Iteration

Die *Iteration* ist eine der drei [Programmablaufstrukturen](http://freiheit.f4.htw-berlin.de/prog1/start/#programmablaufstrukturen), die es gibt. Die Nacheinanderausführung von Anweisungen, die Sequenz ist einfach und wir benutzen es ständig. Die [Selektion](http://freiheit.f4.htw-berlin.de/prog1/start/#die-selektion) haben wir uns [hier](http://freiheit.f4.htw-berlin.de/prog1/selektion/#selektion) genauer angeschaut. Nun geht es um die letzte Programmstruktur, die wir kennenlernen, die [Iteration](http://freiheit.f4.htw-berlin.de/prog1/start/#die-iteration).

Unter einer Iteration verstehen wir die **wiederholte** Ausführung eines Anweisungsblocks. Die Programmkonstrukte, mit denen wir eine Iteration umsetzen, werden *Schleifen* genannt. Wir werden drei Schleifen kennenlernen:

* die for-Schleife und
* die while-Schleife.
* die do...while-Schleife

**Die for-Schleife**

Die for-Schleife verwenden wir, wenn wir eine oder mehrere Anweisungen **abzählbar oft** wiederholen wollen, wenn wir also die **Anzahl** der Ausführungen kennen. Die Idee bei der for-Schleife ist die, dass wir uns

1. eine *Laufvariable* (typischerweise vom Datentyp int) deklarieren und initialisieren (INITIALISIERUNG),
2. eine Bedingung angeben, für welche Werte der *Laufvariablen* die Schleife wiederholt werden soll (BEDINGUNG) und
3. wie sich der Wert der *Laufvariablen* nach jedem Schleifendurchlauf ändern soll (ÄNDERUNG).

Die allgemeine Syntax für eine solche Vorschleife sieht so aus:

for(INITIALISIERUNG ; BEDINGUNG ; ÄNDERUNG)

{

/\*

\* Anweisungsblock, der wiederholt

\* werden soll

\*/

}

Wir betrachten ein einfaches Beispiel:

|  |  |
| --- | --- |
| 1  2  3  4 | for(int i=0; i<5; i++)  {  System.out.println(i);  } |

Der Ablauf dieser Schleife sieht wie folgt aus:

* Zuerst wird die Laufvariable i deklariert und mit dem Wert 0 initialisiert. Das passiert genau ein Mal.
* Dann wird geprüft, ob die Bedingung i<5 den Wert true ergibt. Das passiert vor jedem Schleifendurchlauf. Wenn der Wert true ist, wird der Schleifenkörper, also der Anweisungsblock ausgeführt. i<5 ist true, also wird System.out.println(i); ausgeführt. Der Wert von iist 0, also wird eine 0 auf die
* Konsole ausgegeben. Mehr Anweisungen gibt es nicht im Anweisungsblock, somit sind wir am Ende der Schleife.
* Nun wird der Wert von i geändert. Dazu wird i++ ausgeführt. Der Wert von i ist nun 1.
* Nun wird erneut geprüft, ob die Bedingung i<5 den Wert true ergibt. i<5 ist true, also wird System.out.println(i); ausgeführt. Der Wert von iist 1, also wird eine 1 auf die Konsole ausgegeben.
* Nun wird wieder der Wert von i geändert. Dazu wird i++ ausgeführt. Der Wert von i ist nun 2.
* Es wird erneut geprüft, ob die Bedingung i<5 den Wert true ergibt. i<5 ist true, also wird System.out.println(i); ausgeführt. Der Wert von iist 2, also wird eine 2 auf die Konsole ausgegeben.
* i++ → Wert von iist 3
* i<5 ist true → Anweisungsblock
* System.out.println(i); der Wert von i ist 3 → Ausgabe 3
* i++ → Wert von iist 4
* i<5 ist true → Anweisungsblock
* System.out.println(i); der Wert von i ist 4 → Ausgabe 4
* i++ → Wert von iist 5
* i<5 ist nun false → Deshalb wird der Anweisungsblock nicht mehr ausgeführt! Wir verlassen die Schleife und führen die nächste Anweisung aus, die nach der Schleife kommt.

Oberes Beispiel erzeugt also folgende Ausgabe auf der Konsole:

0

1

2

3

4

In den meisten for-Schleifen wird die Initialisierung wie oben aussehen, also eine Laufvariable (hier i) wird auf 0 am Anfang gesetzt und die Änderung des Wertes erfolgt dann durch die Erhöhung des Wertes um 1 (hier i++). Das kann aber auch anders sein, z.B.:

|  |  |
| --- | --- |
| 1  2  3  4 | for(int i=5; i>0; i--)  {  System.out.println(i);  } |

Hier ist der initiale Wert der Laufvariablen 5. Die Bedingung prüft, ob i größer ist als 0. nach jedem Schleifendurchlauf wird der Wert der Laufvriablen i um 1 rediziert. Es entsteht folgende Ausgabe:

5

4

3

2

1

Für die Änderung des Wertes der Laufvariablen können Sie auch jeden der in [**verkürzte Schreibweisen für arithmetische Operatoren**](http://freiheit.f4.htw-berlin.de/prog1/ausdruecke/#verkurzte-schreibweisen-fur-arithmetische-operatoren) eingeführten Operatoren verenden, z.B.

for(int i=1; i<10; i+=2)

{

System.out.println(i); // 1 3 5 7 9

}

for(int i=1; i<10; i\*=2)

{

System.out.println(i); // 1 2 4 8

}

**Deklaration von Variablen in der for-Schleife**

In dem oberen Beispiel ist die Laufvariable i zwei Mal deklariert, einmal für die erste for-Schleife und ein weiteres Mal für die zweite for-Schleife. Eigentlich hatten wir ja gesagt, dass eine Variable immer nur genau ein Mal deklariert wird. Korrekt ist es, dass eine Variable immer nur in dem Anweisungsblock existiert, in dem sie deklariert wird. Außerhalb dieses Anweisungsblockes existiert sie nicht. Wir hatten das auch schon bei Methoden erwähnt. Dort hatten wir gesagt, dass die Variablen, die in zwei verschiedenen Methoden deklariert werden, miteinander nichts zu tun haben, sondern dass es sich dabei um verschiedene Variablen handelt. Wenn wir eine Variable innerhalb der for-Schleife deklarieren, dann exitiert sie für die for-Schleife. Davor und danch existiert die Variable nicht (mehr). Deshalb müssen wir i in der zweiten for-Schleife auch erneut deklarieren. Wir kommen darauf nochmal ausführlicher zu sprechen, wenn es um die [Lebensdauer und Sichtbarkeit von Variablen](http://freiheit.f4.htw-berlin.de/prog1/methodenstack/#lebensdauer-und-sichtbarkeit-von-variablen) geht.

Weitere Beispiele für einfache for-Schleifen

Wir betrachten noch einige Beispiele für einfache for-Schleifen, um uns mit dem Konzept weiter vertraut zu machen.

**Summe 1 bis n**

Ausgabe der Summe von 1 bis n

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public static void computeSumFrom1ToN(int n)  {  int sum = 0;  String s = "1";  for(int i=1; i<=n; i++)  {  if(i>1)  {  s += " + " + i;  }  sum = sum + i;  System.out.println(s + " = " + sum);  }  } |

In der Methode computeSumFrom1ToN(int n) wird die Summe von 1 + 2 + ... + n berechnet, wobai n als Parameterwert der Methode übergeben wird. Jeder einzelne Schritt wird ausgegeben. Dazu wird ein String s erzeugt, der initial den Wert "1" hat. Für jede Weitere Addition kommt " + 2", " + 3" usw. zu diesem String hinzu.

* Beachten Sie, dass wir die Variable s außerhalb der for-Schleife deklariert haben. Wäre sie innerhalb der for-Schleife deklariert, dann würde sie bei jedem Schleifendurchlauf neu erzeugt werden. So wird ihr Wert bei jedem Schleifendurchlauf aktualisiert.
* Die Selektion wurde eingefügt, damit beim ersten Schleifendurchlauf (für i==1) nichts an den String s angehängt wird, sondern nur für alle weiteren Schleifendurchläufe.
* Beachten Sie auch, dass die Laufvariable i von 1 bis einschließlich n laäuft und wir dadurch die Summe von 1 + 2 + ... + n erzeugen. Wird als Parameterwert eine Zahl kleiner als 1 übergeben, erfolgt keine Ausgabe, denn dann ist die Bedingung 1<=n bereits vor dem ersten Schleifendurchlauf false.

Ausgabe für den Aufruf `computeSumFrom1ToN(10)`

1 = 1

1 + 2 = 3

1 + 2 + 3 = 6

1 + 2 + 3 + 4 = 10

1 + 2 + 3 + 4 + 5 = 15

1 + 2 + 3 + 4 + 5 + 6 = 21

1 + 2 + 3 + 4 + 5 + 6 + 7 = 28

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55

**Fakultät von n**

Die Fakultät von n ist definiert als n! = 1 \* 2 \* ... \* n für alle Natürlichen Zahlen n>=1. Wir schreiben uns dafür eine Methode und übergeben ein n:

Fakultät von n

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public static void fakultaetVonN(int n)  {  int product = 1;  String s = "!";  for(int i=1; i<=n; i++)  {  if(i==2)  {  s += " = 1 \* 2";  }  else if(i>2)  {  s += " \* " + i;  }  product \*= i;  System.out.println(i + s + " = " + product);  }  } |

In der Variablen product speichern wir das Produkt aus den Faktoren 1 \* 2 \* ... \* n. beachten Sie, dass product am Anfang den Wert 1 haben muss, um nicht immer mit 0 zu multiplizieren und somit wäre das Produkt immer 0. Anstelle von product \*= i; hätten wir auch product = product \* i; schreiben können. Weil wir unseren Ausgabestring s noch ein wenig komplizierter gestalttet haben, ist hier sogar eine Fallunterscheidung zwischen i==2 und i>2) notwendig.

Ausgabe für den Aufruf `fakultaetVonN(8)`

1! = 1

2! = 1 \* 2 = 2

3! = 1 \* 2 \* 3 = 6

4! = 1 \* 2 \* 3 \* 4 = 24

5! = 1 \* 2 \* 3 \* 4 \* 5 = 120

6! = 1 \* 2 \* 3 \* 4 \* 5 \* 6 = 720

7! = 1 \* 2 \* 3 \* 4 \* 5 \* 6 \* 7 = 5040

8! = 1 \* 2 \* 3 \* 4 \* 5 \* 6 \* 7 \* 8 = 40320

**Fibonacci-Folge**

Übung Fibonacci-Folge

Verschachtelte for-Schleifen

In den bisherigen Beispielen haben wir immer genau eine for-Schleife benötigt, da wir "nur" etwas aufaddiert oder aufmultipliziert haben, um eine eindimensionale Folge zu berechnen oder auszugeben. Wir wissen aber bereits, dass in dem Anweisungsblock des Schleifenkörpers jede beliebige Kontrollstruktur vorkommen kann, also eine Sequenz und/oder eine Iteration und/oder eine Selektion. Selektion und Sequenz haben wir in unseren Beispielen bereits verwendet. Nun wollen wir auch noch eine Schleife innerhalb der Schleife untersuchen.

**Rechteck**

Angenommen, wir sollen ein Rechteck aus lauter \*-zeichen auf die Konsole ausgeben und sowohl die Breite des Rechtecks als auch dessen Höhe sind variabel. Am Ende soll also so ein Bild herauskommen:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In diesem Beispiel ist die Breite 23 und die Höhe 5. Unsere Überlegungen sind zunächst wie folgt:

1. wir können nur zeilenweise ausgeben (spaltenweise geht nicht auf der Konsole)
2. wir benötigen eine Schleife, um die 23 Sterne in einer Zeile auszugeben
3. wir benötigen eine Schleife, um die 5 Zeilen auszugeben

D.h. wir überlegen uns zunächst, wie wir eine Zeile ausgeben. Wir nehmen dazu an, wir haben eine int-Variable width, die uns die Breite des Rechtecks vorgibt (z.B. 23):

|  |  |
| --- | --- |
| 1  2  3  4 | for(int col = 0; col < width; col++)  {  System.out.print("\*");  } |

Wir geben also in einer Schleife eine Anzahl width von Sternen aus. Beachten Sie,

* dass die Laufvariable col (für column) mit 0 initialisert wird. Deshalb ist die Schleifenbedingung col < width. Hätten wir col <= width geschrieben, würde ein Stern zu viel ausgegeben (außer, wir hätten col mit 1initialisiert). Sie müssen die Initialisierung und die Bedingung immer gut aufeinander abstimmen!
* dass wir zur Ausgabe print("\*") statt println("\*")verwenden, weil sonst nach jedem Stern ein Zeilenumbruch erfolgen würde, die Sterne also nicht nebeneinander sondern untereinander ausgegeben würden.

Nun überlegen wir uns, wie wir die Zeilen ausgeben. Dazu nehmen wir an, wir haben eine int-Variable height, die uns die Höhe des Rechtecks vorgibt (z.B. 5):

|  |  |
| --- | --- |
| 1  2  3  4 | for(int row = 0; row < height; row++)  {  // Hier soll jetzt eine Zeile ausgegeben werden  } |

Wir geben also in einer Schleife eine Anzahl height von Zeilen aus aus. In jeder Zeile soll die Anzahl width von Sternen ausgegeben werden. Wir müssen also die Schleife für die Sterne in die Schleife für die Zeilen einsetzen:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | for(int row = 0; row < height; row++)  {  for(int col = 0; col < width; col++)  {  System.out.print("\*");  }  } |

Jetzt heben wir nur noch ein kleines Problem. Nachdem wir unsere Zeile mit Sternen ausgegeben haben, steht der Kursor noch hinter dem zuletzt ausgegebenen Stern. Er sollte danach aber an den Anfang der neuen Zeile wandern. Wir müssen also noch für einen Zeilenumbruch sorgen. das erledigen wir mit System.out.println();. Diese Anweisung kommt nach der *inneren* Schleife in die *äußere* Schleife. Die gesamte Methode sieht dann so aus:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public static void printRectangle(int width, int height)  {  for(int row = 0; row < height; row++)  {  for(int col = 0; col < width; col++)  {  System.out.print("\*");  }  System.out.println();  }  } |

Die Zeilen 3-10 beschreiben die *äußere* for-Schleife und die Zeilen 5-8 die *innere* for-Schleife. Wir "laufen" einmal durch den Beginn unseres Programms durch. Angenommen, unsere Methode wird mit der Anweisung printRectangle(23,5); aufgerufen, d.h. die Variable width bekommt den Wert 23 und die Variable height den Wert 5 zugewiesen.

* Die Laufvariable row bekommt initial den Wert 0. 0ist kleiner als 5 und somit ist die Bedingung row < height true. Wir betreten also den Anweisungsblock der äußeren Schleife.
* Die erste Anweisung in diesem Anweisungsblock ist die innere for-Schleife. Diese wird nun vollständig abgearbeitet, d.h. die Laufvariable col nimmt alle Werte von 0 bis 22 an und gibt jedes Mal (also 23 Mal) einen \* aus. Wenn der Wert von col auf 23 gesetzt wurde, ist die Bedingung col < width nicht mehr true sondern false und die Abarbeitung der Schleife ist beendet.
* Es wird Zeile 9 und somit ein Zeilenumbruch ausgeführt.
* Dann wird der Wert von row um 1 erhöht (row++) und hat somit den Wert 1. Die Bedingung row < height ist true und somit wird erneut der Anweisungsblock der äußeren Schleife ausgeführt.
* Wieder ist die erste Anweisung in diesem Anweisungsblock die innere for-Schleife. Diese wird nun wieder vollständig abgearbeitet, d.h. die Laufvariable col nimmt alle Werte von 0 bis 22 an und gibt jedes Mal (also 23 Mal) einen \* aus. Wenn der Wert von col auf 23 gesetzt wurde, ist die Bedingung col < width nicht mehr true sondern false und die Abarbeitung der Schleife ist beendet.
* Es wird Zeile 9 und somit ein Zeilenumbruch ausgeführt.
* Dann wird der Wert von row um 1 erhöht (row++) und hat somit den Wert 2. Die Bedingung row < height ist true und somit wird erneut der Anweisungsblock der äußeren Schleife ausgeführt.
* usw. bis der Wert von row 5 ist. Dann wird die äußere for-Schleife verlassen und die Abarbeitung der Methode ist beendet.

Wichtig ist, dass die innere Schleife jedes Mal vollständig abgearbeitet wird, ehe der Zeilenumbruch erfolgt und dann der Wert von row erhöht wird. Wir ändern die Ausgabe unserer Methode mal ein wenig, um das Prinzip besser zu erkennen:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public static void printRectangle(int width, int height)  {  for(int row=0; row<height; row++)  {  System.out.print("(row = " + row + "): ");  for(int col = 0; col < width; col++)  {  System.out.print("[col = " + col + "]");  }  System.out.println();  }  } |

Für den Aufruf der Methode printRectangle(10,5); erhalten wir dann folgende Ausgabe:

(row = 0): [col = 0][col = 1][col = 2][col = 3][col = 4][col = 5][col = 6][col = 7][col = 8][col = 9]

(row = 1): [col = 0][col = 1][col = 2][col = 3][col = 4][col = 5][col = 6][col = 7][col = 8][col = 9]

(row = 2): [col = 0][col = 1][col = 2][col = 3][col = 4][col = 5][col = 6][col = 7][col = 8][col = 9]

(row = 3): [col = 0][col = 1][col = 2][col = 3][col = 4][col = 5][col = 6][col = 7][col = 8][col = 9]

(row = 4): [col = 0][col = 1][col = 2][col = 3][col = 4][col = 5][col = 6][col = 7][col = 8][col = 9]

Durch die Verschachtelung der for-Schleife erzeugen wir somit eine 2-dimensionale Ausgabe. Die innere Schleife entwickelt die horizontale Dimension (eine Zeile mit width Sternen) und die äußere for-Schleife entwickelt entwickelt die vertikale Dimension (height viele Zeilen).

Wir schauen uns noch ein Beispiel an. Nun ist die Bedingung der inneren Schleife vom Wert der Alufvariablen der äußeren Schleife abhängig.

**Dreieck**

Angenommen, wir wollen ein gleichschenkliges rechteckiges Dreieck erzeugen. Dazu übergeben wir die Höhe des Dreiecks als Wert. Angenommen, wir wollen ein Dreieck der Höhe 7, dann soll folgende Ausgabe erscheinen:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*\*\*\*

Unsere äußere Schleife wird sicherlich so aussehen, wie unsere äußere Schleife beim Rechteck. Angenommen, unsere Höhe ist in der Variablen height gespeichert, dann müssen wir height viele Zeilen ausgeben:

|  |  |
| --- | --- |
| 1  2  3  4 | for(int row = 0; row < height; row++)  {  // Hier soll jetzt eine Zeile ausgegeben werden  } |

Daran hat sich also nichts geändert, aber die innere Schleife sieht sicherlich anders aus, denn wir haben keine width-Variable mehr. Die Anzahl der Sterne in einer Zeile ist nicht konstant, sondern hängt davon ab, **in welcher Zeile** wir uns befinden:

Zeile 1: row == 0: 1 Stern ausgeben

Zeile 2: row == 1: 2 Sterne ausgeben

Zeile 3: row == 2: 3 Sterne ausgeben

Zeile 4: row == 3: 4 Sterne ausgeben

Zeile 5: row == 4: 5 Sterne ausgeben

Zeile 6: row == 5: 6 Sterne ausgeben

Zeile 7: row == 6: 7 Sterne ausgeben

Das bedeutet, die Bedingung der inneren Schleife muss sich ändern. Sie muss abhängig sein vom Wert von row:

|  |  |
| --- | --- |
| 1  2  3  4 | for(int col = 0; col <= row; col++)  {  System.out.print("\*");  } |

Wenn row den

* Wert 0 hat, wird die Schleife 1 Mal durchlaufen,
* Wert 1 hat, wird die Schleife 2 Mal durchlaufen,
* Wert 2 hat, wird die Schleife 3 Mal durchlaufen,
* usw.

Die gesamte Methode sieht dann so aus:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public static void printTriangle(int height)  {  for(int row=0; row<height; row++)  {  for(int col = 0; col <= row; col++)  {  System.out.print("\*");  }  System.out.println();  }  } |

**2 Schleifen in einer Schleife**

Wir betrachten noch ein letztes Beispiel. Wir könnten Schleifen natürlich noch weiter verschachteln, also noch eine weitere Schleife in der inneren Schleife implementieren. Das ist möglich, wird aber schnell unübersichtlich. Prinzipiell ist die Verschachtelungstiefe aber unbegrenzt endlich. Stattdessen wollen wir uns in unserem letzten Beispiel aber einmal überlegen, wie wir erneut ein gleichschenkliges rechtwinkliges Dreieck erzeugen könnten. Dieses Mal soll es aber nicht "linksbündig", sondern "rechtsbündig" sein, also so:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*\*\*\*

Der Unterschied zum oberen Beispiel ist der, dass wir nun immer erst eine bestimmte Anzahl an Leerzeichen ausgeben müssen, ehe wir den ersten Stern ausgeben. Dazu überlegen wir uns wieder die Abhängigkeiten für ein Dreieck der Höhe 7:

Zeile 1: row == 0: 6 Leerzeichen ausgeben + 1 Stern ausgeben (height == 7)

Zeile 2: row == 1: 5 Leerzeichen ausgeben + 2 Sterne ausgeben (height == 7)

Zeile 3: row == 2: 4 Leerzeichen ausgeben + 3 Sterne ausgeben (height == 7)

Zeile 4: row == 3: 3 Leerzeichen ausgeben + 4 Sterne ausgeben (height == 7)

Zeile 5: row == 4: 2 Leerzeichen ausgeben + 5 Sterne ausgeben (height == 7)

Zeile 6: row == 5: 1 Leerzeichen ausgeben + 6 Sterne ausgeben (height == 7)

Zeile 7: row == 6: 0 Leerzeichen ausgeben + 7 Sterne ausgeben (height == 7)

Die äußere Schleife bleibt wieder so wie vorher:

|  |  |
| --- | --- |
| 1  2  3  4 | for(int row = 0; row < height; row++)  {  // Hier soll jetzt eine Zeile ausgegeben werden  } |

Allerdings ist die Ausgabe einer Zeile nun in 2 Aufgaben zerlegt. Zuerst eine bestimmte Anzahl von Leerzeichen ausgeben und dann eine bestimmte Anzahl von Sternen:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | for(int row = 0; row < height; row++)  {  // zuerst muss eine bestimmte Anzahl von Leerzeichen ausgegeben werden  // dann wird eine bestimmte Anzahl von Sternen ausgegeben  } |

Für die Anzahl von Sternen haben wir bereits eine Lösung, die wir verwenden können:

|  |  |
| --- | --- |
| 1  2  3  4 | for(int col = 0; col <= row; col++)  {  System.out.print("\*");  } |

Diese Schleife können wir schonmal in unsere äußere Schleife einsetzen:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | for(int row = 0; row < height; row++)  {  // zuerst muss eine bestimmte Anzahl von Leerzeichen ausgegeben werden  for(int col = 0; col <= row; col++)  {  System.out.print("\*");  }  } |

So wie die Anzahl von Sternen abhängig von der Zeile ist, in der wir die Sterne ausgeben, so ist auch die Anzahl der Leerzeichen davon abhängig. Allerdings beginnen wir mit einem größeren Wert und werden dann immer kleiner (von 6 bis 0 bei der Höhe height==7). Die Anzahl der auszugebenden Leerzeichen ist also einerseits abhängig von der Gesamthöhe (height) und andererseits von der aktuellen Zeile row. Wir überlegen uns, mit welchem Startwert wir beginnen: am Anfang wollen wir 6 leerzeichen ausgeben, das sind height-1 viele. Danach ziehen wir von diesem Wert immer so viele ab, wie row groß ist, also erst -0, dann -1, dann -2 usw. Der Startwert ist also height -1 - row. In der letzten Zeile hat row den Wert 6. Dann wäre unser Startwert height -1 - 6 == 7 - 1 - 6 == 0. In der letzten Zeile wollen wir aber gar kein Leerzeichen mehr ausgeben, also muss dort schon unsere Bedingung false sein. Also setzen wir die Bedingung auf >0. Die Schleife für die Ausgabe der Leerzeichen ist dann wie folgt:

|  |  |
| --- | --- |
| 1  2  3  4 | for(int spaces = height -1 -row; spaces > 0; spaces--)  {  System.out.print(" ");  } |

Für viele ist eine solche Schleife schwer zu lesen, da sich der Wert der ALufvariablen reduziert und die Ermittlung des Initialwertes gleich 2 Subtraktionen enthält. Wir haben ja bereits eingangs gesagt, dass eine solche Schleife auch äquivalent in anderer Form geschrieben werden kann. Das gleiche Ergebnis erhalten wir mit der folgenden Implementierung:

|  |  |
| --- | --- |
| 1  2  3  4 | for(int spaces = 1; spaces < height - row; spaces++)  {  System.out.print(" ");  } |

Übung Schleife, initaile Werte und Bedingungen

Die gesamte Methode sieht dann so aus:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public static void printTriangleR(int height)  {  for(int row=0; row<height; row++)  {  for(int spaces = 1; spaces < height - row; spaces++)  {  System.out.print(" ");  }  for(int col = 0; col <= row; col++)  {  System.out.print("\*");  }  System.out.println();  }  } |

Übung linksbündiges DreieckÜbung rechtsbündiges Dreieck

**Success**

Wir haben for-Schleifen kennengelernt und können damit nun auch Iterationen implementieren. for-Schleifen verwenden wir, wenn wir einen Anweisungsblock eine bestimmte Anzahl oft wiederholt ausführen möchten. Wir haben auch for-Schleifen verschachtelt, um variabel in mehrere Dimensionen zu sein. und wir haben mehrere for-Schleifen innerhalb einer for-Schleife verwendet. Nun lernen wir noch zwei weitere Schleifen kennen.

**Die while-Schleife.**

Während die Anzahl der Ausführungen einer for-Schleife von einem *numerischen Wert* festgelegt wird, ist die Anzahl der Ausführungen einer while-Schleife von einem *logischen Ausdruck* abhängig. Prinzipiell muss man jedoch sagen, dass es auch völlig genügen würde, wenn man nur for-Schleifen oder nur while-Schleifen in einer Programmiersprache zur Verfügung hätte. Man kann mit beiden Schleifenarten (und später auch mit der do..while-Schleife) alle Iterationen implemnetieren, die programmierbar sind.

Schauen wir uns zunächst die allgemeine Syntax einer while-Schleife an:

while(BEDINGUNG)

{

/\*

\* Anweisungsblock, der wiederholt

\* werden soll

\*/

}

Eine while-Schleife ist also auf den ersten Blick weniger komplex als eine for-Schleife. Da wir aber gesagt haben, dass man mit beiden Schleifenarten die gleichen Programme umsetzen kann, schauen wir uns die ersten Beispiele der for-Schleife mal als while-Schleife an:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int i=0;  while(i<5)  {  System.out.println(i);  i++;  } |

Diese while-Schleife macht genau das gleiche, wie unser erstes Beispiel für die for-Schleife. Es wird eine Variable i deklariert und mit 0 initialisiert. Als Bedingung unserer while-Schleife wird geprüft, ob der Wert von i kleiner als 5 ist. Wenn ja, wird dieser Wert ausgegeben und der Wert von i um 1 erhöht. Nun wird wieder geprüft, ob der Wert von iimmer noch kleiner als 5 ist. Wenn ja, wird der Wert ausgegeben und um 1erhöht usw.

Es scheint zunächst, als wäre diese beiden Schleifenarten völlig redundant. Was bedeutet es nun, dass eine for-Schleife von einer bestimmten Anzahl und eine while-Schleife von einer Bedingung abhängig ist? Sehen wir uns dazu nochmal unsere beiden Beispielalgorithmen vom Anfang an:

* der [Euklidische Algrorithmus](http://freiheit.f4.htw-berlin.de/prog1/start/#beispiel-euklidischer-algorithmus) und
* die [(3n+1)-Vermutung (Collatz-Problem)](http://freiheit.f4.htw-berlin.de/prog1/start/#beispiel-3n1-vermutung-collatz-problem)

Die Beschreibung der Iteration beim Eukidischen Algorithmus war solange a ungleich b ist, wiederhole. Das bedeutet, dass die Bedingung für die Schleifenwiederholung a ungleich b ist. Nach wieviel Wiederholungen (also nach welcher **Anzahl**) lässt sich nicht sagen. Es lässt sich aber leicht die Bedingung formulieren, die gelten soll, damit die Schleife erneut ausgeführt wird, nämlich (a!=b).

Das gleiche gilt für die (3n+1)-Vermutung. Dort lautet die Beschreibung der Iteration solange n ungleich 1 ist, wiederhole. Nach wieviel Wiederholungen (also nach welcher **Anzahl**) lässt sich vorher nicht sagen, aber die Bedingung dafür, dass die Schleife erneut wiederholt werden soll, lässt sich leicht formulieren, nämlich (n!=1).

Implementierung des Euklidischen Algorithmus

Mithilfe der while-Schleife implementieren wir nun mal beide Allgorithmen. Zuerst den Euklidischen Algorithmus:

Berechnung des ggT nach Euklid

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public static void berechneGGT(int a, int b)  {  while(a != b)  {  if(a > b)  {  a = a -b;  }  else  {  b = b - a;  }  }  System.out.println("ggT: " + a);  } |

Die Ausgabe für z.B. berechneGGT(24, 40); ist 8.

**Veränderung der Werte von Parametern in Methoden**

In unserer Methode zur Berechnung des größten gemeinsamen Teilers nach Euklid haben wir die Werte der Parameter a und b innerhalb unserer Methode geändert (siehe a = a -b; und b = b - a;). **Das ist kein guter Stil!** So haben wir z.B. nicht die Möglichkeit, am Ende der Methode eine Ausgabe der Form Der ggT von 24 und 40 ist 8. zu erstellen, da wir auf die Werte 24 (von a) und 40 (von b) keinen Zugriff mehr haben. Wir sollten uns angewöhnen, die Parameterwerte in Methoden nicht zu ändern, sondern lieber mit Kopien der Werte zu rechnen. Später werden wir unsere Parameter als Konstanten definieren, dann ist eine Änderung gar nicht möglich. Das folgende Beispiel zeigt eine bessere Lösung:

Berechnung des ggT nach Euklid (ohne Änderung der Parameterwerte)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public static void berechneGGT(int a, int b)  {  int nr1 = a;  int nr2 = b;  while(nr1 != nr2)  {  if(nr1 > nr2)  {  nr1 = nr1 - nr2;  }  else  {  nr2 = nr2 - nr1;  }  }  System.out.println("Der ggT von " + a + " und " + b + " ist " + nr1);  } |

Die Ausgabe für z.B. berechneGGT(24, 40); ist nun Der ggT von 24 und 40 ist 8.

Implementierung der (3n+1)-Vermutung

Mithilfe der while-Schleife können wir nun auch die (3n+1)-Vermutung (Collatz-Problem) geeignet implementieren:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public static void printCollatzFolge(int n)  {  int number = n;  while(number != 1)  {  System.out.print(number + " ");  if(number%2 == 0)  {  number = number/2;  }  else  {  number = 3\*number+1;  }  }  System.out.println(number);  } |

Auch hier kopieren wir zunächst den Wert des Parameters, um diesen nicht zu ändern. Mithilfe von number%2 == 0 prüfen wir, ob number gerade oder ungerade ist. Ist number gerade, teilen wir den Wert durch 2, ist number ungerade, multiplizieren wir den Wert mit 3 und addieren 1, um jeweils den Nachfolger zu ermitteln. Solange dieser NAchfolger ungleich 1 ist, wird der nächste Nachfolger berechnet usw.

Die Ausführung der methode mit z.B. printCollatzFolge(17); erzeugt folgende Ausgabe: 17 52 26 13 40 20 10 5 16 8 4 2 1. Wir beginnen mit 17. Diese Zahl ist ungerade, also ist der Nachfolger 52. Diese Zahl und auch der Nachfolger 26 sind gerade. Der nächste Nachfolger 13 ist ungerade, dann kommen drei gerade Zahlen 40, 20 und 10 und erst dann wieder eine ungerade Zahl 5. 16 ist dann aber schon eine Potenz von 2 und somit endet die Folge mit der 1.

Übungen while-Schleife

Die do...while-Schleife

Wir haben ja bereits bei den for- und while-Schleifen erwähnt, dass eines der beiden Konzepte genügt hätte, um alle Iterationen zu implementieren. Da man aber ganz gute Unterscheidungsmöglichkeiten hat, um sich entweder für die for-Schleife (bestimmte Anzahl) oder für die while-Schleife (bestimmte Bedingung) zu entscheiden, haben beide Schleifenarten ihre Berechtigungsexistenz. Für die do ... while fällt die Abgrenzung zur while-Schleife noch schwerer. Generell lässt sich sagen, dass eine while-Schleife nicht unebdingt ausgeführt werden muss (nämlich dann, wenn die Bedingung bereits ganz zu Anfang schon false ist), eine do ... while-Schleife wird aber zumindest ein Mal ausgeführt, da die Prüfung der Bedingung erst nach dem Schleifendurchlauf erfolgt. Die allgemeine Syntax einer do ... while-Schleife ist wie folgt:

do

{

/\*

\* Anweisungsblock, der wiederholt

\* werden soll

\*/

}

while(BEDINGUNG);

Beachten Sie das Semikolon hinter der Bedingung! Es gibt Beispiele für den sinnvollen Einsatz von do ... while-Schleifen, z.B. wenn innerhalb der Schleife eine Eingabe erfolgt und die Bedingung prüft, ob es sich um eine korrekte Eingabe handelt. Wir werden uns aber zunächst nicht weiter um diese Schleife kümmern, da sie nicht wirklich notwendig ist und wir uns hauptsächlich mit for- und while-Schleifen beschäftigen werden.

break und continue

**break und continue**

In (fast) allen Java-Büchern liest man in dem Kapitel über Schleifen auch davon, dass es die beiden Anweisungen break; und continue; gibt. Ich will hier gar nicht darauf eigehen, was diese beiden Anweisungen machen, nur so viel: sie springen aus Schleifen heraus. Solche Art von "Sprüngen" (*go to statements*) gehören nicht in moderne, gute Programme. Wir nutzen diese Anweisungen nicht!!! Stattdessen sei in diesem Zusammenhang ein berühmtes Papier von [Edsger W. Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra) empfohlen: [Go To Statement Considered Harmful](https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf). Siehe dazu auch [hier](https://en.wikipedia.org/wiki/Goto#CITEREFDijkstra1968).