Rekursion

Hier werden Einführung und Überblick über Rekursion gegeben.

Was ist Rekursion?

Der Begriff Rekursion betitelt eine Definitionsweise von Funktionen.
Eine Funktion ist nämlich genau dann rekursiv, wenn der Funktionswert f(n+1) einer Funktion f:N→N von bereits errechneten Werten (f(n), f(n-1), ...) abhängt.
Ein Beispiel hierfür wäre die Fibonacci-Folge:

f(n) = f(n-1) + f(n-2)
mit
f(0) = 0 und f(1) = 1

Wie man sieht, ergibt sich bei der Fibonacci-Folge der Funktionswert f(n) immer durch eine Addition der beiden vorherigen ns.

f(2) = f(1) + f(0) = 1 + 0 = 1
f(3) = f(2) + f(1) = 1 + 1 = 2
f(4) = f(3) + f(2) = 2 + 1 = 3
f(5) = f(4) + f(3) = 3 + 2 = 5
f(6) = f(5) + f(4) = 5 + 3 = 8
...

Rekursionen begegnen uns hauptsächlich in der Mathematik. Wir jedoch konzentrieren uns auf die Rekursion in der Informatik.
Das heißt, dass wir Funktionen so nutzen, dass sie sich selber wieder aufrufen und so zu einem Ergebnis kommen.
Dies kann bei bestimmten Aufgabenstellungen sehr viel einfacher und übersichtlicher sein, als iterative Lösungen.

Einführung

Fangen wir mal einfach an. Hier sieht man ein einfaches Zählen von 1-5.

Einstiegsbeispiel

```
public class Counting {
   public static void main (String[] args) {
      count(1);
   }

   public static void count(int i) {
      System.out.println(i);

      if (i <= 5)
         count(i + 1);

      return;
   }
}
```

Im Folgenden sieht man die Multiplikation (mit Addition) rekursiv umgesetzt.

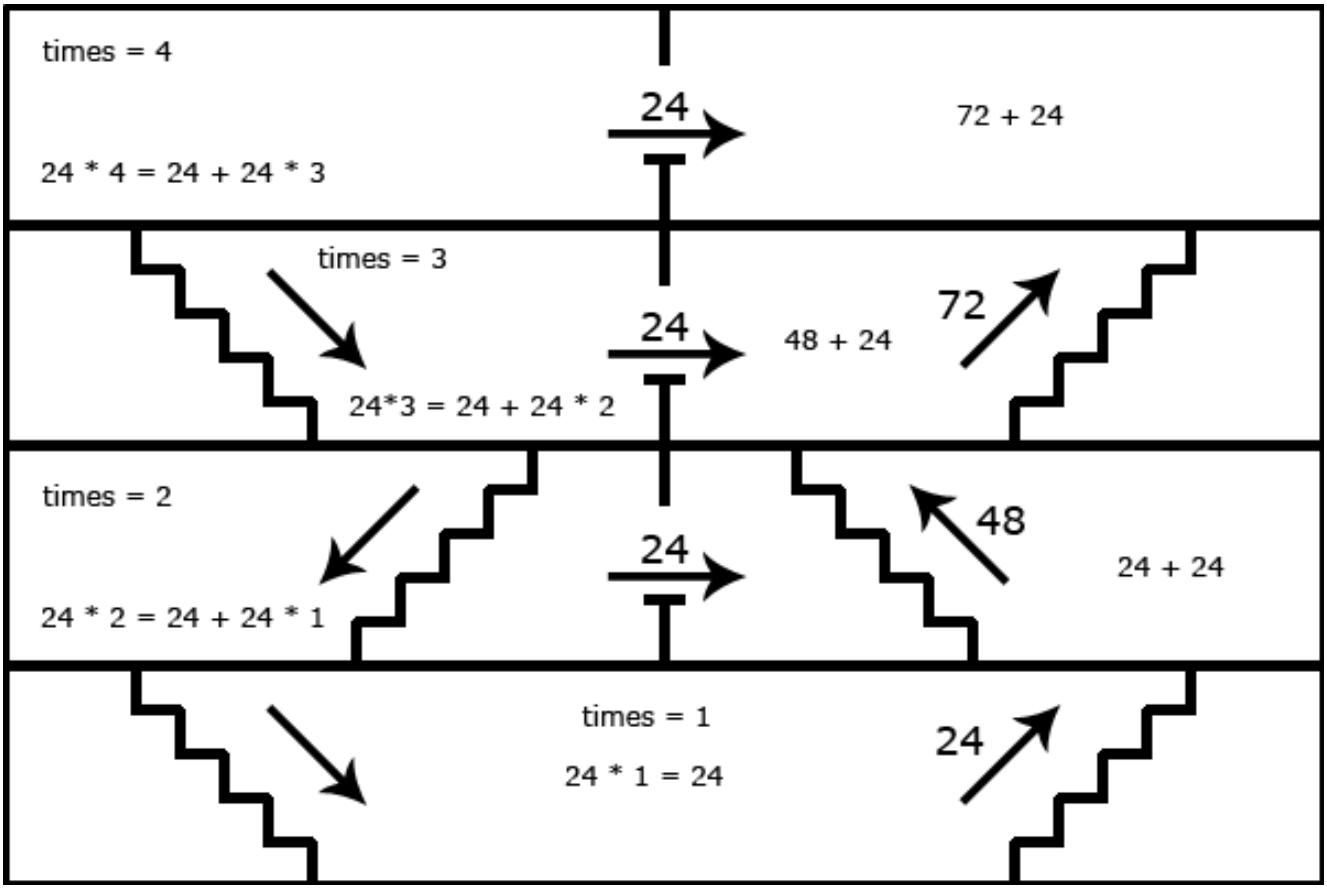
C++ Beispiel

```
#include <iostream>
using namespace std;

int multiply(int, int);

int main() {
   cout << "24 * 4 = " << multiply(24, 4) << "\n";
}

int multiply(int number, int times) {
   if(times == 0)
      return 0;
   return number + multiply(number, times-1);
}
```

Java Beispiel

```
public class multi {
   public static void main (String[] args) {
      System.out.println("24 * 4 = " + multiply(24, 4));
   }

   public static int multiply (int number, int times) {
      if (times == 1)
         return number;
      return number + multiply (number, times-1);
   }
}
```

Es ist schnell ersichtlich, dass die Funktion bzw. Methode multiply sich stets selber wieder aufruft.
Die Abbruchbedingung tritt dann ein, wenn die Variable times, die bei jedem Aufruf um 1 verringert wird, bei 0 angekommen ist.
Man kann sich das Ganze als Stufenmodell vorstellen.



Zunächst geht es die Stufen hinab, indem immer wieder multiply aufgerufen wird. Die Stufen wieder hoch geht es dann, wenn multiply ein Ergebnis liefert. Es ist also so, dass die zuerst aufgerufene Addition erst als allerletzte gelößt wird.
Rekursive Denkweise:

24 * 4 = 24 + 24 * 3
24 * 3 = 24 + 24 * 2
24 * 2 = 24 + 24 * 1
24 * 1 = 24 + 24 * 0

Gefahren von Rekursion

Achtet auf die Abbruchsbedingung!
Die soltet ihr nicht vergessen, denn das kann leicht mal zu Endlosschleifen füren ;)

Man sollte immer abschätzen, ob es nicht unter Umständen einfacher, kürzer und unaufwendiger (auch für den Rechner) ist eine iterative Lösung zu wählen. So ist zum Beispiel die modulare Expontiation iterativ mit wesentlich weniger Rechenaufwand zu realisieren: (Kann übersprungen werden)

Zu berechnen: $n^e \bmod n$

Iterativ: (Aufwand: n Multiplikationen und n Modulationen)

```
modexp(n, e, m) {
      i = 1;
      while (i < e) {
         prod *= n;   //prod = prod * n;
         prod %= m;    //prod = prod % m;
      }
      return prod
}
```

Rekursiv: (Aufwand: 2 * log(n), viel größerer Aufwand als bei iterativer Lösung. Es müssen viele Multiplikationen und Modulos berechnet werden.)

```
modexp(n, e, m){
   if (e == 0)
      return 1;
   t = modexp(n, e/2, m);
   c = (t * t) % m;
   if (e % 2 == 1)
      c = (c * n) % m;
   return c;
}
```

Beispiele

Fakultät berechnen

```
public class Fak {
   private static long fakultaet (long num) {
      return (num == 0) ? 1 : num * fakultaet(num-1);
   }
   public static void main(String[] args) {
      System.out.println(fakultaet(16));
   }
}
```