smartlearn
virtual learning. real success.

MODUL 411
JAVA-REPETITION

RALPH MAURER

VERSION 1.0

Inhaltsverzeichnis

Java-Refresher

Themen

Kurzreferenz Java (Refresher)

Eine Programmiersprache oder Methoden und Konzepte der Softwareentwicklung lernt man nur mit Übung und wenn man selbst Programme in der neuen Sprache schreibt und neue Methoden und Konzepte anwendet. Je mehr, desto besser.

Input/Output (I/O)

Einfache Ausgaben

Für die ersten Schritte in einer neuen Sprache benötigt man immer auch I/O-Routinen, um einfache Ein- und Ausgaben vornehmen zu können. Glücklicherweise kann man in Java nicht nur grafikorientierte Programme schreiben, sondern auch auf die Standardein- und -ausgabe zugreifen. Damit stehen für kleine Programme einfache I/O-Routinen zur Verfügung, die wie in den meisten konventionellen Programmiersprachen verwendet werden können.

Mit Hilfe des Kommandos System.out.println können einfache Ausgaben auf den Bildschirm geschrieben werden. Nach jedem Aufruf wird eine Zeilenschaltung ausgegeben.

Mit System.out.print kann diese auch unterdrückt werden. Beide Methoden erwarten ein einziges Argument, das von beliebigem Typ sein kann. Mit Hilfe des Plus-Operators können Zeichenketten und numerische Argumente miteinander verknüpft werden, so dass man neben Text auch Zahlen ausgeben kann:

```
//  @author:       Ralph Maurer iet-gibb
//  Compilation:   javac AB411_02_Output.java
//  Execution:     java -jar AB411_02_Output.jar

package ab411_02_Output;

public class AB411_02_Output {

public static void main(String[] args) {
        System.out.println(«1+2=» + (1+2));
    }
}
```

Projekt erstellen in NetBeans

1. Starten Sie bmLP1 und öffnen Sie die Entwicklungsumgebung NetBeans.
2. Erstellen Sie ein neues Projekt Menu: File – New Project.
3. Wählen Sie in der Kategorie Java ein Projekt Java Application.
4. Geben Sie dem Projekt den Namen: AB411_02_Output
5. Speichern Sie das Projekt im Verzeichnis /vmadmin/home/Dokumente/Modul411

Einfache Eingaben

Leider ist es etwas komplizierter, Daten zeichenweise von der Tastatur zu lesen. Zwar steht ein vordefinierter Eingabe-Stream System.in zur Verfügung. Er ist aber nicht in der Lage, die eingelesenen Zeichen in primitive Datentypen zu konvertieren. Statt dessen muss zunächst eine Instanz der Klasse InputStreamReader und daraus ein BufferedReader erzeugt werden. Dieser kann dann dazu verwendet werden, die Eingabe zeilenweise zu lesen und das Ergebnis in einen primitiven Typ umzuwandeln.

Das folgende Listing zeigt ein Programm, das zwei Ganzzahlen a und b einliest, die zweite Zahl durch 2 dividiert und das Resultat mit ersten Ganzzahl addiert. Sollte das Resultat eine negative

Zahl sein, wird diese mit Math.abs() in den positiven Zahlenbereich überführt (Absoluter Wert). Das Ergebnis wird auf dem Bildschirm ausgegeben:

```
//  @author:       Ralph Maurer iet-gibb
//  Compilation:   javac AB411_02_Input.java
//  Execution:     java -jar AB411_02_Input.jar

package ab411_02_input;
import java.io.*;

 public class AB411_02_Input
 {
   public static void main(String[] args)
   throws IOException
   {
     double a, b, c;
     BufferedReader din = new BufferedReader(
                          new InputStreamReader(System.in));

     System.out.println(«Bitte a eingeben: «);
     a = Double.parseDouble(din.readLine());
     System.out.println(«Bitte b eingeben: «);
     b = Double.parseDouble(din.readLine());
     c = Math.abs(a + b/2);
     System.out.println(«abs(a + b/2)=»+c);
   }
}
```

Die `import`-Anweisung am Anfang des Listings dient dazu, die Klassen des Pakets java.io bekanntzumachen. Ohne diese Anweisung würden sowohl `BufferedReader` als auch `InputStreamReader` vom Compiler nicht gefunden und es gäbe eine entsprechende Fehlermeldung.

Werden die Zahlen `33` und `-45.5` eingegeben, so lautet die Ausgabe des Programms:

```
Bitte a eingeben:
33
Bitte b eingeben:
-45.5
abs(a + b/2)=10.25
```

Das Ergebnis von din.readLine ist ein String, der den Inhalt der Eingabezeile enthält. Sollen keine numerischen Datenwerte, sondern Zeichenketten eingelesen werden, kann der Rückgabewert auch direkt verwendet werden.

Datentypen

Alle primitiven Datentypen in Java haben eine feste Länge, die von den Designern der Sprache ein für allemal verbindlich festgelegt wurde. Ein sizeof-Operator, wie er in C# vorhanden ist, wird in Java daher nicht benötigt und ist auch nicht vorhanden

Primitive Datentypen von Java

Typname	*Länge*	*Wertebreich*	*Standardwert*
boolean	*1*	*true / false*	*false*
char	*2*	*Alle Unicode-Zeichen*	*\u0000*
byte	*1*	$-2^7 - 2^7-1$	*0*
short	*2*	$-2^{15} - 2^{15}-1$	*0*
int	*4*	$-2^{31} - 2^{31}-1$	*0*
long	*8*	$-2^{63} - 2^{63}-1$	*0*
float	*4*	$+/-3.40282347 * 10^{38}$	*0.0*
double	*8*	$+/-1.79769313486231570 * 10^{308}$	*0.0*

Der logische Typ

Mit boolean besitzt Java einen eigenen logischen Datentyp und beseitigt damit eine oft diskutierte Schwäche von C und C++. Der boolean-Typ muss zwangsweise dort verwendet werden, wo ein logischer Operand erforderlich ist. Ganzzahlige Typen mit den Werten 0 oder 1 dürfen nicht als Ersatz für einen logischen Typen verwendet werden.

Der Datentyp boolean kennt zwei verschiedene Werte, nämlich `true` und `false`. Neben den vordefinierten Konstanten gibt es keine weiteren Literale für logische Datentypen.

Der Zeichentyp

Java wurde mit dem Anspruch entworfen, bekannte Schwächen bestehender Programmiersprachen zu vermeiden, und der Wunsch nach Portabilität stand ganz oben auf der Liste der Designziele. Konsequenterweise wurde der Typ char in Java daher bereits von Anfang an 2 Byte groß gemacht und speichert seine Zeichen auf der Basis des Unicode-Zeichensatzes. Als einziger integraler Datentyp ist char nicht vorzeichenbehaftet.

Zeichen	*Bedeutung*
\b	*Rückschritt (Backspace)*
\t	*horizontaler Tabulator*
\n	*Zeilenschaltung (Newline)*
\r	*Seitenumbruch (Formfeed)*
\f	*Wagenrücklauf (Carriage return)*
\"	*Doppeltes Anführungszeichen*
**	*Backslash*
\nnn	*Oktalzahl nnn (kann auch kürzer als 3 Zeichen sein, darf nicht größer als oktal 377 sein)*

`char`-Literale werden grundsätzlich in einfache Hochkommata gesetzt. Daneben gibt es `String`-Literale, die in doppelten Hochkommata stehen. Ähnlich wie C stellt Java eine ganze

Reihe von Standard-Escape-Sequenzen zur Verfügung, die zur Darstellung von Sonderzeichen verwendet werden können:

Variablen dienen dazu, Daten im Hauptspeicher eines Programms abzulegen und gegebenenfalls zu lesen oder zu verändern. In Java gibt es drei Typen von Variablen:

› Instanzvariablen, die im Rahmen einer Klassendefinition definiert und zusammen mit dem Objekt angelegt werden.
› Klassenvariablen, die ebenfalls im Rahmen einer Klassendefinition definiert werden, aber unabhängig von einem konkreten Objekt existieren.
› Lokale Variablen, die innerhalb einer Methode oder eines Blocks definiert werden und nur dort existieren.

Daneben betrachtet die Sprachdefinition auch Array-Komponenten und die Parameter von Methoden und Exception-Handlern als Variablen.

Eine Variable in Java ist immer typisiert. Sie ist entweder von einem primitiven Typen oder von einem Referenztypen. Mit Ausnahme eines Spezialfalls bei Array-Variablen, auf den wir später zurückkommen, werden alle Typüberprüfungen zur Compile-Zeit vorgenommen. Java ist damit im klassischen Sinne eine typsichere Sprache.

Um einer Variablen vom Typ T einen Wert X zuzuweisen, müssen T und X zuweisungskompatibel sein.

Variablen können auf zwei unterschiedliche Arten verändert werden:

› durch eine Zuweisung
› durch einen Inkrement- oder Dekrement-Operator

Deklaration von Variablen

Die Deklaration einer Variable erfolgt in der Form Typname Variablenname;
Dabei wird eine Variable des Typs Typname mit dem Namen Variablenname angelegt.
Variablendeklarationen dürfen in Java an beliebiger Stelle im Programmcode erfolgen.

Lebensdauer/Sichtbarkeit von Variablen

Die Sichtbarkeit lokaler Variablen erstreckt sich von der Stelle ihrer Deklaration bis zum Ende der Methode, in der sie deklariert wurden. Falls innerhalb eines Blocks lokale Variablen angelegt wurden, sind sie bis zum Ende des Blocks sichtbar. Die Lebensdauer einer lokalen Variable beginnt, wenn die zugehörige Deklarationsanweisung ausgeführt wird. Sie endet mit dem Ende des Methodenaufrufs. Falls innerhalb eines Blocks lokale Variablen angelegt wurden, endet ihre Lebensdauer mit dem Verlassen des Blocks. Es ist in Java nicht erlaubt, lokale Variablen zu deklarieren, die gleichnamige lokale Variablen eines weiter außen liegenden Blocks verdecken. Das Verdecken von Klassen- oder Instanzvariablen ist dagegen zulässig.

Instanzvariablen werden zum Zeitpunkt des Erzeugens einer neuen Instanz einer Klasse angelegt. Sie sind innerhalb der ganzen Klasse sichtbar, solange sie nicht von gleichnamigen lokalen Variablen verdeckt werden. In diesem Fall ist aber der Zugriff mit Hilfe des this-Zeigers möglich: this.name greift immer auf die Instanz- oder Klassenvariable name zu, selbst wenn eine gleichnamige lokale Variable existiert. Mit dem Zerstören des zugehörigen Objekts werden auch alle Instanzvariablen zerstört.

Klassenvariablen leben während der kompletten Laufzeit des Programms. Die Regeln für ihre Sichtbarkeit entsprechen denen von Instanzvariablen.

Eindimensionale Arrays

Arrays in Java unterscheiden sich dadurch von Arrays in anderen Programmiersprachen, dass sie Objekte sind. Obwohl dieser Umstand in vielen Fällen vernachlässigt werden kann, bedeutet er dennoch:

› dass Array-Variablen Referenzen sind,
› dass Arrays Methoden und Instanz-Variablen besitzen,
› dass Arrays zur Laufzeit erzeugt werden.

Dennoch bleibt ein Array immer eine (möglicherweise mehrdimensionale) Reihung von Elementen eines festen Grundtyps. Arrays in Java sind semi-dynamisch, d.h., ihre Größe kann zur Laufzeit festgelegt, später aber nicht mehr verändert werden.Deklaration einer Array-Variablen

Erzeugen eines Arrays und Zuweisung an die Array-Variable

Die Deklaration eines Arrays entspricht syntaktisch der einer einfachen Variablen, mit dem Unterschied, dass an den Typnamen eckige Klammern angehängt werden:

```
int[] zahlen;
double[] kommazahlen;
boolean[] check;
```

Zum Zeitpunkt der Deklaration wird noch nicht festgelegt, wie viele Elemente das Array haben soll. Dies geschieht erst später bei seiner Initialisierung, die mit Hilfe des new-Operators oder durch Zuweisung eines Array-Literals erfolgt. Sollen also beispielsweise die oben deklarierten Arrays 2, 4 und 6 Elemente haben, würden wir das Beispiel wie folgt erweitern:

```
a = new int[2];
b = new double[4];
c = new boolean[6];
```

Ist bereits zum Deklarationszeitpunkt klar, wie viele Elemente das Array haben soll, können Deklaration und Initialisierung zusammen geschrieben werden:

```
int[] a = new int[2];
double[] b = new double[4];
boolean[] c = new boolean[6];
```

Alternativ zur Verwendung des new -Operators kann ein Array auch literal initialisiert werden. Dazu werden die Elemente des Arrays in geschweifte Klammern gesetzt und nach einem Zuweisungsoperator zur Initialisierung verwendet. Die Größe des Arrays ergibt sich aus der Anzahl der zugewiesenen Elemente:

```
int[] x = {1,2};
boolean[] y = {true, false};
```

Das Beispiel generiert ein int-Array x mit zwei Elementen und ein boolean-Array y mit zwei Elementen. Anders als bei der expliziten Initialisierung mit new muss die Initialisierung in diesem Fall unmittelbar bei der Deklaration erfolgen.

Zugriff auf Array-Elemente

Bei der Initialisierung eines Arrays von n Elementen werden die einzelnen Elemente von `0` bis `n-1` durchnummeriert. Der Zugriff auf jedes einzelne Element erfolgt über seinen numerischen Index, der nach dem Array-Namen in eckigen Klammern geschrieben wird. Das nachfolgende Beispiel deklariert zwei Arrays mit Elementen des Typs int bzw. boolean, die dann ausgegeben werden:

```
public class array1 {
	public static void main(String[] args) {
		int[] prim = new int[5];
		boolean[] b = {true,false};
		prim[0] = 2;
		prim[1] = 3;
		prim[2] = 5;
		prim[3] = 7;
		prim[4] = 11;
		System.out.println(«prim hat «+prim.length+» Elemente»);
		System.out.println(«b hat «+b.length+» Elemente»);
		System.out.println(prim[0]);
		System.out.println(prim[1]);
		System.out.println(prim[2]);
		System.out.println(b[0]);
		System.out.println(b[1]);
	}
}
```

Die Ausgabe des Programms ist:
prim hat 5 Elemente
b hat 2 Elemente

```
2
3
5
true
false
```

Der Array-Index muss vom Typ int sein. Aufgrund der vom Compiler automatisch vorgenommenen Typkonvertierungen sind auch short, byte und char zulässig. Jedes Array hat eine Instanzvariable length, die die Anzahl seiner Elemente angibt. Indexausdrücke werden vom Laufzeitsystem auf Einhaltung der Array-Grenzen geprüft. Sie müssen größer gleich 0 und kleiner als length sein.

Mehrdimensionale Arrays

Mehrdimensionale Arrays werden erzeugt, indem zwei oder mehr Paare eckiger Klammern bei der Deklaration angegeben werden. Mehrdimensionale Arrays werden als Arrays von Arrays angelegt. Die Initialisierung erfolgt analog zu eindimensionalen Arrays durch Angabe der Anzahl der Elemente je Dimension.

Der Zugriff auf mehrdimensionale Arrays geschieht durch Angabe aller erforderlichen Indizes, jeweils in eigenen eckigen Klammern. Auch bei mehrdimensionalen Arrays kann eine literale Initialisierung durch Schachtelung der Initialisierungssequenzen erreicht werden. Das folgende Beispiel erzeugt ein Array der Größe 2 * 3 und gibt dessen Elemente aus:

```
public class array2 {
  public static void main(String[] args) {
    int[][] a = new int[2][3];
    a[0][0] = 10;
    a[0][1] = 20;
    a[0][2] = 30;
    a[1][0] = 40;
    a[1][1] = 50;
    a[1][2] = 60;
    System.out.println(«»+a[0][0]+a[0][1]+a[0][2]);
    System.out.println(«»+a[1][0]+a[1][1]+a[1][2]);
  }
}
```

Die Ausgabe des Programms ist:

```
102030
405060
```

Da mehrdimensionale Arrays als geschachtelte Arrays gespeichert werden, ist es auch möglich, nichtrechteckige Arrays zu erzeugen. Das folgende Beispiel deklariert und initialisiert ein zweidimensionales dreieckiges Array und gibt es auf dem Bildschirm aus. Gleichzeitig zeigt es die Verwendung der length-Variable, um die Größe des Arrays oder Sub-Arrays herauszufinden.

```
public class array3 {
  public static void main(String[] args) {
     int[][] a = {{0},
                  {1,2},
                  {3,4,5},
                  {6,7,8,9}};
     for (int i=0; i<a.length; ++i) {
       for (int j=0; j<a[i].length; ++j) {
         System.out.print(a[i][j]);
       }
       System.out.println();
     }
   }
 }
```

Die Ausgabe des Programms lautet:

```
0
12
345
6789
```

Operator	*Bezeichnung*	*Bedeutung*
+	*Positives Vorzeichen*	*+n ist gleichbedeutend mit n*
-	*Negatives Vorzeichen*	*-n kehrt das Vorzeichen von n um*
+	*Summe*	*a + b ergibt die Summe von a und b*
-	*Differenz*	*a - b ergibt die Differenz von a und b*
*	*Produkt*	*a * b ergibt das Produkt aus a und b*
/	*Quotient*	*a / b ergibt den Quotienten von a und b. Bei integralen Typen handelt es sich hierbei um die ganzzahlige Division ohne Rest.*
%	*Modulo*	*a % b ergibt den Rest der ganzzahligen Division von a durch b. In Java lässt sich dieser Operator auch auf Fließkommazahlen anwenden.*
++	*Präinkrement*	*++a ergibt a+1 und erhöht a um 1. In Java lässt sich dieser Operator auch auf Fließkommazahlen anwenden.*
++	*Postinkrement*	*a++ ergibt a und erhöht a um 1. In Java lässt sich dieser Operator auch auf Fließkommazahlen anwenden.*
--	*Prädekrement*	*--a ergibt a-1 und verringert a um 1. In Java lässt sich dieser Operator auch auf Fließkommazahlen anwenden.*
--	*Postdekrement*	*a-- ergibt a und verringert a um 1. In Java lässt sich dieser Operator auch auf Fließkommazahlen anwenden.*

Zuweisungsoperator

Operator	*Bezeichnung*	*Bedeutung*
=	*Einfache Zuweisung*	*a = b weist a den Wert von b zu und liefert b als Rückgabewert.*
+=	*Additionszuweisung*	*a += b weist a den Wert von a + b zu und liefert a + b als Rückgabewert.*
-=	*Subtraktionszuweisung*	*a -= b weist a den Wert von a - b zu und liefert a - b als Rückgabewert.*
*=	*Multiplikationszuweisung*	*a *= b weist a den Wert von a * b zu und liefert a * b als Rückgabewert.*
/=	*Divisionszuweisung*	*a /= b weist a den Wert von a / b zu und liefert a / b als Rückgabewert.*
%=	*Modulozuweisung*	*a %= b weist a den Wert von a % b zu und liefert a % b als Rückgabewert.*
&=	*UND-Zuweisung*	*a &= b weist a den Wert von a & b zu und liefert a & b als Rückgabewert.*
|=	*ODER-Zuweisung*	*a |= b weist a den Wert von a | b zu und liefert a | b als Rückgabewert.*

Operator	*Bezeichnung*	*Bedeutung*
^=	*Exklusiv-ODER-Zuweisung*	*a ^= b weist a den Wert von a ^ b zu und liefert a ^ b als Rückgabewert.*
<<=	*Linksschiebezuweisung*	*a <<= b weist a den Wert von a << b zu und liefert a << b als Rückgabewert.*
>>=	*Rechtsschiebezuweisung*	*a >>= b weist a den Wert von a >> b zu und liefert a >> b als Rückgabewert.*
>>>=	*Rechtsschiebezuweisung mit Nullexpansion*	*a >>>= b weist a den Wert von a >>> b zu und liefert a >>> b als Rückgabewert.*

Relationel Operatoren

Operator	*Bezeichnung*	*Bedeutung*
==	*Gleich*	*a == b ergibt true, wenn a gleich b ist. Sind a und b Referenztypen, so ist der Rückgabewert true, wenn beide Werte auf dasselbe Objekt zeigen.*
!=	*Ungleich*	*a != b ergibt true, wenn a ungleich b ist. Sind a und b Objekte, so ist der Rückgabewert true, wenn beide Werte auf unterschiedliche Objekte zeigen.*
<	*Kleiner*	*a < b ergibt true, wenn a kleiner b ist.*
<=	*Kleiner gleich*	*a <= b ergibt true, wenn a kleiner oder gleich b ist.*
>	*Größer*	*a > b ergibt true, wenn a größer b ist.*
>=	*Größer gleich*	*a >= b ergibt true, wenn a größer oder gleich b ist.*

Logische Operatoren

Operator	*Bezeichnung*	*Bedeutung*
!	*Logisches NICHT*	*!a ergibt false, wenn a wahr ist, und true, wenn a falsch ist.*
&&	*UND mit Short-Circuit-Evaluation*	*a && b ergibt true, wenn sowohl a als auch b wahr sind. Ist a bereits falsch, so wird false zurückgegeben und b nicht mehr ausgewertet.*
||	*ODER mit Short-Circuit-Evaluation*	*a || b ergibt true, wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Ist bereits a wahr, so wird true zurückgegeben und b nicht mehr ausgewertet.*
&	*UND ohne Short-Circuit-Evaluation*	*a & b ergibt true, wenn sowohl a als auch b wahr sind. Beide Teilausdrücke werden ausgewertet.*
|	*ODER ohne Short-Circuit-Evaluation*	*a | b ergibt true, wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Beide Teilausdrücke werden ausgewertet.*
^	*Exklusiv-ODER*	*a ^ b ergibt true, wenn beide Ausdrücke einen unterschiedlichen Wahrheitswert haben.*

Anweisungen

Die if-Anweisung

Syntax

```
 if (ausdruck)
  anweisung;
```

oder

```
 if (ausdruck)
  anweisung1;
else
  anweisung2;
```

Die `if`*-Anweisung wertet zunächst den Ausdruck aus. Danach führt sie die Anweisung genau dann aus, wenn das Ergebnis des Ausdrucks* `true` *ist. Ist Ausdruck hingegen* `false`*, so wird die Anweisung nicht ausgeführt, sondern mit der ersten Anweisung nach der* `if`*-Anweisung fortgefahren.*

Mit der `if-else`*-Anweisung gibt es eine weitere Verzweigung in Java. Falls Ausdruck wahr ist, wird Anweisung1 ausgeführt, andernfalls Anweisung2. Eine der beiden Anweisungen wird also in jedem Fall ausgeführt.*

Die switch-Anweisung

Syntax

```
 switch (ausdruck) {
  case constant:
    anweisung;
  ...
  default:
}
```

Die `switch`*-Anweisung ist eine Mehrfachverzweigung. Zunächst wird der Ausdruck, der vom Typ* `byte, short, char, int` *oder der Wert eines Aufzählungstyps sein muss, ausgewertet. In Abhängigkeit vom Ergebnis wird dann die Sprungmarke angesprungen, deren Konstante mit dem Ergebnis des Ausdrucks übereinstimmt. Die Konstante und der Ausdruck müssen dabei zuweisungskompatibel sein.*

Das optionale `default`*-Label wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird. Ist kein default-Label vorhanden und wird auch keine passende Sprungmarke gefunden, so wird keine der Anweisungen innerhalb der* `switch`*-Anweisung ausgeführt. Jede Konstante eines* `case`*-Labels darf nur einmal auftauchen. Das default-Label darf maximal einmal verwendet werden.*

Die while-Schleife

Syntax

```
 while (ausdruck)
  anweisung;
```

Zuerst wird der Testausdruck, der vom Typ boolean sein muss, geprüft. Ist er `true`*, wird die Anweisung ausgeführt, andernfalls wird mit der ersten Anweisung hinter der Schleife weitergemacht. Nachdem die Anweisung ausgeführt wurde, wird der Testausdruck erneut geprüft usw. Die Schleife wird beendet, sobald der Test false ergibt.*

Die do-Schleife

Syntax

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

Die do*-Schleife arbeitet nichtabweisend, d.h. sie wird mindestens einmal ausgeführt. Da zunächst die Schleifenanweisung ausgeführt und erst dann der Testausdruck überprüft wird, kann die do-Schleife frühestens nach einem Durchlauf regulär beendet werden. Die Bearbeitung der Schleife wird immer dann beendet, wenn der Test des Schleifenausdrucks* `false` *ergibt.*

Die for-Schleife

Syntax

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

Der Kopf der `for`*-Schleife besteht aus drei Ausdrücken, die jeder für sich optional sind:*

Der init-Ausdruck wird einmal vor dem Start der Schleife aufgerufen. Er dient dazu, Initialisierungen durchzuführen, die durch die Auswertung von Ausdrücken mit Nebeneffekten verursacht werden. Der Rückgabewert der Ausdrücke wird vollständig ignoriert.

Der init-Teil darf auch aus mehreren Ausdrücken bestehen, wenn die einzelnen Teilausdrücke durch Kommata getrennt sind. Diese syntaktische Erweiterung ist allerdings nur innerhalb des Initialisierungsteils einer for-Schleife erlaubt.

Fehlt der Initialisierungsteil, wird keine Initialisierung im Kopf der Schleife durchgeführt.

Der init-Teil darf auch Variablendeklarationen enthalten, beispielsweise, um einen Schleifenzähler zu erzeugen. Die Variablen müssen bei der Deklaration initialisiert werden. Sichtbarkeit und Lebensdauer erstrecken sich auf den Block, der die Schleifenanweisungen enthält. Damit ist es möglich, den Namen einer Schleifenvariablen innerhalb einer Methode mehrfach zu deklarieren.

Der test-Teil bildet den Testausdruck der Schleife. Analog zur while-Schleife wird er am Anfang der Schleife ausgeführt und die Schleifenanweisung wird nur ausgeführt, wenn die Auswertung des Testausdrucks true ergibt. Fehlt der Testausdruck, so setzt der Compiler an seiner Stelle die Konstante `true` *ein.*

Der update-Ausdruck dient dazu, den Schleifenzähler zu verändern. Er wird nach jedem Durchlauf der Schleife ausgewertet, bevor der Testausdruck das nächste Mal ausgewertet wird. Wie der init-Teil darf auch der update-Teil aus mehreren Ausdrücken bestehen. Der Rückgabewert des Ausdrucks wird ignoriert. Fehlt der update-Ausdruck, so wird keine automatische Modifikation des Schleifenzählers durchgeführt.

In Java gibt es zwei weitere Möglichkeiten, die normale Auswertungsreihenfolge in einer Schleife zu verändern. Taucht innerhalb einer Schleife eine break-Anweisung auf, wird die Schleife verlassen und das Programm mit der ersten Anweisung nach der Schleife fortgesetzt. Taucht dagegen eine continue-Anweisung auf, springt das Programm an das Ende des Schleifenrumpfs und beginnt mit der nächsten Iteration