

Vorlesungsmitschrift des Vorjahres

Objektorientierte Softwareentwicklung

Prof. Dr. rer. nat. Heinrich Faßbender

Aachen, den 11. Oktober 2019

Zusammenfassung

Das vorliegende Skript basiert auf einer Vorlesungsmitschrift zum Modul “Objektorientierte Softwareentwicklung“ im Wintersemester 2016. Die \LaTeX -Version wurde von B.Sc. Marvin Kuhl erstellt und von Herrn Torsten Petersen überarbeitet, bei denen ich mich an dieser Stelle recht herzlich bedanke.

Da es sich um eine Mitschrift handelt, sind die Textpassagen in der Regel nicht als ausformulierte Sätze, sondern stichwortartig aufgeführt. Es enthält keine verbalen Erläuterungen der Vorlesung und ersetzt damit den Vorlesungsbesuch nicht. Außerdem können in der Vorlesung weitere Passagen vorhanden sein, die hier nicht aufgeführt sind. Neben den handschriftlich erstellten Textpassagen sind die verwendeten Folien eingeordnet.

Es sind ferner Kapitel enthalten, deren Inhalt ähnlich schon in früheren Lehrveranstaltungen des Informatik Bachelor-Studiengangs enthalten sind. Diese dienen zum Selbststudium und werden in den Übungen vertieft.

Da einige Kapitel noch einer weiteren Überarbeitung bedürfen, werden diese erst später zur Verfügung gestellt.

Inhaltsverzeichnis

1 Java	1
1.1 Einleitung	1
1.2 Historie	2
1.3 JDK	2
1.4 Java Plattformen	3
1.5 Java-Einordnung im Sprachenkontext	3
1.6 Eigenschaften von Java	4
1.7 Was soll Java sein?	4
2 Imperativer Teil von Java	7
2.1 Rahmen einer Anwendung	7
2.2 Alphabet & Lexikalische Elemente	7
2.3 Primitive Datentypen	9
2.4 Variablen	11
2.5 Sichtbarkeit von Variablen	11
2.6 Konstanten	11
2.7 Ausdrücke	12
2.8 Vorrangregeln bei Ausdrücken	15
2.9 Anweisungen	17
2.10 Felder (Arrays)	19
2.11 Mehrdimensionale Felder	20
2.12 Zugriff auf Feld-Elemente	21
3 Einführung OOS	23
3.1 Phasen bei Softwareentwicklung	23
3.2 Unterschiede OO und imperative Softwareentwicklung	26
4 Konzepte der OOS	33
4.1 Klassen	33
4.2 Objekte	34
4.3 Methoden	36
4.4 Konstruktoren	39
4.5 Vererbung	41
4.6 Zuweisungskompatibilität	43
4.7 Zuweisungskompatibilität auch in Parametern	44
4.8 Überlagerung von geerbten Methoden in Unterklassen	45
4.9 Dynamisches Binden	45
4.10 Klasse Object	46

4.11 Konstruktoren in Vererbungshierarchien	48
4.12 Aggregation	48
4.13 Konstruktoren bei Aggregation	49
4.14 Abstrakte Klassen, Polymorphismus und dynamisches Binden	50
4.15 Pakete	55
4.16 Kollisionen	57
4.17 Zugriffsspezifikationen	57
4.18 Klassenzugriff	58
4.19 Interfaces (Schnittstellen)	59
4.20 Klassenattribute und -methoden	66
4.21 Dokumentation mit Javadoc	68
5 Ausnahmebehandlung (Exceptions)	71
5.1 Grundprinzip des Exception-Handlings	71
5.2 Fangen (Behandlung) von Ausnahmen (catch)	72
5.3 Mehrere catch-Klauseln	75
5.4 Finally-Klausel	77
5.5 Weitergabe von Ausnahmen (throws)	78
5.6 Exception-Hierarchie	79
5.7 Auslösen (Werfen) von Ausnahmen	79
6 Java 5.0	82
6.1 Generische Typen	82
6.2 Aufzählungstypen	86
6.3 Variable Anzahl von Parametern	86
6.4 For Each	87
6.5 Autoboxing	88
7 Serialisierung, Deserialisierung und Streams	89
7.1 Die Klasse ObjectOutputStream	90
7.2 Die Klasse ObjectInputStream	93
7.3 Überlagern von equals	95
8 Unit-Tests	97
9 Oberflächenprogrammierung	108
9.1 Erstellen von Fenstern	108
9.2 Schließen von Fenstern	112
9.3 JavaFX	116
9.3.1 Einführung	116
9.3.2 1. Möglichkeit der Implementierung: Programmatisch	117
9.3.3 2. Möglichkeit der Implementierung: Deklarativ	118
10 Ereignisverarbeitung (Event Handling) in Java und weitere Konzepte	120
10.1 Vorgehen bei Ereignisverarbeitung	120
10.2 Grundlegendes Konzept der inneren Klassen	122
10.3 Verwendung von inneren Klassen bei Listener	124

10.4 Grundlegendes Konzept der anonymen Klassen	125
10.5 Weitere wichtige Methoden von Fensterelementen	126
10.6 Zusammenfassung verschiedener Ereignistypen	127
10.7 Ein Beispiel für MouseEvent	132
10.8 Grundlegendes Konzept der Adapter-Klassen	133
10.9 JavaFX	133
10.9.1 1. Möglichkeit: analog zu Swing	134
10.9.2 2. Möglichkeit: mit fxml-Datei	134
10.9.3 Hinweise zu Ereignissen im SceneBuilder	135
11 Objekttypen und Objektkommunikation	137
11.1 Objekttypen	137
11.2 Objektkommunikation	138
11.3 Beschreibung eines Szenarios	144
11.4 Sequenzdiagramm	144
11.5 Rückgabe der Kontrolle an Steuerungsobjekt	145
12 Netze	146
12.1 Client und Server verteilen	147
12.2 Stream-Sockets zum Datentransfer im Netz	147
12.3 Die Klasse Socket	148
12.4 Die Klasse ServerSocket	149
12.5 Beispiel: Multiplikationsserver	150
13 Design und Design-Pattern	156
13.1 Grob- und Fein-Design	156
13.2 Vorgehensweise	157
13.3 Design-Regeln/-Heuristik	159
13.4 Design-Pattern	161
13.5 Decorate-Muster	166
13.6 Weitere Pattern	169

1 Java

1.1 Einleitung

- als Programmiersprachen-Beispiel wird Java verwendet
- wegen schnellem Praxisbezug, hier zuerst eine allgemeine Einleitung zu Java und im folgenden Kapitel imperatives Programmieren mit Java

Literaturhinweise zu Java

- **Guido Krüger:** *Handbuch der Java-Programmierung*; Addison-Wesley Verlag
gibt es auch online: <http://www.javabuch.de>
- **Bruce Eckel:** *Thinking in Java*; Prentice Hall
gibt es auch online: <http://www.mindview.net/Books/TIJ/>
- **Christian Ullnboom:** *Java ist auch eine Insel*; Galileo Computing;
gibt es auch online:
<http://www.galileocomputing.de/openbook/javainsel/>
- **Dietmar Abts:** *Grundkurs Java*; Vieweg-Verlag, ähnliche Schwerpunkte wie in Vorlesung
- ...

1.2 Historie

Historie

- offizieller Geburtstag: 23. Mai 1995
- hat C++ als am häufigsten genutzte Programmiersprache abgelöst
- ursprüngliche Aufgabe: Prototyp zur Steuerung und Integration von Geräten bauen
- Daseinsberechtigung erst durch Internet: Laden von Programmen (Applets) mit Internetseiten
- hieß zuerst Oak
- Herbst 94: WebRunner neben html auch Oak-Applets
- in Java umbenannt
- richtig berühmt durch Verwendung der Sun-Technologien in Netscape Navigator

1.3 JDK

Java Development Kit (JDK)

- **JDK (Java Development Kit) 1.0** im Januar 1996
- **JDK 1.1** im März 1997,
Beheben einer Reihe von Bugs weitere Funktionalitäten.
- **Java 2 SDK 1.2** im Dezember 1998 veröffentlicht
- **Java 2 SDK 1.3** im Mai 2000 veröffentlicht
deutliche Geschwindigkeitsgewinne bei grafischer Oberfläche und beim Starten von Java-Applikationen.
- **Java 2 SDK 1.4** im Februar 2002 veröffentlicht:
viele Detailverbesserungen, umfangreiche Erweiterungen der Klassenbibliothek und weitere Geschwindigkeitsgewinne
- **Java 2 SDK 5.0** Ende 2004 veröffentlicht:
Generische Typen, typsichere Aufzählungen, erweitertes for, Boxing
- **Java 2 SDK 6.0** Ende 2006 veröffentlicht:
integrierter XML-Parser, GUI-Builder, weitere APIs
- **Java 2 SDK 7** Ende 2011 veröffentlicht:
erste „eigene“ Version von Oracle **jetzt Version 8.x!!!**

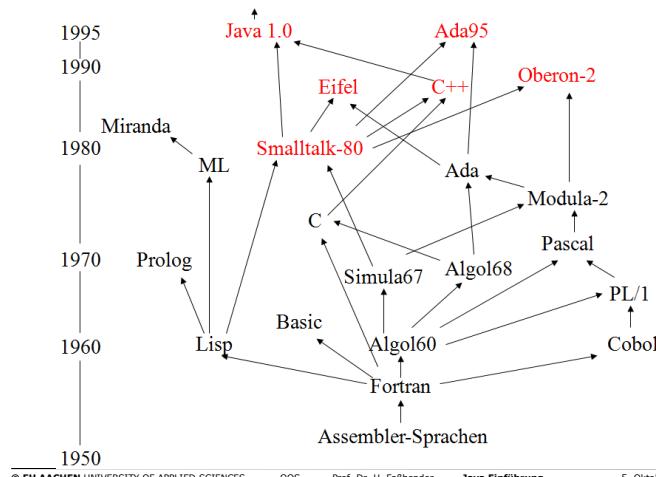
1.4 Java Plattformen

Java-Plattformen

- **3 verschiedene Java 2 Plattformen**
(Virtual Machine + Bibliotheken + Compiler):
- **J2SE (Java 2 Standard Edition)**
hier benutzt
- **J2ME (Java 2 Micro Edition)**
eingeschränkter Sprachstandard für Mobiltelefone und PDAs
- **J2EE (Java 2 Enterprise Edition)**
für verteilte Applikationen
-> vielleicht in Vorlesung verteilte Systeme
- **Download unter: java.sun.com oder oracle.com**
- **zusätzlich: Android Developer Kit**

1.5 Java-Einordnung im Sprachenkontext

Java-Einordnung im Sprachenkontext



1.6 Eigenschaften von Java

Eigenschaften von Java

- Da Java neu entworfen wurde und nicht wie C++ oder Ada95 eine Erweiterung einer imperativen Sprache ist, **viel objektorientierter** (weniger imperative Konzepte).
- Andererseits hat man aus Fehlern reiner objektorientierter Sprachen, wie Smalltalk, wo alles Objekte sind, gelernt und wegen der Pragmatik **auch noch imperative Konzepte** rein genommen.
- **Syntax wie in C und C++**, aber Verzicht auf fehlerträchtige Merkmale wie komplizierte Pointerstrukturen. Zudem keine separaten Header-Dateien.

- Speichermanagement ist automatisch -> kein direkter Zugriff auf Adressen
- Speicherbereinigung automatisch mittels Garbage-Collector

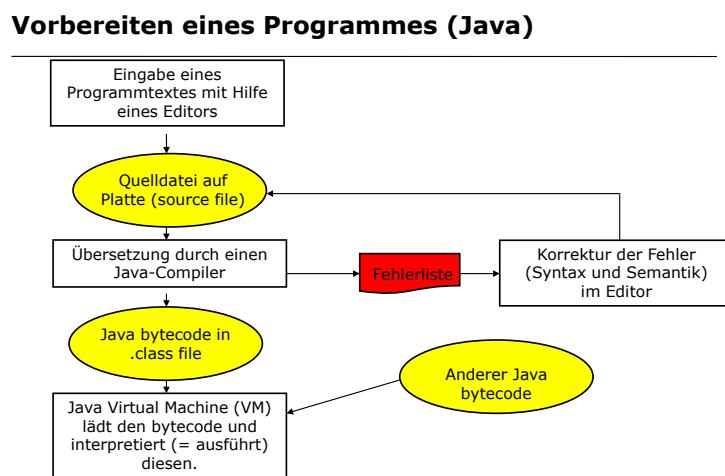
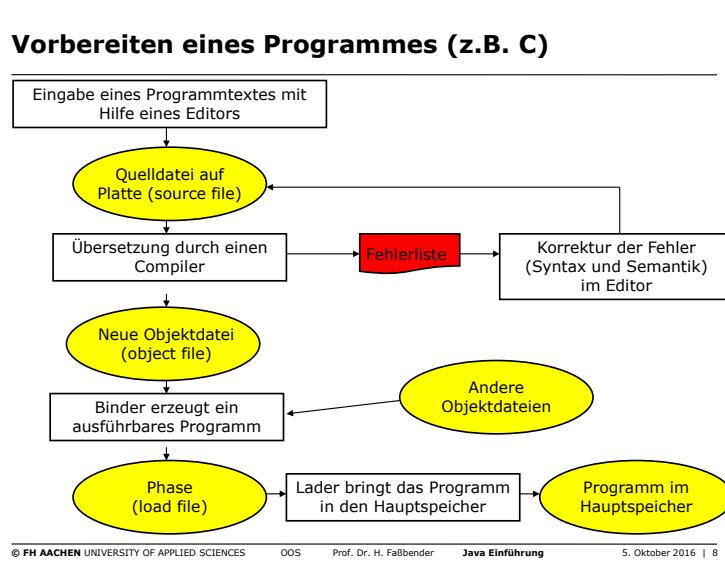
1.7 Was soll Java sein?

Was soll Java sein?

Java soll eine einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architekturneutrale, portable, performante, nebenläufige und dynamische Programmiersprache sein.

Es gibt zwei grundlegende Typen von Java-Programmen:

- 1. Applikation
 - \approx C-Programm



- Unterschiede zu C: kein exe-File, sondern .class-File, welches auf JVM ausgeführt wird (dort wird auch externer Byte-Code integriert)
- 2. Applet
 - läuft in Browser
 - wird in HTML-Seite eingebunden und beim Laden ausgeführt

- Sicherheitsmechanismen achten darauf, dass keine Schweinereien gemacht werden

2 Imperativer Teil von Java

- Java ist eine objektorientierte Sprache, kann aber auch rein imperativ eingesetzt werden
- hier: imperatives Programm in Java und Felder als 1. komplexere Datenstruktur

2.1 Rahmen einer Anwendung

Rahmen einer Anwendung

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



muss in Datei **Anwendung.java** abgespeichert werden.

Unterscheidung zwischen Groß-/Kleinschreibung!!!

2.2 Alphabet & Lexikalische Elemente

- Eingabezeichen
 - Alphabet: Unicode-Zeichensatz (Zusammenfassung aller internationalen Zeichensätze, 30.000 verschiedene Zeichen -> 2 Bytes)
- Kommentare
 - // leitet einen einzeiligen Kommentar ein
 - /* (...) */ stellt einen mehrzeiliger Kommentar dar

- zusätzlich gibt es einen Dokumentationskommentar (wird automatisch in ein HTML-Dokument übersetzt):

Dokumentationskommentar (Beispiel)

```
/*
 * <p>Kommentardemo </p>
 * <p>Mittels dieser Klasse soll javadoc demonstriert werden! </p>
 * @author Heinz Faßbender
 * @version 1.0
 * @see Anwendung1.java
 */
public class Kommentar {
    /**
     * Jetzt kommt eine Methode
     * @param param1 ist der erste Parameter
     * @param param2 ist der zweite Parameter
     * @return ist das Produkt
     */
    public int meth1(int param1, int param2) {
        return (param1 * param2);
    }
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Imperatives Java 5. Oktober 2016 | 2

- Bezeichner und Schlüsselwörter (siehe Programmierrichtlinien)

- bestehen aus einer Sequenz von Zeichen (beliebig lang) und müssen mit einem der folgenden Zeichen beginnen: A-Z, a-z, _, \$
- ist Bezeichner, wenn es kein Schlüsselwort ist

Schlüsselwörter in Java

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue	goto	package	this	

Später: neue Schlüsselwörter ab Java 5.0

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Imperatives Java 5. Oktober 2016 | 3

Unterschiede zu C:

- kein Präprozessor

- bei Java: String-Verkettung durch "+"

primitive Datentypen

Typname	Länge	Wertebereich	Default
boolean	1	true, false	false
char	2	Alle Unicode-Zeichen	\u0000
byte	1	-2 ⁷ ...2 ⁷ -1	0
short	2	-2 ¹⁵ ...2 ¹⁵ -1	0
int	4	-2 ³¹ ...2 ³¹ -1	0
long	8	-2 ⁶³ ...2 ⁶³ -1	0
float	4	+/-3.40282347 * 10 ³⁸	0.0
double	8	+/-1.79769313486231570 * 10 ³⁰⁸	0.0

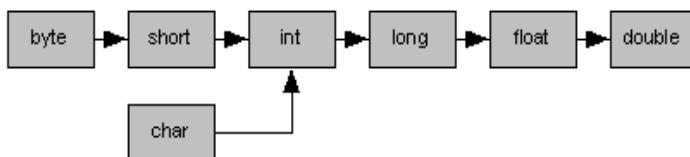
2.3 Primitive Datentypen

- werden auf Stack gespeichert, statt wie bei Objekten auf Heap -> effizienter
- Standardmethoden sind nicht vorhanden
- besitzen feste Größe (kein sizeof() notwendig)
- haben immer definierte Werte (am Anfang immer initialisiert)
- daneben gibt es noch Arrays und Objekttypen (Referenzen auf Objekte)
- bei boolean gibt es kein 1 und 0, sondern nur true und false
- bei char ist eine Transformation nach int (ohne Vorzeichen) möglich
- Literale (einzelne Zeichen): 'a', Strings (Zeichenketten): "Heinz"

Standard-Escape-Sequenzen

Zeichen	Bedeutung
\b	Rückschritt (Backspace)
\t	Horizontaler Tabulator
\n	Zeilenschaltung (Newline)
\f	Seitenumbruch (Formfeed)
\r	Wagenrücklauf (Carriage return)
\"	Doppeltes Anführungszeichen
'	Einfaches Anführungszeichen
\\\	Backslash
\nnn	Oktalzahl nnn (kann auch kürzer als 3 Zeichen sein, darf nicht größer als oktal 377 sein)

Konvertierungen



- **erweiternd:** von links nach rechts
Bsp: `char c = 'h';
int i = c;`
- **einschränkend:** von rechts nach links
Bsp: `int i = 13;
byte b = (byte) i;
i = (int) 13.456
// i wird 13 zugeordnet`

Transformation:

- von links nach rechts (z.B aus char int machen) ohne Probleme
- von rechts nach links mit Cast, aber unvorhersehbare Effekte

Bei ganzzahligen Datentypen:

- vorzeichenbehaftet
- Länge konstant

- KEIN long int, short int, unsigned, signed -> immer feste Länge

Bei Fließkommazahlen:

- z.B. 3.14 , 2f , 1e1 , 6.

2.4 Variablen

- immer typisiert
- Typprüfung passiert zur Compilezeit => Fehler werden früher erkannt
 - z.B. int i = 128; byte b = i; (siehe oben, Fehler wird von Compiler erkannt)

2.5 Sichtbarkeit von Variablen

Wie in C, aber nicht erlaubt: in Unterblöcken gleiche Variable nochmal definieren (später jedoch Attribute überschreibbar)

```
int i = 10;  
{  
    int i = 5;  
}  
-> Fehler
```

2.6 Konstanten

```
final int i; //EINMAL setzbar, nicht zwingend bei Deklaration  
i = 128; //ab jetzt nicht mehr setzbar bzw. veränderbar  
i- -; //Fehler  
Alternativ: final int i = 128; //in einer Zeile
```

2.7 Ausdrücke

- numerische (logische) Berechnungen durchführen
- Spezialfall: Zuweisung: Wert der rechten Seite
- mindestens 1 Operator + 1 oder mehrere Operanden

Zuweisung als Ausdruck nicht gut, weil:

```
int i = 1;
(i=3) * i; // von links nach rechts ausgewertet, Ergebnis = 9
i * (i=3); // Ergebnis = 3, Reihenfolge verändert Ergebnis
```

arithmetische Operatoren

Op.	Bezeichnung	Bedeutung
+	Positives Vorzeichen	+n ist gleichbedeutend mit n
-	Negatives Vorzeichen	-n kehrt das Vorzeichen von n um
+	Summe	a + b -> die Summe von a und b
-	Differenz	a - b -> die Differenz von a und b
*	Produkt	a * b -> das Produkt aus a und b
/	Quotient	a / b -> den Quotienten von a und b
%	Restwert	a % b -> den Rest der ganzzahligen Division von a durch b auch auf Fließkommazahlen anwendbar
++	Präinkrement	++a -> a+1 und erhöht a um 1
++	Postinkrement	a++ -> a und erhöht a um 1
--	Prädekrement	--a -> a-1 und verringert a um 1
--	Postdekrement	a-- -> a und verringert a um 1

relationale Operatoren

Op.	Bezeichnung	Bedeutung
<code>==</code>	Gleich	$a == b \rightarrow \text{true, wenn } a \text{ gleich } b \text{ ist}$
<code>!=</code>	Ungleich	$a != b \rightarrow \text{true, wenn } a \text{ ungleich } b \text{ ist}$
<code><</code>	Kleiner	$a < b \rightarrow \text{true, wenn } a \text{ kleiner } b \text{ ist}$
<code><=</code>	Kleiner gleich	$a <= b \rightarrow \text{true, wenn } a \text{ kleiner oder gleich } b \text{ ist}$
<code>></code>	Größer	$a > b \rightarrow \text{true, wenn } a \text{ größer } b \text{ ist}$
<code>>=</code>	Größer gleich	$a >= b \rightarrow \text{true, wenn } a \text{ größer oder gleich } b \text{ ist}$

SCE: Short Circuit Evaluation

logische Operatoren

Op.	Bezeichnung	Bedeutung
<code>!</code>	Logisches NICHT	$!a \rightarrow \text{false, wenn } a \text{ wahr}$ $\text{true, wenn } a \text{ falsch}$
<code>&&</code>	UND mit SCE	$a \&& b \rightarrow \text{true, wenn } a \text{ und } b \text{ wahr}$ $\text{false, wenn } a \text{ falsch oder } (a \text{ wahr} \& b \text{ falsch})$
<code> </code>	ODER mit SCE	$a b \rightarrow \text{true, wenn } a \text{ wahr oder } (a \text{ falsch} \& b \text{ wahr})$ $\text{false, wenn } a \text{ und } b \text{ falsch}$
<code>&</code>	UND ohne SCE	$a \& b \rightarrow \text{true, wenn } a \text{ und } b \text{ wahr}$ $\text{false, wenn } a \text{ oder } b \text{ falsch (beide ausgewertet)}$
<code> </code>	ODER ohne SCE	$a b \rightarrow \text{true, wenn } a \text{ oder } b \text{ wahr (beide ausgewertet)}$
<code>^</code>	Exklusiv-ODER	$a ^ b \rightarrow \text{true, wenn } a \text{ und } b \text{ unterschiedlich}$

- Bei SCE: zuerst nur 1 Operand ausgewertet, dann ggf. noch 2. Operanden
- Ohne SCE: immer beide auswerten

bitweise Operatoren

Op.	Bezeichnung	Bedeutung
<code>~</code>	Einerkomplement	alle Bits invertiert
<code> </code>	Bitweises ODER	korrespondierende Bits ODER-verknüpft
<code>&</code>	Bitweises UND	korrespondierende Bits UND-verknüpft
<code>^</code>	Bitweises Exklusiv-ODER	korrespondierende Bits Exklusiv-ODER-verknüpft
<code>>></code>	Rechtsschieben mit Vorz.	alle Bits von a um b Positionen nach rechts Falls höchstwertigste Bit gesetzt, auch höchstwertigste Bit des Resultats setzen
<code>>>></code>	Rechtsschieben ohne Vorz.	alle Bits von a um b Positionen nach rechts. höchstwertigste Bit immer auf 0 gesetzt.
<code><<</code>	Linksschieben	alle Bits von a um b Positionen nach links höchstwertigste Bit nicht gesondert behandelt

Zuweisungoperatoren

Op.	Bezeichnung	Bedeutung
<code>=</code>	Einfache Zuweisung	$a = b$ weist a den Wert von b zu; Rückgabewert b
<code>+=</code>	Additionszuweisung	$a += b$ weist a den Wert von $a + b$ zu Rückgabewert $a + b$ als.
<code>-=</code>	Subtraktionszuweisung	$a -= b$ weist a den Wert von $a - b$ zu Rückgabewert $a - b$
<code>*=</code>	Multiplikationszuweisung	$a *= b$ weist a den Wert von $a * b$ zu Rückgabewert $a * b$
<code>/=</code>	Divisionszuweisung	$a /= b$ weist a den Wert von a / b zu Rückgabewert a / b
<code>%=</code>	Modulozuweisung	$a \%= b$ weist a den Wert von $a \% b$ zu Rückgabewert $a \% b$
<code>&=</code>	UND-Zuweisung	$a \&= b$ weist a den Wert von $a \& b$ zu Rückgabewert $a \& b$

Zuweisungoperatoren (Forts.)

Op.	Bezeichnung	Bedeutung
$ =$	ODER-Zuweisung	$a = b$ weist a den Wert von a b zu Rückgabewert a b
$\wedge=$	Exklusiv-ODER-Zuweisung	$a \wedge= b$ weist a den Wert von $a \wedge b$ zu Rückgabewert $a \wedge b$
$<<=$	Linksschiebezuweisung	$a <<= b$ weist a den Wert von $a << b$ zu Rückgabewert $a << b$
$>>=$	Rechtsschiebezuweisung	$a >>= b$ weist a den Wert von $a >> b$ zu Rückgabewert $a >> b$
$>>>=$	Rechtsschiebezuweisung mit Nullexpansion	$a >>>= b$ weist a den Wert von $a >>> b$ zu Rückgabewert $a >>> b$

Typisierungen bei Vorrangregeln

- mögliche Operandentypen:
 - »N« numerische,
 - »I« integrale (also ganzzahlig numerische),
 - »L« logische,
 - »S« String-,
 - »R« Referenz-
 - »P« primitive Typen.
 - »A« alle Typen
 - »V« zeigt an, dass eine Variable erforderlich ist.

Beispiel für einen dreistelligen Operator: $a?b:c$ bedeutet: if a then b else c (a muss boolean sein, b und c müssen vom selben Typ sein)

z.B.: $x>0?x:-x$ bedeutet: Betrag von x

Vorsicht: nur schwer lesbar

2.8 Vorrangregeln bei Ausdrücken

- eine niedrigere Gruppennr. hat Vorrang vor einer höheren Gruppennr.

Vorrangregeln

Gr.	Operator	Typisierung	Ass.	Bezeichnung
1	++	N	R	Inkrement
	--	N	R	Dekrement
	+	N	R	Unäres Plus
	-	N	R	Unäres Minus
	~	I	R	Einerkomplement
	!	L	R	Logisches NICHT
	(type)	A	R	Type-Cast
2	*	N,N	L	Multiplikation
	/	N,N	L	Division
	%	N,N	L	Modulo
3	+	N,N	L	Addition
	-	N,N	L	Subtraktion
	+	S,A	L	String-Verkettung
4	<<	I,I	L	Linksschieben
	>>	I,I	L	Rechtsschieben
	>>>	I,I	L	Rechtsschieben mit Nullexp.

Vorrangregeln (Forts.)

Gr.	Operator	Typisierung	Ass.	Bezeichnung
5	<	N,N	L	Kleiner
	<=	N,N	L	Kleiner gleich
	>	N,N	L	Größer
	>=	N,N	L	Größer gleich
	instanceof	R,R	L	Klassenzugehörigkeit
6	==	P,P	L	Gleich
	!=	P,P	L	Ungleich
	==	R,R	L	Referenzgleichheit
	!=	R,R	L	Referenzungleichheit
7	&	I,I	L	Bitweises UND
	&	L,L	L	Logisches UND ohne SCE
8	^	I,I	L	Bitweises Exklusiv-ODER
	^	L,L	L	Logisches Exklusiv-ODER
9		I,I	L	Bitweises ODER
		L,L	L	Logisches ODER ohne SCE
10	&&	L,L	L	Logisches UND mit SCE

Vorrangregeln (Forts. 2)

Gr.	Operator	Typisierung	Ass.	Bezeichnung
11		L,L	L	Logisches ODER mit SCE
12	?:	L,A,A	R	Bedingte Auswertung
13	=	V,A	R	Zuweisung
	+=	V,N	R	Additionszuweisung
	-=	V,N	R	Subtraktionszuweisung
	*=	V,N	R	Multiplikationszuweisung
	/=	V,N	R	Divisionszuweisung
	%=	V,N	R	Restwertzuweisung
	&=	N,N u. L,L	R	Bitweise (Logische) -UND-Zuw.
	=	N,N u. L,L	R	Bitweise (Logische) -ODER-Zuw.
	^=	N,N u. L,L	R	Bitweise (Logische) -Exklusiv-ODER-Zuw.
	<<=	V,I	R	Linksschiebezuweisung
	>>=	V,I	R	Rechtsschiebezuweisung
	>>>=	V,I	R	Rechtsschiebezuweisung mit Nullexpansion

- Im Zweifel bei Vorrangregeln Klammern setzen

2.9 Anweisungen

- leere Anweisung: ;
- Block: { ...; ...; ...; }
- Variablendeclarationen und Ausdrucksanweisungen (z.B. i++ ;)
in C ist auch 2+4; eine Anweisung, in Java nicht!
- if, switch wie in C

Dangling Else:

```
if(a)
    if(b)
        s1;
    else
        s2;
```

oder:

```
if(a)
    if(b)
        s1;
else
    s2;
```

Bezieht sich immer auf noch offenes (vorangehendes) if (hier also Variante 1), kann nur auftreten, wenn die Klammerung weggelassen wird

- while, do-while, for wie in C
- break und continue
 - break: springt zu erster Anweisung HINTER der Schleife (ggf. mit Label -> dann hinter die Schleife, die gelabelt ist)
 - continue: springt an Schleifenrumpfende (ggf. mit Label -> dann an Schleifenrumpfende der gelabelten Schleife) und fährt mit nächster Iteration fort

continue und break mit Label

```
loop1:
for (int i = 1; i <= 5; ++i)
{
    for (int j = 1; j <= 10; ++j)
    {
        System.out.println("Springen!");
        break loop1;
    }
}
```

Schleifendurchläufe mit break, mit Label (so wie auf Folie): 1

Schleifendurchläufe mit break, ohne Label (nur break;): 5

Schleifendurchläufe mit continue, mit Label: 5

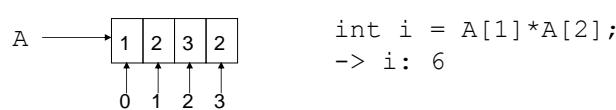
Schleifendurchläufe mit continue, ohne Label (nur continue;): 50

2.10 Felder (Arrays)

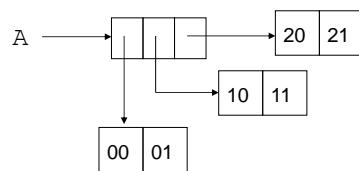
- sind Objekte (d.h. keine neuen imperativen Datentypen)
- Array-Variablen sind Referenzen
- Attribute und Methoden vorhanden
- werden zur Laufzeit auf Heap erzeugt
- lineare Folge von Elementen gleichen Typs

Felder

eindimensional



mehrdimensional



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Imperatives Java 5. Oktober 2016 | 18



Deklaration und Initialisierung

1. Deklaration

`int[] a;` `a` →

2. Erzeugen eines Arrays und Zuweisung an Variable

`a = new int[5];` `a` →

1. und 2. geht auch gemeinsam

`int[] a = new int[5];`

alternativ zu new kann man auch mit Literalen initialisieren

`int[] x = {1,2,3,4,5};` `x` →

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Imperatives Java 5. Oktober 2016 | 19



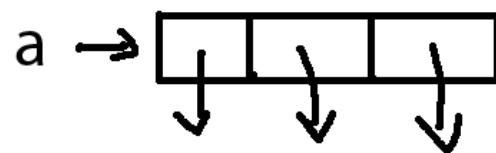
Felder in Java sind semi-dynamisch: Größe zur Laufzeit festlegen, dann aber nicht mehr veränderbar

`x.length` -> ergibt 5, `length` = Attribut, keine Methode

2.11 Mehrdimensionale Felder

- hierarchische Struktur
- 2 oder mehr Paare eckiger Klammern
- `int[][] a = new int[3][2];` (siehe Folie)
- Alternativ: `int[][] a = new int[3][];`

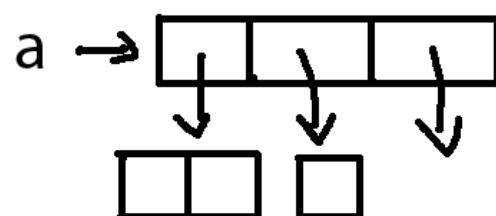
-> Erzeugt drei leere Zeiger:



`a[0] = new int[2];`

`a[1] = new int[1];`

-> Erzeugt:



2.12 Zugriff auf Feld-Elemente

- über Index $0 \leq i \leq \text{length}-1$

z.B. $x[3] \rightarrow 4$

- mehrdimensional: $a[2][1] \rightarrow 21$

nicht rechteckiges Feld

```
public class NrFeld {

    public static void main(String[] args) {
        int[][] a = { {0},
                      {1,2},
                      {3,4,5},
                      {6,7,8,9}
                  };

        for (int i=0; i < a.length; ++i) {
            for (int j=0; j < a[i].length; ++j) {
                System.out.print(a[i][j]);
            }
            System.out.println();
        }
    }
}
```

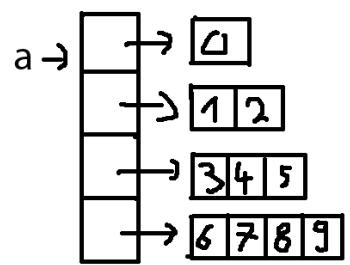
Wie herkömmliche Erzeugung und Initialisierung?

`int[][] a = { {0}, {0,1}, .. } geht`

Erzeugung eines dreieckigen Feldes

```
// Erzeugung eines 1-dim. Feldes mit null-Zeigern
int[][] dreieck = new int[4][];
// Füllungszähler
int füllung = 0;

for (int i = 0; i < dreieck.length; i++) {
// Erzeugung der i-ten Zeile der Länge i
dreieck[i] = new int[i+1];
// Belegung der Felder und Ausgabe
for(int j = 0; j < i+1; j++) {
    dreieck[i][j] = füllung++;
    System.out.print(dreieck[i][j]);
}
// Zeilenumbruch
System.out.println();
}
```

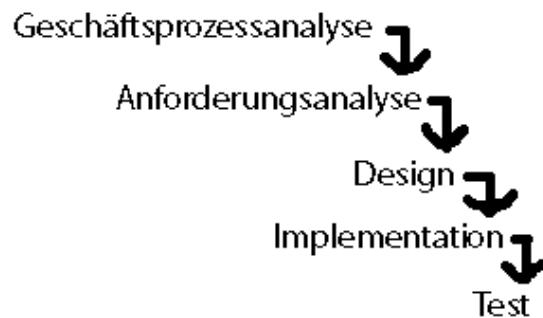


3 Einführung OOS

- jetzt: allgemeine Einführung in die objektorientierte Softwareentwicklung

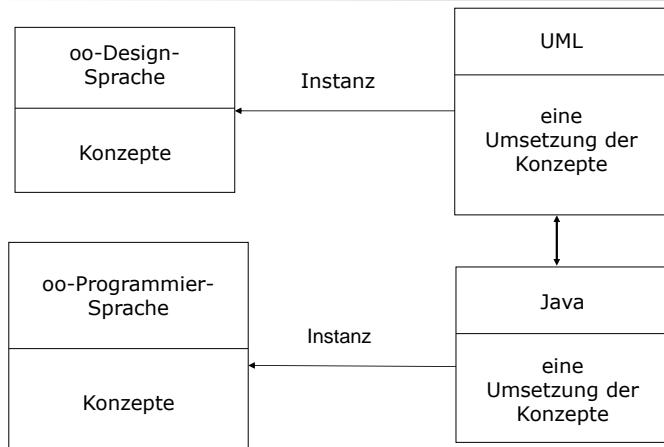
3.1 Phasen bei Softwareentwicklung

Wasserfallmodell:



- zusätzlich bei allen Phasen Projektmanagement und Dokumentation notwendig
- Schwächen:
 - keine Möglichkeit zurückzugehen
 - Fehler in frühen Phasen extrem teuer
 - Wichtig: Rückgang zur vorherigen Phase muss möglich sein => Zyklus
 - beste Lösung: iterativen, inkrementellen Zyklus verwenden

Konzept und Umsetzung



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender OO-Einführung 5. Oktober 2016 | 1

- wichtig: Konzept, da langlebig, jedoch nur verständlich an einer Beispiel-Instanz

Ausschreibung der OMG

OMG = Object Management Group

Zusammenschluss vieler Firmen

Definition von Standards in OOS (zuerst CORBA)

Ziel:

Eine objektorientierte Methode
für alle Phasen der Softwareentwicklung,
die von möglichst vielen verstanden und angewendet
wird.

- wegen Vielfalt der Probleme, kein einheitlicher Prozess möglich
- > **nur Modellierungssprache**

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender OO-Einführung 5. Oktober 2016 | 2

Mitte der 90er Jahre viele OO-Methoden

OO-Methode besteht aus:

- Sprache (Notation)
- Prozess

UML:

- nur noch Notation, kein Prozess
- grafisch (Bild sagt mehr als 1000 Programmzeilen)

Historie der UML

- 3 Methoden-Gurus Rumbaugh, Booch, Jacobson bei Rational Software (jetzt IBM) vereint
- entwarfen 97 Unified Modelling Language UML
- andere Vorschläge zurückgezogen und bei Weiterentwicklung der UML mitgearbeitet
- aktuelle Version >= 2.4.1
- Modellierungssprache zum Spezifizieren, Konstruieren, Visualisieren, Dokumentieren
- unterschiedliche Aspekte durch unterschiedliche Modelle, die in Beziehung stehen können
- Tools oft überladen und sehr einschränkend

Literaturhinweise zur UML

Einführungen:

- M. Hitz, G. Kappel, E. Kapsammer, W. Retschitzegger: *UML at Work -Objektorientierte Modellierung mit UML 2-*; Dpunkt Verlag.
- Bernd Oestereich: *Analyse und Design mit UML 2 -Objektorientierte Softwareentwicklung-*; Oldenbourg.
- Dr. Erler Thomas: *UML*; hbv-Verlag.

Kurzreferenzen:

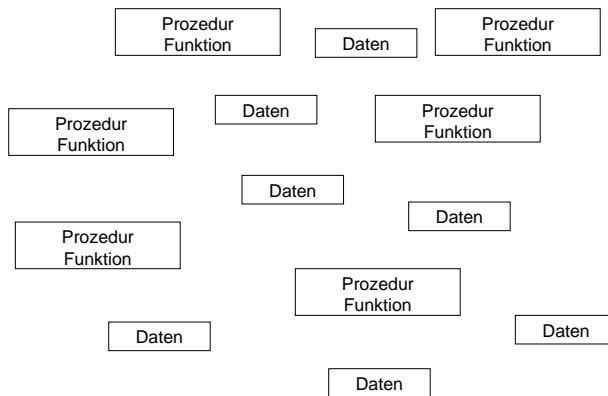
- Bernd Oestereich: *Die UML- Kurzreferenz für die Praxis.* Kurz, bündig, ballastfrei; Oldenbourg.
- Heide Balzert: *UML kompakt. Mit Checklisten;* Spektrum Akademischer Verlag.

Standards:

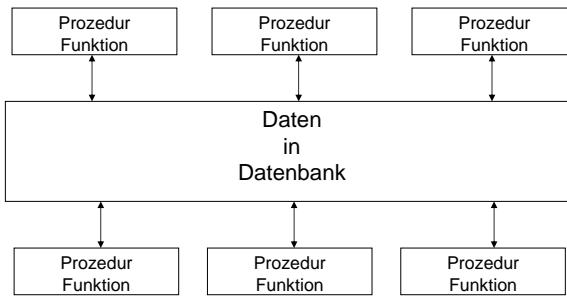
- www.uml.org.

3.2 Unterschiede OO und imperative Softwareentwicklung

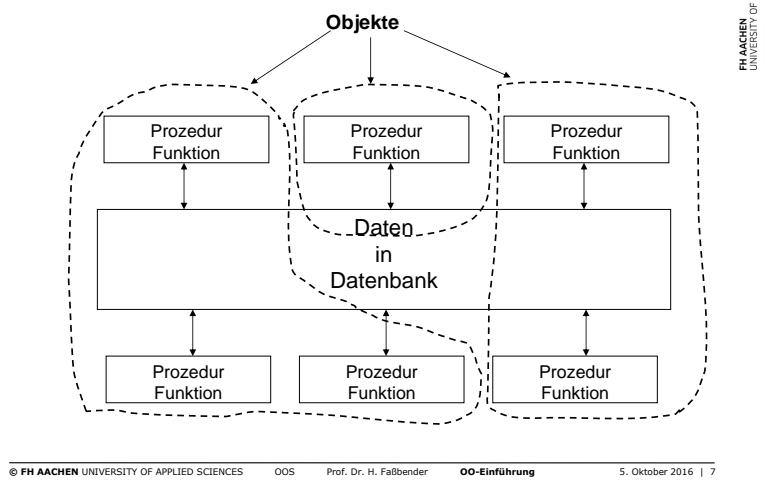
herkömmliche Sprachen



Programme mit Datenbank



objektorientierte Programme

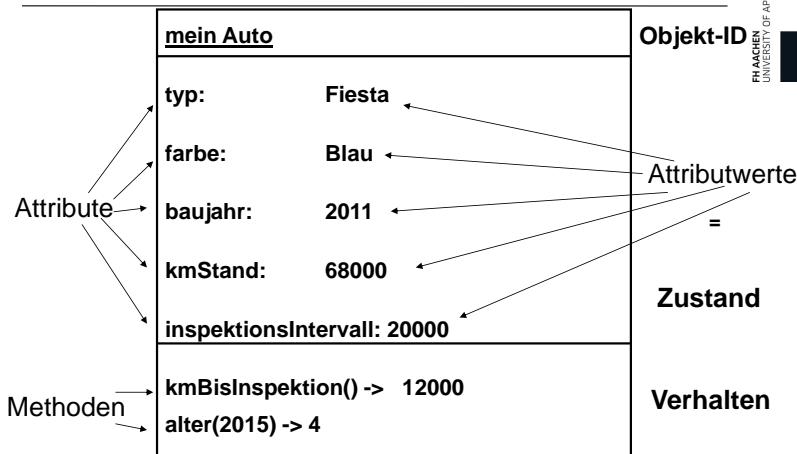


© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender OO-Einführung 5. Oktober 2016 | 7

Konzepte der OOS:

- **1. Kapselung:** Daten und Prozeduren oder Funktionen werden zu Objekten zusammengefasst. Häufig: Kein direkter Zugriff mehr auf Daten (nur noch über getter/setter-Methoden)

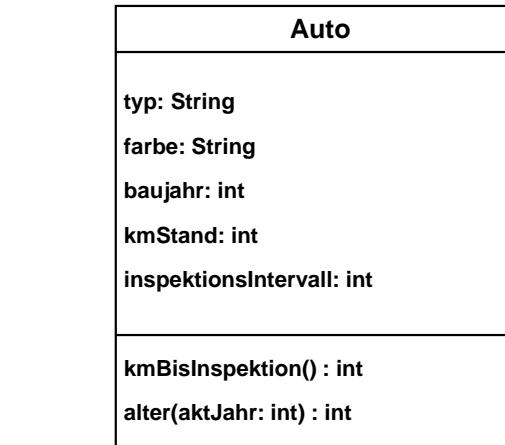
Objekt (Beispiel)



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender OO-Einführung 5. Oktober 2016 | 8

- **2. Abstraktion:** Übergang vom Objekt zur Klasse (Klasse: Schablonen, Stempel): fasst alles, was Objekten mit gleicher Struktur gemeinsam ist, zusammen

Klasse (Beispiel)



Attributdeklaration: String typ;

Methodenimplementierung:

```
int kmBisInspektion() {
    return(inspektionsIntervall-(kmStand%inspektionsIntervall));
}
```

Weitere Konzepte am Unterschied zwischen herkömmlicher imperativer Programmierung zur OO-Programmierung!

Flächenberechnung (imperativ)

Datenstrukturen: Definition Rechteck $(x_1, y_1), (x_2, y_2)$
 Definition Kreis $(x, y), r$

Funktionen: Rechtecksfläche: Rechteck \Rightarrow Flächeninhalt
 Kreisfläche: Kreis \Rightarrow Flächeninhalt

Fläche(f) = IF f is Rechteck THEN Rechtecksfläche(f);
 IF f is Kreis THEN Kreisfläche(f);

Erweiterung um Dreieck:

Funktionalitätserweiterung (imperativ)

Datenstrukturen: Definition Rechteck $(x_1, y_1), (x_2, y_2)$
 \Rightarrow Definition Kreis $(x, y), r$
 Definition Dreieck $(x_1, y_1), (x_2, y_2), (x_3, y_3)$

Funktionen: Rechtecksfläche: Rechteck \Rightarrow Flächeninhalt
 Kreisfläche: Kreis \Rightarrow Flächeninhalt
 \Rightarrow Dreiecksfläche: Dreieck \Rightarrow Flächeninhalt

Fläche(f) = IF f is Rechteck THEN Rechtecksfläche(f);
 IF f is Kreis THEN Kreisfläche(f);
 \Rightarrow IF f is Dreieck THEN Dreiecksfläche(f);

=> man muss an drei Stellen etwas ändern

Flächenberechnung (oo)

	Rechteck	Kreis
DS: Attribute	Definition	Definition
Methoden	flächeninhalt()	flächeninhalt()

Aufruf: **objekt.flächeninhalt()**

Erweiterung um Dreieck:

Funktionalitätserweiterung (oo)

	Rechteck	Kreis	Dreieck
DS: Attribute	Definition	Definition	Definition
Methoden	flächeninhalt()	flächeninhalt()	flächeninhalt()

Aufruf: **objekt.flächeninhalt()**



=> jetzt muss man nur noch an einer Stelle etwas ändern

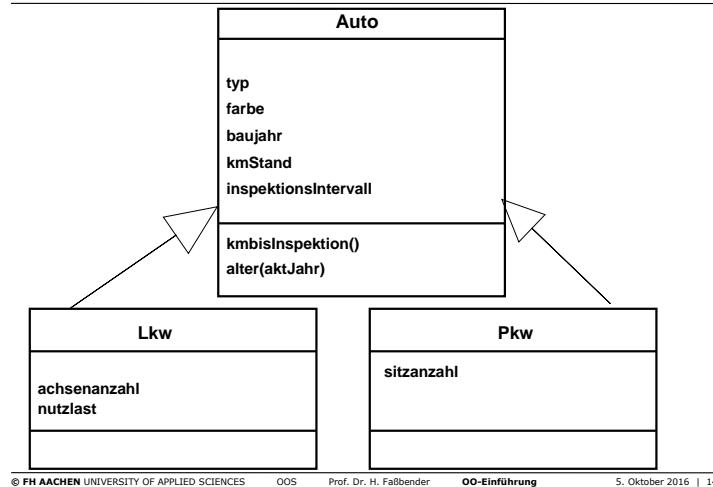
=> Lokalitätsprinzip: Single Point of Maintenance

=> braucht hierzu weitere Grundkonzepte

- **3. Polymorphismus** (Überlagerung): Eine Methode mit gleicher Signatur (Argumente gleich) und gleichem Namen in verschiedenen Klassen verschieden implementierbar

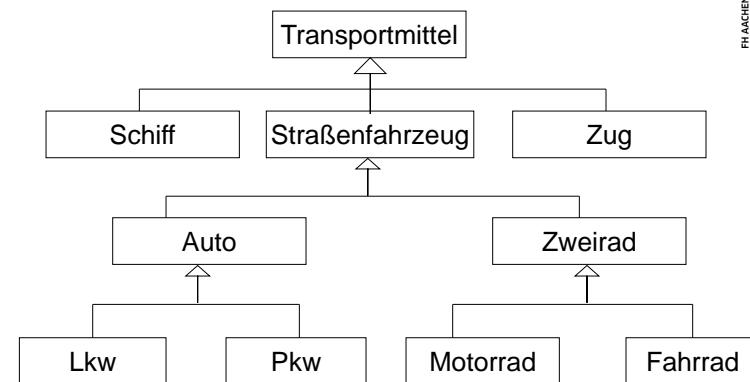
- **4. Dynamisches Binden:** Beim Aufruf einer Methode kann für dynamisch erzeugte Objekte erst zur Laufzeit entschieden werden, von welchem Typ das Objekt ist und somit, welche Implementierung der Methode ausgeführt wird
- **5. Vererbung („ist ein“-Relation -> siehe DB: isA):** Erbende Klasse (speziellere) erbt alle Attribute und Methoden von vererbender Klasse (allgemeinere) und kann neue hinzufügen bzw. Methoden überlagern

Vererbung (Beispiel)



- fördert Wiederverwendung
- beerbt existierende Klasse und fügt noch was dazu oder überlagert Methoden
- kann Vereinigung von Klassen unterstützen

größere Vererbungshierarchie



Vorteile von OOS:

- einfachere und übersichtlichere Wartung
- Nähe zum Anwendungsgebiet
- Wiederverwendbarkeit

Inhalt der Vorlesung

- Was ist Java & imperative Programmierung in Java
- Einführung in die OO-Softwareentwicklung
- allgemeine Konzepte der OO-Softwareentwicklung
- Ausnahmeverarbeitung in Java
- Persistenz durch Serialisierung und Deserialisierung
- Junit-Tests von OO-Programmen
- Oberflächenprogrammierung und Ereignisverarbeitung
- Client-Server-Integration
- kurze Einführung in Design Pattern

4 Konzepte der OOS

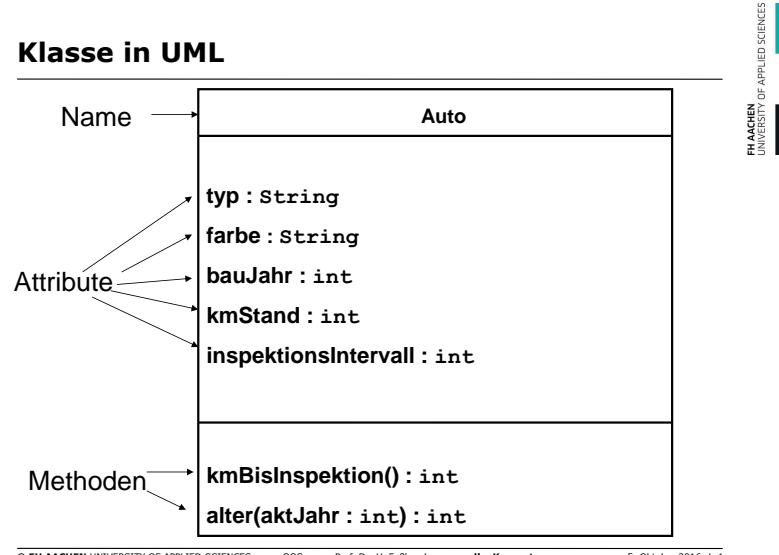
- nach der Einleitung, wird jetzt detailliert auf die Konzepte eingegangen (diese werden dann in den kommenden Kapiteln ergänzt und der Einsatz in Java & UML gezeigt)
- OO-Programme bestehen aus interagierenden Objekten
- **programmiert werden Klassen** (davon werden dann Objekte als Instanzen erzeugt)

4.1 Klassen

Def.: Beschreibung eines oder mehrerer ähnlicher Objekte (Abstraktion)

UML-Notation:

- 3 Rechtecke: Name, Attributdeklaration, Methodendeklaration



- man kann Attributnamen, Rückgabetyp, Parametername und -typ, ... weglassen oder hinschreiben

Java-Notation:

Klassendefinition durch:

- 1. Angabe des Namens
- 2. Attributdeklaration, ggf. mit Initialisierung
- 3. Methodenimplementierungen

Klasse in Java

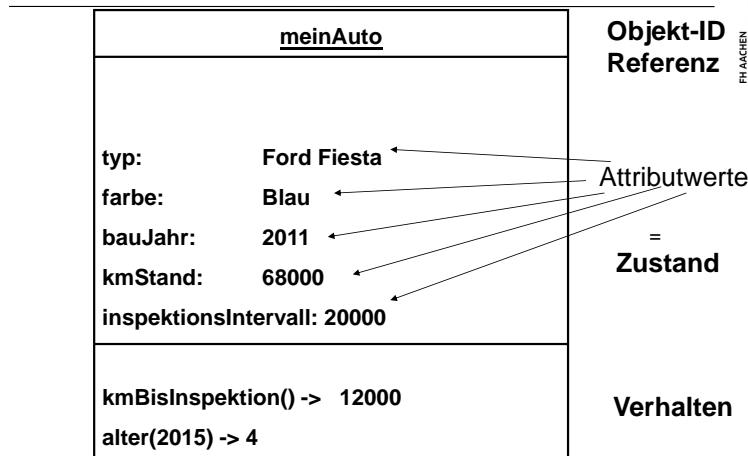
```
//Klassendefinition:  
class Auto {  
//Attributdeklarationen:  
    String typ;  
    String farbe;  
    int bauJahr;  
    int kmStand;  
    int inspektionsIntervall;  
//Methodenimplementierungen:  
    int kmBisInspektion() {  
        return (inspektionsIntervall - (kmStand %  
                                         inspektionsIntervall));  
    }  
    int alter(int aktJahr) {  
        return (aktJahr - bauJahr);  
    }  
}
```

4.2 Objekte

Def.: Ein Objekt ist ein tatsächlich existierendes „Ding“. Es ist eindeutig identifizierbar durch Objekt-ID (Referenz) und hat Zustand (Attributwerte) und Verhalten (Methoden).

UML-Notation:

Objekt in UML



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender allg. Konzepte 5. Oktober 2016 | 3

- meistens Objekte nicht in UML dargestellt, ansonsten wie Klassen, aber Name (Kleinbuchstabe vorne!) unterstrichen, Attribute haben Werte

Java-Notation:

Deklaration einer Referenz auf ein Objekt:

```
//Legt auf Stack Referenz an, die später auf ein Auto-Objekt zeigen wird:
```

```
Auto meinAuto;
```

meinAuto [] ->

```
//Erzeugt Objekt auf Heap:
```

```
meinAuto = new Auto();
```

meinAuto [] -> (Heap) [typ, farbe, bauJahr, kmStand, inspektionsInterval]

Alternativ:

```
Auto meinAuto = new Auto();
```

Vorsicht: Default-Werte der Attribute abhängig vom Compiler, am besten selber besetzen

Schreibender Zugriff auf Attribute durch Punktnotation:

```
meinAuto.typ = "DeLorean";
```

Lesender Zugriff auch über Punktnotation

4.3 Methoden

- haben direkten Zugriff auf Attribute der eigenen Klasse (man muss in der Regel innerhalb der Methoden keine Referenz vor die Attribute schreiben, Ausnahme s.u.)

Beispiel:

```
int mult(int zahl1, int zahl2) {

    // Rückgabetyp kann auch Klassenname (dann Referenz als Rückgabe)

    // oder Arrayname sein

    return (zahl1*zahl2);

}
```

- bei void als Rückgabetyp keine Rückgabe (nicht in Ausdrücken verwendbar), sonst return notwendig
- Aufruf durch Punktnotation (wie bei Attributen)

```
void umspritzen(String farbe) {

    this.farbe = farbe;

    // "this" hier notwendig, um Parameter vom Attribut zu unterscheiden

    // auch weglassen

}
```

- „this“ ist Referenz auf sich selbst
- muss immer dann verwendet werden, wenn Attribut durch Parameter/lokale Variable (innerhalb einer Methode) überlagert wird
- später: mit this Referenz auf sich selbst anderen Objekten mitteilen

Parameterübergabemechanismen:

Allgemein in Programmiersprachen:

- call-by-value
- call-by-reference

```
class ZeitVerwaltung {

    int nächstesJahr(int jahr) {

        return (++jahr);

    }

}

//in main:

int aktJahr = 2008;

ZeitVerwaltung zeitVer = new ZeitVerwaltung();

zeitVer.nächstesJahr(aktJahr);
```

Stack:

aktJahr: 2008

↓ Aufruf von nächstesJahr

jahr: 2008 -> kopieren, Methode hat keinen Zugriff nach aussen

aktJahr: 2008

↓ ++jahr

jahr: 2009

aktJahr: 2008

↓ nach Methodenbearbeitung

aktJahr: 2008

Bsp. für call-by-value

```
// neue Methode in Klasse Auto
void printFarbe(int wieoft) {
    while (wieoft-- > 0) {
        System.out.println("Farbe = " + farbe);
    }
}

// Bsp. für Aufrufe von meinAuto.farbe:
int a = 3;
meinAuto.printFarbe(a);      Wie oft wird die Farbe
meinAuto.printFarbe(a);      ausgegeben?
meinAuto.printFarbe(a);
```

call-by-value: Veränderungen der Parameter werden nach außen nicht sichtbar (s.o.: in aktJahr steht weiterhin 2008)

- a bleibt unverändert
- es wird 9 mal Farbe ausgegeben

Gegenstück: call-by-reference (nicht (direkt) in Java)

- a würde sich verändern
- es würde 3 mal Farbe ausgegeben werden
- call-by-reference geht in Java nicht direkt, aber für Objekte simulierbar, indem Referenzvariable als Parameter übergeben wird
- Unterschied zu C, wo auch für primitive Datentypen auf Speicher zugegriffen werden kann

Bsp. für call-by-reference

```
class Bsp {
    void ändereFarbe(Auto einAuto) {
        einAuto.farbe = "rot";
    }

    // Bsp. für Aufruf von ändereFarbe in main:
    Auto meinAuto = new Auto();
    meinAuto.farbe = "blau";           Welche Farbe wird
    Bsp bsp = new Bsp();              ausgegeben?
    bsp.ändereFarbe(meinAuto);
    System.out.println(meinAuto.farbe);
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender allg. Konzepte 5. Oktober 2016 | 5

- Ausgabe: rot
- hier Änderung außen (auf Heap) nachher noch wirksam
- Vorteil von Java-call-by-reference: nur 1 Referenz neu erzeugen und nicht für alle Attribute Platz schaffen
- Nachteil: Seiteneffekte (andere schreiben in meine Objekte)

einAuto ↴

meinAuto → typ, farbe="blau", ...

4.4 Konstruktoren

- Bei Erzeugung neuer Objekte:

```
Auto meinAuto = new Auto();

// Defaultkonstruktor (Auto()) so lange vorhanden,
// bis manuell anderer Konstruktor definiert
```

- Konstruktor stellt Speicherplatz auf Heap zur Verfügung und initialisiert Attribute defaultmäßig
- man kann auch eigenen Konstruktor implementieren

Def.: Konstruktoren sind Methoden ohne explizit aufgeführten Rückgabetyp (also auch kein void), die den Namen der Klasse erhalten, zu der sie gehören. Die Anzahl der Parameter ist beliebig.

- Rückgabewert des Konstruktors ist Referenz auf neu erzeugten Speicherplatz
- möglichst immer vor die anderen Methoden der Klasse schreiben

Beispiel:

```
Auto(String farbe) {  
  
    this.farbe = farbe;  
  
}
```

mehrere Konstruktoren in einer Klasse

```
Auto(String farbe) {  
    this.farbe = farbe;  
}  
  
Auto(String farbe, int bauJahr) {  
    this.farbe = farbe;  
    this.bauJahr = bauJahr;  
}
```

Überladung auch bei Methoden möglich

Wie sieht Konstruktor aus, der alle Attribute initialisiert?

- Überladung: mehrere Implementierungen einer Methode mit gleichem Namen mit unterschiedlichen Signaturen in gleicher Klasse
- Vorsicht: Default-Konstruktor geht verloren! Muss ggf. neu implementiert werden.

Verkettung von Konstruktoren

```
public Auto(String farbe) {
    this.farbe = farbe;
}

public Auto(String farbe, int bauJahr) {
    this(farbe);
    this.bauJahr = bauJahr;
}
```

- `this(farbe)` ruft anderen Konstruktor auf
- unterschiedliche Konstruktoren können sich gegenseitig aufrufen, Aufruf muss oben als 1. Statement stehen, weil dadurch Speicherplatz zur Verfügung gestellt wird.
- maximal ein einziger weiterer Konstruktor von einem Konstruktor aus aufrufbar

4.5 Vererbung

- unterstützt die Wiederverwendung

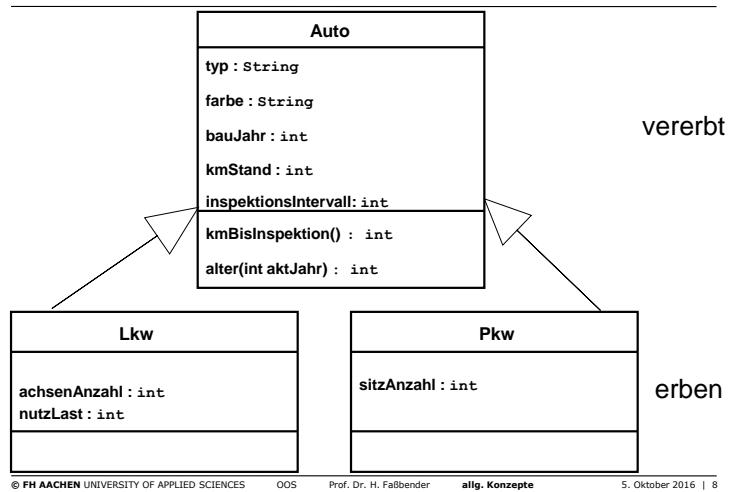
Def.: Die Möglichkeit eine Klasse B so zu definieren, dass sie ohne explizite Deklaration alle Attribute und Methoden einer bereits definierten Klasse übernimmt (erbt), heißt Vererbung.

- A ist die Oberklasse (Superklasse), ist allgemeiner
- B ist die Unterklasse (Subklasse), ist spezieller
- Relation zwischen A und B: `ist_ein`-Relation
- die Unterklasse kann neue Attribute und Methoden hinzufügen oder vererbte Methoden überlagern bzw. überladen

UML-Darstellung:

- Durchgezogener, geschlossener Pfeil von Unterklasse zur Oberklasse

Vererbung in UML

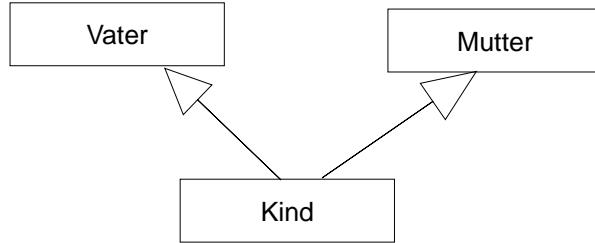


- Alles, was in Unterklassen zu Arbeit führt, dort mit aufführen
- neue Attribute & Methoden
- überlagerte Methoden

Java-Darstellung:

```
class ErbendeKlasse extends VererbendeKlasse {
    ...
}
```

Mehrfachvererbung



Gibt es in Java nicht!!!
Simulation durch Interfaces -> später

Vererbung in Java

```

class Lkw extends Auto {
    int achsenAnzahl;
    int nutzLast;
}

class Pkw extends Auto {
    int sitzAnzahl;
}

//in main:
Lkw lkw = new Lkw();
lkw.achsenAnzahl = 3;
//aber auch Attribute von Auto verwendbar:
lkw.kmStand = 10000;
lkw.inspektionsIntervall = 15000;
//auch Methoden von Auto verwendbar:
System.out.println(lkw.kmBisInspektion());
  
```

- Vererbung beliebig tief schachtelbar
- erbt alles von unmittelbarer Oberklasse
- man kann Vererbung verbieten: *final* vor *class* (falls Klasse noch nicht zuende programmiert, Klasse kann mit *final* niemals Oberklasse einer anderen Klasse sein)

4.6 Zuweisungskompatibilität

Man kann einer Objektreferenz auf ein Oberklassenobjekt eine Referenz auf ein Objekt einer Unterklasse zuweisen.

```

Auto auto = new Auto();

LKW lkw = new LKW();

auto = lkw; // kann nur auf Auto-Attribute / -Methoden zugreifen

// geht, da lkw mindestens die Auto-Attribute hat

lkw = auto; // geht nicht, da lkw mehr Attribute / Methoden

// als auto hat, lkw.nutzLast z.B. würde dann nicht existieren

```

- Da jeder LKW ein Auto ist, sind in LKWs auch alle Dinge der Autos enthalten ⇒ kein Problem
- Umgedreht geht nicht, später aber mit *cast*

4.7 Zuweisungskompatibilität auch in Parametern

Zuweisungskompatibilität in Param.

```

public class AutoFarben {
    String farbe (Auto auto) {
        return (auto.farbe);
    }
}

public static void main(String[] args) {
    LKW lkw = new LKW();
    lkw.farbe = "grau";
    AutoFarben autoFarben = new AutoFarben();
    System.out.println(autoFarben.farbe(lkw));
}
}

```

Vergleiche

- auch als Parameter von Konstruktoren / Methoden kann man anstelle einer Referenz auf ein Oberklassenobjekt eine Referenz auf ein Unterklassenobjekt übergeben
- Methode verwendet nur Dinge des Oberklassenobjekts (in Unterklassenobjekt auch vorhanden)

4.8 Überlagerung von geerbten Methoden in Unterklassen

Überlagern von Methoden (Polymorph.)

```
//Klassendefinition:
class Lkw extends Auto {
    //neue Attribute:
    int achsenAnzahl;
    int nutzLast;
    //Methodenüberlagerung alter jetzt in Monaten:
    int alter(int aktJahr) {
        return (12*(aktJahr - baujahr));
    }
}

// in main:
Lkw lkw = new Lkw();
lkw.baujahr = 1999
System.out.println(lkw.alter(2003)); // -> 48
```

```
Auto auto = new Auto();

LKW lkw = new LKW();

auto = lkw; // kann nur auf Auto-Attribute / -Methoden zugreifen

System.out.println(auto.alter(2003)); // falls Methode aus Auto

// allerdings überlagert wurde, wird die Unterklassenimplementierung

// ausgeführt
```

Da `auto` auf ein Objekt der Klasse LKW zeigt, wird Methodenimplementierung aus LKW ausgeführt.

4.9 Dynamisches Binden

Bei überlagerten Methoden kann erst zur Laufzeit (in Abhängigkeit des Objekttyps) entschieden werden, welche Implementierung ausgeführt wird.

Möchte man die Methodenimplementierung der Oberklasse verwenden, so kann man dies mit `super.methodName(..)` innerhalb der abgeleiteten Klasse ausführen.

Wichtig: Man muss mindestens die gleichen Zugriffsrechte in der UnterkLASSE, wie in der OberKLASSE geben.

Ohne dynamisches Binden wäre Polymorphismus (Überlagerung) nicht möglich.

4.10 Klasse Object

- alle Klassen sind automatisch von *Object* abgeleitet
- stellt nützliche Methoden zur Verfügung, die in erbenden Klassen überlagert werden müssen:

boolean equals(Object obj)

- in *Object* als Identität implementiert (Referenzen müssen gleich sein, sprich das selbe Objekt wird referenziert)
- in Überlagerungen: Test, ob Parameterobjekt und aktuelles Objekt gleiche Attributwerte haben

Hier: im ersten Ansatz als Überladung (gleicher Name, unterschiedliche Signaturen, gleiche Klasse) implementiert

Später: immer als Überlagerung (gleicher Name, gleiche Signaturen, unterschiedliche Klassen)

Überladung: *boolean equals(Auto auto)* (andere Signatur als *boolean equals(Object obj)*, daher Überladung)

Methode equals für Klasse Auto

```
boolean equals(Auto auto) {
    return (this.typ.equals(auto.typ) &&
            this.farbe.equals(auto.farbe) &&
            this.baujahr == auto.baujahr &&
            this.kmStand == auto.kmStand &&
            this.inspektionsIntervall ==
                auto.inspektionsIntervall);
}
```



Was würde passieren, wenn man für typ und farbe auch == verwenden würde?

Referenzen würden überprüft und nicht Inhalte

-> bei Objekten immer equals benutzen und nicht ==

protected Object clone()

- kopiert ein Objekt, nicht nur Referenz darauf
- es wird rekursiv geklont

```
Auto meinAuto = new Auto();

Auto deinAuto = (Auto) meinAuto.clone();

// Typecast notwendig, weil meinAuto.clone() Object-Referenz

// als Rückgabewert hat
```

String toString()

- erzeugt String-Repräsentation des aktuellen Objekts
- wird von Ausgabebefehl print und println aufgerufen
- in Object durch kryptische Objektreferenzausgabe vorimplementiert (sollte also überlagert werden)

Methode **toString** für Klasse Auto

```
public String toString() {
    return ( " typ: " + this.typ.toString() +
            " farbe: " + this.farbe.toString() +
            " bauJahr: " + this.bauJahr +
            " kmStand: " + this.kmStand +
            " inspektionsIntervall: " +
            this.inspektionsIntervall);
}

//in main:
Auto auto1 = new Auto("Fiesta","blau",2011,68000,20000);
System.out.println(auto1);
```



```
typ: Fiesta farbe: blau baujahr: 2011 kmStand: 68000
inspektionsIntervall: 20000
```

⇒ anderen Methoden von *Object* momentan uninteressant

4.11 Konstruktoren in Vererbungshierarchien

Bei Konstruktoraufruf einer erbenden Klasse, wird **implizit** zuerst der Defaultkonstruktor der Oberklasse mit *super()* aufgerufen.

Ausführreihefolge von Konstruktoren

definiere in Klasse Auto Konstruktor wie folgt:

```
Auto() {
    System.out.println("Konstruktor von Auto!");
}
```

definiere in Klasse Lkw Konstruktor wie folgt:

```
Lkw() {
    System.out.println("Konstruktor von Lkw!");
}
```

Ausgabe bei Aufruf von new Lkw() ist:

Konstruktor von Auto!
Konstruktor von Lkw!

Es wird die gesamte Vererbungshierarchie beginnend bei der Wurzel durchlaufen

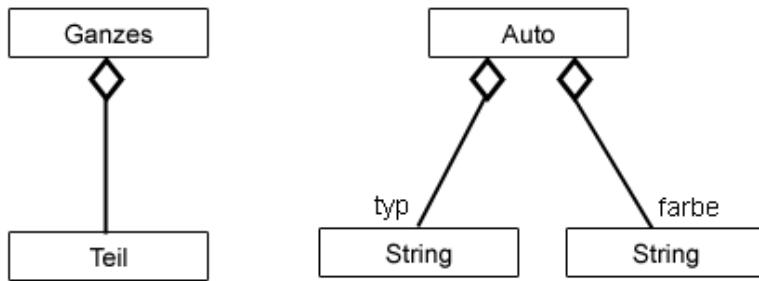
- man kann mit *super(..)* alternativ andere Oberklassenkonstruktoren aufrufen
- oder mit *this(..)* andere Konstruktoren der eigenen Klasse aufrufen
- Konstruktoren werden nicht vererbt!

4.12 Aggregation

- bisher: Vererbung (*ist_ein*) als einzige Relation zwischen Objekten
- jetzt: Aggregation (*hat_ein*, *besteht_aus*)

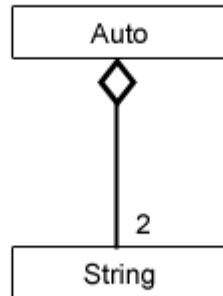
Bei *Auto* zwei Attribute vom Klassentyp *String*

UML-Darstellung:



Kardinalitäten:

- ohne = 1
- * = keins, eins oder beliebig viele
- 1..5 = Bereich



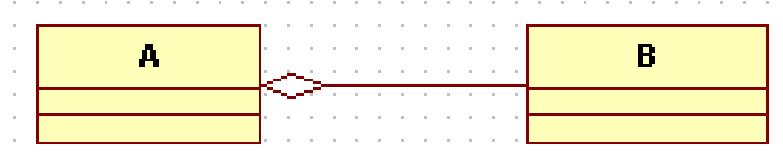
Java-Notation:

- bei Attribut Reinschreiben von Klasse, anstelle des primitiven Typs: `String typ;`, `String farbe;`
- Aggregation der Vererbung vorziehen, da in Java keine Mehrfachvererbung möglich und Vererbung oft auch bei nicht-`ist_ein`-Relationen eingesetzt

4.13 Konstruktoren bei Aggregation

- Unterschied zur Vererbung, wo Konstruktor der Oberklasse implizit aufgerufen wird!

Definiert man Klasse A durch Aggregation mit Klasse B, so muss ggf. im Konstruktor von A der von B explizit aufgerufen werden, um Speicherplatz für das Attribut zur Verfügung zu stellen.



```

class A {
    B b; // nur Referenz

    A(int i) {
        this.b = new B(i); // hier wird Objekt erst erzeugt
    }
}

class B {
    int i;

    B(int i) {
        this.i = i;
    }
}

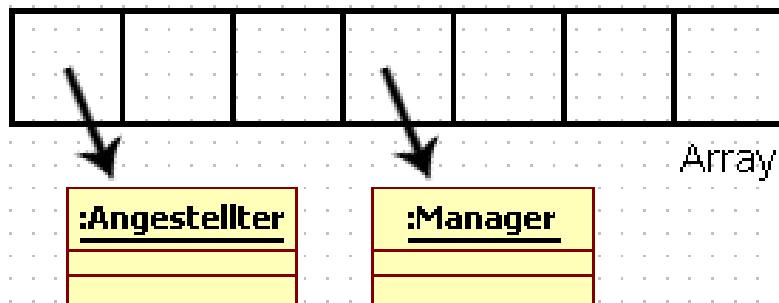
```

- Alternativ: `B b = new B();` ⇒ direkt als Attribut möglich
- Durch `B b;` wird nur Referenz erzeugt und nicht das aggregierte Objekt
- Zugriff über mehrfache Punktnotation: `a.b.i`
- **Sonderfall String:** Kein Konstruktoraufruf notwendig: `String s = "Heinz";`

4.14 Abstrakte Klassen, Polymorphismus und dynamisches Binden

Gegeben ist ein Array mit Angestellten und Managern. Gesucht ist die Summe der Bruttolöhne, dabei ist die Methode `double monatsBrutto()` bei Angestellter und Manager unterschiedlich implementiert.

Das Array ist vom Typ der Superklasse (Mitarbeiter).



Schreiben drei Klassen: Angestellter, Manager und Mitarbeiter (Superklasse)

=> Nehmen in Mitarbeiter alles rein, was die Unterklassen gemeinsam haben

Klasse Angestellter in Java

```
import java.util.*;  
  
class Angestellter {  
    int persNr;  
    String name;  
    Date eintritt;           //aus java.util.*  
    double grundGehalt;  
    double ortsZuschlag;  
    double zulage;  
  
    double monatsBrutto() {  
        return (grundGehalt +  
                ortsZuschlag +  
                zulage);  
    }  
}
```

Klasse Manager in Java

```
import java.util.*;

class Manager {
    int persNr;
    String name;
    Date eintritt; //aus java.util.*
    double fixGehalt;
    double provision1;
    double provision2;
    double umsatz1;
    double umsatz2;

    double monatsBrutto() {
        return (fixGehalt +
            umsatz1*provision1/100 +
            umsatz2*provision2/100);
    }
}
```

abgeleitete Klasse Angestellter in Java

```
class Angestellter extends Mitarbeiter {
    int persNr; Hier weglassen, da schon in
    String name; Klasse Mitarbeiter deklariert
    Date eintritt;

    double grundGehalt;
    double ortsZuschlag;
    double zulage;

    double monatsBrutto() { Analog für die Klasse
        return (grundGehalt +
            ortsZuschlag +
            zulage); Mitarbeiter
    }
}
```

abstrakte Klasse Mitarbeiter in Java

```
abstract class Mitarbeiter {

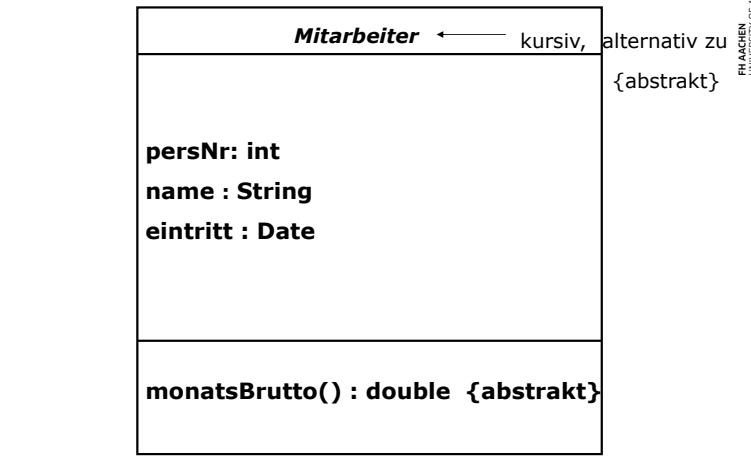
    //gemeinsame Attribute der drei Unterklassen:
    int persNr;
    String name;
    Date eintritt;

    //abstrakte Methode:                 keine Implementierung
    //                                der Methode
    abstract double monatsBrutto();
}
```

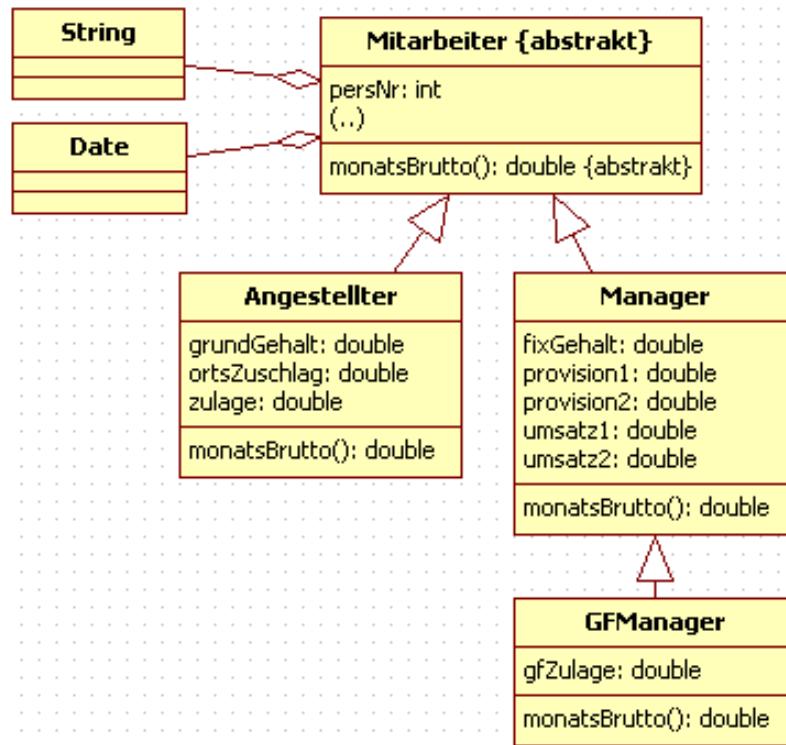
Def.: Abstrakte Methode: Wenn eine Methode nicht implementiert ist, so heißt diese abstrakt

Def.: Abstrakte Klasse: Eine Klasse, die mindestens eine als abstrakt definierte Methode enthält, muss selbst als abstrakt definiert werden

abstrakte Klasse Mitarbeiter in UML



=> Möchten jetzt nur noch spezielle Manager *GManager* hinzufügen!



tiefere Ableitung

```
// Manager der Geschäftsführung:
class GFManager extends Manager {
```

```

    double gfzulage;

    double monatsBrutto() {
        return (super.monatsBrutto() +
            gfzulage);
    }
}
```

Methode der Oberklasse **Manager**

Berechnung des Gesamtbruttos

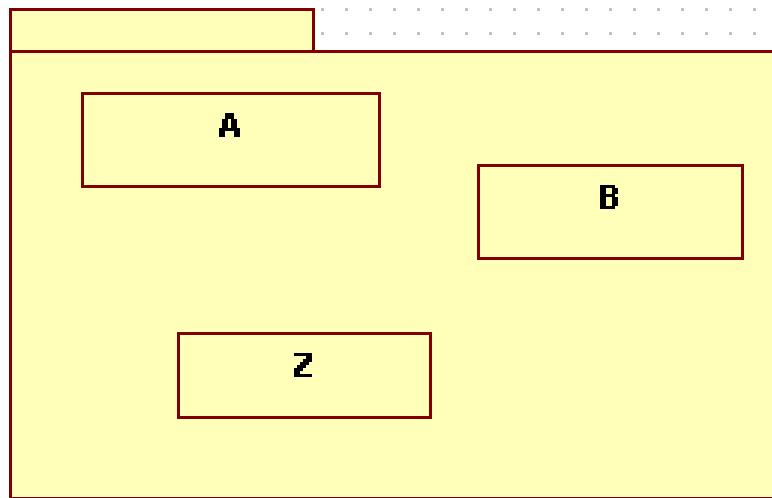
```
class GehaltsBerechnung {
    public static void main(String[] args) {
        Mitarbeiter[] ma = new Mitarbeiter[100];
        //Mitarbeiter-Array füllen, z.B.
        //ma[0] = new Manager();
        //ma[1] = new Angestellter();
        //ma[2] = new GFManager();
        ...
        //Bruttosumme berechnen
        double bruttosumme = 0.0;
        for (int i=0; i<100; ++i) {
            bruttosumme += ma[i].monatsBrutto();
        }
        System.out.println("Bruttosumme = "+bruttosumme);
    }
}
```

Hier wird jeweils die Implementierung der entsprechenden Unterklasse aufgerufen!

4.15 Pakete

- bisher: nur Klassen zur Strukturierung
- jetzt: fasst mehrere Klassen zu Bibliotheken zusammen, in Java und UML hierzu: Pakete
- zur Übersichtlichkeit und um Namensräume zu definieren

In UML:



In Java:

- keine einzelnen Dateien, sondern Verzeichnisse, in denen die Klassen, die dazu gehören, angelegt werden
- muss innerhalb der Klassen-Dateien den Befehl `package paketname;` an den Anfang schreiben
- wenn nichts angegeben, dann in Default-Paket

Zugriff:

Angenommen eine Klasse *Klasse* liegt in einem Verzeichnis mit folgender Struktur:

Pfad/observer/paketname

wobei die CLASSPATH-Umgebungsvariable von Java

Pfad enthält, dann kann man auf *Klasse* wie folgt zugreifen:

observer.paketname.Klasse

Man kann auch das ganze Paket importieren: `import observer.paketname.*;`

Dann braucht man nur noch *Klasse* angeben (anstatt *observer.paketname.Klasse*)

Beispiel:

```
package meinPaket;

public class MeineKlasse {

    (...)

}
```

Aus anderer Datei Zugriff durch:

```
import meinPaket.*;

(...)

MeineKlasse m = new MeineKlasse();
```

Die Datei muss im gleichen Verzeichnis wie *meinPaket* liegen, oder im CLASSPATH stehen.

- Vermeidung von Namenskollisionen
- spannt Namensräume auf

4.16 Kollisionen

Kollisionen können trotzdem auftreten, z.B.:

In pak1 und pak2 jeweils Klasse A definiert

```
import pak1.*;  
  
import pak2.*;  
  
A a = new A(); //Kollision!
```

Lösung durch Qualifizierung: *pak1.A a = new pak1.A();*

Sonderbehandlung: Default-Paket:

- in Klassen-Datei wird nicht explizit *package* aufgeführt, dann ist es in Default-Package
- wird bei Kollisionen bevorzugt behandelt

4.17 Zugriffsspezifikationen

- public: Zugriff auch aus anderen Paketen, Zugriff von allen Klassen aus
- protected: bei Zugriff von anderen Objekten aus absolut wie friendly, ist jedoch zusätzlich in Unterklassen anderer Pakete vorhanden
- ohne (friendly): Zugriff paketweit zulässig (bisher)
- private: Zugriff nur innerhalb der Klasse

Zugriffspezifikationen

Java UML	Sichtbarkeit bei anderer zugreifender Klasse	in Unterklasse vorhanden, falls diese
public +	Überall	beliebige Position hat
protected #	Innerhalb des Pakets	beliebige Position hat
~ "friendly"	Innerhalb des Pakets	innerhalb des Pakets liegt
private -	Nein	Nein

echte Hierarchie & innerhalb Klasse alles aus Klasse sichtbar

Innerhalb der Klasse ist immer alles aus der eigenen Klasse sichtbar. In Programmen immer alles gemäß obiger Reihenfolge sortieren (erst public, dann protected, ..) und innerhalb einer Gruppe:

- Attribute, dann
- Konstruktoren, dann
- Methoden

4.18 Klassenzugriff

Nur Unterscheidung zwischen:

- friendly: im Paket sichtbar (bisher)
- public: überall sichtbar

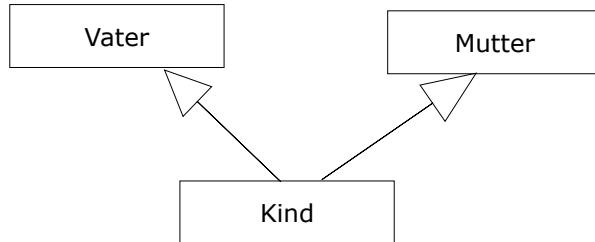
Man kann mehrere Klassen in eine Datei schreiben, aber:

- höchstens eine public-Klasse pro Datei
- diese muss gleichen Namen wie Datei haben
- kann auch Dateien ohne public-Klasse geben
- Datei = Paket

Private-Verhalten dadurch simulierbar, dass alle Konstruktoren private sind (kann nur selbst Objekte meiner Klasse erzeugen).

4.19 Interfaces (Schnittstellen)

Mehrfachvererbung



Gibt es in Java nicht!!!

Simulation durch Interfaces

Problem bei Mehrfachvererbung

```

class Vater {
    int geburtsJahr;
    int vorgegebenesAlter(int aktJahr) {
        return (aktJahr - geburtsJahr);
    }
}

class Mutter {
    int geburtsJahr;
    int vorgegebenesAlter(int aktJahr) {
        return (aktJahr - geburtsJahr - 5);
    }
}
  
```

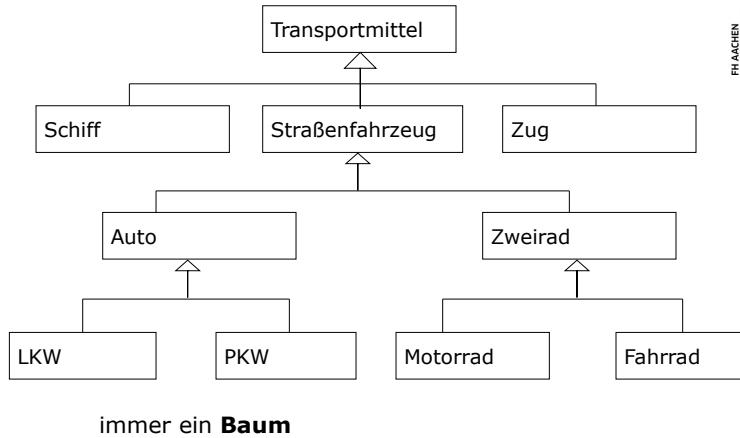
```

class Kind extends Vater,Mutter { // geht nicht in Java
}

Kind kind = new Kind();
kind.geburtsJahr = 1950;
  
```

```
kind.vorgegebenesAlter(2008); // 58 oder 53?  
// soll Methode von Mutter oder Vater genommen werden?
```

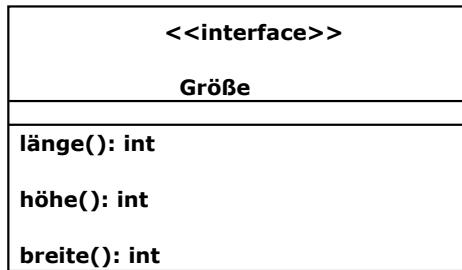
Vererbungshierarchien in Java



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender allg. Konzepte 5. Oktober 2016 | 26

=> Umgeht Problem, wenn vererbte Methoden in Oberklassen abstrakt sind

Interface in UML



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender allg. Konzepte 5. Oktober 2016 | 27

Def.: Eine öffentliche abstrakte Klasse, die ausschließlich abstrakte Methoden und Konstantendefinitionen enthält und anstelle von *class* durch *interface* klassifiziert ist, heißt Interface (Schnittstelle) in Java.

⇒ keine Konstruktoren erlaubt

Interface in Java

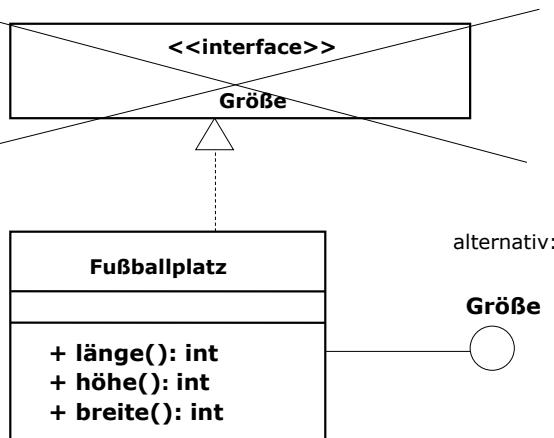
```
public interface Größe {
    public int längen(); ← Methodendeklarationen,
    public int höhe(); ← keine Implementierung
    public int breite();
}
```

Prinzip des Information Hidings:

Es wird nur bekannt gegeben, welche Methoden zur Verfügung gestellt werden, nicht wie sie implementiert werden.

Zum Bereitstellen von Funktionalität muss das **Interface durch eine Klasse implementiert** werden.

Implementierungsrelation



1. Implementierung von Größe

```
public class FußballPlatz implements Größe {
    public int länge() {
        return 105;
    }

    public int Höhe() {
        return 0; ← konstante Werte
    }

    public int breite() {
        return 70; ←
    }
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender allg. Konzepte 5. Oktober 2016 | 30

2. Implementierung von Größe

```
public class Schrank implements Größe {
    int länge;
    int Höhe;
    int breite;

    public int länge() {
        return this.länge; ←
    }
    public int Höhe() {
        return this.Höhe; ← Attributwerte
    }
    public int breite() {
        return this.breite; ←
    }
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender allg. Konzepte 5. Oktober 2016 | 31

Teilweise Implementierung von Größe

```
abstract class Rechteck implements Größe {
    int länge;
    // int Höhe;
    int breite;

    public int länge() {
        return this.länge;
    }
    // public int Höhe() {
    //     return this.Höhe;
    // }
    public int breite() {
        return this.breite;
    }
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender **allg. Konzepte** 5. Oktober 2016 | 32



Verwendung des Interface Größe

```
class InterfaceVerwendung {
    int grundflaeche(Größe g) {
        return g.länge() * g.breite();
    }
    public static void main(String[] args) {
        Schrank schrank = new Schrank();
        schrank.länge = 1;
        schrank.breite = 3;
        FussballPlatz platz = new FussballPlatz();
        InterfaceVerwendung iv = new InterfaceVerwendung();
        //Nun werden sie ausgegeben:
        System.out.println("Schrank: " +
            iv.grundflaeche(schrank));
        System.out.println("Platz: " +
            iv.grundflaeche(platz));
    }
}
```

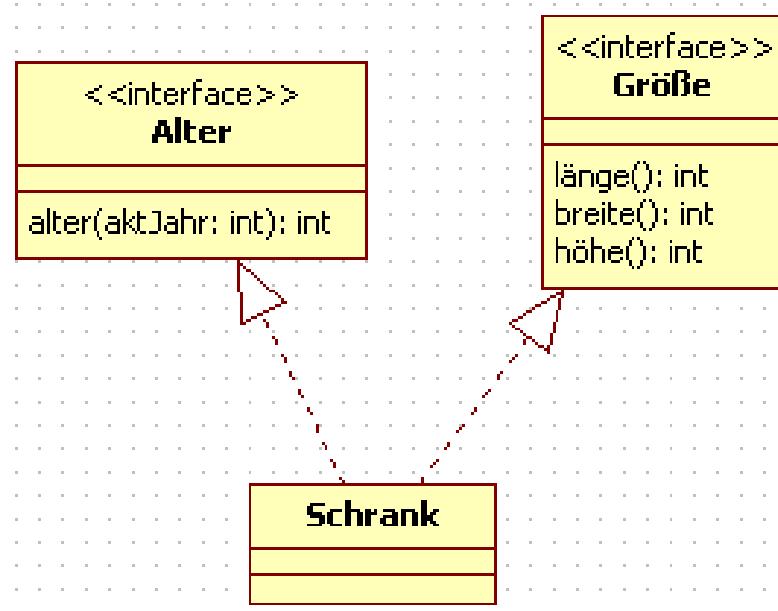
© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender **allg. Konzepte** 5. Oktober 2016 | 33



Zuweisungskompatibilität analog zur Vererbung: Anstelle des Interfaces werden implementierende Klassen in Parametern oder Zuweisungen eingesetzt.

Bisher: mehrere Implementierungen eines Interfaces

Jetzt: Implementierung von mehreren Interfaces



Implementierung von 2 Interfaces

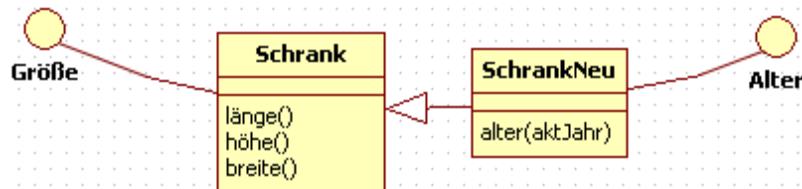
```

public class Schrank2 implements Größe, Alter {
    int längen();
    int höhe();
    int breite();           neu hinzunehmen:
    public int längen() {   int bauJahr;
        return this.längen();
    }
    public int höhe() {     public int alter(int aktJahr) {
        return this.höhe();   return (aktJahr - bauJahr);
    }
    public int breite() {
        return this.breite();
    }
}
  
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender allg. Konzepte 5. Oktober 2016 | 34

Man kann mehrere Interfaces implementieren, die hinter `implements` mit Komma getrennt aufgeführt werden!

Eine Klasse erbt die Interfaces ihrer Oberklasse:



Vererbung von Interf. bei Klassenvererb.

```
public class SchrankNeu extends Schrank
    implements Alter {

    int baujahr;

    public int alter(int aktJahr) {
        return (aktJahr - bauJahr);
    }
}
```

SchrankNeu liefert Implementierung der Methoden in Größe (geerbt von Schrank) und Alter

z.B.: *bsp(Größe g)* ⇒ es kann ein Schrank- oder ein SchrankNeu-Objekt übergeben werden, beide implementieren mindestens Größe

Vererben von Interfaces

```
interface EinDim {
    public int länge();
}

interface ZweiDim extends EinDim {
    public int breite();
}

interface DreiDim extends ZweiDim {
    public int höhe();
}
```

Implementierung von DreiDim

```
class Körper implements DreiDim {  
    public int länge() {  
        return 0;  
    }  
  
    public int breite() {  
        return 0;  
    }  
  
    public int Höhe() {  
        return 0;  
    }  
}
```

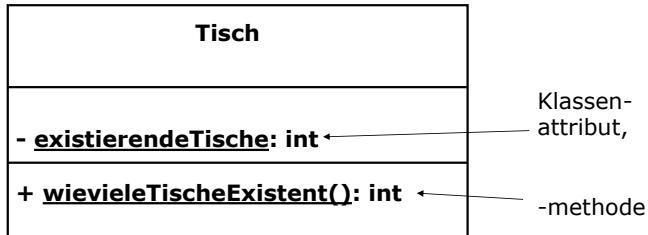
4.20 Klassenattribute und -methoden

- ähnlich zu globalen Variablen und Funktionen
- ⇒ Klassenattribute gezielt einsetzen
- Infos, die mehrere Objekte einer Klasse betreffen
- nicht an Objekte gebunden, sondern 1 mal pro Klasse vorhanden
- Klassenmethoden operieren auf Klassenattributen

UML-Notation:

jeweils unterstrichen

Klassenattribute und -methoden (UML)



Oft auch **statisch** genannt

Java-Notation:

- immer *static* vorschreiben
- Klassenattribute nur 1 mal pro Klasse vorhanden, Lebensdauer erstreckt sich auf gesamtes Programm
- Zugriff aus anderer Klasse durch *Klassenname.Klassenattribut* bzw. *Klassenname.Klassenmethode()*, z.B. *Tisch.wievieleTischeExistent()*;
- in Methoden von Klasse mit Klassenattribut: Zugriff auf Klassenattribut mit *Tisch.existierendeTische* oder *existierendeTische*, kein *this* verwenden
- *finalize()* nicht selbst aufrufen, sondern wird vom System durch Garbage-Collector aufgerufen => enthält Aufgaben, die bei Löschen des Objekts zu tun sind, muss überlagert werden

Klassenattribute und -methoden (Java)

```
public class Tisch {  
    static private int existierendeTische = 0;  
    public Tisch() {  
        ++existierendeTische;           enthält Anzahl der  
                                         Objekte von Tisch  
                                         durch Konstruktor  
                                         inkrementiert  
    }  
  
    public void finalize() {  
        --existierendeTische;          durch Garbage Collector  
                                         aufgerufen  
    }  
  
    static public int wievieleTischeExistent() {  
        return existierendeTische;      gibt Klassenattribut  
                                         aus  
    }  
}
```

4.21 Dokumentation mit Javadoc

Nutzerdoc:

- wie Programm verwendet wird
- Grundfunktionalität (was macht das Programm?)
- Voraussetzungen

Entwicklerdoc:

- Struktur (Designentscheidungen)
- Version, Vor- und Nachbedingung, ..

SDK enthält Tool zur automatischen Erzeugung einer Online-Entwicklerdoku: *javadoc*

Dokumentationskommentar

Für Compiler gleich Blockkommentar
Findet nur bei javadoc Berücksichtigung



```
/**  
 * Socken sind spezielle Kleidungsstücke.  
 */  
  
public class Socke extends Kleidung {  
}
```

Beschrieben werden:

- die Klassen (auch Innere)
- die Vererbung
- die Methoden
- die Attribute
- die Interfaces
- die Konstruktoren
- ⇒ defaultmäßig nur Erzeugung für public-Sachen, ggf. umstellen

wichtige Dokumentationskommentare

Kommentar	Beschreibung
@see Klassenname	Verweis auf eine andere Klasse
@see Methodenname	Verweis auf eine andere Methode
@version Versionstext	Version
@author Autor	Autor
@return Rückgabetext	Rückgabewert
@param Parametername oder Parametertext	Beschreibung der Parameter
@exception Exception- Klassenname oder Exceptiontext	Ausnahmen, die ausgelöst werden können
@throws Exception-Klassenname oder Exceptiontext	Synonym zu oben
{ @link Verweis }	Eingebauter Verweis

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender **allg. Konzepte** 5. Oktober 2016 | 41

z.B. bei Methode `int add(int sum1, int sum2):`

```
/**  
  
 * @param sum1: 1. Summand  
  
 * @param sum2: 2. Summand  
  
 * @return sum1 + sum2  
  
 */
```

Man kann beliebigen HTML-Text außen drum schreiben!

5 Ausnahmebehandlung (Exceptions)

- Mechanismus zur strukturierten Behandlung von Laufzeitfehlern (nicht zur Compilezeit (Syntaxfehler))
- zum Beispiel:
 - `ArrayIndexOutOfBoundsException` ⇒ Indexüberlauf bei Feld
 - `NullPointerException` ⇒ kostet Bier!

Exceptions können:

- geworfen werden: *throw* (selbst werfen, aktiv)
- gefangen (behandelt) werden: *catch* (jemand anders hat Exception geworfen)
- weitergereicht an Aufrufer des Programmteils: *throws* (man selbst hat Exception geworfen)

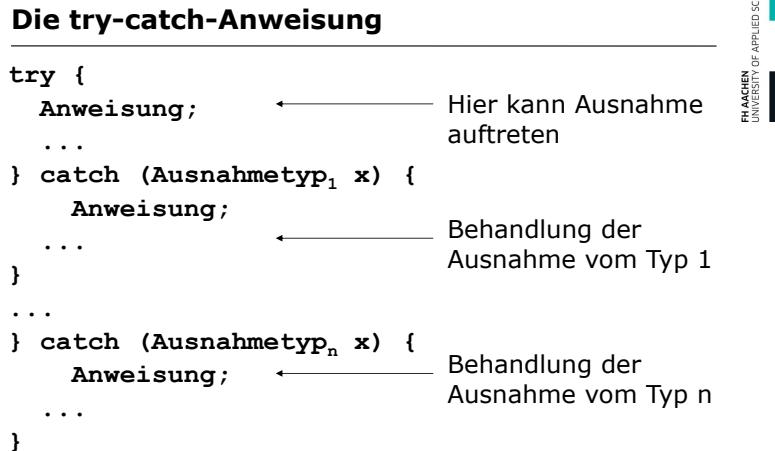
5.1 Grundprinzip des Exception-Handlings

1. Laufzeitfehler löst Exception aus
2. im aktuellen Programmteil a) fangen oder b) weiterleiten an Aufrufer
 - a) Ende
 - b) beim Aufrufer a) fangen oder b) weiterleiten
 - a) Ende
 - b) wenn im Main-Programm angelangt a) fangen oder b) weiterleiten
 - * a) Ende
 - * b) Abbruch durch System

catch-or-throws-Regel: wenn Ausnahme auftreten kann, dann fangen oder weiterleiten

5.2 Fangen (Behandlung) von Ausnahmen (catch)

Fangen per try-catch-Anweisung:



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Exceptions 5. Oktober 2016 | 1

try-Block enthält Anweisungen, in denen Exceptions der folgenden Typen ausgelöst werden können:

Ausnahmetyp 1 bis Ausnahmetyp n

- bei Auslösen einer Ausnahme wird Programmausführung im try-Block unterbrochen und Programmablauf fährt mit 1. Anweisung innerhalb des obersten catch-Blocks fort, dessen Ausnahmetyp passend zur Ausnahme ist
- dort Code unterbringen, der auf Ausnahme reagiert
- nach Ausführung des catch-Blocks, wird hinter den untersten catch-Block gesprungen (anders als bei switch)

Bsp.: Ausnahmebehandlung

```
public class AusnahmeBsp {

    public static void main(String[] args) {
        int i;

        // Basis der Darstellung von 40:
        int base = 0;

        // Dezimalwert von 40 zu Basen 10,...,2:
        for (base = 10; base >= 2; --base) {
            i = Integer.parseInt("40",base);
            System.out.println("40 base "+base+" = "+i);
        }
    }
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Exceptions 5. Oktober 2016 | 2

parseInt(String s, int n) liefert Dezimalwert von s im n-er System

JDK-Ausgabe ohne Ausnahmebehandlung.

```
java.lang.NumberFormatException: 40
    at java.lang.Integer.parseInt(Integer.java:414)
    at übungzuv5.AusnahmeBsp.main(AusnahmeBsp.java:22)

40 base 10 = 40
40 base 9 = 36
40 base 8 = 32
40 base 7 = 28
40 base 6 = 24
40 base 5 = 20
Exception in thread "main"
```

Problem? 40 ist keine gültige Zahl zur Basis 4 und kleiner

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Exceptions 5. Oktober 2016 | 3

⇒ Programm stürzt ab

Abfangen der Ausnahme

```
try {
    for (base = 10; base >= 2; --base) {
        i = Integer.parseInt("40",base);
        System.out.println("40 base "+base+" = "+i);
    }
} catch (NumberFormatException e) {
    System.out.println (
        "40 ist keine Zahl zur Basis "+ base);
}
```

↑ ↑
Exceptiontyp Fehlerobjekt
 wird von Exception-
 Auslöser übergeben

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Exceptions 5. Oktober 2016 | 4

JDK-Ausgabe mit Ausnahmebehandlung

```
40 base 10 = 40
40 base 9 = 36
40 base 8 = 32
40 base 7 = 28
40 base 6 = 24
40 base 5 = 20
40 ist keine Zahl zur Basis 4
```

↑
nicht mehr Exceptionmeldung,
sondern Ausgabe der Ausnahmebehandlung

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Exceptions 5. Oktober 2016 | 5

Fortfahren nach Ausnahmebehandlung

```
for (base = 10; base >= 2; --base) {
    try {
        i = Integer.parseInt("40",base);
        System.out.println("40 base "+base+" = "+i);

    } catch (NumberFormatException e) {
        System.out.println (
            "40 ist keine Zahl zur Basis "+ base);
    }
}
```

⇒ jetzt try innerhalb der for-Schleife

Ausgabe des JDK bei Fortfahren

```
40 base 10 = 40
40 base 9 = 36
40 base 8 = 32
40 base 7 = 28
40 base 6 = 24
40 base 5 = 20
40 ist keine Zahl zur Basis 4
40 ist keine Zahl zur Basis 3
40 ist keine Zahl zur Basis 2
```

⇒ ob Weitermachen oder Abbruch hängt von Fehler ab

5.3 Mehrere catch-Klauseln

- wenn verschiedene Ausnahmetypen auftreten können

- es werden von catch-Klausel alle zuweisungskompatiblen Fehler (also der aktuellen Klasse und aller Unterklassen) behandelt
- catch-Klauseln in Reihenfolge abgearbeitet, d.h. immer nur erste Zutreffende

verschiedene Ausnahmen

```
String[] numbers = new String[3];
numbers[0] = "10";
numbers[1] = "20";
numbers[2] = "30";
try {
    for (int base = 10; base >= 2; --base) {
        for (int j = 0; j <= 3; ++j) {
            int i = Integer.parseInt(numbers[j],base);
            System.out.println(numbers[j]+"base "+base+" = "+i);
        }
    }
} catch (IndexOutOfBoundsException e1) { Feldüberlauf
    System.out.println("****IOOBEx: " );
} catch (NumberFormatException e2) {
    System.out.println("****NFEEx: " );
}
```

mehrere Ausnahmen (JDK-Ausgabe)

```
10 base 10 = 10
20 base 10 = 20
30 base 10 = 30
****IOOBEx:
```

- fangen aller Exceptions mit einer catch-Klausel durch Exception-Typ *Exception* (Oberklasse aller Exceptions)
- oben catches für die Exceptions, die speziell behandelt werden sollen, gefolgt von catch mit *Exception* zur Behandlung des Rests

5.4 Finally-Klausel

- optionaler Bestandteil der try-catch-Anweisung (z.B. Schließen von Dateien oder Ressourcen freigeben)
- enthaltener Code wird immer (am Schluß) ausgeführt, falls try-Block betreten wurde, egal ob Exception ausgelöst oder nicht oder ob mit break, continue, return verlassen

Verwendung der finally-Klausel

```
try {
    for (base = 10; base >= 2; --base) {
        i = Integer.parseInt("40",base);
        System.out.println("40 base "+base+" = "+i);
    }
} catch (NumberFormatException e) {
    System.out.println (
        "40 ist keine Zahl zur Basis "+ base);
} finally {
    System.out.println("Schluss!");
}
```

JDK-Ausgabe mit finally-Klausel

```
40 base 10 = 40
40 base 9 = 36
40 base 8 = 32
40 base 7 = 28
40 base 6 = 24
40 base 5 = 20
40 ist keine Zahl zur Basis 4
Schluss!
```

5.5 Weitergabe von Ausnahmen (throws)

Grundregel: jede Ausnahme entweder behandeln oder weitergeben (catch-or-throws)

Weitergabe einer Ausnahme (throws)

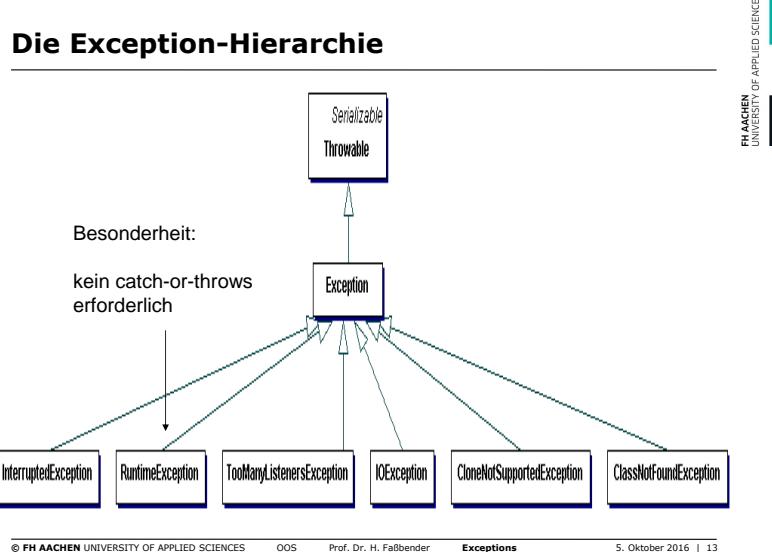
```
public void reziTable() throws ArithmeticException
{
    int x = 0;
    while (x <= 10) {
        System.out.println("rezi("+x+")="+ (double) (1/x));
        x++;
    }
}
```

hier kann Ausnahme auftreten
nicht behandelt,
darum weitergeben mit throws

Genereller Umgang mit Exceptions:

- Fangen: fremd geworfene Exceptions
- Weitergeben: eigene geworfene Exceptions

5.6 Exception-Hierarchie



Jede Ausnahme ist von `java.lang.Exception` abgeleitet.

`RuntimeException`: oft schwerer Programmierfehler

- `Throwable` hat neben `Exception` noch `Error` als direkte Unterklasse (Systemfehler ⇒ nicht drum kümmern)
- Besonderheit: `RuntimeException` (Division durch 0, IndexOutOfBoundsException) müssen nicht (können aber) gefangen oder weitergeleitet werden ⇒ in der Regel Denkfehler des Programmierers, sollte deshalb nicht unbedingt vom Programm abgefangen bzw. weitergeleitet werden

5.7 Auslösen (Werfen) von Ausnahmen

- Ausnahmeobjekte sind Instanzen bestimmter Exception-Klassen
- erzeugt man, wenn man selbst Exception wirft
- Behandlung wie andere Objekte
- z.B. Erzeugung: `new ArithmeticException()` (erstellt neues Fehlerobjekt)
- Werfen einer Ausnahme mit Hilfe von `throw`
- Syntax: `throw new AusnahmeKlasse()`

Auslösen einer Ausnahme mit throw

```
public class Auslösen {
    public void nurPositiv(int zahl)
        throws ArithmeticException {
        if (zahl <= 0) {
            throw new ArithmeticException(
                "Parameter nicht positiv!");
        }
    }
    public static void main(String[] args) {
        Auslösen auslösen = new Auslösen();
        auslösen.nurPositiv(0);
    }
}
java.lang.ArithmeticException: Parameter nicht positiv!
    at übungzuv5.Auslösen.nurPositiv(Auslösen.java:6)
    at übungzuv5.Auslösen.main(Auslösen.java:11)
Exception in thread "main"
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Exceptions 5. Oktober 2016 | 14

- man kann eigene Exception-Klassen definieren

Eigene Exception-Klasse verwenden

```
public class EE {
    public void nurPositiv(int zahl)
        throws NegativerParameterException {
        if (zahl <= 0) {
            throw new NegativerParameterException(
                "Parameter nicht positiv!");
        }
    }
    public static void main(String[] args) {
        EE ee = new EE();
        ee.nurPositiv(0);
    }
}
Exception in thread "main" NegativerParameterException:
Parameter nicht positiv!
at EE.nurPositiv(EE.java:11)
at EE.main(EE.java:16)
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Exceptions 5. Oktober 2016 | 15

Eigene Exception-Klasse definieren

```
public class NegativerParameterException extends  
ArithmeticException {  
    NegativerParameterException(String ausgabe) {  
        super(ausgabe);  
    }  
}
```

Konstruktor implementieren
durch Aufruf des Oberklassenkonstruktors

```
public static void main(String[] args) {  
    EE ee = new EE();  
    try {  
        ee.nurPositiv(0);  
    } catch (NegativerParameterException e) {  
        e.printStackTrace();  
    }  
}
```

- eigene Exception-Klassen meist in Datei, wo die Klasse drin ist, bei der der Fehler auftreten kann
- `e.printStackTrace();` kann man nachher ersetzen, erzeugt hier die Standardfehlerausgabe
- sollte man möglichst verwenden, da damit Position des Fehlers ersichtlich

6 Java 5.0

Erweiterungen in Java 5.0

- Übergang von SDK 1.4 -> 1.5 gewaltig, darum: **Java 5.0**
- wird auch **Tiger-Release** genannt
- hier einige Neuerungen:
 - **Generische Typen**
 - **Aufzählungstypen**
 - **variable Anzahl von Parametern**
 - **for each-Schleife**
 - **Auto Boxing**
- jetzt: sogar schon > Java 7.*,
für unsere Belange uninteressant

6.1 Generische Typen

Generische Typen

- in **Üb-Aufg. 41:**
 - Datenstruktur **Liste** allgemein für **Object** definiert
 - > kann alle Objekte als Einträge enthalten
 - > nicht parametrisierbar
 - schön wäre: verwendbar nur für **Auto** und Unterklassen
- in **C++ Standard Template Library**
 - kann Klassen parametrisieren
- in **Java 5.0 Einführung von generischen Typen** für
 - Library-Klassen: **Vector<Auto>**
 - selbst definierte Klassen:

Generische Liste

- **in Klassenkopf: Einfügen eines Parameters für gen. Typ:**

```
public class Liste<DataFormal> {
    public DataFormal obj = null;
    public void vorhängen(DataFormal obj)
    public DataFormal head() ...
```

- **als Listeneinträge nur noch Objekte**

- der Klasse **DataAktuell**
- und abgeleiteten Klassen

Instanzierung & Verwendung

- **bei Instanzierung Angabe des Eintragstyps**

```
Liste<Auto> listAuto = new Liste<Auto>();
-> nur noch Auto in Liste vorhanden:

listAuto.vorhängen(new Auto())           //o.k.
listAuto.vorhängen(new Kreis())          //geht nicht!
```

- **Aufruf der Methoden**

```
System.out.println(listAuto.head())
ruft toString() von Auto auf
```

- **hier: kein Cast erforderlich, manchmal: doch**
- **aktueller Parameter kann beliebige Klasse sein**

Einschränken des generischen Typs

- **Einschränkung zugelassener Klassen für akt. Par.**

- rechts von formalem Parameter `extends MaxKlassen`

```
class Liste<DataFormal extends Kl & Il & I2> {
```

- nur Klassen als aktuelle Parameter zugelassen, die
 - Klasse `Kl` erweitern und
 - Interfaces `I1` und `I2` implementieren

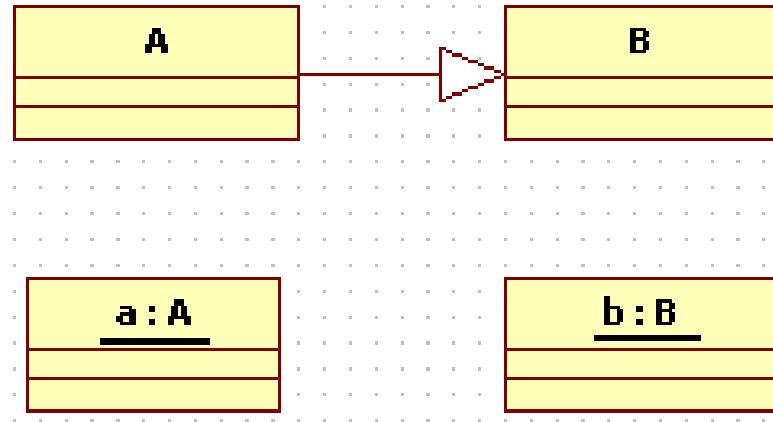
```
class Liste<DataForm extends Auto>
```

- dann:

```
Liste<Lkw> ... // o.k.
```

```
Liste<Kreis> ... // geht nicht
```

=> hinter `extends` beliebig viele Interfaces, aber nur 1 Klasse



```
b = a; // geht

List<B> lb = new List<B>();

List<A> la = new List<A>();

lb = la; // geht nicht
```

⇒ Hier keine Typkompatibilität

Vererbung in generischen Klassen

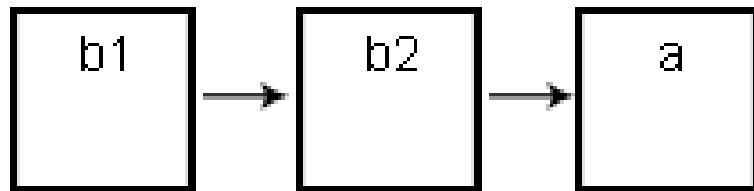
- **generische Klassen nicht typkompatibel**

```
class A extends B ...  
  
Liste<A> lista = new Liste<A>();  
  
Liste<B> listb = new Liste<B>();  
  
listb = lista           //geht nicht!!!
```

- **Einträge der generischen Klassen aber typkompatibel**

```
listb.vorhangen(new A());
```

Mögliche Liste:



6.2 Aufzählungstypen

Aufzählungstypen

- Ein Aufzählungstyp ist eine Menge von Konstanten

- Definition

```
enum GrundFarbe {blau, rot, gelb}
```

- Zugriff auf eine Konstante

```
GrundFarbe.blau
```

- Zuweisung an Variable

```
GrundFarbe variable = GrundFarbe.blau;
```

- Einsatz in switch-Anweisung

```
switch (variable) {
    case blau: ...           //ohne GrundFarbe
    case rot: ...
    ...
}
```

6.3 Variable Anzahl von Parametern

variable Anzahl von Parametern

- Methoden mit variabler Anzahl von Par. eines Typs

- Parameterliste intern als Array dargestellt

-> Zugriff auf Parameterliste wie auf Array

- höchstens eine Liste von Parametern pro Methode

- Liste von Parametern als letzter Parameter

- Beispiel:

- Summiere die ersten Elemente einer int-Liste.
- Falls ersten > Listenlänge: summiere ganze Liste

=> Liste als letzter Parameter, da sonst bei Aufruf nicht klar, was zur Liste gehört!

Summation

```
static int sum(int ersten, int ...is) {
    int out = 0;
    int min = is.length;
    if (ersten < min) {
        min = ersten;
    }
    for(int i=0; i<min; i++) {
        out += is[i];
    }
    return out;
}
```

is
↑
Aufruf: sum(5,1,2,3)

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Java 5.0 5. Oktober 2016 | 9

=> vor der Liste ein Parameter => klar, daß 1,2,3 zur Liste gehören

6.4 For Each

For Each Schleife

- **einfachere Form um über alle Elemente eines Feldes oder einer Collection (Vector, ...) zu laufen**

```
static int sum1(int ...is) {
    int out = 0;
    for(int element:is) {
        out += element;
    }
    // anstelle von:
    // for (int i=0;i<is.length;i++) {out += is[i];}
    return out;
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Java 5.0 5. Oktober 2016 | 10

$\forall element \in is : out+ = element;$

1. Element int ist Typ der Elemente

6.5 Autoboxing

Auto Boxing

- **in Collection Klassen**
 - nur echte Objekte
 - keine primitiven Typen
- **dazu: zu jedem primitivem Typ Wrapper Klasse**
`int -> Integer`
- **bisher: Umwandlung in Wrapper-Objekt umständlich**
`1 -> new Integer(1)`
- **jetzt: in beiden Richtungen Umwandlung automatisch**

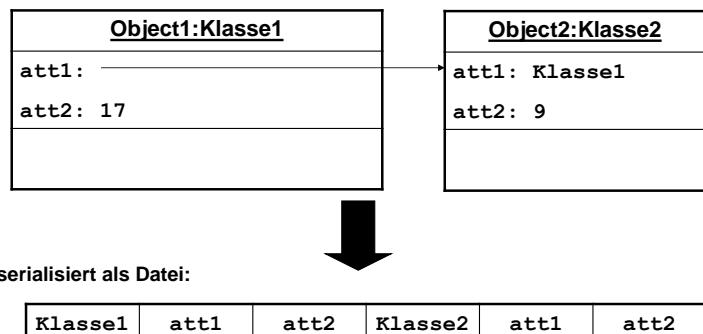
```
Stack<Integer> stack = new Stack<Integer>();  
stack.push(1); //alt: stack.push(new Integer(1))  
int wert = stack.pop(); //alt: stack.pop().intValue()
```

=> braucht jetzt nicht mehr überlegen, ob primitiver Datentyp oder Wrapper-Objekt

7 Serialisierung, Deserialisierung und Streams

Problemstellung

Objekt mit aggregierten Objekten:



- Speicherung von Objekten
- Netzwerkübertragung von Objekten

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender (De)Serialisierung 5. Oktober 2016 | 1

Motivation: Man muss die Möglichkeit haben, Objekte persistent zu machen (auf Hintergrundspeicher zu speichern), um sie wiederzuverwenden (nach Beendigung des Programms).

Normalerweise: Datenbank, hier: File-System

Def.: (Serialisierung, Deserialisierung)

Unter **Serialisierung** verstehen wir die Fähigkeit, ein Objekt aus dem Hauptspeicher, zusammen mit all seinen aggregierten Objekten in ein Format zu konvertieren, das es erlaubt, obige Objekte in eine Datei zu schreiben oder übers Netz zu übertragen.

Deserialisierung: umgekehrt

Zwei Klassen aus Paket *java.io* ⇒ import notwendig

- **ObjectOutputStream:** Serialisierung (schreibend)
- **ObjectInputStream:** Deserialisierung (lesend)

Können *IOException* werfen ⇒ catch-or-throws notwendig!

7.1 Die Klasse ObjectOutputStream

Erzeugung durch Konstruktor:

```
public ObjectOutputStream(OutputStream out) throws IOException
```

ObjectOutputStream ist komplexer als ein herkömmlicher Byte-Stream, nutzt aber dessen Funktionalität, darum Byte-Stream-Object (hier: FileOutputStream (Unterklasse von OutputStream)) in Parameter übergeben.

write-Methoden in ObjectOutputStream

```
public final void writeObject(Object obj)
    throws IOException
public void writeBoolean(boolean data)
    throws IOException
public void writeByte(int data)
    throws IOException
public void writeShort(int data)
    throws IOException
public void writeChar(int data)
    throws IOException
public void writeInt(int data)
    throws IOException
public void writeLong(long data)
    throws IOException
public void writeFloat(float data)
    throws IOException
public void writeDouble(double data)
    throws IOException
```

Bsp.: Anwendung einer write-Methode

gegeben sei folgender ObjectOutputStream:

true	17.3	'H'			
------	------	-----	--	--	--

Ausführung der Methode `writeBoolean(true)` liefert:

true	17.3	'H'	true		
------	------	-----	------	--	--

Bei *write*: Wenn Parameter größer als spezifizierter, dann **Abhacken**

writeObject: schreibt auf OutputStream

- Klasse des übergebenen Objekts
- Signatur der Klasse
- alle nicht-statischen Attribute des übergebenen Objekts inkl. der aus allen Oberklassen geerbten Attribute, falls die Oberklasse *Serializable* (Interface ohne Methoden = Flag) implementiert

Man entscheidet selbst, ob Klasse serialisierbar sein soll, durch Zufügen von *implements Serializable*

=> Aggregierte Objekte müssen auch serialisierbar sein.

Vorsicht: *writeObject* sehr aufwendig!

Bsp.: Die Klasse Zeit

```
import java.io.*;  
  
public class Zeit implements Serializable {  
  
    private int stunde;  
    private int minute;  
  
    public Zeit(int stunde, int minute) {  
        this.stunde = stunde;  
        this.minute = minute;  
    }  
  
    public String toString() {  
        return stunde + ":" + minute;  
    }  
}
```

Serialisierung eines Objekts von Zeit

```
import java.io.*;

public class Serialisiere {
    public static void main(String[] args) {
        try {
            FileOutputStream fs = new FileOutputStream("fos.s");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            Zeit zeit = new Zeit(10,20);
            os.writeObject(zeit);
            os.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Oder: `ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("fos.s"));`

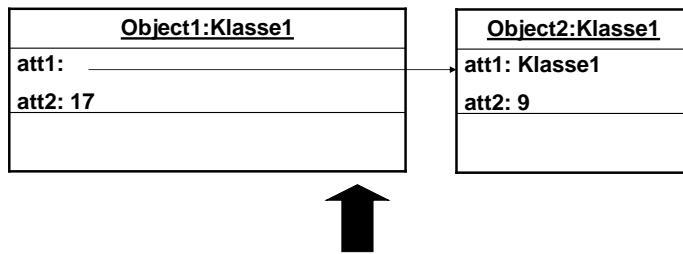
⇒ `new FileOutputStream("fos.s")` erzeugt ein anonymes Objekt (ohne Namen)

Wenn `implements Serializable` in `Zeit` vergessen wurde, dann wird eine `NotSerializableException` ausgelöst. Es können mehrere `write`-Befehle hintereinander auftreten. Wichtig: Reihenfolge bei Deserialisierung

Jetzt: Deserialisierung, umgedrehte Problemstellung

Problemstellung

Objekt mit aggregierten Objekten:



Datei mit serialisierten Objekten:

Klasse1	att1	att2	Klasse2	att1	att2
---------	------	------	---------	------	------

- Rekonstruktion von Objekten aus Dateien
- Netzwerkübertragung von Objekten

7.2 Die Klasse ObjectInputStream

Konstruktor: `public ObjectInputStream(InputStream in)`

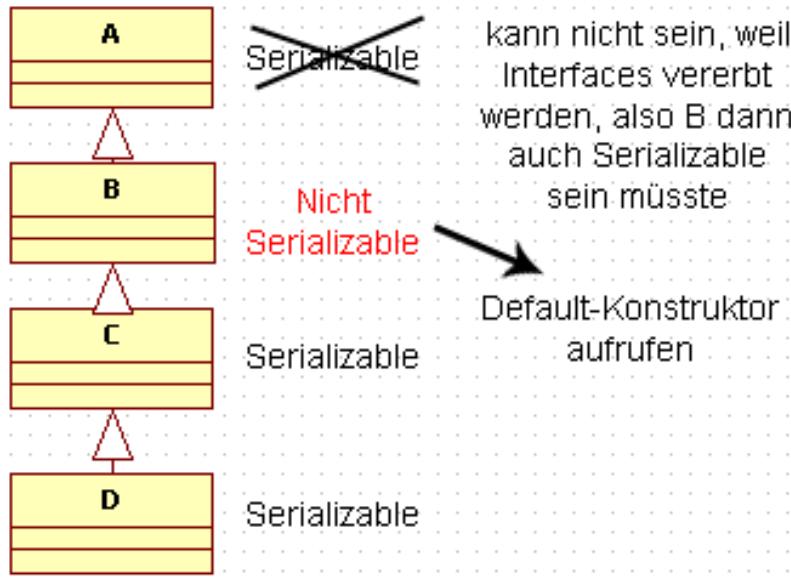
read-Methoden invers zu *write*-Methoden

read-Methoden in ObjectInputStream

```
public final Object readObject()  
    throws IOException, OptionalDataException,  
          ClassNotFoundException  
public boolean readBoolean()  
    throws IOException  
public byte readByte()  
    throws IOException  
public short readShort()  
    throws IOException  
public char readChar()  
    throws IOException  
public int readInt()  
    throws IOException  
public long readLong()  
    throws IOException  
public float readFloat()  
    throws IOException  
public double readDouble()  
    throws IOException
```

Deserialisieren (stark vereinfacht):

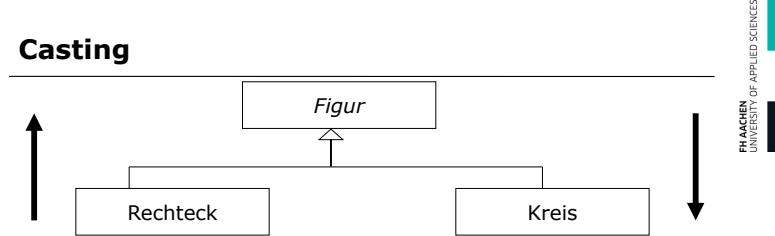
- neues Objekt der zu deserialisierenden Klasse wird angelegt
- Attribute werden mit Defaultwerten vorbelegt
- Default-Konstruktor der ersten nicht-serialisierbaren Oberklasse wird aufgerufen (und damit auch aller Default-Konstruktoren von deren Oberklasse)



- serialisierte Daten vom File gelesen und in Objekt geschrieben

Zeit zeit = (Zeit) is.readObject(); // Typecast (DownCast) notwendig, da Object als Rückgabe, UpCast wäre kein Problem

Einschub:



`Figur[] figuren = new Figur[4];`

UpCast: Zuweisung eines Rechtecks (Kreises) an Figur
`figuren[1] = new Rechteck(1,1,1,1);
 figuren[2] = new Kreis(1,1,1);`

DownCast: Zuweisung einer Figur an Rechteck (Kreis)
`Rechteck rechteck = (Rechteck) figuren[1];
 Kreis kreis = (Kreis) figuren[2];`

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender (De)Serialisierung 5. Oktober 2016 | 8

- UpCast: Oberklassen-Referenz = Unterklassen-Referenz \Rightarrow klappt
- DownCast: Unterklassen-Referenz = (Unterklasse) Oberklassen-Referenz \Rightarrow braucht hier Cast, da in dieser Richtung keine Zuweisungskompatibilität

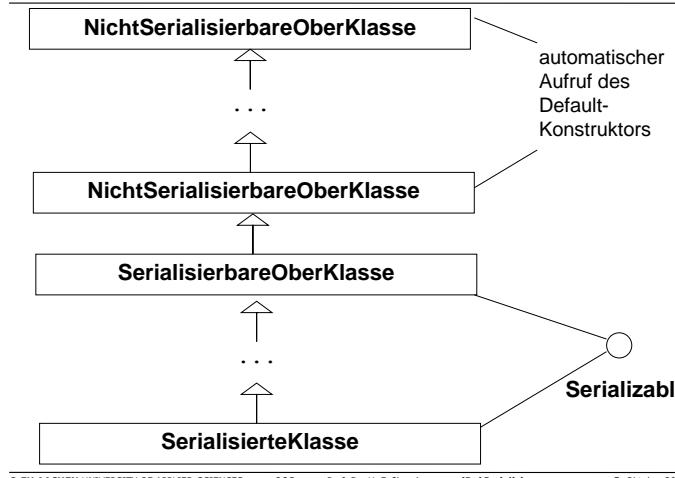
Deserialisierung eines Objekts von Zeit

```
import java.io.*;

public class Deserialisiere {
    public static void main(String[] args) {
        try {
            FileInputStream fs = new FileInputStream("fos.s");
            ObjectInputStream is = new ObjectInputStream(fs);
            Zeit zeit = (Zeit) is.readObject();
            System.out.println(zeit);
            is.close();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender (De)Serialisierung 5. Oktober 2016 | 9

Deserialisieren in Vererbungshierarchien



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender (De)Serialisierung 5. Oktober 2016 | 10

7.3 Überlagern von equals

in *Object*: `public boolean equals(Object obj)`

Bisher: `boolean equals(Kreis k)` in Klasse *Kreis* ⇒ Überladung

Ab Praktikum 2: Überlagern, da sonst Fehler, wenn *equals* aus Bibliotheken aufgerufen wird

In *Zeit*:

```
public boolean equals(Object obj) {  
  
    return ((obj != null) &&  
            (obj instanceof Zeit) &&  
            (this.stunde == ((Zeit) obj).stunde) &&  
            (this.minute == ((Zeit) obj).minute));  
  
}
```

Überlagerung von equals

- bisher: **equals überladen**

```
boolean equals(Zeit zeit) {  
    return(    this.stunde == zeit.stunde  
            &&    this.minute == zeit.minute  );  
}
```

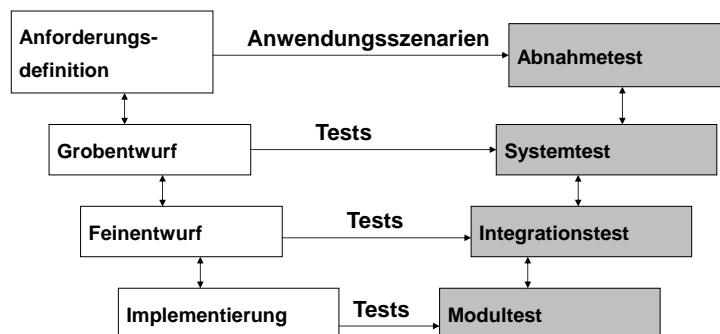
- muss aber überlagert werden, da von Java-Bibs aufgerufen

```
boolean equals(Object zeit) {  
    return(    zeit != null  
            && zeit instanceof Zeit  
            && this.stunde == ((Zeit) zeit).stunde  
            && this.minute == ((Zeit) zeit).minute);  
}
```

8 Unit-Tests

Softwaretests

- verschiedene Testformen
- Testen im V-Modell:



Vorurteile gegen Testen

- **keine Zeit zum Testen:**
 - großer Zeitdruck
 - > weniger Tests
 - > instabiler Code
 - > mehr Fehlermeldungen vom Kunden
 - > mehr Debugging-Zeit
 - > noch größerer Zeitdruck
- **Testen langweilig und stupide**
- **Mein Code ist praktisch fehlerfrei, gut genug**
- **Testabteilung testet**

Testen in Softwareentwicklung integrieren!

Literatur

- Uwe Vigenschow: *Objektorientiertes Testen und Testautomatisierung in der Praxis*; dpunkt-Verlag.
- Johannes Link: *Softwaretests mit JUnit –Techniken der testgetriebenen Entwicklung –*; dpunkt-Verlag.
- Andreas Spillner & Tilo Linz: *Basiswissen Softwaretest – Aus- und Weiterbildung zum Certified Tester (Foundation Level)*; dpunkt-Verlag.
- Robert V. Binder: *Testing Object-Oriented Systems – Models, Patterns, and Tools*; Addison-Wesley.
- Erich Gamma & Kent Beck: *JUnit Framework*; www.junit.org.
- Frank Westphal: *Testgetriebene Entwicklung mit JUnit & FIT*; dpunkt-Verlag

manuelle Tests

- **Testen von BenutzerVerwaltungAdmin durch main-Progr.:**

```
BenutzerVerwaltungAdmin adm =
    new BenutzerVerwaltungAdmin();
adm.dbInitialisieren(); //mit und ohne
Benutzer ben = new Benutzer("Heinz", "Pwd");
System.out.println(adm.benutzerOk(ben));
adm.benutzerEintragen(ben);
System.out.println(adm.benutzerOk(ben));
adm.benutzerEintragen(ben);
adm.benutzerLöschen(ben);
System.out.println(adm.benutzerOk(ben));
adm.benutzerLöschen(ben);
```

- **Auswertung durch scharfes Hinsehen:**

- Vergleich: erwartetes und erhaltenes Ergebnis

Probleme bei manuellen Tests

- **häufige Ausführung der Tests**
 - zeitaufwändig
 - ermüdend
 - > Fehler übersehen
 - Tests vergessen
- **häufig am Ende keine Zeit mehr, noch Tests**
 - zu überlegen und
 - Durchzuführen

->Testautomatisierung

Testautomatisierung

- **Tests & Überprüfung**
 - programmierbar
 - automatisch ausführbar
- **lohnt sich ziemlich schnell, wenn mehrmals ausgeführt**
- **möglichst Tests vor oder während der Entwicklung programmieren (test-getriebene Entwicklung) ->**
 - sichere Programme
 - einfaches Design
 - effektive Schnittstellen
 - bessere Spezifikation
- **zusätzlicher Programmieraufwand der Tests führt meist nicht zu zusätzlichem Gesamtaufwand**

Was automatisiert getestet?

- **Erstellung von Test-Treibern**
- **Testen einzelner Methoden**
 - Vor- und Nachbedingungen der Methoden
 - welche Eingaben, welche Ausgaben
 - Design by Contract
- **Testen des Protokolls einer Klasse (Kettentest)**
 - typische Verwendungsszenarien von Instanzen der Klasse
- **Testen von Interaktionen**
 - zwischen mehreren Objekten

Anforderungen an Testframework

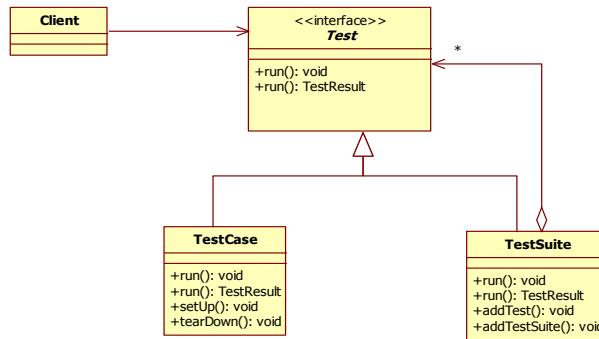
- **Testsprache == Programmiersprache**
 - gibt auch Skriptsprachen zur Erstellung von Testtreibern
 - Verwendung zweier Sprachen: geistiges Umschalten
- **Anwendungscode und Testcode trennbar**
 - wichtig für Auslieferung des Produkts
 - unabhängige Strukturierung des Testcodes
- **Ausführung der Testfälle voneinander unabhängig**
 - sonst Fehlalarm an anderer Stelle
- **Zusammenfassung in Testsuiten**
 - oft viele Tests zusammengehörig
- **Protokollierung der Tests**
 - auf einen Blick, Erfolg oder Misserfolg erkennbar

JUnit

- **Open-Source-Projekt**
- **von Kent Beck und Erich Gamma entwickelt**
- **Schreiben von Tests und Code getrennt**
- **Tests in selber Sprache geschrieben wie Code**
- **Tests laufen automatisch ab**
- **Tests nehmen Verifikation der Testergebnisse vor**
- **automatische Protokollierung der Tests**
- **Tests verschiedener Autoren einfach kombinierbar**
- **gemäß Composite-Pattern strukturiert:**
 - Basisfall: **TestCase**
 - Induktion: **TestSuite**

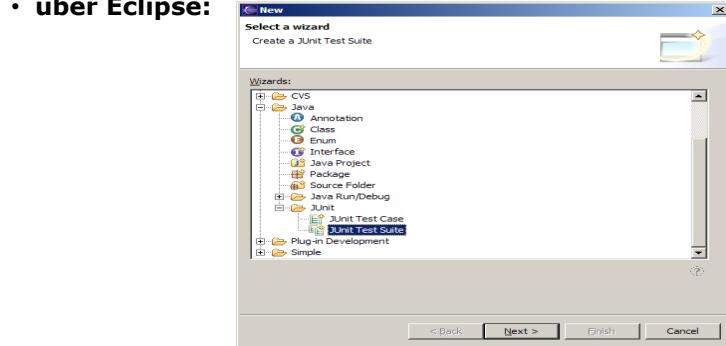
JUnit-Framework

- **in TestCase:** **einzelner Test**
- **in TestSuite:** **Sammlung von Tests**



Anwendung von JUnit

- läuft innerhalb von TestRunner
- über GUI: junit.swingui.TestRunner
- über Konsole: junit.text.TestRunner
- über Eclipse:



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Test 5. Oktober 2016 | 11

Testklasse zu Benutzer erstellen

```
import junit.framework.TestCase;
public class BenutzerTest extends TestCase {
    private Benutzer ben1;
    protected void setUp() throws Exception {
        super.setUp();
    }
    protected void tearDown() throws Exception {
        super.tearDown();
    }
    // hier Testmethoden hin, siehe nächste Folie
    public static void main(String[] args) {
        junit.textui.TestRunner.run(BenutzerTest.class);
    }
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Test 5. Oktober 2016 | 12

Testmethoden zu BenutzerTest

```
public void testBenutzerStringString() {           //Konstruktortest
    Benutzer ben1 = new Benutzer("uid","pwd"); //-> setUp()
    assertEquals("uid",ben1.userId);
    assertEquals("pwd", String.valueOf(ben1.passWort));
}

public void testEqualsBenutzer() {                  //Equalstest
    Benutzer ben1 = new Benutzer("uid","pwd");      //-> setUp()
    Benutzer ben2 = new Benutzer("uid","pwd");
    assertEquals(ben1,ben2);
    assertNotSame(ben1,ben2);
    ben1 = ben2;
    assertEquals(ben1,ben2);
    Benutzer ben3 = new Benutzer("ui","pw");
    assertFalse(ben1.equals(ben3));
}

public void testToStringBenutzer() {                // noch machen
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Test 5. Oktober 2016 | 13

assert-Methoden

• Klasse Assert stellt Methoden zur Verfügung:

- assertEquals(expected: Typ, actual: Typ), für
 - alle primitiven Typen
 - für Object: ruft equals auf
- assertEquals(expected, actual, delta: double)
- assertNotNull(obj: Object)
- assertNull(obj: Object)
- assertSame(expected: Object, actual: Object)
 - Test auf gleiche Referenz
- assertTrue(condition: boolean)
- assertFalse(condition: boolean)
- fail()
-

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Test 5. Oktober 2016 | 14

ben1 und ben2 vergleichen:

- assertTrue(ben1.equals(ben2));
- assertEquals(ben1, ben2); ⇐ bevorzugen, weil Fehlermeldung gehaltvoller (enthält neben erhaltenem Wert auch noch erwarteten Wert)

setUp() - & tearDown() -Methoden

- **mit setUp()**

- Initialisierung vor jeder Testmethode:

```
protected void setUp() throws Exception {
    super.setUp();                                // am Anfang
    Benutzer ben1 = new Benutzer("uid", "pwd");
}
```

- für Variablen private-Attribut in Testklasse anlegen

- **mit tearDown()**

- Freigeben von Ressourcen nach jeder Testmethode:

```
protected void tearDown() throws Exception {
    super.tearDown();                            // am Ende
}
```

TestSuite erstellen

```
import junit.framework.Test;
import junit.framework.TestSuite;
public class AllTests {
    public static Test suite() {
        TestSuite suite = new TestSuite("Test prak3");
        suite.addTestSuite(BenutzerTest.class);

        suite.addTestSuite(BenutzerVerwaltungAdminTest.class);
        // nach Praktikum 3
        return suite;
    }
    public static void main(String[] args) {
        junit.textui.TestRunner.run(AllTests.suite());
    }
}
```

- *fail()*: Test, ob Exceptions korrekt ausgeführt werden => immer hinter Programmtext, der Exception auslösen soll!

Liefert bei Aufruf Fehler!!!!

Test von Exceptions

- möchten testen, ob Exceptions richtig geworfen werden:

- in equals von Benutzer:

```
if (ben == null)
    throw new NullPointerException("Par ist null!");
```

- in BenutzerTest:

```
public void testNullPointerException() {
    try { Benutzer ben = null;
        this.ben.equals(ben1);
        fail("ben must not be null");
    } catch (NullPointerException e) {
        e.printStackTrace();
    }
}
```

- wenn Exception ausgelöst, kommt man nicht zu fail("..."), sondern in catch => kein rot, sondern grün
- sonst kommt man zu fail => wird als Fehler angesehen und bricht mit rot ab

Änderungen in jUnit 4.0

- Testklasse nicht mehr von `TestCase` abgeleitet
- dafür Package `org.junit` importiert
- Klasse muss öffentlichen Default-Konstruktor haben
- Test-Methoden müssen nicht mehr mit "test" beginnen
- dafür mit der Annotation `@Test` markiert sein
Bsp.: `@Test public void adding() { ... }`
- Methode muss Rückgabewert `void` und keine Parameter haben
- `@Before` und `@After` markieren Setup- bzw. Teardown-Aufgaben, die für jeden Testfall wiederholt werden (müssen öffentlich sein)
- `@BeforeClass` und `@AfterClass` markieren Setup- oder Teardown Aufgaben, nur einmal pro Testklasse ausgeführt werden.
Die Methoden müssen statisch sein.
- `@Ignore` kennzeichnet temporär nicht auszuführende Testfälle.

Wie testen?

- **Fehlerfreiheit durch Testen nicht nachweisbar!**
- **kann beliebig viel testen**
- **häufig eher zu wenig getestet**
- **Was ist richtiges Testmaß:**
 - Finden möglichst vieler Bugs mit überschaubarem Aufwand
- **akzeptables Fehlerniveau hängt von Systemart ab**
- **Testaufwand wächst exponentiell zum Nutzen**
- **Regel: "Test everything that could possibly break."**
 - sagt nichts
 - bekommt mit der Zeit heraus, welche Fehler man macht
 - finde eigenes optimales Testniveau

Was testen?

- **für jede nicht triviale Klasse mindestens eine Testklasse**
 - triviale Klassen: Datenklassen, Exceptionklassen
- **setter / getter normalerweise nicht testen**
- **nicht öffentliche Methoden normalerweise nicht testen**
 - werden bei Verwendung der öffentlichen mit getestet
- **Tests sollten normalerweise nicht auf Innereien zugreifen**
- **möglichst keine komplexen Integrationstests**
 - bei Änderungen schwierig nachzuvollziehen
 - finden jedoch oft nicht erwartete Fehler
- **keine Tests für Tests**
 - getestete Klassen dienen automatisch als Tests der Tests

Testabdeckung

- **Wieviel von X wird durch meine Tests abgedeckt?**
- **X steht für**
 - spezifikationsbasierte Abdeckung (Blackbox Test)
 - überprüfe für Eingaben, ob richtige Ausgaben geliefert
 - Problem: Eingabenmenge oft unendlich
 - codebasierte Abdeckung (Whitebox Test)
 - überprüfe, ob alle Pfade durchlaufen werden
 - häufig automatisch durchführbar
 - sagt nichts über korrekte Funktionalität & Fehlerfreiheit aus
 - deutet auf fehlerträchtige Teile hin

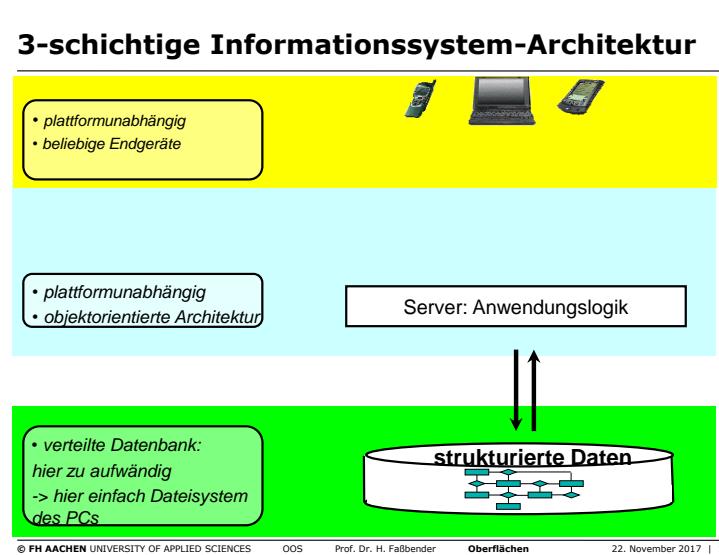
spezifikationsbasierte Tests

- **Testen des Ein-/Ausgabe-Verhaltens für alle nichttrivalen Methoden**
- **finde für möglicherweise unendlich viele Eingaben endlich viele gute Repräsentanten**
Äquivalenzklassenbildung: Eingaben mit gleicher Ausgabe
Randfälle besonders wichtig:
 - leere Liste & volle Liste (Induktionsanfänge)
 - fastleere Liste & fastvolle Liste
 - Mittlere Fälle zusammenfassen (Induktionsschritte)
 - sinnvolle Repräsentanten
- **immer gut- und bösartige Eingaben testen**
 - Löschen für vorhandenen und nicht vorhandenen Benutzer

9 Oberflächenprogrammierung

9.1 Erstellen von Fenstern

Oberflächenprogrammierung um Client zu programmieren:



Benutzerschnittstelle: Graphical User Interface (GUI)

- Client als Fensteroberfläche
- hierzu: diverse Java-Bibliotheken vorhanden

Historie der Java-GUI-Programmierung

• Abstract Windowing Toolkit (AWT)

- steht seit JDK 1.0 als Grafikbibliothek zur Verfügung, enthält:
- grafische Primitivops:
 - > Zeichnen von Linien oder Füllen von Flächen
 - > Ausgabe von Text
- Methoden zur Steuerung des Programmablaufs bei Ereignissen
- Dialogelemente zur Kommunikation mit Anwender (Textfelder)
- fortgeschrittene Grafikfunktionen:
 - > Darstellung und Manipulation von Bitmaps
 - > Ausgabe von Sounds

Historie der Java-GUI-Programmierung 2

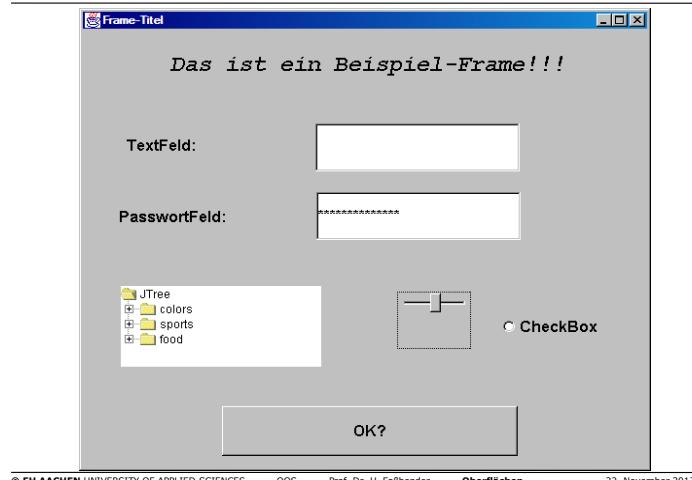
• Probleme bei AWT

- greift auf Ressourcen der Zielplattform zu
 - > kleinster gemeinsamer Nenner, da sonst keine Plattformunabh.
 - > Portierungsprobleme, da Layout jeweils neu überdacht
 - nur Grundmenge an Dialogelementen
 - > vieles nicht oder nur mit viel Zusatzaufwand implementierbar

Historie der Java-GUI-Programmierung 3

- **Probleme bei Swing**
 - ressourcenhungig
 - anfangs noch sehr fehlerhaft
 - teilweise noch nicht direkt von Browern unterstützt
- **Namen von Swing-Klassen**
 - Unterschied zu AWT-Klassen durch vorgestelltes "J,,
 - muss folgende Bibliotheken importieren:
 - > java.awt.*
 - > javax.swing.*

Beispiel-Frame



einfacher Frame



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Oberflächen 22. November 2017 | 6

Frame-Code

```

01: public class FrameBsp extends JFrame {
02:     public FrameBsp() {
03:         this.setTitle("Erster eigener Frame");
04:         this.setSize(new Dimension(400,300));
05:         this.labelHallo.setBounds(67,60,289,28);
06:         this.labelHallo.setFont(new java.awt.Font("Serif",1,20));
07:         this.jButtonAbbrechen.setBounds(87,140,189,128);
08:         this.jButtonAbbrechen.setText("Alles klar?");
09:         this.setLayout(null);
10:         this.add(labelHallo);
11:         this.add(jButtonAbbrechen);
12:         this.setVisible(true);
13:     }
14:     private JLabel labelHallo = new JLabel("Ich bin ein
15:                                         Label!");
16:     private JButton jButtonAbbrechen = new JButton();
17:     public static void main(String[] args) {
18:         FrameBsp frameBsp = new FrameBsp();
19:     }

```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Oberflächen 22. November 2017 | 7

Zeile 01: Ableitung von Frame-Klasse JFrame

Zeile 02 - 13: generellen Einstellungen im Konstruktor

Zeile 14 - 15: private-Attribute für Komponenten ⇒ Attribute außerhalb des Konstruktors, weil später noch drauf zugegriffen werden soll!

Zeile 16 - 18: main: Objekt anlegen

Im Detail: Zeile 03: setzt Name in oberer Fensterleiste

Zeile 04: setzt Größe des Frames (anonymes Objekt vom Typ *Dimension*) \Rightarrow Einstellungen generell für Frame, jetzt Einstellungen für Komponenten:

Zeile 05: Größe und Position des Labels

Zeile 06: Font des Labels (anonymes Objekt vom Typ *Font*)

Zeile 07 - 08: Größe, Position und Beschriftung des Buttons (Beschriftung alternativ per Konstruktor, wie bei JLabel)

Zeile 09: Layout auf null setzen, Einträge dadurch beliebig positionierbar

Zeile 10 - 11: Hinzufügen des Buttons und Labels zum sichtbaren Bereich

Zeile 12: Fenster sichtbar machen

9.2 Schließen von Fenstern

Nicht *setVisible(false)*, sondern *dispose()*

einige Methoden der Klasse JFrame

```
public void setTitle(String title)
public void setSize(int width, int height)
public void setLocation(int x, int y)
public void setLocation(Point p)
public void setBounds(int x, int y, int width, int
height)
public Point getLocation()
public void setVisible(boolean b)
public void dispose()
public void add(FensterElement fe)
public void setLayout(null)
```



einige Methoden der Klasse JLabel

```
public void setText(String title)
public void setSize(int width, int height)
public void setLocation(int x, int y)
public void setLocation(Point p)
public void setBounds(int x, int y, int width,
                     int height)
public Point getLocation()
public voidsetFont(Font f)
```

einige Methoden der Klasse JTextField

```
public void setText(String title)
public void setSize(int width, int height)
public void setLocation(int x, int y)
public void setLocation(Point p)
public void setBounds(int x, int y, int width, int
                     height)
public Point getLocation()
public voidsetFont(Font f)
```

einige Methoden von JPasswordField

```
public void setText(String title)
public void setSize(int width, int height)
public void setLocation(int x, int y)
public void setLocation(Point p)
public void setBounds(int x, int y, int width,
                     int height)
public Point getLocation()
public void setFont(Font f)

public void setEchoChar(char c)
```

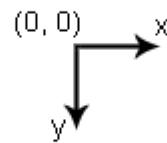
einige Methoden der Klasse JButton

```
public void setText(String title)
public void setSize(int width, int height)
public void setLocation(int x, int y)
public void setLocation(Point p)
public void setBounds(int x, int y, int width,
                     int height)
public Point getLocation()
public void setFont(Font f)
```

einige Methoden von JCheckBox

```
public void setLabel(String title)
public void setSelected(boolean state)
public boolean isSelected()
public void setSize(int width, int height)
public void setLocation(int x, int y)
public void setLocation(Point p)
public void setBounds(int x, int y, int width,
                     int height)
public Point getLocation()
public void setFont(Font f)
```

Vorsicht bei *setLocation*: Koordinatenursprung ist links oben:



Bei Getter-Methoden: Bei *boolean* als Rückgabetyp nicht vorangestelltes *get*, sondern *is* (z.B. *isSelected()*)

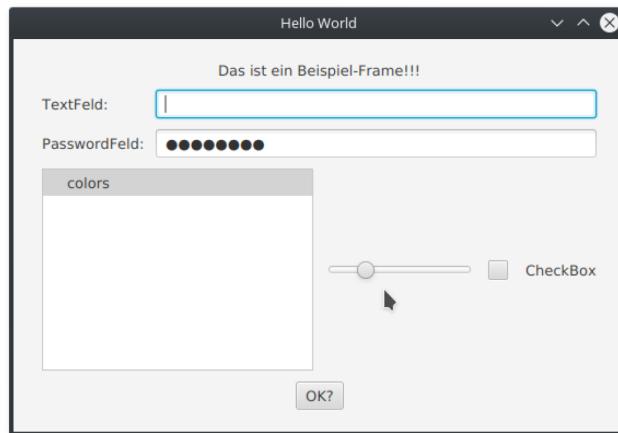
9.3 JavaFX

9.3.1 Einführung

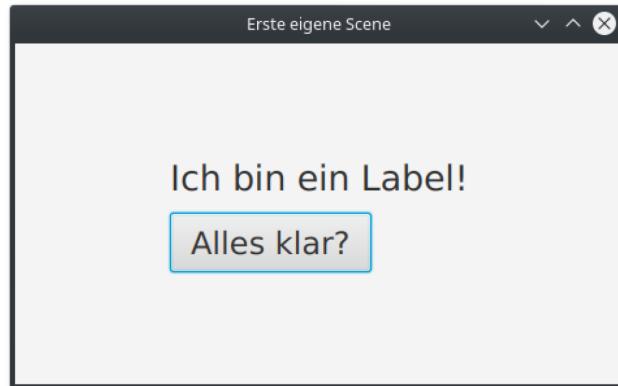
Historie der Java-GUI-Programmierung 4

- **JavaFX**
 - seit JDK 7u6 Bestandteil des JDK
 - seit JDK 8u33 Unterstützung für ARM (Architektur)
 - Design per CSS anpassbar
 - Design kann per XML deklariert werden
(andere Beispiele: Android, XHTML)
 - Scene Builder ermöglicht einfache GUI-Erstellung
 - leichtere Umsetzung des MVC-Patterns
- **Probleme bei JavaFX**
 - kleinere Community als Swing
 - > weniger 3rd-Party-Bibliotheken

Beispiel Scene



einfache Scene



9.3.2 1. Möglichkeit der Implementierung: Programmatisch

Scene-Entwicklung analog zu Swing:

Scene in Java

```

1  @Override
2  public void start(Stage primaryStage) throws IOException {
3      GridPane gridPane = new GridPane();
4      gridPane.setHgap(10);
5      gridPane.setVgap(10);
6      gridPane.setPadding(new Insets(10, 10, 10, 10));
7      gridPane.add(new Label("Ich bin ein Label!"), 0, 0);
8      ;
9      Button button = new Button("Alles klar?");
10     gridPane.add(button, 0, 1);
11
12     primaryStage.setTitle("Erste eigene Scene");
13     primaryStage.setScene(new Scene(gridPane));
14     primaryStage.show();
    }
```

Die Hauptklasse bei JavaFX Applikationen erbt von Application und man überlagert die start Methode.

9.3.3 2. Möglichkeit der Implementierung: Deklarativ

Trennung von Logik und Darstellung: MVC-Muster

Scene-FXML zur Darstellung und Controller zur späteren Ereignisverarbeitung

Scene-FXML und Controller

```

1 <?import javafx.geometry.Insets?>
2 <?import javafx.scene.layout.GridPane?>
3 <?import javafx.scene.control.Button?>
4 <?import javafx.scene.control.Label?>
5 <GridPane fx:controller="sample.Controller"
6     xmlns:fx="http://javafx.com/fxml" alignment=
7         ="center" hgap="10" vgap="10">
8     <padding><Insets top="20" right="20" bottom="20" 
9         left="20"/> </padding>
10    <Label text="Ich bin ein Label!" GridPane.>
11        rowIndex="0"/>
12    <Button text="Alles klar?" GridPane.rowIndex="1">
13        />
14 </GridPane>
15 public class Controller {}
```

Application-Code

```

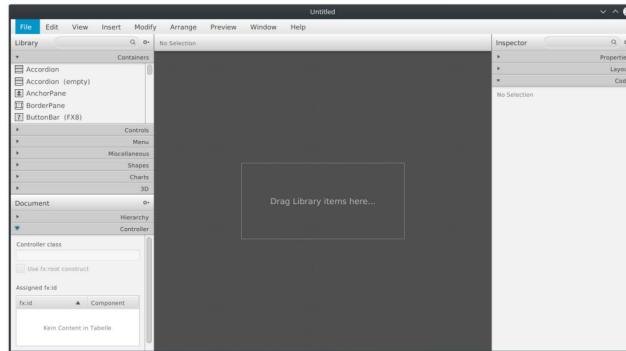
1 public class Main extends Application {
2     @Override
3     public void start(Stage primaryStage) throws 
4         Exception{
5         Parent root = FXMLLoader.load(getClass().>
6             getResource("sample.fxml"));
7         primaryStage.setTitle("Erste eigene Scene");
8         primaryStage.setScene(new Scene(root));
9         primaryStage.show();
10    }
11    public static void main(String[] args) {
12        launch(args);
13    }
14 }
```

Problem: Trennung erschwert Verwaltung

-> Muss verschiedene Dateien konsistent halten.

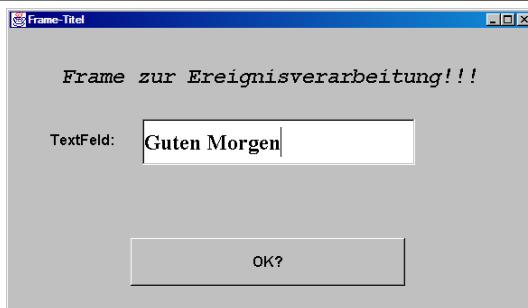
Darum: von Tool, namens *JavaFX Scene Builder*, unterstützen lassen, mit dem man über eine grafische Oberfläche seine Designs erstellen kann. Dieses Tool erzeugt dann die fxml Datei.

SceneBuilder



10 Ereignisverarbeitung (Event Handling) in Java und weitere Konzepte

Ein Beispiel-Frame



Was passiert, wenn Button "Ok?" mit Maus angeklickt?

gar nichts!!!

Zur Reaktion und Weiterverarbeitung von Ereignissen

- Mausklicks
- Bewegungen des Mauszeigers
- Tastatureingaben
- Fensterveränderungen

Der Text im Textfeld soll beim Drücken des Buttons auf Konsole geschrieben werden.

10.1 Vorgehen bei Ereignisverarbeitung

im Quelltext: *import java.awt.event.*;*

1. Definition einer Listener-Klasse, die ein Interface (vom Typ Listener) implementiert

Implementierung eines Listeners

```
import java.awt.event.*;

class Listen implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println(jTextField.getText());
    }
}
```

Interface mit einer Methode
wird hier implementiert

⇒ geht so erstmal nicht, da jTextField private ist

2. Erzeugung eines Listener-Objekts dieser Klasse

3. Anhängen dieses Listener-Objekts an die Fensterelemente auf die Listener reagieren soll

Listenerobjekt erzeugen & anhängen

```
import java.awt.event.*;

public class BspFrame extends JFrame {
    ...
    BspFrame() {
        ...
        // alles bisherige

        // Erzeugen des Listener-Objekts:
        Listen listen = new Listen();

        // Anhängen an Fensterelement:
        jButton.addActionListener(listen);
        ...
    }
    ...
}
```

Hier: jButton

10.2 Grundlegendes Konzept der inneren Klassen

- in JDK 1.0 nur möglich, Klassen auf Paketebene zu definieren (wie bisher bei uns)
- in JDK 1.1 Ereignismodell eingeführt

⇒ viele Klassen, die nur Bedeutung im begrenzten Kontext haben (z.B. Listener)

Schön wäre es, wenn Listener direkt neben dem Fensterelement stehen würden.

- Klassen (hier: Listener) möglichst innerhalb anderer Klassen (hier: JFrames) implementieren
- werden lokale oder innere Klassen genannt

Prinzip:

1. innerhalb einer Klasse wird innere Klasse implementiert
2. Instanzierung der inneren Klasse muss innerhalb der äußeren Klasse durchgeführt werden

Sichtbarkeit:

Die innere Klasse kann auf Attribute und Methoden der Äußeren zugreifen und umgekehrt, sonst keiner.

innere Klasse

```

class Outer {
    String name;
    int number;
    public void createAndPrintInner(String iname) {
        // Erzeugung von Objekt der inneren Klasse
        Inner inner = new Inner();
        // Übergabe: Parameter -> private-Attribut von inner:
        inner.name = iname;
        System.out.println(inner.getQualifiedName());
    }

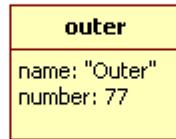
    class Inner {
        private String name;
        private String getQualifiedName() {
            return number + ":" + Outer.this.name + "."
                + name;
        }
    }
}

```

In *main*:

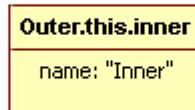
```
Outer outer = new Outer();          (1)  
outer.name = "Outer";            (2)  
outer.number = 77;              (3)  
outer.createAndPrintInner("Inner"); (4)
```

Anlegen von *outer*, (1) bis (3):



⇒ Aufruf von *outer.createAndPrintInner("Inner")*;

Anlegen von *inner*, (4):



⇒ Aufruf von *inner.getQualifiedName()*

Es gibt 3 unterschiedliche Arten des Zugriffs auf Attribute und Methoden:

1. Zugriff ohne Referenz: hier von innen nach außen suchen bis Attribut gefunden

- *number*: nur außen
 - *name*: innen und außen ⇒ Inneres wird genommen
2. Zugriff auf in innerer Klasse überdecktes Attribut der äußeren Klasse
 - *name* in äußerer Klasse: *Klassename.this.name*, hier: *Outer.this.name*
 3. Man kann *Klassename.this* auch allein verwenden, dann Referenz auf Objekt der äußeren Klasse

10.3 Verwendung von inneren Klassen bei Listener

Listener-Implement. als innere Klasse

```
import java.awt.event.*;

public class BspFrame extends JFrame {
    ...
    class Listen implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            System.out.println(jTextFeld.getText());
        }
    }
    BspFrame() {
        ...
        Listen listen = new Listen();
        jButton.addActionListener(listen);
        ...
    }
    ...
}           schöner wäre: Erzeugung und Definition
            des Listeners im Parameter
```

10.4 Grundlegendes Konzept der anonymen Klassen

Listener-Implement. als anonyme Klasse

```
import java.awt.event.*;  
  
public class BspFrame extends JFrame {  
    ...  
  
    BspFrame() {  
        ...  
        jButton.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                System.out.println(jTextField.getText());  
            }  
        });  
        ...  
    }  
    ...  
}
```

⇒ haben schon anonyme Objekte gesehen, z.B. `jButton.addActionListener(new Listen());`

- Anonyme Klassen: implementieren in Parameter und direkt ein Objekt (anonym) davon erzeugen
- muss implementiertes Interface (hier: ActionListener) oder ererbte (abstrakte) Klasse hinter `new` schreiben
- Einwegklasse: kann nur ein Objekt dieser Klasse erzeugen!

Man müsste sich vorstellen, dass dort steht: `new ... implements ActionListener ...`

Interface-Implement. als anonyme Klasse

```
public static void main(String[] args) {
    Test test = new Test();
    Benutzer ben = new Benutzer("Heinz", "Passwort");
    test.meth(ben, new BenutzerVerwaltung() {
        public void benutzerEintragen(Benutzer ben) {
            System.out.println("in benEin!");
        }
        public boolean benutzerOk(Benutzer ben) {
            return true;
        }
    });
}
```

Hier kann: Klasse oder Interface stehen

```
class Test {

    public void meth(Benutzer ben, BenutzerVerwaltung bv) {
        bv.benutzerEintragen(ben);
    }
}
```

⇒ Ausgabe: "in benEin!"

- es wird *meth* ausgeführt
- anonyme Implementation von *bv.benutzerEintragen(ben)* liefert Ausgabe "in benEin!"

10.5 Weitere wichtige Methoden von Fensterelementen

JTextField

- *getText()* liefert Inhalt
- *setEnabled(boolean enabled)* vergibt Schreibrechte

JPasswordField

- *getPassword()* liefert Inhalt

- `copyValueOf` Klassenmethode von `String` zur Trafo nach String verwenden
- `setEnabled(boolean enabled)` vergibt Schreibrechte

JCheckBox

- `setSelected(boolean b)` setzt Box
- `isSelected()` Abfrage, ob Box gesetzt

10.6 Zusammenfassung verschiedener Ereignistypen

Action-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	ActionEvent
Listener-Interface	ActionListener
Registrierungsmethode	addActionListener
Mögliche Ereignisquellen	Button, List, MenuItem, TextField
Ereignismethode	Bedeutung
actionPerformed	Eine Aktion wurde ausgelöst. z.B.: Mouse gedrückt, durch Return angeklickt, ...

Mouse-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	MouseEvent
Listener-Interface	MouseListener
Registrierungsmethode	addMouseListener
Mögliche Ereignisquellen	Component
Ereignismethoden	Bedeutung
mouseClicked	Maustaste gedrückt und wieder losgelassen
mouseEntered	Mauszeiger betritt die Komponente
mouseExited	Mauszeiger verlässt die Komponente
mousePressed	Maustaste wurde gedrückt
mouseReleased	Maustaste wurde losgelassen

MouseMotion-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	MouseEvent
Listener-Interface	MouseMotionListener
Registrierungsmethode	addMouseMotionListener
Mögliche Ereignisquellen	Component
Ereignismethoden	Bedeutung
mouseDragged	Maus wurde bei gedrückter Taste bewegt
mouseMoved	Maus wurde bewegt, ohne daß eine Taste gedrückt

Key-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	KeyEvent
Listener-Interface	KeyListener
Registrierungsmethode	addKeyListener
Mögliche Ereignisquellen	Component
Ereignismethoden	Bedeutung
keyPressed	Eine Taste wurde gedrückt.
keyReleased	Eine Taste wurde losgelassen.
keyTyped	Eine Taste wurde gedrückt und wieder losgelassen.

Focus-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	FocusEvent
Listener-Interface	FocusListener
Registrierungsmethode	addFocusListener
Mögliche Ereignisquellen	Component
Ereignismethoden	Bedeutung
focusLost	Eine Komponente verliert den Focus.
focusGained	Eine Komponente erhält den Focus.

Component-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	ComponentEvent
Listener-Interface	ComponentListener
Registrierungsmethode	addComponentListener
Mögliche Ereignisquellen	Component
Ereignismethoden	Bedeutung
componentHidden	Eine Komponente wurde unsichtbar.
componentMoved	Eine Komponente wurde verschoben.
componentResized	Größe einer Komponente hat sich geändert.
componentShown	Eine Komponente wurde sichtbar.

Container-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	ContainerEvent
Listener-Interface	ContainerListener
Registrierungsmethode	addContainerListener
Mögliche Ereignisquellen	Container
Ereignismethoden	Bedeutung
componentAdded	Eine Komponente wurde hinzugefügt.
componentRemoved	Eine Komponente wurde entfernt.

Window-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	WindowEvent
Listener-Interface	WindowListener
Registrierungsmethode	addWindowListener
Mögliche Ereignisquellen	Dialog, Frame
Ereignismethoden	Bedeutung
windowActivated	Das Fenster wurde aktiviert.
windowClosed	Das Fenster wurde geschlossen.
windowClosing	Das Fenster wird geschlossen.
windowDeactivated	Das Fenster wurde deaktiviert.
windowDeiconified	Das Fenster wurde wiederhergestellt.
windowIconified	Fenster wurde auf Symbolgröße verkleinert.
windowOpened	Das Fenster wurde geöffnet.

Adjustment-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	AdjustmentEvent
Listener-Interface	AdjustmentListener
Registrierungsmethode	addAdjustmentListener
Mögliche Ereignisquellen	Button, List, MenuItem, TextField
Ereignismethode	Bedeutung
adjustmentValueChanged	Der Wert wurde verändert.

Item-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	ItemEvent
Listener-Interface	ItemListener
Registrierungsmethode	addItemListener
Mögliche Ereignisquellen	Checkbox, Choice, List, CheckboxMenuItem
Ereignismethode	Bedeutung
itemStateChanged	Der Zustand hat sich verändert.

Text-Events

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	TextEvent
Listener-Interface	TextListener
Registrierungsmethode	addTextListener
Mögliche Ereignisquellen	TextField, TextArea
Ereignismethode	Bedeutung
itemValueChanged	Der Text wurde verändert.

10.7 Ein Beispiel für MouseEvent

Mouse-Listener als anonyme Klasse

```
import java.awt.event.*;

public class BspFrame extends JFrame {
    ...
    BspFrame() {
        ...
        jButton.addMouseListener(new MouseListener() {
            ...
            public void mouseClicked(MouseEvent e) {
                System.out.println(jTextField.getText());
            }
        });
        ...
    }
    ...
}
```

Adapter

Anonyme Klasse sollte abstrakt deklariert werden; es definiert

- Event soll nur bei Mausklick schalten
- Ersetze actionPerformed durch mouseClicked
- ⇒ reicht nicht, da MouseListener noch 4 andere Methoden zur Verfügung stellt
- müssten immer alle Methoden implementieren, obwohl nur 1 interessant!

Dazu:

10.8 Grundlegendes Konzept der Adapter-Klassen

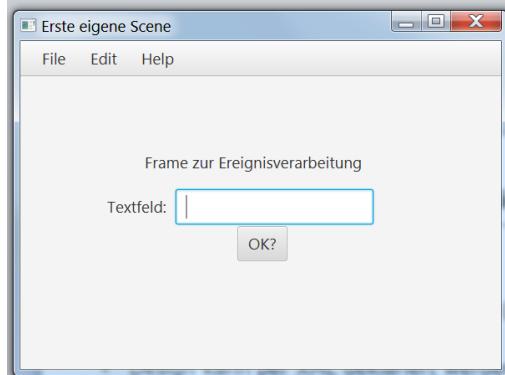
Def.: Eine Adapterklasse in Java ist eine Klasse, die ein vorgegebenes Interface mit leeren Rümpfen oder Default-Rückgaben implementiert

Zu jedem Listener gibt es eine passende Adapter-Klasse!

- nicht `new ..Listener()`, sondern `new ..Adapter()` und nur relevante Methoden implementieren
- kein Problem in Vererbungshierarchie, da Einwegklasse

10.9 JavaFX

Eine Beispiel-Scene



Event Handling bei JavaFX

Wieder 2 Möglichkeiten:

- direkt über Java: analog zu Swing
- über XML-Konfiguration

10.9.1 1. Möglichkeit: analog zu Swing

Event-Handler per anonyme Klasse

Auszug aus dem Konstruktor

```

1 button.addEventHandler(ActionEvent.ACTION, >
2     new EventHandler<ActionEvent>() {
3         @Override
4         public void handle(ActionEvent actionEvent) {
5             System.out.println("Testing");
6         }
7     });

```

analog zu Swing

10.9.2 2. Möglichkeit: mit fxml-Datei

braucht die Anwendungsdatei

Anwendung zur Beispiel-Scene

```
public class AnwendungFXML extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    public void start(Stage primaryStage) {
        Parent root;
        try {
            root=FXMLLoader.load(getClass().getResource("sample_gui.fxml"));
            primaryStage.setTitle("Erste eigene Scene");
            primaryStage.setScene(new Scene(root));
            primaryStage.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Ereignisverarbeitung 24. November 2017 | 23

implementiert die Event-Verarbeitung im Controller

bindet diese in fxml-Datei beim Scene-Element ein:

Implementierung eines Event-Handlers zur Beispiel-Scene

Controller:

```
public class Controller {
    @FXML TextField textField;
    @FXML Button button;
    @FXML public void handleButtonPush(Event event) {
        System.out.println(textField.getText());
    }
}
```

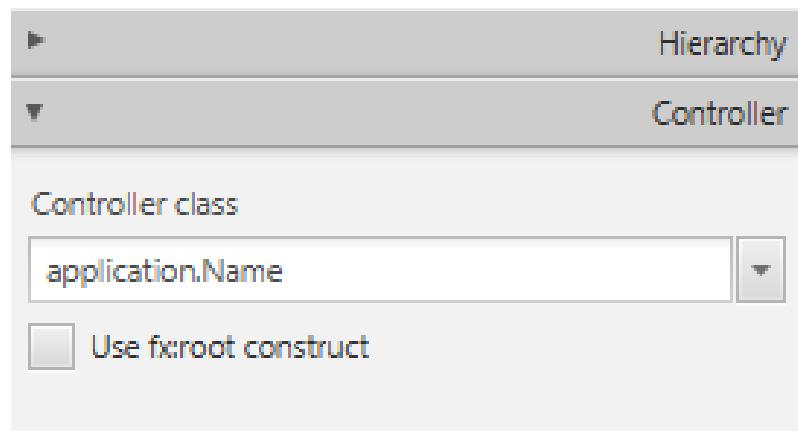
FXML-Auszug:

```
<VBox ... fx:controller="main.ui.Controller"> // außen
<TextField fx:id="textField" layoutX="186.0" layoutY="135.0" />
<Label layoutX="149.0" layoutY="89.0" prefHeight="28.0"
prefWidth="342.0" text="Frame zur Ereignisverarbeitung" />
<Label layoutX="104.0" layoutY="142.0" text="Textfeld:" />
<Button fx:id="button" layoutX="260.0" layoutY="179.0"
mnemonicParsing="false" onAction="#handleButtonPush" text="OK?" />
```

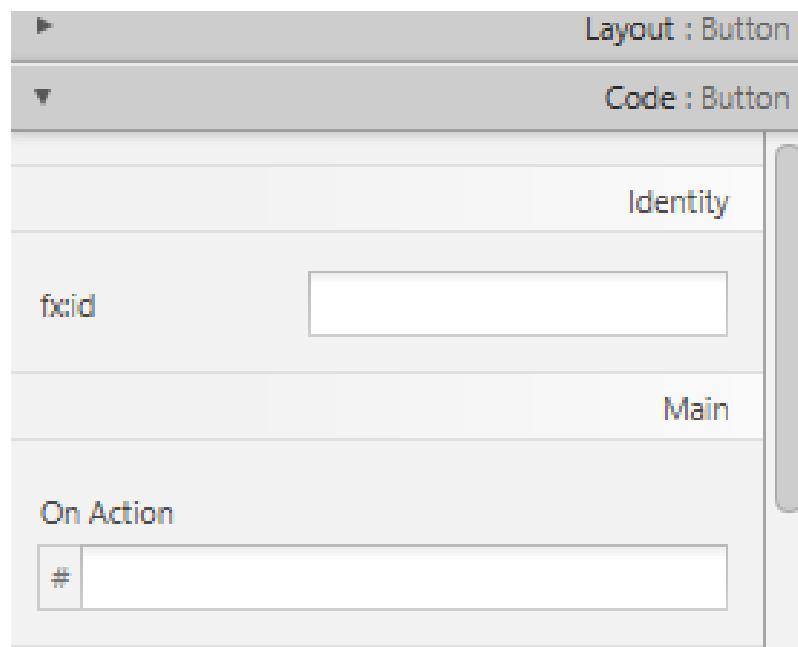
© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Ereignisverarbeitung 24. November 2017 | 24

10.9.3 Hinweise zu Ereignissen im SceneBuilder

Im SceneBuilder sind die fx:ids auswählbar, wenn man den Controller zuerst geschrieben und angegeben hat (z.B. @FXML TextField textField;). Der Controller wird im SceneBuilder auf der linken Seite im untersten Abschnitt Controller angegeben. Hier muss das richtige Package angegeben werden.

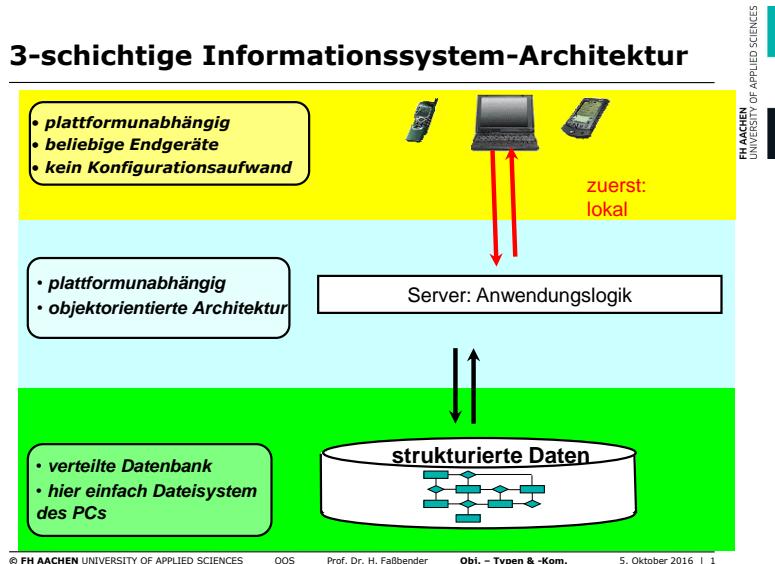


Soll nun eine fx:id einem fxml-Objekt zugeordnet werden, muss im SceneBuilder auf der rechten Seite im Abschnitt Code die fx:id für das ausgewählte Objekt zugeordnet werden. Für Event Handling muss der Name der zu verarbeitenden Event-Funktion im On Action Feld eingetragen werden.



11 Objekttypen und Objektkommunikation

Haben 3 Frames: BenutzerVerwaltung, BenutzerVerwaltungAdmin und Benutzer => wie zusammenbringen!



OO-System: System interagierender Objekte

z.B. Frames müssen mit *BenutzerVerwaltungAdmin* interagieren

11.1 Objekttypen

Objekte: Zusammenfassung von Daten und Funktionalität

Def.: (Datenobjekt)

- Ein Objekt, das lediglich zur Strukturierung von Daten deklariert wird und ausschließlich Standardmethoden und/oder getter-/setter-Methoden enthält, heißt Datenobjekt.
- ⇒ z.B. Klasse *Benutzer*

Def.: (Serverobjekt)

- Ein Objekt, das hauptsächlich dazu deklariert wird, anderen Objekten (womöglich über eine Schnittstelle) Dienste anzubieten, heißt Serverobjekt.
- ⇒ z.B. Klasse *BenutzerVerwaltungAdmin*

In modernen Informationssystemen ist die mittlere Schicht der Architektur zerlegt in:

- Datenschicht: OO-Abbildung der Datenbank als Datenobjekte
- Anwendungsschicht: Serverobjekte für Client

Def.: (GUI-Objekt)

- Ein Objekt, das der Darstellung auf der Systemnutzerschnittstelle dient, heisst GUI-Objekt.

Bisher Objekte immer nur in *main* erzeugt:

```
lf = new LoginFrame(); ⇒ ben ⇒ bv.benutzerOk(ben);
```

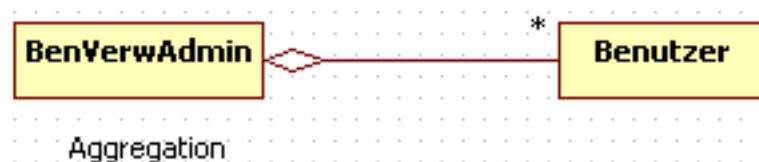
Steuerung des Ablaufs ist spätestens bei GUI-Objekten nicht mehr in *main* möglich!

Def.: (Steuerungsobjekt)

- Ein Objekt, das die Zusammenarbeit von verschiedenen Objekten steuert, heisst Steuerungsobjekt/Controller.

11.2 Objektkommunikation

Bisher:

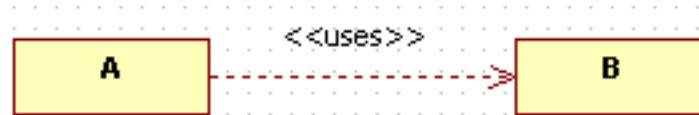


Interaktion zwischen Objekten geht über Methodenaufrufe!

Abhängigkeitsbeziehung (nutzt-Beziehung):

Objekte (Client) nutzen Dienste von anderen Objekten (Server) und bieten (dann selbst Server) anderen Objekten (die dann Clients sind) Dienste an! Hierzu sind Methoden notwendig.

In UML: A (Client) nutzt Dienst eines Objektes von B (Server)



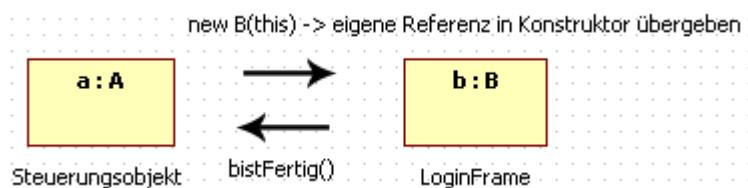
In Java: in A Methodenaufruf einer Methode von B

Hierzu muss folgendes bekannt sein:

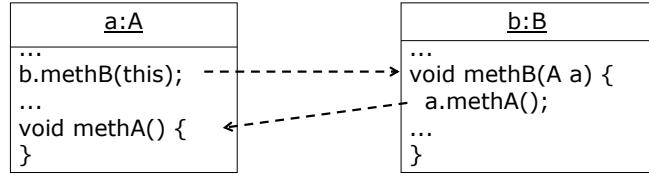
1. Name des Dienstes (Methode), z.B. *benutzerOk* und Signatur (Parameterliste) und Rückgabetyp (z.B. *boolean*)
2. Objekt-ID (Referenz) auf das Server-Objekt (z.B. für *BenutzerVerwaltungAdmin*: *bv*)

Man kann Objektreferenzen wie folgt erhalten:

1. Objekt selbst erzeugen (bisher immer) ⇒ Problem: z.B. *bv* wird zentral erzeugt
2. Referenz auf Server-Objekt kann bei Erzeugung des Clientobjektes im Parameter des Konstruktors übergeben werden, z.B. *ObjectInputStream(InputStream is)*
3. Man kann bei Nutzung eines Dienstes (Methodenaufruf) dem Serverobjekt Referenzen im Parameter übergeben, damit Serverobjekt Dienste der referenzierten Objekte nutzen kann ⇒ Insbesondere kann man dem Serverobjekt seine eigene Referenz übergeben, damit man selbst zum Server wird (s.o. Abhängigkeitsbeziehung)



Call-Back



Wie kommt a an Referenz von b?
Wo werden die beiden Objekte erzeugt?

Def.: (Call-Back)

- Die Nutzung eines Dienstes des Clientobjektes `methA` (Methode von A) durch das Serverobjekt heißt call-back.
- in Java: in `main` Objekt a von A und b von B erzeugen, dann soll a eine Methode von b aufrufen, die eine Methode von a aufruft

Call-Back Beispiel

```

public class A {
    // Konstruktor:
    A(B bRef) {
        this.bRef = bRef; ← Übergabe der Ref. zu b an a.
    }
    // Methode, die von außen aufgerufen wird:
    void rufeBAuf() {
        System.out.println("Bin in A in rufeBAuf!");
        this.bRef.ichBin(this); ← Übergabe der Ref. zu a an b.
    }
    void zurückVonB() {
        System.out.println("Bin in A in zurückVonB!");
    }
    // Merken der Referenz von B:
    private B bRef;
}
  
```

Call-Back Beispiel 2

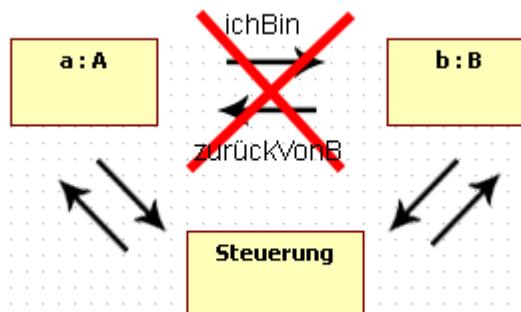
```

public class B {
    void ichBin(A a) {
        System.out.println("Bin in B bei ichBin!");
        a.zurückVonB(); ← Call-Back
    }
}

Orb
public class Steuerung {
    public static void main(String[] args) {
        B b = new B();
        A a = new A(b); ← Übergabe der Ref. zu b an a.
        a.rufeBAuf();      Damit verliert Steuerung die Kontrolle!
    }
}

```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Obj. - Typen & -Kom. 5. Oktober 2016 | 4



Orchestrierung: Steuerung ist Dirigent!

Probleme: Steuerung verliert Kontrolle ⇒ unerwünscht, alles soll über Steuerung gehen

⇒ nur die Referenz von Steuerung darf bekannt gegeben werden

Steuerung dient dann neben der Steuerung des Ablaufs auch als Objektanfragenvermittler (ORB - Object Request Broker)

ORB-Beispiel

```
public class A {
    A(Orb orbRef) {
        this.orbRef = orbRef;
    }

    void rufeBAuf() {
        System.out.println("Bin in A in rufeBAuf!");
        this.orbRef.ichBin(this); ← Übergabe der Ref. von a an orb.
    }

    void zurückVonB() {
        System.out.println("Bin in A in zurückVonB!");
    }

    private Orb orbRef;
}
```

ORB-Beispiel 2

```
public class B {
    void ichBin(Orb orb) {
        System.out.println("Bin in B bei ichBin!");
        orb.zurückVonB(); ← Call-Back geht über orb
    }
}

public class Orb {
    public static void main(String[] args) {
        B b = new B();
        A a = new A(b); ← Hier muss this rein
        a.rufeBAuf();
    } Geht nicht, weil von Orb kein Objekt existiert!
}
```

ORB-Beispiel 3

```
public class Orb {

    Orb() {
        b = new B();
        a = new A(this);
        a.rufeBAuf(); ← Geht schief, weil Orb nicht
                        ichBin(this) implementiert
    }

    private B b;
    private A a;

    public static void main(String[] args) {
        Orb orb = new Orb();
    }
}
```

Da Orb Vermittler, muss er alle Dienste von A und B anbieten, die von anderen genutzt werden!



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Obj. - Typen & -Kom. 5. Oktober 2016 | 7

Orb-Implementierung fremder Dienste

```
public class Orb {
    ...
    void ichBin(A a) {
        System.out.println("Bin in Orb bei ichBin!");
        b.ichBin(this);
    }

    void zurückVonB() {
        System.out.println("Bin in Orb bei zurückVonB!");
        a.zurückVonB();
    }

    private B b;
    private A a; ← Speicherung der Referenzen, da sie in
                    Konstruktor erzeugt und nach Beendigung
                    des Konstruktors weg wären.
    ...
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Obj. - Typen & -Kom. 5. Oktober 2016 | 8

Was muss sichergestellt werden?

1. einzige bekanntgegebene Objektreferenz ist die des Orbs
2. da Orb seine Referenz bekanntgeben muss, muss Objekt von Orb vorhanden sein ⇒ nicht in *main*-Methode, sondern in Konstruktor operieren!
3. Implementierung der fremden Dienste in Orb durch Delegation

Def.: (Delegation)

- Die Implementierung eines Dienstes durch einfaches Weiterleiten des Aufrufs an einen Dienstanbieter und die Rückgabe des Ergebnisses an den Client heisst Delegation.
- Weil Orb hier nur die Steuerung übernimmt, ab jetzt nur noch Steuerung genannt ⇒ nennen ihn erst wieder Orb, wenn Remote-Zugriff implementiert ist

Steuerungsobjekte steuern den Programmfluß ⇒ Szenario

Def.: (Szenario)

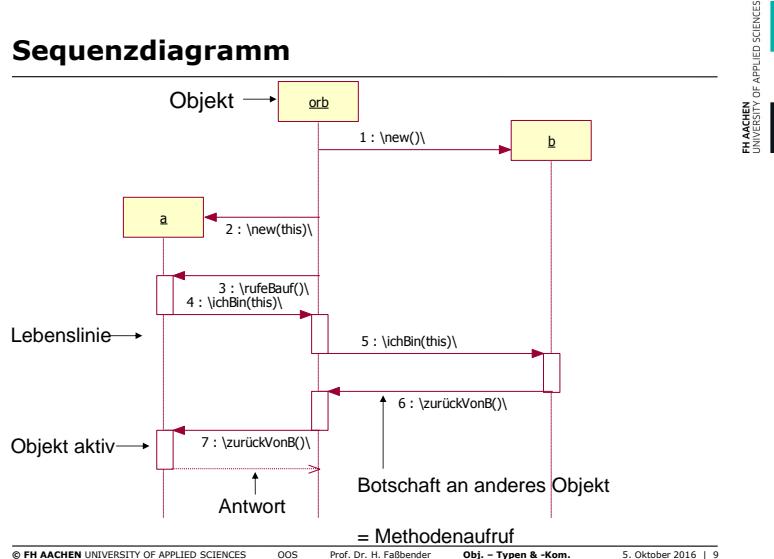
- Eine einzelne Ausprägung eines Anwendungsfalles heisst Szenario

11.3 Beschreibung eines Szenarios

1. verbal: siehe Beschreibung Praktikum 4
2. grafisch (schöner): hierzu 2. Typ von Diagrammen in UML: Sequenzdiagramm

11.4 Sequenzdiagramm

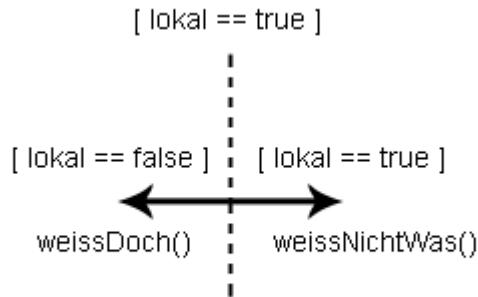
- in Klassendiagramm: statische Struktur
- hier: dynamisches Verhalten



- Kasten an Lebenslinie bedeutet: Objekt aktiv (hat gerade die Kontrolle)
- findet oft noch nicht vorhandene Klassen und Methoden
- eigentlich keine Fallunterscheidung erlaubt (pro Fall ein Diagramm)

- manchmal sollte man doch Fälle berücksichtigen (durch Bedingung)

Fallunterscheidung:

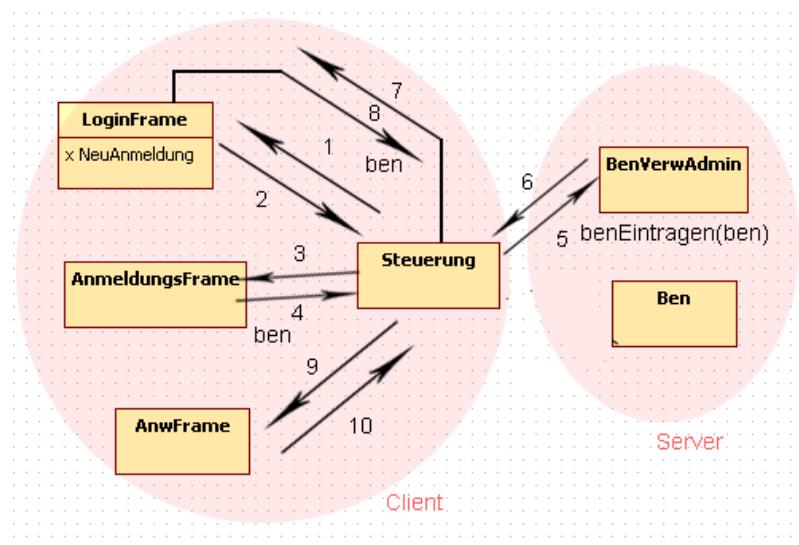
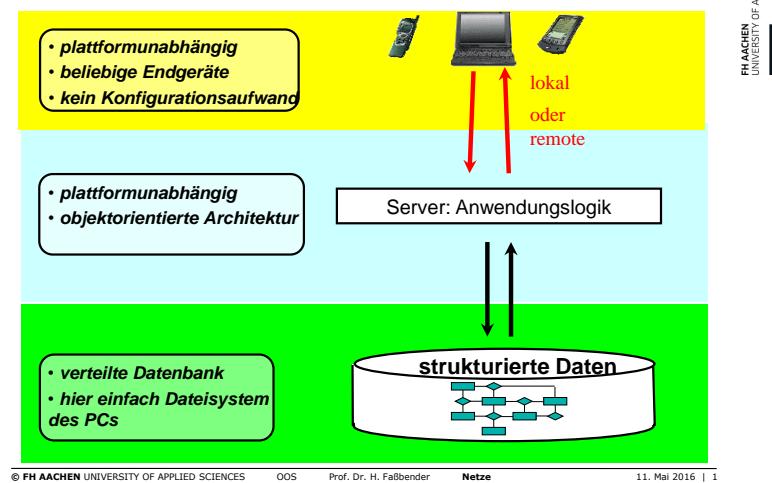


11.5 Rückgabe der Kontrolle an Steuerungsobjekt

- z.B. Button auf Frame gedrückt \Rightarrow Steuerungsobjekt soll Überprüfung des Benutzers durchführen
- normalerweise Rückgabe der Kontrolle bei Ende des Methodenaufrufs \Rightarrow geht bei Frames nicht
- deshalb: Stelle in Steuerung call-back-Methoden zur Verfügung
- hier: ichBin, zurückVon

12 Netze

3-schichtige Informationssystem-Architektur



Zwischen Client und Server befindet sich das Netz. Es werden Benutzer und Exceptions verschickt.

Pakete:

- Client-Dateien
- Server-Dateien
- mit Klassen, die auf beiden Seiten gebraucht werden (hier: *Benutzer* und *Exceptions* ⇒ in beide Pakete importieren)

12.1 Client und Server verteilen

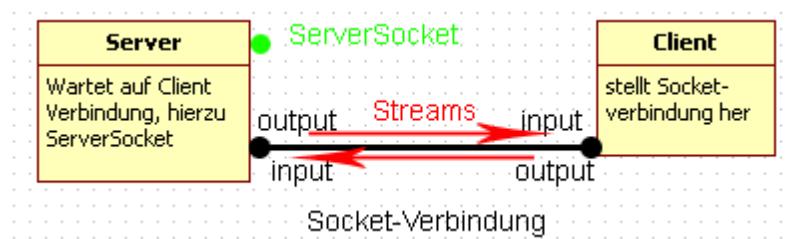
- heutzutage üblicherweise schon programmiert
- bei reinen Java-Anwendungen: Remote Method Invocation (RMI)
- bei gemischten Umgebungen: CORBA (IDL) oder WebServices (WSDL)
- IDL, WSDL zur standardisierten Schnittstellendefinition



Übertragungsprotokolle bei

- CORBA: IIOP
- WebServices: SOAP (Sprache XML) ⇒ später
- erstmal alles von Hand programmieren (nicht mehr ganz so schlimm wie früher, wegen ObjectStreams)

12.2 Stream-Sockets zum Datentransfer im Netz



Def.: (Socket)

- Ein Socket dient zur Abstraktion und ist ein Verbindungspunkt in einem TCP/IP-Netzwerk

Wie kann man eine Verbindung aufbauen?

12.3 Die Klasse Socket

einige Konstr. & Methoden von Socket

```
Socket( String host, int port )
    throws IOException
Socket( InetAddress address , int port)
    throws IOException
void flush()
    throws IOException
void close()
    throws IOException
String getHostName()
InetAddress getLocalHost()
    throws UnknownHostException
InputStream getInputStream()
    throws IOException
OutputStream getOutputStream()
    throws IOException
```

Importieren:

- `java.net.*` (Sockets)
- `java.io.*` (Streams)
- `Socket` auf beiden Seiten gebraucht, auf Server zusätzlich `ServerSocket`

Erläuterungen zu den Methoden von Socket:

- **Konstruktoren:** erzeugen Stream-Socket zum Server mit Host host, der am Port port einen `ServerSocket` laufen hat
- **flush():** schickt noch gepufferte Daten rüber ⇒ mindestens nach letztem gesendetem Datum
- **close():** schließt Socket
- **getHostName():** liefert Hostname, wo Socket läuft ⇒ wenn Verbindung vorhanden, dann 2 Streams vorhanden ⇒ nicht selbst erzeugen, sondern mit `getInputStream()` und `getOutputStream()` deren Referenzen holen

12.4 Die Klasse ServerSocket

einige Konstr. & Meth. von ServerSocket

```
ServerSocket( int port )
    throws IOException

Socket accept( )
    throws IOException

void setSoTimeout(int timeout)
    throws SocketException

public void close()
    throws IOException
```

- **Konstruktor:** richtet einen *ServerSocket* ein, der am Port port horcht ⇒ dient nur zum vom Client initiierten Socketaufbau
- **Socket accept():** eine wartende Verbindungsanfrage wird angenommen ⇒ Konversation läuft nicht über *ServerSocket*, sondern über 2 automatisch zugewiesene Sockets, *ServerSocket* nur für Annahme der Verbindung => *accept()* läuft meistens in Endlosschleife, für jede Verbindung dann eigenen Thread aufmachen
- **setSoTimeout(int timeout):** stoppt Warten auf eine Verbindung nach timeout Millisekunden
- **close():** beendet *ServerSocket*

Mögliche Szenario:

1. Server wartet an *ServerSocket* auf Verbindungsauftbau durch Client
2. Client stellt Socket-Verbindung mit Server her
3. Client sendet Daten über seinen Output-Stream
4. Server empfängt Daten über seinen Input-Stream
5. Server verarbeitet Daten
6. Server sendet Daten über seinen Output-Stream
7. Client empfängt Daten über seinen Input-Stream
8. Gehe zu 3. oder schliesse Verbindung
9. Server kann auf eine neue Verbindung warten oder nicht

12.5 Beispiel: Multiplikationsserver

- Client sendet 2 Bytes
- Server bildet Produkt und sendet Ergebnis zurück
- hat noch nichts mit entfernten Methodenaufrufen zu tun

Multiplikationsserver

```
import java.net.*;
import java.io.*;

public class Server {
    Server() throws IOException {
        ServerSocket server = new ServerSocket( 4711 );
        while ( true ) {
            Socket client = server.accept();
            InputStream in = client.getInputStream();
            OutputStream out = client.getOutputStream();
            int multiplikator1 = in.read();
            int multiplikator2 = in.read();
            int resultat = multiplikator1 * multiplikator2;
            out.write(resultat);
        }
    }
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES

OOS

Prof. Dr. H. Faßbender

Netze

11. Mai 2016 | 4



Multiplikationsserver: main-Methode

```
public static void main (String[] args) {
    try {
        Server server = new Server();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

}

}

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES

OOS

Prof. Dr. H. Faßbender

Netze

11. Mai 2016 | 5



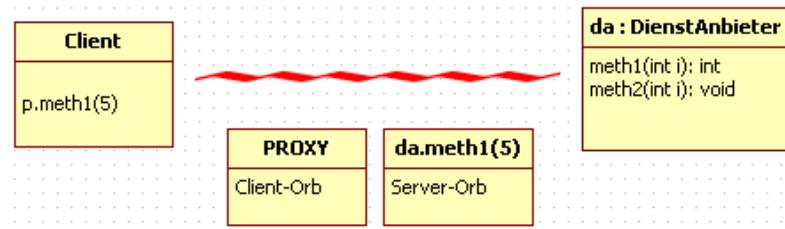
Multiplikationsclient

```
import java.net.*;
import java.io.*;
public class Client {
    Client() throws IOException {
        Socket server = new Socket ("localhost", 4711 );
        InputStream in = server.getInputStream();
        OutputStream out = server.getOutputStream();
        out.write( 4 );
        out.write( 9 );
        int result = in.read();
        System.out.println( result );
        server.close();
    }
}
```

Multiplikationsclient: main-Methode

```
public static void main (String[] args) {
    try {
        Client client = new Client();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Jetzt: Sockets zur Implementierung entfernter Methodenaufrufe verwenden!



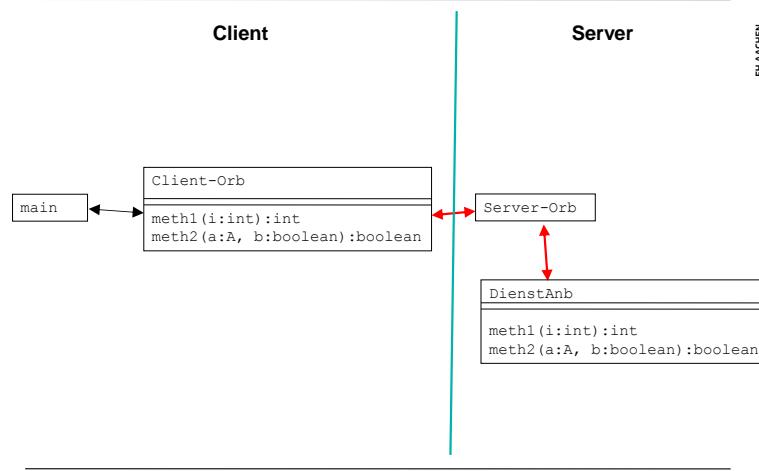
Ein Dienstanbieter

```
public class MalNehmer {  
  
    int mal2 (int in) {  
        return (2*in);  
    }  
  
    int mal3 (int in) {  
        return (3*in);  
    }  
  
}
```

Steuerung mit lokaler Server-Komm.

```
public class SteuerungLokal {  
  
    public SteuerungLokal() {  
  
        MalNehmer malNehmer = new MalNehmer();  
  
        int eingabe = 7;  
        System.out.println(malNehmer.mal2(eingabe));  
        System.out.println(malNehmer.mal3(eingabe));  
    }  
  
    public static void main(String[] args) {  
        SteuerungLokal sL = new SteuerungLokal();  
    }  
}
```

Architektur (mit Methoden)



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Netze 11. Mai 2016 | 10

Steuerung mit remote Server-Komm.

```

public class SteuerungRemote {
    public SteuerungRemote() {
        ClientOrb clientOrb = new ClientOrb();
        int eingabe = 7;
        System.out.println(clientOrb.mal2(eingabe));
        System.out.println(clientOrb.mal3(eingabe));
    }

    public static void main(String[] args) {
        SteuerungRemote sR = new SteuerungRemote();
    }
}

```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Netze 11. Mai 2016 | 11

ClientOrb

```
public class ClientOrb {
    public ClientOrb() throws IOException {
    }
    public int mal2(int in) throws IOException {
        // Verbindung und Streams initialisieren:
        Socket server = new Socket ("localhost", 4711 );
        ObjectOutputStream out =
            new ObjectOutputStream(server.getOutputStream());
        ObjectInputStream is =
            new ObjectInputStream(server.getInputStream());
        // Methodenkennung und Parameter senden:
        out.writeInt( 1 );
        out.writeInt( in );
        out.flush();
        // Ergebnis lesen, zurückgeben, Verbindung schließen:
        int result = is.readInt();
        server.close();                                mal3(int in):
        return (result);                            analog mit Methodenkennung 2
    }
}
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Netze 11. Mai 2016 | 12

ServerOrb (1)

```
public class ServerOrb {
    // ganze Verarbeitung im Konstruktor:
    ServerOrb() throws IOException {
    }
    // Erzeugung des Dienstanbieters, sonst häufig durch
    // main-Programm in extra Klasse Server:
    MalNehmer malNehmer = new MalNehmer();
    // Anlegen eines ServerSockets:
    ServerSocket server = new ServerSocket( 4711 );
    // Warten auf Verbindungsaufnahme durch Client:
    while ( true ) {
        Socket client = server.accept();
        // Verbindung annehmen und korresp. Streams erzeugen:
        ObjectInputStream in =
            new ObjectInputStream(client.getInputStream());
        ObjectOutputStream out =
            new ObjectOutputStream(client.getOutputStream());
```

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Netze 11. Mai 2016 | 13

Wichtig:

- Streams in umgedrehter Reihenfolge als bei ClientOrb
- immer zuerst den einen, dann den anderen erzeugen

ServerOrb (2)

```
// Lesen der Methodenkodierung:  
    int methode = in.readInt();  
// Lesen des Parameters:  
    int par = in.readInt();  
// Methodenimplementierungen durch Delegation an  
// Dienstanbieterobjekt:  
    if (methode == 1) {  
        out.writeInt(malNehmer.mal2(par));  
    }  
    if (methode == 2) {  
        out.writeInt(malNehmer.mal3(par));  
    }  
// Sicherstellen, dass Ergebnis transportiert wird:  
    out.flush();  
}  
}  
}
```

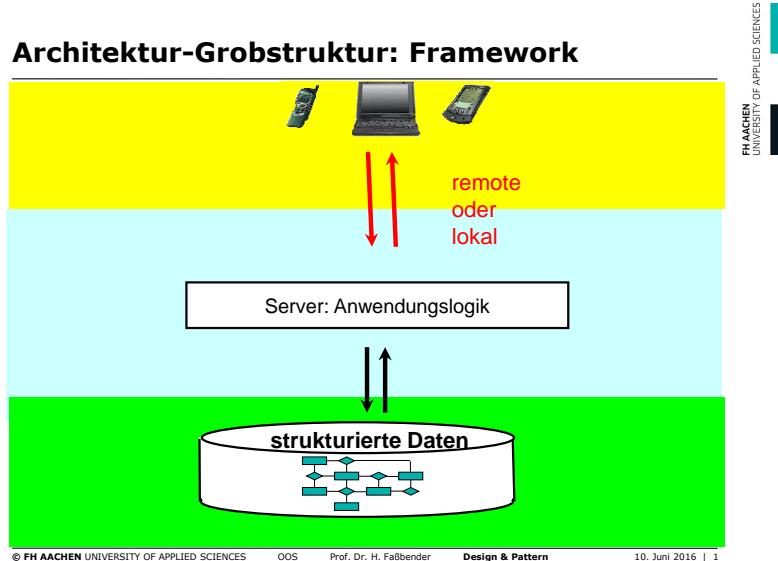
© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Netze 11. Mai 2016 | 14

Setzen zwischen Client und Server:

- Client-Orb: gleiche Schnittstelle wie Server (Dienstanbieter), implementiert sie aber nicht selbst, sondern stellt Socket-Verbindung zu Server-Orb her und schickt Methodenkennung und Parameter an Server-Orb (Delegation) ⇒ dann liest er Ergebnis und gibt dieses an den Client zurück

13 Design und Design-Pattern

13.1 Grob- und Fein-Design



Es wird zwischen Grob- und Fein-Design unterschieden.

Grob-Design:

- verschiedene Schichten
- Kommunikation zwischen den Schichten
- nichts über Implementierung innerhalb der Schichten ausgesagt

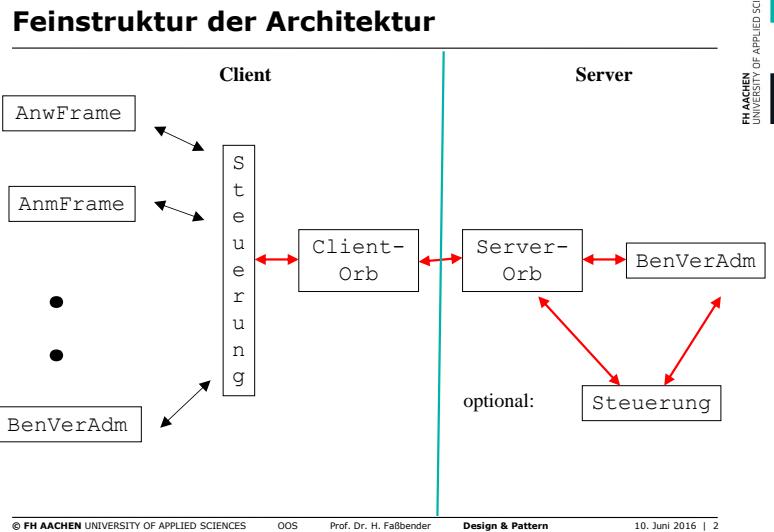
Fein-Design:

- auf DB-Ebene: welche Tabellen, Assoziationen, Schlüssel, ..
- auf GUI-Ebene: welche Fenster, wie sehen die aus?

Anwendungslogik:

- welche Klassen
- welche Methoden

- Beziehungen zwischen Klassen
- welche Attribute
- Kommunikation zwischen Objekten
- ⇒ per Klassen- / Sequenzdiagrammen



13.2 Vorgehensweise

- bisher: alles vorgegeben
- wie kommt man an obige Punkte?
- Ausgangspunkt: Anforderungsbeschreibung

Klassen:

- Substantive sind Kandidaten für Klassen
- Zum Beispiel: *BenutzerVerwaltungAdmin*, *BenutzerVerwaltung*, *Benutzer*
- ⇒ ins Klassendiagramm übernehmen

Beziehungen zwischen Klassen:

Beziehungen zwischen Klassen findet man durch Relationen zwischen Substantiven im Text:

- ist_ein ⇒ Vererbung
- besteht_aus ⇒ Aggregation

- nutzt \Rightarrow Assoziation

Methoden:

- Verben sind Kandidaten für Methoden
- Zum Beispiel: *benutzerEintragen*, *benutzerLöschen*, *benutzerOk*, *dbInitialisieren*
- Call-Back-Methoden: Abläufe in Sequenzdiagramm

Fragen zur Identifikation von Methoden:

- Serviceleistungen \Rightarrow *benutzerOk*
- Zustandsübergänge \Rightarrow Call-Back-Methoden
- Zufüge-/Entfernungs-Operationen \Rightarrow *benutzerLöschen*, *benutzerEintragen*
- Auskünfte \Rightarrow getter-Methoden
- Attributänderungen \Rightarrow setter-Methoden

Problem: Spezifikation von Methoden:

Methoden-Spezifikationsstruktur

- **Schnittstelle** (Name, Parametertypen, Rückgabetyp)
- **Vorbedingung** (vor Ausführung)
- **Nachbedingung** (nach Ausführung)
- **Invariante** (während Ausführung)
- **Semantik** (Beschreibung der Aufgabe und Bedeutung)
- **für Argumente Spezifikation von:**
 - Typprüfungen
 - Wertprüfungen

- verhindern Missverständnisse zwischen Anbieter und Nutzer
- verhindern Missverständnisse zwischen Designer und Programmierer
- dienen als Test-Grundlage
- hierzu: Erweiterung der UML: Object Constraint Language (OCL)

Fakultätsfunktion

```

int fak(int n) {
    int out = 1;
    for (int i=1; i<=n; i++) {
        out *= i;
    }
    return(out);
}

Vorbedingung: n>=0
Nachbedingung: out = n!
Invarianten: out = i!
Semantik: Fakultätsfunktion
oder fak(0) = 1
fak(n) = n*fak(n-1); n>0

```

Attribute:

- sind auch meistens bei Substantiven zu finden

wann Klasse?	wann Attribut?
enthält Ops	enthält keine Ops
wenn zusammengesetzt	wenn nicht zusammengesetzt
hat Eigenleben	hat kein Eigenleben

z.B. Benutzergruppen: student, kommerziell

wenn Benutzergruppe Eigenleben, dann eigene Klasse, sonst nur in Klasse Benutzer ein Attribut Benutzergruppe aggregieren

Spezifikation von Attributen:

- Name
- Typ
- Initialwert, Einschränkung des Wertebereichs

13.3 Design-Regeln/-Heuristik

Es gibt keine eindeutig beste Lösung, aber einige generelle Regeln:

Design-Regeln/-Heuristiken

- möglichst **kohärente Operationen**
nur eine Aufgabe pro Operation
- **anstelle von Funktionsmodi separate Operationen**
Entscheidung über Semantik über Signatur
- **keine Nebeneffekte**
durch globale Variablen oder ähnliches
- **keine Semantikänderung** bei Überlagerung in Unterklassen
- möglichst **allgemeingültig** entwerfen
im Hinblick auf Schnittstellen, nicht auf Implementierung
- Verallgemeinerungen **bleiben abstrakt**
- **Maximiere innere Bindung von Klassen**
zusammengehörende Verantwortlichkeiten in Klasse bündeln
- **Minimiere äußere Bindung von Klassen**
möglichst kleine Schnittstellen

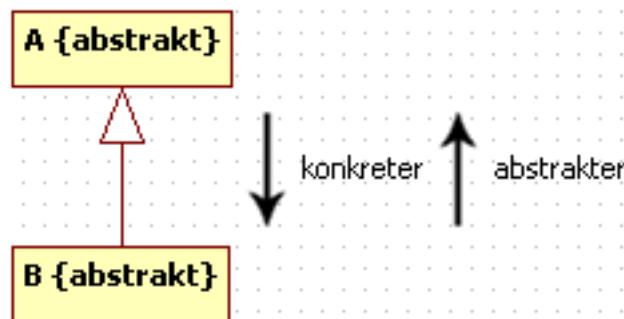
```
aritop(mode:int, op1:int, op2:int)

case 1: return(op1+op2)

case 2: return(op1-op2)
```

⇒ ist hochgradig nicht-kohärent, sowsas vermeiden!

Zu „Verallgemeinerungen bleiben abstrakt“:



⇒ wenn B abstrakt ist, dann muss auch A abstrakt sein

Design-Regeln/-Heuristiken (Forts.)

- pro Operation höchstens **eine Seite Code**
sonst besser Cobol verwenden
- **einheitliche und treffende Namen, Typen, Parameter**
auch Reihenfolge der Parameter wichtig
- möglichst **wenige switch / if-Anweisungen**
Indiz für imperatives Denken
- **Berücksichtigung von Extremwerten & Robustheit**
Minimum, Maximum, nil
- **keine künstlichen Grenzen**
dynamisches Verhalten implementieren
- **Rückgängigfunktionen, Fehlerbehandlung, Nutzerberechtigung, spez. Konfigurationen** berücksichtigen
- **unternehmensspez. & allg. Standards** berücksichtigen

13.4 Design-Pattern

- wichtig: Erfahrung
- Probleme treten immer wieder auf
- ⇒ von Experten Lösungen abschauen
- kein Allheilmittel, wichtig: richtiger Einsatz

Historie der Design Pattern

- **von Architektur abgeguckt**
- **dort wichtigste Literaturstelle:**
C. Alexander: *A Pattern Language: Towns, Buildings, Constructions*. Harvard University Press, Cambridge, MA, 1977.
- **beschreibt Lösungen für immer wiederkehrende Probleme**
- **ersten Arbeiten in Software-Entwicklung in erster Hälfte der 90er**
- **absolutes Standardwerk „Gang of Four“:**

Literatur

- (**GoF**) E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA: 1995
in Deutsch: Entwurfsmuster
- B. Goldfedder: *The Joy of Patterns*. Addison-Wesley: 2002
in Deutsch: Entwurfsmuster einsetzen
- spezielle Pattern: z.B. für Java, EJB, ...

Wie designt man?

- geg.: konkretes Problem: Sortieren von int-Zahlen
- finde: konkretes Design: Quicksort für int-Zahlen
- abstrahiere zu abstraktem Design (= Design-Pattern): Quicksort für geordnete Menge
- wende Lösung auf anderes Problem durch Konkretisierung des abstrakten Designs an (Quicksort für Wörter)

1. Beispiel

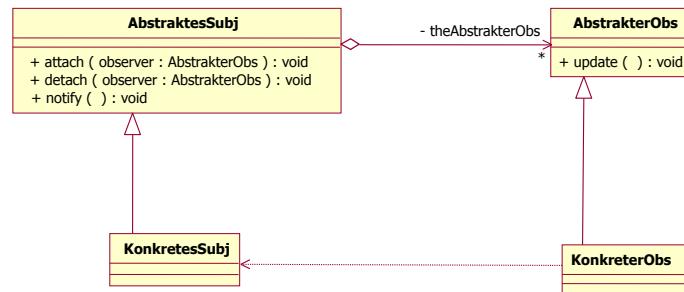
Aufgabe:

Es soll mehrere Darstellungen des Inhalts einer Excel-Tabelle oder einer Datenbank geben (z.B. Tabelle, Balkendiagramm, Kuchendiagramm, ...)

Bei Änderung eines Tabelleneintrags soll die Änderung, falls relevant, in allen Darstellungen nachvollzogen werden.

Beschreiben Sie verbal ein Design zur Lösung dieser Aufgabenstellung!

Observer-Muster (Publisher-Subscriber)



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Design & Pattern 10. Juni 2016 | 10

- Zweite Anwendung von Observer: Ereignisverarbeitung
- Dritte Anwendung von Observer: Implementierung eines Chat-Rooms
- Vierte Anwendung von Observer: Steuerungssystem mit mehreren Monitoren (Observer), die Hardware (Subject) überwachen

Struktur Muster-Beschreibung (aus GoF)

- **Name**
eindeutig & möglichst aussagekräftig
- **Zweck**
Was macht Entwurfsmuster?
Grundprinzip & Zweck?
spezifische Fragestellung oder Entwurfsprobleme
behandelt?
- **auch bekannt als**
andere Namen für dieses Muster
- **Motivation**
Bsp.-Szenario für Entwurfsproblem
wie Muster Problem löst
- **Anwendbarkeit**
in welchen Situationen und woran diese erkennbar

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Design & Pattern 10. Juni 2016 | 11

Struktur Muster-Beschreibung (Forts.)

- **Struktur**
UML-Diagramme (in GoF noch OMT-Diagramme)
- **Teilnehmer**
Beschreibung der beteiligten Klassen & deren Zuständigkeiten
- **Interaktionen zwischen Teilnehmern**
- **Konsequenzen**
wie Ziele erreicht
Vor- und Nachteile
welche Ergebnisse
welche Aspekte variabel
- **Implementierung**
Fallen, Tipps und Techniken bei Implementierung
sprachspezifische Aspekte und Impl.-Möglichkeiten

Struktur Muster-Beschreibung (Forts. II)

- **Beispielcode**
zur Verdeutlichung seiner Umsetzung
- **Bekannte Verwendungen**
mindestens 2 Beispiele aus unterschiedlichen Verwendungen
- **Verwandte Muster**
Beziehungen zu anderen Mustern
relevanten Unterschiede
welche Muster zusammenverwendbar

Muster-Beschreibungen in anderen Büchern teilweise sehr verschieden

Observer-Muster aus GoF (Ausschnitt)

• Zweck

Definiere eine 1-zu-n -Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und (automatisch) aktualisiert werden.

• Anwendbarkeit

- Wenn Abstraktion zwei Aspekte besitzt, von denen der eine von dem anderen abhängt. Geteilte Kapselung ermöglicht, die Objekte zu variieren und unabhängig wieder zu verwenden.
- Wenn Änderung eines Objekts die Änderung anderer Objekte verlangt und man nicht weiß, wie viele geändert werden müssen.
- Wenn Objekt andere benachrichtigen soll, ohne Annahmen zu treffen, wer diese anderen Objekte sind -> lose Kopplung

Observer-Muster aus GoF (Forts.)

• Konsequenzen

- abstrakte Kopplung zwischen Subjekt und Beobachter
- Unterstützung von Broadcasting-Kommunikation

• Implementierung

- Abbildung von Subjekten auf ihre Beobachter: Merken der Beobachter in Liste, allerdings teuer (#Subjekte > #Beobachter)
- Auslösen der Aktualisierung: Push/Pull?.
- Fehlerhafte Referenzen auf gelöschte Subjekte

• Bekannte Verwendungen

- Model-View-Controller: Model Subjekt, View Beobachter

Datei/Verzeichnissystem

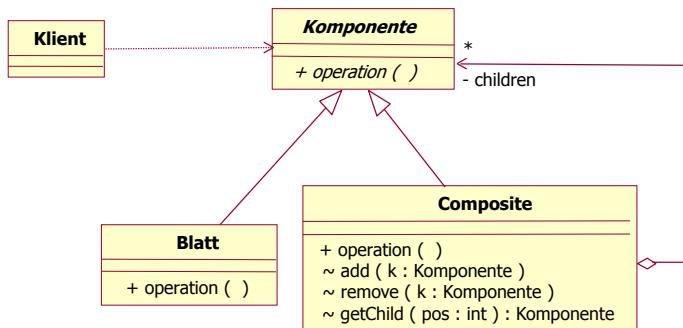
Aufgabenstellung:

Es soll ein Verzeichnissystem modelliert werden, bei dem die einzelnen Verzeichnisse aus Dateien oder wieder aus Verzeichnissen bestehen.

Für obiges System sollen Methoden zur Verfügung gestellt werden, die z.B. die Größe eines Verzeichnisses oder einer Datei liefern.

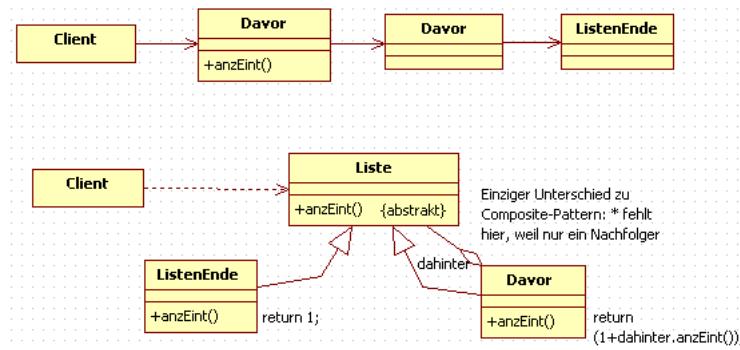
Der Client soll dabei nichts über die interne Struktur des Systems erfahren, insbesondere soll er beim Aufruf der Methoden nicht selbst unterscheiden müssen, ob es sich um eine Datei oder wieder um ein Verzeichnis handelt.

Composite-Muster-Struktur



13.5 Decorate-Muster

- ähnlich zu Composite (Baum), aber nur Liste

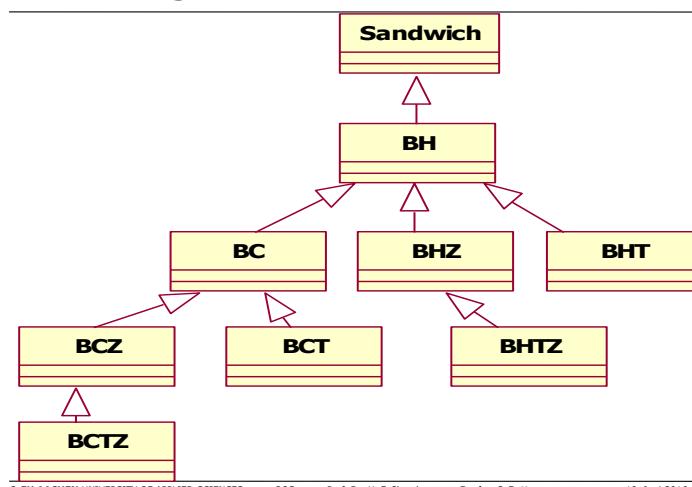


Susies BurgerShop

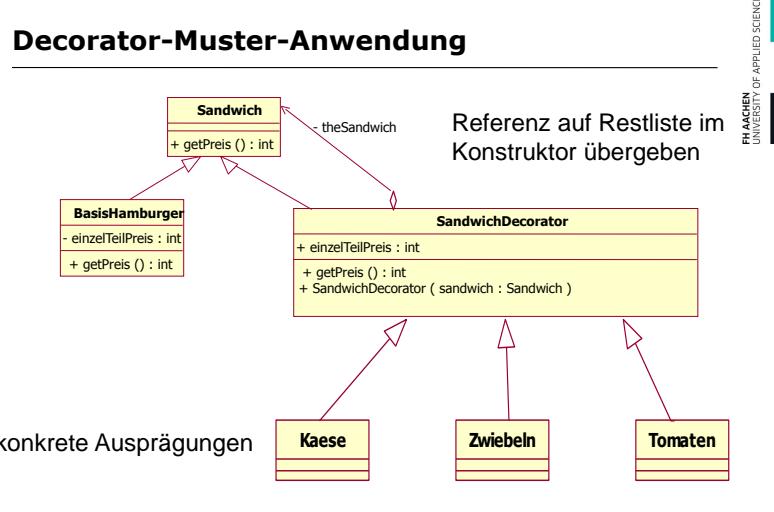
- verkauft nur Burger, aber in allen Variationen:

- (BH) BasisHamburger
- (BC) BasisCheeseburger
- (BHZ) BasisHamburger mit Zwiebeln
- (BCZ) BasisCheeseburger mit Zwiebeln
- (BHT) BasisHamburger mit Tomaten
- (BCT) BasisCheeseburger mit Tomaten
- (BHZT) BasisHamburger mit Zwiebeln und Tomaten
- (BCZT) BasisCheeseburger mit Zwiebeln und Tomaten
- ...

Klassendiagr.: Käse, Zwiebeln, Tomaten



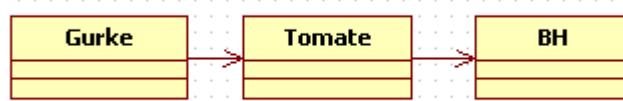
- `getPreis()` im Klassendiagramm: `getEinzelpreis() + super.getPreis()`
- Problem 1: wenn neue Auflagen dazu kommen, Verdoppelung der Klassenstruktur
- Problem 2: wenn Preis sich ändert, in Hälfte der Klassen ändern \Rightarrow Verletzung des Lokalitätsprinzips
- \Rightarrow Dekorator-Pattern



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Design & Pattern 10. Juni 2016 | 20

`getPreis()` bei

- BasisHamburger: return `einzelTeilPreis`;
- SandwichDecorator: return `einzelTeilPreis + theSandwich.getPreis()`;
- \Rightarrow Preisänderung nur in einer Klasse durchführen



```
new Gurke(new Tomate(new BH())))
```

```
Gurke(Sandwich sandwich) {
```

```
    this.theSandwich = sandwich;
```

{}

Decorator-Muster allgemein

- Darstellung einer Liste mit unterschiedlichen Einträgen
- Induktionsanfang: Listenende

Bsp.: BasisHamburger
 Allgemein: KonkreteKomponente

- Induktionsschluss: Listenverlängerung

Bsp.: SandwichDecorator
 Allgemein: Dekorierer

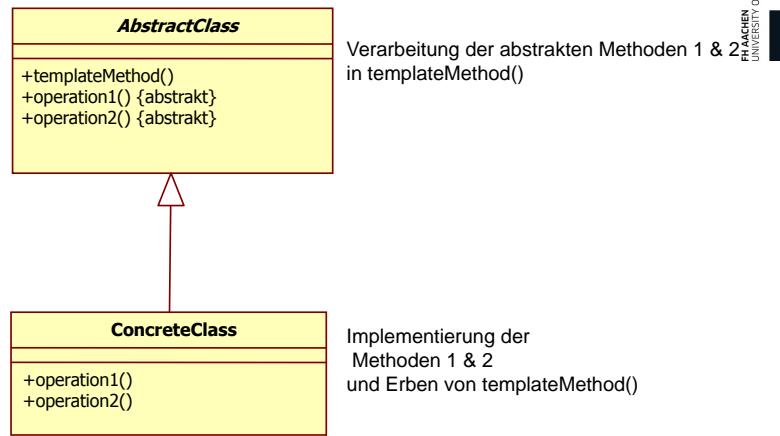
- Vereinigung der beiden Fälle

Bsp.: Sandwich
 Allgemein: Komponente

- Wichtig: Aufbau durch Konstruktor, entspricht Vorhängen

13.6 Weitere Pattern

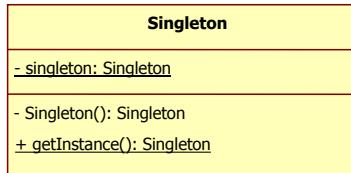
Template Method (Framework)



siehe Threads in ARBKVS: dort

- templateMethod: start
- operation: run

Singleton

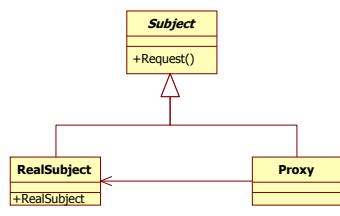


- einziges Objekt als privates statisches Attribut halten
- Konstruktor privat
- Zugriff auf Referenz durch statischen getter

Vorteile der Singleton-Verwendung:

- Sicherstellen, dass nur 1 Objekt verfügbar, z.B.: Drucker-Spooler
- Kontrolle und Steuerung der Erzeugung von Objekten der Klasse

Proxy-Pattern

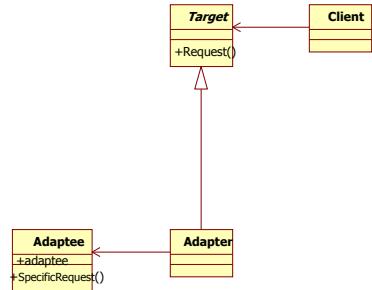


- Proxy implementiert gleiche Schnittstelle wie RealSubject
- Proxy hat Zugriff auf RealSubject-Objekt
- Client hat nur Zugriff auf Proxy
- Bsp.: Client-Orb in OOS

Vorteile der Proxy-Verwendung:

- dient auf Client-Seite als Ersatz für Server-Objekt (remote)
- kann Real-Subject erst bei Bedarf erzeugen (virtual)
- dient als Schutz für Real-Subject (protection)

Adapter-Pattern



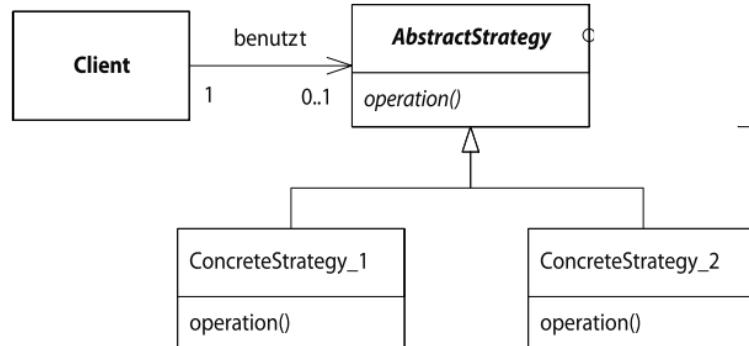
- Funktionalität vorhanden, aber Schnittstelle passt nicht zum Client-Aufruf
->setze Adapter dazwischen, der Schnittstelle anpasst und vorhandene Funktionalität nutzt

Vorteile der Adapter-Verwendung:

- Wiederverwendbarkeit
- Kontrolle des Zugriffs

© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Design & Pattern 10. Juni 2016 | 28

Strategy-Pattern (Diagramm)



© FH AACHEN UNIVERSITY OF APPLIED SCIENCES OOS Prof. Dr. H. Faßbender Design & Pattern 10. Juni 2016 | 30

Strategy-Pattern

- kapselt Algorithmus in Klasse
- soll unabhängig von Clients austauschbar sein

Bsp.:

- Sortieralgorithmensystem
- Feiertage in international nutzbarem Kalender

Vorteile der Strategy-Verwendung:

- Client nur von Abstraktion abhängig, nicht von Implementierung
- zusätzliche Implementierung hinzufügbar
- Flexibilität bei der Auswahl der Algorithmen