Tafelübung 02
Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2016/2017

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Übersicht

Zur Erinnerung

Wichtig!

Bei der Bearbeitung der Praxisaufgaben dürfen die gegebenen Schnittstellen nicht geändert werden (d.h. Funktionsnamen, Parameter, Rückgabetypen, Modifizierer, etc.).

Wenn die Aufgabe das Anlegen einer bestimmten Variable fordert, dann muss diese genauso heißen wie in der Aufgabenstellung verlangt.

Andernfalls gibt es Probleme mit den automatischen Tests $\rightsquigarrow$ 0 Punkte ☹

Java-Grundlagen

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Datentypen

- Daten werden intern als Folgen von 0en und 1en gespeichert
  - *eine* 0 oder 1 wird dabei als Bit (*Binary Digit*) bezeichnet
  - eine Folge von 8 Bits wird als Byte bezeichnet
- der Datentyp einer Variable bestimmt...
  - die Größe der Variable im Speicher (Anzahl der Bits),
  - wie die gespeicherte Information zu interpretieren ist, und
  - welche Operationen auf der Variable möglich sind
- mögliche Datentypen:
  - vordefinierte Datentypen (primitive Datentypen, ...)
  - benutzerdefinierte Datentypen

Primitive Datentypen in Java (I)

- Ganzzahlen:

Typ	Größe	Wertebereich
`byte`	1 Byte	$[-128, 127]$
`short`	2 Byte	$[-32.768, 32.767]$
`int`	4 Byte	$[-2.147.483.648, 2.147.483.647]$
`long`	8 Byte	$[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]$

- Fließkommazahlen:
  - Konstanten verwenden Punkt als Dezimaltrennzeichen (z.B. `13.03`)

Typ	Größe	Genauigkeit	Wertebereich (mit Lücken)
`float`	4 Byte	einfach	$\approx \pm 3.4 \cdot 10^{38}$
`double`	8 Byte	doppelt	$\approx \pm 1.8 \cdot 10^{308}$

Primitive Datentypen in Java (II)

- Wahrheitswerte („wahr“ oder „falsch“):

Typ	Größe	Wertebereich
`boolean`	1 Byte	[`true`, `false`]

- (Unicode-)Zeichen (Buchstaben, ...):

Typ	Größe	Wertebereich	Beispiel
`char`	2 Byte	[0, 65.535]	`'A'`

Arithmetische Operatoren

- für Berechnungen mit „Zahlen“ bietet Java eine Menge von Operatoren:

Operator	Bedeutung	Beispiel	Ergebnis
`+`	Addition	`13 + 3`	`16`
`-`	Subtraktion	`12 - 4`	`8`
`*`	Multiplikation	`4 * 5`	`20`
`/`	Division	`8 / 2`	`4`
`%`	Modulo (Rest)	`9 % 5`	`4`

Achtung: Ganzzahl-Division

Bei einer Division von ganzen Zahlen (z.B. `int`) führt Java eine Ganzzahl-Division durch, d.h. eventuelle Nachkommastellen werden abgeschnitten ($\leadsto$ `13 / 3` = 4).

Arithmetische Operatoren: Auswertungsreihenfolge

- wie in der Mathematik beachtet Java bei Berechnungen Punkt-vor-Strich
  1. „Punkt“-Operatoren: `*`, `/`, `%`
  2. „Strich“-Operatoren: `+`, `-`
- die Auswertungsreihenfolge kann durch Klammerung beeinflusst werden

Beispiele

Ausdruck	Ergebnis
`13 + 3 * 4`	25
`(13 + 3) * 4`	64

Kurz-Schreibweise: Zuweisungen mit Operation

- die Grundoperatoren können mit einer Zuweisung kombiniert werden:

Kurz-Schreibweise	entspricht
`i += 5;`	`i = i + 5;`
`i -= 4;`	`i = i - 4;`
`i *= 8;`	`i = i * 8;`
`i /= 2;`	`i = i / 2;`
`i %= 6;`	`i = i % 6;`

Achtung

Es wird immer zuerst die rechte Seite vollständig ausgewertet, also:

Kurz-Schreibweise	entspricht
`i *= 5 + 4;`	`i = i * (5 + 4);`

Prä-/Post-Inkrement, Prä-/Post-Dekrement

- innerhalb eines beliebigen Ausdrucks kann Variable a verändert werden:
  - `++a`: Prä-Inkrement
    - a wird *zuerst* um 1 erhöht und es wird mit dem *neuen* Wert gerechnet
  - `a++`: Post-Inkrement
    - es wird mit dem *alten* Wert gerechnet und a *anschließend* um 1 erhöht
  - `--a`: Prä-Dekrement
    - a wird *zuerst* um 1 verringert und es wird mit dem *neuen* Wert gerechnet
  - `a--`: Post-Dekrement
    - es wird mit dem *alten* Wert gerechnet und a *anschließend* um 1 verringert
- normalerweise stehen Inkremente / Dekremente aber „alleine“: `a++;`

Prä-/Post-Inkrement, Prä-/Post-Dekrement: Beispiel

Beispiel: Prä-Inkrement

```
int a = 3;
int b = 5 * ++a;

System.out.println("a = " + a);
System.out.println("b = " + b);
```

Beispiel: Post-Inkrement

```
int a = 3;
int b = 5 * a++;

System.out.println("a = " + a);
System.out.println("b = " + b);
```

Ausgabe

```
a = 4
b = 20
```

Ausgabe

```
a = 4
b = 15
```

Vorsicht

Inkrement ist signifikant anders als:

```
int b = 5 * (a + 1); // a bleibt unverändert (meist gewünscht)
```

Vergleiche von Zahlen

- für die numerischen Datentypen gibt es die folgenden Vergleichsoperatoren:
  - das Ergebnis ist stets ein `boolean`-Wert

Operator	Bedeutung	Beispiel	Ergebnis
`==`	gleich	`13 == 3`	`false`
`!=`	ungleich	`13 != 3`	`true`
`<`	kleiner	`13 < 3`	`false`
`>`	größer	`13 > 3`	`true`
`<=`	kleiner-gleich	`13 <= 3`	`false`
`>=`	größer-gleich	`13 >= 3`	`true`

Achtung

Zuweisung mit `=`, Vergleich mit `==`.

Beispiel zum Vergleichen von Zahlen

Frage

Handelt es sich bei einem Rechteck mit gegebenen Kantenlängen um ein Quadrat?

Lösung

Ein Rechteck ist ein Quadrat, wenn seine Breite und seine Höhe gleich sind.

In Java

```
int breite = 5;
int hoehe = 6;

boolean istQuadrat = (breite == hoehe); // = false
```

Logische Operatoren

- auch für Wahrheitswerte (`boolean`) bietet Java eine Menge von Operatoren
  - verknüpfen zwei boolesche Werte zu einem neuen booleschen Wert

⇝ komplexe/zusammengesetzte boolesche Ausdrücke

Beispiel 1: Logisches „Und“

Die Verwendung des Systems wird nur gewährt, wenn...

- der Benutzername stimmt **UND**
- das Passwort stimmt.

Beispiel 2: Logisches „Oder“

Eine Ware kann nur gekauft werden, wenn...

- genügend Bargeld vorhanden ist **ODER**
- genügend Geld auf dem Konto vorhanden ist.

Logisches „Und“

- logisches „Und“ $\rightsquigarrow$ &&-Operator
  - *wahr*, wenn beide Operanden *wahr* sind
  - *falsch*, wenn mindestens ein Operand *falsch* ist

Wahrheitstabelle

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

Beispiel

```
boolean verwendungErlaubt = (benutzernameKorrekt && passwortKorrekt);
```

Logisches „Oder“

- logisches „Oder“ $\rightsquigarrow$ ||-Operator
  - *wahr*, wenn mindestens ein Operand *wahr* ist
  - *falsch*, wenn beide Operanden *falsch* sind

Wahrheitstabelle

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

Beispiel

```
boolean kaufMoeglich = (genuegendBargeld || genuegendAufDemKonto);
```

„Exklusives Oder“

- „exklusives Oder“ $\leadsto$ $\wedge$-Operator
  - *wahr*, wenn *genau* ein Operand *wahr* ist
  - *falsch*, wenn *beide* Operanden *wahr* oder *falsch* sind

Wahrheitstabelle

A	B	A $\wedge$ B
false	false	false
false	true	true
true	false	true
true	true	false

Logische Negation

- logische Negation $\rightsquigarrow$!-Operator
  - einstelliger Operator, d.h. nur ein Operand
  - „dreht den Wahrheitswert um“

Wahrheitstabelle

A	!A
false	true
true	false

Beispiel

```
boolean istKeinQuadrat = !(istQuadrat);
```

Logische Operatoren: Auswertungsreihenfolge (I)

- Auswertungsreihenfolge:
  1. „Punkt“-Operatoren: `*`, `/`, `%`
  2. „Strich“-Operatoren: `+`, `-`
  3. Vergleichs-Operatoren: `==`, `!=`, `<`, `<=`, `>`, `>=`
  4. logisches „Und“: `&&`
  5. logisches „Oder“: `||`

- mit Klammern kann die Auswertungsreihenfolge beeinflusst werden
  - lieber „zu viele“ Klammern verwenden als „zu wenige“...

Logische Operatoren: Auswertungsreihenfolge (II)

Natürlichsprachliche Anforderung

Eine Ware kann nur dann gekauft werden, wenn diese auf Lager ist *und* wenn genügend Bargeld *oder* genügend Geld auf dem Konto vorhanden ist.

Falsch (Wieso?)

```
boolean kaufMoeglich =
          aufLager && genuegendBargeld || genuegendAufDemKonto;
```

Richtig

```
boolean kaufMoeglich =
          aufLager && (genuegendBargeld || genuegendAufDemKonto);
```

Logische Operatoren: Kurzschlusssemantik

- Kurzschlusssemantik:
  - && und || brechen die Auswertung ab, sobald das Ergebnis feststeht
  - macht insb. dann einen Unterschied, wenn Operanden Seiteneffekte haben

Beispiel

```
boolean a = false;
boolean b = true;
boolean c = a && b; // da a schon false ist, muss b gar nicht
                    // angeschaut werden
                    // schon vorher klar, dass auch c false wird
boolean d = b || a; // da b schon true ist, muss a gar nicht
                    // angeschaut werden
                    // schon vorher klar, dass auch d true wird
```

Kurzschlusssemantik vermeiden

Will man die Kurzschlusssemantik vermeiden, so muss man & bzw. | verwenden.

Typkonvertierung

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Typkonvertierung

- **bekannt**: in Java gibt es unterschiedliche Datentypen
  - der Typ bestimmt die Repräsentation und Größe eines Datums im Speicher
  - unterschiedliche Datentypen haben unterschiedliche Wertebereiche

- **jetzt**: Typkonvertierung (*cast*): Überführung eines Wertes eines Datentyps in den entsprechenden Wert eines anderen Datentyps
  - von „kleinerem“ zu „größerem“ Typ: Typerweiterung
  - von „größerem“ zu „kleinerem“ Typ: Typeinschränkung

- man unterscheidet zwei Arten der Typkonvertierung:
  - implizite Typkonvertierung („durch den Compiler“)
  - explizite Typkonvertierung („durch den Programmierer“)

Implizite Typkonvertierung

- implizite Typkonvertierung:
  - nicht direkt aus dem Quelltext des Programms ersichtlich
  - automatisch durch den Compiler an den notwendigen Stellen eingefügt
  - Voraussetzung: durch die Konvertierung entsteht kein Informationsverlust

  $\rightsquigarrow$ implizite Typkonvertierung nur im Falle einer Typerweiterung

Beispiele

```
int a = 13;
double b = a;  // implizite Typkonvertierung von int nach double

double pi = 3.141f; // implizite Typkonvertierung
// von float nach double ('f' kennzeichnet float-Wert)
```

Explizite Typkonvertierung

- Was ist, wenn der Ziel-Datentyp „kleiner“ als der Quell-Datentyp ist?

$\rightsquigarrow$ Typeinschränkung $\rightsquigarrow$ Informationsverlust!

$\rightsquigarrow$ nur mittels expliziter Typkonvertierung durch den Programmierer

So nicht...

```
double a = 9.8;
int b = a;          // 'possible loss of precision'
float pi = 3.141; // 'possible loss of precision'
```

So schon ☺

```
double a  = 9.8;
int b = (int) a;            // b=9, Nachkommastellen abgeschnitten!
float pi = (float) 3.141; // besser: 3.141f (siehe oben)
```

Achtung... (I)

Möglicherweise unerwartetes Ergebnis

```
int a = 13;
double b = a / 2;   // b = 6.0
```

Was ist denn hier passiert?

- der Compiler „sieht“ bei der Division zwei ganzzahlige Operanden
  - a als `int` deklariert $\rightsquigarrow$ ganzzahlig
  - 2 ist `int`-Wert $\rightsquigarrow$ ganzzahlig

$\rightsquigarrow$ es wird eine Ganzzahl-Division durchgeführt

  - Ergebnis ist der `int`-Wert 6

- erst danach „sieht“ der Compiler, dass in eine `double`-Variable geschrieben wird
  - implizite Typkonvertierung von `int` nach `double`
  - in diesem Moment sind die Nachkommastellen aber schon verloren ☹

Achtung... (II)

Wahrscheinlich erwünschtes Ergebnis

```
int a = 13;
double b = a / 2.0;  // b = 6.5

// oder

double b = ((double) a) / 2;  // b = 6.5
```

Warum funktioniert das jetzt?

- in beiden Fällen ist jetzt jeweils ein Operand vom Typ `double`

⇝ es wird eine Fließkomma-Division durchgeführt

- Ergebnis ist der `double`-Wert `6.5`

- Wert wird ohne weitere Konvertierung in die `double`-Variable geschrieben

Arrays (Reihungen)

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Problem mit einfachen Variablen...

- **bisher**:
  - pro zu speicherndem Wert eine Variable (mit jeweils eigenem Bezeichner)
  - Probleme:
    - Was tun bei vielen gleichartigen Werten?
    - Was tun, wenn die Anzahl der Werte zur Entwicklungszeit unbekannt ist?
- **jetzt**: Array (Reihung, Feld)
  - erlaubt Speicherung mehrerer Werte desselben Datentyps
    - vereinfacht: Array in *einer* Variable mit *einem* Bezeichner gespeichert
  - einmalige Festlegung der Anzahl der zu speichernden Werte
    - allerdings: Element-Anzahl kann Wert sein, der erst zur Laufzeit feststeht

Arrays

Beispiel: `int`-Array der Größe 6

Index	0	1	2	3	4	5
Wert	13	-3	0	42	98	-110

- die einzelnen Elemente werden über einen Index adressiert
  - Achtung: in Java beginnt die Indizierung immer bei 0

$\rightsquigarrow$ ein Array der Länge n hat Elemente mit Indizes $\{0, 1, \ldots, n-1\}$

- werden die Elemente nicht explizit gesetzt, haben sie einen Standard-Wert
  - abhängig vom Datentyp, bei `int` beispielsweise `0`

Deklaration von Arrays

- wie bei primitiven Datentypen: Array muss vor Verwendung deklariert werden
  - dabei Angabe von Basis-Datentyp und Bezeichner der Array-Variable

Beispiele für die Deklaration von Arrays

```
int[] zahlen;        // eckige Klammern [] kennzeichnen Array
char[] buchstaben;
```

- anders als bei primitiven Datentypen: Array muss zusätzlich erzeugt werden
  - dabei Angabe der Anzahl an Elementen

Array-Erzeugung: Möglich

```
int[] zahlen;
zahlen = new int[100];
```

Array-Erzeugung: Üblich

```
int[] zahlen = new int[100];
```

Deklaration mit Wertzuweisung

- weitere Möglichkeit für die Array-Erzeugung: Deklaration mit Wertzuweisung
  - dabei werden die Array-Elemente bei der Array-Erzeugung initialisiert
  - die Array-Größe ergibt sich implizit aus der Element-Anzahl

Beispiel: Deklaration mit Wertzuweisung

```
int[] zahlen = {13, -3, 0, 42, 98, -110};
```

Zugriff auf einzelne Elemente

- lesender und schreibender Zugriff auf ein Element über seinen Index
  - Syntax: `<Array-Name>[<Index>]`

Beispiele für den Array-Zugriff

```
int[] zahlen = {13, -3, 0, 42, 98, -110};

System.out.println(zahlen[0]);  // 13

zahlen[0] = zahlen[1] + 10;
System.out.println(zahlen[0]);  // 7
```

Größe eines Arrays bestimmen

- jedes Array „kennt“ seine eigene Größe, die man abfragen kann
  - Syntax: `<Array-Name>.length`

Beispiel für die Bestimmung der Array-Größe

```
int[] zahlen = {13, -3, 0, 42, 98, -110};

System.out.println(zahlen.length);  // 6
```

Beispiel: Berechnung von Quadratzahlen

Ziel

Programm, das die ersten x Quadratzahlen berechnet und in einem Array speichert.

Mögliche Lösung

```
int[] quadratzahlen = new int[x];

for (int i = 0; i < quadratzahlen.length; ++i) {
  quadratzahlen[i] = i*i;
}
```

Array-Elemente nach der Ausführung der Schleife

Index	0	1	2	3	...	$x-1$
Wert	0	1	4	9	...	$(x-1)^2$

Ungültige Array-Indizes

Was geht hier schief?

```
int[] zahlen = {13, -3, 0, 42, 98, -110};

System.out.println("Anfang.");
System.out.println(zahlen[6]);
System.out.println("Ende.");
```

Bei der Ausführung...

```
benutzer@faui06a:~/ordner$ javac Zahlen.java
benutzer@faui06a:~/ordner$ java Zahlen
Anfang.
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
   at Zahlen.main(Zahlen.java:5)
```

Erklärung

Ein ungültiger Array-Index hat eine sog. `ArrayIndexOutOfBoundsException` zur Folge. Ohne weitere Maßnahmen bricht das Programm mit einer Fehlermeldung ab.

Mehrdimensionale Arrays

- Arrays können in Java auch mehrdimensional sein
  - Array von Arrays von Arrays . . . von Arrays eines Basistyps

Beispiel für ein mehrdimensionales Array

```
char[][] ticTacToe = new char[3][3]; // 3 "Zeilen" und 3 "Spalten"
ticTacToe[1][2] = 'X'; // "Zeile" 1 und "Spalte" 2
```

Array-Elemente nach der Ausführung

	0	1	2
0			
1			X
2			

Größe eines mehrdimensionalen Arrays bestimmen

- **bekannt**:
  - Größe eines eindimensionalen Arrays kann mittels `length` bestimmt werden
  - mehrdimensionale Arrays sind Arrays von Arrays ... von Arrays

⇝ Größe eines mehrdimensionalen Arrays in einer Dimension bestimmen:
  - auf das Array in der entsprechenden Dimension zugreifen...
  - ...und dessen Größe mittels `length` bestimmen

Beispiel

```
int[][] feld = new int[10][5];
System.out.println("Zeilen: " + feld.length);
System.out.println("Spalten: " + feld[0].length);
```

Achtung

Im Beispiel muss es ein Element `feld[0]` geben, andernfalls bricht das Programm mit einer Fehlermeldung ab.

Beispiel: Ausgabe eines (mehrdimensionalen) Arrays

Ausgabe eines Tic-Tac-Toe-Feldes

```
char[][] ticTacToe = new char[3][3];
ticTacToe[1][2] = 'X';

for (int z = 0; z < ticTacToe.length; ++z) {
  System.out.print(".");
  for (int s = 0; s < ticTacToe[z].length; ++s) {
    System.out.print(ticTacToe[z][s]);
    System.out.print(".");
  }
  System.out.println();
}
```

Ausgabe

```
. . . .
. . .X.
. . . .
```

Kommandozeilenargumente

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Kommandozeilenargumente

- bei Programmstarts können Kommandozeilenargumente übergeben werden
  - Beispiel: `javac Foo.java`
    - `javac`: Programmname
    - `Foo.java`: Kommandozeilenargument für das Programm `javac`
  - die Argumente werden durch ein Leerzeichen voneinander getrennt
    - Argumente mit Leerzeichen: in doppelte Anführungszeichen einschließen
- auch Java-Programmen können beim Start Argumente übergeben werden
  - landen als Zeichenketten im `args`-Parameter der `main`-Methode

$\rightsquigarrow$ einfache Möglichkeit der Benutzereingabe

Beispiel: Echo

Aufgabe

Programm, das das erste Kommandozeilenargument ausgibt.

Mögliche Lösung

```
public class Echo {
  public static void main(String[] args) {
    if (args.length >= 1) {  // nicht vergessen!
      System.out.println(args[0]);
    }
  }
}
```

Aufrufe des Echo-Programms

```
benutzer@faui06a:~/ordner$ java Echo
benutzer@faui06a:~/ordner$ java Echo "Repetitorium Informatik" RIP
Repetitorium Informatik
```

Beispiel: Summe (I)

Aufgabe

Programm, das die Summe aller als Kommandozeilenargumente übergebenen Zahlen berechnet und ausgibt.

Problem

Die Kommandozeilenargumente werden stets als Zeichenketten (d.h. vom Typ `String`) übergeben. Für Berechnungen müssen diese erst explizit in Zahlen konvertiert werden.

`String` $\mapsto$ `int`

```
int wert =
  Integer.parseInt(strVar);
```

`String` $\mapsto$ `double`

```
double wert =
  Double.parseDouble(strVar);
```

Beispiel: Summe (II)

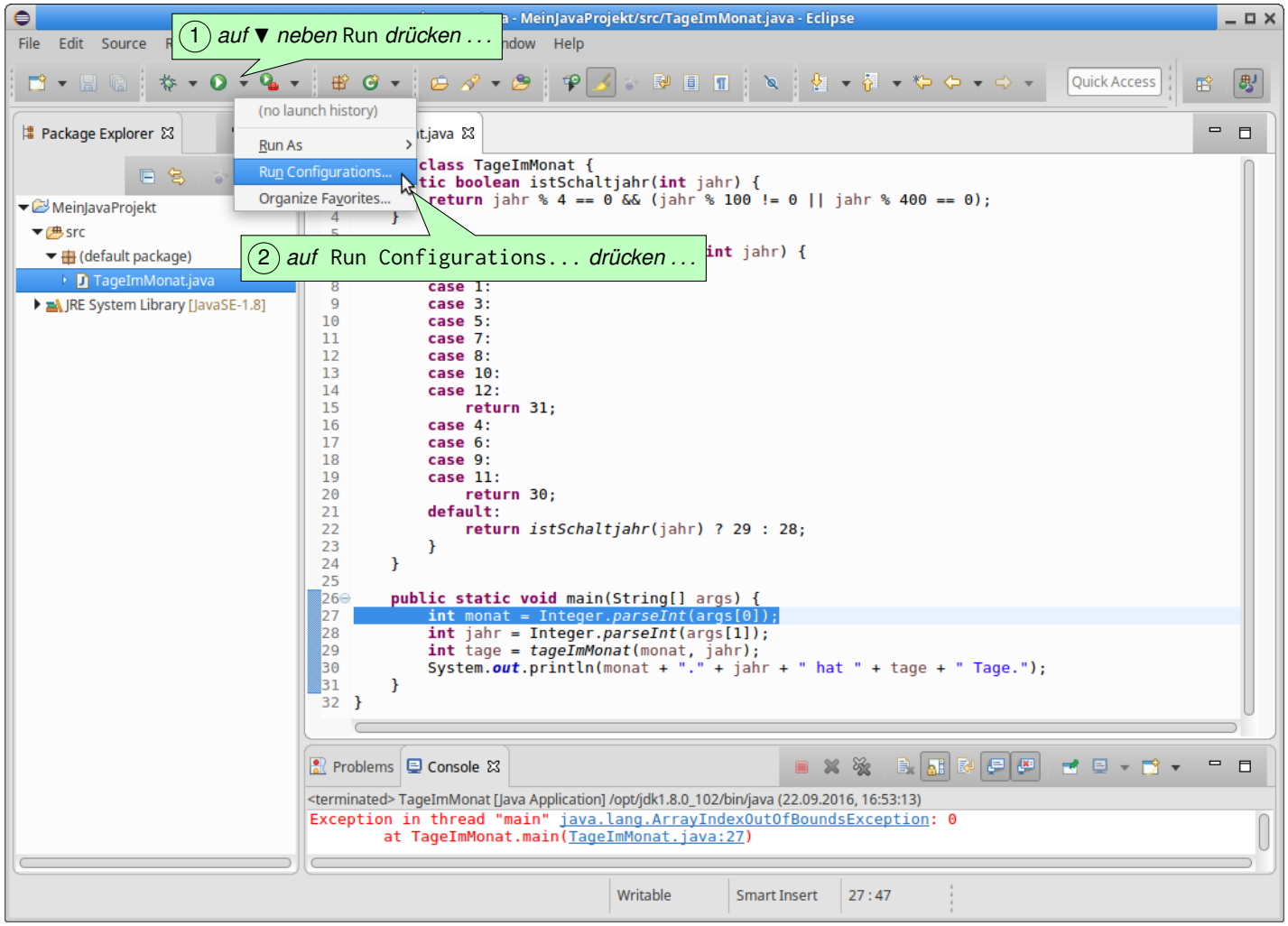
Mögliche Lösung

```
public class Summe {
  public static void main(String[] args) {
    double summe = 0.0;
    for (int index = 0; index < args.length; ++index) {
      summe = summe + Double.parseDouble(args[index]);
    }
    System.out.println(summe);
  }
}
```

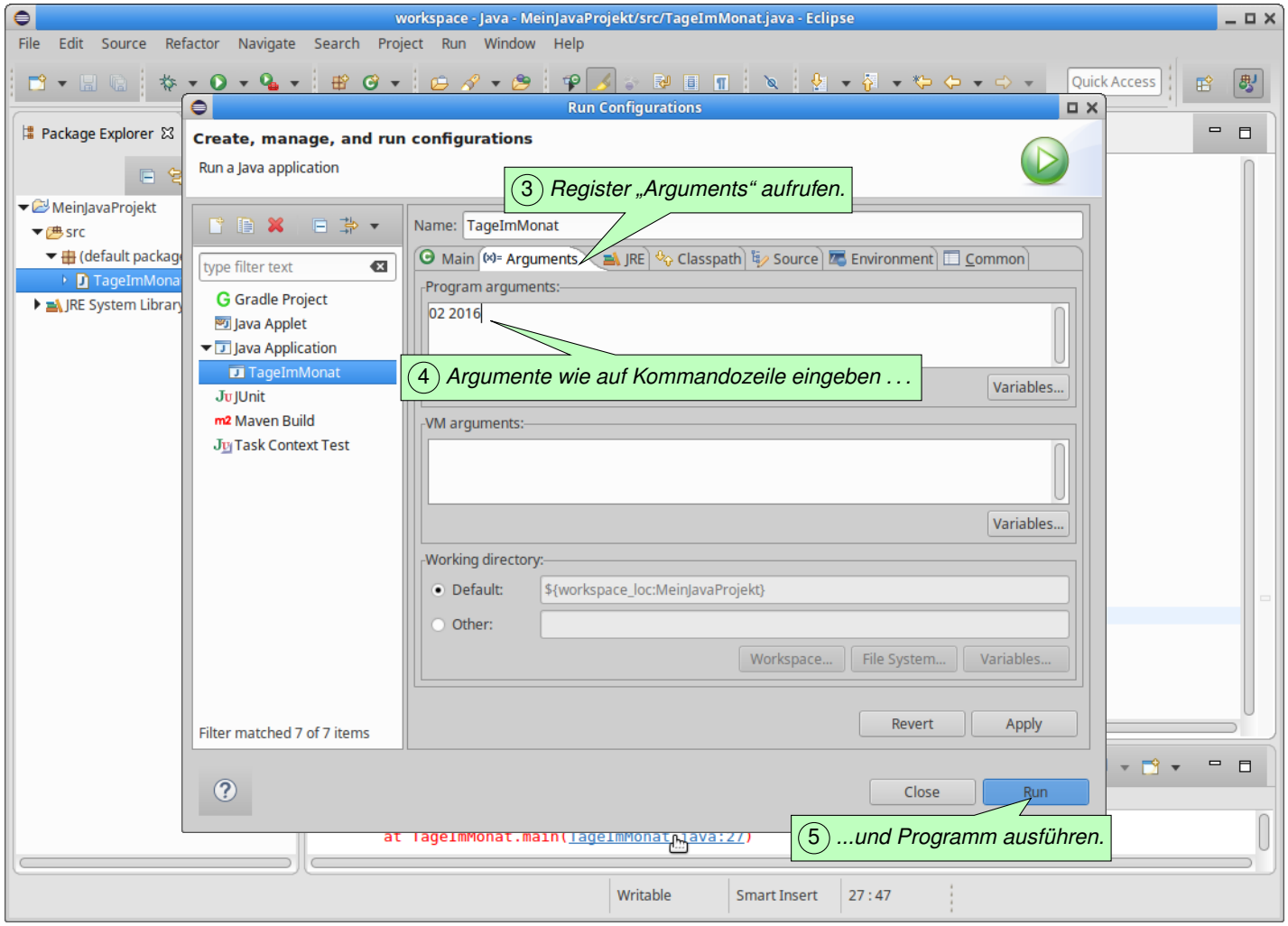
Aufrufe des Summe-Programms

```
benutzer@faui06a:~/ordner$ java Summe
0.0
benutzer@faui06a:~/ordner$ java Summe 13 3 5 8
29.0
```

Kommandozeilenargumente in Eclipse



Kommandozeilenargumente in Eclipse



Fragen? Fragen!

(hilft auch den anderen)

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT