

# Lassen wir die Schleifen fließen

## Tutorium 3

Florian Sihler ◦ KW 46





”

”

**Hate**

**Hate**

**”**

**Hate**

**Hate**

**Hate ”**

**Hate**

**Hate**

**Hate ”**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**”**

**Hate**

**Hate**



**Hate**

**Hate**

**Hate** ”

**Hate**

**Hate**

**Hate**

Hate

Hate

Hate

”

Hate

Hate

Hate

Hate

Hate

Hate

Hate

”

Hate

Hate

Hate

Hate

Hate

**Hate**

**Hate**

**Hate**

**Hate**

**”**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**”**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**”**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**”**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

**Hate**

Hate

**Hate**

Hate

Hate

”

Hate

**Hate**

**Hate**

Hate

Hate

Hate

**Hate**

**Hate**



**Hate**

Hate

**Hate**

Hate

Hate

”

Love

Hate

Hate

**Hate**

Hate

Hate

Hate

**Hate**

**Hate**

**Hate**

Hate

**Hate**

Hate

**”**

Hate

**Love**

**Hate**

**Hate**

**Hate**

**Hate**

Hate

Hate

**Hate**

**Hate**

**Did they...**

**Did they... hurt you?**

**Did they... hurt you?**



**My precious ५५**

**Did they... hurt you?**



**My precious ५५**

# Allgemeines

# Allgemeines

- Übt Java. Beispielprojekte (nach Schwierigkeit):



# Allgemeines

- Übt Java. Beispielprojekte (nach Schwierigkeit):
  - Taschenrechner:

- **Übt Java. Beispielprojekte (nach Schwierigkeit):**
  - Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.

- **Übt Java. Beispielprojekte (nach Schwierigkeit):**
  - Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.
  - TicTacToe:

- **Übt Java. Beispielprojekte (nach Schwierigkeit):**
  - Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.
  - TicTacToe: Tic-Tac-Toe mit netter ASCII-Art im Terminal.

## ■ Übt Java. Beispielprojekte (nach Schwierigkeit):

- Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.
- TicTacToe: Tic-Tac-Toe mit netter ASCII-Art im Terminal. Kleine Herausforderung: Der andere oder auch beide Spieler sollen von einem Computer übernommen werden können.

## ■ Übt Java. Beispielprojekte (nach Schwierigkeit):

- Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.
- TicTacToe: Tic-Tac-Toe mit netter ASCII-Art im Terminal. Kleine Herausforderung: Der andere oder auch beide Spieler sollen von einem Computer übernommen werden können.
- Wer wird Millionär?

## ■ Übt Java. Beispielprojekte (nach Schwierigkeit):

- Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.
- TicTacToe: Tic-Tac-Toe mit netter ASCII-Art im Terminal. Kleine Herausforderung: Der andere oder auch beide Spieler sollen von einem Computer übernommen werden können.
- Wer wird Millionär? Fragen sollen aus den jeweiligen Schwierigkeitsgraden zufällige gewählt werden, sonst normales Wer-Wird Millionär, die Joker arbeiten natürlich mit den korrekten Antworten.

## ■ Übt Java. Beispielprojekte (nach Schwierigkeit):

- Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.
- TicTacToe: Tic-Tac-Toe mit netter ASCII-Art im Terminal. Kleine Herausforderung: Der andere oder auch beide Spieler sollen von einem Computer übernommen werden können.
- Wer wird Millionär? Fragen sollen aus den jeweiligen Schwierigkeitsgraden zufällige gewählt werden, sonst normales Wer-Wird Millionär, die Joker arbeiten natürlich mit den korrekten Antworten.
- Sudoku:



## ■ Übt Java. Beispielprojekte (nach Schwierigkeit):

- Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.
- TicTacToe: Tic-Tac-Toe mit netter ASCII-Art im Terminal. Kleine Herausforderung: Der andere oder auch beide Spieler sollen von einem Computer übernommen werden können.
- Wer wird Millionär? Fragen sollen aus den jeweiligen Schwierigkeitsgraden zufällige gewählt werden, sonst normales Wer-Wird Millionär, die Joker arbeiten natürlich mit den korrekten Antworten.
- Sudoku: Schreibe eine Terminal-Anwendung die Sudokus einliest und dann gelöst wieder ausgibt. (Bonus: ANSI-Escape Codes)

## ■ Übt Java. Beispielprojekte (nach Schwierigkeit):

- Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.
- TicTacToe: Tic-Tac-Toe mit netter ASCII-Art im Terminal. Kleine Herausforderung: Der andere oder auch beide Spieler sollen von einem Computer übernommen werden können.
- Wer wird Millionär? Fragen sollen aus den jeweiligen Schwierigkeitsgraden zufällige gewählt werden, sonst normales Wer-Wird Millionär, die Joker arbeiten natürlich mit den korrekten Antworten.
- Sudoku: Schreibe eine Terminal-Anwendung die Sudokus einliest und dann gelöst wieder ausgibt. (Bonus: ANSI-Escape Codes)
- Conway's Game of Life:

## ■ Übt Java. Beispielprojekte (nach Schwierigkeit):

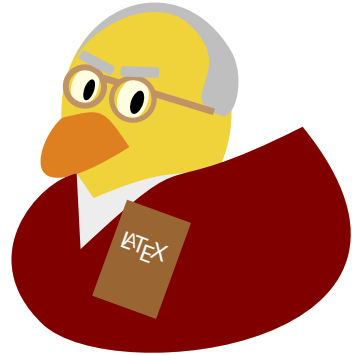
- Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.
- TicTacToe: Tic-Tac-Toe mit netter ASCII-Art im Terminal. Kleine Herausforderung: Der andere oder auch beide Spieler sollen von einem Computer übernommen werden können.
- Wer wird Millionär? Fragen sollen aus den jeweiligen Schwierigkeitsgraden zufällige gewählt werden, sonst normales Wer-Wird Millionär, die Joker arbeiten natürlich mit den korrekten Antworten.
- Sudoku: Schreibe eine Terminal-Anwendung die Sudokus einliest und dann gelöst wieder ausgibt. (Bonus: ANSI-Escape Codes)
- Conway's Game of Life: Programmiere Conway's Game of Life im Terminal oder mittels einer Grafik-Bibliothek deiner Wahl.

## ■ Übt Java. Beispielprojekte (nach Schwierigkeit):

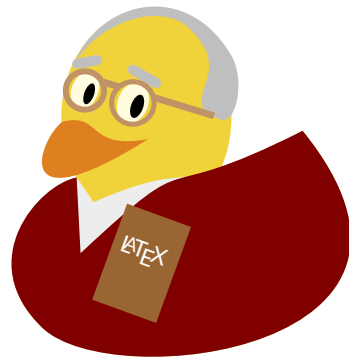
- Taschenrechner: Programmiere einen Taschenrechner für die Kommandozeile der die grundlegenden Rechenoperationen wie +, -, \* und / beherrscht.
- TicTacToe: Tic-Tac-Toe mit netter ASCII-Art im Terminal. Kleine Herausforderung: Der andere oder auch beide Spieler sollen von einem Computer übernommen werden können.
- Wer wird Millionär? Fragen sollen aus den jeweiligen Schwierigkeitsgraden zufällige gewählt werden, sonst normales Wer-Wird Millionär, die Joker arbeiten natürlich mit den korrekten Antworten.
- Sudoku: Schreibe eine Terminal-Anwendung die Sudokus einliest und dann gelöst wieder ausgibt. (Bonus: ANSI-Escape Codes)
- Conway's Game of Life: Programmiere Conway's Game of Life im Terminal oder mittels einer Grafik-Bibliothek deiner Wahl.
- ...



# Keine Plagiate!

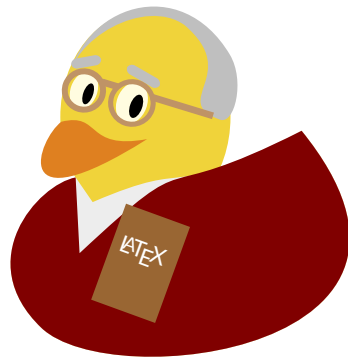


# Danke



# Danke

Das ist besonders lustig, da ich  
die Ente aus den letzten Semestern  
übernommen habe 😊.





# Präsenzaufgabe

1

Starring Mr. Pyramid

# Präsenzaufgabe

1

Starring Mr. Pyramid

Schreiben Sie ein Java-Programm, welches als Eingabe eine positive ganze Zahl N erwartet und daraufhin ein gleichseitiges Dreieck mit Hilfe von \*- und Leerfeld-Charakteren in der Kommandozeile ausgibt.

# Präsenzaufgabe

1

Starring Mr. Pyramid

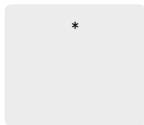
Schreiben Sie ein Java-Programm, welches als Eingabe eine positive ganze Zahl  $N$  erwartet und daraufhin ein gleichseitiges Dreieck mit Hilfe von \*- und Leerfeld-Charakteren in der Kommandozeile ausgibt. Die Ausgaben für  $1 \leq N \leq 5$  sollten folgendermaßen aussehen:

# Präsenzaufgabe

1

Starring Mr. Pyramid

Schreiben Sie ein Java-Programm, welches als Eingabe eine positive ganze Zahl  $N$  erwartet und daraufhin ein gleichseitiges Dreieck mit Hilfe von \*- und Leerfeld-Charakteren in der Kommandozeile ausgibt. Die Ausgaben für  $1 \leq N \leq 5$  sollten folgendermaßen aussehen:



$N = 1$

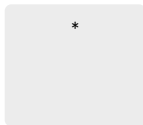
*Hinweis:* Überlegen Sie sich eine allgemeine Formel für die Zeileneinrückung. Fügen Sie außerdem Leerzeichen zwischen \*-en für die alternierenden Positionen in den Reihen ein.

# Präsenzaufgabe

1

Starring Mr. Pyramid

Schreiben Sie ein Java-Programm, welches als Eingabe eine positive ganze Zahl  $N$  erwartet und daraufhin ein gleichseitiges Dreieck mit Hilfe von \*- und Leerfeld-Charakteren in der Kommandozeile ausgibt. Die Ausgaben für  $1 \leq N \leq 5$  sollten folgendermaßen aussehen:



$N = 1$



$N = 2$

*Hinweis:* Überlegen Sie sich eine allgemeine Formel für die Zeileneinrückung. Fügen Sie außerdem Leerzeichen zwischen \*-en für die alternierenden Positionen in den Reihen ein.

# Präsenzaufgabe

1

Starring Mr. Pyramid

Schreiben Sie ein Java-Programm, welches als Eingabe eine positive ganze Zahl  $N$  erwartet und daraufhin ein gleichseitiges Dreieck mit Hilfe von `*`- und Leerfeld-Charakteren in der Kommandozeile ausgibt. Die Ausgaben für  $1 \leq N \leq 5$  sollten folgendermaßen aussehen:

```
  *
```

$N = 1$

```
  *  
 * *
```

$N = 2$

```
  *  
 * *  
* * *
```

$N = 3$

*Hinweis:* Überlegen Sie sich eine allgemeine Formel für die Zeileneinrückung. Fügen Sie außerdem Leerzeichen zwischen `*`-en für die alternierenden Positionen in den Reihen ein.

# Präsenzaufgabe

1

Starring Mr. Pyramid

Schreiben Sie ein Java-Programm, welches als Eingabe eine positive ganze Zahl  $N$  erwartet und daraufhin ein gleichseitiges Dreieck mit Hilfe von \*- und Leerfeld-Charakteren in der Kommandozeile ausgibt. Die Ausgaben für  $1 \leq N \leq 5$  sollten folgendermaßen aussehen:

```
  *
```

$N = 1$

```
  *  
 * *
```

$N = 2$

```
  *  
 * *  
* * *
```

$N = 3$

```
  *  
 * *  
* * *  
* * * *
```

$N = 4$

*Hinweis:* Überlegen Sie sich eine allgemeine Formel für die Zeileneinrückung. Fügen Sie außerdem Leerzeichen zwischen \*-en für die alternierenden Positionen in den Reihen ein.

# Präsenzaufgabe

1

Starring Mr. Pyramid

Schreiben Sie ein Java-Programm, welches als Eingabe eine positive ganze Zahl  $N$  erwartet und daraufhin ein gleichseitiges Dreieck mit Hilfe von `*`- und Leerfeld-Charakteren in der Kommandozeile ausgibt. Die Ausgaben für  $1 \leq N \leq 5$  sollten folgendermaßen aussehen:

```
  *
```

$N = 1$

```
  *
 * *
```

$N = 2$

```
  *
 * *
* * *
```

$N = 3$

```
  *
 * *
* * *
* * * *
```

$N = 4$

```
  *
 * *
* * *
* * * *
* * * * *
```

$N = 5$

*Hinweis:* Überlegen Sie sich eine allgemeine Formel für die Zeileneinrückung. Fügen Sie außerdem Leerzeichen zwischen `*`-en für die alternierenden Positionen in den Reihen ein.



# Präsenzaufgabe

1

Starring Mr. Pyramid

Schreiben Sie ein Java-Programm, welches als Eingabe eine positive ganze Zahl  $N$  erwartet und daraufhin ein gleichseitiges Dreieck mit Hilfe von \*- und Leerfeld-Charakteren in der Kommandozeile ausgibt. Die Ausgaben für  $1 \leq N \leq 5$  sollten folgendermaßen aussehen:

```
  *
```

$N = 1$

```
  *  
 * *
```

$N = 2$

```
  *  
 * *  
* * *
```

$N = 3$

```
  *  
 * *  
* * *  
* * * *
```

$N = 4$

```
  *  
 * *  
* * *  
* * * *  
* * * * *
```

$N = 5$

# Präsenzaufgabe

1

Starring Mr. Pyramid

Schreiben Sie ein Java-Programm, welches als Eingabe eine positive ganze Zahl  $N$  erwartet und daraufhin ein gleichseitiges Dreieck mit Hilfe von `*`- und Leerfeld-Charakteren in der Kommandozeile ausgibt. Die Ausgaben für  $1 \leq N \leq 5$  sollten folgendermaßen aussehen:

```
  *
```

$N = 1$

```
  *  
 * *
```

$N = 2$

```
  *  
 * *  
* * *
```

$N = 3$

```
  *  
 * *  
* * *  
* * * *
```

$N = 4$

```
  *  
 * *  
* * *  
* * * *  
* * * * *
```

$N = 5$

*Hinweis:* Überlegen Sie sich eine allgemeine Formel für die Zeileneinrückung. Fügen Sie außerdem Leerzeichen zwischen `*`-en für die alternierenden Positionen in den Reihen ein.

# Präsenzaufgabe

# Präsenzaufgabe

- Die  $i$ te Zeile ist immer um  $N - i$  Positionen eingerückt.

# Präsenzaufgabe

- Die  $i$ te Zeile ist immer um  $N - i$  Positionen eingerückt.
- Wir erschaffen ein Grundgerüst:

# Präsenzaufgabe

- Die  $i$ te Zeile ist immer um  $N - i$  Positionen eingerückt.
- Wir erschaffen ein Grundgerüst:

```
public class Pyramid {
```

```
}
```

# Präsenzaufgabe

- Die  $i$ te Zeile ist immer um  $N - i$  Positionen eingerückt.
- Wir erschaffen ein Grundgerüst:

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

# Präsenzaufgabe

- Die  $i$ te Zeile ist immer um  $N - i$  Positionen eingerückt.
- Wir erschaffen ein Grundgerüst:

```
public class Pyramid {  
    public static void main(String[] args) {  
        if(args.length != 1)  
            System.exit(1);  
  
    }  
}
```



# Präsenzaufgabe

- Die  $i$ te Zeile ist immer um  $N - i$  Positionen eingerückt.
- Wir erschaffen ein Grundgerüst:

```
public class Pyramid {  
    public static void main(String[] args) {  
        if(args.length != 1)  
            System.exit(1);  
        int N = Integer.parseInt(args[0]);  
        // Oder auch gerne 'n'  
  
    }  
}
```

# Präsenzaufgabe

- Die  $i$ te Zeile ist immer um  $N - i$  Positionen eingerückt.
- Wir erschaffen ein Grundgerüst:

```
public class Pyramid {  
    public static void main(String[] args) {  
        if(args.length != 1)  
            System.exit(1);  
        int N = Integer.parseInt(args[0]);  
        // Oder auch gerne 'n'  
  
        // Pyramide Erschaffen +2  
    }  
}
```

# Präsenzaufgabe

# Präsenzaufgabe

- Nun basteln wir uns eine Pyramide:

# Präsenzaufgabe

- Nun basteln wir uns eine Pyramide:

# Präsenzaufgabe

- Nun basteln wir uns eine Pyramide:

```
for (int i = 1; i <= N; i++) {
```

```
}
```

# Präsenzaufgabe

- Nun basteln wir uns eine Pyramide:

```
for (int i = 1; i <= N; i++) {  
    for (int j = 1; j <= N - i; j++) {  
        System.out.print("_");  
    }  
  
}
```

# Präsenzaufgabe

- Nun basteln wir uns eine Pyramide:

```
for (int i = 1; i <= N; i++) {  
    for (int j = 1; j <= N - i; j++) {  
        System.out.print("_");  
    }  
    for (int j = 1; j <= i; j++) {  
  
    }  
  
}
```



# Präsenzaufgabe

- Nun basteln wir uns eine Pyramide:

```
for (int i = 1; i <= N; i++) {  
    for (int j = 1; j <= N - i; j++) {  
        System.out.print("_");  
    }  
    for (int j = 1; j <= i; j++) {  
        System.out.print("*_"); // zusätzliche Leerzeichen  
    }  
}
```

# Präsenzaufgabe

- Nun basteln wir uns eine Pyramide:

```
for (int i = 1; i <= N; i++) {  
    for (int j = 1; j <= N - i; j++) {  
        System.out.print("_");  
    }  
    for (int j = 1; j <= i; j++) {  
        System.out.print("*_"); // zusätzliche Leerzeichen  
    }  
    System.out.println(); // Zeilenumbruch  
}
```

# Präsenzaufgabe

# Präsenzaufgabe

- Damit ist alles komplett ([Pyramid.java](#)).

# Präsenzaufgabe

- Damit ist alles komplett (`Pyramid.java`).
- Somit ergibt sich `java Pyramid 8`:

# Präsenzaufgabe

- Damit ist alles komplett (`Pyramid.java`).
- Somit ergibt sich `java Pyramid 8`:

```
      *
     * *
    * * *
   * * * *
  * * * * *
 * * * * * *
* * * * * * *
```

# Präsenzaufgabe die Zweite

2

Code! Code! Code! Repeat.

# Präsenzaufgabe die Zweite

2

Code! Code! Code! Repeat.

Drücken Sie die folgenden Anweisungen durch äquivalente Schleifen der angegeben Art aus!



# Präsenzaufgabe die Zweite

2

Code! Code! Code! Repeat.

Drücken Sie die folgenden Anweisungen durch äquivalente Schleifen der angegeben Art aus! Die Variable `int k` sei jeweils mit einem (beliebigen) Wert initialisiert.

# Präsenzaufgabe die Zweite

2

Code! Code! Code! Repeat.

Drücken Sie die folgenden Anweisungen durch äquivalente Schleifen der angegeben Art aus! Die Variable `int k` sei jeweils mit einem (beliebigen) Wert initialisiert.

a) Do-While:

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

# Präsenzaufgabe die Zweite

2

Code! Code! Code! Repeat.

Drücken Sie die folgenden Anweisungen durch äquivalente Schleifen der angegeben Art aus! Die Variable `int k` sei jeweils mit einem (beliebigen) Wert initialisiert.

a) Do-While:

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

b) For-Schleife:

```
int x = 0, i = 0;  
while(i < k) { x += k * ++i; }  
System.out.println("x: " + x);
```

# Präsenzaufgabe die Zweite

2

Code! Code! Code! Repeat.

Drücken Sie die folgenden Anweisungen durch äquivalente Schleifen der angegeben Art aus! Die Variable `int k` sei jeweils mit einem (beliebigen) Wert initialisiert.

a) Do-While:

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

b) For-Schleife:

```
int x = 0, i = 0;  
while(i < k) { x += k * ++i; }  
System.out.println("x:␣" + x);
```

c) For-Schleife:

```
if(k > 0){  
    int i = 1, m = 0;  
    while(i < k){  
        if(k*i > m) m = k*i;  
        ++i;  
    }  
    System.out.println("m:␣" + m);  
}
```

# Präsenzaufgabe 2 - Lösung

# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

```
if(k > 1) {
```

```
}
```



# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

```
if(k > 1) {  
    int i = 1;  
  
}
```

# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

```
if(k > 1) {  
    int i = 1;  
    do {  
  
    } while (i < k);  
}
```

# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

```
if(k > 1) {  
    int i = 1;  
    do {  
        System.out.println(i * i);  
        ++i;  
    } while (i < k);  
}
```

# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

```
if(k > 1) {  
    int i = 1;  
    do {  
        System.out.println(i * i);  
        ++i;  
    } while (i < k);  
}
```

b) Wir erhalten (rechts):

# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

```
if(k > 1) {  
    int i = 1;  
    do {  
        System.out.println(i * i);  
        ++i;  
    } while (i < k);  
}
```

b) Wir erhalten (rechts):

```
int x = 0, i = 0;  
while(i < k) { x += k * ++i; }  
System.out.println("x:␣" + x);
```

# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

```
if(k > 1) {  
    int i = 1;  
    do {  
        System.out.println(i * i);  
        ++i;  
    } while (i < k);  
}
```

b) Wir erhalten (rechts):

```
int x = 0, i = 0;  
while(i < k) { x += k * ++i; }  
System.out.println("x: " + x);
```

```
int x = 0;
```

# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

```
if(k > 1) {  
    int i = 1;  
    do {  
        System.out.println(i * i);  
        ++i;  
    } while (i < k);  
}
```

b) Wir erhalten (rechts):

```
int x = 0, i = 0;  
while(i < k) { x += k * ++i; }  
System.out.println("x:␣" + x);
```

```
int x = 0;  
for(int i = 0; i < k; i++)
```

# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

```
if(k > 1) {  
    int i = 1;  
    do {  
        System.out.println(i * i);  
        ++i;  
    } while (i < k);  
}
```

b) Wir erhalten (rechts):

```
int x = 0, i = 0;  
while(i < k) { x += k * ++i; }  
System.out.println("x: " + x);
```

```
int x = 0;  
for(int i = 0; i < k; i++)  
    x += k * (i + 1);
```



# Präsenzaufgabe 2 - Lösung

a) Wir erhalten (rechts):

```
for (int i = 1; i < k; ++i) {  
    System.out.println(i * i);  
}
```

```
if(k > 1) {  
    int i = 1;  
    do {  
        System.out.println(i * i);  
        ++i;  
    } while (i < k);  
}
```

b) Wir erhalten (rechts):

```
int x = 0, i = 0;  
while(i < k) { x += k * ++i; }  
System.out.println("x:␣" + x);
```

```
int x = 0;  
for(int i = 0; i < k; i++)  
    x += k * (i + 1);  
System.out.println("x:␣" + x);
```

# Präsenzaufgabe 2 - Lösung

# Präsenzaufgabe 2 - Lösung

c) Wir erhalten (rechts):

## Präsenzaufgabe 2 - Lösung

c) Wir erhalten (rechts):

```
if(k > 0){  
    int i = 1, m = 0;  
    while(i < k){  
        if(k*i > m) m = k+i;  
        ++i;  
    }  
    System.out.println("m:␣" + m);  
}
```

## Präsenzaufgabe 2 - Lösung

c) Wir erhalten (rechts):

```
if(k > 0){  
    int i = 1, m = 0;  
    while(i < k){  
        if(k*i > m) m = k+i;  
        ++i;  
    }  
    System.out.println("m:␣" + m);  
}
```

```
if(k > 0){  
  
}
```

## Präsenzaufgabe 2 - Lösung

c) Wir erhalten (rechts):

```
if(k > 0){  
    int i = 1, m = 0;  
    while(i < k){  
        if(k*i > m) m = k+i;  
        ++i;  
    }  
    System.out.println("m:␣" + m);  
}
```

```
if(k > 0){  
    int m = 0;  
  
}
```

## Präsenzaufgabe 2 - Lösung

c) Wir erhalten (rechts):

```
if(k > 0){  
    int i = 1, m = 0;  
    while(i < k){  
        if(k*i > m) m = k+i;  
        ++i;  
    }  
    System.out.println("m:␣" + m);  
}
```

```
if(k > 0){  
    int m = 0;  
    for(int i = 1; i < k; ++i)  
        if(k*i > m) m = k+i;  
}
```

## Präsenzaufgabe 2 - Lösung

c) Wir erhalten (rechts):

```
if(k > 0){  
    int i = 1, m = 0;  
    while(i < k){  
        if(k*i > m) m = k+i;  
        ++i;  
    }  
    System.out.println("m:␣" + m);  
}
```

```
if(k > 0){  
    int m = 0;  
    for(int i = 1; i < k; ++i)  
        if(k*i > m)  
            m = k+i;  
}
```



## Präsenzaufgabe 2 - Lösung

c) Wir erhalten (rechts):

```
if(k > 0){  
    int i = 1, m = 0;  
    while(i < k){  
        if(k*i > m) m = k+i;  
        ++i;  
    }  
    System.out.println("m:␣" + m);  
}
```

```
if(k > 0){  
    int m = 0;  
    for(int i = 1; i < k; ++i)  
        if(k*i > m) m = k+i;  
}
```

## Präsenzaufgabe 2 - Lösung

c) Wir erhalten (rechts):

```
if(k > 0){  
    int i = 1, m = 0;  
    while(i < k){  
        if(k*i > m) m = k+i;  
        ++i;  
    }  
    System.out.println("m:␣" + m);  
}
```

```
if(k > 0){  
    int m = 0;  
    for(int i = 1; i < k; ++i)  
        if(k*i > m) m = k+i;  
    System.out.println("m:␣" + m);  
}
```

# Übungsblatt 3 - Aufgabe 1 a)

## Übungsblatt 3 - Aufgabe 1 a)

- Uns reicht es, wenn dies nur für den zweiten `if-else`-Block geschieht.

## Übungsblatt 3 - Aufgabe 1 a)

- Uns reicht es, wenn dies nur für den zweiten **if-else**-Block geschieht.
- Damit wird dieser zu:

## Übungsblatt 3 - Aufgabe 1 a)

- Uns reicht es, wenn dies nur für den zweiten **if-else**-Block geschieht.
- Damit wird dieser zu:

```
switch (x) {
```

```
}
```

## Übungsblatt 3 - Aufgabe 1 a)

- Uns reicht es, wenn dies nur für den zweiten **if-else**-Block geschieht.
- Damit wird dieser zu:

```
switch (x) {  
    case 1: System.out.println("A");  
    break;
```

```
}
```

## Übungsblatt 3 - Aufgabe 1 a)

- Uns reicht es, wenn dies nur für den zweiten **if-else**-Block geschieht.
- Damit wird dieser zu:

```
switch (x) {  
    case 1: System.out.println("A");  
        break;  
    case 2: System.out.println("B");  
        break;  
  
}
```



## Übungsblatt 3 - Aufgabe 1 a)

- Uns reicht es, wenn dies nur für den zweiten **if-else**-Block geschieht.
- Damit wird dieser zu:

```
switch (x) {  
    case 1: System.out.println("A");  
        break;  
    case 2: System.out.println("B");  
        break;  
    case 3: System.out.println("C");  
        break;  
  
}
```

## Übungsblatt 3 - Aufgabe 1 a)

- Uns reicht es, wenn dies nur für den zweiten **if-else**-Block geschieht.
- Damit wird dieser zu:

```
switch (x) {  
    case 1: System.out.println("A");  
        break;  
    case 2: System.out.println("B");  
        break;  
    case 3: System.out.println("C");  
        break;  
    default: System.out.println("D");  
        break;  
}
```

# Übungsblatt 3 - Aufgabe 1 a)

## Übungsblatt 3 - Aufgabe 1 a)

- Fanatiker dürfen aber auch die Kommandozeilenargumente auf diese Weise überprüfen ([SwitchCaseControlled.java](#)):

## Übungsblatt 3 - Aufgabe 1 a)

- Fanatiker dürfen aber auch die Kommandozeilenargumente auf diese Weise überprüfen ([SwitchCaseControlled.java](#)):

```
switch(args.length) {  
  
}
```

- Ist das **switch** hier sinnvoll?

# Übungsblatt 3 - Aufgabe 1 a)

- Fanatiker dürfen aber auch die Kommandozeilenargumente auf diese Weise überprüfen ([SwitchCaseControlled.java](#)):

```
switch(args.length) {  
    case 1: x = Integer.parseInt(args[0]); break;  
  
}
```

- Ist das **switch** hier sinnvoll? (Nicht wirklich. Mindestens drei Fälle sollten es schon sein, damit sich der Schreibaufwand auch reduziert beziehungsweise die Lesbarkeit verbessert.)

# Übungsblatt 3 - Aufgabe 1 a)

- Fanatiker dürfen aber auch die Kommandozeilenargumente auf diese Weise überprüfen ([SwitchCaseControlled.java](#)):

```
switch(args.length) {  
    case 1: x = Integer.parseInt(args[0]); break;  
    default: System.exit(1); return;  
}
```

- Ist das **switch** hier sinnvoll? (Nicht wirklich. Mindestens drei Fälle sollten es schon sein, damit sich der Schreibaufwand auch reduziert beziehungsweise die Lesbarkeit verbessert.)
- Ist das **return** hier notwendig?

# Übungsblatt 3 - Aufgabe 1 a)

- Fanatiker dürfen aber auch die Kommandozeilenargumente auf diese Weise überprüfen ([SwitchCaseControlled.java](#)):

```
switch(args.length) {  
    case 1: x = Integer.parseInt(args[0]); break;  
    default: System.exit(1); return;  
}
```

- Ist das **switch** hier sinnvoll? (Nicht wirklich. Mindestens drei Fälle sollten es schon sein, damit sich der Schreibaufwand auch reduziert beziehungsweise die Lesbarkeit verbessert.)
- Ist das **return** hier notwendig? (Nein, die JVM bricht durch das `exit` ab.)



# Übungsblatt 3 - Aufgabe 1 a)

- Fanatiker dürfen aber auch die Kommandozeilenargumente auf diese Weise überprüfen ([SwitchCaseControlled.java](#)):

```
switch(args.length) {  
    case 1: x = Integer.parseInt(args[0]); break;  
    default: System.exit(1); return;  
}
```

- Ist das **switch** hier sinnvoll? (Nicht wirklich. Mindestens drei Fälle sollten es schon sein, damit sich der Schreibaufwand auch reduziert beziehungsweise die Lesbarkeit verbessert.)
- Ist das **return** hier notwendig? (Nein, die JVM bricht durch das `exit` ab.)
- Warum könnte man **return** eventuell doch brauchen?

# Übungsblatt 3 - Aufgabe 1 a)

- Fanatiker dürfen aber auch die Kommandozeilenargumente auf diese Weise überprüfen ([SwitchCaseControlled.java](#)):

```
switch(args.length) {  
    case 1: x = Integer.parseInt(args[0]); break;  
    default: System.exit(1); return;  
}
```

- Ist das **switch** hier sinnvoll? (Nicht wirklich. Mindestens drei Fälle sollten es schon sein, damit sich der Schreibaufwand auch reduziert beziehungsweise die Lesbarkeit verbessert.)
- Ist das **return** hier notwendig? (Nein, die JVM bricht durch das `exit` ab.)
- Warum könnte man **return** eventuell doch brauchen? (Theoretisch muss sie da nicht enden und auch wenn der Code danach nicht mehr ausgeführt wird, der Java Compiler weiß in der Regel noch nicht, dass es bei `System.exit` wirklich vorbei ist.)

# Übungsblatt 3 - Aufgabe 1 b)

# Übungsblatt 3 - Aufgabe 1 b)

- Die Bedingung `(x < 0) || (x >= 3)` invertiert den Default-Fall!

## Übungsblatt 3 - Aufgabe 1 b)

- Die Bedingung `(x < 0) || (x >= 3)` invertiert den Default-Fall!
- Damit genügt (`SwitchCaseControlledAlternate.java`)

```
switch (x) {  
    case 0: System.out.println("D"); break;  
    case 1: System.out.println("A"); break;  
    case 2: System.out.println("B"); break;  
    default:  
        System.out.println("C");  
}
```



- Zunächst mit While ():

- Zunächst mit While (FacultyWhileLoop.java):

```
int faculty = 1;  
int i = 0;
```



- Zunächst mit While (FacultyWhileLoop.java):

```
int faculty = 1;  
int i = 0;  
while (i <= n) {
```

- Zunächst mit While (FacultyWhileLoop.java):

```
int faculty = 1;
int i = 0;
while (i <= n) {
    if (i != 0) {

    }

}
```

- Zunächst mit While (FacultyWhileLoop.java):

```
int faculty = 1;
int i = 0;
while (i <= n) {
    if (i != 0) {
        faculty *= i;
    }

}
```

- Zunächst mit While (FacultyWhileLoop.java):

```
int faculty = 1;
int i = 0;
while (i <= n) {
    if (i != 0) {
        faculty *= i;
    }
    System.out.println(i + "!_=_ " + faculty);
    i++;
}
```



- Non mit Do-While ():

- Non mit Do-While ([FacultyDoWhileLoop.java](#)):

```
int faculty = 1;  
int i = 0;
```

- Non mit Do-While (`FacultyDoWhileLoop.java`):

```
int faculty = 1;  
int i = 0;  
do {
```

```
} while (i <= n);
```



- Non mit Do-While (`FacultyDoWhileLoop.java`):

```
int faculty = 1;  
int i = 0;  
do {  
    if (i != 0) {  
  
    }  
  
} while (i <= n);
```

- Non mit Do-While (`FacultyDoWhileLoop.java`):

```
int faculty = 1;
int i = 0;
do {
    if (i != 0) {
        faculty *= i;
    }

} while (i <= n);
```

- Non mit Do-While (`FacultyDoWhileLoop.java`):

```
int faculty = 1;
int i = 0;
do {
    if (i != 0) {
        faculty *= i;
    }
    System.out.println(i + "!_=_ " + faculty);
    i++;
} while (i <= n);
```

# Übungsblatt 3 - Aufgabe 2 b)

# Übungsblatt 3 - Aufgabe 2 b)

- „Sie möchten alle  $n$  Elemente einer Liste ausgeben.“

## Übungsblatt 3 - Aufgabe 2 b)

- „Sie möchten alle  $n$  Elemente einer Liste ausgeben.“

Da wir hier das Maximum kennen und nacheinander auf alle Elemente zugreifen, empfiehlt sich eine **for**-Schleife.

## Übungsblatt 3 - Aufgabe 2 b)

- „Sie möchten alle  $n$  Elemente einer Liste ausgeben.“

Da wir hier das Maximum kennen und nacheinander auf alle Elemente zugreifen, empfiehlt sich eine **for**-Schleife.

- „Sie möchten solange einzelne Zeichen einlesen, bis ein ‚x‘ eingelesen wird.“

## Übungsblatt 3 - Aufgabe 2 b)

- „Sie möchten alle  $n$  Elemente einer Liste ausgeben.“

Da wir hier das Maximum kennen und nacheinander auf alle Elemente zugreifen, empfiehlt sich eine **for**-Schleife.

- „Sie möchten solange einzelne Zeichen einlesen, bis ein ‚x‘ eingelesen wird.“

Hier kennen wir nicht das Maximum. Da wir aber mindestens einmal einlesen müssen um auf „x“ zu prüfen, empfiehlt sich **do-while**.



## Übungsblatt 3 - Aufgabe 2 b)

- „Sie möchten alle  $n$  Elemente einer Liste ausgeben.“

Da wir hier das Maximum kennen und nacheinander auf alle Elemente zugreifen, empfiehlt sich eine **for**-Schleife.

- „Sie möchten solange einzelne Zeichen einlesen, bis ein ‚x‘ eingelesen wird.“

Hier kennen wir nicht das Maximum. Da wir aber mindestens einmal einlesen müssen um auf „x“ zu prüfen, empfiehlt sich **do-while**. An sich ginge aber auch **while** wenn wir im Schleifenkopf einlesen.

# Übungsblatt 3 - Zusatzaufgabe a)

# Übungsblatt 3 - Zusatzaufgabe a)

- Jetzt können wir auch alles über Bord werfen.

## Übungsblatt 3 - Zusatzaufgabe a)

- Jetzt können wir auch alles über Bord werfen.
- Da der Argumenttest nicht ternär gemacht werden kann, sind wir mal nicht fanatisch 😊 ([EvenOddTernary.java](#)):

```
System.out.println(x + ((x % 2) == 0 ? "_ist_gerade" :  
                        "_ist_ungerade"));
```

# Übungsblatt 3 - Zusatzaufgabe b)

## Übungsblatt 3 - Zusatzaufgabe b)

- So schlimm es auch ist, wir können die ternären Operationen verschachteln.

## Übungsblatt 3 - Zusatzaufgabe b)

- So schlimm es auch ist, wir können die ternären Operationen verschachteln.
- Der Argumenttest geht aber immer noch nicht ([DivisibleTernary.java](#)):

```
String output = (x % 2) == 0 ? "_ist_durch_2_teilbar"  
                : (x % 3) == 0 ? "_ist_durch_3_teilbar"  
                : "_ist_weder_durch_2_noch_3_teilbar";
```

# Exkurs: (Array-)Kellerspeicher



# Exkurs: (Array-)Kellerspeicher

- Einfache Datenstruktur (Stack). (Wir werden später mehr kennenlernen)

# Exkurs: (Array-)Kellerspeicher

- Einfache Datenstruktur (Stack). (Wir werden später mehr kennenlernen)
- Repräsentiert durch Array und Zeiger (`int` pointer).

# Exkurs: (Array-)Kellerspeicher

- Einfache Datenstruktur (Stack). (Wir werden später mehr kennenlernen)
- Repräsentiert durch Array und Zeiger (`int` pointer).
- Betrachten wir ein Array mit 5 Elementen (initial pointer = `-1`).

```
int[] arr: 

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|


```

0    1    2    3    4

# Exkurs: (Array-)Kellerspeicher

- Einfache Datenstruktur (Stack). (Wir werden später mehr kennenlernen)
- Repräsentiert durch Array und Zeiger (`int` pointer).
- Betrachten wir ein Array mit 5 Elementen (initial pointer = `-1`).

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

01234

## 1. Hinzufügen von 8

pointer = 0 ▼  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

01234

# Exkurs: (Array-)Kellerspeicher

- Einfache Datenstruktur (Stack). (Wir werden später mehr kennenlernen)
- Repräsentiert durch Array und Zeiger (`int` pointer).
- Betrachten wir ein Array mit 5 Elementen (initial pointer = `-1`).

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

01234

## 1. Hinzufügen von 8

pointer = 0  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

01234

## 2. Hinzufügen von 3

pointer = 1  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

01234

# Exkurs: (Array-)Kellerspeicher

- Einfache Datenstruktur (Stack). (Wir werden später mehr kennenlernen)
- Repräsentiert durch Array und Zeiger (`int pointer`).
- Betrachten wir ein Array mit 5 Elementen (initial `pointer = -1`).

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

0

1

2

3

4

## 1. Hinzufügen von 8

`pointer = 0` ▼  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

0

1

2

3

4

## 3. Hinzufügen von 4

`pointer = 2` ▼  
`int[] arr:`

8	3	4	0	0
---	---	---	---	---

0

1

2

3

4

## 2. Hinzufügen von 3

`pointer = 1` ▼  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

0

1

2

3

4

# Exkurs: (Array-)Kellerspeicher

- Einfache Datenstruktur (Stack). (Wir werden später mehr kennenlernen)
- Repräsentiert durch Array und Zeiger (`int` pointer).
- Betrachten wir ein Array mit 5 Elementen (initial pointer = `-1`).

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

01234

## 1. Hinzufügen von 8

pointer = `0` ▼  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

01234

## 3. Hinzufügen von 4

pointer = `2` ▼  
`int[] arr:`

8	3	4	0	0
---	---	---	---	---

01234

## 2. Hinzufügen von 3

pointer = `1` ▼  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

01234

## 4. Hinzufügen von 9

pointer = `3` ▼  
`int[] arr:`

8	3	4	9	0
---	---	---	---	---

01234

# Exkurs: (Array-)Kellerspeicher

- Einfache Datenstruktur (Stack). (Wir werden später mehr kennenlernen)
- Repräsentiert durch Array und Zeiger (`int` pointer).
- Betrachten wir ein Array mit 5 Elementen (initial pointer = `-1`).

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

01234

## 1. Hinzufügen von 8

pointer = 0  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

01234

## 2. Hinzufügen von 3

pointer = 1  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

01234

## 3. Hinzufügen von 4

pointer = 2  
`int[] arr:`

8	3	4	0	0
---	---	---	---	---

01234

## 4. Hinzufügen von 9

pointer = 3  
`int[] arr:`

8	3	4	9	0
---	---	---	---	---

01234

## 5. Entferne Element

pointer = 2  
`int[] arr:`

8	3	4	0	0
---	---	---	---	---

01234



# Exkurs: (Array-)Kellerspeicher

- Einfache Datenstruktur (Stack). (Wir werden später mehr kennenlernen)
- Repräsentiert durch Array und Zeiger (`int pointer`).
- Betrachten wir ein Array mit 5 Elementen (initial `pointer = -1`).

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

0 1 2 3 4

## 1. Hinzufügen von 8

`pointer = 0` ▼  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

0 1 2 3 4

## 2. Hinzufügen von 3

`pointer = 1` ▼  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

0 1 2 3 4

## 3. Hinzufügen von 4

`pointer = 2` ▼  
`int[] arr:`

8	3	4	0	0
---	---	---	---	---

0 1 2 3 4

## 4. Hinzufügen von 9

`pointer = 3` ▼  
`int[] arr:`

8	3	4	9	0
---	---	---	---	---

0 1 2 3 4

## 5. Entferne Element

`pointer = 2` ▼  
`int[] arr:`

8	3	4	0	0
---	---	---	---	---

0 1 2 3 4

## 6. Entferne Element

`pointer = 1` ▼  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

0 1 2 3 4

# Exkurs: (Array-)Kellerspeicher

# Exkurs: (Array-)Kellerspeicher

- Für unser Beispiel, muss die Zahl gar nicht gelöscht werden.

# Exkurs: (Array-)Kellerspeicher

- Für unser Beispiel, muss die Zahl gar nicht gelöscht werden.
- Es reicht, den Zeiger zu verschieben:

```
int[] arr: 

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|


```

0 1 2 3 4

# Exkurs: (Array-)Kellerspeicher

- Für unser Beispiel, muss die Zahl gar nicht gelöscht werden.
- Es reicht, den Zeiger zu verschieben:

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

## 1. Hinzufügen von 8

`pointer = 0` ▼  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

# Exkurs: (Array-)Kellerspeicher

- Für unser Beispiel, muss die Zahl gar nicht gelöscht werden.
- Es reicht, den Zeiger zu verschieben:

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

## 1. Hinzufügen von 8

`pointer = 0` ▼  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

## 2. Hinzufügen von 3

`pointer = 1` ▼  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

# Exkurs: (Array-)Kellerspeicher

- Für unser Beispiel, muss die Zahl gar nicht gelöscht werden.
- Es reicht, den Zeiger zu verschieben:

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

## 1. Hinzufügen von 8

`pointer = 0` ▼  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

## 3. Hinzufügen von 4

`pointer = 2` ▼  
`int[] arr:`

8	3	4	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

## 2. Hinzufügen von 3

`pointer = 1` ▼  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

# Exkurs: (Array-)Kellerspeicher

- Für unser Beispiel, muss die Zahl gar nicht gelöscht werden.
- Es reicht, den Zeiger zu verschieben:

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

01234

## 1. Hinzufügen von 8

`pointer = 0` ▼  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

01234

## 3. Hinzufügen von 4

`pointer = 2` ▼  
`int[] arr:`

8	3	4	0	0
---	---	---	---	---

01234

## 2. Hinzufügen von 3

`pointer = 1` ▼  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

01234

## 4. Hinzufügen von 9

`pointer = 3` ▼  
`int[] arr:`

8	3	4	9	0
---	---	---	---	---

01234



# Exkurs: (Array-)Kellerspeicher

- Für unser Beispiel, muss die Zahl gar nicht gelöscht werden.
- Es reicht, den Zeiger zu verschieben:

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

0

1

2

3

4

## 1. Hinzufügen von 8

`pointer = 0` ▼  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

0

1

2

3

4

## 3. Hinzufügen von 4

`pointer = 2` ▼  
`int[] arr:`

8	3	4	0	0
---	---	---	---	---

0

1

2

3

4

## 5. Entferne Element

`pointer = 2` ▼  
`int[] arr:`

8	3	4	9	0
---	---	---	---	---

0

1

2

3

4

## 2. Hinzufügen von 3

`pointer = 1` ▼  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

0

1

2

3

4

## 4. Hinzufügen von 9

`pointer = 3` ▼  
`int[] arr:`

8	3	4	9	0
---	---	---	---	---

0

1

2

3

4

# Exkurs: (Array-)Kellerspeicher

- Für unser Beispiel, muss die Zahl gar nicht gelöscht werden.
- Es reicht, den Zeiger zu verschieben:

`int[] arr:`

0	0	0	0	0
---	---	---	---	---

0   1   2   3   4

## 1. Hinzufügen von 8

`pointer = 0` ▼  
`int[] arr:`

8	0	0	0	0
---	---	---	---	---

0   1   2   3   4

## 3. Hinzufügen von 4

`pointer = 2` ▼  
`int[] arr:`

8	3	4	0	0
---	---	---	---	---

0   1   2   3   4

## 5. Entferne Element

`pointer = 2` ▼  
`int[] arr:`

8	3	4	9	0
---	---	---	---	---

0   1   2   3   4

## 2. Hinzufügen von 3

`pointer = 1` ▼  
`int[] arr:`

8	3	0	0	0
---	---	---	---	---

0   1   2   3   4

## 4. Hinzufügen von 9

`pointer = 3` ▼  
`int[] arr:`

8	3	4	9	0
---	---	---	---	---

0   1   2   3   4

## 6. Entferne Element

`pointer = 1` ▼  
`int[] arr:`

8	3	4	9	0
---	---	---	---	---

0   1   2   3   4

# Exkurs: Umgekehrte polnische Notation

# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$

# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$
- In der umgekehrten polnischen Notation (UPN) ist dies anders!

# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$
- In der umgekehrten polnischen Notation (UPN) ist dies anders!
- Zuerst die Operanden (  $\langle a \rangle$ ,  $\langle b \rangle$  ), dann der Operator (  $\langle op \rangle$  ).

# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$
- In der umgekehrten polnischen Notation (UPN) ist dies anders!
- Zuerst die Operanden (  $\langle a \rangle$ ,  $\langle b \rangle$  ), dann der Operator (  $\langle op \rangle$  ).
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2\ +\ 3\ 4\ +\ +$

# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$
- In der umgekehrten polnischen Notation (UPN) ist dies anders!
- Zuerst die Operanden ( $\langle a \rangle$ ,  $\langle b \rangle$ ), dann der Operator ( $\langle op \rangle$ ).
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2\ +\ 3\ 4\ +\ +$
  - $(2 * 3) + (20 * 3) \Rightarrow 2\ 3\ *\ 20\ 3\ *\ +$



# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$
- In der umgekehrten polnischen Notation (UPN) ist dies anders!
- Zuerst die Operanden ( $\langle a \rangle$ ,  $\langle b \rangle$ ), dann der Operator ( $\langle op \rangle$ ).
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2\ +\ 3\ 4\ +\ +$
  - $(2 * 3) + (20 * 3) \Rightarrow 2\ 3\ *\ 20\ 3\ *\ +$
  - $13 * 12 * -5 + 4 \Rightarrow 13\ 12\ *\ -5\ *\ 4\ +$

# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$
- In der umgekehrten polnischen Notation (UPN) ist dies anders!
- Zuerst die Operanden ( $\langle a \rangle$ ,  $\langle b \rangle$ ), dann der Operator ( $\langle op \rangle$ ).
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2\ +\ 3\ 4\ +\ +$
  - $(2 * 3) + (20 * 3) \Rightarrow 2\ 3\ *\ 20\ 3\ *\ +$
  - $13 * 12 * -5 + 4 \Rightarrow 13\ 12\ *\ -5\ *\ 4\ +$
- Wir werden uns auf „+“ und „\*“ fokussieren.

# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$
- In der umgekehrten polnischen Notation (UPN) ist dies anders!
- Zuerst die Operanden ( $\langle a \rangle$ ,  $\langle b \rangle$ ), dann der Operator ( $\langle op \rangle$ ).
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2\ +\ 3\ 4\ +\ +$
  - $(2 * 3) + (20 * 3) \Rightarrow 2\ 3\ *\ 20\ 3\ *\ +$
  - $13 * 12 * -5 + 4 \Rightarrow 13\ 12\ *\ -5\ *\ 4\ +$
- Wir werden uns auf „+“ und „\*“ fokussieren.
- Anstelle von „\*“ schreiben wir „mult“, anstelle von „+“, „add“.

# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$
- In der umgekehrten polnischen Notation (UPN) ist dies anders!
- Zuerst die Operanden ( $\langle a \rangle$ ,  $\langle b \rangle$ ), dann der Operator ( $\langle op \rangle$ ).
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2 + 3\ 4 + +$
  - $(2 * 3) + (20 * 3) \Rightarrow 2\ 3 * 20\ 3 * +$
  - $13 * 12 * -5 + 4 \Rightarrow 13\ 12 * -5 * 4 +$
- Wir werden uns auf „+“ und „\*“ fokussieren.
- Anstelle von „\*“ schreiben wir „mult“, anstelle von „+“, „add“.
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2\ add\ 3\ 4\ add\ add$

# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$
- In der umgekehrten polnischen Notation (UPN) ist dies anders!
- Zuerst die Operanden ( $\langle a \rangle$ ,  $\langle b \rangle$ ), dann der Operator ( $\langle op \rangle$ ).
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2\ +\ 3\ 4\ +\ +$
  - $(2 * 3) + (20 * 3) \Rightarrow 2\ 3\ *\ 20\ 3\ *\ +$
  - $13 * 12 * -5 + 4 \Rightarrow 13\ 12\ *\ -5\ *\ 4\ +$
- Wir werden uns auf „+“ und „\*“ fokussieren.
- Anstelle von „\*“ schreiben wir „mult“, anstelle von „+“, „add“.
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2\ \text{add}\ 3\ 4\ \text{add}\ \text{add}$
  - $(2 * 3) + (20 * 3) \Rightarrow 2\ 3\ \text{mult}\ 20\ 3\ \text{mult}\ \text{add}$

# Exkurs: Umgekehrte polnische Notation

- Herkömmliche Darstellung für arithmetischer Ausdrücke:  $\langle a \rangle \langle op \rangle \langle b \rangle$
- In der umgekehrten polnischen Notation (UPN) ist dies anders!
- Zuerst die Operanden ( $\langle a \rangle$ ,  $\langle b \rangle$ ), dann der Operator ( $\langle op \rangle$ ).
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2\ +\ 3\ 4\ +\ +$
  - $(2 * 3) + (20 * 3) \Rightarrow 2\ 3\ *\ 20\ 3\ *\ +$
  - $13 * 12 * -5 + 4 \Rightarrow 13\ 12\ *\ -5\ *\ 4\ +$
- Wir werden uns auf „+“ und „\*“ fokussieren.
- Anstelle von „\*“ schreiben wir „mult“, anstelle von „+“, „add“.
  - $1 + 2 + 3 + 4 \Rightarrow 1\ 2\ \text{add}\ 3\ 4\ \text{add}\ \text{add}$
  - $(2 * 3) + (20 * 3) \Rightarrow 2\ 3\ \text{mult}\ 20\ 3\ \text{mult}\ \text{add}$
  - $13 * 12 * -5 + 4 \Rightarrow 13\ 12\ \text{mult}\ -5\ \text{mult}\ 4\ \text{add}$

# Exkurs: Umgekehrte polnische Notation mit Stack

# Exkurs: Umgekehrte polnische Notation mit Stack

- Annahme: Die Ausdrücke mit denen wir arbeiten sind gültig.



# Exkurs: Umgekehrte polnische Notation mit Stack

- Annahme: Die Ausdrücke mit denen wir arbeiten sind gültig.
- Wir arbeiten von links nach rechts.

# Exkurs: Umgekehrte polnische Notation mit Stack

- Annahme: Die Ausdrücke mit denen wir arbeiten sind gültig.
- Wir arbeiten von links nach rechts.
- Wir speichern die Operanden auf dem Stack.

# Exkurs: Umgekehrte polnische Notation mit Stack

- Annahme: Die Ausdrücke mit denen wir arbeiten sind gültig.
- Wir arbeiten von links nach rechts.
- Wir speichern die Operanden auf dem Stack.
- Treffen wir auf einen Operator: (Ein Beispiel kommt mit der Lösung der alten Präsenzaufgabe)

# Exkurs: Umgekehrte polnische Notation mit Stack

- Annahme: Die Ausdrücke mit denen wir arbeiten sind gültig.
- Wir arbeiten von links nach rechts.
- Wir speichern die Operanden auf dem Stack.
- **Treffen wir auf einen Operator:** (Ein Beispiel kommt mit der Lösung der alten Präsenzaufgabe)
  1. entnehmen der obersten beiden Elemente des Stack.

# Exkurs: Umgekehrte polnische Notation mit Stack

- Annahme: Die Ausdrücke mit denen wir arbeiten sind gültig.
- Wir arbeiten von links nach rechts.
- Wir speichern die Operanden auf dem Stack.
- **Treffen wir auf einen Operator:** (Ein Beispiel kommt mit der Lösung der alten Präsenzaufgabe)
  1. entnehmen der obersten beiden Elemente des Stack.
  2. verarbeiten mit dem Operator (Multiplikation/Addition).

# Exkurs: Umgekehrte polnische Notation mit Stack

- Annahme: Die Ausdrücke mit denen wir arbeiten sind gültig.
- Wir arbeiten von links nach rechts.
- Wir speichern die Operanden auf dem Stack.
- Treffen wir auf einen Operator: (Ein Beispiel kommt mit der Lösung der alten Präsenzaufgabe)
  1. entnehmen der obersten beiden Elemente des Stack.
  2. verarbeiten mit dem Operator (Multiplikation/Addition).
  3. ablegen der Summe/des Produktes auf dem Stack.

# Eine alte Präsenzaufgabe

3

Do da' UPN

# Eine alte Präsenzaufgabe

3

Do da' UPN

Schreiben Sie ein Programm, dass mit (korrekter) UPN gefüttert, den oberen Wert des Stacks ausgibt.



# Eine alte Präsenzaufgabe

3

Do da' UPN

Schreiben Sie ein Programm, dass mit (korrekter) UPN gefüttert, den oberen Wert des Stacks ausgibt. Die Operanden sollen Integer und die Operatoren `add` und `mult` sein. Sie können eine Zeichenkette mit „`Integer.parseInt(x)`“ in einen Integer umwandeln.

# Eine alte Präsenzaufgabe

3

Do da' UPN

Schreiben Sie ein Programm, dass mit (korrekter) UPN gefüttert, den oberen Wert des Stacks ausgibt. Die Operanden sollen Integer und die Operatoren `add` und `mult` sein. Sie können eine Zeichenkette mit „`Integer.parseInt(x)`“ in einen Integer umwandeln. Sie können den Stack auf eine kleine Zahl begrenzen.

# Eine alte Präsenzaufgabe

3

Do da' UPN

Schreiben Sie ein Programm, dass mit (korrekter) UPN gefüttert, den oberen Wert des Stacks ausgibt. Die Operanden sollen Integer und die Operatoren `add` und `mult` sein. Sie können eine Zeichenkette mit „`Integer.parseInt(x)`“ in einen Integer umwandeln. Sie können den Stack auf eine kleine Zahl begrenzen.

```
java ReversePolishNotation 2 3 mult 20 3 mult add:
```

# Eine alte Präsenzaufgabe

3

Do da' UPN

Schreiben Sie ein Programm, dass mit (korrekter) UPN gefüttert, den oberen Wert des Stacks ausgibt. Die Operanden sollen Integer und die Operatoren `add` und `mult` sein. Sie können eine Zeichenkette mit „`Integer.parseInt(x)`“ in einen Integer umwandeln. Sie können den Stack auf eine kleine Zahl begrenzen.

```
java ReversePolishNotation 2 3 mult 20 3 mult add:
```

```
66
```

# Eine alte Präsenzaufgabe - Lösung

# Eine alte Präsenzaufgabe - Lösung

- Das volle Programm findet sich hier: [ReversePolishNotation.java](#).

# Eine alte Präsenzaufgabe - Lösung

- Das volle Programm findet sich hier: [ReversePolishNotation.java](#).
- Initialisierung:

```
1 int[] stack = new int[100];  
2 int pointer = -1;
```

# Eine alte Präsenzaufgabe - Lösung

- Das volle Programm findet sich hier: [ReversePolishNotation.java](#).
- Initialisierung:

```
1 int[] stack = new int[100];  
2 int pointer = -1;
```

- Wir iterieren über alle Kommandozeilenargumente:

```
3 for (String arg : args) {  
4     // Cast epic magic  
5 }  
6  
7
```



# Eine alte Präsenzaufgabe - Lösung

- Das volle Programm findet sich hier: [ReversePolishNotation.java](#).
- Initialisierung:

```
1 int[] stack = new int[100];  
2 int pointer = -1;
```

- Wir iterieren über alle Kommandozeilenargumente:

```
3 for (String arg : args) {  
4     // Cast epic magic  
5 }  
6  
7 System.out.println(stack[pointer]);
```

# Eine alte Präsenzaufgabe - Lösung

# Eine alte Präsenzaufgabe - Lösung

- Die epic magic der Addition:

4

# Eine alte Präsenzaufgabe - Lösung

## ■ Die epic magic der Addition:

```
4  if (arg.equals("add")) {  
5  
   }
```

# Eine alte Präsenzaufgabe - Lösung

## ■ Die epic magic der Addition:

```
4  if (arg.equals("add")) {  
5      pointer -= 1;  
6      stack[pointer] += stack[pointer + 1];  
7  }
```

# Eine alte Präsenzaufgabe - Lösung

- Die epic magic der Addition:

```
4  if (arg.equals("add")) {  
5      pointer -= 1;  
6      stack[pointer] += stack[pointer + 1];  
7  }
```

- Die epic magic der Multiplikation:

```
10
```

# Eine alte Präsenzaufgabe - Lösung

## ■ Die epic magic der Addition:

```
4  if (arg.equals("add")) {  
5      pointer -= 1;  
6      stack[pointer] += stack[pointer + 1];  
7  }
```

## ■ Die epic magic der Multiplikation:

```
10 else if (arg.equals("mult")) {  
11  
    }
```

# Eine alte Präsenzaufgabe - Lösung

- Die epic magic der Addition:

```
4  if (arg.equals("add")) {  
5      pointer -= 1;  
6      stack[pointer] += stack[pointer + 1];  
7  }
```

- Die epic magic der Multiplikation:

```
10 else if (arg.equals("mult")) {  
11     pointer -= 1;  
12     stack[pointer] *= stack[pointer + 1];  
13 }
```



# Eine alte Präsenzaufgabe - Lösung

# Eine alte Präsenzaufgabe - Lösung

- Die epic magic der Sonstigkeit:

16

# Eine alte Präsenzaufgabe - Lösung

- Die epic magic der Sonstigkeit:

```
16 else { // Annahme: Zahl  
17  
  
}
```

# Eine alte Präsenzaufgabe - Lösung

- Die epic magic der Sonstigkeit:

```
16 else { // Annahme: Zahl
17     pointer += 1;
18     stack[pointer] = Integer.parseInt(arg);
19 }
```

# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

- Betrachten wir ein Beispiel für `3 2 add 5 1 add add 4 mult`

# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

- Betrachten wir ein Beispiel für `3 2 add 5 1 add add 4 mult`

## 1. Einlesen von `3`:

pointer = `0`

`int[]` arr:

3	0	0	0	0
0	1	2	3	4

# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

- Betrachten wir ein Beispiel für `3 2 add 5 1 add add 4 mult`

## 1. Einlesen von `3`:

pointer = `0`

`int[] arr:`

3	0	0	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

## 2. Einlesen von `2`:

pointer = `1`

`int[] arr:`

3	2	0	0	0
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---



# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

- Betrachten wir ein Beispiel für `3 2 add 5 1 add add 4 mult`

## 1. Einlesen von `3`:

pointer = `0`

`int[] arr:`

3	0	0	0	0
0	1	2	3	4

## 2. Einlesen von `2`:

pointer = `1`

`int[] arr:`

3	2	0	0	0
0	1	2	3	4

## 3. Einlesen von `add`:

pointer = `0`

`int[] arr:`

5	2	0	0	0
0	1	2	3	4

# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

- Betrachten wir ein Beispiel für `3 2 add 5 1 add add 4 mult`

## 1. Einlesen von `3`:

pointer = 0

int[] arr:

3	0	0	0	0
0	1	2	3	4

## 4. Einlesen von `5`:

pointer = 1

int[] arr:

5	5	0	0	0
0	1	2	3	4

## 2. Einlesen von `2`:

pointer = 1

int[] arr:

3	2	0	0	0
0	1	2	3	4

## 3. Einlesen von `add`:

pointer = 0

int[] arr:

5	2	0	0	0
0	1	2	3	4

# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

- Betrachten wir ein Beispiel für 3 2 add 5 1 add add 4 mult

## 1. Einlesen von 3:

pointer = 0

int[] arr: 

3	0	0	0	0
0	1	2	3	4

## 4. Einlesen von 5:

pointer = 1

int[] arr: 

5	5	0	0	0
0	1	2	3	4

## 2. Einlesen von 2:

pointer = 1

int[] arr: 

3	2	0	0	0
0	1	2	3	4

## 5. Einlesen von 1:

pointer = 2

int[] arr: 

5	5	1	0	0
0	1	2	3	4

## 3. Einlesen von add:

pointer = 0

int[] arr: 

5	2	0	0	0
0	1	2	3	4

# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

- Betrachten wir ein Beispiel für 3 2 add 5 1 add add 4 mult

## 1. Einlesen von 3:

pointer = 0

int[] arr:

3	0	0	0	0
0	1	2	3	4

## 2. Einlesen von 2:

pointer = 1

int[] arr:

3	2	0	0	0
0	1	2	3	4

## 3. Einlesen von add:

pointer = 0

int[] arr:

5	2	0	0	0
0	1	2	3	4

## 4. Einlesen von 5:

pointer = 1

int[] arr:

5	5	0	0	0
0	1	2	3	4

## 5. Einlesen von 1:

pointer = 2

int[] arr:

5	5	1	0	0
0	1	2	3	4

## 6. Einlesen von add:

pointer = 1

int[] arr:

5	6	1	0	0
0	1	2	3	4

# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

- Betrachten wir ein Beispiel für 3 2 add 5 1 add add 4 mult

## 1. Einlesen von 3:

pointer = 0

int[] arr: 

3	0	0	0	0
0	1	2	3	4

## 2. Einlesen von 2:

pointer = 1

int[] arr: 

3	2	0	0	0
0	1	2	3	4

## 3. Einlesen von add:

pointer = 0

int[] arr: 

5	2	0	0	0
0	1	2	3	4

## 4. Einlesen von 5:

pointer = 1

int[] arr: 

5	5	0	0	0
0	1	2	3	4

## 5. Einlesen von 1:

pointer = 2

int[] arr: 

5	5	1	0	0
0	1	2	3	4

## 6. Einlesen von add:

pointer = 1

int[] arr: 

5	6	1	0	0
0	1	2	3	4

## 7. Einlesen von add:

pointer = 0

int[] arr: 

11	6	1	0	0
0	1	2	3	4

# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

- Betrachten wir ein Beispiel für 3 2 add 5 1 add add 4 mult

## 1. Einlesen von 3:

pointer = 0  
int[] arr: 

3	0	0	0	0
0	1	2	3	4

## 2. Einlesen von 2:

pointer = 1  
int[] arr: 

3	2	0	0	0
0	1	2	3	4

## 3. Einlesen von add:

pointer = 0  
int[] arr: 

5	2	0	0	0
0	1	2	3	4

## 4. Einlesen von 5:

pointer = 1  
int[] arr: 

5	5	0	0	0
0	1	2	3	4

## 5. Einlesen von 1:

pointer = 2  
int[] arr: 

5	5	1	0	0
0	1	2	3	4

## 6. Einlesen von add:

pointer = 1  
int[] arr: 

5	6	1	0	0
0	1	2	3	4

## 7. Einlesen von add:

pointer = 0  
int[] arr: 

11	6	1	0	0
0	1	2	3	4

## 8. Einlesen von 4:

pointer = 1  
int[] arr: 

11	4	1	0	0
0	1	2	3	4

# Eine alte Präsenzaufgabe - Lösung, ein Beispiel

- Betrachten wir ein Beispiel für 3 2 add 5 1 add add 4 mult

## 1. Einlesen von 3:

pointer = 0

int[] arr: 

3	0	0	0	0
0	1	2	3	4

## 2. Einlesen von 2:

pointer = 1

int[] arr: 

3	2	0	0	0
0	1	2	3	4

## 3. Einlesen von add:

pointer = 0

int[] arr: 

5	2	0	0	0
0	1	2	3	4

## 4. Einlesen von 5:

pointer = 1

int[] arr: 

5	5	0	0	0
0	1	2	3	4

## 5. Einlesen von 1:

pointer = 2

int[] arr: 

5	5	1	0	0
0	1	2	3	4

## 6. Einlesen von add:

pointer = 1

int[] arr: 

5	6	1	0	0
0	1	2	3	4

## 7. Einlesen von add:

pointer = 0

int[] arr: 

11	6	1	0	0
0	1	2	3	4

## 8. Einlesen von 4:

pointer = 1

int[] arr: 

11	4	1	0	0
0	1	2	3	4

## 9. Einlesen von mult:

pointer = 0

int[] arr: 

44	4	1	0	0
0	1	2	3	4





# ungsblatt 3

Blatt3.pdf +

11. November 2021, 18:42

unte Gruppen

Mi. 14-16 Uhr - Florian Sihler - Präsenz

## wertungüberblick

Teilnehmer/innen verborgen

Nein

open

7

gegeben

3

gkeitsdatum

Sonntag, 14. November 2021, 23:59

leiber

5 Stunden 3 Minuten



Alle Abgaben anzeigen

Bewertung

# Arbeitsblatt 3

Arbeitsblatt3.pdf +

11. November 2021, 18:42

Arbeitsgruppen

Mi. 14-16 Uhr - Florian Sihler - Präsenz

## Leistungsüberblick

Teilnehmer/innen verborgen

Nein

Erreichte Punkte

7

Erreichte Punkte

4

Erreichte Punkte

Sonntag, 14. November 2021, 23:59

Erreichte Punkte

4 Stunden 4 Minuten



Alle Abgaben anzeigen

Bewertung

# Klausurblatt 3

Blatt3.pdf +

11. November 2021, 18:42

Ante Gruppen

Mi. 14-16 Uhr - Florian Sihler - Präsenz

## Leistungsüberblick

Teilnehmer/innen verborgen

Nein

open

7

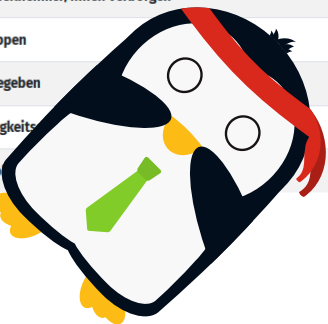
gegeben

5

gkeits

Sonntag, 14. November 2021, 23:59

3 Stunden 6 Minuten



Alle Abgaben anzeigen

Bewertung

# Arbeitsblatt 3

Blatt3.pdf +

11. November 2021, 18:42

Ante Gruppen Mi. 14-16 Uhr - Florian Sihler - Präsenz

## Leistungsüberblick

Teilnehmer/innen vorhanden

Nein

Erreichte Punkte

7

Erreichte Punkte

6

Erreichte Punkte

Sonntag, 14. November 2021, 23:59

Erreichte Punkte

1 Stunde 26 Minuten



Alle Abgaben anzeigen

Bewertung

# Arbeitsblatt 3

Blatt3.pdf +

11. November 2021, 18:42

Ante Gruppen Mi. 14-16 Uhr - Florian Sihler - Präsenz

## Leistungsüberblick

Teilnehmer/innen von

Nein

open

7

gegeben

6

gl

Sonntag, 14. November 2021, 23:59

lt

1 Stunde



Alle Abgaben anzeigen

Bewertung

# ungsblatt 3

Blatt3.pdf +

11. November 2021, 18:42

nte Gruppen

Mi. 14-16 Uhr - Flo

wertungüberbl

Teilnehmer/innen

Nein

open

7

egel

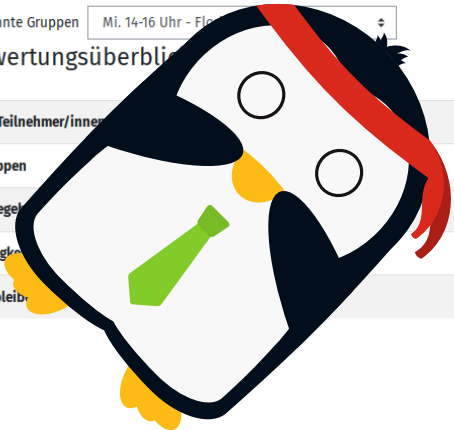
6

gke

Sonntag, 14. November 2021, 23:59

leib

48 Minuten 32 Sekunden



Alle Abgaben anzeigen

Bewertung

# ungsblatt 3

Blatt3.pdf +

11. November 2021, 18:42

nte Gruppen Mi. 14-16 Uhr

wertungüberb

Teilnehmer/in

Nein

open

7

egel

6

gk

Sonntag, 14. November 2021, 23:59

leibende Zeit

30 Minuten 59 Sekunden

Alle Abgaben anzeigen

Bewertung



## ungsblatt 3

Blatt3.pdf +

nte Gruppen Mi. 14-16

wertungsbü

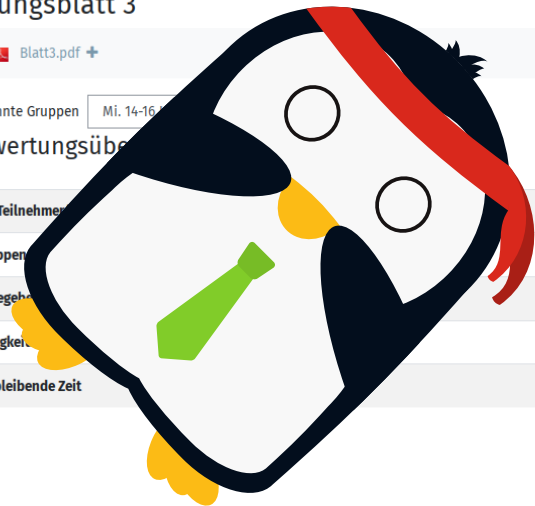
Teilnehmer

open

gebe

gker

leibende Zeit



Nein

7

6

Sonntag, 14. November 2021, 23:59

27 Minuten 30 Sekunden

Alle Abgaben anzeigen

Bewertung



## ungsblatt 3

Blatt3.pdf +

nte Gruppen Mi. 14

wertungsl

Teilneh

open

ge.

gkeitsd

leibende Zeit



Nein

7

6

Sonntag, 14. November 2021, 23:59

17 Minuten 19 Sekunden

Alle Abgaben anzeigen

Bewertung

## ungsblatt 3

Blatt3.pdf +

ante Gruppen

wertungs

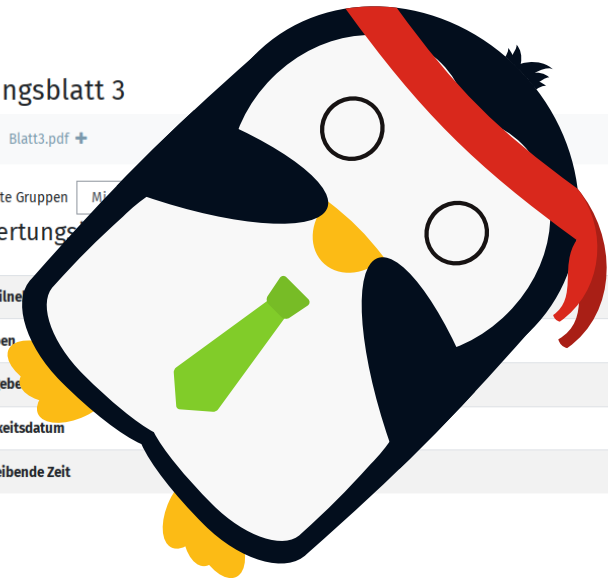
teilne

open

gebe

gkeitsdatum

leibende Zeit



Nein

7

6

Sonntag, 14. November 2021, 23:59

6 Minuten 38 Sekunden

Alle Abgaben anzeigen

Bewertung

## ungsblatt 3

Blatt3.pdf +

nte Gruppen

wertung

teilneh

oper.

geben

gkeitsdatum

leibende Zeit



Nein

7

7

Sonntag, 14. November 2021, 23:59

2 Minuten 41 Sekunden

Alle Abgaben anzeigen

Bewertung



