Prozedurale Programmiertechnik

Mark Keinhörster

FOM
Hochschule für Ökonomie und Management

27. Oktober 2013

Inhaltsverzeichnis

Einführung in die Programmierung

Ihre Erwartungen an die Veranstaltung

Was möchten Sie gerne behandeln?

- Parallele Programmierung mit Threads
-
-
-

Erwartungen an die Veranstaltung

Was sollten Sie am Ende können?

- Eigenständig einfache Programme schreiben
- Anweisungen, Ausdrücke, Operatoren, Kontrollstrukturen und Schleifen kennen
- Datentypen, Variablen und Konstanten kennen und sinnvoll einsetzen
- Mit Strings und Arrays sicher umgehen
- Das Prinzip der Rekursion verstehen und anwenden
- Sich selbständig in Java weiterentwickeln
- Die vorgestellten Konzepte verstehen und anwenden

In der Veranstaltung verwendete Werkzeuge

Programmiersprache

Entwicklungsumgebung

Warum Java?

Java ist. . .

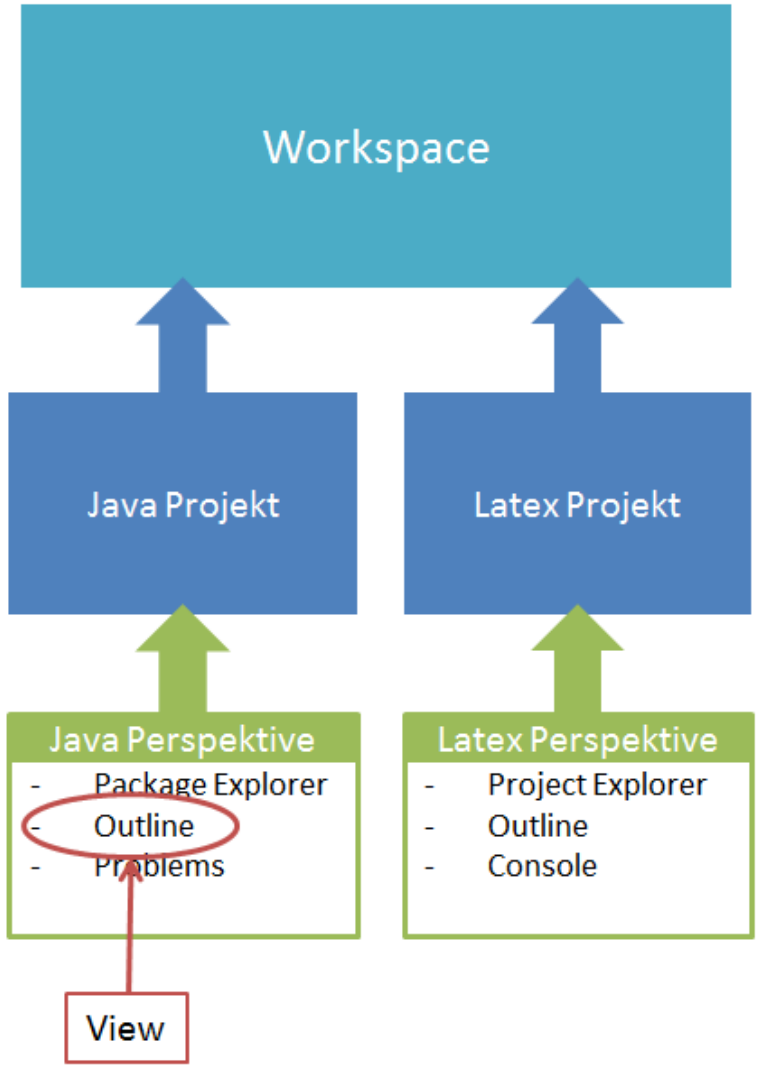
- Weit verbreitet.
- Verhältnismäßig leicht zu erlernen.
- Plattformunabhängig.

Dokumentation

`http://docs.oracle.com/javase/7/docs/api/`

Eclipse erleichtert uns die Entwicklung

- Integrated Development Environment
- Verwaltet Dateien in Projekten
- Es existieren 4 Hauptkomponenten:
  1. Workspaces
  2. Projekte (Projects)
  3. Perspektiven (Perspectives)
  4. Sichten (Views)



Einordnung der prozeduralen Programmierung

Imperative Programmierung

- Prozedurale Programmierung
  z.B. C, Cobol, Pascal
- Objektorientierte Programmierung
  z.B. Java, C#, SmallTalk
- Skriptsprachenorientierte Programmierung
  z.B. PHP, JavaScript, Perl, Python

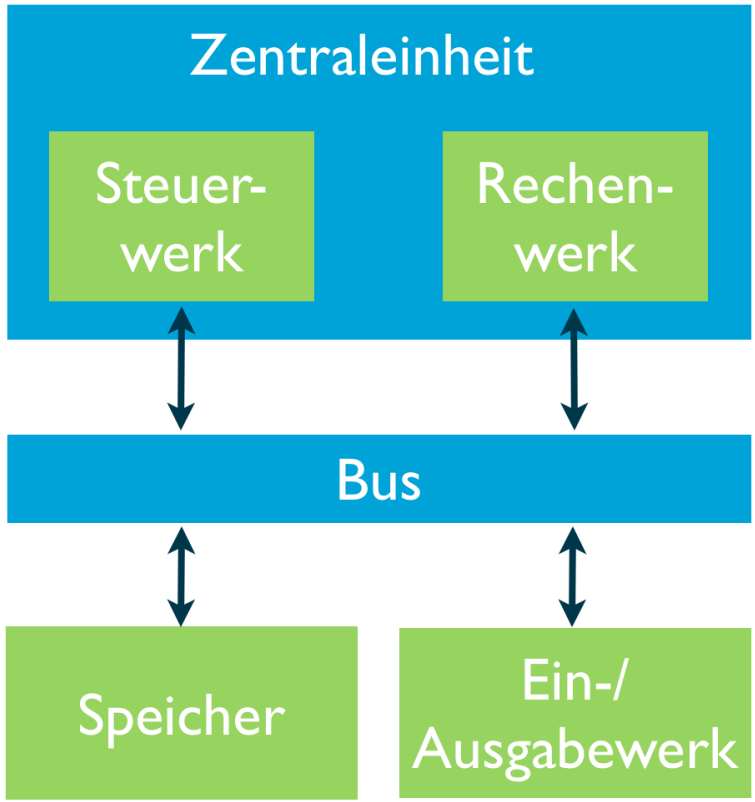
Deklarative Programmierung

- Funktionale Programmierung
  z.B. Lisp, Haskell
- Prädikative Programmierung
  z.B. Prolog

Gesamtkonzept für den Aufbau eines universellen Rechners.

Besteht aus folgenden Komponenten:

- Steuerwerk
- Rechenwerk
- Speicher
- Eingabewerk
- Ausgabewerk



Steuerwerk

Steuerwerk ist zentrale Komponente, die eine endliche Menge von Operationen ausführen kann.

Es existieren verschiedene Arten von Operationen:

- Transportoperationen
  z.B. Daten von Speicher in Rechenwerk
- Arithmetische Operationen
- Logische Operationen
  z.B. Vergleiche (und/oder/not)
- Operationen zur Steuerung des Kontrollflusses
  z.B. Sprünge um von gespeicherter Operationsreihenfolge während Ausführung abzuweichen
- Spezialoperationen
  z.B. Ein- / Ausgabeoperationen

Wichtige Eigenschaften eines von-Neumann-Rechners

Eigenschaften der von-Neumann-Architektur:

- Technischer Aufbau ist unabhängig von Aufgabenstellung
- Lösung der Aufgabe durch vorgegebene Befehlsabfolge
- Befehlsabfolge = Programm
- Ablage von Programm und Daten im gleichen Speicher
- Zentraleinheit besitzt weitere, eigene, Speicherzellen (Register) z.B. Akkumulator, Basisregister, Zählregister, Datenregister
- Zum Ausführen von Operationen werden diese in Befehlsregister geladen

Programmierung?

- Programm = Befehlsfolge im Speicher = Folge von Nullen und Einsen
- Programm in Binärcodierung = Maschinensprache
- Programmierung in Maschinensprache ist aufwendig und fehleranfällig
- Daher Einführung von Assembler: Ersetzung von Binärcodes durch Mnemonics
  - Beispiel: Zahl 5 zum Akkumulator addieren
    in Maschinensprache: 0110100100000101
    in ASM: add 5
- Nach Formulierung des Programms folgt die Übersetzung in Maschinensprache

Gründe für höhere Programmiersprachen?

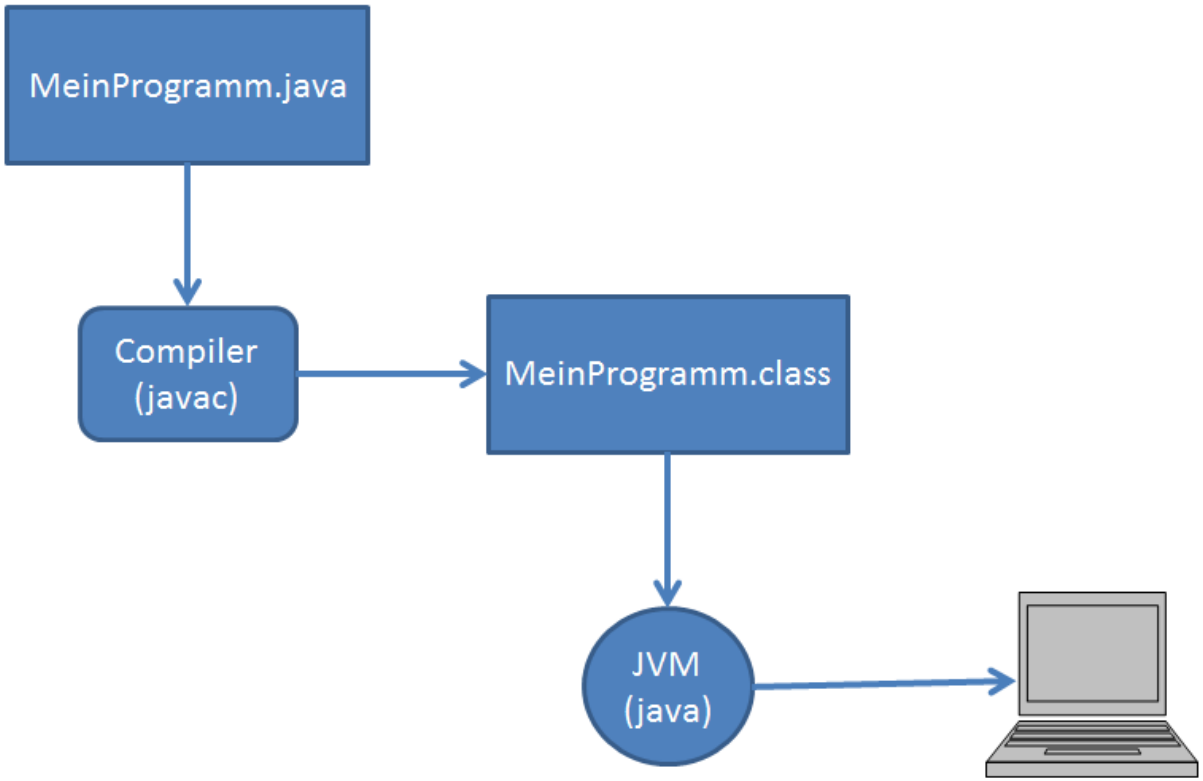
- Im Laufe der Zeit entstanden Prozessoren mit unterschiedlichen Maschinensprachen
  Portierung bedeutete Neuprogrammierung
- ASM orientiert sich an Computer, nicht an Problemlösung
  Mit höherer Programmiersprache sollte sich Lösung leichter formulieren lassen

Übersetzung höherer Programmiersprachen

- Übersetzung von Hochsprache in Maschinensprache ist komplizierter als Übersetzung von ASM
- Ausgangsprogramm = Quellprogramm/Quellcode
- Übersetztes Programm = Zielprogramm
- Übersetzer = Compiler

Grundlagen der Programmierung mit Java

Java - Von der Übersetzung zur Ausführung



Erste Schritte

Hello World!

- Das Code-Listing zeigt unser erstes Java-Programm
- Es wird mithilfe des Befehls "javac HelloWorld.java" übersetzt
- Anschließend wird es mit "java HelloWorld" gestartet
- mit "javap -c HelloWorld" lässt sich der Bytecode lesen

Test.java

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

Aufbau von Java-Dateien

- Jede .java Datei enthält eine Klassendefinition
- Wichtig: Dateiname = Klassenname
- Klassen enthalten Methoden
- Methoden enthalten Anweisungen

Test.java

```
//Klassendefinition
class Test {
// Der Methodenkopf
void test(){
//Eine Anweisung
System.out.println("Dies ist ein
     Test");
}
}
```

Aufbau von Java-Dateien

- Programm besteht (meistens) aus mehreren Klassen
- Eine Klasse beeinhaltet eine Main-Methode
- JVM startet Main-Methode
- Programm endet nach Ablauf der Main-Methode

TestMitMain.java

```
class TestMitMain {
public static void main(String[]
      args){
System.out.println("Dies ist ein
      Test");
}
}
```

Grundbegriffe

Begrifflichkeiten

- Variablen und Konstanten
  - Eine Variable ist ein mit Namen versehener Speicherplatz inklusive Inhalt
  - Eine Konstante ist eine Variable, der nur genau einmal ein Wert zugewiesen werden darf
- Datentypen
  - Kombination aus Wertebereich und dazugehörigen Operationen
  - Beispiel Ganzzahl:
    Wertebereich $= 2^{32}$,
    Operationen = Addition,Subtraktion...
- Operatoren
  - Bestimmen wie mit Variablen, Konstanten und Literalen umgegangen wird

Begrifflichkeiten

- Anweisungen
  - Verarbeitungsvorschrift in imperativem Programm
  - Durch Ausführung einer Anweisung werden Daten oder Adressen verarbeitet
  - Verändert den Programmzustand
  - Wichtige Anweisungstypen sind Variablendeklaration, Zuweisung, Auswahl, Schleife, Block...
  - Anweisungsfolge = Sequenz
  - Einfachste Form besteht in Java lediglich aus ';'
- Ausdrücke
  - Verknüpfung von Variablen, Konstanten, Literalen oder anderen Ausdrücken durch Operatoren
  - Werden in bestimmter Reihenfolge ausgewertet und haben Wert
  - Einfachste Form besteht lediglich aus Konstante/Variable
  - Wird in Java durch Abschluss mit Semikolon zu Anweisung

Datentypen

Datentypen

- Java enthält 8 primitive Datentypen
- Primitive Datentypen sind keine Objekte

Name	Wertebereich	Standard
char	Unicode-Zeichen	u0000
byte	-128 bis 127	0
short	-32768 bis 32767	0
int	-2147483648 bis 2147483647	0
long	-2^{63} bis $2^{63}-1$	0
float	+/-3.40282347*1038	0.0
double	+/-1.79769313486231570*10308	0.0
boolean	false, true	false

Zeichen

Der primitive Datentyp "char"

- speichert Zeichen aus der Unicode-Zeichentabelle
- 2 Byte (16 Bit) groß
- jedes Zeichen besitzt einen code

```
char zeichen;
zeichen = 'a';
zeichen = 97;
```

Zeichen

ASCII Tabelle

Scan-code	ASCII hex	ASCII dez	Zeichen	Scan-code	ASCII hex	ASCII dez	Zeichen	Scan-code	ASCII hex	ASCII dez	Zeichen	Scan-code	ASCII hex	ASCII dez	Zeichen
	00	0	NUL		20	32	SP		40	64	@	0D	60	96	`
	01	1	SOH ^A	02	21	33	!	1E	41	65	A	1E	61	97	a
	02	2	STX ^B	03	22	34	"	30	42	66	B	30	62	98	b
	03	3	ETX ^C	29	23	35	#	2E	43	67	C	2E	63	99	c
	04	4	EOT ^D	05	24	36	$	20	44	68	D	20	64	100	d
	05	5	ENQ ^E	06	25	37	%	12	45	69	E	12	65	101	e
	06	6	ACK ^F	07	26	38	&	21	46	70	F	21	66	102	f
	07	7	BEL ^G	0D	27	39	'	22	47	71	G	22	67	103	g
0E	08	8	BS ^H	09	28	40	(	23	48	72	H	23	68	104	h
0F	09	9	TAB ^I	0A	29	41	)	17	49	73	I	17	69	105	i
	0A	10	LF ^J	1B	2A	42	*	24	4A	74	J	24	6A	106	j
	0B	11	VT ^K	1B	2B	43	+	25	4B	75	K	25	6B	107	k
	0C	12	FF ^L	33	2C	44	,	26	4C	76	L	26	6C	108	l
1C	0D	13	CR ^M	35	2D	45	-	32	4D	77	M	32	6D	109	m
	0E	14	SO ^N	34	2E	46	.	31	4E	78	N	31	6E	110	n
	0F	15	SI ^O	08	2F	47	/	18	4F	79	O	18	6F	111	o
	10	16	DLE ^P	0B	30	48	0	19	50	80	P	19	70	112	p
	11	17	DC1 ^Q	02	31	49	1	10	51	81	Q	10	71	113	q
	12	18	DC2 ^R	03	32	50	2	13	52	82	R	13	72	114	r
	13	19	DC3 ^S	04	33	51	3	1F	53	83	S	1F	73	115	s
	14	20	DC4 ^T	05	34	52	4	14	54	84	T	14	74	116	t
	15	21	NAK ^U	06	35	53	5	16	55	85	U	16	75	117	u
	16	22	SYN ^V	07	36	54	6	2F	56	86	V	2F	76	118	v
	17	23	ETB ^W	08	37	55	7	11	57	87	W	11	77	119	w
	18	24	CAN ^X	09	38	56	8	2D	58	88	X	2D	78	120	x
	19	25	EM ^Y	0A	39	57	9	2C	59	89	Y	2C	79	121	y
	1A	26	SUB ^Z	34	3A	58	:	15	5A	90	Z	15	7A	122	z
01	1B	27	Esc	33	3B	59	;		5B	91	[		7B	123	{
	1C	28	FS	2B	3C	60	<		5C	92	\		7C	124	|
	1D	29	GS	0B	3D	61	=		5D	93	]		7D	125	}
	1E	30	RS	2B	3E	62	>	29	5E	94	^		7E	126	~
	1F	31	US	0C	3F	63	?	35	5F	95	_	53	7F	127	DEL

Ganzzahlige Datenypen

- byte
  - für sehr kleine Zahlen
  - 1 Byte (8 Bit) groß
  - Fängt bei Über- oder Unterschreitung des Wertebereichs wieder von vorn an
- short
  - für kleine Zahlen
  - 2 Byte (16 Bit) groß
  - Fängt bei Über- oder Unterschreitung des Wertebereichs wieder von vorn an

Ganzzahlige Datenypen

- int
  - für gewöhnliche Zahlen
  - 4 Byte (32 Bit) groß
  - Fängt bei Über- oder Unterschreitung des Wertebereichs wieder von vorn an
- long
  - für sehr große Zahlen
  - 8 Byte (64 Bit) groß
  - Fängt bei Über- oder Unterschreitung des Wertebereichs wieder von vorn an

Fließkommazahlen

- float
  - für gewöhnliche Fließkommazahlen
  - 4 Byte (32 Bit) groß
  - Fängt bei Über- oder Unterschreitung des Wertebereichs wieder von vorn an
  - indikator: f (float x = 0.6f)
- double
  - für größere Fließkommazahlen
  - 8 Byte (64 Bit) groß
  - Fängt bei Über- oder Unterschreitung des Wertebereichs wieder von vorn an
  - indikator: d (double x = 0.6d)

Boolesche Datentypen

- boolean
  - für Wahrheitswerte
  - kann Wahr (true) oder Falsch (false) annehmen
  - 1 Byte (8 Bit) groß

Variablen

Variablen

- Speichern Werte
- Können gelesen und geschrieben werden
- Namen bestehen aus
  - Buchstaben
  - Ziffern
  - Unterstrich
- Konstanten in Großbuchstaben
- Variablen sollten mit Kleinbuchstaben beginnen (anschließend Camel-Case)
- Keine Schlüsselwörter als Namen

Java Schlüsselwörter

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

(Übernommen von Herrn Prof. Dr. Dirk Wiesmann)

Variablendeklaration

Deklaration bedeutet...

- Variable benennen und dem Compiler bekanntmachen
- Speicher für die Variable reservieren

Durch Initialisierung kann die Variable nun auf einen Anfangswert gesetzt werden.

```
// Syntax:
// Datentyp VARNAME;

// Variablen einzeln deklarieren
int i;
int x;
int y;

// Diese Schreibweise ist auch
     gueltig
int i, x, y;
```

Variableninitialisierung

Initialisierung bedeutet...

- Variablendeklaration um eine Zuweisung ergänzen
- die deklarierte Variable mit einem Wert zu befüllen

Der Wert einer Konstante darf nach der Initialisierung nicht mehr geändert werden

```
// Variablen einzeln deklarieren
int i = 20;
int x = 10;
int y = i + x;

// Diese Schreibweise ist auch
     gueltig
int i = 20, x = 10, y = i+x;

// Auch getrennt moeglich
int a;
a = 100;
```

Typen von Variablen

Es gibt in Java drei Arten von Variablen

1. Klassenvariablen
2. Lokale Variablen
3. Instanzvariablen (OOP)

```
class Variablen {

//Klassenvariable
static boolean bool = true;

//Instanzvariable
int i = 5;

public static void main(String[]
     args){

//lokale Variable
int i = 1;

}
}
```

Lebensdauer

- Klassenvariablen: Gesamte Programmlaufzeit
- Lokale Variablen: Bis zum Ende des Methodenaufrufs
- Instanzvariablen: Existenz des Objekts (OOP)

Sichtbarkeit

- Klassenvariablen: Innerhalb der Klasse
- Lokale Variablen: Innerhalb eines Blocks
- Instanzvariablen: Innerhalb des Objekts (OOP)

Casting

- Java kann Typen implizit casten
- Entwickler kann Typen explizit casten
- Casten in größeren Datentyp geht implizit
- Casten in kleineren Datentyp explizit da Informationsverlust!

```
int i = 10;

// Cast in groesseren Typ
// kein Problem
long l = i;

// Cast in kleineren Typ explizit
short s = (short) i;
```

Aufgaben

1. Weisen sie in der Main-Methode die Zahl 25 einer ganzzahligen Variable zu, anschließend:
   1. Geben Sie die Variable in der Form: "Wert: 25" aus
   2. Geben Sie 25/5, 25/3, 25/3.0 aus
2. Deklarieren sie die short-Variable "einShort" und initialisieren Sie sie mit 4096, anschließend casten Sie die Variable in den Datentyp "byte" und geben Sie die Zahl aus, was fällt auf?

Operatoren und Ausdrücke

Arten von Operatoren

Die drei wichtigsten Operatorgruppen sind

- Arithmetische Operatoren
- Logische Operatoren
- Zuweisungsoperatoren

Je nach Programmiersprache existieren weitere Operatoren (Inkrement-/Dekrement-/Bit-Operatoren).

Unterteilung nach Anzahl der Operanden

Jeder Operator wird auf eine bestimmte Anzahl von Operanden angewendet. Dadurch entsteht eine weitere Unterteilung:

- Unäre Operatoren = 1 Operand, der hinter dem Operator folgt
- Binäre Operatoren = 2 Operanden, Infix-Notation in Java
- Tertiäre Operatoren = 3 Operanden

Auswertungsreihenfolge von Operatoren

Auswertungsreihenfolge ergibt sich aus 2 Faktoren:

- Priorität des Operators
  z.B. "Punkt-vor-Strich"
- Assoziativität des Operators = Bindung zwischen gleichwertigen Operatoren
  z.B: Ausdruck mit arithmetischen Operatoren der gleichen Priorität wird von links nach rechts ausgewertet.

Durch Klammerung von Teilausdrücken kann die Auswertungsreihenfolge erzwungen werden.

Operatoren für numerische Datentypen:

Name	Erläuterung
-	Subtraktion, neg. Vorzeichen
*	Multiplikation
/	Division
%	Modulo
++	Prä -/ Postinkrement
--	Prä -/ Postdekrement

Vergleichsoperatoren

Name	Erläuterung
==	Gleich
!=	Ungleich
<	Kleiner
>	Größer
<=	Kleiner gleich
>=	Größer gleich

Logische Operatoren

Name	Erläuterung
&&	UND (Shortcircuit)
||	ODER (Shortcircuit)
!	NICHT
&	UND
|	ODER
^	Exklusiv ODER

Assoziativität

Operator	Assoziativität
[ ] . ( ) (Methodenaufruf)	→
! ~ ++ -- + (unär) – (unär) ( ) new	←
* / %	→
+ -	→
<< >> >>>	→
< <= > >= instanceof	→
== !=	→
&	→
^	→
|	→
&&	→
||	→
? :	→
= += *= /= %= &= |= ^= <<= >>=	←

(Übernommen von Herrn Prof. Dr. Dirk Wiesmann)

Aufgaben

1. Erstellen Sie ein Programm zur Multiplikation zweier Zahlen, die im Programm einen festen Wert zugewiesen bekommen.
2. Verbessern Sie dieses Programm, indem mit Hilfe der Klasse "Tastatur" nun Zahlen über die Konsole eingegeben werden können.
3. Schreiben Sie ein Programm, dass den Benzinverbrauch eines Autos in Litern je 100 Kilometer errechnet. Als Eingabe benötigt das Programm den Benzinverbrauch in Litern und die gefahrenen Kilometer. Der Verbrauch pro 100 Kilometer ergibt sich aus: $Liter * 100/km$.

Kontrollstrukturen

Kontrollstrukturen definieren die Reihenfolge in der Anweisungen ausgeführt werden.

Kontrollstrukturen - Block

- Fasst mehrere Anweisungen Zusammen
- Kann stehen wo auch einzelne Anweisungen stehen
- Kann geschachtelt werden

```
{
System.out.println("Ausgabe");
System.out.println("Ausgabe");
}
```

- Bedingte Anweisung
- Ausdruck "Bedingung" muss boolescher Ausdruck sein
  d.h. *true* oder *false*

```
if(Bedingung){
System.out.println("Die Bedingung
      ist True");
}
```

Kontrollstrukturen - Fallunterscheidung

- Mehrfachauswahl
- Auch hier:
  Ausdrücke "Bedingung1" und "Bedingung2" müssen boolesche Ausdrücke sein
- else-Zweig wird durchlaufen wenn kein Ausdruck wahr ist

```
if(Bedingung1){
System.out.println("Bedingung1 ist
      True");
} else if(Bedingung2){
System.out.println("Bedingung2 ist
      True");
} else {
System.out.println("Keine Bedingung
      ist True");
}

// If-Then-Else als
// ternaerer Operator
Ausdruck = ( BedingungX ) ? Wahr :
      Falsch;
```

Kontrollstrukturen - Fallunterscheidung

- Mehrfachauswahl
- Ausdruck vom Typ: byte, short, char, int
- Nach case darf genau 1 Konstante stehen
- Wenn keine passende Konstante dann "default" Anweisung (wenn vorhanden)
- Ohne break: ausführen aller Anweisungen ab Übereinstimmung
- "break" ist syntaktisch nicht erforderlich

```
switch(Ausdruck){

case konst1:
Anweisung1;
break;

case konst2:
Anweisung2;
break;

default:
Anweisung3;
break;

}
```

- Kopfgesteuerte Schleife
- Prüft Ausdruck zu Beginn

```
while (Bedingung){
Anweisung1;
...
Anweisungn;
}
```

- Fußgesteuerte Schleife
- Prüft Ausdruck am Ende der Schleife

```
do{
Anweisung1;
...
Anweisungn;
}while (Bedingung);
```

- Zählschleife
- Auch Kopfgesteuert
- "Initial" wird vor 1. Durchlauf ausgewertet
- "Bedingung" wird vor jedem Durchlauf ausgewertet
- "Inc/Dec" wird nach jedem Durchlauf ausgewertet

```
for(Initial; Bedingung; Inc/Dec)
{
Anweisung1;
...
AnweisungN;
}
```

Aufgaben

1. Lassen Sie ein Rechteck von 10*10 Zeichen mit Sternchen ausfüllen
2. Erstellen Sie einen Taschenrechner, bei dem der Benutzer zwei Eingabewerte und die jeweilige Rechenoperation eingibt. Das Ergebnis der gewünschten Rechenoperation soll ermittelt und ausgegeben werden. Implementieren Sie dazu ein rudimentäres Menü, sodass der Nutzer den Taschenrechner solange nutzen kann, bis er das Programm beenden möchte.
   Nutzen Sie für die Eingabe die Klasse "Tastatur".
3. Erweitern Sie Ihr Programm um die Berechnung der Fakultät (n! = n * n-1 * n-2 * 1; 0! = 1).
4. Berechnen Sie die Tabelle für das kleine Einmal-Eins für die Werte von 1 bis 9 und geben das Ergebnis formatiert aus

1. Geben Sie die Fibonacci-Zahlen von 1 bis n aus. Die Fibonacci-Folge ist eine Zahlenfolge, bei der sich die jeweils folgende Zahl durch Addition ihrer beiden vorherigen ergibt.
   Beispiele: $1 + 2 = 3$, $2 + 3 = 5$, $3 + 5 = 8$, $5 + 8 = 13$.
   Die Formel lautet also: $F(n + 2) = F(n + 1) + F(n)$
   mit $f(0) = 0$ und $f(1) = 1$

Aufgaben

1. Erstellen Sie ein Programm zur Addition der ersten 15 ungeraden Zahlen (d. h. 1+3+5...+15)
2. Verbessern Sie das Programm, indem die Endzahlen nun über die Konsole eingegeben werden können.
3. Erstellen Sie ein Programm zur Bestimmung der Primzahlen zwischen 1 und 100.
   Die Idee: Jede Zahl n ($1 < n < 100$) wird durch alle Zahlen d ($1 < d < n$) dividiert. Das Ergebnis e wird mit d wieder multipliziert. Ist n = e*d, so ist n keine Primzahl.

Arrays

Arrays

- Array (Feld) fasst mehrere Variablen des gleichen Typs zusammen
- Auf die einzelnen Elemente wird über einen Index zugegriffen
- In Java ist ein Array ein Objekt (OOP)
  - Array-Variable ist Referenz (OOP)
  - Erzeugung zur Laufzeit (OOP)
- Größe des Feldes wird in "length" gespeichert

Arrays

- Der Index beginnt bei 0
- Index muss vom Typ int sein
- Die Feldgrenzen werden von Java überprüft
- Bei Nicht-Einhaltung der Grenzen: Fehler!

Deklaration:

```
// Reine Deklaration, Objekt
     existiert noch nicht
int[] intArray;
```

Initialisierung:

```
// Erzeugung des Array-Objekts auf
      dem Heap
intArray = new int[5];

// Via Literale deklarieren und
     initialisieren:
int[] meinZweitesArray =
     {1,2,3,4,5};
```

Zuweisung von Werten:

```
intArray[0] = 1;
intArray[1] = 2;
...
```

Mehrdimensionale Arrays

- Mehrdimensionales Array ist ein "Array von Arrays"
- Anzahl der Dimensionen ist unbegrenzt

Deklaration:

```
int[][] multiDimIntArray;
```

Initialisierung:

```
multiDimIntArray = new int[2][3];

// oder via literale
int[][] meinZweitesMultiDimArray =
      {{1,2,3},{4,5,6}};
```

Zuweisung von Werten:

```
multiDimIntArray[0][0] = 1;
multiDimIntArray[0][1] = 2;
...
```

Aufgaben

1. Entwerfen Sie ein Programm, das eine vorgegebene Anzahl Integer-Werte von der Tastatur einliest. Anschließend sollen zwei Funktionen den kleinsten und den größten eingegebenen Wert finden, die dann ausgegeben werden.
2. Schreiben Sie ein Programm, dass den Benutzer auffordert, zehn Schulnoten als Ganzzahlen einzugeben. Diese Zahlen sollen in einem Array gespeichert werden. Im Anschluss berechnen Sie die Summe sowie den Durchschnitt und geben diese aus.

Aufgaben

1. Schreiben Sie ein Java-Programm das Testet, ob eine Matrix ein magisches Quadrat ist. Dazu ist die Methode istMagisch(int[][] matrix) zu implementieren. (Ein magisches Quadrat ist eine n x n Matrix, in der die Summe aller Zeilen, Spalten sowie der beiden Diagonalen gleich ist.)
2. Entwickeln Sie eine kommandozeilenbasierte Version des Spiels "Tic-Tac-Toe"

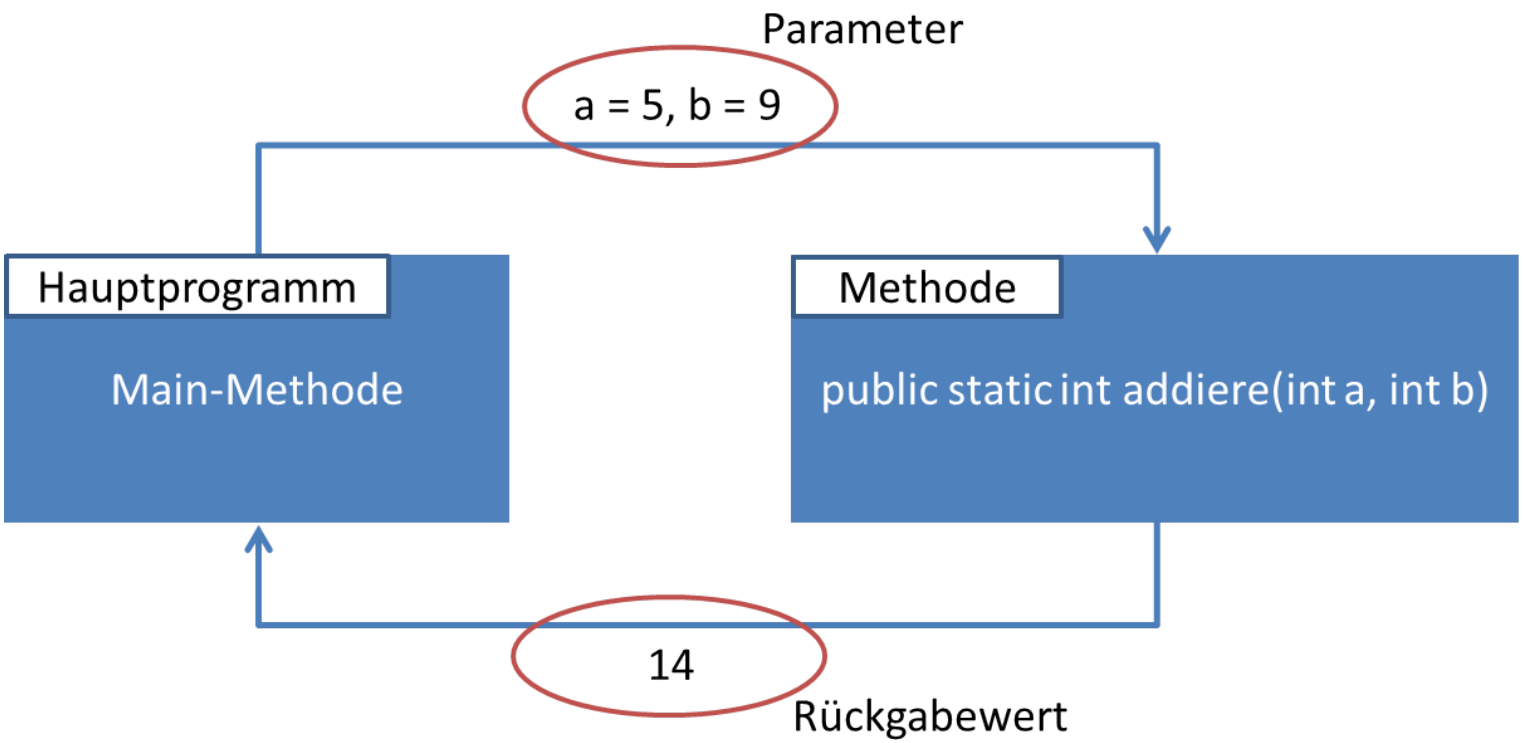
Methoden

Divide-and-Conquer (Teile und herrsche)

Divide-and-Conquer bedeutet große Probleme in kleinere Teilprobleme zu zerlegen.

- Dieses Prinzip wird in Java durch Methoden ermöglicht.

Main-Methode und ausgelagerte Teilfunktionalität



Vorteile von Methoden

Vorteile:

- Bessere Lesbarkeit des Programms
- Wiederverwendung von Code
- Fehler lassen sich schneller finden
- Fehler müssen nur an einer Stelle behoben werden

Methoden

Methoden besitzen:

1. Einen Methodennamen
2. Anweisungsblock
3. Lokale Variablen
4. 0..* Parameter
5. 0..1 Rückgabewerte (ohne Rückgabe: void)
6. Wenn Rückgabewert dann "return wert;"
7. Zugriff auf Klassenattribute
8. Können überladen werden

```
static Rueckgabetyp Methodenname (
     Typ Parameter, ...){
Anweisung1;
...
AnweisungN;

return wert;
}
```

Beispielmethode: summieren

Beispiel einer Java-Methode

```
static int summiere (int a, int b){
int summe = a + b;
return summe;
}
```

Übergabe von Parametern

Parameter werden in Java nach dem Call-by-Value Prinzip übergeben.

Call-by-Value bedeutet. . .

- das Kopieren der Übergabeparameter per Wert,
- ungeachtet ob es primitive Datentypen oder Objektreferenzen sind.

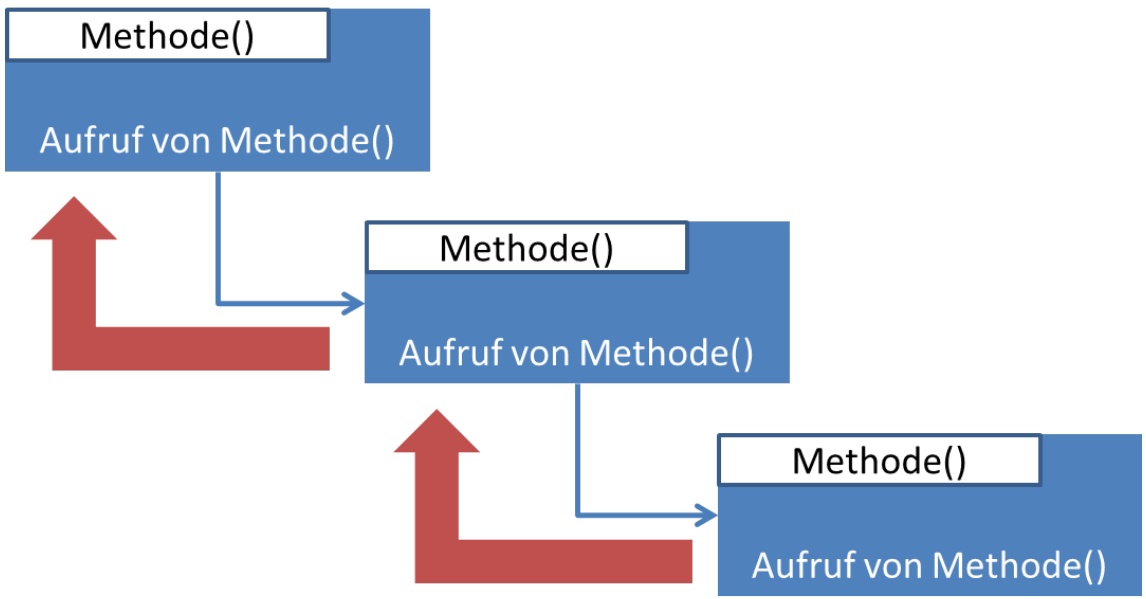
Aufgaben

1. Strukturieren Sie den implementierten Taschenrechner indem Sie eigenständige Anweisungsblöcke in einzelne Methoden auslagern
2. Entwerfen Sie eine Funktion berechneUmfang(), die den Umfang eines Kreises anhand des Radius berechnet.
   pi = 3,141492
   Umfang = 2 * pi * Radius
3. Entwerfen Sie eine Funktion berechneFlaeche(), die die Flaeche eines Kreises anhand des Radius berechnet.
   pi = 3,141492
   Flaeche = $pi * Radius^2$

Rekursion

Eine Funktion ist rekursiv, wenn sie sich selbst aufruft.

- oftmals eine alternative zu Schleifen



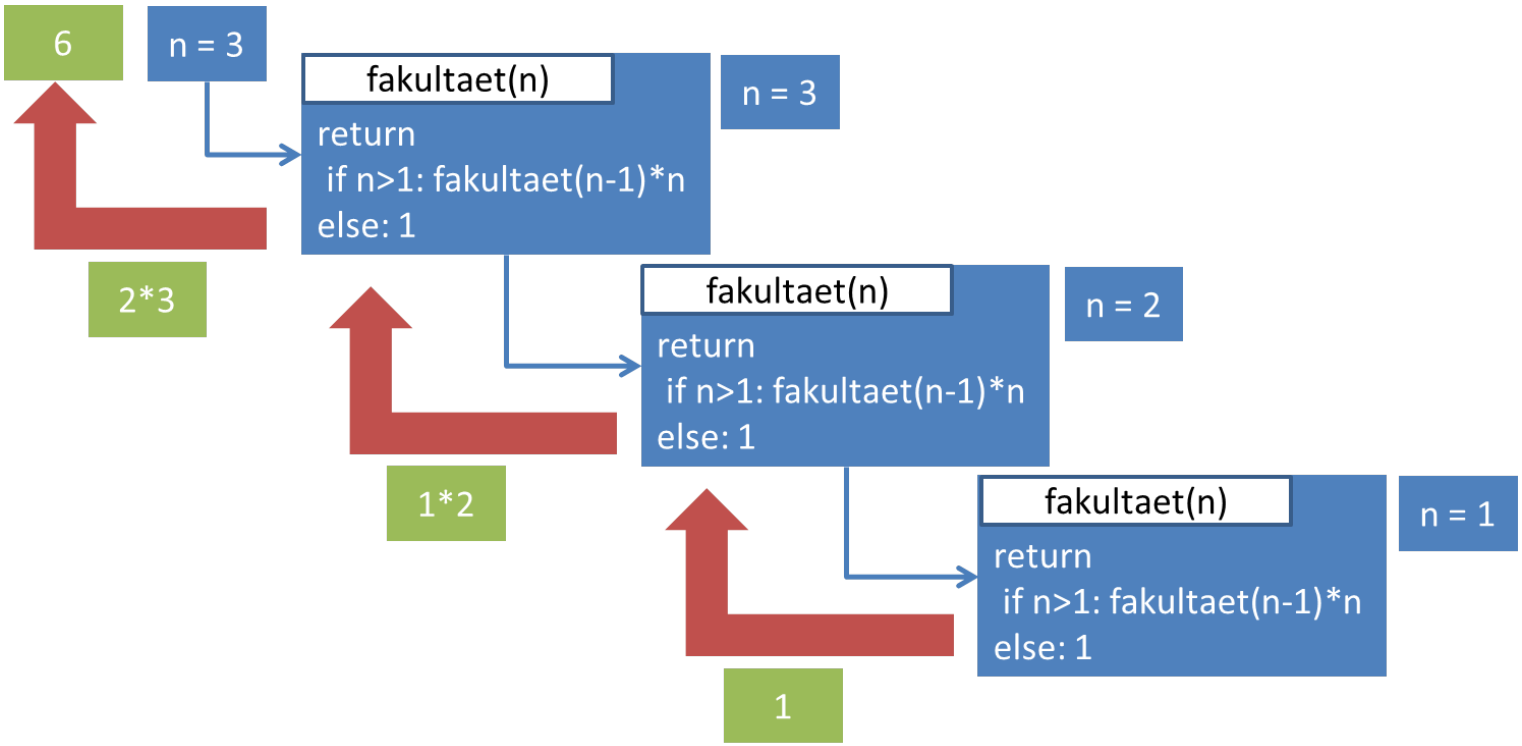
Rekursive Berechnung der Fakultät

Fakultät (!n)
= Multiplikation der Zahlen von 1 bis n
Beispiel: 3! = 3 * 2 * 1

- Die ersten (n-1) Faktoren des Produkts n! ergeben (n-1)!
  1. n! = (n-1)! · n falls n>1
  2. n! = 1 falls n=1
- zu 1)
  n! zu berechnen wurde auf (n-1)! reduziert
- zu 2)
  Notwendig um Rekursion zu beenden

```
static int fakultaet (int
      n){
int f;
if(n > 1){
f = fakultaet(n-1)*n;
} else {
f = 1;
}
return f;
}
```

Analyse der Funktionsausführung



Aufgaben

1. Geben Sie die Fibonacci-Zahlen von 1 bis n rekursiv aus.
2. Schreiben Sie ein rekursives Programm, das den größten gemeinsamen Teiler (GGT) zweier ganzer Zahlen berechnet.
   1. ist Zahl1 == Zahl2 dann Ergebnis = Zahl1
   2. ist Zahl1 > Zahl2 dann Ergebnis = ggT(Zahl1-Zahl2, Zahl2)
   3. ist Zahl1 < Zahl2 dann Ergebnis = ggT(Zahl1, Zahl2-Zahl1)
3. Der Springer darf beim Schach nur in L-Form bewegt werden. Ermitteln Sie rekursiv alle Züge, damit der Springer (von unten links angefangen) einmal alle Felder besucht ohne eines doppelt zu besuchen. Geben Sie anschließend das besuchte Schachfeld aus.

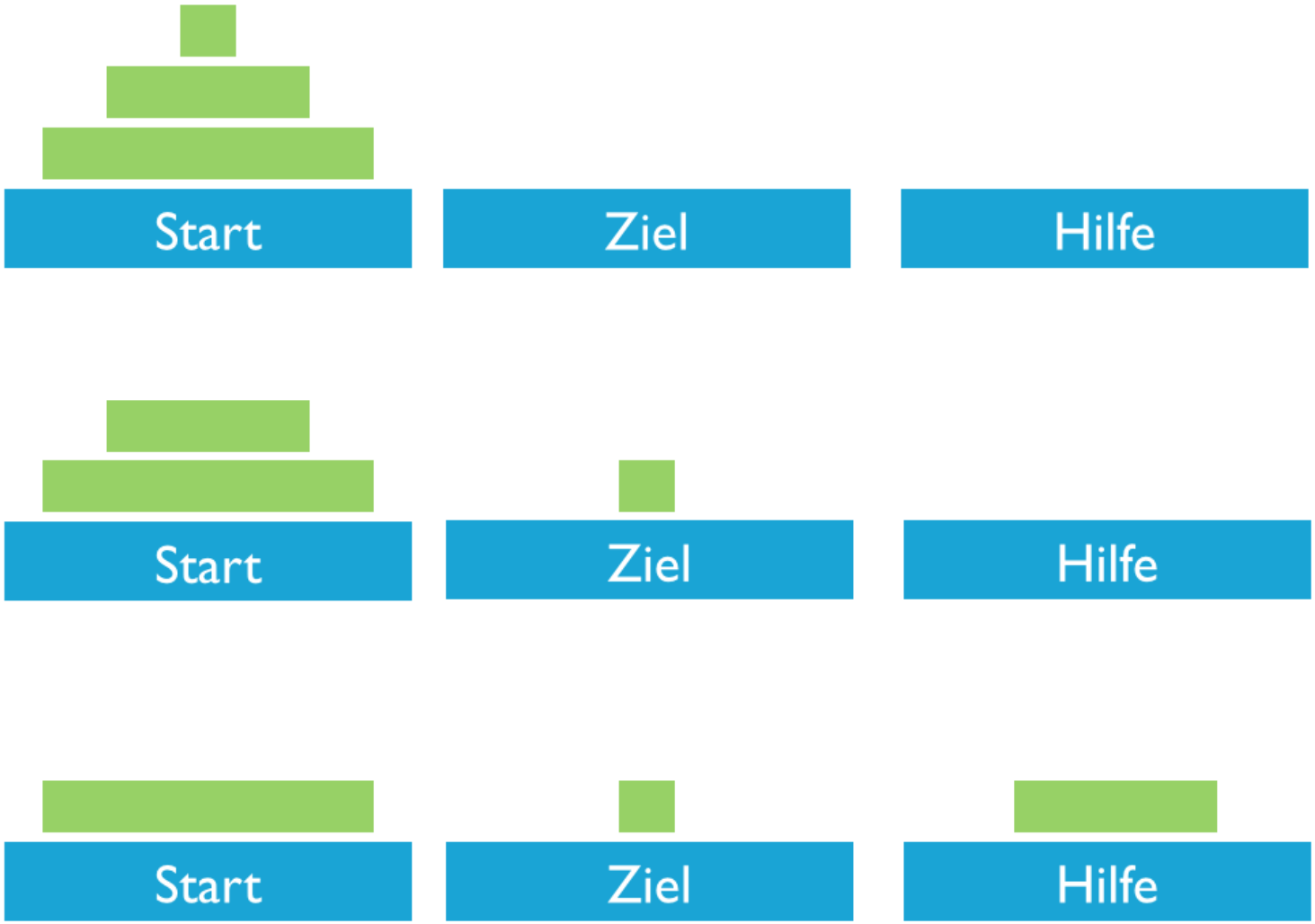
Aufgaben - Türme von Hanoi

Lösen Sie das Problem der Türme von Hanoi rekursiv.

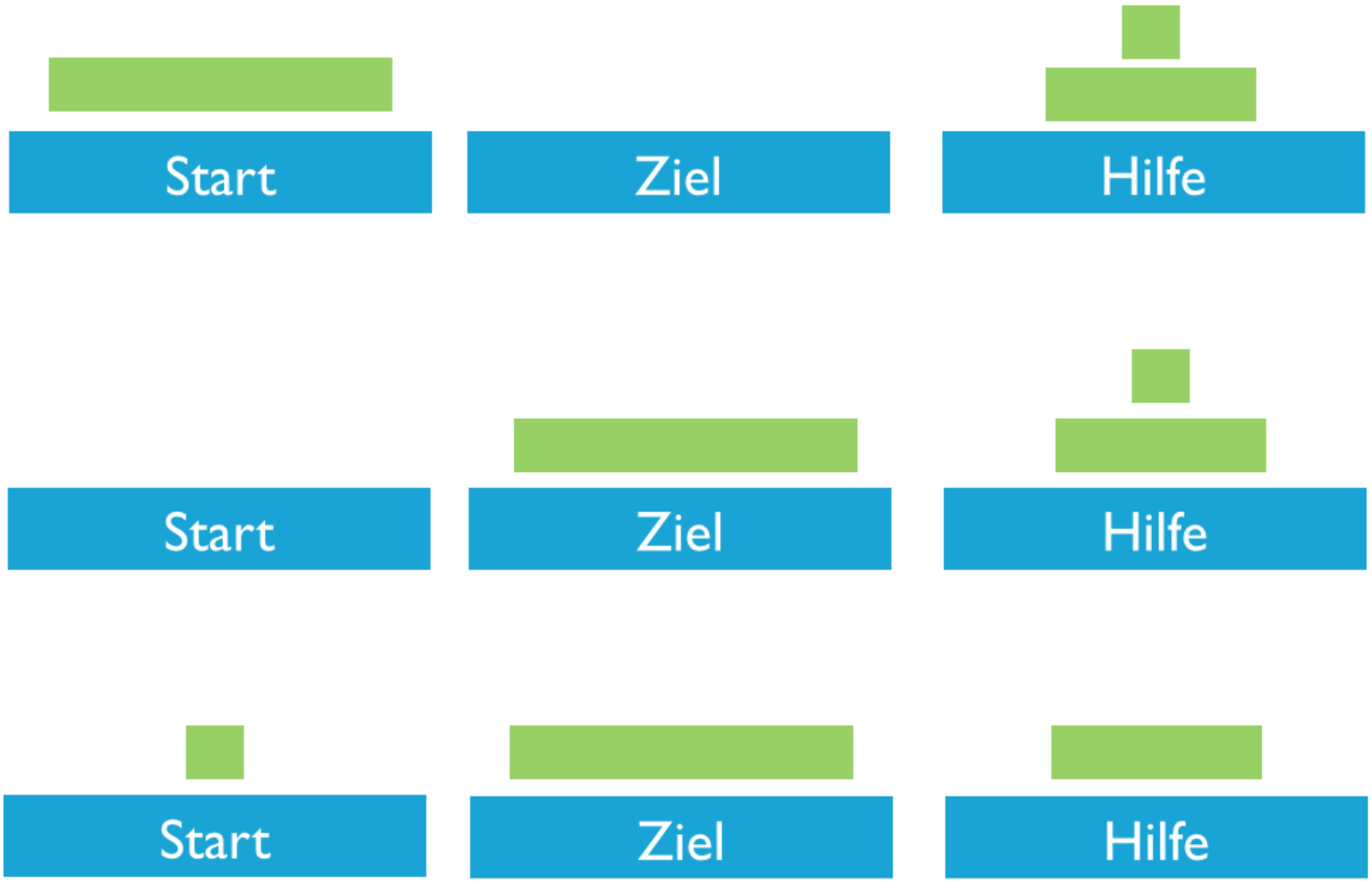
1. n Scheiben unterschiedlichen Durchmessers, die der Größe nach sortiert übereinander liegen, bilden mit der größten Scheibe unten einen Turm. Der Turm soll von einem Platz 1 zu einem Platz 2 transportiert werden.
2. Dabei steht ein Hilfsplatz 3 zur Verfügung.
3. Es darf jeweils nur die oberste Scheibe eines Turms bewegt werden.
4. Außerdem darf auf eine Scheibe nur eine kleinere Scheibe gelegt werden

Implementieren Sie dazu die Funktion:
static void bewegeTurm(int n, int s, int z, int h)
Sie soll die notwendigen Scheibenbewegungen ausgeben, um einen Turm mit n Scheiben vom Startplatz s zum Zielplatz z zu bewegen. Dazu existiert ein zusätzlicher Hilfsplatz h.

Aufgaben - Türme von Hanoi



Aufgaben - Türme von Hanoi



Aufgaben - Türme von Hanoi

Strings

Strings

- Strings sind Objekte (OOP)
- Können ohne "new" angelegt werden
- Konstruktoren existieren auch (OOP)
- Strings sind immutable (nicht veränderbar)
- Ändern eines Zeichens erzeugt neuen String
- Vergleich mit equals-Methode

```
// Erstellen ohne ''new''
String farbe = "rot";
String farbe2 = "blau";

   // Klasse hat viele Methoden
   // Hier 3 Beispiele,
   // mehr sind in der API-Doku

// Gibt die Laenge zurueck
int laenge = farbe.length();

// Gibt das Zeichen
// an Position 2
// zurueck
// Wichtig: 1. Zeichen
// steht an Position 0
char c = farbe.charAt(2);

// Ein Vergleich
// Ergebnis: false
farbe.equals(farbe2);

// Noch ein Vergleich
// Ergebnis: true
"blau".equals(farbe2);
```

Strings

- "+"-Operator fügt Strings zusammen
- Andere Datentypen werden beim Zusammenfügen automatisch in Strings umgewandelt

```
// String concat
String a = "rot";
String b = "blau";
String c = "gruen";
System.out.println(a+b+c);

// Umwandlung in String
int i = 10;
String prefix = "Geld: ";
String postfix = "Euro";
String message = prefix + i +
     postfix;
System.out.println(message);
```

Aufgaben

1. Wandeln Sie einen übergebenen String in Großbuchstaben um.
   (Tipp: Schauen Sie in der Java-Doku)
2. Entwickeln Sie die Methode "replace(String toReplace, String replacement, String original)" die die Zeichenkette "toReplace" in "original" durch die Zeichenkette "replacement" ersetzt.
3. Sie bekommen eine aus Ganzzahlen bestehende Zeichenkette übergeben, die durch Semikolon separiert sind. Berechnen Sie die Summe aus den Zahlen in der Zeichenkette und geben Sie diese aus.

Enumerations

Enumerations

- Enumerations sind Aufzählungsobjekte (OOP).
- Sie ermöglichen auf einfache und sichere Art und Weise die Realisierung von Aufzählungen, als Alternative zu Konstanten.
- Enums lassen sich erweitern, diese Funktionalität ist jedoch nicht Teil der Veranstaltung.

```
enum Wochentag {
MONTAG,
DIENSTAG,
MITTWOCH,
DONNERSTAG,
FREITAG,
SAMSTAG,
SONNTAG
}
```

Strukturierte Programmierung

Die 4 Regeln der strukturierten Programmierung

1. Top-Down Ansatz
2. Trennung von Steuerung und Verarbeitung
3. Jeder Block hat nur einen Anfangs- und einen Endpunkt
4. Bildung mehrfach verwendbarer Blöcke

Top-Down Ansatz

- Beginn der Überlegungen bei prinzipieller Aufgabe des Programms
- Konkrete Fragen der Verarbeitung werden durch schrittweise Verfeinerung später betrachtet

Trennung von Steuerung und Verarbeitung

- Bildung von steuernden oder verarbeitenden Blöcken
- Hauptprogramm ist wichtigster steuernder Block
- Verarbeitung wird komplett in Unterprogramm-Blöcke verlagert

Jeder Block hat nur einen Anfangs- und einen Endpunkt

- Aufteilung des Programms in Blöcke, die genau einen Anfangs- und einen Endpunkt haben
- Verringerung der maximalen Größe zusammenhängender Programmstücke auf übersichtliche Größe

Bildung mehrfach verwendbarer Blöcke

- Bildung von mehrfach verwendbaren Blöcken
- Diese Blöcke werden als Unterprogramme realisiert

Ein- und Ausgabe mit Java-IO

Die Ein- und Ausgabe wird in Java mit Streams gelöst.

Dabei können als Datenquellen bzw. -Senken

- lokale Dateien (Quelle und Senke)
- Dateien im Internet (Quelle und Senke)
- der Bildschirm (Senke)
- oder auch die Tastatur (Quelle)

verwendet werden.

Stream

Ein Stream ist eine geordnete Folge von Bytes (Bytestrom).

- Kommt der Stream aus einer Datenquelle heißt er "InputStream".
- Mündet er in einer Datensenke wird er "OutputStream" genannt.
- Streams sind im Package "java.io" zusammengefasst.

Streams vs. Reader/Writer

- Streams arbeiten grundsätzlich byte-orientiert.
- Reader und Writer erweitern die Funktionalität von Streams und ermöglichen eine zeichen-orientierte Verarbeitung

Input

Dateien als Datenquellen

- FileInputStream dient zum Einlesen von Dateien
- Arbeitet mit den Methoden der Klasse "InputStream"
- read() liest ein byte, dass als int von 0 bis 255 zurückgeliefert wird
- read(byte[] b) liest je nach Arraygröße mehrere Bytes
- Nach Benutzung wird der Stream mit close() geschlossen um alle Ressourcen wieder freizugeben

Dateien als Datenquellen

```
public static void main(String[] args) throws IOException {
// Deklarieren und initialisieren des Streams
FileInputStream fis = new FileInputStream("resource/TestDatei.txt");

// Unsere Ausgabe
String ausgabe = "";

// Solange ''r'' nicht -1 ist,
// ist noch ungelesener Inhalt in der Datei
byte[] b = new byte[1];
while (fis.read(b) != -1) {
// Aktuell gelesenes Byte der Ausgabe anhaengen
ausgabe += (char) (b[0]);
}
System.out.println(ausgabe);

// Stream schliessen
fis.close();
}
```

Dateien als Datenquellen

- Reader arbeiten zeichenorientiert
- zusätzlich zu read() besitzen sie read(char[] b)
- der BufferedReader enthält soger: readLine()
- readLine() liest die Datei Zeilenweise ein und gibt am Ende der Datei "null" zurück

- Reader werden wie folgt instanziiert:

```
InputStreamReader isr = new InputStreamReader(new
FileInputStream("test.txt"));
```

- Und der BufferedReader:

```
BufferedReader b = new BufferedReader(new InputStreamReader(new
      FileInputStream("test.txt")));
```

- Alternativ kann der Reader auch mit Hilfe der Klasse FileReader angelegt werden:

```
BufferedReader b = new BufferedReader(new FileReader("test.txt"));
```

Dateien als Datenquellen

```
public static void main(String[] args) throws IOException {
// Deklarieren und initialisieren des Streams
FileInputStream fis = new FileInputStream("resource/TestDatei.txt");

// Deklarieren und initialisieren des BufferedReaders
BufferedReader br = new BufferedReader(new InputStreamReader(fis));

// Unsere Ausgabe
String ausgabe = "";

// solange ''zeile'' nicht null ist
String zeile = null;
while ((zeile = br.readLine()) != null) {
// Aktuell gelesenes Byte der Ausgabe anhaengen
ausgabe += zeile;
}
System.out.println(ausgabe);

// Stream schliessen
br.close();
}
```

Java Stanardeingabe

Java stellt bestimmte Standard-Eingaben zur Verfügung.

- Eingabe über statisches Attribut "in" der Klasse System
- "in" ist Referenz auf Objekt vom Typ "InputStream"
- Zur Eingabe unter der Konsole

Aufgaben

1. Die Datei "stars.txt" enthält 20 Zeilen und in jeder Zeile ist eine bestimmte Anzahl an * sowie Zahlen. Geben Sie die Anzahl der Sterne sowie die Summe der in der Datei enthaltenen Zahlen, je Zeile mit jeweiligen Zeilennummern, aus.
2. Bisher verwendeten Sie die Klasse "Tastatur" zur Eingabe von Informationen auf der Konsole. Schreiben Sie diese Klasse selbst, mithilfe des Standard-Inputstreams "System.in". Ermöglichen Sie dabei das Lesen von Strings, Integer und Floats.

Output

Dateien als Datensenken

- FileOutputStream dient zum Schreiben in Dateien
- Arbeitet mit den Methoden der Klasse "OutputStream"
- write(int b) schreibt ein byte
- flush() schreibt gepufferte Daten
- Nach Benutzung wird der Stream mit close() geschlossen um alle Ressourcen wieder freizugeben

Dateien als Datenksenken

```
public static void main(String[] args) throws IOException {
// Deklarieren und initialisieren des Streams
FileOutputStream fos = new FileOutputStream("resource/TestDatei.txt");
fos.write('H');
fos.write('e');
fos.write('l');
fos.write('l');
fos.write('o');

// Puffer schreiben
fos.flush();

// Schliessen des OutputStreams
fos.close();
}
```

Dateien als Datenksenken

- Writer arbeiten zeichenorientiert
- zusätzlich zu write() besitzen sie write(char[] b) und write(String s)
- der PrintWriter enthält soger: println()
- println() schreibt zeilenweise in die Datei

Dateien als Datenksenken

- Writer werden wie folgt instanziiert:

```
   OutputStreamWriter osw = new
   OutputStreamWriter(new FileOutputStream("resource/TestDatei.txt
        "));
```

- Und der PrintWriter:

```
   PrintWriter pw = new PrintWriter(new OutputStreamWriter(
new FileOutputStream("resource/TestDatei.txt")));
```

- Alternativ kann der Writer auch mit Hilfe der Klasse FileWriter angelegt werden:

```
PrintWriter pw = new PrintWriter( new FileWriter("resource/
     TestDatei.txt"));
```

Dateien als Datenksenken

```
public static void main(String[] args) throws IOException {
// Deklarieren und initialisieren des Streams
FileOutputStream fos = new FileOutputStream("resource/TestDatei.txt");

// Deklarieren und initialisieren des Writers
PrintWriter pw = new PrintWriter(new OutputStreamWriter(fos));

// Zeile schreiben
pw.println("Eine Testzeile.");

// Puffer schreiben
pw.flush();

// schliessen des Writers
pw.close();
}
```

Java Standardausgabe

Java stellt bestimmte Standard-Ausgaben zur Verfügung.

- Ausgabe über statisches Attribut "out" der Klasse System
- "out" ist Referenz auf Objekt vom Typ "PrintStream"
- Zur Ausgabe auf die Konsole

Aufgaben

1. Schreiben Sie die Ergebnisse der Verarbeitung der Datei "stars.txt" in die Datei "starsgezaehlt.txt". Schreiben sie dabei zu Beginn der Zeile die Zeilennummer, anschließend durch "," getrennt die Anzahl der Sterne, sowie die Summe der Zahlen. Dahinter folgt der eigentliche Datei-Inhalt je Zeile.

Parallele Programmierung mit Java-Threads

Definition eines Threads

Ein Thread ist ein Programmstück, dass parallel zu anderen Programmstücken ausgeführt wird.

Beispiele für parallel auszuführende Programmstücke sind

- Benutzerinteraktionen
- komplexe Berechnungen
- ...

Auf Einprozessorsystemen werden Threads mittels Zeitscheiben und möglichst häufigen Wechseln realisiert. (Nicht wirklich parallel)

Threads erzeugen

Threads erzeugen

- Threads sind einfache Java-Klassen (java.lang.Thread)
- Können auf zwei Arten erzeugt werden:
  1. Unterklasse (OOP)
  2. Runnable implementieren (OOP)
- In der Methode run() steht der parallel zu verarbeitende Code
- Gestartet wird der Thread durch start()

Threads erzeugen durch ableiten von "Thread"

- MeinThread erbt von Thread (OOP)
- run() wird überschrieben (OOP)
- in Main-Methode wird start() aufgerufen, nicht run()

```
public class MeinThread extends
      Thread {
public void run() {
System.out.println("MeinThread");
}
}
//...
public static void main(String[]
      args) throws IOException {
MeinThread t = new MeinThread();
t.start();
}
```

Threads erzeugen durch implementieren von "Runnable"

- MeinThreadRunner implementiert Runnable(OOP)
- run() wird überschrieben (OOP)
- In Main-Methode wird Instanz von Thread erzeugt (OOP) und das implementierte Runnable übergeben
- Anschließend wird start() auf den Thread aufgerufen, nicht run()

```
public class MeinThreadRunner
      implements Runnable {
public void run() {
System.out.println("MeinThreadRunner
      ");
}
}
//...
public static void main(String[]
      args) throws IOException {
MeinThreadRunner runner = new
      MeinThreadRunner();
Thread t = new Thread(runner);
t.start();
}
```

Threads erzeugen

- Threads laufen unabhängig voneinander
- Gefahr bei gemeinsamen Ressourcen
- Main läuft in eigenem Thread
- Main-Thread wird von Laufzeitsystem erzeugt
- Programm terminiert wenn der letzte Thread terminiert
- Thread terminiert wenn die run()-Methode durchlaufen wurde

Die Methode sleep(long millis)

- sleep() bewirkt, dass der aufrufende Thread für die übergebene Zeit schlafen gelegt wird
- schlafende Threads verbrauchen keine Rechenleistung

Asynchrone Aufträge mit join()

- join()-Methode wird verwendet um auf Thread-Ende zu warten
- Beispiel: Aufgabenverteilung auf mehrere Threads und anschließendes Zusammenfügen der Ergebnisse

Aufgaben

1. Zählen Sie in einem Array vom Typ "boolean" alle Felder deren Wert "true" ist. Verteilen Sie diese Aufgabe auf eine bestimmte Anzahl von Zähler-Threads.
   - Das Array besitzt 200000000 Felder
   - Jeder Thread zählt in einem bestimmten Bereich des Arrays
   - Testen Sie die Anwendung ohne, mit 2, 10 und 100 Zähler-Threads
   - Wie wirkt sich die Simulation einer komplexen Berechnung in Form von sleep() auf die ansynchrone Beauftragung aus?

Synchronisation und gegenseitiger Ausschluss

Synchronisation

- Das Schlüsselwort "synchronized" dient als Sperre für Ressourcen
- Sperrt Blöcke indem bestimmtes Objekt gelocked wird (OOP)
- Kann Zugriff auf alle Methoden eines Objekts locken (locken der eigenen Instanz)
- synchronized führt Informationen von einem konsistenten Zustand in einen anderen konsistenten Zustand

Beispiel für synchronize

- Gegeben sind zwei Threads: zaehler1 und zaehler2
- Beide Threads erhöhen den Wert eines gemeinsamen Integers

Beispiel für synchronize

```
// Die run()-Methode unserer Runnable-Implementierung
public void run() {
while (true) {
MainMitInt.inc();
}
}
// ..................................................
public class MainMitInt {
// gemeinsamer Integer
    static int zahl = 0;
    //gemeinsam genutzte Methode
public static void inc(){
int neueZahl = zahl + 1;
    zahl = neueZahl;
}
// Main-Methode
    public static void main(String[] args) {
// Zaehler-Runnable instanziieren
Thread zaehler1 = new Thread(new Zaehler());
Thread zaehler2 = new Thread(new Zaehler());

// Threads starten
zaehler1.start();
zaehler2.start();

// Ausgabe
while (true) {
    System.out.println(zahl);
}
    }
}
```

Beispiel für synchronize

Problemszenario bei dieser Implementierung:

1. zaehler1 ruft inc() auf, berechnet neueZahl und lädt Ergebnis (z.B. 5) in zahl
2. zaehler1 wird suspendiert, zaehler2 wird ausgeführt, führt inc() aus und berechnet neueZahl ($5 + 1 = 6$)
3. zaehler2 wird suspendiert, zaehler1 wird fortgesetzt, berechnet in inc() neueZahl ($5 + 1 = 6$) und schreibt 6 in Variable zahl
4. zaehler1 wird suspendiert, zaehler2 wird fortgesetzt und schreibt in pausiertes inc() neueZahl($= 6$) in Variable zahl

Das Problem

Die gemeinsam genutzte Variable zahl ist weiterhin 6 obwohl sie 7 sein sollte!

Anwendung von synchronize

- Gemeinsam genutze Variable darf immer nur von einem Thread gleichzeitig benutzt werden
- Sichergestellt durch synchronized im Methodenkopf
- Wird auch als "Gegenseitiger Ausschluss" bezeichnet
- Methode kann somit nur von einem Thread gleichzeitig aufgerufen werden
- Andere Threads werden blockiert bis aktueller Thread fertig ist
- Besitzt Objekt weitere synchronized-Methoden, so sind sie auch gelocked

```
// gemeinsam genutzte
      Methode
// versehen mit
     synchronized
public static
     synchronized
     void inc(){
int neueZahl = zahl +
      1;
    zahl = neueZahl;
}
```

Wann synchronize verwenden?

Wann sollte synchronized verwendet werden:

Synchronisation ist immer nur dann notwendig, wenn mehrere Threads auf gemeinsame Daten zugreifen und mindestens einer dieser Threads die Daten verändert. In diesem Fall ist es wichtig, alle Methoden, die auf die Daten zugreifen, als synchronized zu kennzeichnen, gleichgültig, ob die Daten in einer Methode nur gelesen oder auch geändert werden.

Aufgaben

1. Entwerfen Sie ein Bankkonto das Thread-Safe ist
   - Das Konto besitzt die Methode buchen(int betrag)
   - In der Methode wird der übergebene Betrag auf den aktuellen Kontostand aufaddiert
   - Mehrere "Einzahler" können die Methode aufrufen und Geld einzahlen

wait() und notify()

Nebenbedingungen

Bedingung fuer Ausführung einer Methode wird erweitert:

- zusätzlich zu konsistenten Zustand
- wird auf Erfüllung von Nebenbedingung gewartet
- Thread soll mit Methodenausführung warten, bis diese Bedingungen erfüllt sind

Beispielszenario: Parkhaus

Parkhaus das Anzahl noch freier Parkplätze managed:

- Verschiedene Threads können Parkhaus passieren() und verlassen()
- Bei Einfahrt wird Anzahl freier Plätze vermindert (0 wenn keiner mehr frei, Parkhaus voll, keine Einfahrt möglich)
- Bei Ausfahrt wird Anzahl freier Platze erhöht
- Da Parkhaus von mehreren Auto-Threads genutzt, und zustand geändert werden kann, müssen einfahren() und passieren() synchronized sein

Problem:

Wie wird das Warten auf einen freien Platz gelöst?

Wait-Notify zur Lösung des Problems

Zur Lösung werden die Methoden wait() und notify() verwendet

- wait() blockiert aufrufenden Thread und packt ihn in Warteschlange des Objekts auf das wait() aufgerufen wird
- wait() gibt locks auf Objekt frei
- notify() entfernt Thread aus der Warteschlange
- notify() hat auf leere Warteschlange keine Wirkung
- keine Garantie das der Thread der am längsten wartet, geweckt wird

Aufgaben

1. Implementieren Sie die Parkhaus-Simulation
   - Entwerfen Sie die Klasse Parkhaus (OOP)
   - Implementieren Sie die Methoden passieren() und verlassen()
   - Implementieren Sie die Auto-Threads die nach und nach das Parkhaus verlassen und passieren
   - Die Auto-Threads rufen dazu jeweils passieren() oder verlassen() auf dem Parkaus auf

Semaphoren

Definition

Eine Semaphore ist ein Verwaltungsobjekt, dass den Zugriff mehrerer Threads/Prozesse auf eine gemeinsame Ressource kontrolliert.

- Parkhaus entspricht der Struktur einer Semaphore
- Semaphoren dienen der Verwaltung beschränkter Ressourcen auf mehrere Prozesse
- Bei Verwendung besitzt jeder Thread die Semaphore als Attribut
- Verwendet für gegenseitigen Ausschluss

Semaphore als Quelltext

```
public class Semaphore {
private int value = 0;

// Konstruktor (OOP)
public Semaphore(int val){
if(val > 0){
value = val;
}
}

// passieren() aus dem Parkhaus
public synchronized void p(){
while(value == 0){
try{
wait();
}catch(InterruptedException e){
// left blank
}
}
value--;
}

// Verlassen aus dem Parkhaus
public synchronized void v(){
value++;
notify();
}
}
```

Anwendung der Semaphore

- mit p() kann Thread kritischen Bereich betreten
- mit v() kann Thread kritischen Bereich verlassen
- value gibt an wie viele Threads maximal gleichzeitg den kritischen Bereich betreten dürfen

Anwendungsbeispiel

```
// Semaphore die anderweitig bereits initialisiert wurde
Semaphore s;

// Unsere run-Methode
public void run() {
while (true) {
// Wir betreten einen kritischen Bereich
s.p();

this.doCriticalStuff();

// Wir verlassen den kritischen Bereich
s.v();
}
}
```

Deadlocks

Deadlocks

Ein Deadlock ist eine Situation, in der sich zwei oder mehr Threads in einem dauernden Wartezustand befinden.

Situation

Zwei Köche, benötigen Schüssel und Löffel zum Kochen

- Koch 1 nimmt zuerst Löffel und dann Schüssel
- Koch 2 nimmt zuerst Schüssel und dann Löffel

Szenario

Folgendes Szenario entsteht:

1. Koch 1 nimmt Löffel, Thread wird unterbrochen
2. Koch 2 nimmt Schüssel, Thread wird unterbrochen
3. Koch 1 wartet darauf das Schüssel frei wird
4. Koch 2 wartet darauf das Löffel frei wird

Das Problem:

Deadlock!

Die vier Bedingungen für Deadlocks

1. Wenn Ressource nur unter Ausschluss nutzbar
2. Genutzte Ressourcen können nutzendem Thread nicht entzogen werden
3. Threads besitzen Ressourcen und fordern weitere an
4. Es existiert zyklische Kette von Threads, von denen jeder mindestens eine Ressource besitzt, die der nächste Thread in der Kette benötigt

Vermeidung von Deadlocks

- Thread darf nur Ressourcen anfordern, wenn er keine Besitzt
- Thread forder Ressourcen in bestimmter Reihenfolge an um zyklische Abhängigkeiten zu verhindern
- Prüfung ob es bei Ressourcenanforderung zu einem Deadlock kommen kann