Vielleicht ist Ihnen noch das folgende Kinderlied bekannt:

Ein Mops kam in die Küche und stahl dem Koch ein Ei,
da nahm der Koch die Kelle und schlug den Mops entzwei.
Da kamen viele Möpse und gruben ihm ein Grab
und setzten ihm ein Grabstein, auf dem geschrieben stand:
Ein Mops kam in die Küche und stahl dem Koch ein Ei,
da nahm der Koch die Kelle und schlug den Mops entzwei.
Da kamen viele Möpse und gruben ihm ein Grab
und setzten ihm ein Grabstein, auf dem geschrieben stand:
Ein Mops kam in die Küche …

Soll dieses Kinderlied von einem Computer ausgegeben werden, so bietet sich natürlich eine (unendliche) Schleife an. Einfacher und eleganter wäre es jedoch, wenn die Prozedur, welche eine Strophe ausgibt, sich immer wieder selbst aufrufen könnte. Eine solche Prozedur nennt man rekursive Prozedur.

In einem Lexikon findet man unter dem Begriff Rekursion die Erläuterungen

- **Definition eines Verfahrens durch sich selbst**
- **… zurückgehen bis zu bekannten Werten (math.)**

In der Informatik bezeichnet man eine Prozedur bzw. Funktion, die sich selbst wieder aufruft und dadurch schließlich das Problem löst als rekursive Prozedur bzw. Funktion.

Das oben genannte Beispiel zeigt schon recht deutlich, welche Möglichkeiten und Gefahren die Rekursion mit sich bringt. Einerseits lassen sich bestimmte Probleme elegant lösen, andererseits können damit unendliche Prozesse endlich beschrieben werden. Dies widerspricht eigentlich der Definition eines Algorithmus.

Im Gegensatz zu einem rekursiven löst ein iterativer Algorithmus ein Problem, indem er eine Reihe von Anweisungen solange durchläuft, bis das Problem gelöst ist.

Zu jeder rekursiven Prozedur gibt es eine iterative Prozedur – und umgekehrt.

Wir wollen uns an einem Beispiel aus der Mathematik je eine iterative und eine rekursive Lösung für ein und dasselbe Problem anschauen. Eine Funktion soll die so genannte Fakultät berechnen.

Es ist z. B. $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$.

Oft wird das Bildungsgesetz für eine beliebige Fakultät jedoch auch so dargestellt:

$$n! = \begin{cases} 1 & \text{für} \quad n = 1 \\ n(n-1) & \text{für} \quad n > 1 \end{cases}$$

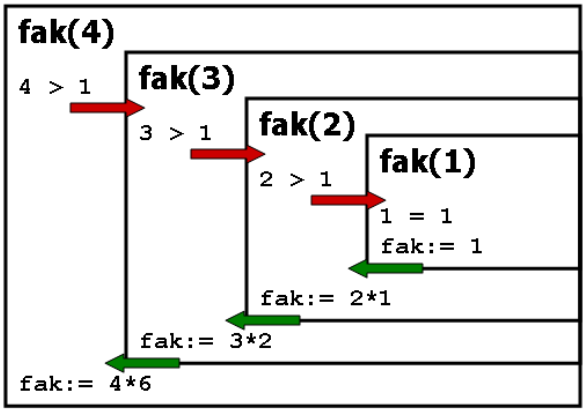
Dies ist eine rekursive Beschreibung der Fakultät, denn n! wird berechnet durch das Produkt aus n und der Fakultät des Vorgängers von n. Wir tun so, als könnten wir (n-1)! berechnen. Können wir auch, denn (n-1)! = (n-2)! * (n-1) und so weiter.

Dies wäre jedoch ein unendlicher Prozess, gäbe es nicht die Abbruchbedingung (auch Rekursionsanfang), die hier lautet: 1!=1. Eine entsprechende Java-Methode kann aus der rekursiven Definition leicht generiert werden:

```
public long fakrekursiv(int n) {
  if (n > 1) {
    return n * fakrekursiv(n - 1);
  } else {
    return 1;
  }
}
```

So einfach ist das. Wir haben nur die Rekursionsvorschrift von n! in Java-Anweisungen übersetzt. Schauen wir uns die Arbeitsweise dieser Methode an dem Beispiel der Berechnung von 4! an.

Beim ersten Aufruf der Methode erhält die Variable n den Wert 4; also wird der else-Zweig ausgeführt. Damit wird erneut die Methode aufgerufen, diesmal erhält n den Wert 3. Erst im vierten Aufruf der Methode erhält n den Wert 1, so dass die Methode im then-Zweig den Rückgabewert 1 erhält. Damit ist der vierte Aufruf beendet. Mit dem Rückgabewert aus dem vierten Aufruf wird jetzt der Rückgabewert des dritten Aufrufs berechnet: 1*2= 3. Letztlich gibt die Methode als Ergebnis für 4! das Ergebnis 24 aus.



Aufgabe 1

Implementieren Sie ein Java-Programm, welches die Fakultät einer Zahl rekursiv ermittelt. Testen Sie das Programm. Analysieren Sie das Programm:

- Wie viele Variable werden während des gesamten Programmablaufs vereinbart?
- Wieso könnte es beim Aufruf der Funktion für größere n Probleme geben?
- Weshalb müssen für die Arbeit mit rekursiven Prozeduren / Funktionen lokale Variablen verwendet werden?

Hätte man ohne die Möglichkeit der Rekursion eine Funktion Fakultät nicht schreiben können? Doch, sicherlich. Der Vorteil der Rekursion liegt allerdings in der Einfachheit und Übersichtlichkeit. Dass Rekursionen auch erhebliche Nachteile haben können, sehen wir an einem späteren Beispiel.

Eine iterative Lösung für eine Methode `fakiterativ()` könnte so aussehen:

```
public static long fakiterativ(int n) {
  long a;

  a = 1;
  for (int i = 1; i <= n; i++) {
    a = a * i;
  }
  return a;
}
```

Aufgabe 2:

Testen Sie auch die iterative Methode zum Berechnen der Fakultät einer Zahl.

- Worin bestehen die Vorteile der Verwendung einer rekursiven Funktion?
- Weshalb wäre für die Berechnung der Fakultät die iterative Funktion effektiver?

Betrachten wir ein weiteres Beispiel.

In der Mathematik gibt es eine Menge von Zahlenreihen, die durch genau definierte mathematische Bildungsgesetze generiert werden, z. B. die Fibonacci-Zahlen. Sie werden gebildet, indem man jeweils die letzte und die vorletzte Fibonacci-Zahl addiert:

$$fib(0) = 0$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2)$$

Auch dies ist wieder eine rekursive Definition, die sofort in einer Java-Methode umgesetzt werden kann.

Aufgabe 3:

Implementieren Sie eine Java-Methode (Konsolenprogramm), welche für ein eingegebenes n die Fibonacci-Zahl berechnet.

Testen Sie diese Methode. Es müsste die Zahlenfolge 0 – 1 – 1 – 2 – 3 – 5 – 8 – 13 – 21 – 34 – … entstehen. Berechnen Sie auch die Werte für `fib(25)`, `fib(45)` und `fib(100)` ☺.

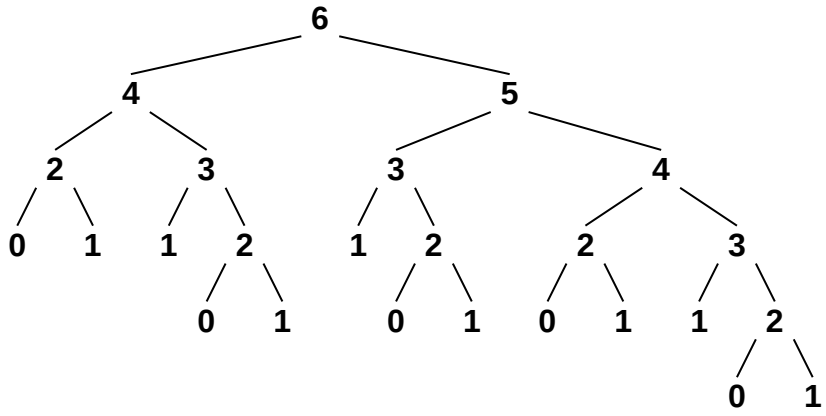
Nach unseren obigen Überlegungen sollte es auch eine iterative Lösung geben …

Aufgabe 4:

Implementieren Sie eine Java-Methode (Konsolenprogramm), welche für ein eingegebenes n die Fibonacci-Zahl **iterativ** berechnet.

Testen Sie dieses Programm mit den gleichen Zahlen wie in Aufgabe 3.

Woher kommt dieser große Zeitunterschied bei der Berechnung der Werte? Hier lohnt es sich darüber nachzudenken, wie z. B. die Fibonacci-Zahl für die Zahl 6 berechnet wird. Eine mögliche hilfreiche Darstellung ist ein Baum:



Es ist zu sehen, dass schon für die Berechnung von `fib(6)` die Funktion 25-mal aufgerufen wird. Für große Zahlen wird also die Rechenzeit enorm ansteigen. Für die Iteration wird die Funktion nur einmal aufgerufen und eine Schleife von 2 bis n abgearbeitet.

Ein weiteres Problem bei Rekursionen stellt der Stapelspeicher (Stack) dar. Wenn eine Funktion aufgerufen wird, muss sich der Rechner die Rücksprungadresse merken, denn nach Abarbeiten der Methode soll es ja an der gleichen Stelle weitergehen. Ruft sich eine Methode zu häufig auf, kann es so zu einer Überlastung des Stack kommen.

Zusammenfassend eine kurze Gegenüberstellung von Iteration und Rekursion.

	Iteration	**Rekursion**
Prinzip	Wiederholung durch Reihung (Bottom – up)	Wiederholung durch Schachtelung (Top – down)
Codierung	Schleifen	sich selbst aufrufende Prozeduren und Funktionen
Geschwindigkeit	schnell	langsam
Speicherbedarf	wenig	viel
... benötigt ...	... eine Abbruchbedingung	... einen Rekursionsanfang
Anwendung	Immer, wenn iterative Lösung offensichtlich bzw. bekannt ist.	Wenn iterative Lösung nicht offensichtlich ist – meist ist Datenstruktur oder das Problem selbst rekursiv

Aufgabe 5:

Der größte gemeinsame Teiler zweier Zahlen kann mit dem *Euklidischen Algorithmus* berechnet werden. Dafür gilt folgende Vorschrift:

$$ggT(a,b) = \begin{cases} a & \text{für} \quad a = b \\ ggT(a-b,b) & \text{für} \quad a > b \\ ggT(b-a,a) & \text{für} \quad a < b \end{cases}$$

a) Implementieren Sie die gegebene rekursive Methode ggT in Java.

b) Erstellen Sie einen Aufrufbaum für den Aufruf `ggT(15,12)`.

Aufgabe 6:

Gegeben ist die folgende rekursiv definierte Funktion F, die mit F(3,4) aufgerufen wird:

```
public static long F(int x, int y) {
  if (x == 0) {
    return 1;
  } else {
    if (x == 1) {
      return y;
    } else {
      return (F(x - 1, y + 1) + F(x - 1, y)) - F(x - 1, y - 1);
    }
  }
}
```

Geben Sie die Funktion in mathematischer Schreibweise an.

Welchen Wert berechnet sie? Legen Sie einen Aufrufbaum an.

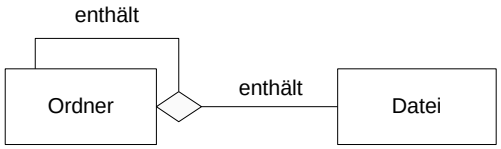
Aufgabe 7:

Eine Zeichenfolge soll zeichenweise eingegeben und anschließend in umgekehrter Reihenfolge wieder ausgegeben werden. Dabei soll die Eingabe durch die Eingabe von „!“ beendet werden.

Aufgabe 8:

Gesucht ist ein Programm welches die Potenz einer Zahl (a^n) rekursiv berechnet.

Aufgabe 9:



Ein Ordner auf einer Festplatte enthält Dateien und Ordner. Es liegt eine rekursive Struktur vor, die wie gezeigt als UML-Diagramm dargestellt werden kann. Zum Durchlaufen eines Ordners setzt man daher rekursive Algorithmen ein. Schreibe eine Prozedur, die die Namen aller Dateien in einem Ordner samt allen Unterordnern ausgibt. Benutze die Klasse File.