PEARSON Education

Markt+Technik

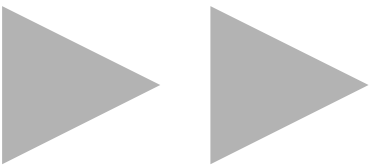
▶▶ Schnell**übersicht**

Java 5

Die praktische Referenz

DIRK LOUIS PETER MÜLLER

Markt+Technik



3 Ausdrücke und Anweisungen

Das Pendant zur Datenverwaltung ist die Datenverarbeitung. Die technische Grundlage der Datenverarbeitung sind die Operatoren. Java kennt Operatoren für

- Zuweisungen
- arithmetische Operationen
- Vergleiche
- logische Verknüpfungen
- Manipulationen auf Bitebene
- Typumwandlung und -identifizierung
- sowie verschiedene andere Aufgaben

3.1 Vom Ausdruck zur Methode

Operatoren sind spezielle Symbole der Sprache, die auf Datenwerte (die so genannten Operanden) angewendet werden können. Dabei ist zu beachten, dass

- nur Werte elementarer Datentypen, Referenzen und `null` als Operanden in Frage kommen (wobei diese Werte durch Literale, Variablen oder Rückgabewerte von Methoden repräsentiert sein können).
- Operatoren typspezifisch sind, d.h. nur Operanden bestimmter Datentypen werden akzeptiert.

3.1.1 Ausdrücke

Die Kombination aus Operatoren und Operanden heißt **Ausdruck**.

```
zahl1 + zahl2
x + 3 * y - 17
j < n
```

Ausdrücke werden zur Laufzeit berechnet und liefern dann einen Ergebniswert zurück. Arithmetische Operatoren liefern beispielsweise einen numerischen Wert, die Vergleichsoperatoren liefern einen booleschen Wahrheitswert.

```
zahl1 + zahl2          // 12, wenn zahl1 gleich 7 und
                       //     zahl2 gleich 5
x + 3 * y - 17         // 0, wenn x = 2 und y gleich 5
j < n                  // true, wenn j kleiner n
```

Der Ergebniswert eines Ausdrucks kann als Argument an eine Methode übergeben:

```
obj.skalieren(3)
obj.skalieren(eingabe * 0.1)
```

oder in einer Anweisung (siehe unten) weiterverarbeitet werden:

```
reduzierterPreis = reduzierterPreis * 0.9;

if (j < n)
  n = 0;  // wird nur ausgeführt, wenn j kleiner als n ist
```

Anmerkungen

- Ausdrücke sind hierarchisch aufgebaut. Der einfachste Ausdruck besteht aus einem einfachen Datenwert, also beispielsweise aus einem Literal oder einer Variablen: `3` oder `zahl`. Die nächste Stufe ist die Anwendung eines Operatoren auf seine Operanden: `3 + zahl`. Schließlich können in einem Ausdruck mehrere Operatoren mit ihren Operanden kombiniert werden: `3 + zahl - 3 * zahl`.
- Tauchen in einem Ausdruck mehrere Operatoren auf, entscheiden Priorität, Assoziativität und Klammerung (siehe 3.2, Abschnitt »Allgemeines«) der Operatoren darüber, welche Operationen zuerst berechnet werden. Deren Ergebniswerte dienen dann den anderen Operatoren als Operanden.

3.1.2 Anweisungen

Ausdrücke stehen in der Regel nicht allein, sondern sind Teil einer Anweisung, beispielsweise

- einer Zuweisung (siehe 3.2)
  ```
  j = i + n;
  ```
- einer Kontrollanweisung (siehe 3.3)
  ```
  if (x < 100) {
     x = 100;
  }
  ```

- eines Methodenaufrufs (siehe 4.3)

```
eineMethode(x - y);
```

Einzelne Anweisungen können mithilfe der geschweiften Klammern zu einem Anweisungsblock zusammengefasst werden:

```
{
   ersteAnweisung;
   zweiteAnweisung;
   dritteAnweisung;
}
```


Anweisungen enden mit einem Semikolon. Die Ausnahme von dieser Regel bilden die Kontrollanweisungen, die auch mit einem Anweisungsblock enden können.

Anweisungen können im Quelltext an drei Positionen auftauchen:

- als Initialisierungen für Felder (siehe 4.1, Abschnitt »Initialisierungscode«)
- als statischer Initialisierungscode einer Klasse (siehe 4.1, Abschnitt »Initialisierungscode«)
- als Anweisungsblock einer Methodendefinition (siehe 4.3)

Letzter Fall kommt am häufigsten vor und ist mit Abstand am bedeutendsten.

Fazit: Java-Programme sind in Klassen organisiert. Klassen wiederum bestehen aus Feldern und Methoden, von denen Letztere den eigentlichen operativen Code enthalten.

Anmerkung

- Ein einzelnes Semikolon wird ebenfalls als Anweisung – als leere Anweisung – gedeutet, wenn das Semikolon an einer Stelle steht, wo eine Anweisung erlaubt ist. Wozu aber soll eine leere Anwendung, die nichts tut, gut sein? Der Grund ist, dass es in Java Konstruktionen gibt, in denen Anweisungen zwingend angegeben werden müssen, damit die Konstruktion syntaktisch korrekt ist. Es gibt aber Fälle, in denen man an den betreffenden Stellen eigentlich gar keine Anweisungen ausführen lassen möchte (siehe for-Schleife in 3.3). Für diese Fälle ist die leere Anweisung gedacht.

3.2 Die Operatoren

Dieses Kapitel beginnt mit der Erläuterung verschiedener Konzepte wie Priorität und Klammerung, die für die Programmierung mit Operatoren von Bedeutung sind. Auf eine tabellarische Übersicht folgen dann die Einzeldarstellungen der Operatoren, geordnet nach Funktionalität.

3.2.1 Allgemeines

Die Operatoren sind charakterisiert durch:

- **Typspezifität**. Die Operation, die ein Operator ausführt, ist immer typspezifisch. Das heißt:
  Ein Operator akzeptiert nur Operanden passenden Typs:
  ```
  /* z1, z2 seien vom Typ int,
     str1 und str2 vom Typ String */
  z1 - z2       // korrekt
  str1 - str2   // falsch
  ```
  Ist ein Operator für mehrere Typen definiert, entscheidet der Typ der Operanden über die auszuführende Operation:
  ```
  /* i1, i2 seien vom Typ int,
     d1 und d2 vom Typ double
     str1 und str2 vom Typ String */
  i1 + i2      // Integer-Addition
  d1 + d2      // Gleitkomma-Addition
  str1 + str2  // String-Konkatenation
  ```
- **Priorität**. Jeder Operatoren gehört einer bestimmten Prioritätsstufe an. Sind in einem Ausdruck Operatoren verschiedener Prioritätsstufen kombiniert, werden die Operationen höherer Priorität zuerst ausgeführt (siehe unten).
- **Assoziativität**. Die Assoziativität entscheidet darüber, in welcher Reihenfolge Operatoren gleicher Prioritätsstufe ausgewertet werden. Am weitesten verbreitet ist die Auswertung von links nach rechts (siehe unten).
- **Anzahl Operanden**. Operatoren können nach der Anzahl ihrer Operanden in
  - unäre (beispielsweise Negation: `!op`),

- binäre (beispielsweise Addition: `op1 + op2`)

und

- ternäre (in Java ausschließlich der Bedingungsoperator: `op1 ? op2 : op3`)

Operatoren unterschieden werden.

Binäre numerische Promotionen

Die meisten Java-Operatoren sind binär, so z.B. auch die arithmetischen Operatoren für Addition, Subtraktion und so weiter.

```
int   a = 120000;
short b = 3;
int   c;
...
c = a + b;      // Addition
```

Auf CPU-Ebene können die binären Operationen aber nur dann durchgeführt werden, wenn die beiden Operanden exakt den gleichen Typ haben. Obige Operation, bei der ein `short`-Wert (`b`) und ein `int`-Wert (`a`) addiert werden, ist an sich also gar nicht zulässig. Die meisten Sprachen, darunter auch Java, gleichen in solchen Fällen die Typen der beiden Operanden gemäß den binären numerischen Promotionen an.

Die binären numerischen Promotionen legen fest, wie der Compiler die Typen der Operanden ohne größere Informationsverluste angleichen kann. Eine mögliche, wenn auch nicht sehr effiziente Vorschrift zur Durchführung einer numerischen Promotion könnte lauten:

1. Führe eine integrale Promotion durch, d.h., wandele alle `char`-, `byte`- und `short`-Operanden in `int`-Operanden um. Hierbei entsteht kein Informationsverlust.
2. Bestimme, welcher Operand gemäß der Folge `int` > `long` > `float` > `double` den »größten« Typ besitzt, und wandele den anderen Operanden in diesen Typ um. Hierbei kann es zu Informationsverlusten kommen, wenn Operanden vom Typ `int` oder `long` in `float` oder `double` umgewandelt werden.

Anmerkung

- Die integrale Promotion wird auch bei `char`-, `byte`- und `short`-Operanden unärer Operatoren vorgenommen.

Reihenfolge der Operatorenauswertung

In Ausdrücken, die aus mehreren Operatoren bestehen, stellt sich die Frage, wie diese Ausdrücke berechnet werden. Dabei spielen die Priorität und die Assoziativität der Operatoren, die Reihenfolge der Operandenauswertung sowie die Klammerung eine entscheidende Rolle.

- Sind in einem Ausdruck Operatoren unterschiedlicher **Priorität** kombiniert, werden die Operationen höherer Priorität zuerst ausgeführt.
  Beispielsweise hat die Multiplikation – wie in der Mathematik – eine höhere Priorität als die Addition:
  ```
  int y, x = 2;
  y = 3 + x * 4;  // ergibt y gleich 11
  ```
- Die **Assoziativität** entscheidet darüber, in welcher Reihenfolge Operatoren gleicher Prioritätsstufe ausgewertet werden. Am häufigsten ist die Auswertung von links nach rechts.
- **Klammerung**. Durch das Setzen von Klammern kann der Programmierer die Auswertungsreihenfolge selbst steuern (oder einfach die natürliche Auswertung eines Ausdrucks verdeutlichen).
  ```
  int y, x = 2;
  y = 3 + (x * 4);  // ergibt y gleich 11
  y = (3 + x) * 4;  // ergibt y gleich 20
  ```
- **Operandenauswertung**. In komplexen Ausdrücken, bei denen Teilausdrücke als Operanden auftauchen, gilt:
  - Bevor eine Operation ausgeführt wird, werden die Operanden des Operators berechnet.
  - Der rechte Operand wird erst berechnet, nachdem der linke Operand fertig berechnet ist.
    ```
    int y, x = 3;
    y = x * ++x;  // ergibt 12, da zuerst der
                  // linke Operand ausgeführt
                  // wird -> 3 * 4,
                  // statt 4 * 4
    ```

Tabelle 3.1 listet die Operatoren nach ihrer Priorität geordnet auf. 1 ist die höchste Priorität:

Prior.	Operatoren	Bedeutung	Assoz.
1	`()`	Methodenaufruf	L-R
	`[]`	Array-Index	
	`.`	Elementzugriff	
2	`++`	Inkrement	R-L
	`--`	Dekrement	
	`+, -`	Vorzeichen	
	`~`	Bitkomplement	
	`!`	logische Negation	
	`()`	Typumwandlung	
3	`*`	Multiplikation	L-R
	`/`	Division	
	`%`	Modulo (Rest der Division)	
4	`+`	Addition	L-R
	`-`	Subtraktion	
	`+`	Konkatenation (String-Verkettung)	
5	`<<`	Linksverschiebung	L-R
	`>>`	Rechtsverschiebung	
	`>>>`	Rechtsverschiebung	
6	`<, <=`	kleiner, kleiner gleich	L-R
	`>, >=`	größer, größer gleich	
	`instanceof`	Typüberprüfung eines Objekts	
7	`==`	gleich	L-R
	`!=`	ungleich	
8	`&`	bitweises UND	L-R
	`&`	logisches UND	
9	`^`	bitweises XOR	L-R
	`^`	logisches XOR	
10	`|`	bitweises ODER	L-R
	`|`	logisches ODER	
11	`&&`	logisches UND	L-R

Tabelle 3.1: Priorität und Assoziativität der Operatoren

Prior.	Operatoren	Bedeutung	Assoz.
12	`||`	logisches ODER	L-R
13	`?:`	Bedingungsoperator	R-L
14	`=`	Zuweisung	R-L
	`*=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=`	zusammengesetzte Zuweisung	
15	`,`	Komma-Operator	L-R

Tabelle 3.1: Priorität und Assoziativität der Operatoren (Forts.)

Anmerkung

- Beim Aufbau komplexerer Ausdrücke, die mehrere Operatoren und Werte enthalten, sollte man unbedingt auf mögliche unerwartete Nebeneffekte achten.

  Nebeneffekte ergeben sich immer dann, wenn eine Anweisung implizit weitere Anweisungen auslöst – beispielsweise wenn Sie in den Ausdruck auf der rechten Seite einer Anweisung einen Methodenaufruf einbauen oder einen Operanden inkrementieren oder dekrementieren.

  ```
  y = Math.sin(2.4) * 3;
  y = 3 * ++x;
  ```

  Meist sind solche Nebeneffekte erwünscht und werden gezielt eingesetzt. Ungenaue Kenntnisse der Arbeitsweise des Compilers können aber ebenso wie »besonders geschickt« ausgetüftelte Ausdrücke schnell dazu führen, dass sich Nebeneffekte einstellen, die vom Programmierer weder gewünscht noch vorhergesehen wurden.

3.2.2 Vorzeichen

Mit den unären Operatoren + und - können Sie numerischen Typen ein Vorzeichen zuweisen. Werte ohne Vorzeichen sind automatisch positiv.

Operator	Operanden	Beispiel (für int v = 3;)
+	numerisch	Positive Zahl
		`+5; // redundant für 5`
-	numerisch	Negative Zahl, Negation
		`-5;` `-v;  // gleich -3`

Anmerkung

- Zu den elementaren numerischen Datentypen gehören die Integer-Typen (`byte`, `short`, `int`, `long` und `char`) sowie die Gleitkommatypen (`float` und `double`).

3.2.3 Arithmetische Operatoren

Mit den arithmetischen Operatoren können Sie, wie es der Name vermuten lässt, einfache arithmetische Operationen durchführen.

Operator	Operanden	Beispiel
+	numerisch	Addition
		`4 + 3;  // 7`
-	numerisch	Subtraktion
		`4 - 3;  // 1`
*	numerisch	Multiplikation
		`4 * 3;  // 12`
/	Integer	(abgerundete) Ganzzahldivision
		`7 / 4;  // 1`
/	Gleitkomma	Division
		`7.0 / 4.0;  // 1.75`
%	Integer	Rest einer Ganzzahldivision
		`7 % 4;  // 3`
%	Gleitkomma	Rest einer Ganzzahldivision
		`7.0 % 4.0;  // 3`

Bei der Auswertung gilt: Punktrechnung geht vor Strichrechnung. Bei gleicher Priorität werden die Operatoren von links nach rechts abgearbeitet. Sie können durch Setzen von Klammern andere Prioritäten vergeben.

Die Division durch null führt bei Integer-Operanden zur Auslösung einer `ArithmeticException` (siehe 5.5), während für Gleitkommaoperanden als Ergebnis unendlich, repräsentiert durch die Gleitkommakonstante `NEGATIVE_INFINITY`, zurückgeliefert wird.

```
double quotient  = 13.65 / 0.0;

if( Double.isInfinite(quotient) )
   System.out.println("Unendlichkeit heisst in Java: "
                      + quotient);  // Ausgabe: Infinity
```

Beachten Sie, dass Sie keine direkten Vergleiche gegen `Double.NEGATIVE_INFINITY` (oder `Double.POSITIVE_INFINITY`) vornehmen können, sondern die Methode `isInfinite()` aufrufen müssen.

Anmerkungen

- Zu den elementaren numerischen Datentypen gehören die Integer-Typen (`byte`, `short`, `int`, `long` und `char`) sowie die Gleitkommatypen (`float` und `double`).
- Verwechseln Sie den Additionsoperator nicht mit dem Konkatenationsoperator für Strings (siehe unten). Beide haben zwar das gleiche Symbol (+), repräsentieren aber gänzlich unterschiedliche Operationen.

3.2.4 Zuweisungen

Mit Zuweisungsoperatoren können Werte zugewiesen und verändert werden. Neben dem einfachen Zuweisungsoperator gibt es auch die so genannten zusammengesetzten Zuweisungsoperatoren, mit denen in einer Anweisung sowohl eine Berechnung als auch eine Zuweisung durchgeführt werden kann.

Operator	Operanden	Beispiel
=	alle	Einfache Zuweisung

```
var = 3;
var1 = var2 + 3;
var = var + 3;      // Erhöhung um 3
obj1 = obj2;        // Referenz auf
                    // Objekt kopieren
```

Operator	Operanden	Beispiel
=<op>	numerisch	Kombinierte Zuweisung, wobei <op> für einen arithmetischen oder bitweisen Operatoren stehen kann (+, -, *, /, %, <<, >>, >>>, &, ^, |)

```
var += 3;           // entspricht:
                     // var = var + 3;
var1 *= var2 + 3;  // entspricht:
          // var1 = var1 * (var2 + 3);
```


L- und R-Werte

Grundsätzlich gilt, dass mit einer Zuweisung dem linken Operanden der Wert des rechten Operanden zugewiesen wird. Damit dies möglich ist, muss der linke Operand veränderbar sein, sich also auf eine Adresse im Speicher beziehen (Variable). Diese Ausdrücke werden als »L-Wert« bezeichnet. Auf der rechten Seite einer Zuweisung (R-Wert) werden dagegen auch Literale oder Methodenaufrufe akzeptiert.

3.2.5 In- und Dekrement

Java stellt zwei Operatoren zur Verfügung, mit denen der Wert einer numerischen Variablen inkrementiert (um eins erhöht) oder dekrementiert (um eins vermindert) werden kann, wobei die Operatoren vor oder nach der Variablen stehen können.

```
lwert++
++lwert
lwert--
--lwert
```

Operator	Operanden	Beispiel (für int var = 3;)
++	numerisch	Erhöhung um 1

		`++var;  // var gleich 4`
		`var++;  // entspricht var += 1;`

--	numerisch	Verminderung um 1

		`--var;  // var gleich 3`
		`var--;  // entspricht var -= 1;`

Anmerkungen

- Zu den elementaren numerischen Datentypen gehören die Integer-Typen (`byte`, `short`, `int`, `long` und `char`) sowie die Gleitkommatypen (`float` und `double`).
- **Terminologie:** Die Erhöhung eines numerischen Werts um 1 bezeichnet man als **Inkrement**, die Verminderung um 1 als **Dekrement**.

Präfix- und Postfix-Notationen

Das Besondere an den Operatoren `++` und `--` ist, dass man sie auch in Ausdrücken (also auf der rechten Seite von Zuweisungen) verwenden kann:

```
var1 = 3 * ++var2;
```

und dass man sie sowohl vor als auch hinter den Namen der Variablen setzen kann:

```
++var;
var++;
```

Zwischen beiden Formen gibt es einen wichtigen semantischen Unterschied:

- Bei der Voranstellung (Präfix-Notation) erhöht der Operator den Wert der Variablen und gibt den neuen Wert an den umliegenden Ausdruck weiter.
- Bei Nachstellung (Postfix-Notation) erhöht der Operator den Wert der Variablen, gibt aber den alten Wert an den umliegenden Ausdruck weiter.

Beispiel

```
int x = 0, y = 0;

x = ++y * 5;        // danach ist y = 1 und x = 5
x = 0;
y = 0;
x = y++ * 5;        // danach ist y = 1 und x = 0
```

Anmerkungen

- Der nachgestellte Operator funktioniert so, dass intern eine temporäre Hilfsvariable eingerichtet wird. In dieser wird der aktuelle Wert der Variablen gespeichert. Erst danach wird der Wert der Variablen um 1 erhöht. Zurückgeliefert wird aber vom Operator der alte, in der Hilfsvariablen gespeicherte Wert.
- **Tipp:** Wenn einer der Inkrement- oder Dekrement-Operatoren allein in einer Anweisung verwendet wird, ist es prinzipiell egal, ob Sie den Operator voran- oder nachstellen:

```
++var;    // Wert von var wird um 1 erhöht
var++;    // Wert von var wird um 1 erhöht
```

Der vorangestellte Operator wird allerdings schneller ausgeführt. (Oft kann der Compiler unnötige Postfix-Aufrufe selbständig optimieren, Compiler-Option `-O`.)

3.2.6 String-Konkatenation

Mit die wichtigste Operation zur Bearbeitung von Strings ist das Anhängen eines Strings an einen anderen. Im Programmierjargon bezeichnet man dies als **Konkatenation**.

Operator	Operanden	Beispiel
+	Strings	Aneinanderreihung

		`String gruss = "Hallo";`
		`String name  = "Programmierer";`
		`String ausgabe;`
		`ausgabe = gruss + " " + name + "!";`

Steht das +-Zeichen zwischen zwei Operanden, von denen einer dem String-Typ, der andere einem numerischen Typ angehört, wandelt der Compiler den numerischen Wert automatisch in einen String um und interpretiert das + als Konkatenationsoperator.

Beispiel

```
int zahl = 144;
int wurzel;

wurzel = Math.sqrt(zahl);
System.out.println("wurzel : " + wurzel);
```

Anmerkung

- Verwechseln Sie den Konkatenationsoperator nicht mit dem Additionsoperator für numerische Datentypen. Beide haben zwar das gleiche Symbol +, repräsentieren aber gänzlich unterschiedliche Operationen.

3.2.7 Relationale Operatoren

Mit den vergleichenden oder relationalen Operatoren wird ein Vergleich zwischen zwei Operanden, die auch Ausdrücke sein können, durchgeführt. Das Ergebnis dieser Operation ist entweder wahr (`true`) oder falsch (`false`).

Operator	Operanden	Beispiel (für int v = 3;)
==	alle	Gleichheit

		`v == 5 // false`
		`boolean option = false;`
		`option == false // true`
		`Demo obj1 = new Demo();`
		`Demo obj2 = new Demo();`
		`obj1 == null // false`
		`obj1 == obj2 // false`

!=	alle	Ungleichheit

		`v != 5 // true`
		`boolean option = false;`
		`option != false // false`
		`Demo obj1 = new Demo();`
		`Demo obj2 = new Demo();`
		`obj1 != null // true`
		`obj1 != obj2 // true`

<	numerisch	Kleiner als
		`v < 3 // false`
<=	numerisch	Kleiner als oder gleich
		`v <= 3 // true`
>	numerisch	Größer als
		`v > 1 // true`
>=	numerisch	Größer als oder gleich
		`v >= 1 // true`

Ausdrücke, die vergleichende Operatoren enthalten, werden meist als Kontrollbedingung in Schleifen oder Verzweigungen (siehe 3.3) eingesetzt.

```
if (alter < 16) {
   System.out.println("Dieser Film ist erst ab " +
                      "16 Jahren freigegeben!:");
```

Das Ergebnis eines Vergleichs ist ein boolescher Wahrheitswert (`true`, `false`).

Anmerkungen

- Zu den elementaren numerischen Datentypen gehören die Integer-Typen (`byte`, `short`, `int`, `long` und `char`) sowie die Gleitkommatypen (`float` und `double`).
- Beachten Sie, dass der Gleichheitsoperator == aus zwei Gleichheitszeichen besteht und der Operator für die einfache Zuweisung = aus einem Gleichheitszeichen.
- String-Literale und »interned« String-Objekte (siehe 6.1) können mit == und != verglichen werden. Ansonsten gibt es als Ersatz für die relationalen Operatoren <, <=, > und >= entsprechende Methoden der Klasse `String`: `equals()`, `equalsIgnoreCase()`, `contentEquals()`, `compareTo()` und `compareToIgnoreCase()` (siehe Referenz zur Klasse `String`).

Relationale Operatoren und Referenztypen

Beachten Sie, dass die relationalen Operatoren ausschließlich Werte vergleichen. Dies gilt nicht nur für die numerischen Vergleichsoperatoren (>, < ...), sondern auch für == und !=. Wenn Sie diese auf Referenzen anwenden, prüfen Sie, ob die verglichenen Referenzen identisch sind (auf dasselbe Objekt verweisen).

Wenn Sie lediglich feststellen möchten, ob zwei Referenzen (bzw. Objekte) demselben Klassentyp angehören, verwenden Sie dazu den `instanceof`-Operator (siehe unten).

Anmerkung

- Der ==-Operator entspricht der von `Object` vererbten `equals()`-Methode (siehe 5.1, Abschnitt »Die Klasse Object«). Beachten Sie

aber, dass `equals()` in abgeleiteten Klassen überladen und eine andere Semantik zeigen kann.

3.2.8 Logische Operatoren

Java kennt sechs logische Operatoren zur Durchführung logischer UND-, ODER- und NICHT-Verknüpfungen. Bei den logischen UND- und ODER-Operatoren wird der Wahrheitswert der Operanden verknüpft, der logische NICHT-Operator konvertiert den logischen Wert seines Operanden ins Gegenteil. Das Ergebnis einer logischen Verknüpfung ist wiederum ein boolescher Wahrheitswert.

Operator	Operanden	Beispiel
&&, &	boolean	Logisches Und `if (i > 1 && i < 10)`
||, |	boolean	Logisches Oder `if (i < 1 || i > 10)`
^	boolean	Logisches XOR (eXklusives Oder) `if (i < 1 ^ i > 10)`
!	boolean	Logisches Nicht `if ! (i == 0)`

&&, & – logisches UND

Mit dem Operator `&&` wird eine logische UND-Verknüpfung durchgeführt. Das Ergebnis der Verknüpfung hat nur dann den Wert `true`, wenn beide Operanden den Wert `true` besitzen.
Die folgende Wahrheitstabelle 3.2 zeigt Ihnen die möglichen Kombinationen.

1. Operand	2. Operand	Ergebnis
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	falsch
falsch	falsch	falsch

Tabelle 3.2: UND-Verknüpfung

Mithilfe der logischen UND-Verknüpfung und einer `if`-Bedingung können Sie beispielsweise überprüfen, ob der Wert einer Variablen zwischen 10 und 100 liegt:

```
if( (x > 10) && (x < 100) )
```

Der `&&`-Operator wird von links nach rechts nur so weit ausgewertet, bis das Ergebnis feststeht. Wenn der linke Operand bereits den Wert `false` liefert, wird die Auswertung abgebrochen.

Soll der zweite Operand stets ausgewertet werden – etwa weil er eine wichtige Zuweisung (Inkrement, Dekrement) oder einen auszuführenden Methodenaufruf enthält –, kann die Auswertung mit dem logischen Operator `&` erzwungen werden:

```
if( (x > 10) & (obj.leseWertEin() < 100) )
```

Anmerkung

- Grundsätzlich ist vom Einsatz des `&`-Operators sowie vom Einbau von Nebeneffekten in Operanden abzuraten.

||, | – logisches ODER

Der logische ODER-Operator verknüpft seine Operanden so, dass das Ergebnis der Verknüpfung den Wert `true` hat, wenn mindestens einer der Operanden den Wert `true` hat.

Die folgende Wahrheitstabelle 3.3 zeigt Ihnen die möglichen Kombinationen.

1. Operand	2. Operand	Ergebnis
wahr	wahr	wahr
wahr	falsch	wahr
falsch	wahr	wahr
falsch	falsch	falsch

Tabelle 3.3: ODER-Verknüpfung

Mithilfe der logischen ODER-Verknüpfung und einer `if`-Bedingung können Sie beispielsweise überprüfen, ob der Wert einer Variablen kleiner als 10 oder größer als 100 ist:

```
if( (x < 10) || (x > 100) )
```

Wie der `&&`-Operator wird auch der `||`-Operator nur so weit ausgewertet, bis das Ergebnis feststeht. Wenn Sie möchten, dass immer

beide Operanden ausgewertet werden, verwenden Sie den Operator | (Achtung, Nebeneffekte!, siehe oben).

^ – logisches XOR

Der logische XOR-Operator verknüpft seine Operanden so, dass das Ergebnis der Verknüpfung den Wert `true` hat, wenn genau einer der Operanden den Wert `true` hat.
Die folgende Wahrheitstabelle 3.4 zeigt Ihnen die möglichen Kombinationen.

1. Operand	2. Operand	Ergebnis
wahr	wahr	falsch
wahr	falsch	wahr
falsch	wahr	wahr
falsch	falsch	falsch

Tabelle 3.4: XOR-Verknüpfung

! – logisches NICHT

Der logische NICHT-Operator verkehrt den Wahrheitswert seines logischen Operanden ins Gegenteil.

```
boolean sechsRichtige = false;

if (!sechsRichtige) {
  System.out.println("Schade: wieder kein Hauptgewinn!");
}
```

3.2.9 Bitweise Operatoren

Im Rechner werden sämtliche Daten binär als Bitfolgen kodiert. Mit den Bitoperatoren können diese Bits direkt manipuliert werden – jedoch nur für Integer-Typen.

Operator	Operanden	Beispiel
&	Integer	bitweise UND-Verknüpfung `0xFF & 223`
|	Integer	bitweise ODER-Verknüpfung `0xFF | 0x20`

Operator	Operanden	Beispiel
^	Integer	bitweises XOR (exklusives ODER) `j ^0x0`
~	Integer	bitweises Komplement `~j`
<<	Integer	Linksverschiebung `i << 3`
>>	Integer	Rechtsverschiebung mit Erhalt des Vorzeichens `i >> 3`
>>>	Integer	Rechtsverschiebung ohne Erhalt des Vorzeichens `i >>> 3`

Anmerkung

- Die Operanden unterliegen der integralen Promotion.

Bitweises UND &

Der &-Operator geht die Bits der beiden Operanden nacheinander durch. Immer wenn für eine Bitposition in beiden Operanden das jeweilige Bit auf 1 gesetzt ist, wird auch das entsprechende Bit im Ergebniswert auf 1 gesetzt. Andernfalls wird das Ergebnisbit auf 0 gesetzt.

```
     0110 0001
  &  1101 1111
 --------------
     0100 0001
```

Dieser Operator kann beispielsweise eingesetzt werden, um Bits in einer Integer-Variablen gezielt zu löschen.

Beispiel

```
int i = 0xFF;     // i = 255
i &= 0xF0;        // setzt alle Bits außer den
                  // Bits 5 bis 8 auf 0 -> i = 240
```

Bitweises ODER |

Der |-Operator geht die Bits der beiden Operanden nacheinander durch. Immer wenn für eine Bitposition in einem der Operanden das

jeweilige Bit auf 1 gesetzt ist, wird auch das entsprechende Bit im Ergebniswert auf 1 gesetzt. Andernfalls wird das Ergebnisbit auf 0 gesetzt.

```
    0110 0001
  | 1101 1111
 --------------
    1111 1111
```

Dieser Operator kann eingesetzt werden, um gezielt zusätzliche Bits in einer Integer-Variablen zu setzen.

Beispiel

```
int i = 0xFF;     // i = 255
i &= 0xF0;        // setzt alle Bits außer den Bits 5 bis 8
                  // auf 0 -> i = 240
i |= 0xF;         // setzt die ersten 4 Bits auf 1
                  // i wieder gleich 255
```

Bitweises XOR ^

Der Operator für die bitweise exklusive ODER-Verknüpfung vergleicht die Bitmuster seiner Operanden und setzt das entsprechende Bit im Ergebnis, wenn eines der Bits in den Operanden, aber nicht beide, gesetzt sind.

```
    0110 0001
  ^ 1101 1111
 --------------
    1011 1110
```

Dieser Operator kann eingesetzt werden, um Bits umzuschalten (gesetzte Bits werden gelöscht und umgekehrt).

```
int i = 0x8;      // i = 8
i ^= 0xB;         // 0xB = 11 -> löscht 4. Bit
                  //             setzt Bit 1 und 2
```

Bitweises Komplement ~

Der bitweise Komplement-Operator ~ kippt alle Bits seines Operanden. Durch Invertierung werden im Ergebnis alle Bits, die im Operanden gesetzt waren, gelöscht, und alle Bits, die gelöscht waren, gesetzt. Die Zahl 4 besitzt folgendes (auf 0 Bit verkürztes) Bitmuster:

```
0 0 0 0    0 1 0 0
```
Durch die Operation ~4 ergibt sich folgendes Muster:
```
1 1 1 1    1 0 1 1
```

Linksverschiebung <<

Der Operator für die Linksverschiebung kopiert die Bits des ersten Operanden in den Ergebniswert – allerdings um so viele Positionen nach links gerückt, wie der zweite Operand angibt. Die rechts entstehenden Leerstellen werden mit o aufgefüllt:

`int i = 3;` `i = i << 2;`	int-Wert	Bitmuster (nur 8 Bit)
vor Verschiebung	3	0000 0011
nach Verschiebung	12	0000 1100

Tabelle 3.5: Linksverschiebung

Die Anzahl der Positionen, um die verschoben werden kann, ist begrenzt. Ist der linke Operand vom Typ `int`, wertet der Compiler nur die niederwertigsten fünf Bits aus, wodurch Verschiebungen um 0 bis 31 Positionen möglich sind. Ist der linke Operand vom Typ `long`, werden die niederwertigsten sechs Bits des rechten Operanden ausgewertet, wodurch Verschiebungen um maximal 63 Positionen möglich sind. Negative Verschiebungen sind nicht möglich, da das Vorzeichenbit ja gar nicht ausgewertet wird (`i = i << 2;` ist also gleichbedeutend mit `i = i << -2;`).

Beispiel: Multiplikation um Potenzen von 2

Jede Linksverschiebung um eine Position entspricht einer Multiplikation mit 2.

Eine Linksverschiebung um n Stellen entspricht daher einer Multiplikation mit 2^n.

```
int i = 10;
i <<= 3;      // Multiplikation mit 8 (2^3) -> i = 80;
```

Rechtsverschiebung >>

Der Operator für die Rechtsverschiebung kopiert die Bits des ersten Operanden in den Ergebniswert – allerdings um so viele Positionen nach rechts gerückt, wie der zweite Operand angibt. Die links entstehenden Leerstellen werden mit Nullen oder Einsen aufgefüllt – je nachdem, ob der linke Operand positiv (führende Null) oder negativ (führende Eins) ist. So bleibt das Vorzeichen erhalten.

int i = 12; i = i >> 2;	int-Wert	Bitmuster (nur 8 Bit)
vor Verschiebung	12	0000 1100
nach Verschiebung	3	0000 0011

Tabelle 3.6: Rechtsverschiebung

Die Anzahl der Positionen, um die verschoben werden kann, ist wie bei der Linksverschiebung auf 31 bzw. 63 Positionen begrenzt.

Beispiel: Division um Potenzen von 2

Jede Rechtsverschiebung um eine Position entspricht einer Division durch 2.

Eine Rechtsverschiebung um n Stellen entspricht daher einer Division durch 2^n.

```
int i = 72;
i >>= 3;      // Division durch 8 (2^3) -> i = 9;
```

Rechtsverschiebung ohne Erhalt des Vorzeichens: >>>

Der >>>-Operator für die Rechtsverschiebung ohne Erhalt des Vorzeichens kopiert die Bits des ersten Operanden in den Ergebniswert – allerdings um so viele Positionen nach rechts gerückt, wie der zweite Operand angibt. Die links entstehenden Leerstellen werden mit Nullen aufgefüllt.

3.2.10 Typumwandlung (Cast)

Gelegentlich ist es wünschenswert, einen Wert vom Typ A in einen Typ vom Wert B umzuwandeln. Sofern eine solche Typumwandlung durchführbar ist (siehe 2.8), kann sie durch den Cast-Operator `()` explizit herbeigeführt werden.

Operator	Operanden	Beispiel

`()`	Typ, Ausdruck	Typumwandlung
		`double d = 3.5;`
		`int i = (int) d;  // i = 3`

Der Cast-Operator wird meist eingesetzt, um in einer Anweisung den Typ des Ausdrucks der rechten Seite in den Typ der Variablen der linken Seite umzuwandeln:

```
TypA eineVar = (TypA) andereVar;  // andereVar ist vom Typ
                                  // TypB und die Umwandlung von
                                  // TypB in TypA ist möglich
```

Beispiel

Bei der Programmierung mit Objekten wird der Cast-Operator meist eingesetzt, um eine Referenz, die in einen Basisklassentyp verwandelt wurde, wieder in den Typ der ursprünglichen abgeleiteten Klasse zurückzuverwandeln.

Beispiel: Typumwandlung

```
class Basis {
...
}
class Abgeleitet extends Basis {
...
}

Basis objB = new Basis();
Basis objA = new Abgeleitet();
Abgeleitet obj;

obj = (Abgeleitet) objA;  // okay
obj = (Abgeleitet) objB;  // Fehler, da objB nicht auf ein
```

```
// Abgeleitet-Objekt verweist
// wirft zur Laufzeit eine
// ClassCastException aus
```

Beachten Sie, dass die Umwandlung eines Objekts vom Typ einer seiner Basisklassen in eine abgeleitete Klasse nur dann durchgeführt werden kann, wenn das Objekt vom Typ der abgeleiteten Klasse ist. Ansonsten wird zur Laufzeit eine Exception ausgelöst.

Anmerkungen

- Narrowing-Typumwandlungen (siehe 2.8), die Sie mit dem Cast-Operator erzwingen, gehen in der Regel mit einem Informationsverlust einher. Ausnahme: die Rücknahme von in Basisklassentypen umgewandelte Referenzen.
- Ist eine gewünschte Typumwandlung nicht möglich (siehe 2.8), wird eine `ClassCastException` ausgelöst.

3.2.11 instanceof

Mit dem `instanceof`-Operator können Sie feststellen, ob ein Objekt einem bestimmten Klassentyp angehört.

Operator	Operanden	Beispiel

`instanceof`	Referenz, `null`	Bestätigt den Typ eines Objekts
		`Demo obj = new Demo();`
		`obj instanceof Demo  // true`

Der `instanceof`-Operator verhält sich so, wie der Versuch einer impliziten oder expliziten Typumwandlung, nur dass die Typumwandlung nicht durchgeführt, sondern simuliert und ihr Ergebnis als boolescher Wert zurückgeliefert wird. Dies bedeutet im Einzelnen:

- Der Operator liefert `true` zurück, wann immer eine automatische Typumwandlung vom Typ des Objekts in den angegebenen Klassentyp möglich ist. Mit anderen Worten: Der Operator liefert auch dann `true`, wenn Sie ein Objekt mit einem seiner Basisklassentypen vergleichen.

```
obj instanceof EigeneKlasse      // true
obj instanceof BasisKlasse       // true
```

- Der Operator liefert `false`, wenn nur eine explizite Typumwandlung möglich ist (Downcast).

```
obj instanceof AbgKlasse         // false
```

- Der Vergleich erzeugt eine Fehlermeldung des Compilers, wenn Vergleiche zwischen Typen durchgeführt werden, die weder durch Up- noch durch Downcast ineinander umwandelbar sind:

```
obj instanceof AndereKlasse      // Fehler
```


Anmerkung

- Der `instanceof`-Operator kommt in der Regel dann zum Einsatz, wenn der Typ eines Objekts zur Laufzeit nicht feststeht, etwa weil es reduziert auf ein Basisklassenobjekt an eine Methode übergeben wurde (siehe Polymorphie 5.2).

3.2.12 Der Bedingungsoperator

Java kennt einen einzigen ternären Operator, der Bedingungsoperator oder bedingte Bewertung genannt wird.

Operator	Operanden	Beispiel

`?:`	boolescher Ausdruck,
	Ausdruck,
	Ausdruck
		Liefert in Abhängigkeit von der Bedingung (boolescher Ausdruck) einen von zwei Ausdrücken zurück:
		`int max, a = 1, b = 2;`
		`max = (a > b) ? a : b;`

```
Bedingung ? Ausdruck1 : Ausdruck2
```

Die Auswertung geht folgendermaßen vor sich:

- Wenn die Auswertung der Bedingung `true` ergibt, wird `Ausdruck1` ausgeführt.
- Ergibt die Auswertung der Bedingung `false`, wird `Ausdruck2` ausgewertet.

Anmerkung

- Der Bedingungsoperator (`?:`) ist eine Kurzform der if-else-Anweisung (siehe 3.3):

```
if (eins > zwei)
   max = eins;
else
   max = zwei;
```

3.2.13 Operatorähnliche Symbole

Neben den eigentlichen Operatoren kennt Java noch eine Reihe von Symbolen und Schlüsselwörtern, die mit den Operatoren eng verwandt sind und ihre feste Rolle und Priorität bei der Ausdrucksauswertung haben.

Operator	Operanden	Beispiel
`new`	`new Typ(Argumente)`	Instanzbildung, siehe 2.6, Abschnitt »Klassen«, und 4.5
`()`	`methodenname(Argumente)`	Aufruf einer Methode, siehe 4.3
`[]`	`arrayvar[Ausdruck]`	Indizierung (Zugriff auf ein Array-Element), siehe 2.6, Abschnitt »Arrays«
`.`	`bez.bez`	Zugriff auf ein Klassenelement, siehe 4 Qualifizierung von Bezeichnern, siehe 1.5
`,`	`Anweisung, Anweisung`	Erlaubt den Einbau mehrerer Anweisungen, wo eigentlich nur eine einfache Anweisung möglich ist, siehe for-Schleife in 3.3

3.3 Kontrollstrukturen

Je komplexer Programme werden, desto öfters ist es erforderlich, je nach Situation und Zustand des Programms zu verschiedenen Anweisungen zu verzweigen. Zu unterscheiden sind:

- **Verzweigungen**, die anhand einer Bedingung entscheiden, ob ein nachfolgender Anweisungsblock oder, wenn mehrere alternative Blöcke zur Verfügung stehen, welcher ausgeführt werden soll.

- **Schleifen**, die es erlauben, einen Anweisungsblock mehrfach hintereinander auszuführen.
- **Exceptions** – ein spezielles Konzept zur Trennung von normalem Programmablauf und Fehlerbehandlung (siehe 5.5).

3.3.1 if-Anweisung

Die einfache `if`-Anweisung entscheidet, ob ein nachfolgender Anweisungsblock ausgeführt werden soll oder nicht.

```
if (Bedingung) {
   Code;
}
```

Bedingung	Die Bedingung, die ausgewertet wird, muss einen booleschen Wert ergeben. Hier kann beispielsweise eine Ganzzahl, eine arithmetische Operation, ein Vergleich oder der Aufruf einer Methode mit entsprechendem Rückgabetyp stehen.
Code	Der Code kann aus einer einzelnen Anweisung oder aus einem Anweisungsblock bestehen. Im Falle einer einzelnen Anweisung können die `{}`-Klammern entfallen.

Beispiel: Bedingte Ausführung

Ein Konsolenprogramm, das bei Übergabe entsprechender Befehlszeilenargumente verschiedene Ausgaben ausführt, könnte beispielsweise mithilfe einer `if`-Anweisung prüfen, ob hinter dem Programmnamen ein Argument übergeben wurde und, wenn kein Argument vorliegt, ein Menü ausgeben:

```
char befehl;

if(args.length == 0) {
   System.out.println("\n Herr, was befiehlst du?");
   menue();
   Scanner sc = new Scanner(System.in);
   befehl = sc.next().charAt(0);
}
```

Der vollständige Code dieses Programms ist im Abschnitt zur `switch`-Verzweigung abgedruckt.

Anmerkungen

- Hinter die `if`-Bedingung kommt kein Semikolon. Würden Sie eines setzen, interpretiert der Compiler dieses als leere Anweisung, die von der `if`-Bedingung kontrolliert wird.
- **Tipp:** Komplexe Bedingungen können Sie mithilfe der logischen Operatoren aufbauen.

3.3.2 Die if-else-Verzweigung

Es gibt auch Fälle, in denen alternativ (also dann, wenn die zu `if` gehörende Bedingung als Ergebnis `false` liefert, und nur dann) zu einem anderen Anweisungsblock verzweigt werden soll. Dieser wird dann mit dem Schlüsselwort `else` angehängt.

```
if (Bedingung) {
   Code1;
} else {
   Code2
}
```

Bedingung	Die Bedingung, die ausgewertet wird, muss einen booleschen Wert ergeben. Hier kann beispielsweise eine Ganzzahl, eine arithmetische Operation, ein Vergleich oder der Aufruf einer Methode mit entsprechendem Rückgabetyp stehen.
Code1 Code2	Die Codeteile können aus einer einzelnen Anweisung oder aus einem Anweisungsblock bestehen. Im Falle einer einzelnen Anweisung können die `{}`-Klammern entfallen.

Beispiel: Bedingte Ausführung

```
char befehl;

if(args.length == 0) {
   System.out.println("\n Herr, was befiehlst du?");
   menue();
   Scanner sc = new Scanner(System.in);
   befehl = sc.next().charAt(0);
} else {
   befehl = args[0].charAt(0);
}
```

Der vollständige Code dieses Programms ist im Abschnitt zur switch-Verzweigung abgedruckt.

Verschachtelung

Selbstverständlich ist es auch möglich, if-Verzweigungen zu verschachteln, d.h., im if- oder else-Teil weitere if-Verzweigungen einzubauen.

Eine spezielle Form der Verschachtelung sind die else-if-Ketten, bei denen sich an jeden else-Teil weitere if-else-Verzweigung anschließt.

```
if (Bedingung1) {
   Code1;
} else if (Bedingung2) {
   Code2;
} else if (Bedingung3) {
   Code3;
} else {
   Code4;
}
```

Auswertung der else-if-Ketten:

- Die Bedingungen werden in der Reihenfolge ausgewertet, in der sie im Programmcode stehen.
- Wenn eine der Bedingungen als Ergebnis true liefert, wird der zugehörende Anweisungsteil ausgeführt und damit die Abarbeitung der Kette beendet.
- Die zum letzten else gehörenden Anweisungen werden ausgeführt, wenn keine der vorher überprüften Bedingungen das Ergebnis true liefert.
- Das letzte else ist optional, kann also entfallen, wenn keine Standardaktion ausgeführt werden soll.

Anmerkungen

- **Terminologie:** Beim Verschachteln von if-Verzweigungen stellt sich gelegentlich die Frage, zu welcher if-Bedingung ein gegebener else-Teil gehört (**Dangling-else**-Problem). Der else-Teil gehört immer zu dem letzten vorangehenden if-Teil, der noch über keinen else-Teil verfügt.

- Die Zuordnung von `if` und `else` wird nicht durch die Einrückung bestimmt.
- Einfache `if-else`-Verzweigungen können auch mithilfe des Bedingungsoperators `?:` ausgedrückt werden (siehe 3.2).

3.3.3 Die switch-Verzweigung

Während `if-else`-Verzweigungen anhand einer booleschen Bedingung zu einem von zwei Anweisungsblöcken verzweigen (bzw. durch Verschachtelung mehrere Bedingungen prüfen), springt die `switch`-Anweisung in Abhängigkeit vom Wert eines numerischen Ausdrucks an verschiedene Stellen in einem Anweisungsblock.

```
switch(Ausdruck) {
   case Konstante1:  Anweisungen;
   case Konstante2:  Anweisungen;
   case Konstante3:  Anweisungen;
   case Konstante4:  Anweisungen;
   default:     Anweisungen;
}
```

Ausdruck	Ein ganzzahliger Wert (auch Aufzählungstyp), der mit allen innerhalb der `switch`-Anweisung stehenden `case`-Marken verglichen wird. Hier kann auch der Aufruf einer Methode stehen, die einen numerischen Typ als Ergebnis zurückgibt.
Konstante	Konstanten, die mit dem Ausdruck verglichen werden.
Anweisungen	Alle im `switch`-Block stehenden Anweisungen bilden einen gemeinsamen Anweisungsblock. Die Ausführung dieses Anweisungsblocks beginnt mit der Anweisung, die neben der ersten `case`-Konstanten steht, deren Wert mit dem Ausdruck übereinstimmt. Von dort aus werden alle nachfolgenden Anweisungen (auch die der weiteren `case`-Marken) bis zum Ende des `switch`-Blocks oder dem Auftreten einer `break`-Anweisung (siehe unten) ausgeführt.
default	Zum optionalen `default`-Block wird verzweigt, wenn die Überprüfung des `switch`-Ausdrucks mit allen `case`-Marken keine Übereinstimmung ergibt.

Soll für jeden Wert ein spezieller Anweisungsblock ausgeführt werden, müssen alle zu einer `case`-Marke gehörenden Anweisungsfolgen mit `break` abgeschlossen werden.
Sollen mehrere Werte mit demselben Anweisungsblock verbunden werden, führt man die `case`-Marken untereinander auf und verbindet die letzte mit der Anweisungsfolge.

Beispiel: Menü für Konsolenprogramme

```
import java.util.*;
import java.text.*;

public class Menue {

   public static void menue() {
      System.out.println();
      System.out.println(" \t Begruessung         <b>");
      System.out.println(" \t Uhrzeit anzeigen    <u>");
      System.out.println(" \t Datum   anzeigen    <d>");
      System.out.println(" \t Verabschiedung      <v>");
      System.out.println();
   }

   public static void main(String[] args)  {
      char befehl;
      Date datum;
      SimpleDateFormat df;

      if(args.length == 0) {
         System.out.println("\n Herr, was befiehlst du?");
         menue();
         Scanner sc = new Scanner(System.in);
         befehl = sc.next().charAt(0);
      } else {
         befehl = args[0].charAt(0);
      }

      switch(befehl) {
         case 'b':
```

```
            case 'B': System.out.println(" Morgn, morgn");
                      break;
            case 'u':
            case 'U': datum = new Date();
                      df = new SimpleDateFormat("hh:mm:ss",
                                          Locale.GERMANY);
                      System.out.println(" Es ist "
                               + df.format(datum) + " Uhr");
                      break;
            case 'd':
            case 'D': datum = new Date();
                      df = new SimpleDateFormat("dd'. 'MMMM",
                                          Locale.GERMANY);
                      System.out.println(" Es ist der "
                                        + df.format(datum));
                      break;
            case 'v':
            case 'V': System.out.println(" Bis dann");
                      break;
            default:  System.out.println(" Unbekannter "
                                        + "Befehl");
        }
    }
}
```

Anmerkungen

- Innerhalb einer `switch`-Anweisung dürfen keine zwei Konstanten den gleichen Wert haben. Sollte dies doch der Fall sein, erzeugt der Compiler eine Fehlermeldung.
- Innerhalb einer `switch`-Anweisung dürfen keine Variablen mehrmals definiert werden.
- **Tipp:** Wenn Sie mehrere Konstanten mit demselben Code verbinden wollen, stellen Sie die Konstanten einfach übereinander und definieren den Code nach der letzten Konstanten, siehe Beispiel.

3.3.4 for-Schleife

Mit dem Schlüsselwort `for` wird eine bedingte Schleife eingeleitet. Der Schleifenausdruck steht in runden Klammern und enthält die drei Elemente Initialisierung, Bedingung und Veränderung, die durch Semikolons voneinander getrennt werden.

```
for (Initialisierung; Bedingung; Veränderung) {
   Code;
}
```

Initialisierung	Die im Initialisierungsteil stehenden Anweisungen werden ausgeführt, bevor die Schleife beginnt. Dort werden üblicherweise die Schleifen- oder Zählvariablen initialisiert. Es ist auch möglich, in dieser Anweisung mehrere Variablen zu initialisieren. Die Zuweisungen werden dann durch Kommata voneinander getrennt.
Bedingung	Der Bedingungsteil enthält einen Ausdruck, der `true` sein muss, damit der zur `for`-Schleife gehörende Anweisungsteil ausgeführt wird.
Veränderung	Die hier stehenden Anweisungen werden nach jedem Durchlauf des zu `for` gehörenden Anweisungsblocks abgearbeitet. Hier wird meist die Zählervariable erhöht oder vermindert. Es kann dort jedoch auch jede andere Anweisung aufgenommen werden. Stehen hier mehrere Anweisungen, werden sie durch Kommata voneinander getrennt.
Code	Der Code kann aus einer einzelnen Anweisung oder aus einem Anweisungsblock bestehen. Im Falle einer einzelnen Anweisung können die `{}`-Klammern entfallen.

Beispiel: for-Schleife

```
public class For {
   public static void main(String[] args)  {

      int zaehler;

      for(zaehler = 0; zaehler < 10; ++zaehler) {
         System.out.println(" zaehler = " + zaehler);
      }
   }
}
```

Die for-Schleife für Arrays und Collections

Ab Java 5 gibt es eine weitere Variante der `for`-Schleife: die so genannte »for-each«-Schleife. Mit ihr können Sie alle Elemente eines Arrays (siehe 2.6, Abschnitt »Arrays«), einer Aufzählung (siehe 2.6, Abschnitt »Aufzählungen«), einer Collection (siehe 6.3) bzw. ganz allgemein jedes Objekts, das das Interface `Iterable` implementiert, durchlaufen:

```
// gegeben sei ein Array 'zahlen' mit 10 Integer-
// Elementen.
for(int i : zahlen)  {
    System.out.println(i);
}
```

for(;;)-Schleifen

Eine Sonderform der `for`-Schleife stellt die Endlosschleife dar. Sie erhält man, wenn alle drei Elemente der Definition der `for`-Schleife wegfallen und nur die die Elemente trennenden Semikolons stehen bleiben.

Endlosschleifen dieser Art können, da keine Abbruchbedingung vorhanden ist, nur mit `break` (siehe unten) oder `return` (siehe 4.3) verlassen werden.

Anmerkung

- Schleifenvariablen, die nur in der Schleife benötigt werden, können auch direkt im Initialisierungsteil definiert werden:

```
for(int zaehl = 0; zaehl < 10; ++zaehl) {
  System.out.println("zaehler = " + zaehl);
}
```

3.3.5 Die while-Schleifen

Die `while`-Schleife besteht aus den gleichen Komponenten wie die `for`-Schleife, ihre Anordnung im Programmcode ist jedoch ein wenig anders.

```
while (Bedingung) {
   Code;
}
```

Bedingung	Die Bedingung, die ausgewertet wird, muss einen booleschen Wert ergeben. Hier kann beispielsweise eine Ganzzahl, eine arithmetische Operation, ein Vergleich oder der Aufruf einer Methode mit entsprechendem Rückgabetyp stehen.
Code	Der Code kann aus einer einzelnen Anweisung oder aus einem Anweisungsblock bestehen. Im Falle einer einzelnen Anweisung können die `{}`-Klammern entfallen.

Beispiel

```
int zaehler = 0;
while(zaehler < 10) {
   ++zaehler;
   System.out.println(" zaehler = " + zaehler);
}
```

Ausführung: Auswertung der while-Schleife

Die Bedingung der `while`-Schleife wird getestet, bevor die zur Schleife gehörende Anweisung ausgeführt wird.
Die Schleife endet, wenn die Bedingung nicht mehr den Wert `true` ergibt.
Liefert die Auswertung der Bedingung der `while`-Schleife bereits beim Eintritt den Wert `false`, werden die zur Schleife gehörenden Anweisungen nie ausgeführt.

3.3.6 Die do-while-Schleife

Neben der `while`-Schleife existiert das Konstrukt der `do-while`-Schleife. Der wichtigste Unterschied zur `while`-Schleife besteht darin, dass der Anweisungsteil der `do-while`-Schleife mindestens einmal ausgeführt wird. Die Bewertung des Ausdrucks, der die Bedingung zum Abbruch der Schleife enthält, findet immer nach dem Durchlaufen des Anweisungsteils statt.

```
do {
   Code;
} while (Bedingung);
```

Bedingung	Die Bedingung, die ausgewertet wird, muss einen booleschen Wert ergeben. Hier kann beispielsweise eine Ganzzahl, eine arithmetische Operation, ein Vergleich oder der Aufruf einer Methode mit entsprechendem Rückgabetyp stehen.
Code	Der Code kann aus einer einzelnen Anweisung oder aus einem Anweisungsblock bestehen. Im Falle einer einzelnen Anweisung können die `{}`-Klammern entfallen.

Beispiel: Benutzereingaben einlesen

Das folgende Programm fordert den Benutzer so oft wiederholt zur Eingabe einer Integer-Zahl auf, bis die eingegebene Zahl zwischen 0 und 10 liegt.

```
import java.util.Scanner;

public class DoWhile{

   public static void main(String[] args)  {

      int zahl;
      Scanner sc = new Scanner(System.in);

      do {
         System.out.print(" Geben Sie eine ganze Zahl "
                          + "< 10 ein: ");
         zahl = sc.nextInt();
      } while (zahl < 0 || zahl > 10);

      System.out.println("\n Danke!");
   }
}
```

Auswertung der do-Schleife

Die Bedingung der `do`-Schleife wird getestet, nachdem die zur Schleife gehörende Anweisung ausgeführt wurde.

Die zur `do`-Schleife gehörenden Anweisungen werden mindestens einmal abgearbeitet.

Die Schleife endet, wenn die Abbruchbedingung erfüllt ist.

3.3.7 Abbruchbefehle

Die folgenden Schlüsselwörter werden verwendet, um Anweisungsblöcke zu verlassen.

Abbruchbefehl	Beschreibung
`continue`	Mit der `continue`-Anweisung wird zum Anfang einer Schleife zurückgesprungen. Die Ausführung aller nach `continue` stehenden Anweisungen findet also nicht statt.
`break`	Mit der `break`-Anweisung wird im Gegensatz zu `continue` die gesamte Schleife und nicht nur der aktuelle Schleifendurchgang abgebrochen.
`return`	Mit dem Schlüsselwort `return` kann die Ausführung einer Methode unterbrochen und sofort an der Stelle fortgesetzt werden, an der die Methode aufgerufen wurde. Das Schlüsselwort `return` ist damit auch geeignet, um Schleifen zu verlassen.

Aus verschachtelten Blöcken springen

Eine normale `continue`- oder `break`-Anweisung beendet lediglich den aktuellen Block.

Um aus einem verschachtelten Block heraus in einen äußeren Block zu springen, müssen Sie eine Sprungmarke setzen und zu dieser springen.

Beispiel

```
      int i = 1;
ganzaussen:
      while(true) {
         if( true ) {
            for(int i = 0; i < 5; ++i) {
               ...
               if( true )
                  break ganzaussen;
            }
            System.out.println("hinter for");
         }
         System.out.println("hinter if");
     }
     System.out.println("hinter while");
```

Die `break`-Anweisung in der `for`-Schleife springt zu der Sprungmarke `ganzaussen`, sprich zur äußeren `while`-Schleife, die sofort beendet wird. Der Code, so, wie er hier steht, erzeugt daher nur eine einzige Ausgabe:

```
hinter while
```

Anmerkungen

- Die `continue`-Anweisung kann nur zu Zielen springen, die auf den Kopf einer umliegenden Schleife verweisen. Die aktuelle Iteration der Schleife wird daraufhin beendet, die Schleife mit der nächsten Iteration fortgesetzt.
- Die `break`-Anweisung kann zu beliebigen umliegenden Schleifen- oder Verzweigungsköpfen springen. Die Schleife/Verzweigung, die angesprungen wird, wird daraufhin direkt beendet und das Programm mit der Anweisung unter der Schleife/Verzweigung fortgesetzt.