

JAVA Programmierung

ECLIPSE & JAVA 8



Themenübersicht

01

Kontrollstrukturen

1. Verzweigung => if
2. Verzweigung => switch case
3. Schleife => for & for(each)
4. Schleife => while & do while
5. Try, Catch, Finally

02

Aufgaben

1. Aufgabe
2. Lösung

03

Methoden

1. Generelles
2. Signatur
3. Modifizierer

04

Aufgaben

1. Aufgabe
2. Lösung



**Java**

01 Kontrollstrukturen

1. Verzweigung => *if*
2. Verzweigung => *switch case*
3. Schleifen => *for & for(each)*
4. Schleifen => *while & do while*
5. *Try Catch Finally*



1.1

Kontrollstrukturen

Verzweigung



1.1 Kontrollstrukturen

- **Verzweigung => if & else**

If-Anweisungen dienen zur Kontrolle von Bedingenungen, welche nach dem Schlüsselwort „if“ in Klammern aufgeführt wird. Sie werten => Evaluieren den Wahrheitsgehalt dieser Bedingung aus. **Eine boolsche Operation**. Sollte dieser wahr sein wird der if Anweisungsblock betreten. Dieser befindet sich zwischen den Geschweiften Klammern. Sollte dies nicht der fall sein, wird der Else Zweig Aufgerufen. Deshalb auch => Verzweigung!

```
// Hat dein Auto 4 Räder ? Ja / Nein
if (Bedingung) {
    // if Anweisungs Block
    // wird ausgeführt wenn die Bedingung wahr ist
    // z.B. Mein Auto hat 4 Räder => Ja also True
} else {
    // else Anweisungs Block
    // wird ausgeführt wenn die Bedingung falsch ist
    // z.B. Ich habe kein Auto => Nein also False
}
```

1.1 Kontrollstrukturen

- Dies geht auch kürzer => der Ternärer Operator

if -> else	if(Bedingung)		{ code }	else	{ code }
Ternär	Bedingung == true	?	true	:	false

```
// Hat dein Auto 4 Räder ? Ja / Nein
if (Bedingung) {
    // if Anweisungs Block
    // wird ausgeführt wenn die Bedingung wahr ist
    // z.B. Mein Auto hat 4 Räder => Ja also True
}else {
    // else Anweisungs Block
    // wird ausgeführt wenn die Bedingung falsch ist
    // z.B. Ich habe kein Auto => Nein also False
}
```



```
// Pseudo Code für if / else
// wenn wahr ? dann : sonst
// Übertragen auf if else
// Bedingung ? wahr=if : falsche=else
boolean test = Bedingung == true ? true : false;
```

1.1 Kontrollstrukturen

- Verzweigung => if -> else if -> else

Nun erweitern wir unsere Verzweigung um eine weitere Bedingung.

```
// Hat dein Auto 4 Räder ? Ja / Nein
if (Bedingung) {
    // if Anweisungs Block
    // wird ausgeführt wenn die Bedingung wahr ist
    // z.B. Mein Auto hat 4 Räder => Ja also True
}
// Hat dein Auto 3 Räder ? Ja / Nein
else if (Bedingung2) {
    // else if Anweisungs Block
    // wird ausgeführt wenn die Bedingung wahr ist
    // z.B. Mein Auto hat 3 Räder => Ja also True
} else {
    // else Anweisungs Block
    // wird ausgeführt wenn die Bedingung falsch ist
    // z.B. Ich habe kein Auto => Nein also False
}
```

1.1 Kontrollstrukturen

- **Verzweigung => Verschachtelt**

Natürlich ist es möglich, Verzweigungen ineinander zu verschachteln.

Dies bedeutet das innerhalb der einzelnen Verzweigungen beliebig viele weitere Bedingungen geprüft werden können.

```
// Hat dein Auto 4 Räder ? Ja / Nein
if (Bedingung) {
    // if Anweisungs Block
    // wird ausgeführt wenn die Bedingung wahr ist
    // z.B. Mein Auto hat 4 Räder => Ja also True
}
// Hat dein Auto 3 Räder ? Ja / Nein
else if(Bedingung2){
    // Ist dein Auto eine Trike ? Ja / Nein
    if(Bedingung3) {
        // if Anweisungs Block
        // wird ausgeführt wenn die Bedingung wahr ist
        // z.B. Mein Auto ist ein Trike => Ja also True
    }
    // else if Anweisungs Block
    // wird ausgeführt wenn die Bedingung wahr ist
    // z.B. Mein Auto hat 3 Räder => Ja also True
} else {
    // else Anweisungs Block
    // wird ausgeführt wenn die Bedingung falsch ist
    // z.B. Ich habe kein Auto => Nein also False
}
```


1.2 Kontrollstrukturen

Verzweigung => Switch Case



1.2 Kontrollstrukturen

- **Verzweigung => Switch Case**

Das Switch Case funktioniert nach dem selben Prinzip wie die Kontrollstruktur „if -> else“. Hier wird geprüft welcher Fall (Wert) zutrifft und so werden die entsprechenden Anweisungen ausgeführt.

Wenn es nicht explizit erforderlich/sinnvoll ist, sollte JEDER Anweisungsblock mit einem **break** geschlossen werden, da es sonst zu unerwünschtem Verhalten kommen kann!

```
String bedingungHallo = "Hallo";  
// erster Teil der Bedingung  
// hallo  
switch(bedingungHallo)  
{  
    // zweiter Teil der Bedingung  
    //Übertragen bedingungHallo == hallo ? Ja : Nein  
    case "hallo":  
        // case Anweisungs Block  
        // wird ausgeführt wenn die Bedingung wahr ist  
        System.out.println("hallo");  
        break;  
    // zweiter Teil der Bedingung  
    //Übertragen bedingungHallo == Egon ? Ja : Nein  
    case "Egon":  
        System.out.println("Egon");  
        break;  
    //Sollte keine der Bedingungen Wahr sein dann wird default aufgerufen  
    default:  
        // default Anweisungs Block  
        // wird ausgeführt wenn die Bedingung falsch ist  
        System.out.println("Keins von denen");  
}
```

1.3

Kontrollstrukturen

Schleifen for & for(each)



1.3 Kontrollstrukturen

- **Schleifen: for**

Schleifen dienen zum mehrfachen durchlaufen von Datenstrukturen. Bei der Initialisierung wird der Startwert festgelegt, mit dem die Schleife beginnen soll. Mit der Abbruchbedingung legen wir fest, zu welchem Zeitpunkt der Durchlauf => Iteration diese beendet werden soll. Um diesen zu erreichen benötigen wir einen Zähler => Iterator, welcher nach jedem Durchlauf verändert wird um die Abbruchbedingung zu erreichen.

```
//Initialisierung;           Abbruch Bedingung; zum Erreichen der Abbruch Bedingung
for(   int i = 0;           i < 10;           i++)
{
    // für jeden Durchlauf => Iteration => führen wir den Schleifen Körper aus
    System.out.println("Hello World " + i);
}
```

Achtung ⚠:

Sollte die Abbruchbedingung nie erreicht werden, kommt es zu einem StackOverflowError

1.3 Kontrollstrukturen

- **Schleifen for => foreach**

Die for => foreach Schleife durchläuft die komplette Datenstruktur iterativ => Element für Element. Hierbei wird zuerst der Datentyp des jeweiligen Elements Benannt um auf dieses zuzugreifen => der Iterator. Ist die Datenstruktur einmal komplett durchlaufen endet die foreach Schleife automatisch.

```
3 // Initialisierung des Iterators : zu durchlaufende Datenstruktur
3 for (String element : eineDatenStruktur) {
1     System.out.println(element);
2 }
```

Achtung ⚠ :

Mit der foreach Schleife lässt sich auf eine Datenstruktur nur lesend zugreifen => nicht schreibend oder manipulierend!

1.4

Kontrollstrukturen

Schleifen while & do-while



1.4 Kontrollstrukturen

- **Schleifen: while**

Die While Schleife zählt zu den Kopfgesteuerten Schleifen. Zum betreten dieser muss die Eintrittsbedingung zu true evaluieren. Gleichzeitig ist dies auch die Abbruchsbedingung nach betreten des Schleifen Körpers. Folglich muss eine Veränderung der Bedingung innerhalb der Schleife stattfinden.

```
// Zum Eintritt muss die Bedingung true sein
// sonst wird die while nie betreten
// Abbruch und Eintritts Bedingung
while (Bedingung) {
    System.out.println("in der While");
    // update der Eintrittsbedingung
    Bedingung = false;
}
```

Achtung ⚠:

Sollte die Abbruchbedingung nie erreicht werden, kommt es zu einem StackOverflowError

1.4 Kontrollstrukturen

• Schleifen: do-while

Die Do-While Schleife zählt zu den fußgesteuerten Schleifen. Sie wird im Vergleich zur While-Schleife mindestens einmal durchlaufen, da die Abbruchbedingung erst am Ende geprüft wird. Auch hier ist zu beachten, daß die Abbruchbedingung innerhalb des Schleifen Körpers geändert wird, da dies sonst zu einer Endlosschleife führt.

```
do {  
    System.out.println("in der Do - While");  
    // update der Eintrittsbedingung  
    Bedingung = false;  
} while (Bedingung);  
// Zum Austritt muss die Bedingung true sein  
// sonst wird die while nie betreten  
// Abruch und Eintritts Bedingung
```

Achtung ⚠:

Sollte die Abbruchbedingung nie erreicht werden, kommt es zu einem StackOverflowError

1.5 Kontrollstrukturen

Try-Catch-Finally



1.4 Kontrollstrukturen

• Try-Catch-Finally

Der Try Catch Block dient zur Bahndlung von schweren Fehlern, die „möglicherweise“ auftreten können. Bei diesen Fehlern spricht man von Exceptions. Er besteht immer aus mindestens einem Try und einem Catch. Optional kann es auch mehrere Catch Blöcke geben. Ergänzt wird das Konstrukt durch einen Finally Block.

- Try
=> zum ausführen von unsicherem Quellcode z.B.:

```
byte b = 5;  
int b1 = b / 0;
```
- Catch
=> abfangen des etwaigen Fehlers : in diesem Fall ein fehlerhafter Berechnung : Null Division
- Finally
=> dieser Block, sollte er aufgeführt sein, wird immer ausgeführt.

1.4 Kontrollstrukturen

- Try-Catch-Finally

Im Moment reicht es zu verstehen, daß es eine besondere Kontrollstruktur für die Validierung von User-Eingaben gibt.

Try-Catch wird uns im weiteren Verlauf des Kurses noch häufig begegnen. Daher ist es sinnvoll, schon früh Bekanntschaft damit zu machen.

```
try {  
    // probiere etwas  
    byte b = 5;  
    int b1 = b / 0;  
} catch (ArithmeticException ex) {  
    // bei Fehlern kann hier das Programm vor dem Absturz  
    // bewahrt werden und z.B. den Fehler loggen  
    System.out.println("Fehler bei der berechnung");  
} catch (Exception e) {  
    // optional => wenn der Fehler nicht dem ersten Fehler entspricht  
} finally {  
    // hier könnten Aufräumarbeiten erledigen werden  
}
```



Java

02 Aufgaben

1. *Aufgabe*
2. *Lösung*



2.1 Aufgaben

Aufgabe



2.1 Aufgaben

Erstellen Sie ein Programm welches ein Quadrat auf der Konsole ausgibt. Die Kantenlänge können Sie selbst bestimmen.

Zum Beispiel:

```
#####  
#####  
#####  
#####  
#####  
#####  
#####  
#####  
#####  
#####
```



2.2 Aufgaben

Lösung



2.2 Aufgaben

Erstellen Sie ein Programm welches ein Quadrat auf der Konsole ausgibt. Die Kantenlänge können Sie selbst bestimmen.

```
int kantenlänge = 10;
for (int i = 0; i < kantenlänge; i++) {
    for (int j = 0; j < kantenlänge; j++) {
        System.out.print("# ");
    }
    System.out.println();
}
```





03 Methoden

1. *Generelles*
2. *Signatur*
3. *Modifier*



3.1 Methoden

Generelles



3.1 Methoden

- **Generelles**

Methoden werden in der Programmierung vor allem verwendet, um Code besser zu strukturieren und diesen auch wiederverwendbar zu machen. Der Begriff stammt aus der OOP in der Objekte über Methoden manipuliert werden. Eine der wichtigsten Methoden haben wir bereits kennengelernt

=> Die main-Methode:

```
13  
14 public static void main(String[] args) {  
15
```

Die Begriffe „Funktion“ und „Methode“ sind nahezu gleichwertig. Wir müssen uns einfach nur an die neue Benennung gewöhnen, da sie sich in der OOP durchgesetzt hat.

Mit dem Aufbau einer Methode werden wir uns nun genauer beschäftigen.

3.2 Methoden

Signatur



3.2 Methoden

• Signatur

Der Aufbau einer Funktion folgt immer dem selben Schema:

Deklaration

```
//Modifizier  StatischerZugriff  Rückgabety  
public      static           void
```

```
Methoden Name  Parameter-Liste  
main          (String[] args) {
```

Signatur

1. Zugriffs-Modifizierer (Beschränkt den Zugriff auf eine Methode)
2. Statischer Zugriff (Aufruf ohne Objektbezug)
3. Rückgabety
 1. Void => diese Methode liefert keinen Wert zurück
 2. Ein Datentyp (z.B. int) => diese Methode liefert einen Integer-Wert zurück

Deklaration

4. Methoden-Name (im Rahmen der üblichen Konventionen frei wählbar)
5. Parameter (optional können der Methode ein oder mehrere Wert(e) übergeben werden)

Signatur

3.2 Methoden

• Methoden-Rumpf

Nach dem Methoden-Kopf folgt der Methoden-Rumpf. Dieser steht innerhalb der geschweiften Klammern. Hier wird die Methoden-Logik implementiert.

```
//Modifizier Rückgabetyt Methoden Name Parameter Liste  
public String meineFunction ()  
{  
    // Methoden Rumpf  
    System.out.println("Meine Methode");  
    // Rückgabe  
    return "meine Funktion";  
}
```

```
//Modifizier Rückgabetyt Methoden Name Parameter Liste  
public String meineFunction ()  
{  
    // Methoden Rumpf  
    System.out.println("Meine Methode");  
    // Rückgabe  
    return "meine Funktion";  
}
```

Bei einer Methode mit Rückgabetyt wird zwingend das Schlüsselwort „return“, in Verbindung mit einem Wert, der dem Rückgabetyt entspricht, benötigt.

3.3 Methoden

Modifizierer



3.1 Modifizierer

- Modifizierer (Zugriffsbeschränkung)

Java bietet die Möglichkeit Zugriffsbeschränkungen zu vergeben. Dies dient der Kapselung von Daten.

Die Zugriffsmodifizierer regeln, von wo innerhalb der Gesamt-Struktur auf ein Element zugegriffen werden kann.

Diese Modifizierer können für Attribute/Variablen, Methoden und auch Klassen eingesetzt werden.

Reichweite	Innerhalb der Klasse	Paket-Klassen/ innere Klassen	Unterklassen	Sonstige Klassen
private	Ja	Nein	Nein	Nein
(ohne = default)	Ja	Ja	Nein	Nein
protected	Ja	Ja	Ja	Nein
public	Ja	Ja	Ja	Ja



Java

04 Aufgaben

1. *Aufgabe*
2. *Lösung*



4.1

Aufgaben

Aufgabe



4.1 Aufgaben

Erstellen Sie ein Projekt „Schiffe versenken“ im Ordner „Baustein Projekt“. Es muss eine „Programm“ Klasse im Package „main“ haben. Diese beinhaltet die main Methode sowie eine Klasse „Spielbrett“ im Package „ressourcen“. Beide Packages liegen im Ordner „src“. Die Spielbrett Klasse muss auf der Konsole 2 Quadrate von 10 auf 10 mit den Koordinaten 1-10 => horizontal und A-J => vertikal ausgeben.

4.2 Aufgaben

Lösung



4.2 Aufgaben

BausteinProjekt_V1.7z





VIELEN DANK!

