2 Teil 2: Nassi-Schneiderman

Wie kann man Nassi-Schneiderman in einer objektorientierten Sprache verwenden? Jedes Objekt besitzt Methoden, welche die Attribute des Objektes verändern. Das Verhalten der Methode wird durch einen Algorithmus beschrieben. Ein Algorithmus läßt sich in der Struktur und Ablauf gut mit Nassi-Schneiderman darstellen.
Im folgenden wollen wir die verschiedenen Nassi-Schneiderman Konstrukte, und ihre Umsetzung in Java, vorstellen.

2.1 Anweisungen

Eine Anweisung ist sozusagen das Atom (die kleinste unteilbare Einheit) eines Nassi-Schneiderman Diagramms.

Einzelne Anweisung — statement

$\pi = 3.14$

Anweisung - Code

```
double pi = 3.14;
```

2.2 Verzweigung

Eine Verzweigung findet immer anhand einer Bedingung (condition) statt. Diese entscheidet darüber, welcher Zweig ausgeführt wird.
Eine Bedingung ist ein Ausdruck, der den logischen Wert *wahr* (*true*) oder *falsch* (*false*) ergibt.

Verzweigung — if - then - else

radius ≥ 0 ? (Y / N)	
$umfang = 2 * \pi * r$	FEHLER, negativer radius
$flaeche = \pi * r^2$	

Verzweigung – Code

```
if (radius >= 0) {
  umfang  = 2*pi*r;
  flaeche = pi*r*r;
} else {
  throw new IllegalArgumentException("negativer Radius");
}
```

2.3 Alternative

Bei einer Alternative wird zwischen mehreren Bedingungen unterschieden, die sich (meist) auf ein bestimmtes Attribut beziehen. Die Bedingungen werden in der Reihenfolge geprüft, in der sie geschrieben werden. Sobald eine Bedingung *wahr* ist, werden die zugehörigen Anweisungen ausgeführt und das Konstrukt verlassen. Meist findet sich an letzter Stelle noch eine letzte Alternative, deren Anweisungen ausgeführt werden, wenn alle anderen Bedingungen *falsch* sind. In nachfolgenden Beispiel ist diese Alternative natürlich überflüssig.

Alternative — if then elseif

<table>
<tr><td colspan="4">radius</td></tr>
<tr><td>radius > 0
?</td><td>radius $= 0$
?</td><td>radius < 0
?</td><td rowspan="3">sonst</td></tr>
<tr><td>$umfang =$
$2 * \pi * r$</td><td>$umfang =$
0</td><td rowspan="2">FEHLER,
negativer
radius</td></tr>
<tr><td>$flaeche =$
$\pi * r^2$</td><td>$flaeche =$
0</td></tr>
</table>

Alternative mit verschiedenen Bedingungen – Code

```
if (radius > 0) {
    umfang  = 2*pi*r;
    flaeche = pi*r*r;
} else
if (radius == 0) {
    umfang  = 0;
    flaeche = 0;
} else
if (radius < 0) {
    throw new IllegalArgumentException("negativer Radius");
} else {
}
```

Für den Fall das es sich, bei der Unterscheidung, um ganze Zahlen handelt verwenden wir das gleiche Nassi-Schneiderman Konstrukt. Lediglich die Umsetzung nach Java ändert sich. Diese Art der Unterscheidung ist schneller als die für beliebige Bedingungen.

Alternative — switch - case

sign(radius) [a]			
1 ?	0 ?	-1 ?	sonst
$umfang = 2 * \pi * r$	$umfang = 0$	FEHLER, negativer radius	
$flaeche = \pi * r^2$	$flaeche = 0$		

[a]
$$sign(x) = \begin{cases} -1 & \text{wenn x negativ} \\ 0 & \text{wenn x 0} \\ 1 & \text{wenn x positiv} \end{cases}$$

Alternative mit ganzen Zahlen – Code

```
switch(sign(radius)) {
    case -1:
        throw new IllegalArgumentException("negativer Radius");
    case  0:
        umfang  = 0;
        flaeche = 0;
        break;
    case  1:
        umfang  = 2*pi*r;
        flaeche = pi*r*r;
        break;
    default:
        break;
}
```

2.4 Schleifen

Schleifen bestehen immer aus einer Laufbedingung und einem Schleifenkörper, der solange wiederholt ausgeführt wird, wie die Laufbedingung *wahr* ist. Man unterscheidet zwischen kopf- und fußgesteuerten Schleifen.

2.4.1 Kopfgesteuerte Schleifen

Bei einer kopfgesteuerten Schleife steht die Laufbedingung am Anfang des Schleifen-Konstrukts. Diese Bedingung wird vor dem Betreten des Schleifenkörpers ausgewertet. Nur wenn sie den Wert *wahr* ergibt wird die Schleife betreten.

Kopfgesteuerte Schleife — while - do

zahl ← Eingabe > 0
temp = 1
stellen = 0
temp <= zahl
temp = temp * 10
stellen = stellen + 1

Kopfgesteuerte Schleife – Code

```
int temp    = 1;
int stellen = 0;
while (temp <= zahl) {
    temp *= 10;
    stellen++;
}
```

Eine spezielle Form der kopfgesteuerten Schleife ist die Zähl-Schleife. Hier wird ein Index (der Schleifenindex) gezählt. Eine Zähl-Schleife besitzt also eine klar definierte Anzahl von Durchläufen. Der Kopf der Zähl-Schleife besteht aus drei Teilen:

1. Initialisierung `int i=0;`
   Beim Betreten der Schleife wird der Schleifenindex initialisiert
2. Laufbedingung `i<args.length;`
   Vor dem Betreten des Schleifenkörpers wird die Laufbedingung geprüft, nur wenn sie *wahr* ist wird der Schleifenkörper betreten.
3. Inkrementierung / Dekrementierung `i++;`
   Nach dem Durchlaufen des Schleifenkörpers wird die Zählvariable entsprechend in- bzw. dekrementiert.

Zähl-Schleife — for - loop

args[] ← Feld von Programmparameter
Durchlaufe alle Parameter im Feld args[]
Schreibe Parameter aus

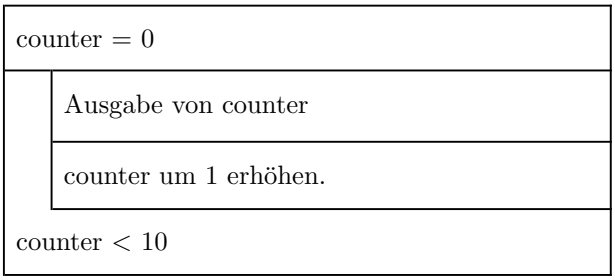
Zählschleife – Code

```
for (int i=0; i<args.length; i++) {
  System.out.println("argument no. " + i + " ist " + args[i]);
}
```

2.4.2 Fußgesteuerte Schleifen

Bei einer fußgesteuerten Schleife wird die Laufbedingung am Ende des Schleifenkörpers geprüft. Das bedeutet, das die Schleife mindestens einmal durchlaufen wird. Diese Schleife wird durchlaufen solange die Laufbedingung den Wert *wahr* ergibt.

Fußgesteuerte Schleife — do - while - loop



Fußgesteuerte Schleife – Code

```
int count = 0;
do {
  System.out.println("argument no. " + i + " ist " + args[i]);
  count++;
} while (count < 10);
```

2.5 break

break ist ein Schüsselwort der Sprache Java. Es wird verwendet um aus einem der Konstrukte: *while*, *do*, *for* oder *switch* vorzeitig „auszusteigen“.

Beispiel

```
for (int i=0; i<data.length; i++) {  // durchlaufe Daten-Feld
    if (data[i] == target) {         // ziel gefunden!
        index = i;                   // merken wo gefunden
        break;                       // Suche beenden
    }
} // hier wird nach dem break fortgesetzt.
```

In geschachtelten Schleifen kann es vorkommen, das in der inneren Schleife ein Zustand erkannt wird, der zum Beenden der geschachtelten Iteration führt. Ein einfaches *break* würde aber nur die innerste Schleife verlassen. Will man aber auch aus der äußeren Schleife „aussteigen“, so kann man dies durch ein „labeled break“ verwirklichen. Man schreibt dabei vor das Konstrukt, welches man verlassen will ein Label in der Form:

`label:` *statement*

Beispiel

```
testfornull: if (data != null){        // Prüfe ob Feld definiert
  for (int row=0; row<numrows; row++){ // Durchlaufe Reihen
    for (int col=0; col<numcols;col++){// Durchlaufe Zeilen
      if (data[row][col] == null){     // Wenn Element nicht def.
        break testfornull;             // Behandle Feld als nicht
      }                                //               definiert
    }
  }
} // Hier wird fortgesetzt nach dem "break testfornull"
```

Es ist zu beachten, daß ein *labled break* auch andere Konstrukte als Schleifen und *switch* verlassen kann. Im oben genannten Beispiel wird das erste *if* Statement mit einem Label versehen. Beim Ausführen des *break* Statements wird also dieses *if* Konstrukt verlassen.

2.6 continue

Auch das *continue* Statement ist Bestandteil der Java-Schlüsselwörter. Es wird in Schleifen dazu verwendet die aktuelle Iteration (Schleifendurchlauf) zu beenden und mit einem neuen Durchlauf zu beginnen. Das *continue* Statement bezieht sich auf die innerste Schleife, aber es ist auch hier möglich eine Variante mit Label zu schreiben, die zu einer äußeren Schleife springt.

Beipiel

```
for (int i=0; i<data.length; i++) { // Datenfeld durchlaufen
  if (data[i] == -1) {              // kein Wert eingetragen
    continue;                       // aktuellen Durchlauf
  }                                 // abbrechen.
  process(data[i]);                 // mit Wert rechnen...
}
```

Es gibt kleine Unterschiede, wie sich das *continue* bei den verschiedenen Schleifen verhält. Hier eine Beschreibung der Verhaltensweisen für die drei verschiedenen Schleifen.

- `while(condition){ ...statements... }`
  Der Interpreter springt zum Schleifenkopf. Die Bedingung (condition) wird ausgewertet. Ist das Ergebnis der Auswertung *wahr*, so wird die Schleife erneut durchlaufen.
- `do { ...statements... } while(condition)`
  Der Interpreter springt zum Schleifenfuß, wertet die Bedingung aus um zu entscheiden, ob der Schleifenkörper erneut durchlaufen werden darf.
- `for (init; condition; increment) { ...statements... }`
  Der Interpreter springt zum Schleifenkopf. Die Inkrementierungs-Anweisung wird ausgeführt. Danach wird die Bedingung geprüft und entschieden ob die Schleife ein weiteres mal durchlaufen werden darf.

2.7 Programmiert

Setzt folgendes Struktogramm in eine Java Methode um. Erzeugt dafür eine Klasse mit dem Namen **Factorial**. Die Klassen-Methode muß folgende Schnittstelle erfüllen:

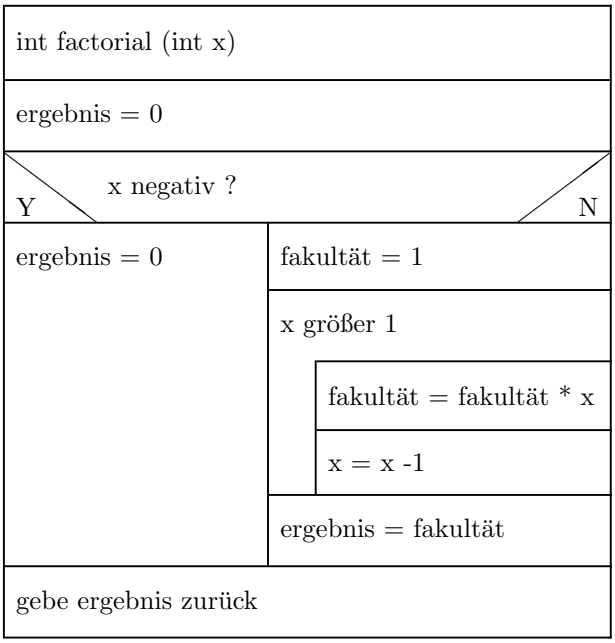
liefert die Fakultät von x	
Zugriff	`public`
Rückgabetyp	`int`
Name	`factorial`
Parameter	`int x`

Für ungültige Parameter soll die Methode **0** zurückgeben. Die *main* Methode ist nicht Teil der Aufgabe, deshalb drucken wir sie hier ab:

```
public static void main(String[] args) {
    // Umwandeln einer Zeichenkette in einen int.
    int eingabe  = Integer.parseInt(args[0]);
    // Die Fakultät berechnen.
    int ergebnis = factorial(eingabe);
    // Ausgabe auf dem Bildschirm.
    System.out.println(eingabe + "! = "+ ergebnis);
}
```

Das Struktogramm

Fakultät — Berechnen von x!



2.8 Aufgaben

1. Umsetzen von Schleifen
   Schreibe die folgende Zählschleife als äquivalente kopfgesteuerte Schleife (*while*):

   ```
   for (initialize; test condition; increment) {
     statement;
   }
   ```

   Die Teile sind bewußt allgemein gehalten. Wenn es Dir einfacher fällt kannst Du aber auch eine spezielle Zählschleife umsetzen.
   Unter welchen Umständen kann man keine allgemeine Umsetzung der *for*- in eine *while*- Schleife durchführen? (schwere Frage ;o))

2. Erweitern der Klasse Factorial
   Das Hauptprogramm liest den ersten Aufruf-Parameter des Programms ein (args[0]). Verbessere das Programm so, daß es alle Parameter, die der Benutzer angibt, in einer Zählschleife durchläuft und für jeden Parameter die Fakultät berechnet. Ein Fehlerabfang für nicht gültige Eingaben (keine ganzen Zahlen) ist nicht notwendig. Das Thema Fehlerbehandlung wird später näher erläutert.