

# Kapitel 09

## Programmfluss und Objektinteraktion





# Inhalt von Kapitel 09

## Programmfluss und Objektinteraktion

### 9.1 Programmfluss

Was bedeutet Programmfluss und wie kann man ihn steuern?

- Schleifen
- Bedingungen, bzw. Verzweigungen
- Einschub: Der Debugger
- Rekursion

### 9.2 Methoden- und Objektinteraktion

Wie können Objekte miteinander kommunizieren?

- Methodenaufrufe



## 9.1 Programmfluss

### Was bedeutet Programmfluss?

Aus der ersten Vorlesung ist bekannt:

- Der Java-Quelltext wird durch den Compiler `javac` in eine idealisierte Form von Maschineninstruktionen, den Java-Bytecode, übersetzt.
- Durch den Interpreter `java` wird das Programm in seiner Form als Java-Bytecode ausgeführt.
- Das eigentliche Programm läuft dann durch die Hardware, d. h. Speicherzugriffe, Additionen, usw., ab.



# Programmfluss

## Was bedeutet Programmfluss?

Eine interne Uhr zerlegt die Zeit in einzelne Taktzyklen.

- In jedem Taktzyklus wird eine Maschinenanweisung abgearbeitet.
- Im Prinzip werden die Maschinenanweisungen Takt für Takt in der Reihenfolge ihrer Speicheradressen abgearbeitet.
  - Vergleichbar mit dem Lesen eines Buches.
- Sprunganweisungen mit Sprungbedingungen ermöglichen Abweichungen von der seriellen Abfolge.
  - Verweis auf vorherige Speicheradresse erzeugt eine Schleife
  - Verweis auf spätere Speicheradresse erzeugt eine Verzweigung



# Programmfluss

## Was bedeutet Programmfluss?

Die Verarbeitung von Java-Bytecode durch den Interpreter läuft im Prinzip gleich ab, wie die Verarbeitung von Maschinencode.

- Elementare Anweisungen im Quelltext werden in eine oder mehrere Anweisungen in Java-Bytecode übersetzt.
- Verzweigungen und Schleifen werden in bedingte und unbedingte Sprünge übersetzt.

## Programmfluss

Programmfluss bezeichnet die Abfolge der verarbeiteten Anweisungen zur Laufzeit.

- Der Programmfluss kann statisch sein.
- Der Programmfluss kann aber auch eine dynamische Reihenfolge von Anweisungen innerhalb eines statischen Programmtextes erzeugen.



# Programmfluss

## Allgemeiner Zusammenhang zwischen Programmstruktur und Programmfluss.

- Sind keine Schleifen, Verzweigungen, Methodenaufrufe, etc. vorhanden, folgt der Programmfluss strikt der sequentiellen Programmstruktur.
  - sequentiell meint die Reihenfolge der Anweisungen im Quelltext
- Schleifen und Verzweigungen sind die einfachsten Konstrukte zur dynamischen Steuerung des Programmflusses.
  - folgt nicht mehr der sequentiellen Programmstruktur.
- Diese Konstrukte lassen sich auf bedingte und unbedingte Sprungbefehle zurückführen.



# Steuerung des Programmflusses

## Schleifen

Was sind Schleifen und welche gibt es?

Schleifen wiederholen bestimmte Anweisungen, ohne dass diese erneut implementiert werden müssen.

Es gibt

- `while`-Schleifen
- `do-while`-Schleifen
- `for`-Schleifen

### Schleife

Eine Schleife kann benutzt werden, um einen Block von Anweisungen wiederholt ausführen zu lassen, ohne dass die Anweisungen mehrfach in den Quelltext geschrieben werden müssen.



# Steuerung des Programmflusses

## Schleifen

### while-Schleifen

Diese Art der Schleifen wird durch das Schlüsselwort `while` eingeleitet. Weitere Bestandteile sind

- die Schleifenbedingung: ein boolescher Ausdruck, mit dem ermittelt wird, ob die Schleife noch mindestens ein weiteres Mal ausgeführt wird.
- der Schleifenrumpf: Wenn die Schleifenbedingung das Ergebnis `true` zurückliefert, werden alle Anweisungen im Schleifenrumpf der Reihe nach ausgeführt. Der Schleifenrumpf ist ein Block.

```
1 while (Schleifenbedingung) {  
2     Schleifenrumpf  
3 }
```





# Steuerung des Programmflusses

## Schleifen

### while-Schleifen

Ein simples Beispiel:

- Die Schleife gibt alle geraden Zahlen bis 28 aus.
- Der Block von Zeile 3 bis 4 wird so lang ausgeführt, bis die Bedingung in Klammern in Zeile 2 falsch wird, d. h. `index 30` wird.

```
1 int index = 0;
2 while ( index != 30 ){
3     System.out.println(index);
4     index += 2;
5 }
```



# Steuerung des Programmflusses

## Schleifen

### while-Schleifen

Ein etwas anderes Beispiel:

- Was passiert?
- Wie sieht die Ausgabe aus?

```
1 double d = 0.0;
2 while ( d != 1.0 )
3 {
4     d += 0.1;
5     System.out.println( d );
6 }
```



# Steuerung des Programmflusses

## Schleifen

### while-Schleifen

Die Ausgabe zum vorherigen

Beispiel:

- Endlosschleife!!
- Warum?

Lösung des Fehlers:

- Verwendung des < oder > Operators

```
while ( d <= 1.0 )
```

```
0.1
0.2
0.300000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
1.3
```



# Steuerung des Programmflusses

## Schleifen

### do-while-Schleifen

Diese Art der Schleifen wird durch die Schlüsselwörter `do` und `while` begrenzt. Weitere Bestandteile sind

- der Schleifenrumpf: Alle Anweisungen im Schleifenrumpf werden der Reihe nach ausgeführt. Wenn die Schleifenbedingung das Ergebnis `true` zurückliefert, wird der Schleifenrumpf erneut ausgeführt. Der Schleifenrumpf ist ein Block.
- die Schleifenbedingung: vgl. Schleifenbedingung der `while`-Schleife

```
1 do{  
2   Schleifenrumpf  
3 }  
4 while(Schleifenbedingung); // Vorsicht: Beachte das Semikolon!
```



# Steuerung des Programmflusses

## Schleifen

### while-Schleife

```
1 int i = 10;
2 while ( i > 0 && i < 10 ){
3     System.out.println(i);
4     i--;
5 }
```

### do-while-Schleife

```
1 int j = 10;
2 do {
3     System.out.println(j);
4     j--;
5 }
6 while ( j > 0 && j < 10 );
```

Welche Ausgaben erzeugen die beiden Beispiele?

Wo liegen die Unterschiede?

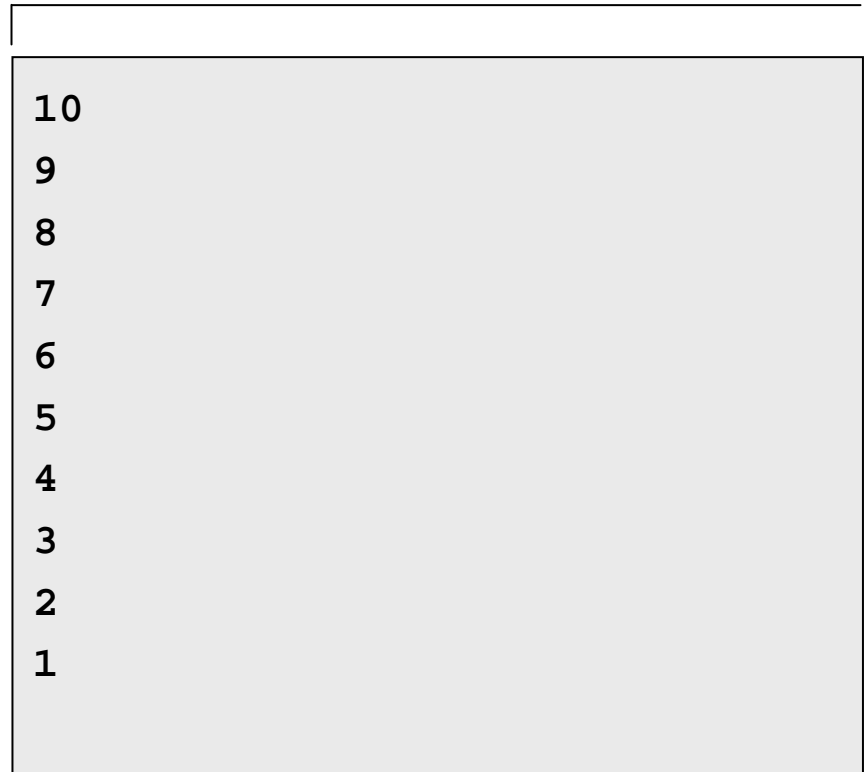
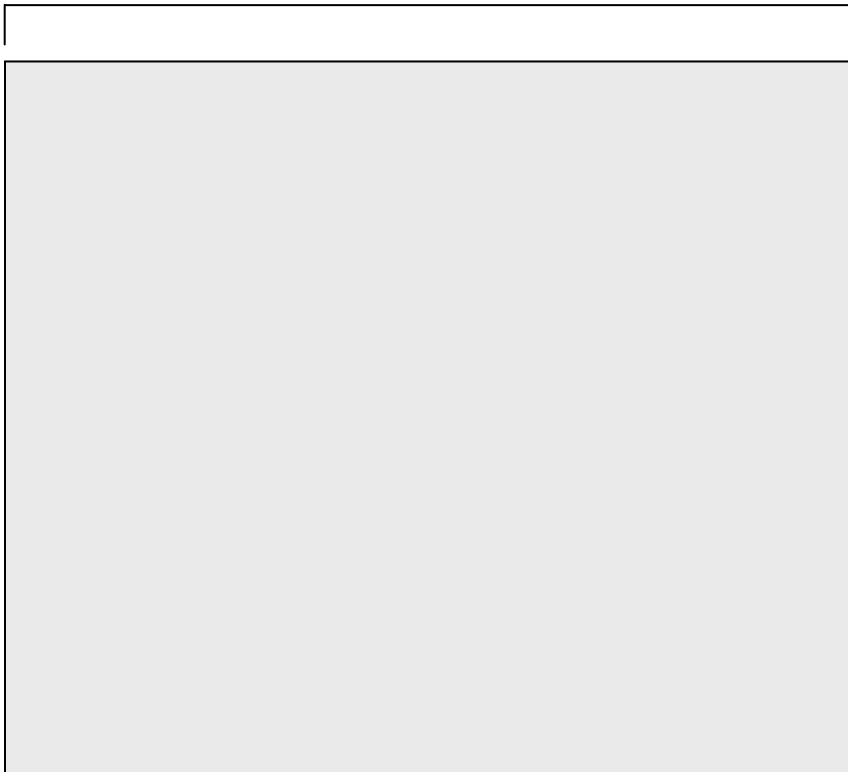


# Steuerung des Programmflusses

## Schleifen

**while-Schleife Ausgabe**

**do-while-Schleife Ausgabe**





# Steuerung des Programmflusses

## Schleifen

Der Unterschied zwischen `while`-Schleifen und `do-while`-Schleifen

Der Schleifenrumpf einer `while`-Schleife wird nur dann ausgeführt, wenn die Schleifenbedingung `true` zurückliefert.

- Eine `while`-Schleife kann durchlaufen werden, muss es aber nicht.

Der Schleifenrumpf einer `do-while`-Schleife wird einmal ausgeführt, bevor die die Schleifenbedingung geprüft wird. Liefert diese `true` zurück, wird der Schleifenrumpf erneut ausgeführt.

- Eine `do-while`-Schleife wird mindestens einmal durchlaufen.



# Steuerung des Programmflusses

## Schleifen

### for-Schleifen

Diese Art der Schleifen wird durch das Schlüsselwort `for` eingeleitet.

Weitere Bestandteile sind

- die Initialisierung: Bevor der Schleifenrumpf ausgeführt wird, wird eine Anweisung zur Initialisierung genau einmal ausgeführt.
- die Schleifenbedingung: Die Schleifenbedingung wird vor jeder Ausführung der Schleife ausgewertet. Wenn die Bedingung das Ergebnis `true` zurückliefert, wird der Schleifenrumpf einmal ausgeführt.

```
1 for( Initialisierung; Schleifenbedingung; Inkrement){  
2     Schleifenrumpf;  
3 }
```





# Steuerung des Programmflusses

## Schleifen

### for-Schleifen

Fortsetzung der vorherigen Folie.

Weitere Bestandteile sind

- der Schleifenrumpf: Alle Anweisungen im Schleifenrumpf werden der Reihe nach ausgeführt.
- das Inkrement: Die Inkrementanweisung wird nach jeder Ausführung des Schleifenrumpfes ausgeführt.

```
1 for( Initialisierung; Schleifenbedingung; Inkrement){  
2     Schleifenrumpf;  
3 }
```



# Steuerung des Programmflusses

## Schleifen

### for-Schleifen

Eine `for`-Schleife ist äquivalent zu einer `while`-Schleife mit

- Initialisierung einer Zählvariablen vor der Schleife und
- Inkrement der Zählvariablen innerhalb des Schleifenrumpfs.
- Die Schleifenbedingung ist immer gleich. Vgl. Folie 9.13

Initialisierung, Schleifenbedingung und Inkrement müssen immer durch Semikola voneinander getrennt werden.

```
1 for( Initialisierung; Schleifenbedingung; Inkrement){  
2     Schleifenrumpf;  
3 }
```



# Steuerung des Programmflusses

## Schleifen

### for-Schleifen

Ein simples Beispiel (vgl. Übung 3):

- Die Schleife durchläuft das Array `text`.
- Jeder `String` im Array `text` wird in Großbuchstaben umgewandelt.
- Der Modifizierte `String` wird wieder im Array gespeichert.

```
1 String [] text = new String[10];
2 for (int i = 0; i < text.length; i++) {
3     text[i] = text[i].toUpperCase();
4 }
```



# Steuerung des Programmflusses

## Schleifen

### Feinsteuerung im Schleifenrumpf

Bei jeder der drei Schleifenarten, `while`-, `do-while`- und `for`-Schleife, kann die Ausführung der Schleife im Schleifenrumpf durch spezielle Anweisungen noch feiner gesteuert werden.

- `break`: Bricht die ganze Schleife ab, d. h. der Interpreter springt zur ersten Anweisung nach der Schleife.
- `continue`: Bricht die aktuelle Ausführung des Schleifenrumpfs ab und lenkt den Programmfluss zur Auswertung der Schleifenbedingung zurück.



# Steuerung des Programmflusses

## Schleifen

Ein etwas sinnfreies Beispiel

- Welche Ausgabe erzeugt das Beispiel?
- Wie könnte man das Beispiel einfacher gestalten?

```
1 for (int i = 0; i < 100; i++) {  
2     if(i % 2 == 1)  
3         continue;  
4     System.out.println(i);  
5     if(i == 20)  
6         break;  
7 }
```



# Steuerung des Programmflusses

## Schleifen

### Quellcode

```
1 for (int i = 0; i < 100; i++){
2     if(i % 2 == 1)
3         continue;
4     System.out.println(i);
5     if(i == 20)
6         break;
7 }
```

### Ausgabe

```
0
2
4
6
8
10
12
14
16
18
20
```

Ausgabe aller geraden Zahlen von 0 bis 20.

- Zeile 2: ungerade werden ausgelassen
- Zeile 5: bei 20 wird abgebrochen



# Steuerung des Programmflusses

## Schleifen

kompliziert

```
1 for (int i = 0; i < 100; i++){
2     if(i % 2 == 1)
3         continue;
4     System.out.println(i);
5     if(i == 20)
6         break;
7 }
```

einfach

```
1 for(int i = 0; i <= 20; i += 2){
2     System.out.println(i);
3 }
```

Dies nur als ergänzenden Hinweis darauf, dass immer verschiedene Lösungen existieren. Einige Lösungen sind kompliziert, andere einfach.

- Es sollte immer die einfachste Lösung für ein Problem gewählt werden, damit das Programm verständlich bleibt und Fehler vermieden werden.



# Steuerung des Programmflusses

## Bedingungen und Verzweigungen

Was sind Bedingungen und Verzweigungen?

Bedingungen und Verzweigungen führen beim positiver Auswertung bestimmter boolescher Ausdrücke unterschiedliche Codestücke aus.

Es gibt

- `if`-Bedingungen
- `if-else`-Bedingungen
- `switch`-Verzweigungen

### Bedingungen

Eine Bedingung führt eine (von zwei) Aktionen aus, abhängig vom Ergebnis einer Prüfung.

### Verzweigungen

Eine Verzweigung führt in Abhängigkeit eines Werts einen von beliebig vielen möglichen Fällen aus.





# Steuerung des Programmflusses

## Bedingungen und Verzweigungen

### if-Anweisung

Es gibt diese Anweisung in zwei Formen:

- `if`: Wenn die Bedingung erfüllt ist, wird der folgende Block ausgeführt.
- `if-else`: Wenn die Bedingung erfüllt ist, wird der folgende Block ausgeführt. Andernfalls wird der Block nach `else` ausgeführt.

```
1 if (Bedingung) {  
2     Anweisungsblock  
3 }
```

```
1 if (Bedingung) {  
2     Anweisungsblock1  
3 }  
4 else {  
5     Anweisungsblock2  
6 }
```



# Steuerung des Programmflusses

## Bedingungen und Verzweigungen

Ein Beispiel vom verbesserten Ticketautomaten aus Vorlesung 3:

```
01 public void geldEinwerfen(int betrag)
02 {
03     if (betrag > 0) {
04         bisherGezahlt = bisherGezahlt + betrag;
05     }
06     else {
07         System.out.println("Bitte nur positive Beträge verwenden:
08                             + betrag);
09     }
10 }
```



# Steuerung des Programmflusses

## Bedingungen und Verzweigungen

`if`-Anweisungen können auch geschachtelt werden:

```
01 public void geldEinwerfen(int betrag)
02 {
03     if (betrag > 0) {
04         bisherGezahlt = bisherGezahlt + betrag;
05         if(bisherGezahlt >= preis){
06             ticketDrucken();
07             wechselgeldAuszahlen();
08         }
09     }
10     else {
11         System.out.println("Bitte nur positive Beträge verwenden:
12                             + betrag);
13     }
14 }
```



# Steuerung des Programmflusses

## Bedingungen und Verzweigungen

### Blöcke von Anweisungen

Geschweifte Klammern fassen mehrere Anweisungen zusammen.

- Eine einzelne Anweisung kann ohne Veränderung der Ausführung in geschweifte Klammern gesetzt werden.
- Sollen nach einer `if`- oder `else`-Anweisung mehrere Anweisungen ausgeführt werden, müssen diese durch geschweifte Klammern zu einem Block zusammen gefasst werden.
- Die geschweiften Klammern sind außerdem wichtig, um eine `else`-Anweisung einer `if`-Anweisung zuordnen zu können.



# Steuerung des Programmflusses

## Bedingungen und Verzweigungen

`if`-Anweisungen können auch geschachtelt werden:

- Ausgabe nur im Fall  $-1 < x < 1$

```
01 if( x > 0 ){
02     if( x < 1 )
03         System.out.println("x zwischen 0 und 1");
04 }
05 else{
06     if( x < 0 ){
07         if( x > -1 )
08             System.out.println("x zwischen -1 und 0");
09     }
10     else
11         System.out.println("x ist 0");
12 }
```



# Steuerung des Programmflusses

## Bedingungen und Verzweigungen

Was passiert, wenn man die Klammern weg lässt?

- Ausgabe nur im Fall  $0 < x < 1!$

```
1  if( x > 0 )
2      if( x < 1 )
3          System.out.println("x zwischen 0 und 1");
4  else
5      if( x < 0 )
6          if( x > -1 )
7              System.out.println("x zwischen -1 und 0");
8  else
9      System.out.println("x ist 0");
```



# Steuerung des Programmflusses

## Bedingungen und Verzweigungen

### switch-Verzweigung

Die `switch`-Anweisung verzweigt auf Basis eines einzelnen Werts in eine beliebige Anzahl von Hällen.

- Kann beliebig viele `case`-Klauseln haben.
- Die `break`-Anweisung sorgt dafür, dass die Ausführung nicht alle folgenden `case`-Klauseln durchläuft.
- Die `default`-Klausel ist optional. Sie deckt alle nicht spezifizierten Fälle ab.

```
01 String getMonth(int month) {
02     String monat;
03     switch (month) {
04         case 1:
05             monat = "Januar";
06             break;
07         ...// Auslassung
37     case 12:
38         monat = "Dezember";
39         break;
40     default:
41         monat = „No month!";
42         break;
43     }
44     return monat;
45 }
```



# Steuerung des Programmflusses

## Einschub: Der Debugger

### Was ist ein Debugger?

Ein Programm, das parallel zur Ausführung des programmierten Codes läuft und folgende Funktionen bietet:

- Setzen von Breakpoints, bzw. schrittweises Abarbeiten.
- Anzeige der Reihenfolge der Methodenaufrufe.
- Anzeige der Inhalte aller Variablen, die zum jeweiligen Zeitpunkt existieren.

### Debugger

Ein Debugger ist ein Werkzeug zum Auffinden, Diagnostizieren und Beheben von Fehlern. Er ermöglicht eine Ablaufverfolgung des zu untersuchenden Programms in einzelnen Schritten oder zwischen definierten Haltepunkten. In jedem Schritt können die Inhalte der Variablen eingesehen und Methodenaufrufe oder Verzweigungen beobachtet werden.





# Steuerung des Programmflusses

## Einschub: Der Debugger

```
String word;
char c;
for (int i = 0; i < text.length; i++) {
    word = text[i].toUpperCase();
    for (int j = 0; j < word.length(); j++) {
        c = word.charAt(j);
        intArray[c-'A']++;
    }
}
return intArray;
}

/**
 * Gibt eine Liste aus. Pro Zeile ein Buchstabe und die Anzahl seiner Vorkommen
 * im String. A: letterCounters[0], ... Z: letterCounters[25].
 * Ausgabe nur, wenn letterCounters[i] einen positiven Eintrag enthält.
 * @param letterCounters Array von Integerwerten
 */
public static void printCounterArray (int[] letterCounters){
    char c;
    for (int i = 0; i < letterCounters.length; i++) {
        if (letterCounters[i] != 0) {
            c = (char) (i + 'A');
            System.out.println(c + ": " + letterCounters[i]);
        }
    }
}
```



# Steuerung des Programmflusses

## Einschub: Der Debugger


```


* @return intArray Array von Integeren. intArray[0]
*/
public static int[] count (String[] text){
    int [] intArray = new int[26];
    String word;
    char c;
    for (int i = 0; i < text.length; i++) {
        word = text[i].toUpperCase();
        for (int j = 0; j < word.length(); j++) {
            c = word.charAt(j);
            intArray[c-'A']++;
        }
    }
    return intArray;
}


/**
 * Gibt eine Liste aus. Pro Zeile ein Buchstabe und
 * im String. A: letterCounters[0], ... Z: letterCo
 * Ausgabe nur, wenn letterCounters[i] einen posit
 * @param letterCounters Array von Integerwerten
 */
public static void printCounterArray (int[] letterC
    char c;
    for (int i = 0; i < letterCounters.length; i++)


```


|  |  |
|--|--|
| Call Sequence<br>LetterCounter2.count<br>LetterCounter2.main | Static variables   |
|  | Instance variables   |
|  | Local variables<br>String[] text = <object reference><br>int[] intArray = <object reference><br>int i = 0<br>String word = "BUCHSTABEN"<br>int j = 0<br>char c = B |

  
 Halt

  
 Step

  
 Step Into

  
 Continue

  
 Terminate



# Steuerung des Programmflusses

## Rekursion

### Was bedeutet Rekursion?

Als Rekursion bezeichnet man den Aufruf einer Funktion durch sich selbst.

- Rekursion stammt vom lateinischen recurrere (deutsch: zurücklaufen) ab.

```
1 void meineRekursiveMethode( int n ){
2     if( n < 0 )
3         return;
4     System.out.print( n + ">" );
5     meineRekursiveMethode( n - 1 ); // rekursiver Aufruf
6     System.out.print( n + "<" );
7     return;
8 } // Was geschieht bei einem Aufruf meineRekursiveMethode(6)?
```



# Steuerung des Programmflusses

## Rekursion

### Was bedeutet Rekursion?

Im Prinzip könnte der Ausdruck in Zeile 5 durch die Zeilen 4 bis 6 ersetzt werden.

- Statt  $n$  müsste aber überall  $n-1$  stehen.
- Anschließend könnte der Ausdruck erneut ersetzt werden.
  - Statt  $n-1$  müsste aber überall  $n-2$  stehen.
  - Anschließend könnte der Ausdruck erneut ersetzt werden.
    - Statt  $n-2$  müsste aber überall  $n-3$  stehen. usw.

6>5>4>3>2>1>0>0<1<2<3<4<5<6<



# Steuerung des Programmflusses

## Rekursion

Plakatives Beispiel, Vorsicht, nicht 100% korrekt (vgl. Zeile 02).

```
01 void meineRekursiveMethode( int n ){
02     if( n < 0 )
03         return;
04     System.out.print( n + ">" );
05     System.out.print( n-1 + ">" );
06     System.out.print( n-2 + ">" );
07     meineRekursiveMethode( n - 3 );
08     System.out.print( n-2 + "<" );
09     System.out.print( n-1 + "<" );
10     System.out.print( n + "<" );
11     return;
12 }
```



# Steuerung des Programmflusses

## Rekursion

### Was bedeutet Rekursion?

Ein typisches Beispiel für eine Funktion, die rekursiv definiert wird, ist die Berechnung der Fakultät einer Zahl (z. B. 6!).

- $1! = 1$
- $n! = n (n-1)!$
- Durch wiederholtes Einsetzen entsteht so der Term
  - $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$
  - $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$



# Steuerung des Programmflusses

## Rekursion

Beispiel Fakultät rekursiv gelöst.

In Zeile 08 ruft sich die Funktion selber auf. Dies geschieht bis zum Erreichen der Abbruchbedingung in Zeile 04 (vgl. Def.:  $1! = 1$ ).

```
01 static int fakultaet(int i) {
02     int ergebnis = 0;
03     // Rücklauf bei Erreichen von 1!
04     if( i == 1){
05         ergebnis = 1;
06         return ergebnis;
07     }
08     ergebnis = i * fakultaet(i-1);
09     return ergebnis;
10 }
```



# Steuerung des Programmflusses

## Rekursion

The screenshot shows an IDE with a Java program for calculating a factorial using recursion. The code is as follows:

```
public class Fakultaet {  
  
    private static int fakultaet(int i) {  
        // TODO Auto-generated method stub  
        int ergebnis = 0;  
        if( i == 1){  
            ergebnis = 1;  
            return ergebnis;  
        }  
        ergebnis = i * fakultaet(i-1);  
        return ergebnis;  
    }  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // int uebergabe = Integer.valueOf(args[0])  
        int uebergabe = 6;  
        System.out.println(fakultaet(uebergabe));  
    }  
}
```

The IDE interface includes a menu bar with options: Compile, Undo, Cut, Copy, Paste, Find..., Find Next, and Close. On the left side of the code editor, there are four red circular icons labeled 'STEP', with an arrow pointing to the line `if( i == 1){`. The right side of the IDE shows a 'Debugger' window with the following sections:

- Options**
- Threads**: main (stopped)
- Call Sequence**:
  - Fakultaet.fakultaet (highlighted)
  - Fakultaet.fakultaet
  - Fakultaet.fakultaet
  - Fakultaet.fakultaet
  - Fakultaet.fakultaet
  - Fakultaet.fakultaet
  - Fakultaet.main
- Static variables**
- Instance variables**
- Local variables**:
  - int i = 1
  - int ergebnis = 0





# Steuerung des Programmflusses

## Rekursion

Beispiel Fakultät iterativ gelöst.

Viele Funktionen lassen sich sowohl rekursiv als auch iterativ lösen.

- Sobald aber Strukturen vorliegen, die nicht linear sind, ist die Verwendung von Rekursion häufig einfacher.
- Welche Auswirkungen hat dies auf Laufzeit und Speicherverbrauch?

```
1 static int fakultaet(int i) {  
2     int ergebnis = 1;  
3     for (int j = 1; j <= i; j++) {  
4         ergebnis *= j; // Iterative Lösung der Fakultät  
5     }  
6     return ergebnis;  
7 }
```



## 9.2 Methoden- und Objektinteraktion

### Wie interagieren Methoden und Objekte?

Bei einem Methodenaufruf wird der Programmfluss in diese Methode hineingelenkt.

- Durch ein `return` wird der Programmfluss wieder an die aufrufende Stelle zurück gelenkt.
- Bei einer `void`-Methode ist kein `return` notwendig und der Programmfluss wird trotzdem zurück gelenkt.
- Bei anderen Methoden darf das `return` aber auf keinen fall fehlen, da der Compiler sonst nicht weiß, welcher Wert als Ergebnis einer Funktion zurück geliefert werden soll.



# Methoden- und Objektinteraktion

```
01 String monat(int i){ // gibt Monatsname zu Monatszahl
02     String monat = "";
03     // ... Auslassung
04     return monat;
05 }
06 double wurzelAus(double i){ // berechnet die Wurzel
07     double ergebnis = 0.0;
08     // ... Auslassung
09     return ergebnis;
10 }
11 boolean bierKalt(){ // gibt an, ob das Bier kalt ist
12     boolean temperatur = false;
13     // ... Auslassung
14     return temperatur;
15 }
```



# Methoden- und Objektinteraktion

## Rückgabewerte von Methoden

Eine Methode vom Typ verschieden von `void` kann innerhalb anderer Ausdrücke aufgerufen werden, z. B. `while (nextToABeeper ()) pickBeeper ();`

- `nextToABeeper ()` besitzt den Rückgabebetyp `boolean` und gibt `true` zurück, wenn ein Beeper vorhanden ist, sonst `false`.
- Bei der Methodendeklaration steht der
  - Rückgabebetyp am Anfang des Methodenkopfs, bzw. –signatur.
  - Rückgabewert hinter dem Schlüsselwort `return` in der Methode.
  - Hinweis: `return` beendet die Abarbeitung der Methode. Eine Methode darf dabei mehrere `return` enthalten, wird aber beendet, sobald eines erreicht wird.



# Methoden- und Objektinteraktion

## Rückgabewerte von Methoden und Seiteneffekte

Ausgaben auf dem Bildschirm sind keine Rückgabewerte.

- Ausgaben auf dem Bildschirm werden als Seiteneffekte bezeichnet.
- Alle Veränderungen, die durch die Methode hervorgerufen werden und Zustände, Werte, etc. außerhalb der Methode betreffen, werden als Seiteneffekte bezeichnet.
- Ausnahme: Die Rückgabe eines Wertes durch `return` ist kein Seiteneffekt sondern die Hauptaufgabe einer Methode.



# Methoden- und Objektinteraktion

Der Code zu einer bestimmten Methode bildet immer einen separaten Codeblock mit fester Anfangsadresse.

Immer, wenn eine Methode aufgerufen wird, wird ein Sprung zur Anfangsadresse der Methode im Speicher durchgeführt.

- Zusätzlich zum Sprung werden noch folgende Anweisungen hinterlegt:
  - Mit welchen Parametern wird die Methode aufgerufen?
  - An welcher Stelle wird ein Rückgabewert gespeichert?
  - Wo wird das Programm weitergeführt, wenn die aufgerufene Methode beendet ist? Rücksprungadresse



# Methoden- und Objektinteraktion

## Der Run-Time-Stack

Eine Methode kann intern wieder andere Methoden aufrufen, die erneut andere Methoden aufrufen, etc.

- In der Regel befindet sich der Programmfluss innerhalb mehrerer Methoden zur gleichen Zeit.
- Aber nur die zuletzt aufgerufene Methode ist aktiv und wird abgearbeitet.
- Die vorher aufrufenden Methoden sind in Wartestellung, bis sie nach Ende der Methode wieder durch die Rücksprungadresse angesprungen werden.
  - Vgl. Stapel Getränkekisten, die oberste muss geleert werden, bevor die unten liegenden angerührt werden dürfen.



# Methoden- und Objektinteraktion

```
01 static int[] count (String[] text){
02     int [] intArray = new int[26];
03     // ... Auslassung
04 }
05     return intArray;
06 }
07 static void printCounterArray (int [] letterCounters){
08     //     ... Auslassung
09 }
10 public static void main(String[] args) {
11     printCounterArray(count(args));
12 }
```





# Methoden- und Objektinteraktion

## Hierarchie von Aufrufen

Im Beispiel auf der vorherigen Folie geschieht folgendes:

- Durch den Aufruf `java LetterCounter2` wird der Java-Interpreter gestartet.
- Dieser beginnt die Abarbeitung des Programms mit der `main`.
- Die `main` ruft ihrerseits die Methode `printCounterArray` auf. Während diese abgearbeitet wird, befindet sich die Abarbeitung der `main` in Wartestellung.
- Die `printCounterArray` erwartet allerdings ein Array von `int` und ruft deshalb noch die Methode `count` auf.



# Methoden- und Objektinteraktion

## Hierarchie von Aufrufen

Im Beispiel auf der vorherigen Folie geschieht folgendes:

- D. h. die meiste Zeit während der Programmlaufzeit befindet sich die Abarbeitung innerhalb von drei Methoden, `main`, `printCounterArray` und `count`. Es wird zu diesen Zeitpunkten aber nur die Methode `count` abgearbeitet.
- Erst wenn `count` beendet und ein Array von `int` zurück gegeben wurde, wird die Methode `printCounterArray` weiter ausgeführt, bis auch diese beendet wurde.
- Zum Schluss wird auch die `main` bis zum Ende ausgeführt und das Programm terminiert.



# Methoden- und Objektinteraktion

## Wie interagieren Methoden und Objekte?

Objekte können sich auch gegenseitig Erstellen, um auf Funktionen der anderen Klassen zugreifen zu können.

- Bisher wurden hauptsächlich Klassenmethoden (`static`) aufgerufen.
- Es können aber auch beliebige andere Funktionen aufgerufen werden.

```
1 int generiereZufallszahl(int maxGroesse) {  
2     java.util.Random rd = new java.util.Random();  
3     int r = Math.abs(rd.nextInt(maxGroesse));  
4     return r;  
5 }
```



# Methoden- und Objektinteraktion

## Wie interagieren Methoden und Objekte?

Werte, die durch Methoden berechnet wurden, müssen gespeichert werden, sonst sind sie nach Abarbeitung der Methode nicht mehr verfügbar.

- Im Beispiel auf der vorherigen Folie wurde eine Zufallszahl erzeugt.
- Die Methode `int nextInt(int maxGroesse)` liefert ein `int` zurück.
- Dieser Wert muss entweder direkt weiter verwendet oder gespeichert werden. Andernfalls existiert keine Referenz mehr und ein Zugriff ist ausgeschlossen.

– `int r = Math.abs(rd.nextInt(maxGroesse));`



# Methoden- und Objektinteraktion

## Hierarchie von Aufrufen

Analog zur Hierarchie bei Methodenaufrufen, ist nur die zuletzt aufgerufene Methode und das zugehörige Objekt aktiv.

- `int generiereZufallszahl(int maxGroesse)` ist eine Objektmethode der Klasse `Tester`.
- In der Methode wird ein Objekt `rd` vom Typ `Random` erzeugt.
- Die Methode befindet sich in Wartestellung, während die Methoden `Math.abs(int value)` und `rd.nextInt(int maxGroesse)` aufgerufen werden.
- Anschließend wird die ursprüngliche Methode beendet und das Objekt `rd` verschwindet.



# Methoden- und Objektinteraktion

## Wie interagieren Methoden und Objekte?

Es wird zwischen internen und externen Methodenaufrufen unterschieden.

- Methoden der einen Klasse können direkt aufgerufen werden.
- Methoden anderer Klassen müssen über die Punkt-Notation aufgerufen werden.
  - `Klasse.klassenMethode()`
  - `objekt.objektMethode()`

## Interner Methodenaufruf

Methoden können andere Methoden der eigenen Klasse als Teil ihrer Implementierung direkt aufrufen.

## Externer Methodenaufruf

Methoden können Methoden von anderen Klassen und Objekten über die Punkt-Notation indirekt aufrufen.