



# **PM2 - Java - Kontrollstrukturen**



# Fahrplan

- Ausdrücke-Anweisungen-Blöcke
- Kontrollanweisungen



# AUSDRÜCKE-ANWEISUNGEN- BLÖCKE



# Ausdrücke, Anweisungen, Blöcke

## expressions, statements, blocks

- Operatoren können verwendet werden um Ausdrücke (expressions) zu formen.
- Ausdrücke haben immer ein Wert
  - Zuweisungen haben den Wert, der der Variable zugewiesen wird. ABER: Kombinationen aus Deklaration und Zuweisung haben **keinen** Wert.

```
i = 78          // Wert der Zuweisung ist 78
int j  = 78;   // Deklaration und Zuweisung: kein Wert
```

- arithmetische und logische Ausdrücke haben den Wert der Auswertung des Ausdrucks.
- **instanceof** hat einen booleschen Wert
- **cast** hat als Ergebnis ein Objekt (oder Wert) vom Typ in den gecastet wird.



# Ausdrücke, Anweisungen, Blöcke

## expressions, statements, blocks

- Aus einigen Ausdrücken können einfache Anweisungen (statements) konstruiert werden.

- Zuweisungen:

```
int j = 78;           // Deklaration und Zuweisung kein Wert
j = 100;              // Wert = 100
```

- Inkrement und Dekrement Operator (haben einen Wert)

```
int j = 78;
j++;           // Wert = 79
--j;          // Wert = 77
```

- Objekt Erzeugung (hat einen Wert)

```
new String("einString"); // Wert = "einString"
```

- Methodenaufrufe (haben einen Wert, den die Methode zurückgibt)

```
String s;
s = new String("einString");
s.length();
```

- Einfache Anweisungen enden immer mit einem Semikolon.



# Ausdrücke, Anweisungen, Blöcke

## expressions, statements, blocks

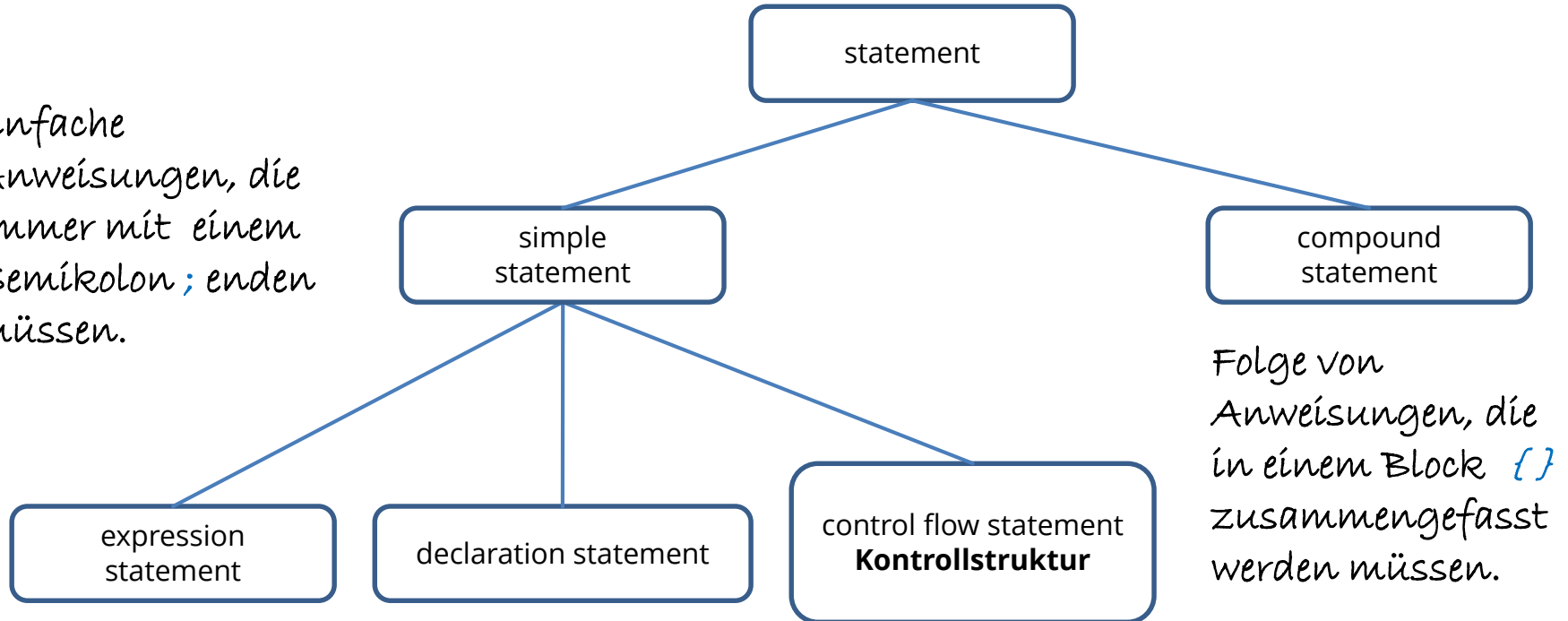
- Mehrere Anweisungen können hintereinandergeschrieben werden. Sie müssen dann von einem Block umgeben sein.
- Ein Block ist ein Bereich mit null oder mehr Anweisungen zwischen geschweiften Klammern `{}`.

```
public static void main(String[] args) { // begin block 1
    boolean condition = true;
    if (condition) { // begin block 2
        System.out.println("Condition is true.");
    } // end block 2
    else { // begin block 3
        System.out.println("Condition is false.");
    } // end block 3
} // end block 1
```



# Anweisungen (statements) in Java

einfache  
Anweisungen, die  
immer mit einem  
Semikolon ; enden  
müssen.



Folge von  
Anweisungen, die  
in einem Block {}  
zusammengefasst  
werden müssen.

Ausdrücke =  
Anweisungen, die  
ein Ergebnis  
haben.

Anweisungen, die  
Variablen deklarieren  
oder Variablen bei der  
Deklaration einen  
Wert zuweisen.

Anweisungen, die  
den sequentiellen  
Programmablauf  
modifizieren



# Beispiele für Ausdrücke

```
public class ExpressionExamples {
    public static void main(String[] args) {
        // Variablen-Deklarationen;
        int i; short sh; boolean b; String str; Object o;
        // arithmetischer Ausdruck
        i = 7 + 5;
        // logischer Ausdruck
        b = true;
        b = b && true;
        b = b || false;
        // Zuweisungen
        str = "";
        o = str;
        // Typprüfung
        b = str instanceof String;
        // Type-Cast
        sh = (short)i;
        str = (String)o;
        // Methoden mit Ergebnis
        str = String.format("ganze Zahl:%d", 15);
    }
}
```





# Beispiele für Deklarationen

```
public class DeclarationExamples {

    // Deklaration Instanzvariablen
    private Integer numOfExamples;
    private String[] exampleText = { "Instanzvariable numOfExamples", "Parameter von swap val1",
        "Parameter von swap val2", "lokale Variable in swap t1",
        "lokale Variable in swap t2", "lokale Variable in swap tmp",
        "lokale Variable in for i"};
    public DeclarationExamples() {
        numOfExamples = 7;
    }

    public void swap(int val1, int val2) { // Deklaration Parameter val1, val2
        int t1; // Deklaration lokale Variable t1
        t1 = val1; // Zuweisung Wert zu t1
        int t2 = val2; // Deklaration und Zuweisung Wert zur lokalen Variablen t2
        printf("vor dem Tausch t1:%d t2:%d", t1, t2);
        int tmp = t1; // Deklaration lokale Variable für den Tausch
        t1 = t2;
        t2 = tmp;
        printf("nach dem Tausch t1:%d t2:%d", t1, t2);
    }

    public void howManyDeklarations() {
        for (int i = 0; i < numOfExamples; i++) {
            printf("Decl %d:%s\n", i, exampleText[i]);
        }
    }
}
```



# Fragen & Aufgaben

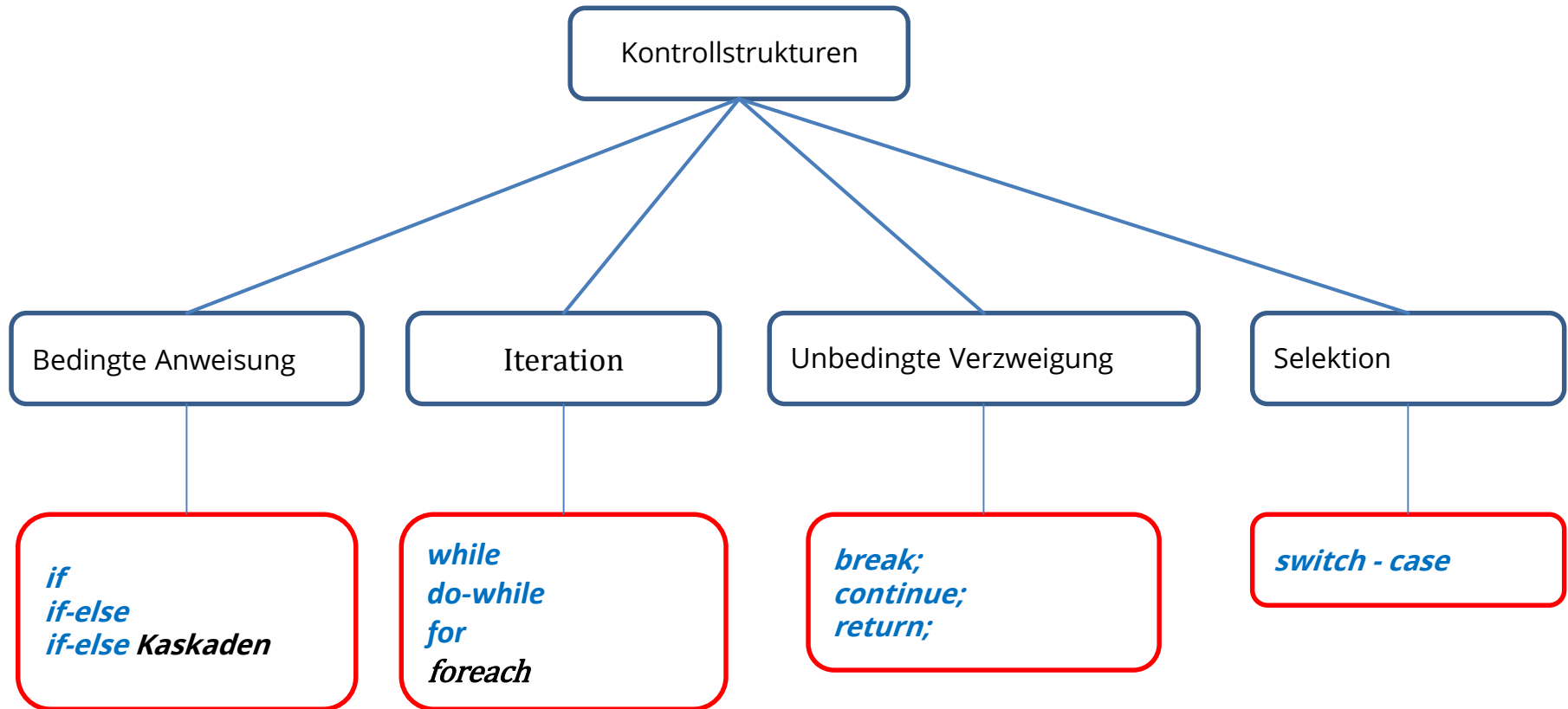
1. Wie wird die Klasse *DeclarationExamples* zu einem ausführbaren Programm?
2. Verwandeln Sie die Klasse in ein ausführbares Programm und rufen Sie die Methode *howManyDeclarations* auf.
3. Korrigieren Sie die Klassendefinition derart, dass die Methode *howManyDeclarations* ein korrektes Ergebnis liefert? Löschen von Quelltext ist nicht erlaubt!



# KONTROLLSTRUKTUREN



# Control Flow Statements (Kontrollstrukturen)



**Merke:** bedingte Anweisungen und die **while** Konstrukte verlangen **boolesche** Werte in den Bedingungssteilen.



# Typographische Konventionen

- In den nachfolgenden Folien sind die
  - in schwarz geschriebenen Symbole und Zeichen syntaktische Bestandteile der Kontrollstruktur.
  - in rot geschriebenen und in spitzen Klammern gesetzten Begriffe Platzhalter für Ausdrücke und Anweisungen.

Beispiel:

**if ( *<condition>* ) *<statement>***

- **if, (, )**: syntaktische Elemente der Kontrollstruktur
- ***<condition>*** Platzhalter für einen beliebigen booleschen Ausdruck
- ***<statement>*** Platzhalter für eine beliebige Anweisung



Bedingte Anweisungen

# KONTROLLSTRUKTUREN



# Einfache if-Anweisungen

**if ( *<condition>* ) *<statement>***

- wenn *<condition>* gilt, dann wird *<statement>* ausgeführt. *<condition>* muss ein boolescher Ausdruck sein, der einen Wahrheitswert liefert.
- **Merke:** muss immer in runden Klammern stehen.
- sonst wird *<statement>* nicht ausgeführt. *<statement>* ist entweder ein **simple statement** oder ein **compound statement**.

```
double temperature = -9.0;
```

```
if (temperature < 0)  
    System.out.println("Eiszeit");
```

```
if (temperature < 0) {  
    System.out.println("Eiszeit");  
    temperature = -temperature;  
    System.out.println(temperature);  
}
```



```
Eiszeit  
Eiszeit  
9.0
```

# Zweiseitige if-Anweisung (if-else)



if ( *<condition>* )

*<stmt1>*

else

*<stmt2>*

- wenn *<condition>* gilt, dann wird *<stmt1>* ausgeführt
- sonst wird *<stmt2>* ausgeführt.

```
double x = -17;
```

```
double a;
```

```
if (x >= 0) {
```

```
    a = x;
```

```
    System.out.println("x >= 0" + x);
```

```
    System.out.println("a = x");
```

```
}
```

```
else {
```

```
    a = -x;
```

```
    System.out.println("x < 0: " + x);
```

```
    System.out.println("a = -x");
```

```
}
```

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







# Geschachtelte **if**-Anweisungen am Beispiel der quadratischen Gleichung

- eine **if-Anweisung** kontrolliert eine untergeordnete **if-Anweisung**

- Beispiel:** Die Anzahl der Lösungen der Gleichung:

$$ax^2 + bx + c$$

ist abhängig vom Vorzeichen der Diskriminante d:

$$d = b^2 - 4ac$$

- Lösungsformel:**

$$\frac{(-b \pm \sqrt{b^2 - 4ac})}{2 * a}$$

- Wenn  $d < 0$ , dann existiert keine Lösung
- Wenn  $d == 0$ , dann nur eine Lösung:  
$$\frac{-b}{2 * a}$$

- Wenn  $d > 0$ , dann zwei Lösungen:  
$$\frac{(-b \pm \sqrt{d})}{2 * a}$$



# Geschachtelte **if**-Anweisungen am Beispiel der quadratischen Gleichung

```
public class QuadratischeGleichung {  
  
    public static void main(String[] args) {  
        double a = 1, b = 1, c = 1;  
        // b=2 1 Lsg, , b=3 2 Lsgen  
        double d = b * b - 4 * a * c;  
        if (d < 0) {  
            print("keine Lösung");  
        } else if (d == 0)  
            printf("1 Lösung : " + "%g%n ", -b / (2 * a));  
        else  
            printf("2 Lösungen: %g,"  
                + " %g%n ",  
                (-b + Math.sqrt(d)) / (2 * a),  
                (-b - Math.sqrt(d)) / (2 * a));  
    }  
}
```



# If-Else-Kaskaden

- sind beliebig tief geschachtelte **if-else** Anweisungen.
- werden verwendet, um eine Liste von Möglichkeiten zu vergleichen.
- haben ein **spezielles Layout**: keine Einrückung der **else if** Zweige.
- neben stehendes Programmfragment berechnet die Tage eines Monats.

```
int month = 5;  
int days = -1;
```

```
if (month == 1)  
    days = 31;  
else if (month == 2)  
    days = 28;  
else if (month == 3)  
    days = 31;  
else if (month == 4)  
    days = 30;
```

```
printf("Month %d Days %d\n", month,  
      days);
```



# Der ternärer Ausdruck – die Ausnahme

```
public class TernaererAusdruck {  
  
    /**  
     * Methode berechnet den Betrag einer Zahl  
     * mithilfe eines ternären Ausdrucks  
     * @param x, eine beliebige Zahl  
     * @return Betrag von x  
     */  
    public static double abs(double x) {  
        return x >= 0 ? x : -x;  
    }  
}
```

**<condition> ? <expr1> : <expr2>;**

- Der ternäre Ausdruck ist eine **Kontrollstruktur mit** einem **Wert**.
- Wenn die Bedingung **<condition>** erfüllt ist, ist das Ergebnis der Ausdruck nach dem Fragezeichen **? <expr1>**, sonst der Ausdruck nach dem Doppelpunkt **: <expr2>**.



Iteration

# KONTROLLSTRUKTUREN



# Bedingungsgesteuerte Iteration – while

```
while ( <condition> )  
    <stmt>
```

*<stmt>* wird solange ausgeführt bis *<condition>* nicht mehr erfüllt ist

- Beispiel: Euklids Algorithmus für den größten gemeinsamen Teiler (GGT) von **m** und **n**:
- Teile die Größere der beiden Zahlen durch die Kleinere
  - Ist der Divisionsrest = 0, dann ist die kleinere Zahl der GGT
  - Ist der Divisionsrest > 0, dann wiederhole das Verfahren mit der Kleineren der beiden Zahlen und dem Divisionsrest.



# Algorithmus für den GGT nach Euklid

```
package controlflow;
import static util.Print.*;

public class EuclidGGT {
    public static void main(String[] args) {
        int m = 27;
        int n = 18;
        int mcp = m;
        int ncp = n;
        int r = m%n;

        while (r > 0) {
            m = n;    // kleinere Zahl
            n = r;    // Rest
            r = m%n;
        }
        printf("GGT von %d und %d ist " +
            "%d\n", mcp, ncp, n);
    }
}
```



GGT von 27 und 18 ist 9



# Bedingungsgesteuerte Iteration nachprüfend **do-while**

**do** *<stmt>* **while** ( *<condition>* )

- *<stmt>* wird ausgeführt und anschließend *<condition>* geprüft, solange bis *<condition>* nicht mehr erfüllt ist.
- Im Gegensatz zu **while**, wird hier *<stmt>* mindestens einmal ausgeführt.
- **Übung:**
  - Implementieren Sie den GGT nach Euklid mit **do-while**





## Iteration mit Fortschaltanweisung: **for**

```
for ( <initexpr> ; <condition> ; <stepexpr> )  
    <stmt>
```

- zu Beginn wird *<initexpr>* einmal ausgeführt
- dann wird die Bedingung ( *<condition>* ) geprüft
- ist die Bedingung erfüllt
  - wird zuerst *<stmt>* ausgeführt
  - und dann *<stepexpr>*, die „Fortschaltanweisung“, ausgewertet
- ist die Bedingung nicht erfüllt
  - wird die **for** Schleife beendet und *<stmt>* nicht mehr ausgeführt
- Jedes der Ausdrücke *<initexpr>*, *<condition>* und *<stepexpr>* darf leer sein



# Iteration mit Fortschaltanweisung: Beispiel

```
package controlflow;

import static util.Printer.*;

public class ListeKleinBuchstaben {

    public static void main(String[] args) {

        for (char c = 0; c < 128; c++) { // c lokale Variable der
            // for Schleife
            if (Character.isLowerCase(c))
                print("Ascii " + (int) c + " Zeichen " + c);
        }
    }
}
```



# Iteration mit Fortschaltanweisung: Alternative 1 /2

```
package controlflow;
import static util.Printer.*;

public class ListeKleinBuchstaben {
    public static void alternative1() {

        char c = 0;
        for (; c < 128; c = (char) (c + 1)) { // Initial. leer,
            if (Character.isLowerCase(c))
                print("Ascii " + (int) c + " Zeichen " + c);
        }

        print("Ascii " + (int) c + " Zeichen " + c);
        // c definiert und modifiziert
    }

    public static void alternative2() {
        char c = 0; // entspricht a
        for (;;) { // alle Ausdrücke sind leer.
            if (c >= 128)
                break;
            if (Character.isLowerCase(c))
                print("Ascii " + (int) c + " Zeichen " + c);
            c = (char) (c + 1);
        }
    }
}
```



Pseudozufallszahlen

# EXKURS

# Einschub-Pseudozufallszahlen und Zufallszahlengenerator



## Random

- eine Pseudozufallszahl ist eine nach einem deterministischen Algorithmus berechnete Zahl in einem definierten Wertebereich.
- Der Startwert des Zufallszahlengenerators (**seed**) bestimmt die Zahlenfolgen eindeutig.
- Zufallszahlengeneratoren in Java sind Objekte der Klasse **Random**.
- Der **seed** kann bei der Erzeugung eines Random Objektes im Konstruktor übergeben werden. (**new Random()**, **new Random(898778)**)
- **Random** kann Zahlenfolgen für **boolean, int, long, float, double** berechnen.  
  
**Random rand = new Random();**  
**rand.nextBoolean(), nextInt(), etc...**
- Für **int** Werte kann der Wertebereich der Zufallszahlen auf ein Intervall positiver Zahlen inklusive 0 eingeschränkt werden.  
  
**rand.nextInt(10)** liefert Zahlen im Intervall **[0,10)**
- **rand.nextFloat()** / **rand.nextDouble()** liefern Werte im Intervall **[0,1)**
- **Math.rand()** ist eine Alternative zu **rand.nextDouble()**



# Beispiele für die Verwendung von **Random**

```
package controlflow;

import java.util.Random;
import static util.Printer.*;

public class RandomDemo {
    public static void main(String[] args) {
        Random rand1 = new Random();
        Random rand2 = new Random();
        Random rand3 = new Random(256);
        Random rand4 = new Random(256);
        Random rand5 = new Random(257);

        // nur rand3, rand4 erzeugen gleiche Zahlenfolgen
        for (int i = 0; i < 5; i++) {
            print("Iteration " + (i + 1));
            print("rand1.nextInt() " + rand1.nextInt());
            print("rand2.nextInt() " + rand2.nextInt());
            print("rand3.nextInt() " + rand3.nextInt());
            print("rand4.nextInt() " + rand4.nextInt());
            print("rand5.nextInt() " + rand5.nextInt());
        }
    }
}
```



# Ausgabe von *RandomDemo*

## Iteration 1

```
rand1.nextInt() 1375772241  
rand2.nextInt() -581106638  
rand3.nextInt() -1056988858  
rand4.nextInt() -1056988858  
rand5.nextInt() -1057373607
```

## Iteration 2

```
rand1.nextInt() 554329545  
rand2.nextInt() -828972692  
rand3.nextInt() -175263421  
rand4.nextInt() -175263421  
rand5.nextInt() 980221155
```

## Iteration 3

```
rand1.nextInt() 1443250727  
rand2.nextInt() -1002679323  
rand3.nextInt() -699345345  
rand4.nextInt() -699345345  
rand5.nextInt() 28842292
```

## Iteration 4

```
rand1.nextInt() 92973968  
rand2.nextInt() -2008949880  
rand3.nextInt() -2035245795  
rand4.nextInt() -2035245795  
rand5.nextInt() 1405429233
```

## Iteration 5

```
rand1.nextInt() 668239049  
rand2.nextInt() -8915339  
rand3.nextInt() -1540701258  
rand4.nextInt() -1540701258  
rand5.nextInt() 908707517
```



# Beispiele für die Verwendung von *Random*

```
package controlflow;
import java.util.Random;
import static util.Printer.*;
public class RandomDemo2 {
    public static void main(String[] args) {
        Random rand1 = new Random();
        Random rand2 = new Random();
        Random rand3 = new Random(256);
        Random rand4 = new Random(256);
        Random rand5 = new Random(257);
        // auch für Boolean, Long, Float, Double
        print("rand1.nextBoolean() " + rand1.nextBoolean());
        print("rand3.nextLong() " + rand3.nextLong());
        print("rand4.nextFloat() " + rand4.nextFloat());
        print("rand5.nextDouble() " + rand5.nextDouble());

        // Intervall der Zufallszahlen: [0-10)
        for (int i = 0; i < 30; i++) {
            print("Iteration " + (i + 1));
            print("rand1.nextInt(10) " + rand1.nextInt(10));
            print("rand2.nextInt(10) " + rand2.nextInt(10));
        }
    }
}
```





# Ausgabe von *RandomDemo2*

```
rand1.nextBoolean() false
rand1.nextLong() -
    4539732577521651389
rand1.nextFloat() 0.7539006
rand1.nextDouble()
    0.7538110200796354
```

```
Iteration 1
rand1.nextInt(10) 0
rand2.nextInt(10) 0
...
Iteration 9
rand1.nextInt(10) 7
rand2.nextInt(10) 6
...
Iteration 19
rand1.nextInt(10) 1
rand2.nextInt(10) 4
...
Iteration 30
rand1.nextInt(10) 8
rand2.nextInt(10) 7
```



Iteration

# KONTROLLSTRUKTUREN



# Iteration mit foreach-Syntax: **for**

```
for ( <type> <var> : <elemseq> )  
    <stmt>
```

- in dieser Variante nimmt **<var>** nacheinander Werte in der Sequenz von Elementen **<elemseq>** an.
- Sequenzen von Elementen sind z.B. Arrays oder Objekte der Collection-Klassen. (dazu später mehr).
- Im Bsp. rechts ist **floats** das Array, über das im zweiten **for** mit **foreach** iteriert wird. **f** nimmt dabei nacheinander die Werte in **floats** an.

```
package controlflow;  
  
import java.util.Random;  
import static util.Printer.*;  
  
public class ForEachDemo {  
    public static void main(String[]  
        args) {  
        Random rand = new Random(50);  
        float[] floats = new float[10];  
        // klassisches for  
        for (int i = 0; i < 10; i++)  
            floats[i] = rand.nextFloat();  
        // foreach Syntax  
        for (float f : floats)  
            print("f " + f);  
    }  
}
```

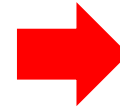


# Iteration mit foreach-Syntax: **for**

```
package controlflow;

import java.util.Random;
import static util.Printer.*;

public class ForEachDemo {
    public static void main(String[]
        args) {
        Random rand = new Random(50);
        float[] floats = new float[10];
        // klassisches for
        for (int i = 0; i < 10; i++)
            floats[i] = rand.nextFloat();
        // foreach Syntax
        for (float f : floats)
            print("f " + f);
    }
}
```



```
f 0.7297136
f 0.597892
f 0.6141579
f 0.82166934
f 0.6215813
f 0.48963797
f 0.50235754
f 0.7030554
f 0.780494
f 0.7904046
```



Unbedingte Verzweigung

# KONTROLLSTRUKTUREN




# Unbedingte Verzweigung mit **return**

- **return** erfüllt zwei Funktionen:
  - legt das Ergebnis einer Methode fest.
  - beendet die Ausführung der Methode.

```
package controlflow;
import static util.Printer.*;

public class IfElse2 {
    static int test(int source, int target) {
        if (source > target)
            return 1;
        else if (source == target)
            return 0;
        else
            return -1;
    }
    public static void main(String[] args) {
        print(test(10, 5));
        print(test(2, 5));
        print(test(5, 5));
        return;
    }
}
```

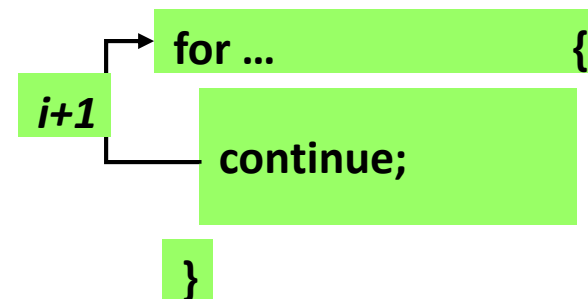
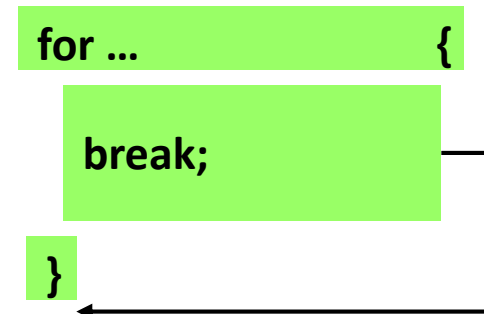


1
-1
0



# Unbedingte Verzweigung: **break** und **continue**

- In Iterationen kann die sequentielle Ausführung des Statement-Blocks durch **break** und **continue** modifiziert werden.
- **break** bricht die Ausführung des Iterationsblocks ab. Das Programm wird hinter dem Block der Iteration fortgesetzt.
- **continue** bricht den aktuellen Schleifendurchlauf ab und beginnt mit dem nächsten Iterationsschritt.





# Unbedingte Verzweigung: **break** und **continue**

```
package controlflow;

public class BreakUndContinue {

    public static void main(String[]
        args) {
        for (int i = 0; i < 100; i++) {
            if (i == 67)
                break;
            if (i % 9 != 0)
                continue;
            System.out.print(i + " ");
        }
    }
}
```

- Welche Ausgaben erzeugt das Programm?
- Wann wird die Methode **main** und damit das Programm beendet?





# Verarbeitung von Benutzereingabe

- **Aufgabe:**
  - Schreiben Sie ein Programm, das zählt, wie viele unterschiedliche geraden Zahlen in 30 Sekunden eingegeben werden.
  - Wird keine Zahl eingegeben, dann wird nach einer sprechenden Ausgabe die Leseschleife von vorne begonnen.
  - Wird keine gerade Zahl eingegeben, dann wird nach einer sprechenden Ausgabe die Leseschleife von vorne begonnen.
  - Wird eine bereits gespeicherte Zahl eingegeben, dann wird nach einer sprechenden Ausgabe die Leseschleife von vorne begonnen.
  - Nur wenn es sich um eine neue gerade Zahl handelt, dann gibt das Programm aus, welche Zahl eingegeben wurde und wieviel gerade Zahlen in wieviel Millisekunden eingegeben wurden.
  - Das Programm endet nach 30 Sekunden.
- **Lösung:** Klasse *EingabeVerarbeitungMitBreakUndContinue*



# Verarbeitung von Benutzereingaben

```
Scanner scanner = new Scanner(System.in);
long start = System.currentTimeMillis();
long end = start;
Set<Integer> geradeZahlen = new HashSet<>();
System.out.println("Geben Sie möglichst viele unterschiedliche gerade Zahlen ein!");
while (true) {
    end = System.currentTimeMillis();
    // Bedingung, die die Schleife beendet
    if ((end - start) > 30000)
        break;
    if (scanner.hasNextInt()) {
        int zahl = scanner.nextInt();
        if (zahl % 2 != 0) {
            System.out.printf("Die Zahl %d ist nicht durch 2 teilbar! Nächster Versuch!\n", zahl);
            // mit dem nächsten Schleifendurchlauf starten
            continue;
        }
        if (!geradeZahlen.add(zahl)) {
            System.out.printf("%d leider schon vorhanden! Nächster Versuch!\n", zahl);
            continue;
        }
        System.out.printf("Toll! %d ist eine neue gerade Zahl.\n", zahl);
        System.out.printf("Sie haben %d gerade Zahlen in %.3f Sekunden eingegeben\n",
                           geradeZahlen.size(), (end-start)/1000.0);
    } else {
        System.out.println("Die Eingabe ist keine ganze Zahl! Nächster Versuch!");
        // Eingabe lesen und ignorieren
        scanner.next();
    }
}
```



Selektion

# KONTROLLSTRUKTUREN



# Selektion mit *switch* und *case*

```
switch (<selector> ) {  
    case <value1> : <stmt1> ; break ;  
  
    case <value2> : <stmt2> ; break ;  
  
    case <value3> : <stmt3> ; break ;  
  
    case <value4> : <stmt4> ; break ;  
  
    // ...  
    default: <stmt> ;  
}
```

- **selector** ist ein Ausdruck, der nur einen integralen Wert (**char**, Numerae, **Enum**) oder einen **String** enthalten kann.
- Wenn das Ergebnis von **selector** mit einem Wert **value<sub>i</sub>**... übereinstimmt wird die nachfolgende Anweisung **stmt<sub>i</sub>**... ausgeführt.
- Wenn keines der Werte mit dem **selector** übereinstimmt, wird **stmt** nach dem **default** ausgeführt.
- **break** nach jedem **stmt** bricht die Auswertung der nachfolgenden **case** Klauseln ab. Ohne **break** würden alle **case** Klauseln durchlaufen.



# Selektion mit *switch* und *case*

```
public class VokaleUndKonsonanten {  
    public static void main(String[] args) {  
        Random rand = new Random(47);  
        for (int i = 0; i < 100; i++) {  
            int c = rand.nextInt(26) + 'a';  
            printnb((char) c + ", " + c + ": ");  
            switch (c) {  
                case 'a':  
                case 'e':  
                case 'i':  
                case 'o':  
                case 'u': print("Vowel"); break;  
                case 'y':  
                case 'w': print("Sometimes a vowel"); break;  
                default: print("Consonant");  
            }  
        }  
    }  
}
```



# Übungen

1. Schreiben Sie ein Programm, dass Werte von 1 bis 100 in 5'er Schritten ausgibt.
2. Schreiben Sie ein Programm, das den Modulo Operator **%** verwendet, um Primzahlen zu testen. Ist eine Zahl eine Primzahl, dann soll diese ausgegeben werden.
3. Schreiben Sie ein Programm, dass 25 Zufallszahlen (**int**-Werte) generiert und in einer **if-else** Anweisung überprüft, ob die erste Zufallszahl größer, kleiner oder gleich der generierten 24 Zufallszahlen ist und dieses auf der Konsole ausgibt.
4. Wie würde die **test**-Methode der **IfElse** Klasse aussehen, wenn nur die Verwendung **eines return** erlaubt ist?



# Übungen

5. Schreiben Sie für die Klasse **IfElse2** eine Methode **test**, die für den **source** Wert überprüft, ob er in einem Intervall mit den Grenzen **begin**, **end** liegt. **test** wird mit 3 Parametern aufgerufen.
6. Schreiben Sie Übung 1 unter Verwendung von **break** beim Wert 99 um. Was ist der Unterschied, wenn Sie **break** durch **return** ersetzen?
7. Entfernen Sie in der Klasse **VowelsAndConsonants** das **break**. Was passiert?



# Zusammenfassung

- In Java unterscheiden wir **Ausdrücke** – Anweisungen, die einen Wert haben – von **Anweisungen, die keinen Wert** haben.
- Zu den Anweisungen ohne Wert gehören:
  - Deklarationen
  - Kontroll-Fluss-Anweisungen (Kontrollstrukturen) / Sequenzen von Anweisungen (Blöcke)
  - **Ausnahme**: ternärer Ausdruck
- **Kontrollstrukturen** unterteilen sich in
  - Sequenzen
  - Bedingte Verzweigung (if / if-else / if-else Kaskaden / ternärer Ausdruck)
  - Iteration (while / for / foreach)
  - Unbedingte Verzweigung (return / break / continue)
  - Selektion (switch-case)





# Zusammenfassung

- **Iteration** unterscheidet sich in
  - Bedingungs-gesteuerte Schleifen
  - Fortschaltschleifen häufig als Zählschleifen realisiert
  - foreach-Konstrukte: Iterieren über die Inhalte eines Arrays oder einer Objektsammlung
- **Unbedingte Verzweigungen** unterscheiden sich in
  - Abbruch einer Schleife und Weiterarbeiten nach dem Ende der Schleife (**break**)
  - Abbruch des aktuellen Schleifendurchlaufs und Starten des nächsten Schleifendurchlaufs (**continue**)
  - Beenden eines Methodenaufrufs (**return**)
  - **break** und **continue** sind u.a. nützliche Hilfsmittel für die Steuerung von Benutzerdialogen.
- Selektion, die **switch-case** Anweisung,
  - erlaubt im Target **nur integrale Datentypen** (Zahlen, Zeichen, Enums (später)) oder **Strings**.
  - arbeitet alle **case** Klauseln bis zum ersten **break** ab!