# Standardkonstrukte in Java

Robert Buchholz, Sven Schneider

JavaKurs 2006 – 1. Tag
Freitagsrunde / Tutoren

03. April 2006

# Was lernen wir heute noch kennen?

1. Methoden
   - Definition einer Methode
   - Aufruf einer Methode
2. Datenstruktur Array
   - Definition und Anwendung
   - Stolpersteine bei Arrays
3. Schleifen
   - While–Schleife
   - For-Schleife
4. Namen
   - Wofür wir Namen brauchen
5. Testen
   - Wie teste ich mein Programm?
   - Noch ein Beispiel

# Probleme bei den Vormittagsaufgaben

```
if (restgeld > anzahlBuch * preisBuch){
  restgeld = restgeld - anzahlBuch * preisBuch;
  System.out.println("Gekauft: "+
    (preisBuch * anzahlBuch) + ": "
    + anzahlBuch + "x "+ nameBuch);
}
if (restgeld > anzahlCD * preisCD){
  restgeld = restgeld - anzahlCD * preisCD;
  System.out.println("Gekauft: "+
    (preisCD * anzahlCD) + ": "
    + anzahlCD + "x "+ nameCD);
}
if (restgeld > anzahlDVD * preisDVD){
  restgeld = restgeld - anzahlDVD * preisDVD;
  System.out.println("Gekauft: "+
    (preisDVD * anzahlDVD) + ": "
    + anzahlDVD + "x "+ nameDVD);
}
```

# Das Beispiel – verbessert

**Was wir eigentlich machen wollen**

```
restgeld = kaufe(restgeld, nameBuch, anzahlBuch, preisBuch);
restgeld = kaufe(restgeld, nameCD, anzahlCD, preisCD);
restgeld = kaufe(restgeld, nameDVD, anzahlDVD, preisDVD);
```

```
static double kaufe (double restgeld, String name,
                     int anzahl, double preis) {
  if(restgeld > anzahl * preis){
    restgeld = restgeld - anzahl * preis;
    System.out.println("Gekauft: "+ (preis * anzahl)
      + ": "+ anzahl + "x "+ name);
  }
  return restgeld;
}
```

# Allgemeines zu Methoden

Eine Methode

- bündelt eine Menge von Anweisungen in sich
- hat einen Namen, mit der sie an verschiedenen Stellen aufgerufen werden kann
- bekommt Eingaben und liefert Ergebnisse
- dient der Strukturierung im Programm
- verhindert Code-Wiederholungen

**Probleme:**

- An mehreren Stellen im Programm steht ähnlicher Code
- Der Code wird sehr lang

# Wieso Abstraktion?

- Information hiding: Dem Aufrufenden kann es egal sein, wie aus den Eingaben die Rückgabe berechnet wird
- Wartung: Wenn sich Fehler einschleichen, so müssen diese nur an einer Stelle korrigiert werden
- Testmöglichkeiten: Einzelne Programmteile können als Methoden sehr leicht mit verschiedenen Eingaben getestet werden.
- Strukturierung: Der Programmierer kann Probleme in Teilprobleme zerlegen und diese einzeln lösen.

# Methodendeklaration in Java

```
public static double sin ( double x ) {
        ...
}
```

# Methodendeklaration in Java

Schlüsselwörter

```
public static double sin ( double x ) {
        ...
}
```

z.B. `public`, `private`, `static`

# Methodendeklaration in Java

Rückgabetyp

```
public static double sin ( double x ) {
        ...
}
```

z.B. `int`, `boolean`, `void`

# Methodendeklaration in Java

Name der Methode

```
public static double sin ( double x ) {
        ...
}
```

Namen müssen selbsterklärend sein!

# Methodendeklaration in Java

Parameterliste

```
public static double sin ( double x ) {
        ...
}
```

Mehrere Parameter durch Komma getrennt.
Wenn keine Parameter, dann: `sin ( )`

# Methodendeklaration in Java

Schlüsselwörter

Name

```
public static double sin ( double x ) {
        ...
}
```

Rückgabetyp

Parameterliste

# Definiton einer Methode (1)

`return` gibt einen Wert zurück ...

## Beispiel

```
static int addPositiveNumbers (int a, int b) {
        if (a == 0) {
                return b;
        }
        return addPositiveNumbers(a - 1, b + 1);
}
```

# Definiton einer Methode (2)

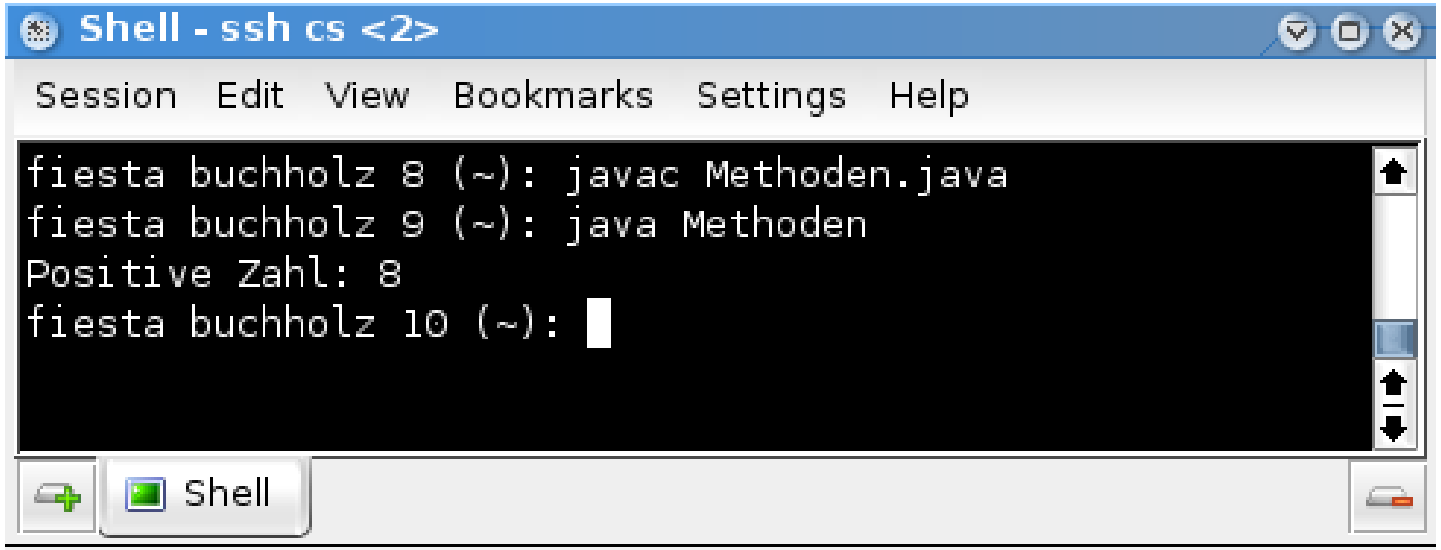
...oder bricht die Methode ab

**Beispiel**

```
static void printPositive(int zahl) {
        if (zahl < 0) {
                return;
        }
        System.out.println("Positive Zahl: "+ zahl);
}
```

# Aufruf einer Methode

```
int summe = addPositiveNumbers(3, 5);
printPositive(summe);
```



# Aufruf: Call by value

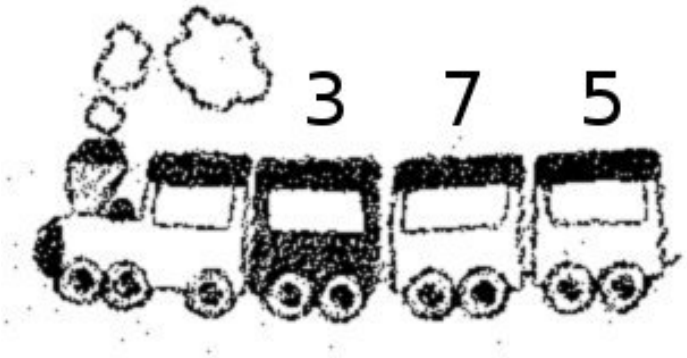
Methoden arbeiten mit Kopien der Parameter.

```
static void zieheEinsAb(int num) {
        num = num - 1;
}
```

```
...
int x = 10;
zieheEinsAb(x);
System.out.println(x); // gibt 10 aus!
```

# Das Array: Ein Beispiel

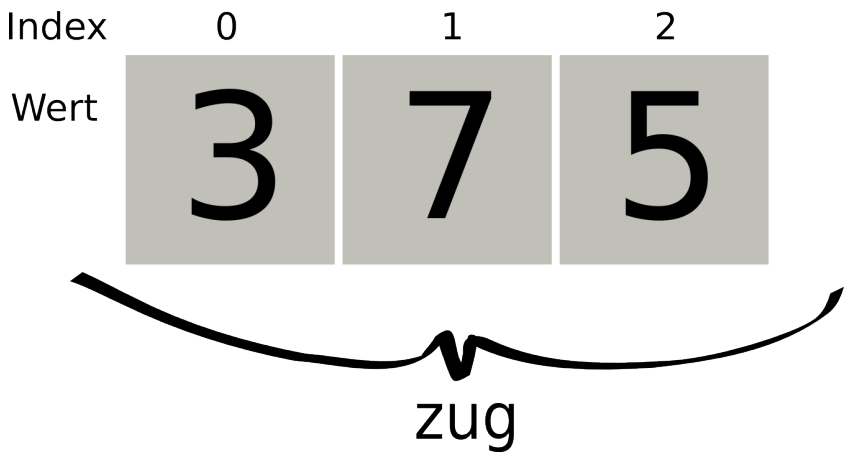
Wir wollen die Anzahl der Fahrgäste eines Zuges speichern.




**Dafür bietet sich das Array an.**

```
int zug[] = new int[3];
zug[0] = 3;
zug[1] = 7;
zug[2] = 5;
```

# Indizierung und Länge




## Das Array sagt mir, wie lang es ist!

```
System.out.println("Länge des Arrays: " + zug.length);
```

## Ausgabe

Länge des Arrays: 3

# Do's and Dont's für das Array

## Don't: Nicht initialisiert!

```
int zug[];
zug[0] = 3; // Fehler!
```

## Don't: Definition nur bis 2!

```
int zug[] = new int[3];
zug[3] = 10; // Fehler!
```

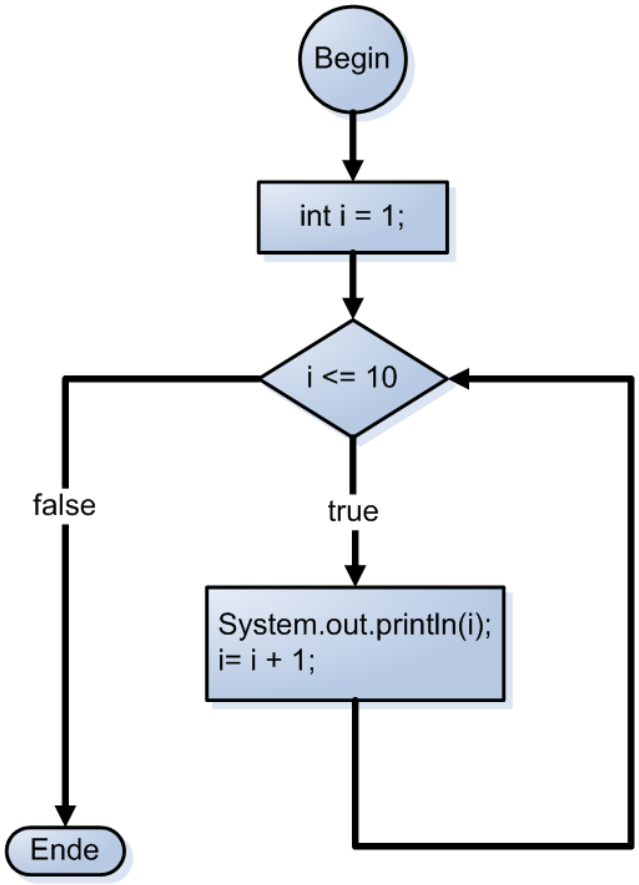
# Was sind Schleifen?

## Beispiel: Die **while**–Schleife

```
int i = 1;
while (i <= 10) {
     System.out.println(i);
     i = i + 1;
}
```

## Ausgabe

```
1
2
3
...
9
10
```



# Von der Rekursion zur Schleife (1)

**addPositiveNumbers berechnet die Summe von zwei positiven Zahlen**

```
static int addPositiveNumbers (int a, int b) {
        if (a == 0) {
                return b;
        }
        return addPositiveNumbers(a - 1, b + 1);
}
```

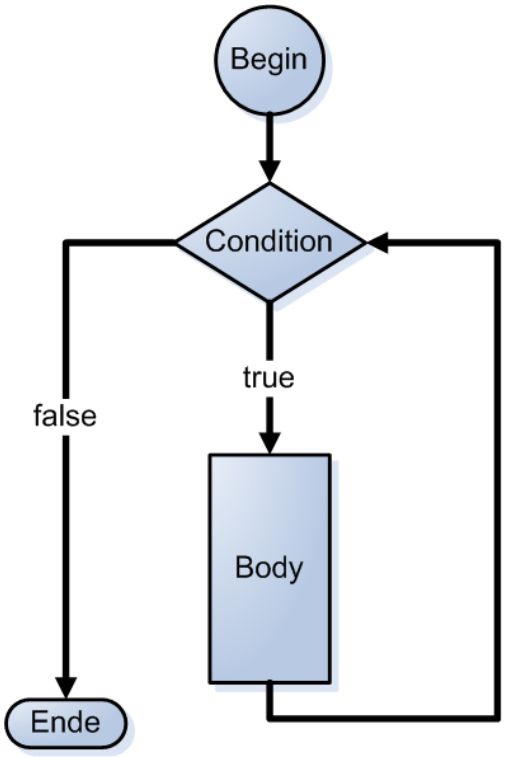
Diese Methode sollte nur positive Zahlen übergeben bekommen. (Warum?)

# Von der Rekursion zur Schleife (2)

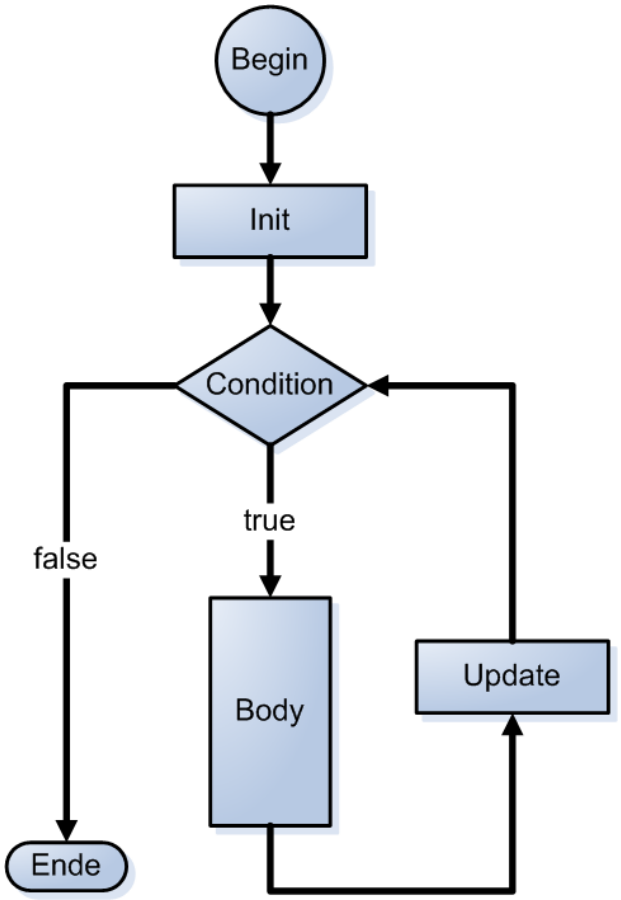
**addPositiveNumbers kann aber auch als Schleife realisiert werden**

```
static int addPositiveNumbers(int a, int b) {
        while (a > 0) {
                a = a - 1;
                b = b + 1;
        }
        return b;
}
```

# Schleifen kompakter mit **for**




while-Schleife




for-Schleife

# Schleifen kompakter mit **for**

## **while**-Schleife

```
int i = 0;
while (i < 4) {
    System.out.println(i);
    i = i + 1;
}
```

## **for**-Schleife

```
for (int i = 0; i < 4; i = i + 1) {
    System.out.println(i);
}
```

- Beide Programmstücke erzeugen dieselbe Ausgabe.
- Die **while**-Schleife lässt sich in eine **for**-Schleife umformen.

# **for**-Schleifen als **while**-Schleifen

## **for**-Schleife

```
for (⟨⟨Init⟩⟩; ⟨⟨Condition⟩⟩; ⟨⟨Update⟩⟩) {
    ⟨⟨Body⟩⟩
}
```

## **while**-Schleife

```
⟨⟨Init⟩⟩
while (⟨⟨Condition⟩⟩) {
    ⟨⟨Body⟩⟩
    ⟨⟨Update⟩⟩
}
```

- Alles, was man mit **for** machen kann, geht auch mit **while**.

# Warum gibt es nun zwei Schleifentypen?

**while**-Schleife, *verständlich*

```
while ( calculationIsRunning() ) {
    System.out.println("Berechnung läuft noch.");
    calculateForOneMinute();
}
```

**for**-Schleife, *unverständlich*

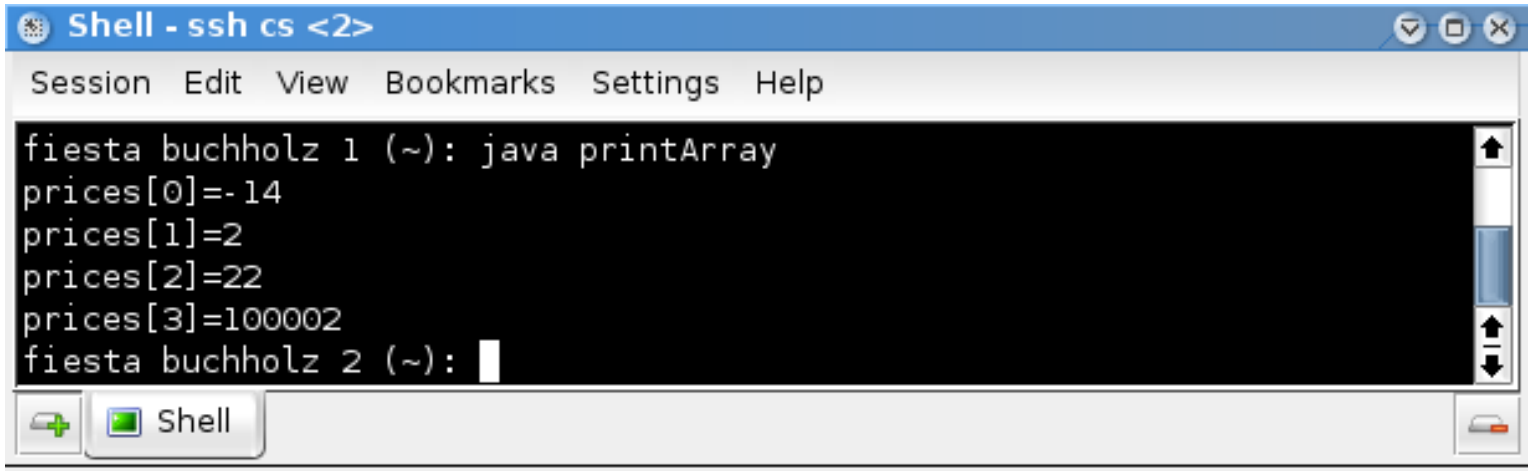
```
for (; calculationIsRunning(); ) {
    System.out.println("Berechnung läuft noch.");
    calculateForOneMinute();
}
```

- **for**- und **while**-Schleifen implizieren eine Semantik
- Wähle denjenigen Schleifentyp, welcher dem Problem am ehesten entspricht.

# Noch zwei Beispiele

## Arrayausgabe

```
int prices[] = new int[] {-14, 2, 22, 100002};
for (int i = 0; i < prices.length; i = i + 1)
    System.out.println("prices[" + i + "]=" + prices[i]);
```




## Ausgabe aller geraden Zahlen in $\{0, .., 30\}$

```
for (int i = 0; i <= 30; i = i + 2)
    System.out.println(i);
```

# Gute Schleifenbedingungen finden

## Wie oft wird diese Schleife durchlaufen?

```
int i = 11;
...
while ( i != 9 ) {
    i = i + 1;
}
```

Was der Programmierer wohl eigentlich meinte:

## Keine Endlosschleife

```
int i = 11;
...
while ( i < 9 ) {
    i = i + 1;
}
```

# Das Problem mit den Gleitkommazahlen

## Dieses Programm gibt zehn Zahlen aus?

```
double d = 0.0;
while ( d != 1.0 ) {
    d = d + 0.1;
    System.out.println(d);
}
```

Ausgabe:

```
...
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
...
```

# Was berechnet diese Methode?

```
static boolean myMethod(int a) {
    if (a == 0)
        return true;
    else if (a > 0) {
        boolean boolValue = myMethod(a - 1);
     // boolValue is the result of myMethod(a - 1)
        return !boolValue;
    }
    else {
        boolean boolValue = myMethod(a + 1);
     // we return the complement of boolValue
        return !boolValue;
    }
}
```

# Dieselbe Methode — sieht man es jetzt?

```
static boolean isEven(int number) {
    if (number == 0)
        return true;
    else if (number > 0) {
        boolean predecessorIsEven = isEven(number - 1);
     // predecessor is even <=> number is not even
        return !predecessorIsEven;
    }
    else {
        boolean successorIsEven = isEven(number + 1);
     // successor is even <=> number is not even
        return !successorIsEven;
    }
}
```

# Was macht gute Namen aus?

Gute Namen ...

- identifizieren lokale Variable oder Methoden.
- müssen deutlich machen, was sich hinter ihnen versteckt (selbsterklärend, aussagekräftig).
- erhöhen die Lesbarkeit des Programms auch dadurch, dass sie kurz und prägnant sind: 3 Worte max.

Kommentare ...

- können die Namen sinnvoll unterstützen.
- enthalten die Informationen, die nicht im Namen allein stecken können: Was denke ich mir beim Schreiben des Programms?
- erklären den Sinn eines Code-Stückes für die Programmkorrektheit.
- erklären nicht die Syntax.

# 1.: Testwerte finden und Ergebnisse überlegen

## Code:

```
static double pow(double base, int exp) {
    if (exp % 2 == 0 && exp > 0) {
        return pow(base * base, exp / 2);
    } if (exp > 0) {
        return base * pow(base, exp - 1);
    } return 1;
}
```

## erwartete Ausgabe

```
pow(10, 2)= 100   pow(2, 2)= 4     pow(3, 2)= 9
pow(2, 8)= 256    pow(2, 9)= 512   pow(25, 2)= 625
pow(0, 0)= 1      pow(0, 1)= 0     pow(0, -1)= infinity
pow(1, 0)= 1      pow(1, 1)= 1     pow(1, -1)= 1
pow(-1, 0)= 1     pow(-1, 1)= -1   pow(-1, -1)= -1
```

# 2.: Testwerte eingeben und mit Erwartungswert vergleichen

## Code:

```
static double pow(double base, int exp) {
    if (exp % 2 == 0 && exp > 0) {
        return pow(base * base, exp / 2);
    } if (exp > 0) {
        return base * pow(base, exp - 1);
    } return 1;
}
```

## tatsächliche Ausgabe

```
pow(10, 2)= 100   pow(2, 2)= 4    pow(3, 2)= 9
pow(2, 8)= 256    pow(2, 9)= 512  pow(25, 2)= 625
pow(0, 0)= 1      pow(0, 1)= 0    pow(0, -1)= 1
pow(1, 0)= 1      pow(1, 1)= 1    pow(1, -1)= 1
pow(-1, 0)= 1     pow(-1, 1)= -1  pow(-1, -1)= 1
```

# 3.: Fehler finden und beheben. Dann Test wiederholen!

Code:

```
static double pow(double base, int exp) {
    if (exp % 2 == 0 && exp > 0) {
        return pow(base * base, exp / 2);
    } if (exp > 0) {
        return base * pow(base, exp - 1);
    } if (exp < 0) {
        return 1 / pow(base, -exp);
    } return 1;
}
```

- Diese Methode scheint nun robust zu sein.
- Das heißt, es gibt keine Eingabe, die ein unerwartetes Ereignis auslöst.

# Welche Variablen lässt man sich wann/wo ausgeben?

- Die Wahl der Variablen hängt immer vom Problem ab.
- Bei komplexen Abläufen lohnt es sich, den Fehler einzugrenzen.

**Beispiel:**

```
int i = 1;
System.out.println("bis hier kommt das Programm..");
while (i < 5) {
    i = (i + 1) % 3;
}
System.out.println("dies wird nicht ausgegeben..");
```

- Problematisch ist das Testen mit Ausgaben, wenn der zu testende Code häufig ausgeführt wird.
- Die sich ergebenden Datenmengen müssen vom Tester verarbeitet werden können.

# Fazit

Was ist Testen?

- Testen ist einer der wichtigsten Schritte der Softwareentwicklung.
- Testen kann viele grobe Fehler finden und so Kosten vermindern.
- Fehler sind unabhängig von Programmkomplexität und Programmierfähigkeiten.

Was ist Testen nicht?

- Testen stellt Fehlerfreiheit **NICHT** sicher.
- Testen stellt Lesbarkeit **NICHT** sicher.
- Testen stellt Effizienz **NICHT** sicher.
- Selbst ein korrektes Ergebnis kann Zufall sein.

# 1.: Testwerte finden und Ergebnisse überlegen

## Code:

```
static int fak(int x) {
    if (x == 0)
        return 1;
    return x * fak(x - 1);
}
```

## erwartete Ausgabe

```
fak(4)= 24
fak(3)= 6
fak(2)= 2
fak(1)= 1
fak(0)= 1
fak(-1)= -1 // die Methode muss ja etwas zurückgeben.
```

# 2.: Testwerte eingeben und mit Erwartungswert vergleichen

## Code:

```
static int fak(int x) {
    if (x == 0)
        return 1;
    return x * fak(x - 1);
}
```

## erwartete Ausgabe

```
fak(4)= 24
fak(3)= 6
fak(2)= 2
fak(1)= 1
fak(0)= 1
fak(-1)= kein Ergebnis?
```

# 3.: Fehler finden und beheben. Dann Test wiederholen!

**Code: (korrigiert)**

```
static int fak(int x) {
    if (x == 0)
        return 1;
    if (x < 0)
        return -1;
    return x * fak(x - 1);
}
```