44:244848  45:244985  46:245573  47:244426  48:244670
49:244664
```

Mehrdimensionale Arrays 1

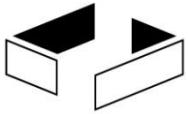


- Achtung niemals dadurch Klassen ersetzen! (schlechte Idee: jede studierende Person mit Array aus 6 Strings modellieren)

```
public void mehrdimensional1(){
   EinUndAusgabe io = new EinUndAusgabe();
   String[][] werte = {{"Mo", "Jo", "USA"}
                      ,{"Luc", "Lack", "FRA"}};
   for(String[] stud: werte){
     for(String dat: stud){
       io.ausgeben(dat + " ");
     }
     io.ausgeben("\n");
   }
 }
```

```
Mo Jo USA
Luc Lack FRA
```

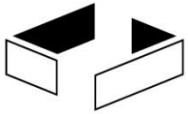
Mehrdimensionale Arrays 2



```
public void mehrdimensional2(){
  EinUndAusgabe io = new EinUndAusgabe();
  String[][][] all = {{{"Mo", "Jo", "USA", "NY"}
                       ,{"Luc", "Lack", "FRA"}}
                      ,{{"Sue", "Wue", "PRC"}
                       ,{"Sam", "Fox", "GB", "ENG"}}};
  for(String[][] sem: all){
    for(String[] stud: sem){
      for(String dat: stud){
        io.ausgeben(dat + " ");
      }
      io.ausgeben("\n");
    }
    io.ausgeben("*****\n");
  }
}
```

```
Mo Jo USA NY
Luc Lack FRA
*****
Sue Wue PRC
Sam Fox GB ENG
*****
```

Erinnerung: Arbeit mit Referenzen

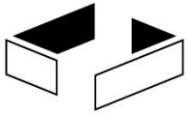


Ausgangspunkt

- Gegeben Klasse Punkt
  - mit Objektvariablen x, y
  - zugehörigen get- und set-Methoden
  - passender equals-Methode
  - und toString-Methode
- Klasse Interaktionsbrett u. a. zum Zeichnen von Linien

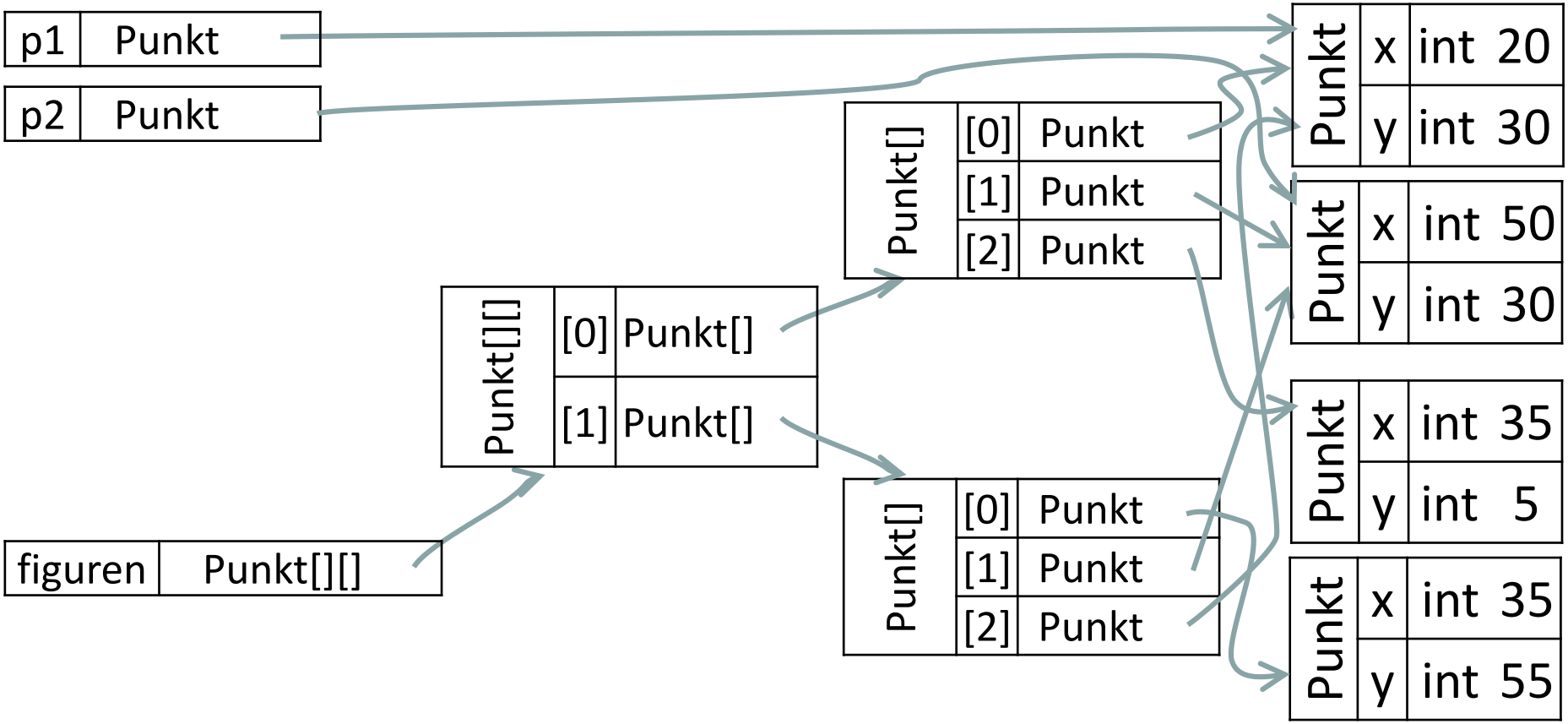
void	**neueLinie**(int x1, int y1, int x2, int y2) Methode zum Zeichnen einer neuen Linie.

Arrays und Referenzen (1/5)

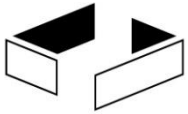


```
public class FigurMalerei{

  private Interaktionsbrett ib = new Interaktionsbrett();
  private Punkt p1 = new Punkt(20, 30);
  private Punkt p2 = new Punkt(50, 30);
  private Punkt[][] figuren = {
                    this.dreieck(p1, p2, new Punkt(35, 5))
                  , this.dreieck(new Punkt(35, 55), p2, p1)};

  public Punkt[] dreieck(Punkt p1, Punkt p2, Punkt p3){
      return new Punkt[]{p1, p2, p3};
  }
```

Ausgangssituation im Speicher

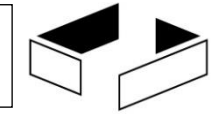


Arrays und Referenzen (2/5)

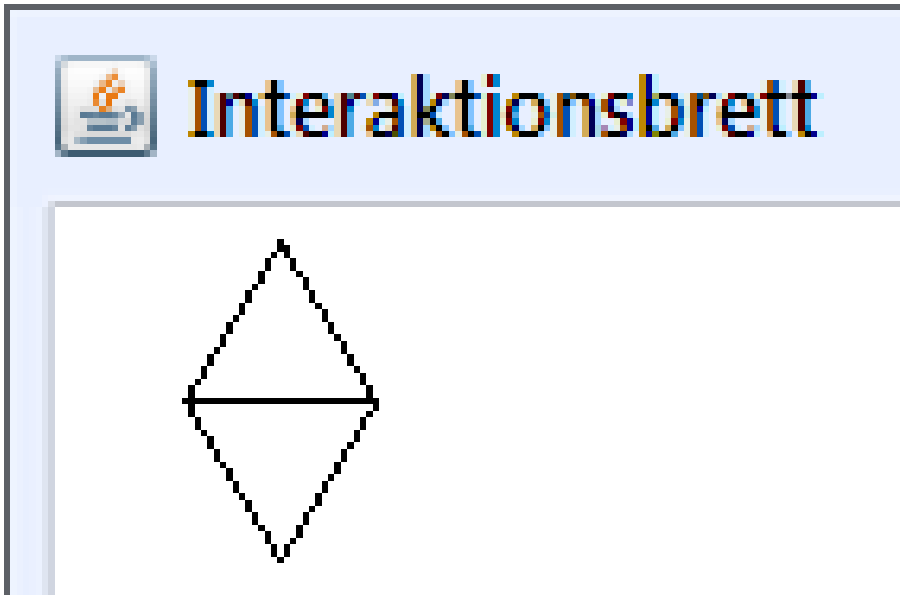


```
public void malen(Punkt[] punkte){
        for(int i = 0; i < punkte.length; i = i + 1){
            this.ib.neueLinie(punkte[i].getX(),
                punkte[i].getY(),
                punkte[(i+1) % punkte.length].getX(),
                punkte[(i+1) % punkte.length].getY());
        }
    }

    public void malen(){
        for(Punkt[] arr:this.figuren){
           this.malen(arr);
        }
    }
```

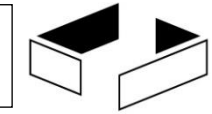
Arrays und Referenzen (3/5)



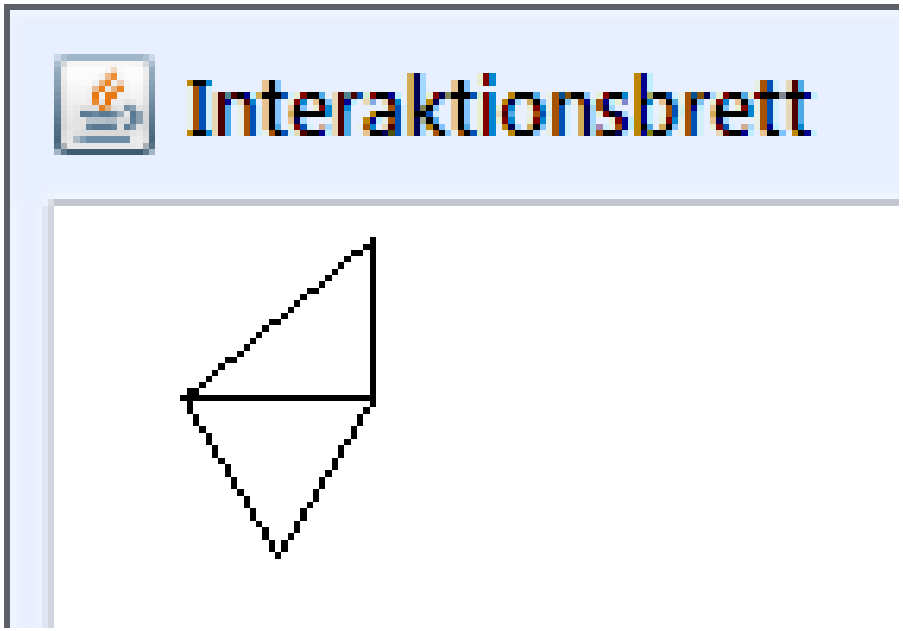
```
public void analyse1(){
    malen();
}
```



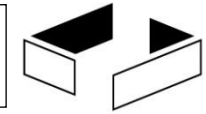
Arrays und Referenzen (4/5)



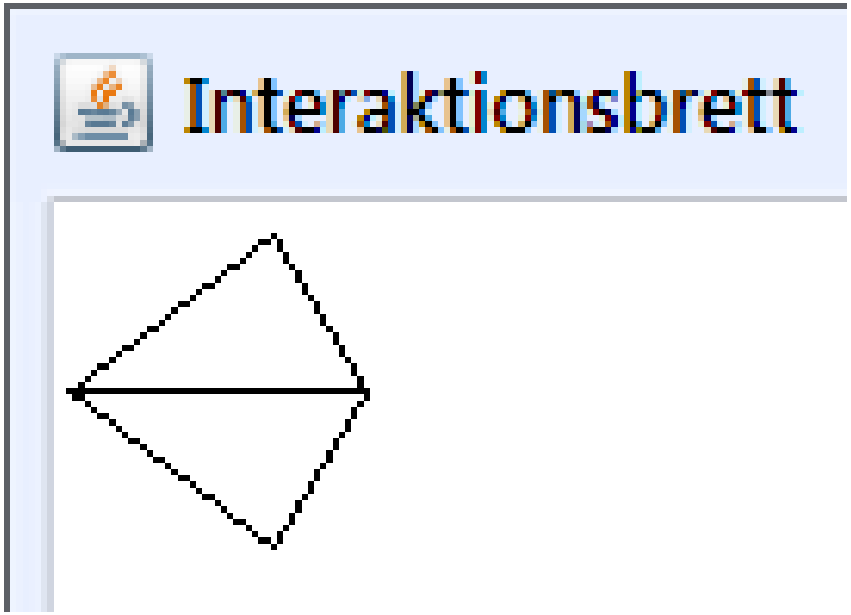
```
public void analyse2(){
    this.figuren[0][2].setX(50);
    malen();
}
```



Arrays und Referenzen (5/5)



```
public void analyse3(){
    this.p1.setX(2);
    malen();
}
```

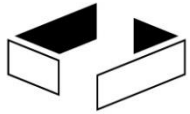


Beispiel

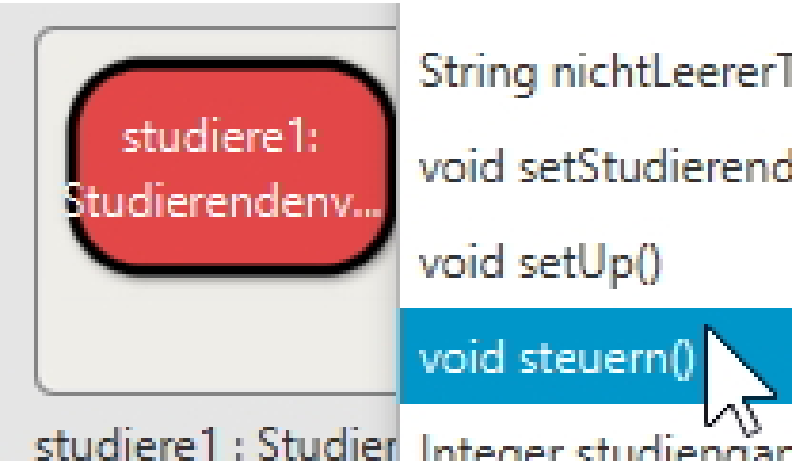
Video

Start von Java-Programmen

Bisher



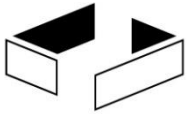
- In BlueJ wurde Objekt erzeugt und dann Methode ausgewählt



- alternativ könnte man jetzt direkt auf Klasse eine noch zu schreibende Klassenmethode ausführen
- Ansatz weiter gedacht: Klasse heißt ausführbar, wenn sie eine Methode mit folgender Signatur hat

```
public static void main(String[] arg){…
```

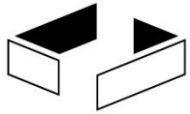
Nutzung einer Main-Klasse



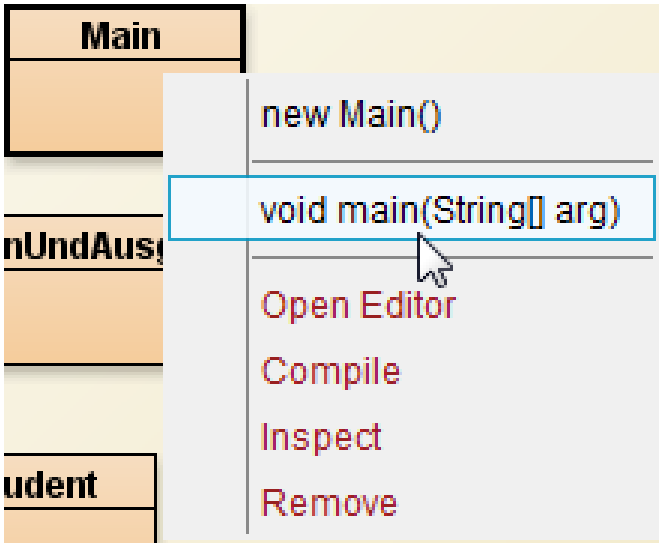
- sinnvoll ist es, eigene Klasse Main zu ergänzen, die ausschließlich zum Start des Programmes genutzt wird
- Erstellung notwendiger Objekte und Aufruf von Methoden, damit Objekte "sich kennen"
- Start einer Hauptmethode

```
public class Main{

  public static void main(String[] arg){
    Studierendenverwaltung s = new Studierendenverwaltung();
    s.steuern(); // ruft Dialog auf
  }
}
```

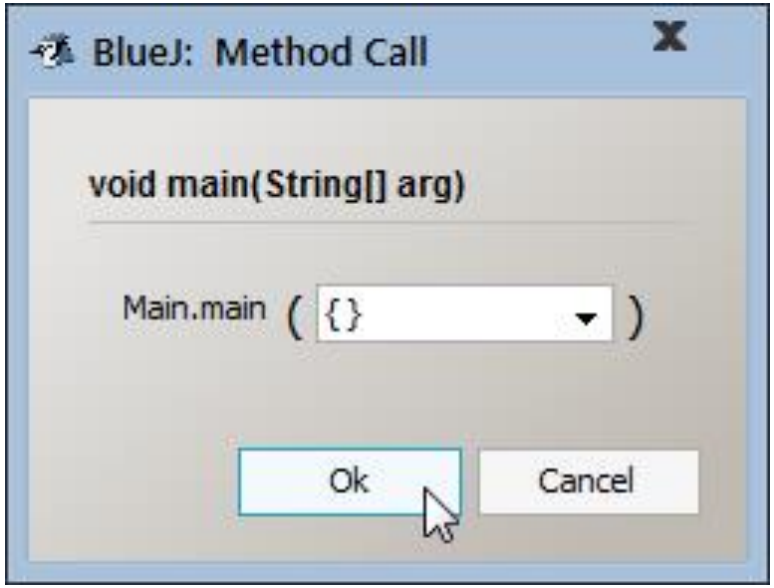
Aufruf von main in BlueJ



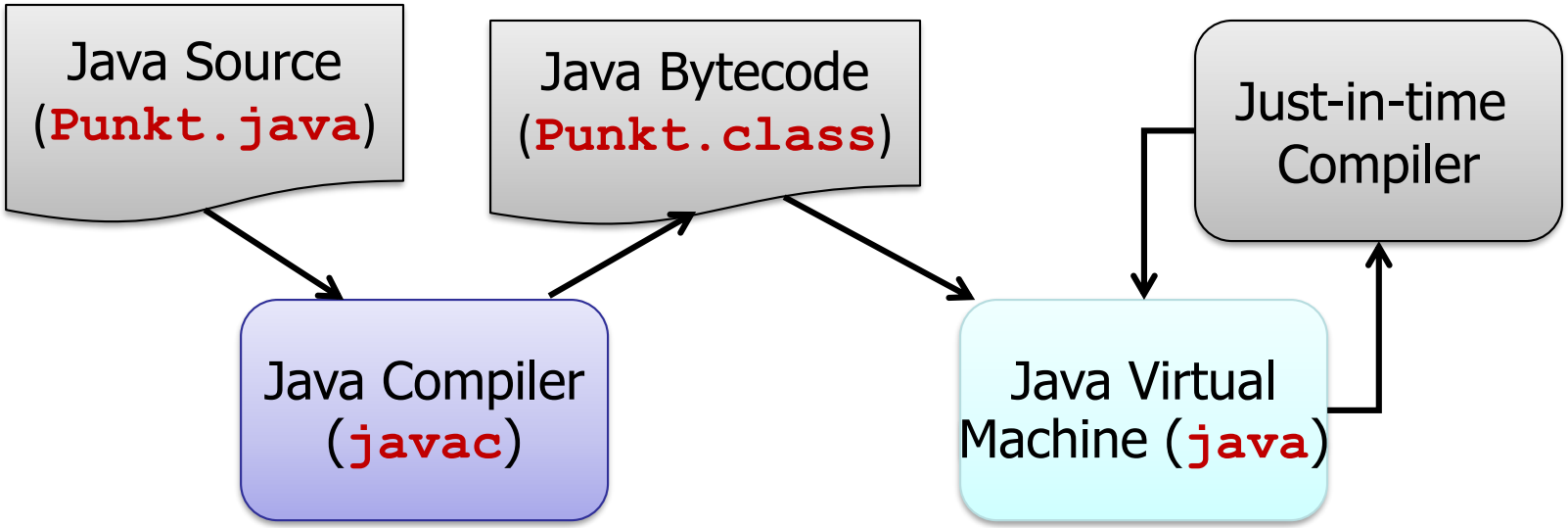
- Klassenmethode auswählen



- Array kann direkt eingegeben werden (meist ohne Bedeutung, dann leer lassen)

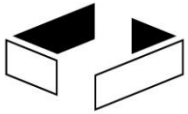


Vom Source Code zum ausführbaren Programm



- Source Code wird vom Compiler in Bytecode übersetzt
- Bytecode wird von JVM ausgeführt (d.h. interpretiert)
- Ggf. werden über den JIT-Compiler bestimmte Befehlssequenzen in nativen Code übersetzt

Kompilierung mit der Konsole - Ausgangssituation



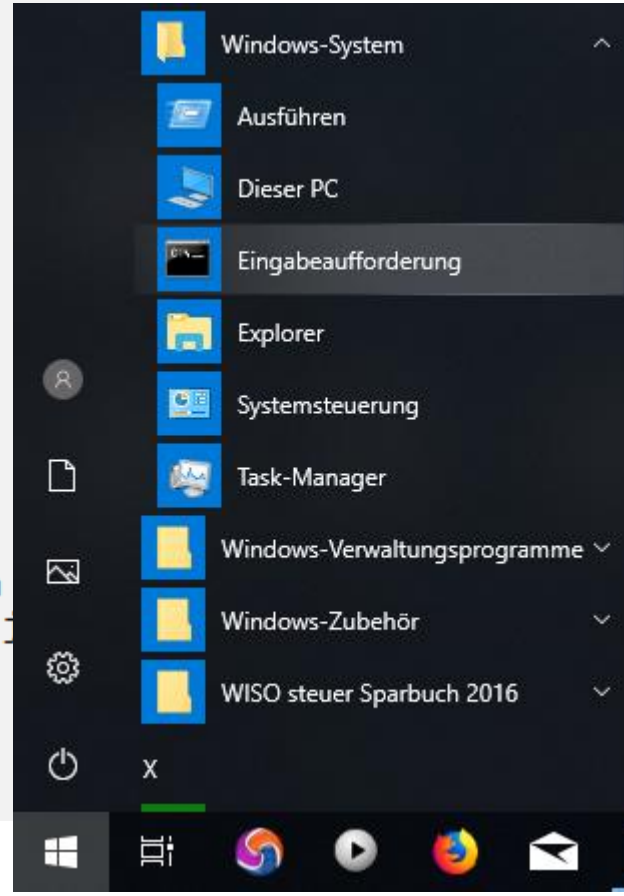
Konsolenfenster

```
Microsoft Windows [Version 10.0.19041.1110]
(c) Microsoft Corporation. Alle Rechte vorbehalten.

C:\kleukersSEU>f:

F:\>cd workspaces\BluejWork\bsp

F:\workspaces\BluejWork\bsp>dir
 Datenträger in Laufwerk F: ist F
 Volumeseriennummer: 7C52-E167

 Verzeichnis von F:\workspaces\BluejWork\bsp

21.07.2021  21:05    <DIR>          .
21.07.2021  21:05    <DIR>          ..
21.07.2021  11:48             3.780 Analyse.java
21.07.2021  11:28             1.284 Austauschstudierend.java
21.07.2021  11:28             5.220 EinUndAusgabe.java
21.07.2021  21:06               180 Main.java
21.07.2021  11:28             6.277 Studierend.java
21.07.2021  11:28            11.087 Studierendenverwaltung.java
21.07.2021  11:25             3.111 StudierendenverwaltungTest.j
               7 Datei(en),         30.939 Bytes
               2 Verzeichnis(se), 3.773.898.313.728 Bytes frei

F:\workspaces\BluejWork\bsp>
```

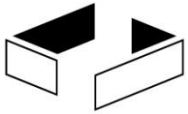
kleukersSEU

Name

StartKonsole.bat



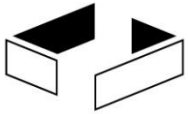
Kompilierung mit der Konsole - Kompilieren



Konsolenfenster

```
F:\workspaces\BluejWork\bsp>javac Main.java

F:\workspaces\BluejWork\bsp>dir
 Datenträger in Laufwerk F: ist F
 Volumeseriennummer: 7C52-E167

 Verzeichnis von F:\workspaces\BluejWork\bsp

21.07.2021  21:08    <DIR>          .
21.07.2021  21:08    <DIR>          ..
21.07.2021  11:48             3.780 Analyse.java
21.07.2021  21:08             1.711 Austauschstudierend.class
21.07.2021  11:28             1.284 Austauschstudierend.java
21.07.2021  21:08             2.159 EinUndAusgabe.class
21.07.2021  11:28             5.220 EinUndAusgabe.java
21.07.2021  21:08               326 Main.class
21.07.2021  21:06               180 Main.java
21.07.2021  21:08             3.939 Studierend.class
21.07.2021  11:28             6.277 Studierend.java
21.07.2021  21:08             7.563 Studierendenverwaltung.class
21.07.2021  11:28            11.087 Studierendenverwaltung.java
21.07.2021  11:25             3.111 StudierendenverwaltungTest.java
               12 Datei(en),         46.637 Bytes
                2 Verzeichnis(se), 3.773.898.072.064 Bytes frei

F:\workspaces\BluejWork\bsp>_
```

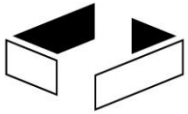
Kompilierung mit der Konsole - Ausführen



Konsolenfenster - java Main

```
F:\workspaces\BluejWork\bsp>java Main
Nächste Aktion:
 (0) Programm beenden
 (1) Neue studierende Person einfügen
 (2) Studierende Person mit Matrikelnummer suche
 (3) Studierendendaten ändern
 (4) Studierende pro Studiengang
 (5) Studierende Person löschen
 (6) Studierende eines Fachs löschen
 (7) Neuen Austauschstudi einfügen
 (8) Anzahl Austauschstudi ausgeben:
1
Vorname: Leila
Nachname: Schmidt
Studiengang: ITI
Geburtsjahr: 1999
Matrikelnummer: 424242
```

ausführbares Programm in BlueJ (1/4)



- Annahme: gibt Klasse Main mit main-Methode
- Packen des Programms in eine jar-Datei (vergleichbar einer zip-Datei mit einem bestimmten Aufbau)

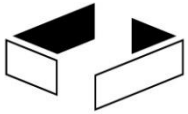
BlueJ: BeispielStudierendenlisteMitSpeichernundThrow

Project Edit Tools View Help

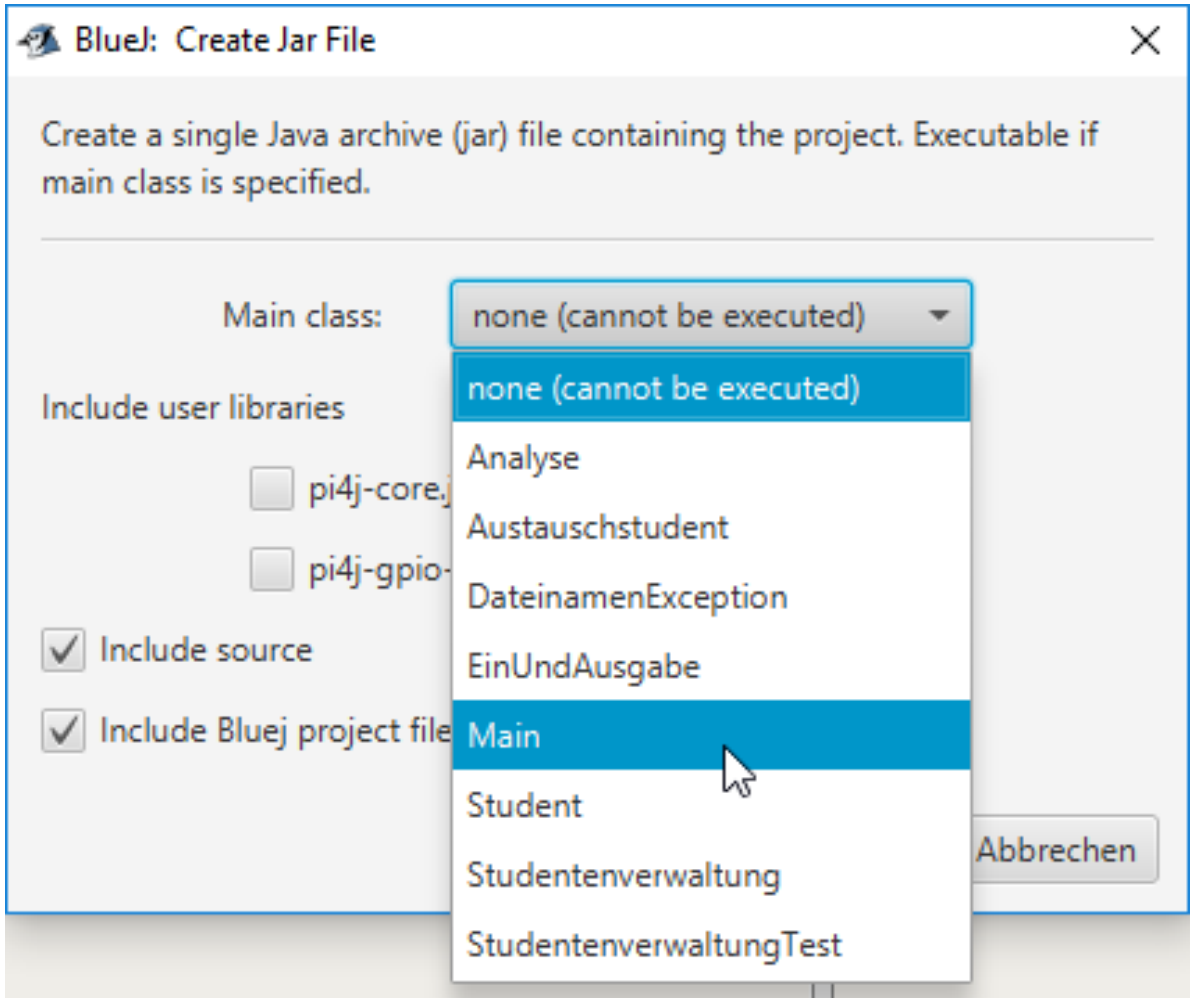
New Project...
Open Project... Ctrl+O
Open Recent
Open Non BlueJ...
Open ZIP/JAR...
Close Ctrl+W
Save Ctrl+S
Save As...
Import...
Create Jar File...
Print... Ctrl+P
Quit Ctrl+Q

Analyse
Stude
Main

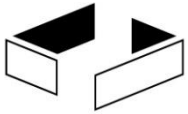
ausführbares Programm in BlueJ (2/4)



- Auswahl der Startklasse (könnte mehrere main geben)



ausführbares Programm in BlueJ (3/4)



- Speichern der jar-Datei

Specify name for jar file

« Volume (F:) › workspaces › BluejWork

"BluejWork" durchsuchen

Organisieren ▾ Neuer Ordner

Schnellzugriff
Desktop
Downloads
Dokumente

Name	Änderungsdatum	Typ	Größe
AufgabeArbeitsgruppenReferenzen	29.10.2018 11:51	Dateiordner	
AufgabeBasketballLoesung	21.11.2018 19:27	Dateiordner	
AufgabeBoolescheAusdrueckeAuswerten	31.10.2018 12:34	Dateiordner	

Dateiname: Studiverwaltung

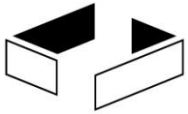
Dateityp: JAR file (*.jar)

Ordner ausblenden

Speichern Abbrechen

- jar-Datei unter jedem Betriebssystem nutzbar

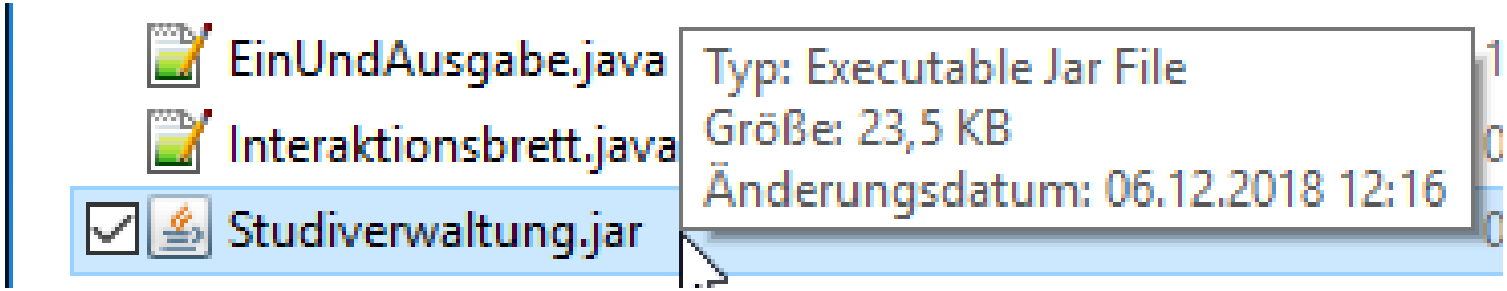
ausführbares Programm in BlueJ (4/4)



- Kommando-Zeile (oder Batch-Datei)

```
Konsolenfenster - java  -jar Studiverwaltung.jar

F:\workspaces\BluejWork>java -jar Studiverwaltung.jar
Nächste Aktion:
 (0) Programm beenden
 (1) Neue studierende Person einfügen
```

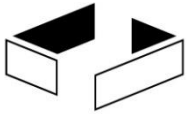
- wenn graphisch (JavaFX oder Swing) und im Betriebssystem konfiguriert, dann Start durch Doppelklick



- bei Problemen:

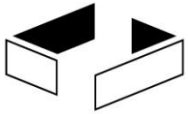
https://wiki.byte-welt.net/wiki/Jar-Datei_mit_Doppelklick_nicht_ausf%C3%BChrbar%3F#Spezialfall_Windows

Nutzung von Übergabeparametern (1/2)



```
public class MainAnalyse{

    public static void main(String[] s){
        if(s == null){
            System.out.println("null");
        } else {
            if(s.length == 0){
                System.out.println("kein Parameter");
            } else {
                for(int i = 0; i < s.length; i = i + 1){
                    System.out.println(i + ": " + s[i]);
                }
            }
        }
    }
}
```

Nutzung von Übergabeparametern (2/2)



D:\Windows\system32\cmd.exe

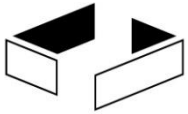
```
c:\bsp>java MainAnalyse
kein Parameter

c:\bsp>java MainAnalyse Hai wo
0: Hai
1: wo

c:\bsp>java MainAnalyse "Hai wo" "Ei nu da" "Au ah"
0: Hai wo
1: Ei nu da
2: Au ah

c:\bsp>
```

Beispiel

Video

Exception

Erinnerung: Studierendenverwaltung

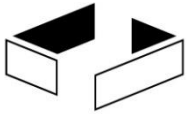


- Aufgabe: Objekte in eine Datei schreiben und aus einer Datei lesen (persistieren von Daten)

- Beispiel von zu verwaltenden Objekten:

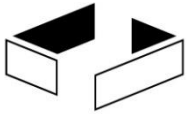
```
public class Studierendenverwaltung {
  private ArrayList<Studierend> studierende;
  private String semester = "Semester XXXX";
  …
```

Schreiben – Ansatz (1/2)



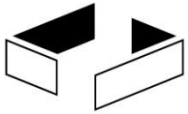
- Basisansatz: Lesen (Input) oder Schreiben (Output) einzelner Bytes einer Datei (machbar, aber für Datentypen, die größer als Byte sind, unhandlich)
- Es handelt sich um Streams
- Schreiben: Datei öffnen und immer hinten anfügen
- Lesen: Datei öffnen und von vorne nach hinten lesen
- Streams sind immer unidirektional (lesen / schreiben)
- Das Paket java.io stellt verschiedene Klassen zur Verfügung, die auf dem Streamkonzept basieren

Schreiben – Ansatz (2/2)



- Prinzipieller Ansatz
  - öffne Datei
  - schreibe relevante Informationen
  - schließe Datei (WICHTIG !!!! -> schleichender Fehler)
- Frage was wird geschrieben
- Antwort: hängt von der genutzten Realisierung ab, abhängig vom Einsatzbereich
- Ansatz hier: es können beliebige Beans geschrieben werden
  - Beans:
    - parameterloser Konstruktor
    - get- und set-Methoden für alle Objektvariablen
    - Typen der Objektvariablen sind Beans

Öffnen einer Datei

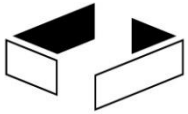


- Java erlaubt einfaches Lesen und Speichern im XML-Format, wenn alle zu speichernden Objekte get- und set- Methoden in Standard-Form und parameterlosen Konstruktor haben
- Zum Öffnen einer Datei "datei.xml" zum Schreiben von Daten wird folgende Konstruktion genutzt

```
XMLEncoder file = null;
 file = new XMLEncoder(
                new BufferedOutputStream(
                    new FileOutputStream("datei.xml")));
```

- Ermöglicht das sequentielle Schreiben (hintereinander) von Objekten in die Datei
- Konstruktion wird (deutlich) später erläutert, muss jetzt erstmal hingenommen werden

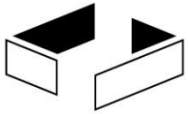
Notwendige Imports



- Hier kompakte Übersicht, weitere Erklärungen bei Nutzung

```
import java.beans.XMLDecoder;
import java.beans.XMLEncoder;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.util.ArrayList;
```

Programmerweiterungsfragmente

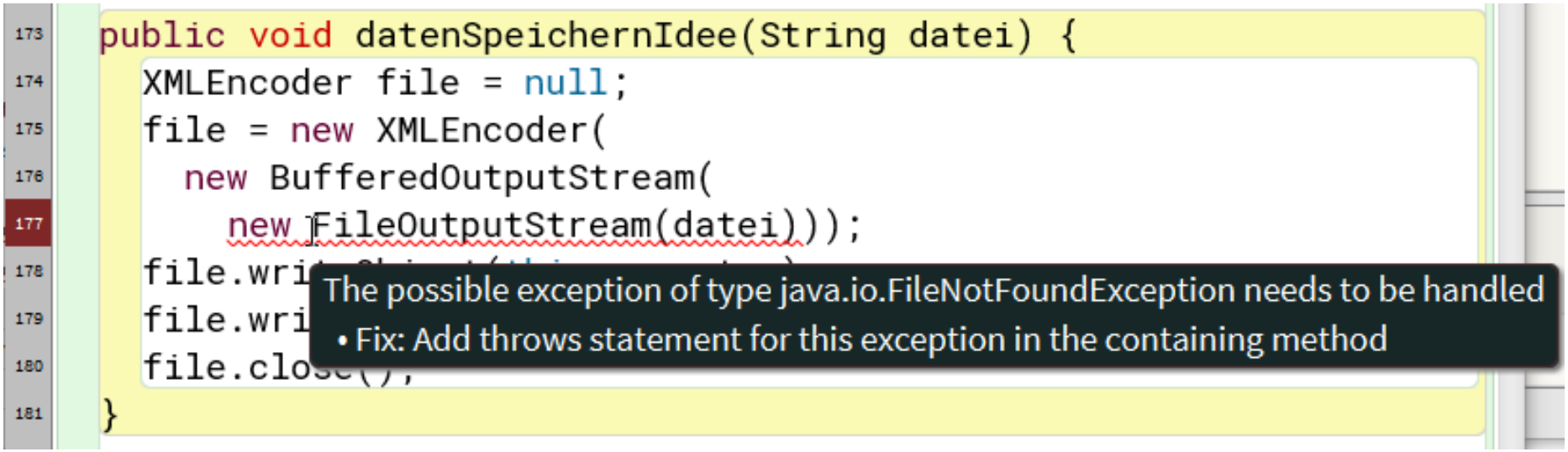


```
        + "(11) Daten speichern\n"
        + "(12) Daten laden: ");
```

```
    case 11: {
      this.datenSpeichern();
      break;
    }
    case 12: {
      this.datenLaden();
      break;
    }
```

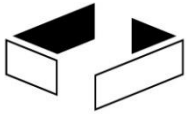
```
public void datenSpeichern() {
  String datei = this.nichtLeererText(
                      "Dateiname (Abbruch mit \"Ende\")");
  if (!datei.toUpperCase().equals("ENDE")) {
    this.datenSpeichern(datei);
  }
}
```

Programmfragment – nicht lauffähig



- der markierte Konstruktor kann eine sogenannte Exception (Ausnahme) werfen, die behandelt werden muss

Exception - Anwendungsszenario



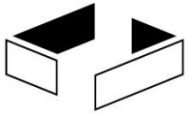
bisher bekannt

- typischer Ablauf: Sequenz und Schleifen mit typischen Verhalten
- alternative Abläufe: erwartete Spezialfälle, die den typischen Ablauf variieren (if- Anweisungen)

neu

- Ausnahme-Situationen, die keinen typischen Programmablauf ermöglichen, aber in denen das Programm sinnvoll reagieren kann, Beispiele:
  - erfolgreicher Aufbau einer Datenbankverbindung, Netzwerkverbindung oder zu einer Datei, die während der Nutzung abbricht
- Wunsch: Programm wieder in einen nutzbaren Zustand führen

Exception - Ablaufprinzip



Ausnahmesituation tritt auf

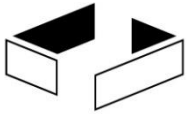
- wenn Situation erkannt, wird typischer Programmablauf sofort verlassen und Programmstück für diesen Fehler angesprungen
- das Programmstück ist entweder in der Methode, die gerade läuft, oder in einer aufrufenden Methode

- sinnvoll: unabhängig ob Ausnahmesituation aufgetreten ist, sollten Abschlussaktionen möglich sein

wichtige Hinweise:

- Exception sollen nicht einfaches if ersetzen
- Exception-Handling soll keine Programmierfehler kaschieren

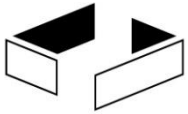
Beispiel: lokale Behandlung



```
public void datenSpeichern(String datei) {
  XMLEncoder file = null;
  try {
    file = new XMLEncoder(
               new BufferedOutputStream(
                   new FileOutputStream(datei)));
    file.writeObject(this.semester);
    file.writeObject(this.studierende);
  } catch (FileNotFoundException e) {
    EinUndAusgabe io = new EinUndAusgabe();
    io.ausgeben("Probleme beim Schreiben: " + e);
  } finally {
    if (file != null) {
      file.close();
    }
  }
}
```

```
public class Studierendenverwaltung {
  private ArrayList<Studierend> studierende;
  private String semester …
```

Konzept try-catch-finally



```
try {
```

Programmstück (Block), in dem eine Exception auftreten könnte; wenn keiner Exception gehe am Ende zu finally-Block, wenn existent

```
} catch (ExceptionKlasse1 e1) {
```

wenn Exception dieses Typs (oder eines davon erbenden Typs) geworfen, mache dieses; gehe zu finally Block, wenn existent

```
} catch (ExceptionKlasse2 e1) {
```

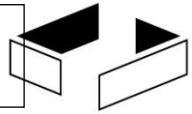
wenn Exception dieses Typs (oder eines davon erbenden Typs) geworfen, mache dieses; gehe zu finally Block, wenn existent

```
} finally {
```

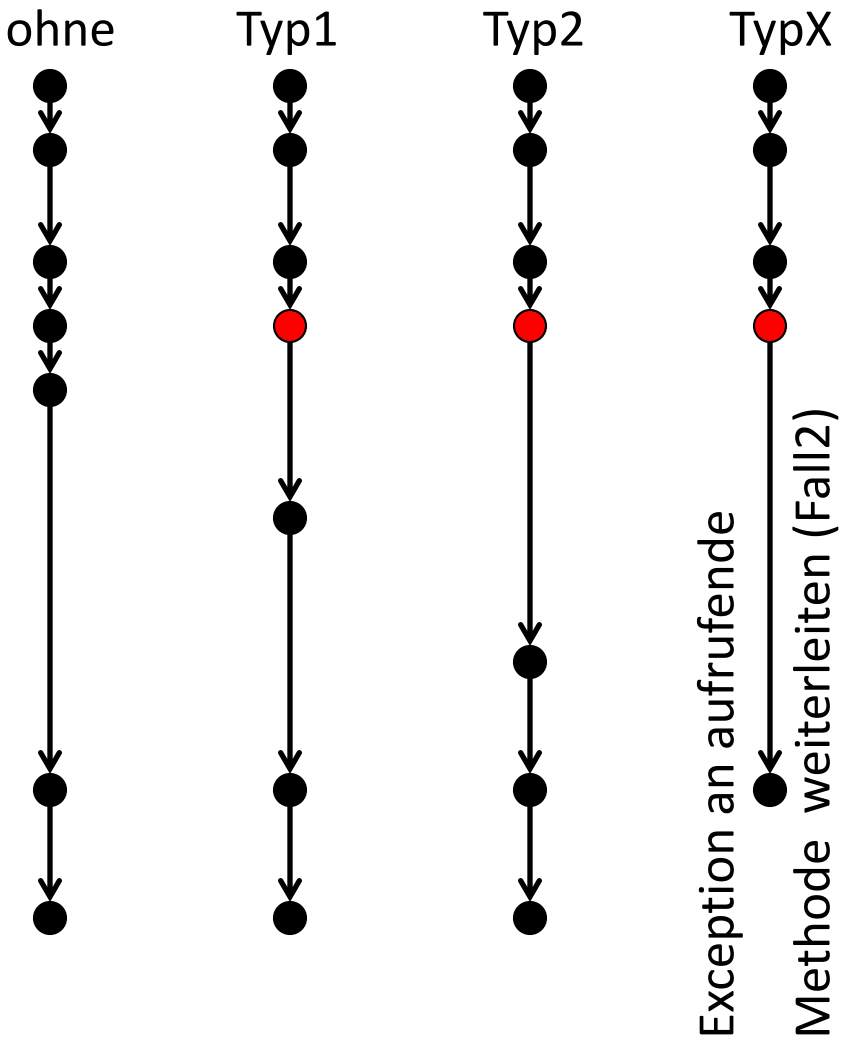
optionaler Block, wird immer am Ende ausgeführt

```
}
```

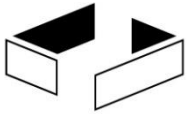
Fall 1: lokale Exception-Behandlung



```
public void machWas(...) {
  normalesProgramm;
  normalesProgramm;
  try {
     normalesProgramm;
     AnweisungEvtlException;
     normalesProgramm;
  } catch (ExceptionTyp1 e) {
    ReaktionAufException;
  } catch (ExceptionTyp2 e) {
    ReaktionAufException;
  } finally {
    Abschlussaktion;
  }
  normalesProgramm;
}
```



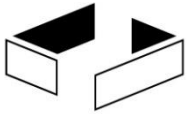
Speicherergebnis (Ausschnitt)



Dateiname (Abbruch mit "Ende"): daten.xml

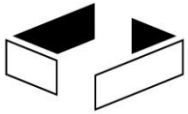
```
<?xml version="1.0" encoding="UTF-8"?>
<java version="16.0.1" class="java.beans.XMLDecoder">
 <string>Semester XXXX</string>
 <object class="java.util.ArrayList">
  <void method="add">
   <object class="Studierend">
    <void property="geburtsjahr">
     <int>1987</int>
    </void>
    <void property="matrikelnummer">
     <int>424241</int>
    </void>
    <void property="nachname">
     <string>X</string>
    </void>
    ...
```

Laden - Ansatz



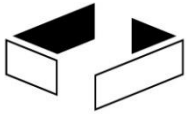
- Der Ansatz zum Laden ist sehr ähnlich zum Speichern
- Es wird wieder ein Stream benutzt, aus dem Daten gelesen werden können (eine Datei kann z. B. als Stream geöffnet werden)
- die Daten werden wieder Schritt für Schritt eingelesen; dabei muss die Reihenfolge des Schreibens beachtet werden
- mit readObject werden Objekte gelesen, die gecastet werden müssen
- wieder ist das Schließen der Datei eminent wichtig
- wieder müssen Ausnahmen (Datei nicht lesbar, oder Verbindung bricht ab) beachtet werden
- Anmerkung: Klasse java.io.File kann zur Analyse von Dateien genutzt werden

Datei zum Laden auswählen



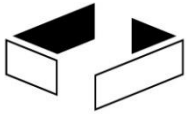
```
public void datenLaden() {
  EinUndAusgabe io = new EinUndAusgabe();
  String datei=nichtLeererText("Dateiname (Abbruch \"Ende\")");
  if (datei.toUpperCase().equals("ENDE")){
    return;
  }
  File file = new File(datei);
  if (!file.exists()){
    io.ausgeben("Datei existiert nicht.\n");
    return;
  }
  if (file.isDirectory()){
    io.ausgeben("Verzeichnis nicht als Datei nutzbar.\n");
    return;
  }
  if (!file.canRead()) {
    io.ausgeben("Datei nicht lesbar.\n");
    return;
  }
  this.datenLaden(datei);
}
```

Datei laden



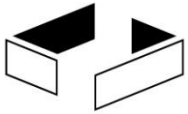
```
public void datenLaden(String datei) {
  EinUndAusgabe io = new EinUndAusgabe();
  XMLDecoder file = null;
  try {
    file = new XMLDecoder(
                new BufferedInputStream(
                    new FileInputStream(datei)));
    this.semester = (String) file.readObject();
    this.studierende = (ArrayList<Studierend>) file.readObject();
    io.ausgeben("Daten geladen.\n");
  } catch (FileNotFoundException e) {
      io.ausgeben("Probleme beim Lesen: " + e);
  } finally {
    if(file!=null){
      file.close();
    }
  }
}
```

Ausschnitt Nutzungsdialog



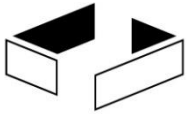
```
(12) Daten laden: 12
Dateiname (Abbruch mit "Ende"): Hallo.text
Datei existiert nicht.

(12) Daten laden: 12
Dateiname (Abbruch mit "Ende"): ..
Verzeichnis nicht nutzbar.

(12) Daten laden: 12
Dateiname (Abbruch mit "Ende"): eNDe

(12) Daten laden: 12
Dateiname (Abbruch mit "Ende"): daten.xml
Daten geladen.
```

Test (1/2)



```
public class StudierendenverwaltungTest{

  private Studierendenverwaltung studis;
  private Studierend s1 =new Studierend("Uwe","X",1987,"IMI",424241);
  private Studierend s2 =new AustauschStudierend("CHN","Xi","Yu"
                                                ,1988,"IMI",424242);
  private Studierend s3 =new AustauschStudierend("USA","Mo","Jo"
                                                ,1989,"MID",424243);
  private Studierend s4 =new Studierend("Uta","B",1987,"MID",424244);

  @BeforeEach
  public void setUp(){
    this.studis = new Studierendenverwaltung();
    this.studis.hinzufuegen(this.s1);
    this.studis.hinzufuegen(this.s2);
    this.studis.hinzufuegen(this.s3);
    this.studis.hinzufuegen(this.s4);
  }
```

Test (2/2)

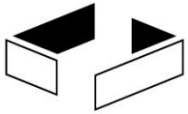


- Wichtig: Aufräumen nicht vergessen (noch besser in @BeforeEach- und @AfterEach-Methoden, da dann garantiert ausgeführt)

```
@Test
public void testPersistenz1(){
  this.studis.datenSpeichern("tmp.txt");
  this.studis.studiengangLoeschen("ITI");
  this.studis.studiengangLoeschen("MID");
  this.studis.datenLaden("tmp.txt");
  Assertions
      .assertTrue(this.studis.StudierendenZaehlenIn("IMI") == 2);
  Assertions
      .assertTrue(this.studis.StudierendenZaehlenIn("MID") == 2);
  File file = new File("tmp.txt");
  file.delete();
}
```

- weitere Testszenarien denkbar und sinnvoll

Fall 2: Exception weiterreichen



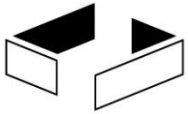
Video

- Wird eine Exception nicht lokal behandelt, wird sie an die aufrufende Methode weitergeleitet
- Damit Weiterleitung möglich, muss hinter der Parameterliste eine throws-Zeile mit allen möglichen Exceptiontypen stehen

```
public void mach() throws ExceptionTyp1, ExceptionTyp2
```

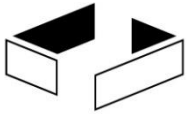
- in der aufrufenden Methode muss Exception bearbeitet werden, wieder
  - Fall 1: lokal behandeln
  - Fall 2: über throws-Deklaration an aufrufende Methode weiterleiten
- existiert aufrufende Methode nicht, endet Programm mit Fehlermeldung

Beispiel: Weiterreichen einer Exception (1/2)



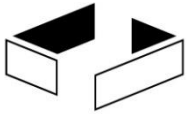
```
public void datenLaden(String datei)
                 throws FileNotFoundException {
  XMLDecoder file = null;
  file = new XMLDecoder(new BufferedInputStream(
                 new FileInputStream(datei)));
  this.semester = (String) file.readObject();
  this.studierende = (ArrayList<Studierend>) file.readObject();
  file.close();
}
```

Beispiel: Weiterreichen einer Exception (2/2)



```
public void datenLaden() {
  EinUndAusgabe io = new EinUndAusgabe();
  String datei = this.nichtLeererText("Dateiname "
                                +"(Abbruch mit \"Ende\")");
  if (datei.toUpperCase().equals("ENDE")) {
    return;
  }
  try {
    this.datenLaden(datei);
    io.ausgeben("Daten geladen.\n");
  } catch (FileNotFoundException e) {
    io.ausgeben("Problem beim Laden.\n");
  }
}
```

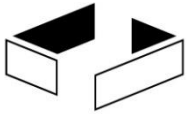
Variante



- Problem bei letzter Version: Datei könnte offen bleiben
- zu try muss es zumindest catch- oder finally-Block geben

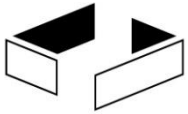
```
public void datenLaden2(String datei)
            throws FileNotFoundException {
  XMLDecoder file = null;
  try {
    file = new XMLDecoder(new BufferedInputStream(
             new FileInputStream(datei)));
    this.semester = (String) file.readObject();
    this.studierende = (ArrayList<Studierend>) file.readObject();
  } finally {
    if (file != null) {
      file.close();
    }
  }
}
```

Testen von unerwünschten Exceptions



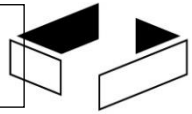
```
@Test
public void testPersistenz1(){
  try {
    this.studis.datenSpeichern("tmp.txt");
    this.studis.studiengangLoeschen("ITI");
    this.studis.studiengangLoeschen("MID");
    this.studis.datenLaden("tmp.txt");
    Assertions
        .assertTrue(this.studis.studierendenZaehlenIn("IMI") == 2);
    Assertions
        .assertTrue(this.studis.studierendenZaehlenIn("MID") == 2);
    File file = new File("tmp.txt");
    file.delete();
  } catch (FileNotFoundException e) {
    Assertions.assertTrue(false, "unerwartete Exception");
  }
}
// statt Assertions.assertTrue(false) schöner Assertions.fail()
```

Testen von erwünschten Exceptions

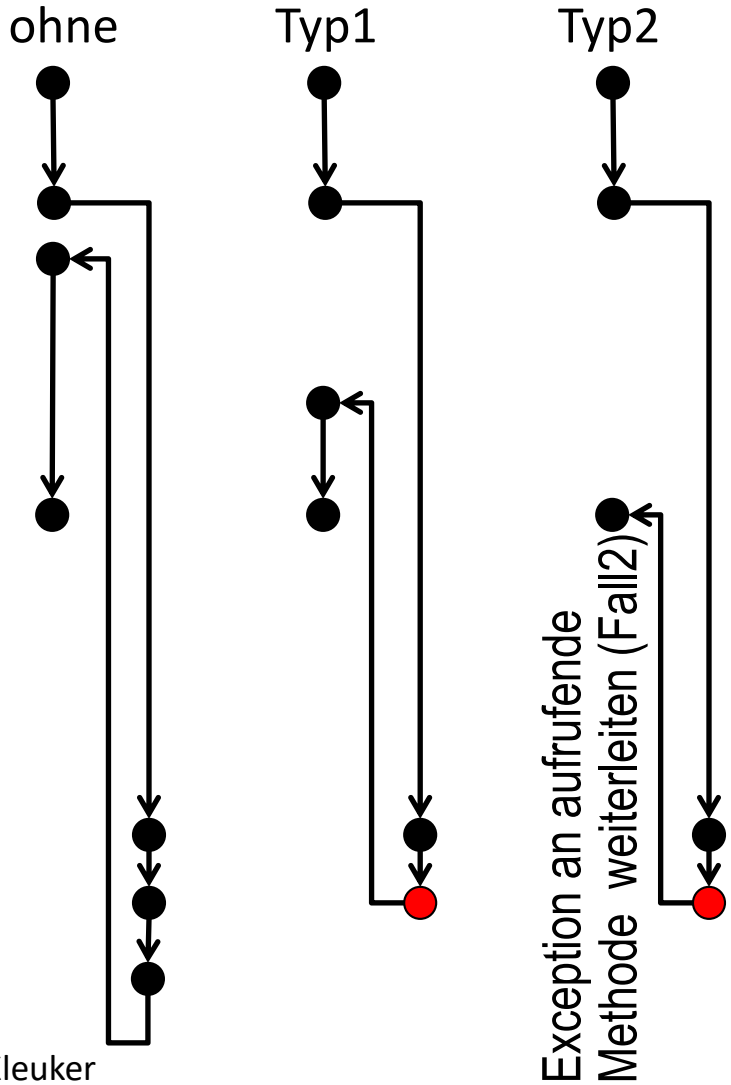


```
@Test
public void testPersistenz2(){
  try {
    File file = new File("tmp.txt");
    file.delete();
    this.studis.datenLaden("tmp.txt");
    Assertions.assertTrue(false);
  } catch (FileNotFoundException e) {
    Assertions
        .assertTrue(this.studis.studierendenZaehlenIn("IMI") == 2);
    Assertions
        .assertTrue(this.studis.studierendenZaehlenIn("MID") == 2);
  }
}
```

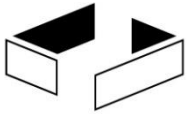
Fall 2: weiterleitende Exception-Behandlung



```
public void rufAuf(...)
  throws ExTyp2 {
  normalesProgramm;
  try {
     machWas(...);
     normalesProgramm;
  } catch (ExTyp1 e) {
    ReaktionAufException;
  } finally {
    Abschlussaktion;
  }
}
public void machWas(...)
    throws ExTyp1, ExTyp2 {
  normalesProgramm;
  AnweisungEvtlException;
  normalesProgramm;
}
```



Exceptions selbst werfen (auslösen)

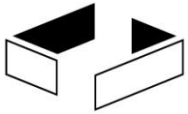


- In unerwünschten Situationen kann man selbst Exceptions werfen; hierzu dient der Befehl throw, genauer

```
throw new ExceptionTyp("Grund der Ausnahme");
```

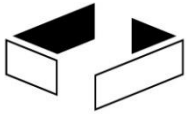
- Man kann hier bereits in Java existierende Exceptions nutzen, die typischerweise einen Konstruktor mit zu übergebendem Grund enthalten
- Man kann selbst Exception-Klassen schreiben; diese müssen von der Klasse Exception oder einer anderen existierenden Exception erben
  - üblich ist ein Konstruktor, dem der Grund als String übergeben werden kann
- Konzept der Exception-Nutzung muss einheitlich im Projekt geklärt werden

Nutzung existierender Exception (1/2)



```
public void datenSpeichern3(String datei)
    throws FileNotFoundException, IllegalArgumentException {
  if(!datei.startsWith("StudSave")){
    throw new IllegalArgumentException("Dateianfang "
        + "ohne StudSave ignoriert");
  }
  try{
    XMLEncoder file = null;
    file = new XMLEncoder(new BufferedOutputStream(
                  new FileOutputStream(datei)));
    file.writeObject(this.semester);
    file.writeObject(this.studierende);
    file.close();
  } finally {
    if(file != null){
      file.close();
    }
  }
}
```

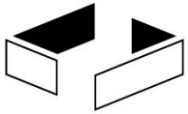
Nutzung existierender Exception (2/2)



```
public void datenSpeichern() {
  EinUndAusgabe io = new EinUndAusgabe();
  String datei = this.nichtLeererText("Dateiname "
                                +"(Abbruch mit \"Ende\")");
  if (!datei.toUpperCase().equals("ENDE")) {
    try {
      this.datenSpeichern(datei);
      io.ausgeben("Speichern erfolgreich.\n");
    } catch (FileNotFoundException e) {
      io.ausgeben("Speicherproblem: " + e.getMessage() + "\n");
    } catch (IllegalArgumentException e) {
      io.ausgeben("Namensproblem: " + e.getMessage() + "\n");
    }
  }
}
```

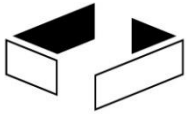
```
(11) Daten speichern
(12) Daten laden: 11
Dateiname (Abbruch mit "Ende"): Sicherung
Namensproblem: Dateianfang ohne StudSave ignoriert
```

Eigene Exception



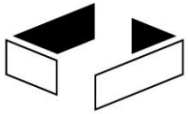
```
public class DateinamenException extends Exception {
  public DateinamenException(){
    super("Muss mit StudSave beginnen");
  }

  public DateinamenException(String grund){
    super(grund);
  }
}
```

Nutzung eigener Exception (1/2)



```
  public void datenSpeichern4(String datei)
      throws FileNotFoundException, DateinamenException {
    if(!datei.startsWith("StudSave")){
      throw new DateinamenException();
    }
    try{
      XMLEncoder file = null;
      file = new XMLEncoder(new BufferedOutputStream(
                      new FileOutputStream(datei)));
      file.writeObject(this.semester);
      file.writeObject(this.studierende);
      file.close();
    } finally {
      if(file!=null){
        file.close();
      }
    }
}
```

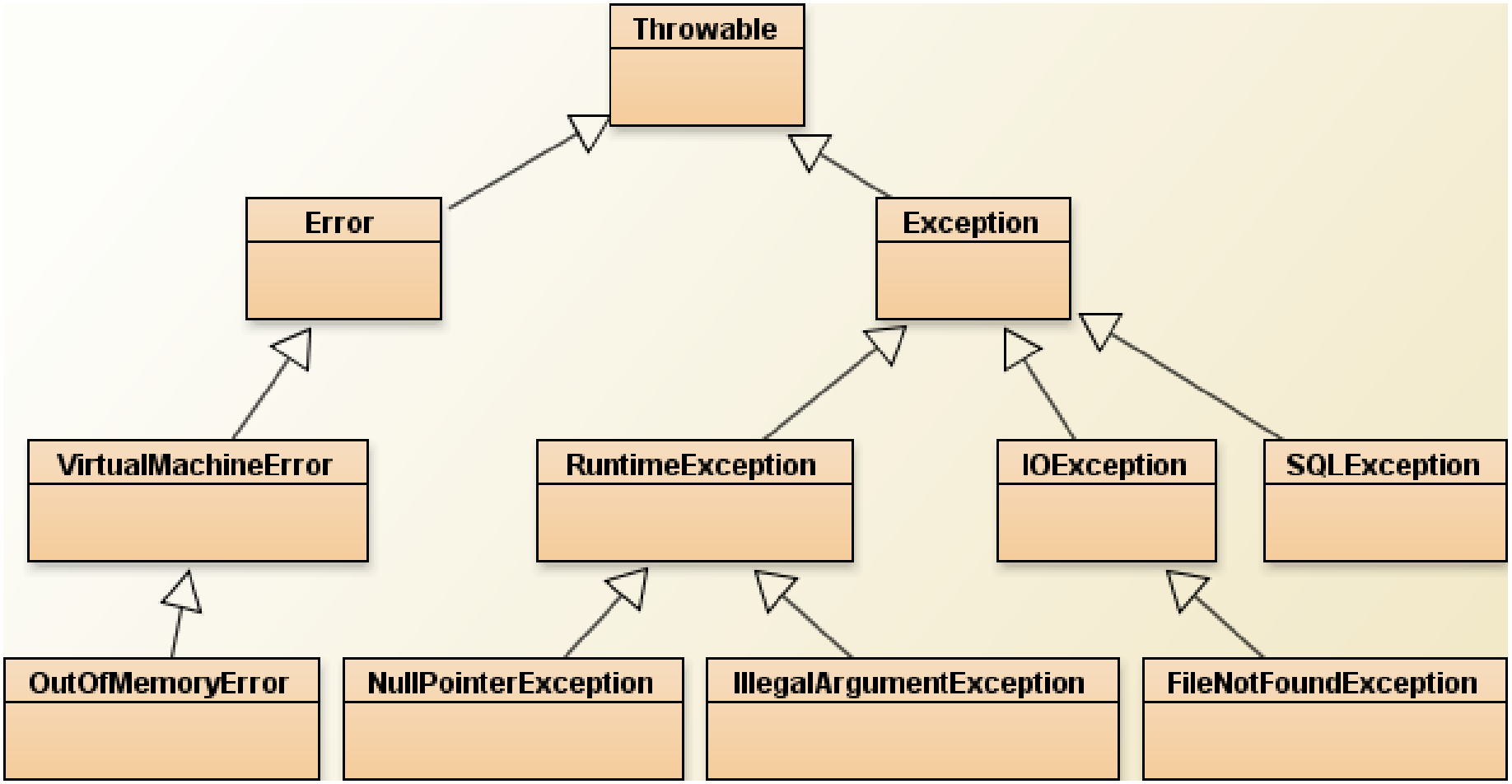
Nutzung eigener Exception (2/2)



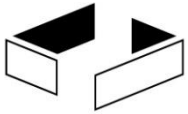
```
public void datenSpeichern() {
  EinUndAusgabe io = new EinUndAusgabe();
  String datei = this.nichtLeererText("Dateiname "
                                +"(Abbruch mit \"Ende\")");
  if (!datei.toUpperCase().equals("ENDE")) {
    try {
      this.datenSpeichern4(datei);
      io.ausgeben("Speichern erfolgreich.\n");
    } catch (FileNotFoundException e) {
      io.ausgeben("Speicherproblem: " + e.getMessage()+ "\n");
    } catch (DateinamenException e) {
      io.ausgeben("Namensproblem: " + e.getMessage()+ "\n");
    }
  }
}
```

```
(11) Daten speichern
(12) Daten laden: 11
Dateiname (Abbruch mit "Ende"): Sicherung
Namensproblem: Muss mit StudSave beginnen
```

Ausschnitt aus Java-Exception-Klassen

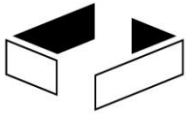


Ausnahmenarten



- Alle neuen Sprachen erlauben Exception Handling
- Ausnahme werfen, wenn nicht erwarte Situation auftritt
- Programmierfehler (z. B. NullPointerException), führt zum Programmabbruch; nie ernsthaft behandeln, nur z. B. in einer speziellen Log-Datei notieren
- Echte Fehler (Error) sollen nie behandelt werden
- *Exceptions, die von RuntimeExeption erben, müssen nicht behandelt werden (unchecked Exception) und müssen nicht (dürfen schon) in throws-Listen stehen*
- Andere (checked) Exceptions müssen behandelt werden; nach außen geben (throws) oder behandeln (try-catch-finally)

Anmerkungen zu try-catch

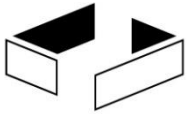


- Man kann try-catch-Blöcke beliebig ineinander schachteln
- Man kann einen oder mehr catch-Blöcke zu einem try definieren
- Compiler prüft, ob catch-Blöcke theoretisch erreichbar sind (nie erst allgemeine, dann spezielle Exception fangen)
- Es können neue Exceptions auch in catch-Blöcken (weiter) geworfen werden; es beginnt neue Exception-Behandlung
- Exceptions können gefangen und wieder geworfen werden

```
catch(ExeptionTypX e){ …
   throw e;
```

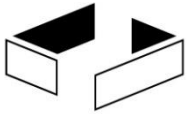
- Bei Vererbung darf eine überschreibende Methode in einer erbenden Klasse nur die Exceptions der überschriebenen Methode der beerbten Klasse (oder weniger) werfen

Zusammenfassung zu Exceptions



- In größeren Projekten wird festgelegt, wann es sich um unerwünschte, aber behandelbare Ausnahmen handelt
- Für Ausnahmen werden eigene Klassen geschrieben (erben von Exception)
- Für Ausnahmen wird festgelegt, wer reagieren kann (wenn lokal möglich, dann da, sonst weitergeben)
- Für jeden Konstruktor und jede Methode wird dokumentiert, welche Exceptions sie werfen können (Erklärung in der Dokumentation)
- Typisch ist, dass Ausnahmen protokolliert werden
- `catch (Throwable e){}` ist schlechtester Stil

AutoCloseable



```
public void datenLaden(String datei) {
  EinUndAusgabe io = new EinUndAusgabe();
  XMLDecoder file = null;
  try (XMLDecoder file = new XMLDecoder(
                 new BufferedInputStream(
                     new FileInputStream(datei)))) {
    this.semester = (String) file.readObject();
    this.studierende = (ArrayList<Studierend>) file.readObject();
    io.ausgeben("Daten geladen.\n");
  } catch (FileNotFoundException e) {
      io.ausgeben("Probleme beim Lesen: " + e);
  }
}
```

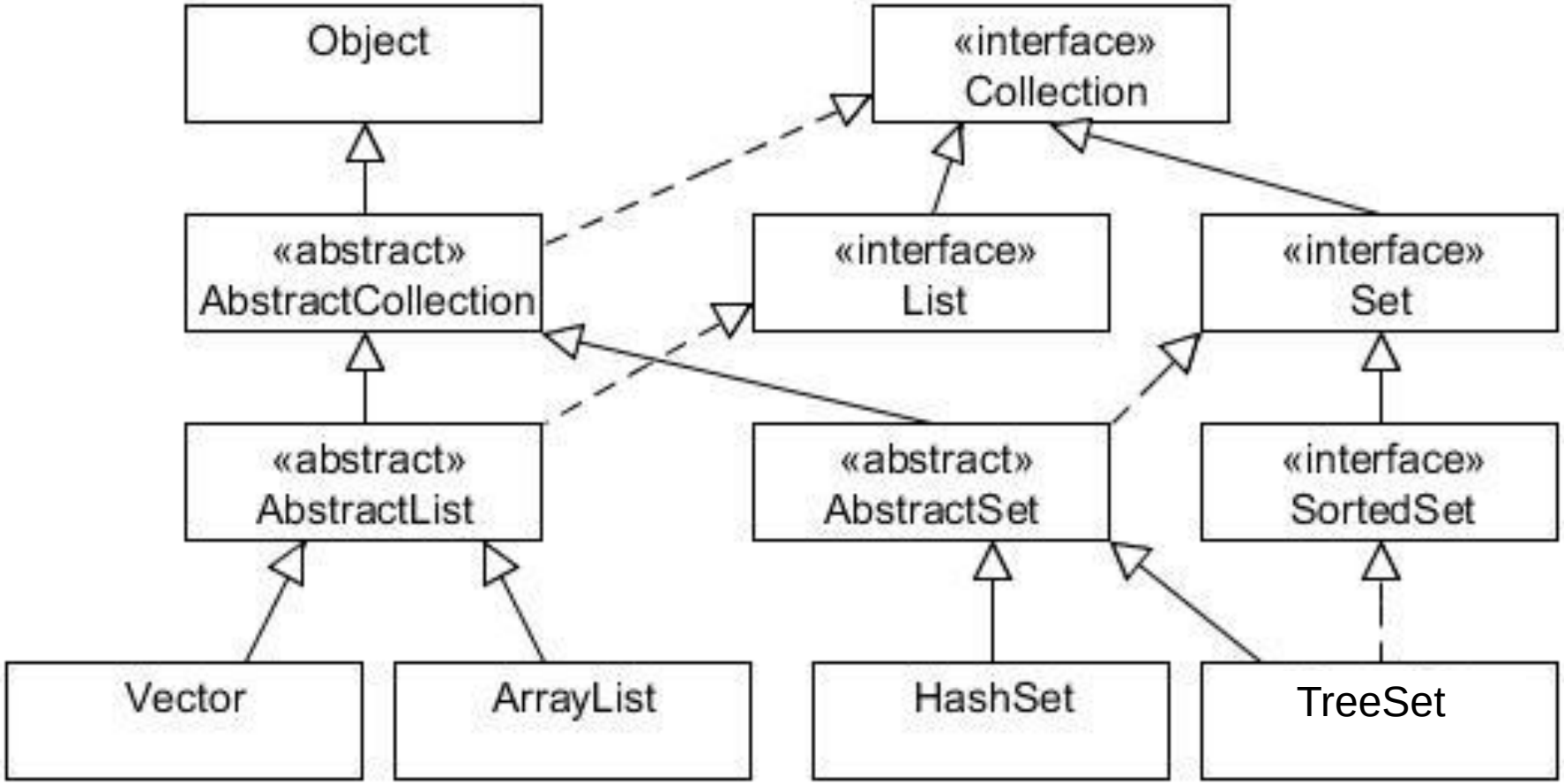
- viele Java-Klassen implementieren AutoCloseable-Interface (ab Java 7, try-with-resources)
- Variablen werden in den runden Klammern nach try deklariert
- egal, wie der Ablauf ist, auf allen Variablen wird am Ende immer close() ausgeführt

Beispiel

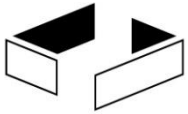
Video

Collection Framework

Ausschnitt aus Collection Framework (java.util)

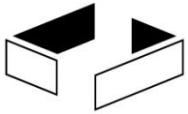


Das Interface Collection (Ausschnitt)



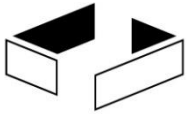
Method Summary	
boolean	add(Object o) Ensures that this collection contains the specified element.
boolean	addAll(Collection c) Adds all of the elements in the specified collection to this collection.
void	clear() Removes all of the elements from this collection.
boolean	contains(Object o) Returns true if this collection contains the specified element.
boolean	isEmpty() Returns true if this collection contains no elements.
Iterator	iterator() Returns an iterator over the elements in this collection.
boolean	remove(Object o) Removes a single instance of the specified element from this collection, if it is present.
boolean	retainAll(Collection c) Retains only the elements in this collection that are contained in the specified collection
int	size() Returns the number of elements in this collection.
Object[]	toArray() Returns an array containing all of the elements in this collection.

Das Interface List (Ausschnitt)

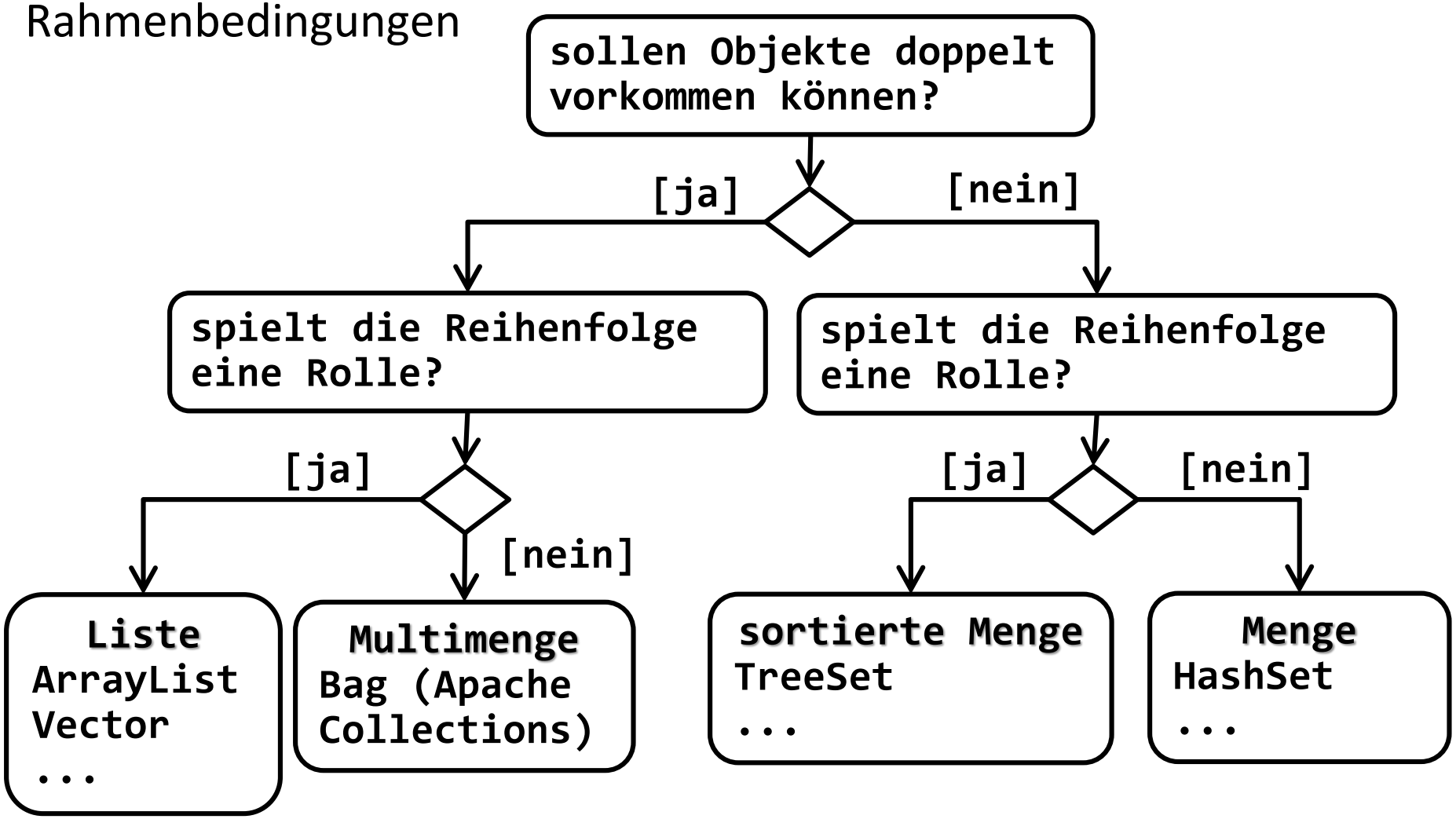


Method Summary	
void	add(int index, Object element) Inserts the specified element at the specified position in this list.
boolean	addAll(int index, Collection c) Inserts all of the elements in the specified collection into this list at the specified position.
Object	get(int index) Returns the element at the specified position in this list.
int	indexOf(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.
int	lastIndexOf(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element.
Object	remove(int index) Removes (and returns) the element at the specified position in this list.
Object	set(int index, Object element) Replaces the element at the specified position in this list with the specified element.

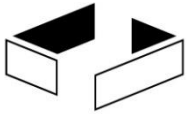
Warum verschiedene Sammlungen



Jede Implementierung besonders geeignet unter bestimmten Rahmenbedingungen



Beispielnutzung einer Menge (mit Problem)

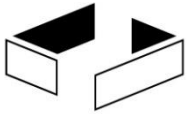


```
public void objektInMengeLoeschen(){
  EinUndAusgabe io = new EinUndAusgabe();
  Set<Punkt> ap = new HashSet<Punkt>();
  Punkt p1 = new Punkt(0, 0);
  ap.add(p1);
  ap.add(new Punkt(0, 0));
  ap.add(p1);
  io.ausgeben("Menge1: " + ap + "\n");
  ap.remove(new Punkt(0,0));
  io.ausgeben("Menge2: " + ap + "\n");
  ap.remove(p1);
  io.ausgeben("Menge3: " + ap + "\n");
}
```

```
Menge1: [[0,0], [0,0]]
Menge2: [[0,0], [0,0]]
Menge3: [[0,0]]
```

- Problem, da es den Punkt insgesamt nur einmal geben sollte, er aber doppelt ist, nur Hinzufügen eines identischen Objekts wird verhindert

Idee von hashCode()

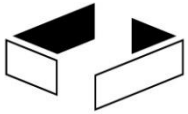


- equals-Berechnungen können sehr aufwändig sein
- ergänze schnell zu berechnende Methode, die für gleiche Elemente immer gleichen Wert liefert und für unterschiedliche Objekte möglichst unterschiedlichen Wert
- Gleichheitsprüfung: prüfe, ob schnell berechnete Werte übereinstimmen und nur dann wird equals ausgeführt
- ohne HashCode gilt aber:

```
public void zeigeHashCode(){
   EinUndAusgabe io = new EinUndAusgabe();
   Punkt p1 = new Punkt(0, 0);
   Punkt p2 = new Punkt(0, 0);
   io.ausgeben("p1: " + p1.hashCode() + "\n");
   io.ausgeben("p2: " + p2.hashCode() + "\n");
 }
```

```
p1: 14576877
p2: 12677476
```

Realisierung mit hashCode()



- in Punkt (der Methode public boolean equals(Objekt obj) hat)

```
@Override
public int hashCode() {
      return this.x + (this.y * 33);
}
```

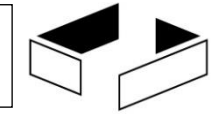
- Methode zeigeHashCode() liefert

```
p1: 0
p2: 0
```

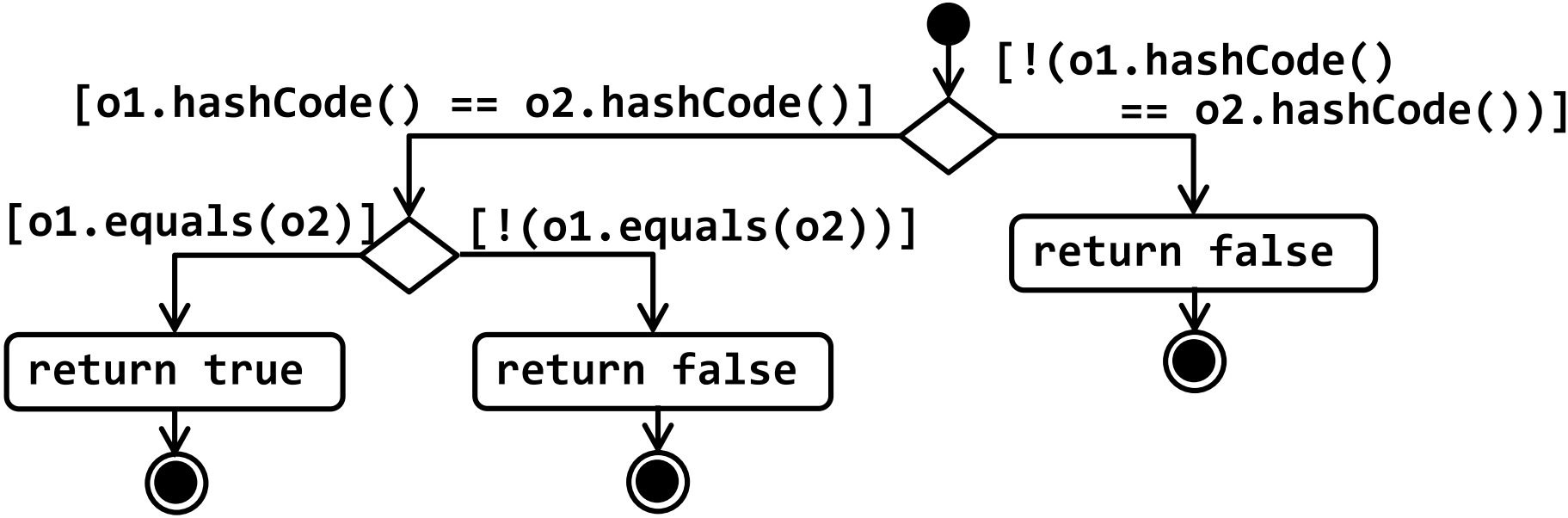
- Methode objektInMengeLoeschen() liefert

```
Menge1: [[0,0]]
Menge2: []
Menge3: []
```

Objektvergleich mit hashCode



- Prüfung auf Gleichheit für zwei Objekte o1, o2



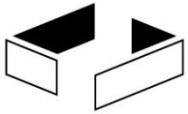
- minimale Anforderung an hashCode(): liefert für gleiche Objekte gleichen Wert; soll Berechnungen bei Ungleichheit beschleunigen

Hinweis: Hashing wird in verschiedenen Bereichen genutzt, u. a. Thema in A & D

Video

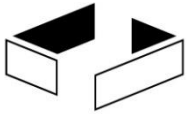
Konstanten

Klassenvariablen und Vererbung (1/4)



```
public class K1{ // zunaechst nur Wiederholung
    protected int objv;
    protected static int klav = 0;

    public K1(){
        this.objv = K1.klav;
        K1.klav = K1.klav + 1;
    }

    public static int klav(){  // Name getKlav() auch sinnvoll
        return K1.klav;
    }

    @Override
    public String toString(){
        return"K1: " + this.objv;
    }
}
```

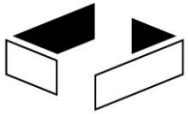
Klassenvariablen und Vererbung (2/4)



```
public class KSpielerei{

    public void analyse1(){
        K1[] ka = {new K1(), new K1(), new K1()};
        for(K1 k:ka){
            System.out.println(k);
        }
        System.out.println(K1.klav());
    }
}
```

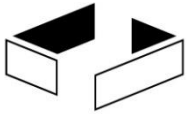
```
K1: 0
K1: 1
K1: 2
3
```

Klassenvariablen und Vererbung Vererbung (3/4)



```
public class K2 extends K1{
    public K2(){
        super();
        klav = klav + 1; // besser K1.klav schreiben
    }

    //darf kein @Override stehen da static
    public static int klav(){
        return K2.klav; // schlechtester erlaubter Stil
    }

    @Override
    public String toString(){
        return"K2: "+super.objv;
    }
}
```

Analysebeispiel Vererbung (4/4)

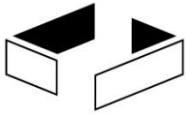


- in KSpielerei

```
public void analyse2(){
    EinUndAusgabe io = new EinUndAusgabe();
    K1[] ka = {new K2(), new K1(), new K2()};
    for(K1 k: ka){
        io.ausgeben(k + "\n");
    }
    io.ausgeben(K1.klav() + "\n");
    io.ausgeben(K2.klav() + "\n");
}
```

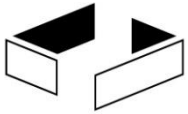
```
K2: 0
K1: 2
K2: 3
5
5
```

Anmerkung zu Sichtbarkeiten



- Grundsätzlich sind Klassenvariablen auch private oder protected, Klassenmethoden public (oder auch private oder protected)
- theoretisch kann man Objektvariablen und Klassenvariablen auch public machen, so dass man direkt über die Punktnotation darauf zugreifen kann
- zentrale Coding-Guideline: NIEMALS Objektvariablen und maximal nur konstante Klassenvariablen public machen
- Java wird seit 1991 entwickelt, wenige historisch unsaubere Stellen, für Kompatibilität drin gelassen
- public Objektvariable bei Array a: a.length
- public Klassenvariable in System: System.out

Definition von Konstanten

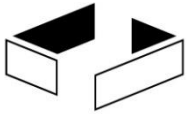


- Konstanten sinnvoll in Klassenvariablen abgelegt
- diese Variablen sind dann public
- damit unveränderbar, wird Schlüsselwort final ergänzt
- damit Konstanten schnell erkennbar, werden sie ausschließlich in Großbuchstagen geschrieben
- Beispiele in Math:

Fields

Modifier and Type	Field and Description
`static double`	`E` The `double` value that is closer than any other to *e*, the base of the natural logarithms.
`static double`	`PI` The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

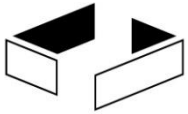
Beispiel: Nutzung der Math-Klassenmethoden



```
public class MathAnalyse{
    public void analyse(){
        EinUndAusgabe io = new EinUndAusgabe();
        io.ausgeben(Math.PI +"\n");
        io.ausgeben(Math.E +"\n");
        // Math.PI = 42.0; wg. final nicht ok
        io.ausgeben(Math.random() +"\n");
        io.ausgeben(Math.pow(10,Math.pow(10,10)) +"\n");
        io.ausgeben(Math.round(-0.5) +"\n");
        io.ausgeben(Math.round(0.5) +"\n");
        io.ausgeben(Math.sqrt(256.) +"\n");
        io.ausgeben(Math.log(Math.exp(42)) +"\n");
    }
}
```

```
3.141592653589793
2.718281828459045
0.770401046591034 3
Infinity
0
1
16.0
42.0
```

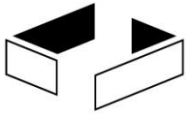
Beispiel: Nutzung von Konstanten (1/3)



- Auslagerung der Konstanten in eine neue Klasse (hier ggfls. auch Hilfsmethoden)

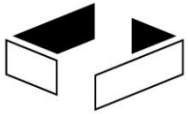
```
public class Firma{
    public static final String NAME = "Abmahnwahn KO";
    public static final double ZINS = 1.30;
    public static final int FRIST = 7;
}

public class Post{
    public static void oeffnungsbrief(Schuldend s){
        EinUndAusgabe io = new EinUndAusgabe();
        io.ausgeben("Ey " + s.getName() + ",\n"
          + "Du hast "+s.getKohle() + " bei " + Firma.NAME
          + " geliehen.\nRück " + s.getKohle() * Firma.ZINS
          + " bis in " + Firma.FRIST + " Tagen raus,\n"
          + "sonst Stress.\n\tMfG ein Freund\n");
    }
}
```

Beispiel: Nutzung von Konstanten (2/3)



```
public class Schuldend{
    private String name;
    private double kohle;

    public Schuldend(String n, double k){
        this.name = n;
        this.kohle = k;
    }

    public String getName(){
        return this.name;
    }

    public double getKohle(){
        return this.kohle;
    }
}
```

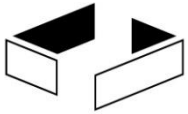
Beispiel: Nutzung von Konstanten (3/3)



```
Schuldner s = new Schuldner("Donald",100.0);
Post.oeffnungsbrief(s);
```

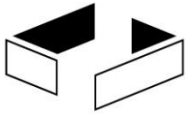
```
Ey Donald,
Du hast 100.0 bei Abmahnwahn KO geliehen.
Rück 130.0 bis in 7 Tagen raus,
sonst Stress.
      MfG ein Freund
```

Analyse des Ansatzes



- Recht einfache Änderung z. B. des Firmennamens, da nur eine Datei angepasst werden muss
- Nachteil bleibt, dass Programm verändert wird und deshalb neu übersetzt werden muss
- Flexiblerer Ansatz, wenn Firmendaten nur in einer mit normalen Editor änderbaren Datei stehen
- Ansatz: Konstanten werden durch Klassenmethoden geliefert, bei erster Nutzung sollen die Werte aus einer Datei eingelesen werden

multiple Möglichkeiten von final



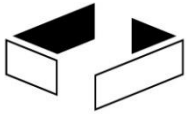
```
public final class Finale{ // kann man nicht von erben
    private final String text;  // hier oder im Konstruktor setzen
    private static final String konstante = "nur hier setzbar";

    public Finale(){
        this.text = "nur im Konstruktor einmal setzbar";
    }

    public final void nichtUeberschreibbar(){}

    public void referenzNichtAenderbar(
                                final ArrayList<String> liste){
        liste.add("Objektbearbearbeitung geht");
        // liste = new ArrayList<String>(); // geht nicht
        final String lokal;
        lokal = "einmal geht";
        // lokal = "zweimal nicht";
    }
}
```

Beispiel

Video

Aufzählungen

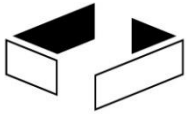
Aufzählungen für Anfänger und Klassiker



```
public class Programmiererfahrung {
  private String[] stufen={"nicht vorhanden",
    "Grundkenntnisse", "alte Projekterfahrung",
    "Projektmitarbeiter", "Experte"};
     private int stufenwert = 0;
    ...
}
```

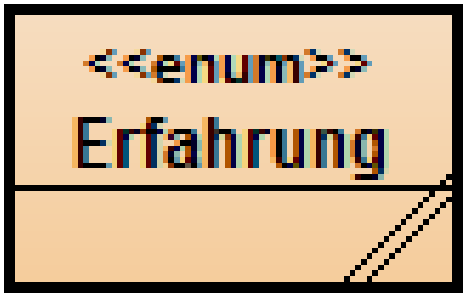
```
public class Erfahrung {
  public final static int NICHT_VORHANDEN = 0;
  public final static int GRUNDKENNTNISSE = 1;
  public final static int ALTE_PROJEKTERFAHRUNG = 2;
  public final static int PROJEKTMITARBEITER = 3;
  public final static int EXPERTE = 4;
}
Zugriff mit:
int meineErfahrung = Erfahrung.NICHT_VORHANDEN;
```

Aufzählungen mit Enum

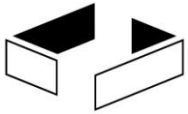


- Enum ist spezielle Art von Klasse (kann damit Methoden beinhalten)
- Werte als einfache Texte, beginnend mit Buchstaben aufschreiben

```
public enum Erfahrung {
   NICHT_VORHANDEN
   , GRUNDKENNTNISSE
   , ALTE_PROJEKTERFAHRUNG
   , PROJEKTMITARBEITER
   , EXPERTE
}
```



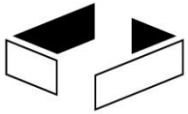
Aufzählungen mit Enum - Nutzung



```
public class Spielerei {
  public static void main (String[] s){
    Erfahrung ich = Erfahrung.NICHT_VORHANDEN;
    System.out.println(ich);
    for(Erfahrung e: Erfahrung.values()){
      System.out.println(e);
    }
  }
}
```

```
NICHT_VORHANDEN
NICHT_VORHANDEN
GRUNDKENNTNISSE
ALTE_PROJEKTERFAHRUNG
PROJEKTMITARBEITER
EXPERTE
```

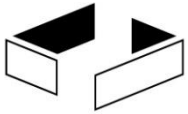
Weitere Methoden von enum-“Klassen“ (1/2)



```
Erfahrung ich = Erfahrung.NICHT_VORHANDEN;
System.out.println(ich.getClass());
System.out.println(ich.compareTo(Erfahrung.NICHT_VORHANDEN));
System.out.println(ich.compareTo(Erfahrung.EXPERTE));
System.out.println(ich.equals(0));
System.out.println(ich.equals(1));
System.out.println(ich.equals(Erfahrung.NICHT_VORHANDEN));
```

```
class Erfahrung
0
-4
false
false
true
```

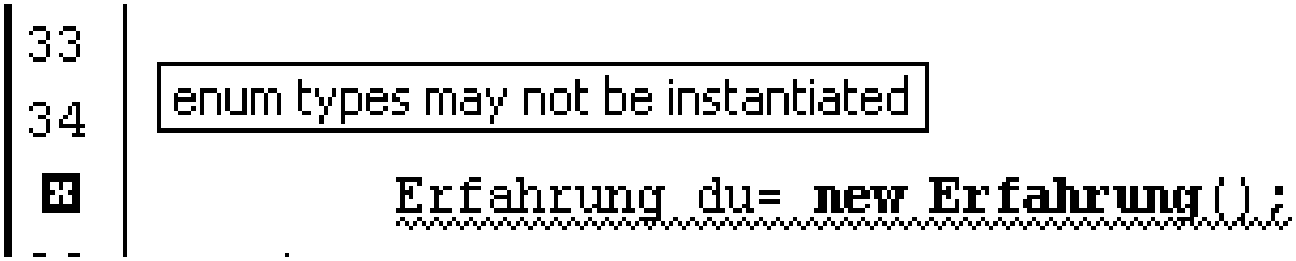
Weitere Methoden von enum-“Klassen“ (2/2)



```
System.out.println(ich.getDeclaringClass());
System.out.println(ich.name());
System.out.println(ich.ordinal());
System.out.println(Erfahrung.valueOf("EXPERTE"));
try{
  System.out.println(Erfahrung.valueOf("Experte"));
}catch (IllegalArgumentException e){
  System.out.println(e);
}
System.out.println(Erfahrung.valueOf(Erfahrung.class,"EXPERTE"));
```

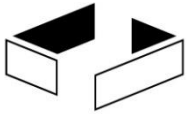
```
class Erfahrung
NICHT_VORHANDEN
0
EXPERTE
java.lang.IllegalArgumentException: No enum const Experte
EXPERTE
```

Objekte von enum-Klassen



- Mit enum werden besondere Klassen definiert, sind final (keine Vererbung möglich), keinen von außen nutzbaren Konstruktor
- Durch **`Erfahrung ich = Erfahrung.NICHT_VORHANDEN;`** erhält man ein Objekt der Klasse Erfahrung
- D. h. Enum-Klassen können zusätzlich „normale“ Exemplarmethoden und Klassenmethoden enthalten (auch z. B. toString())
- Klassenmethodenaufruf **`Erfahrung.klassenmethode();`**
- Methodenaufruf: **`ich.exemplarmethode();`**

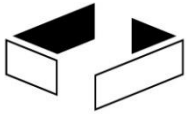
Erweiterte Möglichkeiten mit enum (1/2)



- Aufzählungswerte können Parameter haben, die im Konstruktor verarbeitet werden

```
public enum Coin {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25);
    private final int value;

    Coin(int value) {
        this.value = value;
    }

    public int value() {
        return value;
    }
}
```

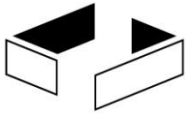
Erweiterte Möglichkeiten mit enum (2/2)



```
public class CoinAnalyse {
  private enum CoinColor { COPPER, NICKEL, SILVER }

  public static void main(String[] args) {
    for (Coin c : Coin.values()){
      System.out.println(c +":"+c.value()+"c "+color(c));
    }
  }

  private static CoinColor color(Coin c) {
    switch(c) {
      case PENNY: {return CoinColor.COPPER;}
      case NICKEL: {return CoinColor.NICKEL;}
      case DIME:
      case QUARTER: {return CoinColor.SILVER;}
      default: {
        throw new AssertionError("Unknown coin: " + c);
      }
    }
  }
}
```

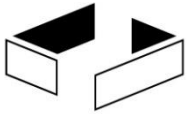
```
PENNY:1c COPPER
NICKEL:5c NICKEL
DIME:10c SILVER
QUARTER:25c SILVER
```

Kartenstapel (1/3)



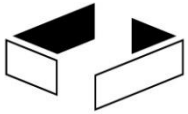
```
public enum Farbe {
       KARO, HERZ, KREUZ, PIK;

  @Override
  public String toString(){
    switch(this){
      case KARO: {return("Karo");}
      case HERZ: {return("Herz");}
      case KREUZ: {return("Kreuz");}
      case PIK: {return("Pik");}
      default: {return("");}
     }
   }
}
```

Kartenstapel (2/3)



```
public enum Wert {
      SIEBEN("7"), ACHT("8"), NEUN("9"), ZEHN("10"),
      BUBE("Bube"), DAME("Dame"), KOENIG("Koenig"),
   AS("As");
    private String text;

    Wert(String text){
        this.text = text;
    }

    @Override
    public String toString(){
        return text;
    }
}
```

Kartenstapel (3/3)



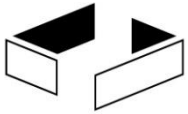
```
public class Karte {
    private Farbe farbe;
    private Wert wert;
    ...

public class Kartenstapel {
    private ArrayList<Karte> stapel = new ArrayList<Karte>();

    public static Kartenstapel gibAlleKarten(){
        Kartenstapel k = new Kartenstapel();
        for(Farbe f: Farbe.values())
            for(Wert w: Wert.values())
                k.hinzu(new Karte(f, w));
        return k;
    }
//...
```

Beispiel

Video

Pakete

Idee von Paketen



- Große Mengen von Klassen oft sehr unübersichtlich
- Ähnlich zu Dateisystemen ist eine logische Ordnung in verschiedenen Ordnern sinnvoll
- Ordner werden in Java Paket (package genannt)
- Paketzugehörigkeitsdeklaration steht immer am Anfang der Klasse

```
package de.ossoft.gui;
public class Maske{
```

- Datei Maske.java muss im Unterordner de/ossoft/gui des Projekts liegen
- Nutzung von Klassen aus anderen Paketen über import

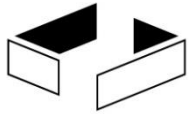
```
import de.ossoft.gui.Maske;
```

- statt „Paket“ auch „Namensraum“ (namespace) gebräuchlich

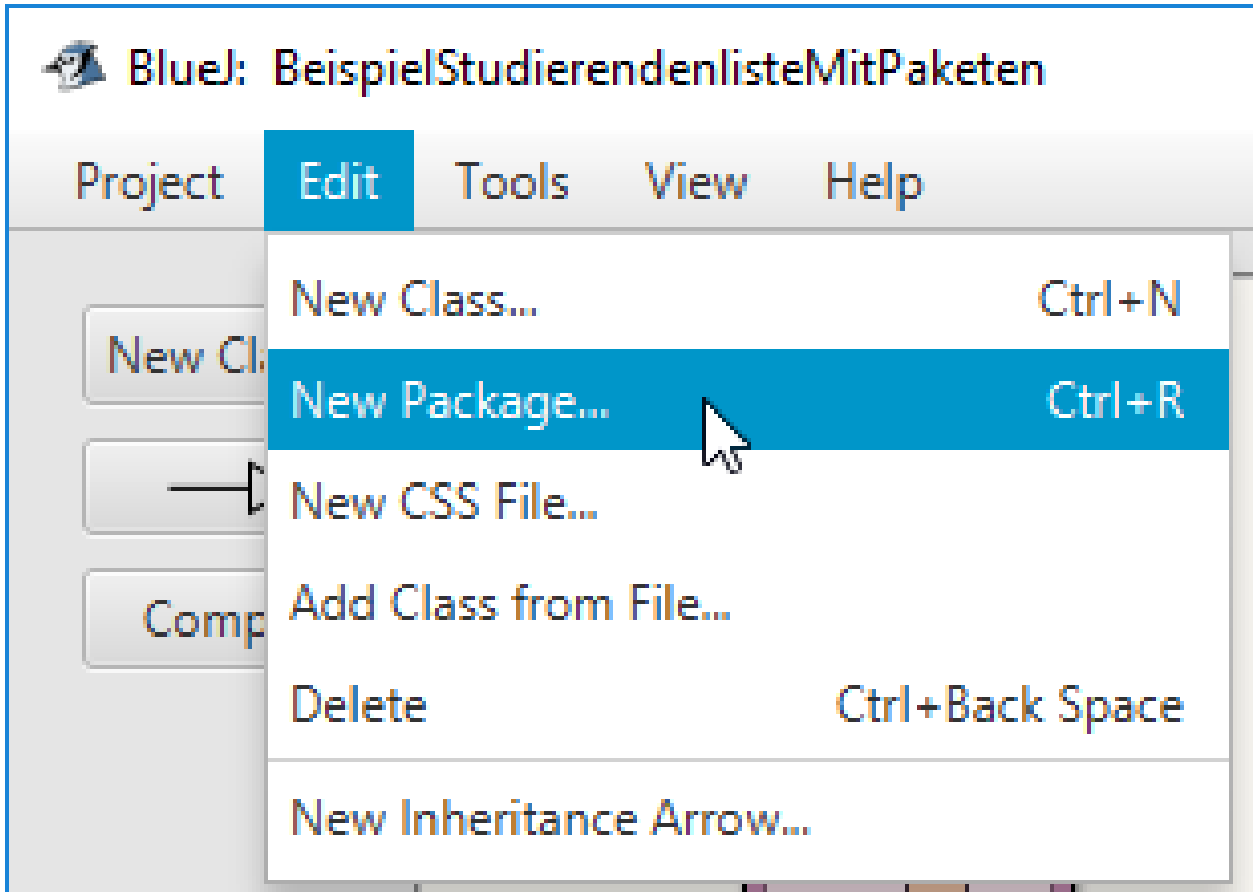
Ausschnitt: Java-Pakete der Klassenbibliothek

java.util	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).
java.util.concurrent	Utility classes commonly useful in concurrent programming.
java.util.concurrent.atomic	A small toolkit of classes that support lock-free thread-safe programming on single variables.
java.util.concurrent.locks	Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors.
java.util.jar	Provides classes for reading and writing the JAR (Java ARchive) file format, which is based on the standard ZIP file format with an optional manifest file.
java.util.logging	Provides the classes and interfaces of the Java™ 2 platform's core logging facilities.

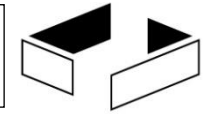
Pakete in BlueJ



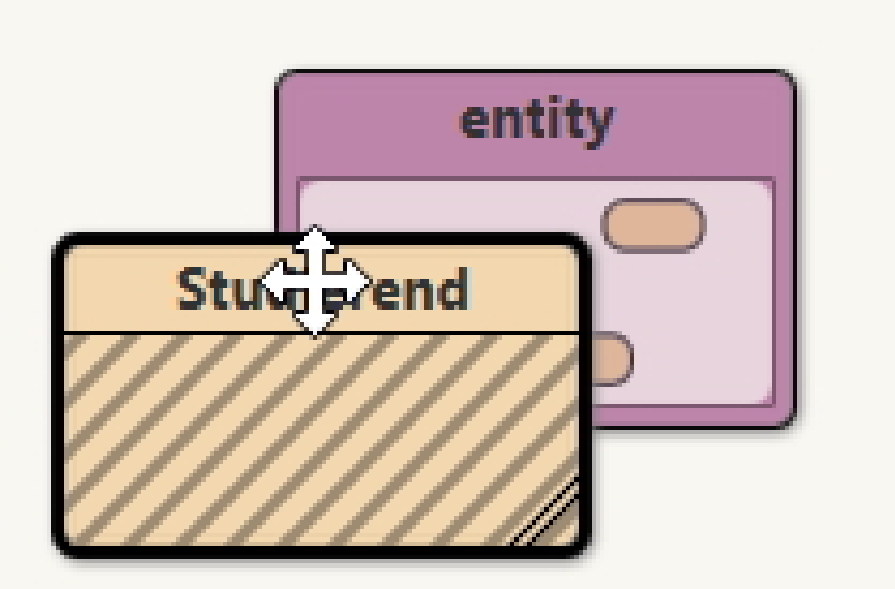
- etwas aufwändig über das Menü



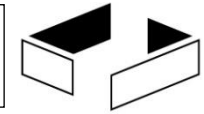
Klassen in Pakete verschieben (1/6)



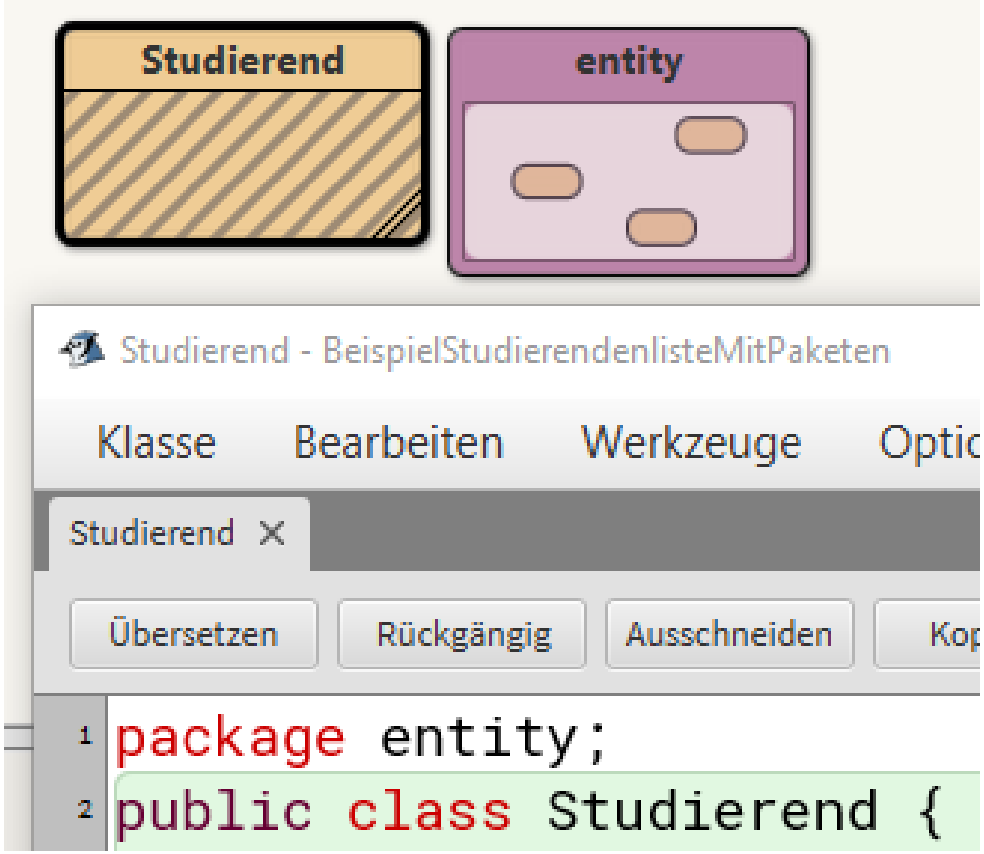
- leider funktioniert hier kein Drag & Drop in BlueJ



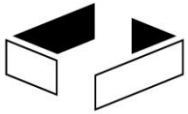
Klassen in Pakete verschieben (2/6)



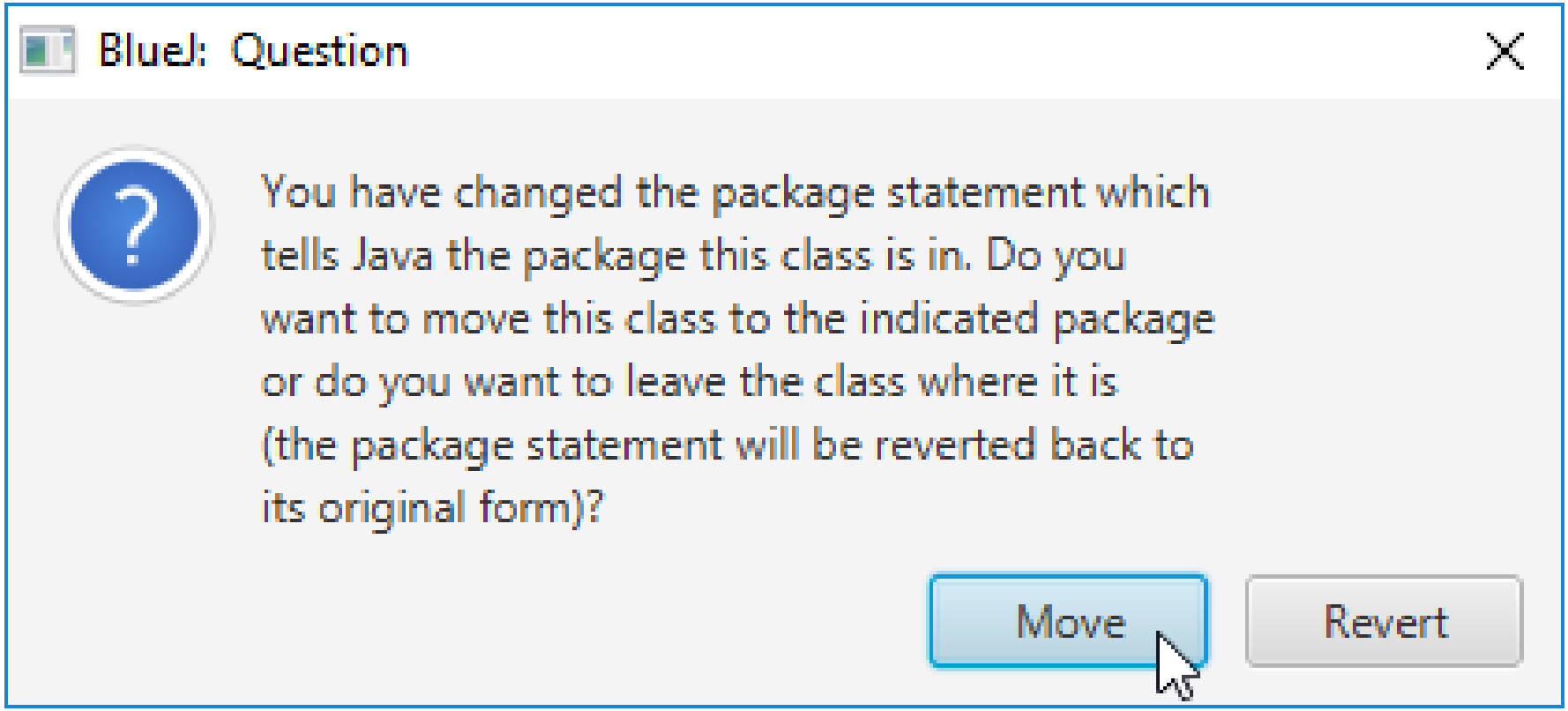
- Trick: Paketdeklaration direkt in Klasse eintragen



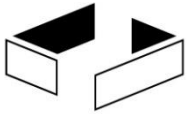
Klassen in Pakete verschieben (3/6)



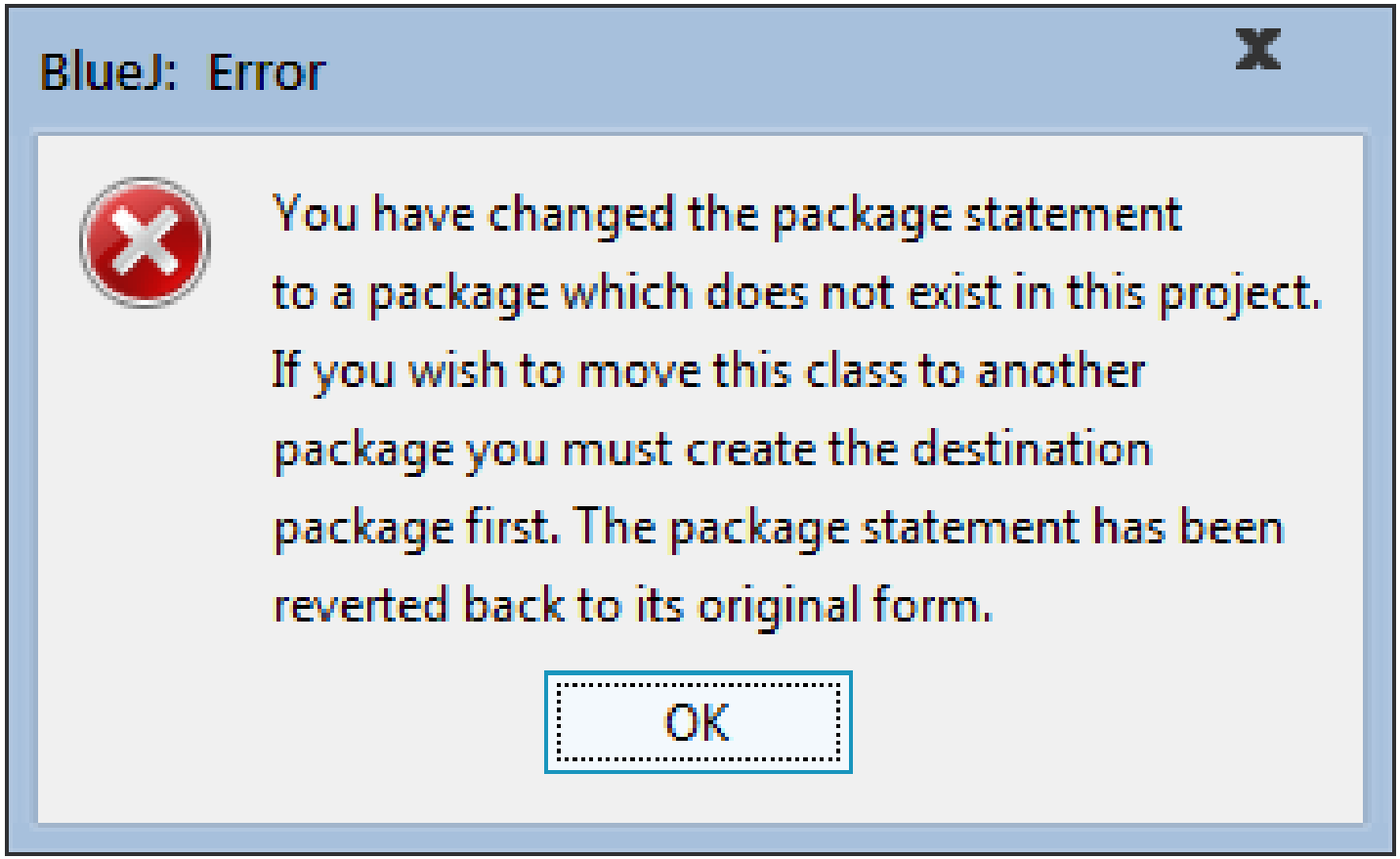
- Frage bei "Compile" bestätigen mit "Move" (Verschieben) bestätigen



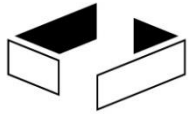
Klassen in Pakete verschieben (4/6)



- Generell müssen Pakete erst angelegt werden, sonst



Klassen in Pakete verschieben (5/6)



- Später Paket (Ordner) mit Doppelklick öffnen

BlueJ: BeispielStudierendenlisteMitPaketen [entity]

Projekt Bearbeiten Werkzeuge Ansicht Hilfe

Neue Klasse...

Übersetzen

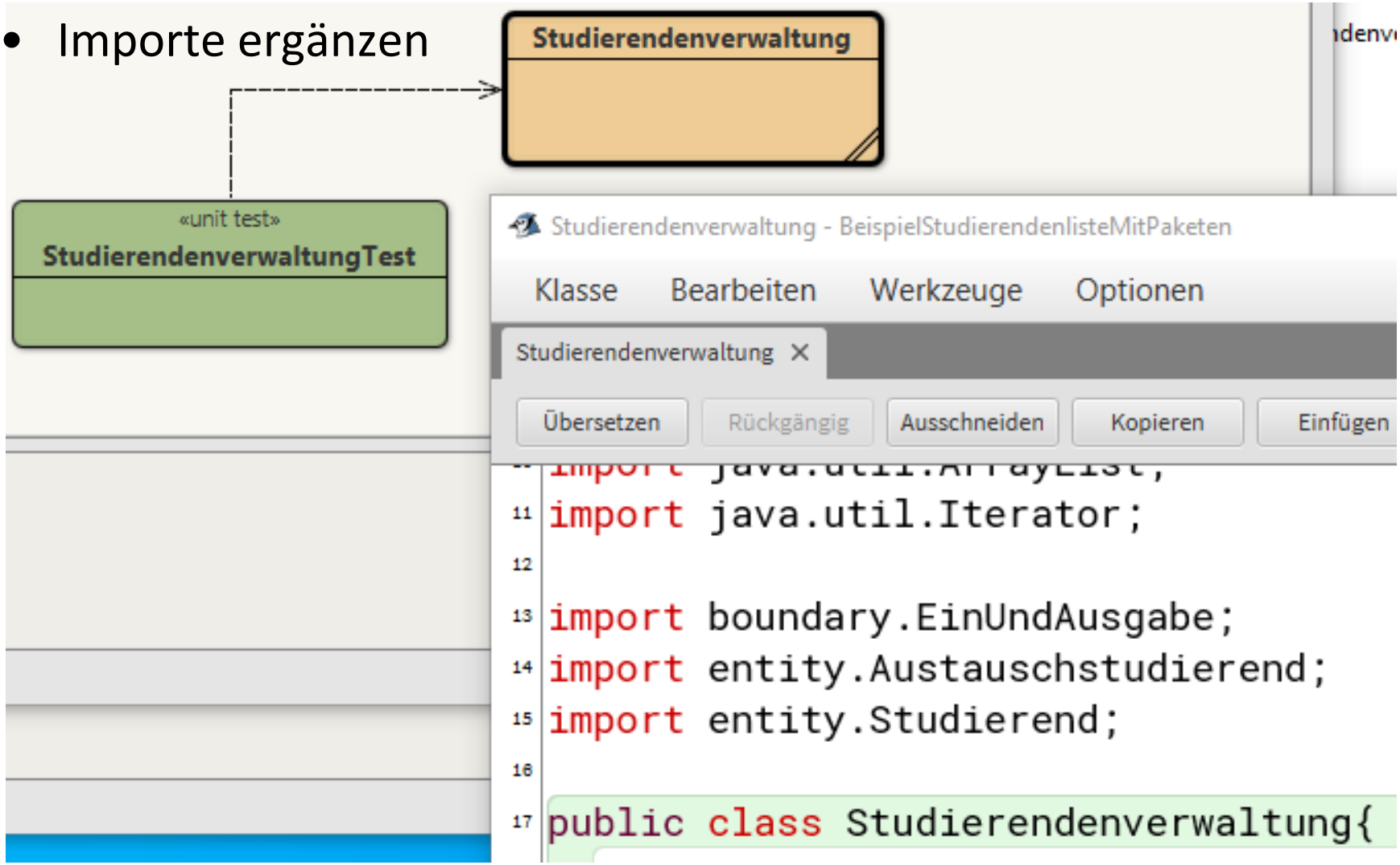
Teamwork

<go up>

Studierend

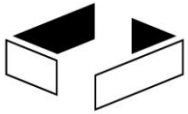
Austauschstudierend

Klassen in Pakete verschieben (6/6)



- Importe ergänzen

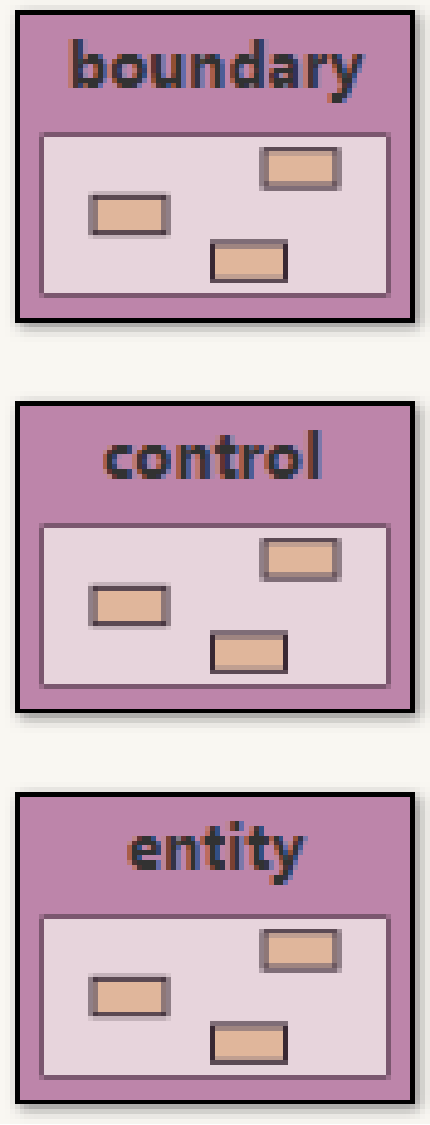
Beispiel: einfache Software-Architektur (1/3)



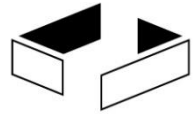
- bisher bei Studierendenverwaltung alle Klassen in einem (dem default-) Paket

eine typische Aufteilung:

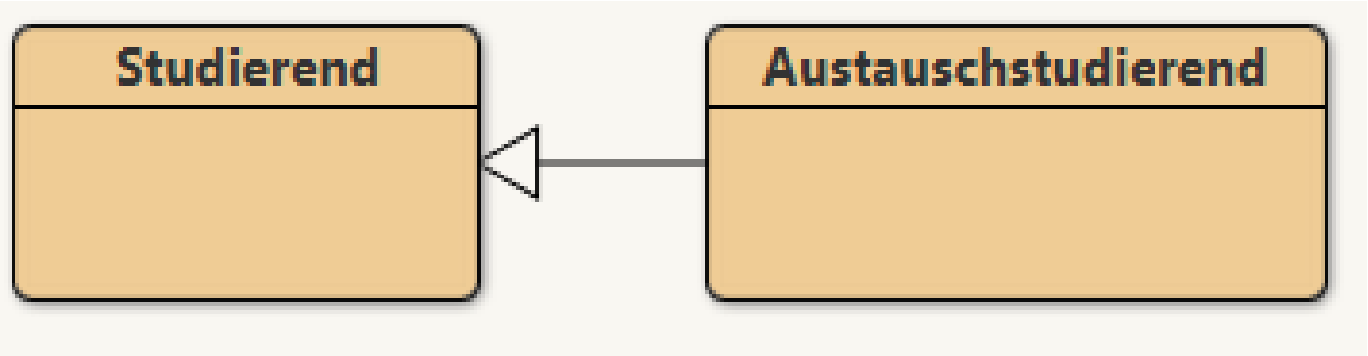
- entity-Paket: alle zu verwaltenden Datenklassen (typisch fast nur get- und set- Methoden)
- control-Paket: Verwaltungsklassen, die Sammlungen von Entity-Objekten verwalten; Zugriff auf Entity-Objekte nur über Control-Klassen
- boundary-Paket: externer Zugriff auf Verwaltungsklassen, z. B. Nutzungsoberfläche



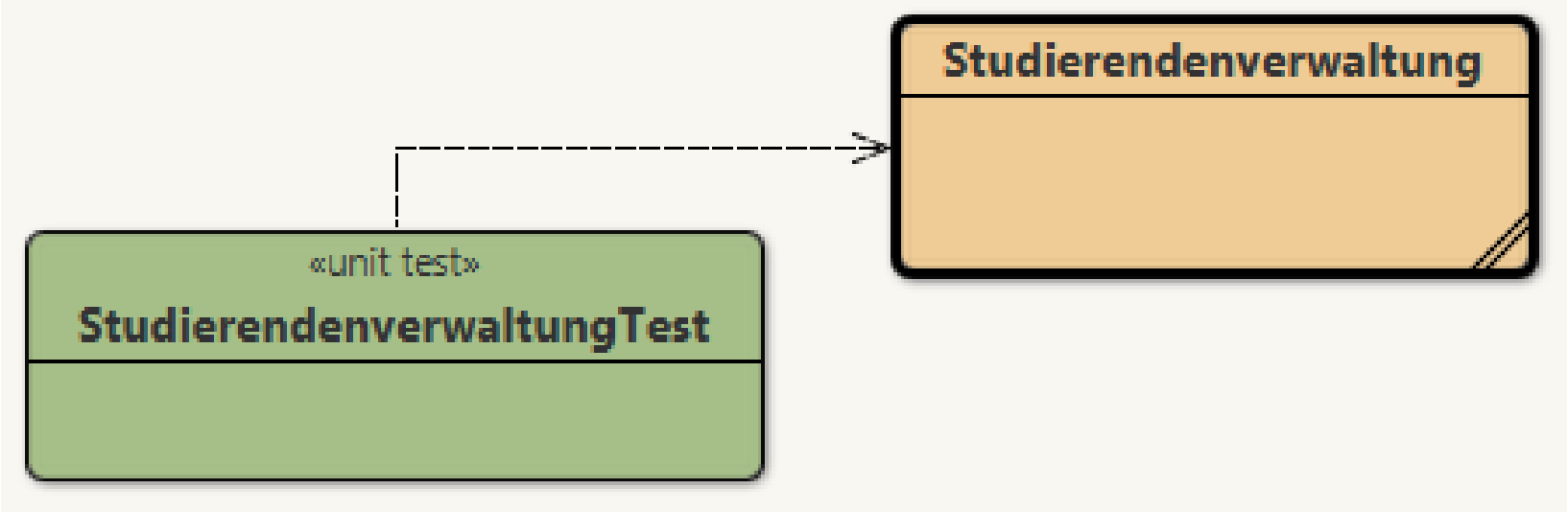
Beispiel: einfache Software-Architektur (2/3)



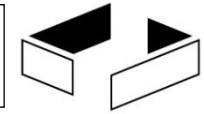
entity



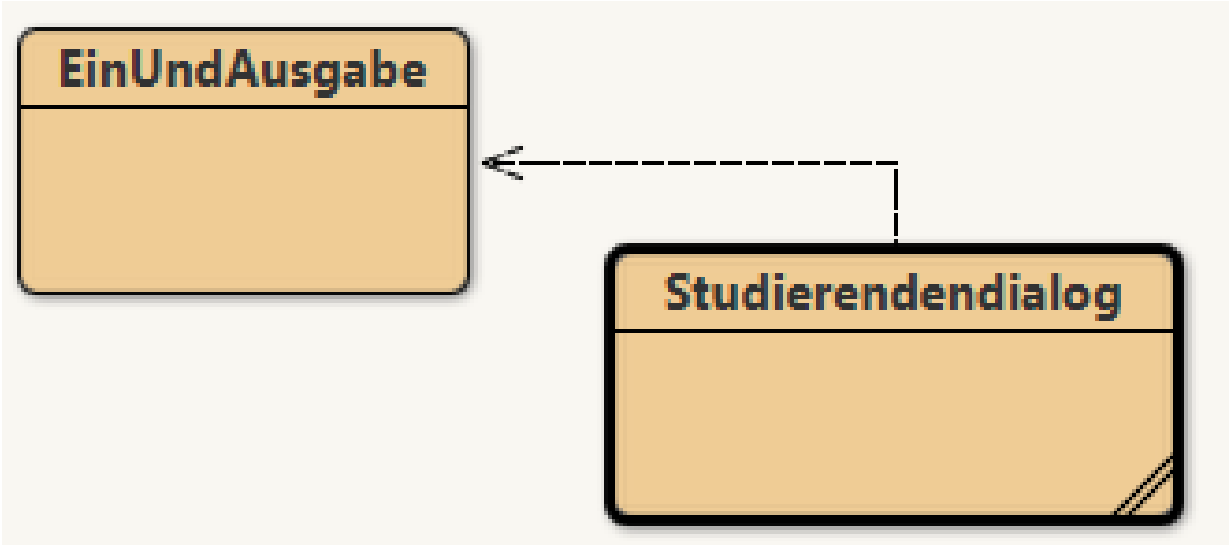
control (Zugriff auf entity-Objekte)



Beispiel: einfache Software-Architektur (3/3)

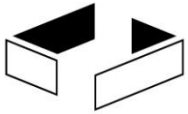


- boundary (Zugriff auf control-Objekte)



- fehlt: Aufteilung von Boundary und Control

Pakete und Imports



```
package boundary;
import control.Studierendenverwaltung;

public class Studierendendialog {
    private Studierendenverwaltung verwaltung
                                = new Studierendenverwaltung();
```

```
package control;
import entity.Studierend;
import entity.AustauschStudierend;
// weitere Imports

public class Studierendenverwaltung {
    private ArrayList<Studierend> studierende;
    private String semester = "Semester XXXX";
```

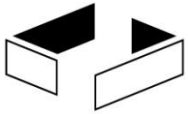
Aufteilung in Boundary und Control-Funktionalität (1/2)

```
//alt in Studierendenverwaltung
public void anzahlAustauschStudierendeAusgeben() {
  EinUndAusgabe io = new EinUndAusgabe();
  io.ausgeben("Es gibt "
          + this.anzahlAustauschStudierende()
          + " Austauschstudirende\n");
}

public int anzahlAustauschstudierende() {
  int ergebnis = 0;
  for (Studierend s : this.studierende) {
    ergebnis = ergebnis + s.zaehlenAlsAustauschstudierende();
  }
  return ergebnis;
}
```

Aufteilung in Boundary und Control-Funktionalität (2/2)

```
// neu: boundary (StudierendenDialog)
public void anzahlAustauschstudierendeAusgeben() {
  EinUndAusgabe io = new EinUndAusgabe();
  io.ausgeben("Es gibt "
        + this.verwaltung.anzahlAustauschStudierende()
        + " Austauschstudierende\n");
   }

// neu: control (Studierendenverwaltung)
public int anzahlAustauschstudierende() {
  int ergebnis = 0;
  for (Studierend s : this.studierende) {
    ergebnis = ergebnis + s.zaehlenAlsAustauschstudierende();
  }
  return ergebnis;
}
```

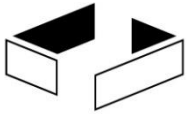
alle Studierenden in Control-Klasse



```
public ArrayList<String> studierendeAusgeben() {
    ArrayList<String> ergebnis = new ArrayList<String>();
    for (Studierend s : this.studierende) {
       ergebnis.add(s.toString());
    }
    return ergebnis;
 }
```

- Beispiel zeigt deutlich, dass Boundary-Klassen keinen direkten Zugriff auf Entity-Klassen erhalten
- (Anmerkung: Ansatz wird nicht immer so konsequent eingehalten)
- Ansatz erlaubt leichte Erstellung einer grafischen Oberfläche

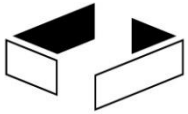
Ergänztes Main-Paket



```
package main;

import boundary.Studierendendialog;

public class Main{

  public static void main(
                    String[] arg){
    Studierendendialog s =
        new Studierendendialog();
    s.steuern();
  }
}
```

- prog1BeispielStudierendenlisteMitPaket
  - src
    - boundary
      - EinUndAusgabe.java
      - Studierendendialog.java
    - control
      - Studierendenverwaltung.java
      - StudierendenverwaltungTest.java
    - entity
      - Austauschstudierend.java
      - Studierend.java
    - main
      - Analyse.java
      - Main.java
  - JRE System Library [java]
  - JUnit 5
  - studis.xml

Kompilieren und Ausführen in der Konsole



Administrator: Eingabeaufforderung - java main.Main

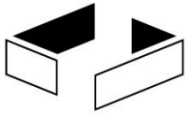
```
F:\workspaces\BluejWork\bsp2>dir
 Datenträger in Laufwerk F: ist Volume
 Volumeseriennummer: 88AE-E635

 Verzeichnis von F:\workspaces\BluejWork\bsp2

06.12.2018  13:08    <DIR>          .
06.12.2018  13:08    <DIR>          ..
06.12.2018  13:04    <DIR>          boundary
06.12.2018  13:04    <DIR>          control
06.12.2018  13:04    <DIR>          entity
06.12.2018  13:08    <DIR>          main
               0 Datei(en),              0 Bytes
               6 Verzeichnis(se), 2.115.863.416.832 Bytes frei

F:\workspaces\BluejWork\bsp2>javac main\Main.java

F:\workspaces\BluejWork\bsp2>java main.Main
***Semester XXXX***
Nächste Aktion:
 (0) Programm beenden
```

Beispiel

Video

Entwicklungsumgebungen

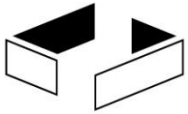
Programmentwicklung in Java



- Grundsätzlich reicht ein beliebiger Texteditor zur Programmerstellung, der Inhalt unformatiert speichert
- Klasse K muss in Datei K.java stehen
- java und javac benötigt

- Problem: viele projekteinheitlich zu lösende Aufgaben müssen individuell geregelt werden
  - welche Einrückungen
  - woher kommen benötigte Klassen anderer
  - wer kompiliert Gesamtprojekt
  - wer erstellt wie die Dokumentation
  - … … …

Entwicklungsumgebung BlueJ

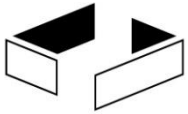


BlueJ übernimmt einige typische Aufgaben:

- Code-Formatierung (Einrückungen)
- Syntax-Highlighting von Schlüsselwörtern
- Einfaches Manövrieren in Klassen
- Unmittelbare Möglichkeit zur Ausführung
- Testklassendefinition und Testausführung (JUnit)
- Einfach Debug-Möglichkeiten

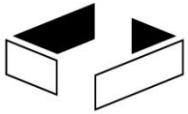
- aber, Entwicklungsumgebungen für Profis können wesentlich mehr
- aber, nur BlueJ kann direkt mit Objekten kommunizieren
- Fazit: BlueJ sehr gut zum Lernen geeignet, da man gezwungen wird, alles selbst zu tippen; für größere Projekte eher ungeeignet (ab jetzt nicht mehr als Standardwerkzeug nutzen)

Entwicklungsumgebung Eclipse



- Editor mit Syntax-Highlighting und Formatierung
- Anzeige aller Fehler
- einfaches Starten von Programmen
- einfaches Einbinden anderer Programme und Bibliotheken
- einfaches Ausführen von JUnit-Tests
- viele Möglichkeiten zur Code-Generierung: get-, set- Methoden, Varianten von Konstruktoren, equals, hashCode, toString
- Einbau umgebender try-catch-Blöcke
- mächtige Code-Completion (<Strg>+<Space>)
- einfache Nutzung eines sehr mächtigen Debuggers
- schnelles Manövrieren zu nutzenden und genutzten Methoden
- … … …
- Achtung: man benötigt zunächst nur sehr wenige der vielen Möglichkeiten, Schritt für Schritt vortasten

Beispiel: Studierendenverwaltung (1/4) - Eingabe



eclipseWS - prog1BeispielStudierendenlisteMitPaketen/src/main/Main.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Run | Main (12)

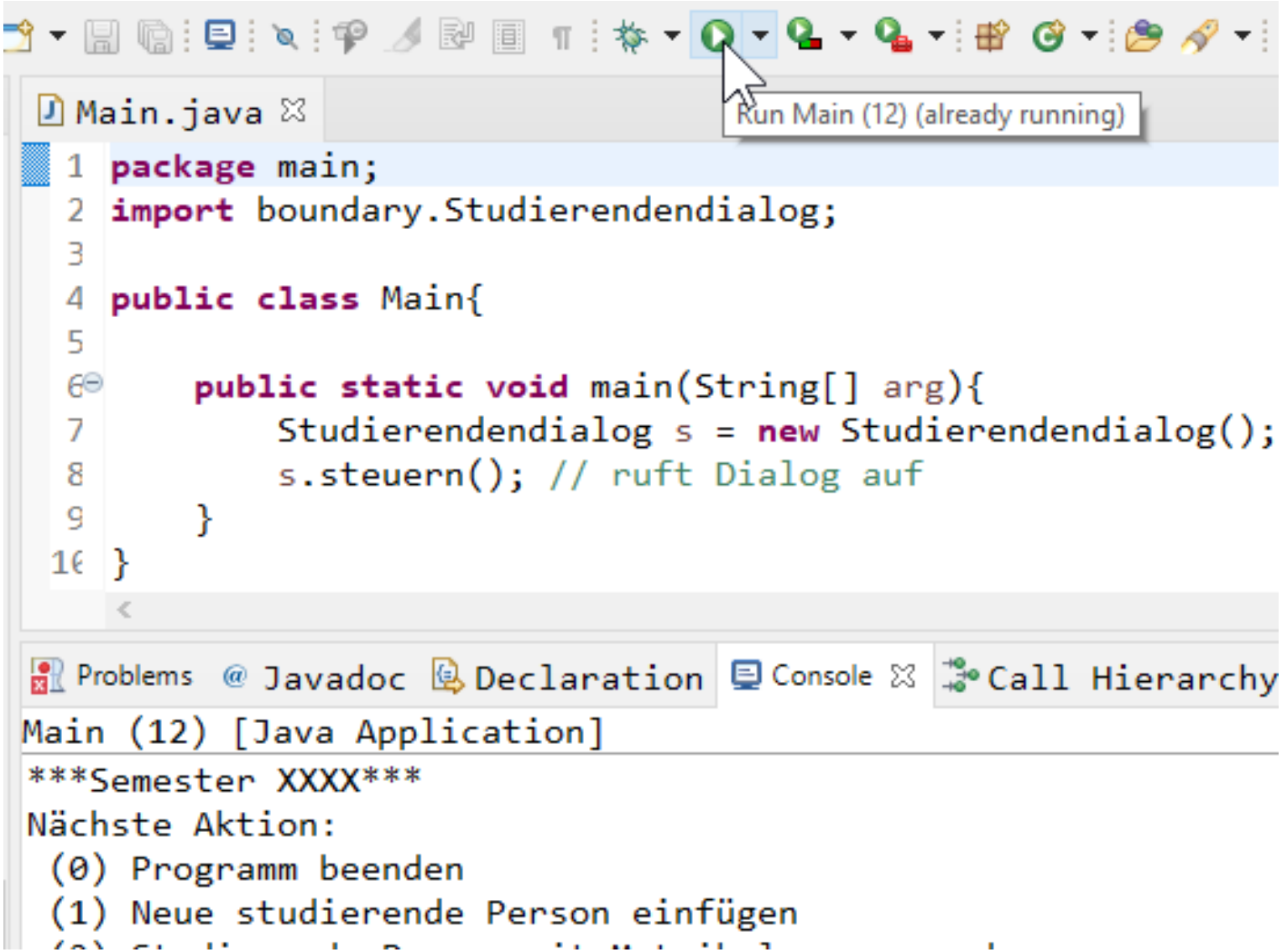
Package Explorer | JUnit

- prog1BeispielStudierendenlisteMitPaketen
  - src
    - boundary
      - EinUndAusgabe.java
      - Studierendendialog.java
    - control
      - Studierendenverwaltung.java
      - StudierendenverwaltungTest.java
    - entity
      - Austauschstudierend.java
      - Studierend.java
    - main
      - Analyse.java
      - Main.java
  - JRE System Library [java]
  - JUnit 5

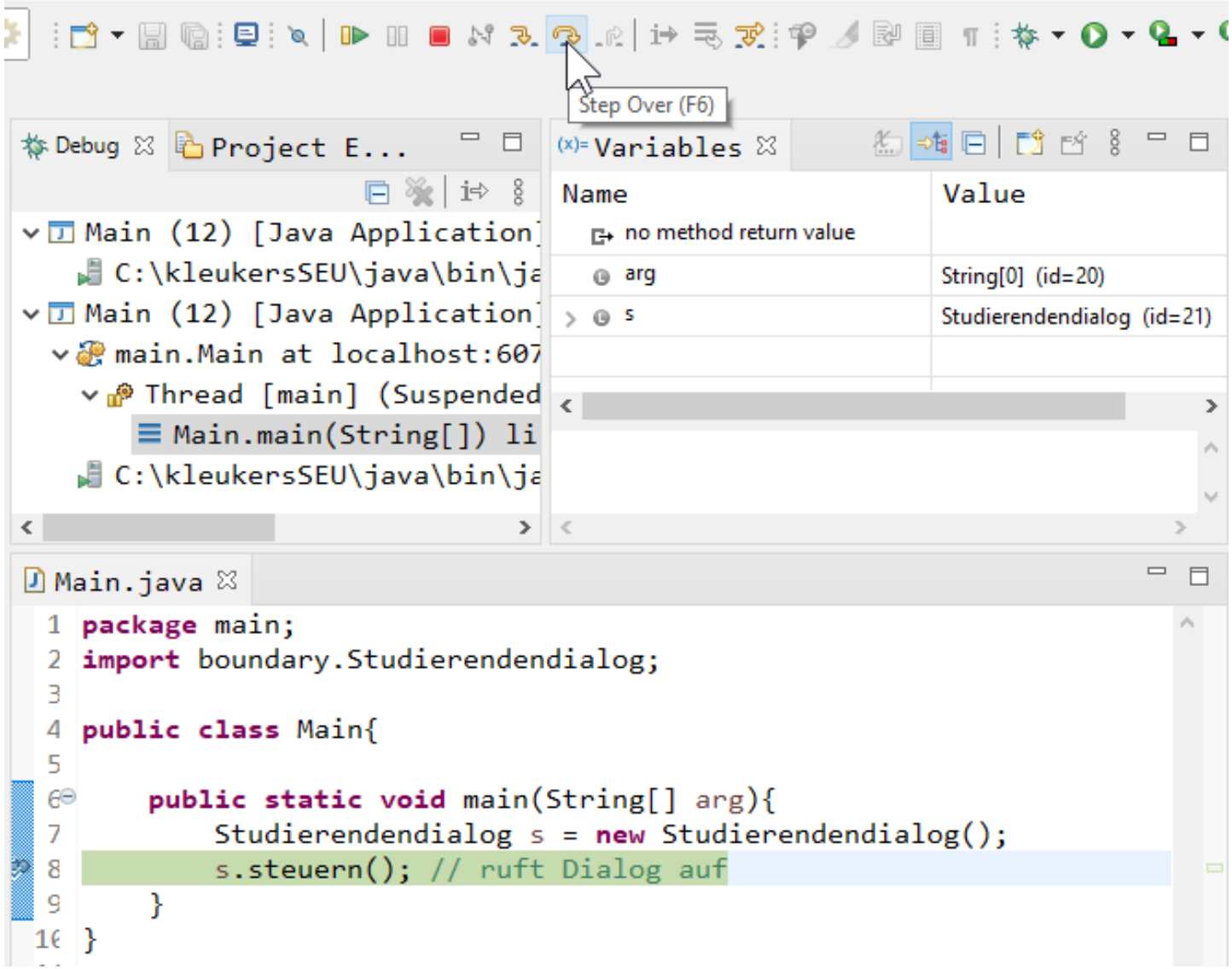
Main.java

```
1  package main;
2  import boundary.Studierendendialog;
3
4  public class Main{
5
6      public static void main(String[]
7          Studierendendialog s = new St
8          s.steuern(); // ruft Dialog a
9      }
10 }
11
```

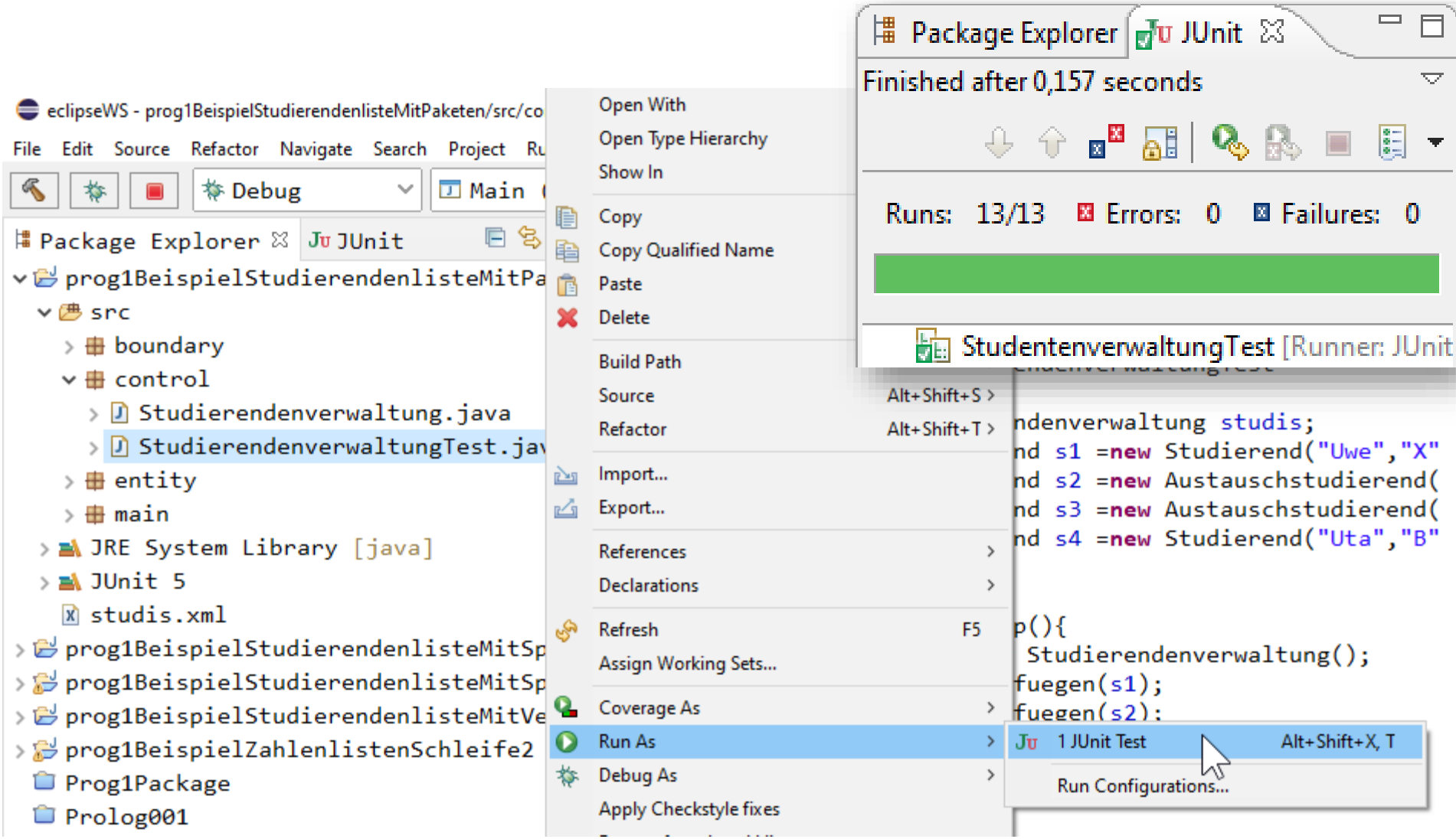
Beispiel: Studierendenverwaltung (2/4) - Ausführen



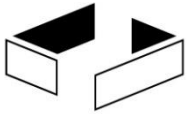
Beispiel: Studierendenverwaltung (3/4) - Debugging



Beispiel: Studierendenverwaltung (4/4) - Testen

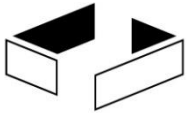


Erweiterbarkeit von Eclipse



- Eclipse kann einfach durch Plug Ins erweitert werden
- viele verschiedene Programmiersprachen und Werkzeuge z. B. zur Datenbankverwaltung möglich
- Vorwegwarnung: Jedes Plug In macht Eclipse langsamer
- Einige weitere Unterstützung bei der Programmierung
- statische Quellcode-Analyse
  - detaillierte Prüfungen von Style-Guides (Einrückungen bis magic Numbers)
  - Bewertung der Komplexität von Programmen (Anzahl der Verzeigungen)
- Messung von Testüberdeckungen (wird Anweisung oder Zweig geprüft)
- … … …

Beispiel: Testüberdeckungsberechnung



Coverage As
Run As
Debug As

1 JUnit Test
Coverage Configurations...

StudierendenverwaltungTest.java | Studierend.java

```
14!
14⊖     public void setVorname(String vorname) {
14:         if(vorname.equals("")) {
14:             this.vorname="<unbekannt>";
14:         } else {
15:             this.vorname = vorname;
15:         }
15:     }
```

Problems | Javadoc | Declaration | Console | Call Hierarchy | Coverage

Element	Cove...	Covered I...	Missed In...	Total Ins...
prog1BeispielStudierendenlist	25,4 %	720	2.118	2.838
src	25,4 %	720	2.118	2.838
control	39,2 %	536	832	1.368
boundary	1,4 %	8	581	589
main	0,0 %	0	405	405
entity	37,0 %	176	300	476
Studierend.java	38,0 %	147	240	387
Austauschstudierend.jav	32,6 %	29	60	89