Programmiersprache

Java

2022 / Teil 3

Detlef Wilkening
www.wilkening-online.de

Programmiersprache Java

6 Operatoren 2
6.1 Zuweisungs-Operatoren 3
6.2 Gleich- und Ungleich-Operatoren 4
6.3 Geteilt- und Modulo-Operator 5
6.4 Bitweises AND, OR, XOR 7
6.5 Boolesches bzw. bedingtes AND, OR, XOR 7
6.6 Prä- und Post-Inkrement und -Dekrement Operatoren 8
6.7 Schiebe-Operatoren 9

7 Kontroll-Strukturen 9
7.1 Bedingter-Kontrollfluss – If 9
7.2 Mehrfach-Verzweigung – Switch 12
7.3 Switch-Expressions und Pattern-Matching 15
7.4 For-Schleife 17
7.5 For-Schleife für Container und Arrays 18
7.6 While-Schleife 18
7.7 Do-Schleife 18
7.8 Break- und Continue-Anweisung 19
7.9 Gelabelte Sprung-Anweisungen 20

6 Operatoren

Folgende Operatoren stellt Java zur Verfügung.

Vorr.	Operator	Operandentyp(en)	As.	Operation
1	.	Objekt	L	Zugriff auf Objekt-Element
	[ ]	Array, Integer	L	Zugriff auf Array-Element
	(args)	Funktion	L	Funktionsaufruf
	++, --	arithmetisch	L	Postinkrement, -dekrement (unär)
2	++, --	arithmetisch	R	Präinkrement, -dekrement (unär)
	+, -	arithmetisch	R	unäres Plus, unäres Minus
	~	integral	R	bitweises Komplement (unär)
	!	boolean	R	logisches Komplement (unär) (logisches Not)
3	new	Klasse	R	Objekterzeugung
	(Typ)	beliebig	R	Typumwandlung
4	*, /, %	arithmetisch	L	Multiplikation, Division, Rest
5	+, -	arithmetisch	L	Addition, Subtraktion
	+	String	L	String-Verkettung
6	<<	integral	L	Links-Shift

	>>	integral	L	Rechts-Shift mit Vorzeichen-Erweiterung
	>>>	integral	L	Rechts-Shift mit Null-Erweiterung
7	<, <=	arithmetisch	L	kleiner, kleiner-gleich
	>, >=	arithmetisch	L	grösser, grösser-gleich
	instanceof	Typ	L	Typvergleich
8	==	einfach	L	gleich (identische Werte)
	!=	einfach	L	ungleich (unterschiedliche Werte)
	==	Objekt	L	gleich (identisches Objekt)
	!=	Objekt	L	ungleich (unterschiedliche Objekte)
9	&	integral	L	bitweises AND
	&	boolean	L	boolsches AND (alle Operanden werden ausgewertet)
10	^	integral	L	bitweises XOR
	^	boolean	L	boolsches XOR (alle Operanden werden ausgewertet)
11	|	integral	L	bitweises OR
	|	boolean	L	boolsches OR (alle Operanden werden ausgewertet)
12	&&	boolean	L	bedingtes AND (Abbruch bei feststehendem Ergebnis)
13	||	boolean	L	bedingtes OR (Abbruch bei feststehendem Ergebnis)
14	?:	boolean, beliebig	R	bedingter (ternärer) Operator
15	=	beliebig	R	Zuweisung
	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	beliebig	R	Zuweisung mit Operation

- Die meisten dieser Operatoren sollten intuitiv klar sein, da Sie für diese Vorlesung etwas Programmier-Erfahrung mitbringen mussten.
- Ein paar Operatoren sind nicht zwingend intuitiv in ihrer Benutzung und Semantik, so dass sie im Weiteren besprochen werden.
- Ein weiterer Teil der Operatoren macht erst im späteren Verlauf der Vorlesung Sinn, da uns hier noch das entsprechende notwendige Sprachverständnis fehlt. Sie werden in den jeweiligen Themen-Kapiteln vorgestellt, z.B. der Operator „instanceof“.

6.1 Zuweisungs-Operatoren

Der normale Zuweisungs-Operator ‘=’ weist den Wert des rechten Ausdrucks der Variablen auf der linken Seite zu.

Zusätzlich gibt es in Java noch Zuweisungs-Operatoren mit Operationen. Die Anweisung ‘erg #= arg;’ entspricht dabei immer der Anweisung ‘erg = erg # (arg);’.

```
int i = 7;
int j = 4;
i += 3*j;                    // entspricht:  i = i + (3*j);
```

```
System.out.println(i);     // Ausgabe: 19
```

6.2 Gleich- und Ungleich-Operatoren

Bei den elementaren Datentypen vergleichen die Operatoren '==' und '!=' den Wert, während bei Objekten die Identität (gleiches bzw. verschiedenes Objekt) verglichen wird.

Fangen wir mit einem Beispiel an, wo wir sehen, dass bei int's mit den Werten gearbeitet wird:

```
int i = 7;
int j = 7;

if (i==j)
   System.out.println("Gleich");      // Ausgabe: Gleich
else
   System.out.println("Ungleich");

j = 8;
if (i==j)
   System.out.println("Gleich");
else
   System.out.println("Ungleich");  // Ausgabe: Ungleich
```

Ganz anders ist das Verhalten bei Objekten, wo mit „==" bzw. „!=" die Identität verglichen wird:

```
StringBuilder sb1 = new StringBuilder("sb");
StringBuilder sb2 = new StringBuilder("sb");

if (sb1==sb2)
   System.out.println("Gleich");
else
   System.out.println("Ungleich");    // Ausgabe: Ungleich

sb2 = new StringBuilder("xxx");
if (sb1==sb2)
   System.out.println("Gleich");
else
   System.out.println("Ungleich");    // Ausgabe: Ungleich

sb2 = sb1;
if (sb1==sb2)
   System.out.println("Gleich");      // Ausgabe: Gleich
else
   System.out.println("Ungleich");
```

Achtung – dies ist ein gern gemachter Fehler. Wert-Vergleiche auf Objekten müssen meist mit der Element-Funktion „equals" gemacht werden, da „==" nur die Identität vergleicht. Leider muss man hier sehr vorsichtig sein, da es Klassen gibt bei denen auch „equals" nur die Identität vergleicht und nicht den Wert.

```
StringBuilder sb1 = new StringBuilder("sb");
StringBuilder sb2 = new StringBuilder("sb");

if (sb1.equals(sb2))
   System.out.println("Gleich");      // Ausgabe: Gleich
else
   System.out.println("Ungleich");

sb2 = new StringBuilder("xxx");
if (sb1.equals(sb2))
   System.out.println("Gleich");
else
   System.out.println("Ungleich");    // Ausgabe: Ungleich
```

```
sb2 = sb1;
if (sb1.equals(sb2))
   System.out.println("Gleich");      // Ausgabe: Gleich
else
   System.out.println("Ungleich");
```

Achtung – ganz gefährlich wird es bei Strings. In einem späteren Kapitel werden wir lernen, dass Strings „immutable“, d.h. unveränderlich, sind. Daher kann die JVM gleiche Strings ohne Gefahr zusammenlegen, um z.B. Speicher zu sparen. Daher können einzelne – eigentlich nicht identische Objekte – identisch sein. Es kann also sein, dass Sie in Ihrem Programm bei Strings fehlerhafterweise „==“ statt „equals“ verwenden, und es erstmal läuft. Gefährlich ist dies deshalb, weil dieses Verhalten nicht zugesichert ist, sondern sich in anderen JVMs wieder ändern kann.

```
String s1 = "Hallo";
String s2 = "Hallo";      // Kann vielleicht auf das gleiche Objekt zeigen

if (s1==s2) {                                    // Identitaets-Abfrage
   System.out.println("Identisch ");
}
else {
   System.out.println("NICHT identisch ");
}

if (s1.equals(s2)) {                             // Defenitiv Wert-Vergleich
   System.out.println("Gleicher Wert");
}
```

```
mögliche Ausgabe
NICHT identisch
Gleicher Wert
```

6.3 Geteilt- und Modulo-Operator

Während die *normalen* mathematischen Operatoren (plus, minus und mal) in ihrem Verhalten ziemlich intuitiv sind, ist dies beim Geteilt- und Modulo-Operator vielleicht nicht so.

6.3.1 Geteilt-Operator

Im Prinzip arbeitet auch der Geteilt-Operator ganz intuitiv, außer man wendet ihn auf integrale Typen (byte, short, int und long) an und erwartet dann ein Gleitkomma-Ergebnis (z.B. double). Das funktioniert so nicht – die Division bleibt im Bereich der Typen der Operanden, und das betrifft auch das Ergebnis.

Teilen Sie Integer durch Integer, so ist das Ergebnis wieder ein Integer – selbst wenn das Ergebnis damit nicht exakt ist. Das Ergebnis ist nur der ganz-teilige Anteil der Division, den Rest kann man mit dem Modulo-Operator bestimmen – siehe Kapitel 6.3.2.

```
System.out.println("10 /  1 -> " + (10 / 1));    // -> 10
System.out.println("10 /  2 -> " + (10 / 2));    // ->  5
System.out.println("10 /  3 -> " + (10 / 3));    // ->  3
System.out.println("10 /  4 -> " + (10 / 4));    // ->  2
System.out.println("10 /  5 -> " + (10 / 5));    // ->  2
System.out.println("10 /  6 -> " + (10 / 6));    // ->  1
System.out.println("10 /  7 -> " + (10 / 7));    // ->  1
```

```
System.out.println("10 /  8 -> " + (10 / 8));    // ->  1
System.out.println("10 /  9 -> " + (10 / 9));    // ->  1
System.out.println("10 / 10 -> " + (10 / 10));   // ->  1
System.out.println("10 / 11 -> " + (10 / 11));   // ->  0
System.out.println("10 / 12 -> " + (10 / 12));   // ->  0
```

Ausgabe

```
10 /  1 -> 10
10 /  2 -> 5
10 /  3 -> 3
10 /  4 -> 2
10 /  5 -> 2
10 /  6 -> 1
10 /  7 -> 1
10 /  8 -> 1
10 /  9 -> 1
10 / 10 -> 1
10 / 11 -> 0
10 / 12 -> 0
```

Wie dividiert man aber nun zwei Integer-Zahlen so, dass man das korrekte Gleitkomma-Zahlen Ergebnis erhält? Ganz einfach: man muss vor der Division mindestens einen der beiden Operanden in den gewünschten Gleitkomma-Zahlen-Typ („float“ oder „double“) konvertieren.

```
final int divisor = 7;
final int divident = 2;

int res1 = divisor / divident;
double res2 = (double) divisor / divident;
double res3 = divisor / (double) divident;
double res4 = (double) divisor / (double) divident;

System.out.println("res1 -> " + res1);   //  -> 3
System.out.println("res2 -> " + res2);   //  -> 3,5
System.out.println("res3 -> " + res3);   //  -> 3,5
System.out.println("res4 -> " + res4);   //  -> 3,5
```

Ausgabe

```
res1 -> 3
res2 -> 3.5
res3 -> 3.5
res4 -> 3.5
```

Hinweis – für Konvertierungen zwischen elementaren Datentypen siehe früheres Kapitel.

6.3.2 Modulo-Operator

Der Modulo-Operator kann nur auf integrale Typen angewendet werden, und liefert dann den Rest der Integer-Division – siehe Kapitel 6.3.1.

```
System.out.println("10 %  1 -> " + (10 % 1));    // ->  0
System.out.println("10 %  2 -> " + (10 % 2));    // ->  0
System.out.println("10 %  3 -> " + (10 % 3));    // ->  1
System.out.println("10 %  4 -> " + (10 % 4));    // ->  2
System.out.println("10 %  5 -> " + (10 % 5));    // ->  0
System.out.println("10 %  6 -> " + (10 % 6));    // ->  4
System.out.println("10 %  7 -> " + (10 % 7));    // ->  3
System.out.println("10 %  8 -> " + (10 % 8));    // ->  2
System.out.println("10 %  9 -> " + (10 % 9));    // ->  1
System.out.println("10 % 10 -> " + (10 % 10));   // ->  0
System.out.println("10 % 11 -> " + (10 % 11));   // -> 10
System.out.println("10 % 12 -> " + (10 % 12));   // -> 10
```

Ausgabe

```
10 %  1 -> 0
```

```
10 %  2 -> 0
10 %  3 -> 1
10 %  4 -> 2
10 %  5 -> 0
10 %  6 -> 4
10 %  7 -> 3
10 %  8 -> 2
10 %  9 -> 1
10 % 10 -> 0
10 % 11 -> 10
10 % 12 -> 10
```

6.4 Bitweises AND, OR, XOR

Überarbeiten todo

Die Operatoren ‘&’, ‘|’ und ‘^’ arbeiten sowohl mit integralen als auch mit booleschen Datentypen zusammen.

Bei integralen Datentypen werden die interne Darstellungen bitweise miteinander verknüpft.

AND

Op. 1	Op.2	Erg.
0	0	0
0	1	0
1	0	0
1	1	1

OR

Op. 1	Op.2	Erg.
0	0	0
0	1	1
1	0	1
1	1	1

XOR

Op. 1	Op.2	Erg.
0	0	0
0	1	1
1	0	1
1	1	0

6.5 Boolesches bzw. bedingtes AND, OR, XOR

Bei booleschen Operanden werden die Operanden logisch miteinander verknüpft, wobei immer **alle** Operanden ausgewertet werden.

Bei den bedingten Operatoren ‘&&’ und ‘||’ werden die Operanden auch logisch miteinander verknüpft, aber die Auswertung wird abgebrochen wird, sobald das Ergebnis feststeht. Es kann daher passieren, dass nicht alle Operanden ausgewertet werden.

```
  public static boolean fct(int i) {
    System.out.println("fct(" + i + ')');
    return true;
  }

  public static void main(String[] args) {
    boolean erg, b = true;
    erg = b & fct(1);
    erg = b | fct(2);
    erg = b ^ fct(3);
    erg = b && fct(4);
    erg = b || fct(5);
    b = false;
    erg = b & fct(6);
    erg = b | fct(7);
    erg = b ^ fct(8);
    erg = b && fct(9);
```

```
    erg = b || fct(10);
  }
```

```
Ausgabe
fct(1)
fct(2)
fct(3)
fct(4)
fct(6)
fct(7)
fct(8)
fct(10)
```

6.6 Prä- und Post-Inkrement und -Dekrement Operatoren

Den Operator '++' gibt es in Prä- und Postfix-Notation. Er ist für alle arithmetischen Typen definiert und entspricht einer Erhöhung um '1'. Analog entspricht der '—' Operator einer Erniedrigung um '1'.

```
int i = 7;
i++;
System.out.println(i);          // Ausgabe: 8
++i;
System.out.println(i);          // Ausgabe: 9
i--;
System.out.println(i);          // Ausgabe: 8
--i;
System.out.println(i);          // Ausgabe: 7
```

Im obigen Beispiel ist die Unterscheidung in Prä- und Postfix-Notation egal, aber in anderen Zusammenhängen ist sie es nicht.

Diese Operatoren können direkt in arithmetischen Ausdrücken benutzt werden - in so einem Fall entscheidet die Notation, ob die Variable zuerst ausgewertet und dann verändert wird, oder umgekehrt:

- Präfix-Notation '++x' bzw. '–-x'
  1. verändern von 'x'
  2. auswerten von 'x'
- Postfix-Notation 'x++' bzw. 'x--'
  1. auswerten von 'x'
  2. verändern von 'x'

```
int i = 7;
int j = ++i;
System.out.println(i + " " + j);        // Ausgabe: 8 8
j = i--;
System.out.println(i + " " + j);        // Ausgabe: 7 8
```

Im Gegensatz zu einigen anderen Programmiersprachen wie z.B. C und altem C++ ist auch bei Verwendung mehrere dieser Operatoren in einer Anweisung das Ergebnis exakt definiert, da es in Java eine eindeutige Abarbeitung von Ausdrücken gibt: zuerst werden bei einer Anweisung die Operanden von links nach rechts ausgewertet, dann wird der Rest der Anweisung durchgeführt – mit der Ausnahme von Operanden bei bedingtem Und und Oder!

```
int i = 2;
int j = ++i + ++i * ++i;     // entspricht 3+4*5
```

```
System.out.println(i);        // Ausgabe: 23
```

6.7 Schiebe-Operatoren

Die Schiebe-Operatoren '<<', '>>' und '>>>' wirken nur auf integrale Datentypen.

Der Operator '<<' schiebt die Bits des integralen Operanden um n-Bits nach links, d.h. er bewirkt bei allen Zahlen (negativ und positiv) eine Multiplikation mit 2.

```
int i = 3;
int j = -5;
System.out.println((i<<1) + " " + (j<<2));       // Ausgabe: 6 -20
```

Der Operator '>>' schiebt die Zahl um n-Bits nach rechts mit Vorzeichen-Erweiterung. D.h. das Vorzeichen der Zahl bleibt erhalten. Dagegen füllt der Operator '>>>' die Zahl immer mit Null-Bits auf - er betrachtet die Zahl quasi immer als vorzeichenlos.

```
int i = 5;
int j = -5;
int k = -5;
System.out.println((i>>1)+" "+(j>>1)+" "+(k>>>1));    // Ausgabe: 2 -3 2147483645
```

7 Kontroll-Strukturen

7.1 Bedingter-Kontrollfluss – If

Die wichtigste Kontroll-Struktur ist die If-Anweisung. Sie ermöglicht einen bedingten Sprung, d.h. in Abhängigkeit von einem Booleschen-Ausdruck unterschiedliche Pfade im Programm zu durchlaufen.

7.1.1 Einfachste Form

Die einfachste Form der If-Anweisung sieht folgendermaßen aus:

Syntax:

```
if (boolean-ausdruck)
   true-anweisung
```

Sie beginnt mit dem Schlüsselwort „if" – dann folgt in runden Klammern ein Ausdruck, der vom Typ „boolean" sein muss. Abgeschlossen wird sie durch eine Anweisung. Diese Anweisung wird nur ausgeführt, wenn der Ausdruck zu „true" ausgewertet wurde. Im Falle von „false" wird die Anweisung übersprungen.

```
int n = 5;

if (n<7)
   System.out.println("n<7");

if (n>20)
   System.out.println("n>20");
```

```
if (n!=6)
   System.out.println("n!=6");
```

Ausgabe
```
n<7
n!=6
```

7.1.2 Blöcke für mehrere Anweisungen

Soll im If-Zweig mehr als eine Anweisung ausgeführt werden, so muss mit den geschweiften Klammern ein Block gebildet werden. Ein solcher Block steht dabei syntaktisch für eine einzelne Anweisung.

```
int n = 5;

if (n<7) {
   System.out.println("n<7");
   System.out.println("n ist " + n);
   System.out.println("Und weiter geht es...");
}
```

Ausgabe
```
n<7
n ist 5
Und weiter geht es...
```

Hinweis – dies gilt genauso für Schleifen oder andere Konstrukte, die normalerweise nur auf eine Anweisung wirken.

7.1.3 If-Else Anweisung

Für den Fall, dass neben dem True-Fall auch beim False-Fall Aktionen ausgeführt werden sollen, kann auf die Anweisung (bzw. dem Block) des True-Falls das Schlüsselwort „else“ mit einer Anweisung für den False-Fall folgen.

Syntax:

```
if (boolean-ausdruck)
   true-anweisung
else
   false-anweisung
```

Beispiel:

```
int n = 5;

if (n<7)
   System.out.println("n<7");
else
   System.out.println("n>=7");

if (n>20)
   System.out.println("n>20");
else
   System.out.println("n<=20");
```

Ausgabe
```
n<7
n<=20
```

Hinweis – auf für den False-Zweig gilt natürlich: Wenn mehr als eine Anweisung notwendig ist, dann muss ein Block genutzt werden.

7.1.4 If-Anweisung ohne True-Zweig

Es gibt keine Syntax, um eine If-Anweisung nur mit False-Zweig ohne True-Zweig zu implementieren. Ist dies gewünscht, so gibt es zwei Implementierungs-Möglichkeiten:

- True-Zweig mit leerer Anweisung
- Bool-Ausdruck in der If-Anweisung umformulieren

True-Zweig mit leerer Anweisung

In Java sind leere Anweisungen erlaubt – sie bestehen aus einem einzelnen Semikolon. Damit lässt sich der True-Zweig einer If-Anweisung sehr schnell abhandeln.

```
int n = 10;

if (n<7);                    // <= dieses Semikolon ist der leere True-Zweig
else
   System.out.println("n>=7");
```

```
Ausgabe
n>=7
```

Achtung – so ein leerer True-Zweig ist nicht besonders gut lesbar. Und so ein einzelnes Semikolon ist schnell überlesen bzw. verwirrend – von daher ist die Lösung nicht zu empfehlen. Formulieren Sie besser den Bool-Ausdruck um.

Bool-Ausdruck in der If-Anweisung umformulieren

Formulieren Sie den boolschen Ausdruck so um, dass er statt „true“ „false“ liefert und umgekehrt. Wenn sich der Ausdruck nicht umformulieren läßt, bzw. die Umformulierung aufwändig und fehlerträchtig ist – dann nutzen Sie einfach den booleschen Not-Operator „!“ um den boolschen Ausdruck zu negieren.

```
int n = 10;

if (n>=7)                          // Ausdruck umformuliert
   System.out.println("n>=7");

if (!(n<7))                        // Ausdruck negiert mit !
   System.out.println("n>=7");
```

```
Ausgabe
n>=7
n>=7
```

7.1.5 Mehrfach-Verzweigungen mit If-Else-If

Nicht immer gibt es nur einen True-Fall bzw. einen True- und einen False-Fall. Häufig hat man eine Art Auflistung von verschiedenen Fällen, d.h. eine Mehrfach-Verzweigung. In diesem Fällen nimmt man verschachtelte If-Anweisungen, die auch als If-Else-If-Anweisung bezeichnet wird.

Im Prinzip ist eine If-Else-If-Anweisung nichts Neues – hier wird nur der False-Zweig durch eine If-Anweisung dargestellt. Der Unterschied ist eine eigenständige Einrückungs-Konvention, bei der das „if“ direkt auf das „else“ folgt.

Syntax:

```
if (boolean-ausdruck)
   true-anweisung
else if (boolean-ausdruck)
   true-2-anweisung
else
   false-anweisung
```

Beispiel:

```
int age = 27;
if (age<10)
   System.out.println("Kind im Alter von " + age);
else if (age<18)
   System.out.println("Jugendliche(r) im Alter von " + age);
else if (age<30)
   System.out.println("Junge(r) Erwachsene(r) im Alter von " + age);
else if (age<70)
   System.out.println("Erwachsene(r) im Alter von " + age);
else
   System.out.println("Alte(r) Frau/Mann im Alter von " + age);
```

```
Ausgabe
Junge(r) Erwachsene(r) im Alter von 27
```

Hinweis – auch hier ist der Else-Zweig natürlich optional.

7.2 Mehrfach-Verzweigung – Switch

Für spezielle Mehrfach-Verzweigungen in Abhängigkeit von einer integralen Variablen gibt es in Java die Switch-Anweisung. Sie funktioniert für die Typen „char“, „byte“, „short“ und „int“, d.h. Typen mit integralen Werten mit max. 4 Byte Größe, und zusätzlich seit dem JDK 1.5 mit Enums und seit dem JDK 1.7 auch mit Strings.

Außerdem kann der integrale Ausdruck nur auf Gleichheit zu Compile-Zeit Konstanten des entsprechenden Typs getestet werden. Wenn man mit diesen Einschränkungen leben kann, ist eine Switch-Anweisung einer If-Else-If Mehrfach-Verzweigung vorziehen.

Syntax

```
switch (ausdruck) {
case constant1:
   anweisung; ...
   [break;]
case constant2:
   anweisung; ...
   [break;]
...
default:
   anweisung; ...
   [break;]
}
```

- **ausdruck** muss vom Typ „char“, „byte“, „short“, „int“, „Enum“ (seit JDK 1.5) oder „String“ (seit JDK 1.7) sein.
- **constant** muss eine Compile-Zeit-Konstante vom entsprechenden Typ sein.

```
public class Appl {

   public static void main(String[] args) {
      for (int i = 0; i < 5; i++) {
         switch (i) {
         case 0:
            System.out.println("Null");
            break;
         case 1:
            System.out.println("Eins");
            break;
         case 3:
            System.out.println("Drei");
            break;
         default:
            System.out.println("Unbekannt");
         }
      }
   }

}
```

```
Ausgabe
Null
Eins
Unbekannt
Drei
Unbekannt
```

Erst die „break“ Anweisung beendet einen Case-Block. Ist kein „break“ vorhanden, so läuft der Programmfluss in den nächsten Case-Block hinein – auch in den Default-Block. Dies nennt man „Fall-Through“. Auch im Default-Block darf das „break“ stehen – hat hier aber keine Bedeutung – siehe Beispiel:

```
for (int i=0; i<7; i++) {
  switch (i) {
    case 1:
      System.out.println("i ist 1");
      break;
    case 2:
    case 3:
      System.out.println("i ist 2 oder 3");
    case 4:
      System.out.println("Fluss kommt aus 2/3 oder i ist 4");
      break;
    default:
      System.out.println("i ist weder 1,2,3 oder 4");
      break;                                          // Nicht notwendig
  }
}
```

```
Ausgabe
i ist weder 1,2,3 oder 4
i ist 1
i ist 2 oder 3
Fluss kommt aus 2/3 oder i ist 4
i ist 2 oder 3
Fluss kommt aus 2/3 oder i ist 4
Fluss kommt aus 2/3 oder i ist 4
i ist weder 1,2,3 oder 4
```

```
i ist weder 1,2,3 oder 4
```

Seit JDK 1.5 kann man die Switch-Anweisung auch für Enums (siehe späteres Kapitel) verwenden. Achtung, das Beispiel enthält mehrere Sprachmittel (u.a. die Enums), die erst in späteren Kapiteln eingeführt werden:

```
public class Appl {

   private enum Color {
      RED, GREEN, BLUE
   };

   private static void switchIt(Color c) {
      switch (c) {
      case RED:
         System.out.println("rot");
         break;
      case GREEN:
         System.out.println("gruen");
         break;
      case BLUE:
         System.out.println("blau");
         break;
      }
   }

   public static void main(String[] args) {
      switchIt(Color.RED);
      switchIt(Color.GREEN);
      switchIt(Color.BLUE);
   }

}
```

Ausgabe

```
rot
gruen
blau
```

Und seit JDK 1.7 funktioniert die Switch-Anweisung auch mit Strings.

```
public class Appl {

   private static void doStringSwitch(String s) {
      switch (s) {
      case "Text 1":
         System.out.println("Text 1 gefunden");
         break;
      case "Text 2":
         System.out.println("Text 2 gefunden");
         break;
      default:
         System.out.println("Unbekannter Text");
         break;
      }
   }

   public static void main(String[] args) {
      doStringSwitch("Text 1");
      doStringSwitch("Text 2");
      doStringSwitch("Text 3");
   }

}
```

Ausgabe

```
Text 1 gefunden
Text 2 gefunden
```

```
Unbekannter Text
```

Hinweis – der Java Compiler wandelt die Switch-Anweisungen mit Strings in eine Art „Hash-Table“ mit Sprunganweisung um. Daher wird die Switch-Anweisungen bei vielen Case-Fällen wahrscheinlich performanter sein als If-Else-If-Else-Anweisungen.

7.3 Switch-Expressions und Pattern-Matching

Über die Java Versionen 12 (erste Preview) bis 18 hinweg sind in Java Stück für Stück Expressions und Pattern-Matching für u.a. die Switch Anweisung eingeführt worden – und der Prozess ist noch nicht am Ende. Die normale Switch Anweisung hat viele Einschränkungen und Unschönheiten, die man durch diese Erweiterungen beseitigen möchte, z.B.:

- Einschränkung auf wenige Datentypen
- Überprüfung nur auf Gleichheit
- Das Fall-Trough Verhalten ist fehleranfällig
- Schwierig zu überprüfen, ob alle Fälle abgedeckt sind
- Null ist nicht erlaubt

Mit Java 12-14 wurde die Switch Anweisung zuerst um die sogenannten Switch-Expressions erweitert. Mit Java 14-16 wurden Records (die wir nicht in der Vorlesung besprechen werden) in die Sprache Java eingeführt. Außerdem wurde mit Java 14-16 Pattern-Matching für „instanceof“ integriert (ohne Unterstützung der Switch-Anweisung). Mit Java 17 begann die Integration von Pattern-Matching für die Switch-Anweisung – in Java 17 als Preview für „instanceof“, mit Java 18 als Preview für Records und Arrays. Records werden wir aus Zeitmangel nicht in der Vorlesung besprechen können. „instanceof“ wird erst gegen Ende der Vorlesung vorgestellt, da wir dafür Klassen, Vererbung und Polymorphie benötigen. Damit sollte klar sein, dass eine tiefe Vorstellung von Switch-Expressions und Pattern-Matching im Rahmen dieser Vorlesung nicht möglich ist. Trotzdem möchte ich hier mit ein paar kleinen Beispielen zeigen, wohin die Reise geht.

7.3.1 Switch-Expressions

Switch-Expressions sind vollständig ab Java 14 in Java enthalten (Java 12 und 13 enthielten nur Previews). Die einfachste Form einer Switch-Expression ist eine Kurzform der normalen Switch-Anweisung, bei der das gesamte Switch als ein Ausdruck interpretiert wird und damit z.B. einer Variablen zuweisbar ist. Mehrere Case-Fälle können einfach durch Komma getrennt werden, das Ergebnis wird direkt mit einem Pfeil zurückgegeben, und es gibt kein „Fall-Through“ mehr.

```
public class Kap_07_03_Bsp_01_SwitchExpression {

   public static String season(int month) {
      String res = switch (month) {
        case 1 -> "Januar";
        case 2,3,4 -> "Frühling";
        case 5,6,7,8,9 -> "Sommer";
        default -> "Kalte Jahreszeit";
      };
```

```
        return res;
     }

   public static void main(String[] args) {
      for (int i = 1; i < 13; i++) {
         System.out.println(i + ": " + season(i));
      }
   }

}
```

```
Ausgabe
1: Januar
2: Frühling
3: Frühling
4: Frühling
5: Sommer
6: Sommer
7: Sommer
8: Sommer
9: Sommer
10: Kalte Jahreszeit
11: Kalte Jahreszeit
12: Kalte Jahreszeit
```

Hierbei können auch Code-Stücke in den Case-Zweigen vorkommen – sie müssen in geschweiften Klammern stehen und geben den Ergebniswert mit „yield“ zurück.

```
public class Kap_07_03_Bsp_02_SwitchExpression2 {

   public static int sum(int a, int b) {
      return switch (a) {
        case 0 -> 0;
        case 1 -> b;
        case 2 -> b+b;
        case 3 -> { yield b+b+b; }
        default -> { yield a * b; }
      };
    }

   public static void main(String[] args) {
      for (int i = 0; i < 6; i++) {
         System.out.println(i + ": " + sum(i, 7));
      }
   }

}
```

```
Ausgabe
0: 0
1: 7
2: 14
3: 21
4: 28
5: 35
```

7.3.2 Pattern-Matching für „instanceof“

Seit Java 17 ist das Pattern-Matching für „instanceof“ als Preview für die Switch-Anweisung vorhanden. Kommen Sie nochmal zu diesem Kapitel zurück, wenn Sie gegen Ende des Semesters Vererbung und „instanceof“ kennengelernt und verstanden haben. Dann verstehen Sie das Beispiel auch, dass den zur Laufzeit den Typ überprüft.

Pattern-Matching für „instanceof“ ermöglicht eine kompaktere Schreibweise statt des bis dahin

notwendigen If-Else-If mit zusätzlich Cast. Mit der Integration in die Switch-Anweisung wird der Code nochmal kürzer.

```
public class Kap_07_03_Bsp_03_PatternMatching {

   public static String evaluateObject(Object o) {
      return switch (o) {
         case null -> "nix";
         case String s -> "String";
         case Integer i -> "Integer";
         default -> "Irgendwas";
      };
   }

   public static void main(String[] args) {
      System.out.println(evaluateObject(null));
      System.out.println(evaluateObject("Java"));
      System.out.println(evaluateObject(42));
      System.out.println(evaluateObject(3.14));
   }

}
```

Ausgabe
```
nix
String
Integer
Irgendwas
```

Achtung – dies ist zurzeit (Java 17 & 18) noch ein Preview Feature und muss daher explizit aktiviert werden.

7.4 For-Schleife

Syntax

```
for (init; test; update)
   anweisung
```

Beispiel

```
for (int i=0; i<8; i++) {
   System.out.print(i);
}
```

Ausgabe
```
01234567
```

Hinweis – werden im Initialisierungs-Teil ein oder mehrere Variablen definiert, so sind diese nur im Scope der Schleife bekannt.

```
for (int i=0; i<8; i++) {  // Definition von i gilt nur fuer die Schleife
   System.out.print(i);
}
System.out.print(i);       // Compiler-Fehler - Variable i ist unbekannt
```

Hinweis – mit dem JDK 1.5 wurde ein weiterer For-Schleifentyp für Container und Arrays eingeführt – dieser wird im nächsten Kapitel 7.5 beschrieben.

7.5 For-Schleife für Container und Arrays

In Java 5 (JDK 1.5) wurde ein neuer For-Schleifen-Typ für Container und Arrays (siehe spätere Kapitel) eingeführt.

Achtung – die folgenden beiden Programme können Sie hier noch nicht vollständig verstehen, da in ihnen sowohl Arrays als auch typisierte Container vorkommen. Da es aber hier im Kapitel u.a. um Schleifen geht, habe ich diesen Schleifen-Typ hier auch beschrieben – kommen Sie später nochmal zurück zu diesem Kapitel.

Neuer Schleifentyp für Arrays:

```
int[] a = new int[]{ 1, 2, 3, 5, 7, 11 };
for (int n : a) {
   System.out.print("" + n + ' ');
}
```

```
Ausgabe
1 2 3 5 7 11
```

Neuer Schleifentyp für Container:

```
ArrayList<String> al = new ArrayList<String>();
al.add("eins");
al.add("zwei");
al.add("drei");
for (String s : al) {
   System.out.println("-> " + s);
}
```

```
Ausgabe
-> eins
-> zwei
-> drei
```

7.6 While-Schleife

Syntax

```
while (boolan-ausdruck)
   anweisung
```

Die While-Schleife wird solange wiederholt, wie der Ausdruck true ist.

Beispiel

```
int i = 0;
while (i<3) {
   System.out.print(i + " ");
   i++;
}
```

```
Ausgabe
0 1 2
```

7.7 Do-Schleife

Syntax

```
do
   anweisung
while (boolean-ausdruck);
```

Die Do-Schleife wird solange wiederholt, wie der Ausdruck true ist.

Beispiel

```
int i = 0;
do {
   System.out.print(i + " ");
   i++;
} while (i<3);
```

```
Ausgabe
0 1 2
```

Im Gegensatz zur While-Schleife wird die Do-Schleife immer mindestens einmal durchlaufen.

7.8 Break- und Continue-Anweisung

7.8.1 Break

Das Schlüsselwort „break“ in einer beliebigen Schleife sorgt für einen sofortigen Abbruch der Schleife.

```
for (int i=0; i<20; i++) {
   System.out.print(i + " ");
   if (i>=5) break;
}
System.out.print("Schleifenende");
```

```
Ausgabe
0 1 2 3 4 5 Schleifenende
```

Dies gilt nur für die innerste Schleife, falls Sie mehrere Schleifen ineinander verschachtelt haben.

```
for (int i=0; i<2; i++) {
   for (int j=0; j<20; j++) {
      System.out.print(j + " ");
      if (j>=5) break;
   }
   System.out.print("Ende-von-j-Schleife\n");
}
System.out.print("Schleifenende");
```

```
Ausgabe
0 1 2 3 4 5 Ende-von-j-Schleife
0 1 2 3 4 5 Ende-von-j-Schleife
Schleifenende
```

7.8.2 Continue

Das Schlüsselwort „continue“ in einer Schleife sorgt für einen sofortigen Sprung an das Schleifen-Ende. Daher wird bei der Do-Schleife die Test-Bedingung noch mit ausgeführt.

Achtung – continue wirkt bei For-Schleifen anders als bei While- und Do-Schleifen, da sich bei diesen Schleifen das Springen zum Schleifenende unterscheidet:

- Bei einer For-Schleife wird die Update-Ausdruck ausgeführt, und dann der Test-Ausdruck ausgewertet – mit einem möglichen Abbruch der Schleife.
- Bei einer While- oder einer Do-Schleife wird nur der Test-Ausdruck ausgewertet – mit einem möglichen Abbruch der Schleife. Wird eine implizite Update-Anweisung übersprungen, so wird diese natürlich nicht ausgeführt. Dies kann zu einer Endlos-Schleife führen.

```
System.out.println("For");
for (int i = 0; i < 7; i++) {
   System.out.print(i + " ");
   if (i == 3) {
      System.out.print(" ** ");
      i += 2;
      continue;
   }
   System.out.print(i + " - ");
}

System.out.println("\nWhile");
int i = 0;
while (i < 7) {
   System.out.print(i + " ");
   if (i == 3) {
      System.out.print(" ** ");
      i += 2;
      continue;
   }
   System.out.print(i + " - ");
   i++;
}

System.out.println("\nDo");
i = 0;
do {
   System.out.print(i + " ");
   if (i == 3) {
      System.out.print(" ** ");
      i += 2;
      continue;
   }
   System.out.print(i + " - ");
   i++;
} while (i < 7);
```

```
Ausgabe
For
0 0 - 1 1 - 2 2 - 3  ** 6 6 -
While
0 0 - 1 1 - 2 2 - 3  ** 5 5 - 6 6 -
Do
0 0 - 1 1 - 2 2 - 3  ** 5 5 - 6 6 -
```

Hinweis – auch „continue“ wirkt nur für die innerste Schleife. Wollen Sie zum Schleifenkopf einer äußeren Schleife springen, so müssen Sie eine gelabelte Sprung-Anweisung nutzen – siehe nächstes Kapitel.

7.9 Gelabelte Sprung-Anweisungen

Gelabelte Sprung-Anweisungen ermöglichen ein break oder continue über mehrere Schleifen-Ebenen hinweg.

Syntax

```
break label;
continue label;
```

Dazu muss eine Schleife mit einem Label versehen werden – dies geschieht mit dem Label-Namen und einem folgenden Doppelpunkt. Wird jetzt innerhalb einer solchen Schleife eine gelabelte Sprunganweisung ausgeführt, so bezieht sie sich auf die Schleife mit dem passenden Label, auch wenn diese nicht die innerste ist.

Beispiel

```
outerLoop: for (int i=0; i<5; i++) {
   for (int j=0; j<3; j++) {
      System.out.print(i + " " + j + " / ");
      if (i+j > 3) break outerLoop;
   }
}
```

```
Ausgabe
0 0 / 0 1 / 0 2 / 1 0 / 1 1 / 1 2 / 2 0 / 2 1 / 2 2 /
```

Hinweis – eine gelabelte Sprunganweisung darf natürlich nur innerhalb einer Schleife mit einem passenden Label stehen. Man kann mit ihnen also nicht in eine ganz andere Schleife springen.

