Technische Hochschule Rosenheim
Technical University of Applied Sciences

Objektorientierte Programmierung
Kapitel 6 – Rekursion

Prof. Dr. Kai Höfig

Begriff Rekursion in der Programmierung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Unter Rekursion versteht man in der Programmierung eine Methode (Funktion), die sich selbst direkt oder indirekt (über Zwischenaufrufe anderer Methoden) wiederaufruft.
- Üblicherweise verkleinern sich mit jedem Selbstaufruf einer Methode die übergebenen rekursionssteuernden Parameterwerte
- Häufig wird die Berechnung eines Funktionswertes f(n) („großes Problem“) auf die Berechnung des Funktionswertes f(n−1) („kleineres Problem“) zurückgeführt, bis triviale Probleme wie die Berechnung von f(1) oder f(0) entstehen
  - direkter Selbstaufruf: f(5)→f(4)→f(3)→f(2)...
  - indirekter Selbstaufruf: f(5)→g(5)→h(5)→f(4)→g(4)...

Beispiel: Fakultät iterativ und rekursiv

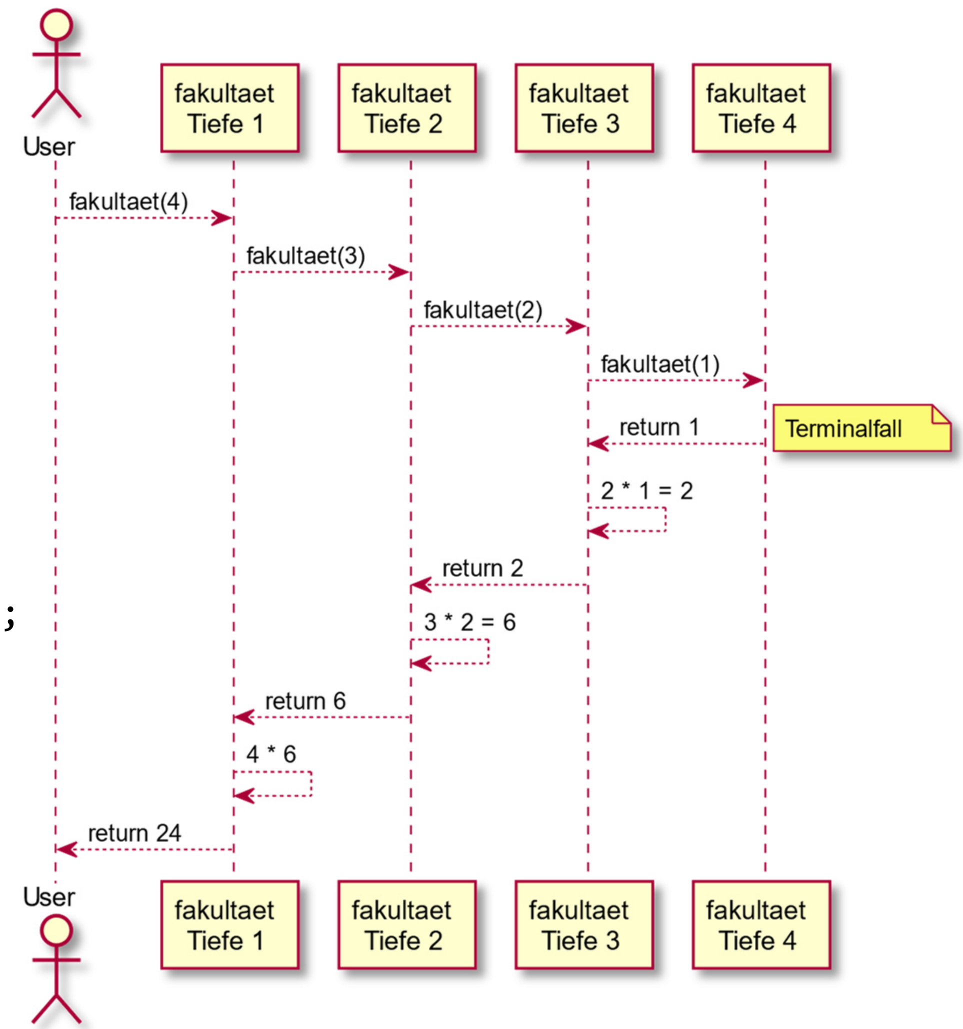
$$n! = \begin{cases} 1 & \text{für n} = 1 \text{ (terminal)} \\ n \cdot (n-1)! & \text{für n} > 1 \text{ (rekursiv)} \end{cases}$$

```
static int fakultaetIT(int n) {
    int faku =1;
    // Iterative Berechnung
    for(int i = 1; i<=n; i++)
    {
        faku *= i;
    }
    return faku;
}
```

```
static int fakultaetRK(int n) {
    if (n == 1) {
        // Regel 1: terminal
        return 1;
    } else {
        // Regel 2: rekursiv
        return n * fakultaetRK(n - 1);
    }
}
```

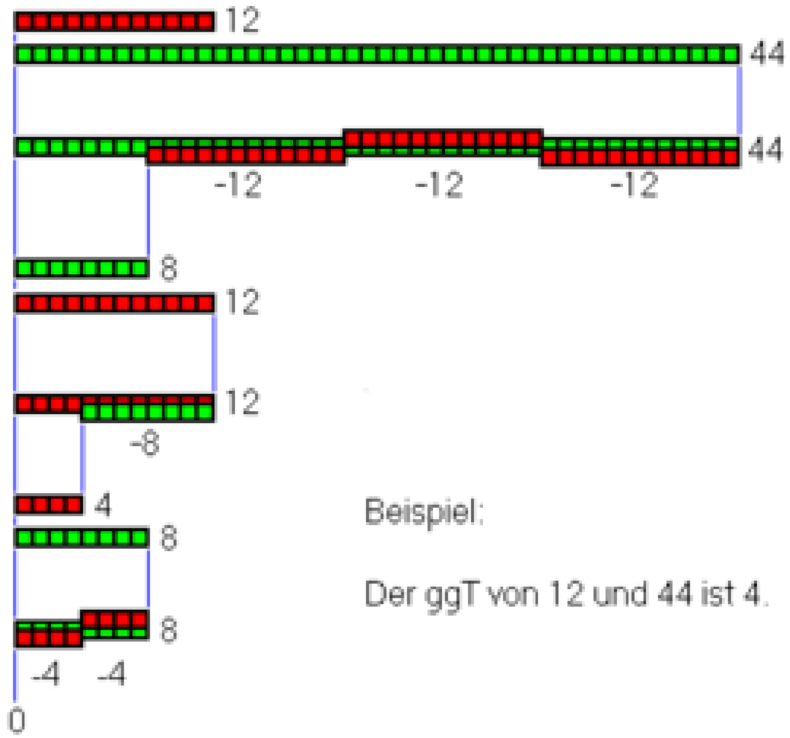
Rekursion zur Fakultät schematisch

```
static int fakultaetRK(int n) {
    if (n == 1) {
        // Regel 1: terminal
        return 1;
    } else {
        // Regel 2: rekursiv
        return n * fakultaetRK(n - 1);
    }
}
```



Größter gemeinsamer Teiler (ggT) iterativ nach Euklid

- **Euklidischer Algorithmus**:
  - Gesucht ist das gemeinsames *Maß* für die Längen a und b. Es muss möglich sein, die beiden Längen voneinander abzuziehen, bis das *gemeinsame Maß* übrig bleibt.



Beispiel:

Der ggT von 12 und 44 ist 4.

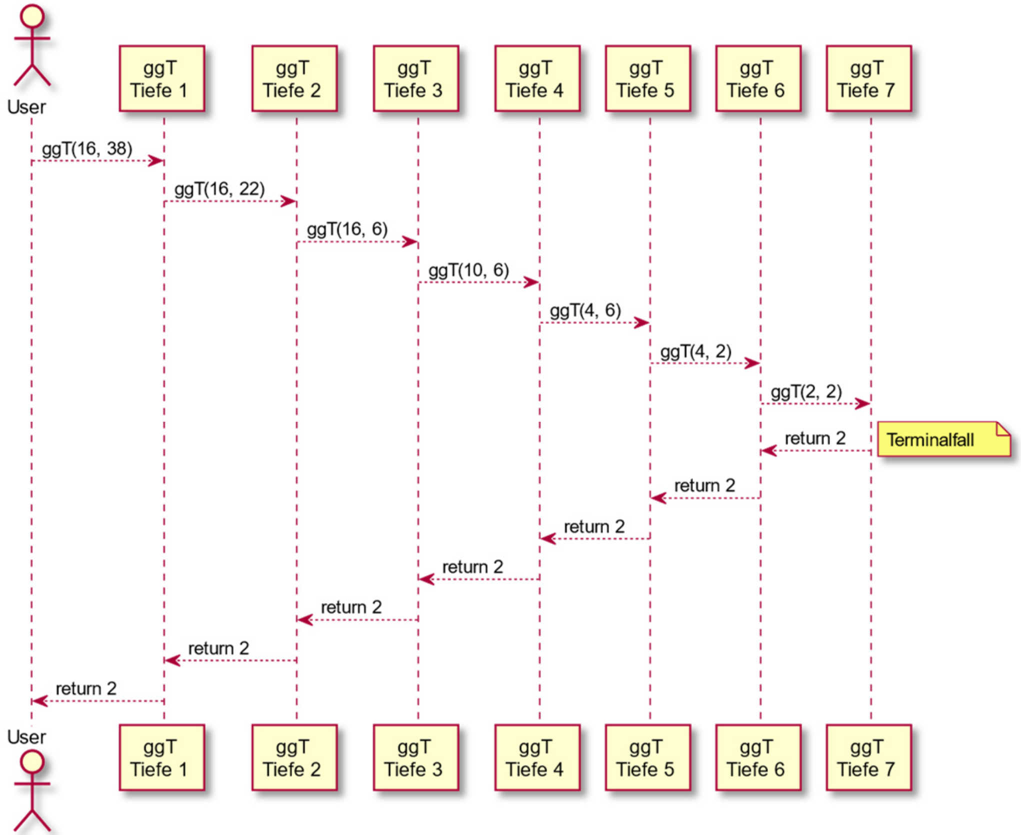
```
int ggTIT(int a, int b) {
    while (b != 0) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }

    return a;
}
```

Rekursion Kochrezept

1. Terminalfälle bestimmen. Wann ist die Lösung trivial?
2. Rekursionsfälle bestimmen. Wie kann ich das Problem auf ein kleineres runterbrechen?
3. Rekursion zusammensetzen: Brauche ich eine Hilfsmethode, wie muss die Signatur aussehen, wie müssen die Argumente beim rekursiven Aufruf verändert werden?

```java
// kein valides Java...
int rekursiv(...) {
    if (Terminalfall) {
        return /* fester Wert */
    } else {
        // Rekursionsfall: mind. 1x rekursiv aufrufen!
        return rekursiv(/* veränderte Argumente*/);
    }
}
```

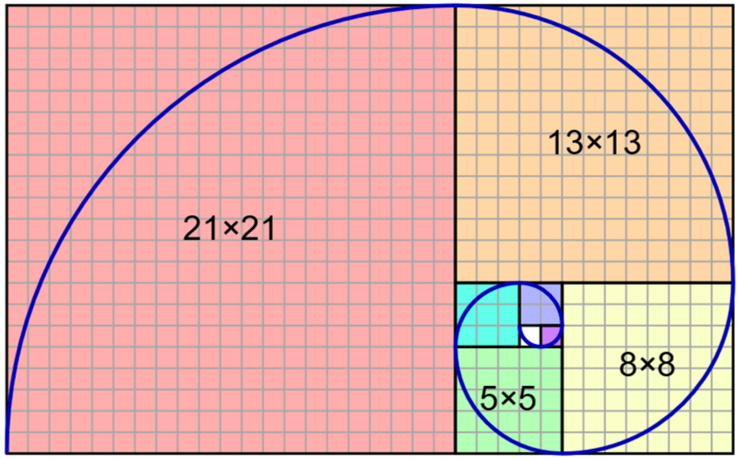
Größter gemeinsamer Teiler (ggT) rekursiv schematisch




```java
static int ggtRK(int a, int b) {
    // Abbruchbedingung
    if (b == 0)
        return a;
    // Rekursionsfall
    if (a > b)
        return ggtRK(a-b, b);
    return  ggtRK(a, b-a);
}
```

Fibonacci

Technische Hochschule Rosenheim
Technical University of Applied Sciences



$$\mathrm{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{für } n > 1 \end{cases}$$

```
static int fibIT(int n) {
    int x = 0, y = 1, z = 1;
    for (int i = 0; i < n; i++) {
        x = y;
        y = z;
        z = x + y;
    }
    return x;
}

static int fibRE(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibRE(n-1) + fibRE(n-2);
}
```

Fibonacci als einfache Rekursion

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```java
static int fibRE(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibRE(n-1) + fibRE(n-2);
}
```

Diese einfache Implementierung hat aber einen Nachteil: Im Rekursionsfall wird die Methode gleich zwei Mal aufgerufen. Allein ein Aufruf von fib(70) benötigt bereits mehrere Sekunden bis Minuten zur Berechnung.

```
fib(5) =>
 |    \
 |     \
fib(4) + fib(3) =>
 |    \    \-----+--------
 |     \          \       \
fib(3) + fib(2) + fib(2) + fib(1) =>
 |    \    \        \
 |     \    \        \-----------+---------
 |      \    \---+---------         \         \
 |       \        \        \         \         \
fib(2) + fib(1) + fib(1) + fib(0) + fib(1) + fib(0) + fib(1) =>
 |    \
 |     \
fib(1) + fib(0) + ...
```

Fibonacci mit Cache

```java
static private Map<Integer, Integer> cache = new HashMap<>();

static int fibCached(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
        // bereits ausgerechnet?
    else if (cache.containsKey(n)) return cache.get(n);
    else {
        int a = fibCached(n-1);
        int b = fibCached(n-2);
        if (!cache.containsKey(n-1))
            cache.put(n-1, a);
        if (!cache.containsKey(n-2))
            cache.put(n-2, b);

        return a + b;
    }
}
```

Fibonacci mit Hilfsfunktion

- Eine weitere Optimierung der obigen Rekursion wäre die Vorschrift genauer zu betrachten:

$$\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$$

- Ein Wert hängt also immer genau von seinen zwei Vorgängern ab.

- Diese kann man nun auch als Argumente in einer Hilfsfunktion "mitschleifen".

```java
static int fibBesser(int n) {
    // initialisiere Terminalfälle
    return fibHilf(n, 0, 1);
}

private static int fibHilf(int n, int a, int b) {
    if (n == 0)     return a;
    else if (n == 1) return b;
        // angepasste Parameter!
    else return fibHilf(n-1, b, a+b);
}
```

Palindrom

```java
static boolean istPalindromIT(String s) {
    for (int i = 0; i < s.length()/2; i++)
        if (s.charAt(i) != s.charAt(s.length()-1-i))
            return false;
    return true;
}

static boolean istPalindromRK(String s) {
    if (s.length() < 2)
        // Leer und ein Zeichen sind immer Palindrom
        return true;
    else if (s.charAt(0) != s.charAt(s.length() - 1))
        return false;  // Oops.
    else
        // angenommen erster und letzter passen,
        // was ist mit dem Rest?
        return istPalindromRK(
                s.substring(1, s.length() - 1));
}
```

Rekursion für Listen

- Möchte man nun die Größe (size) der Liste bestimmen, so muss man wieder Terminal- und Rekursionsfälle betrachten.

  - Eine Liste welche kein erstes Element hat ist leer.
  - Gibt es ein erstes Element, so kann man dieses Fragen wie lang es denn ist.
  - Ein Element ist in jedem Fall mind. 1 lang; gibt es einen `next` Nachfolger, so muss man dazu noch die Länge des Nachfolgers addieren.

```
class Liste<T> {
    Element first;

    public int size() {
        if (first == null)  return 0; // Terminalfall 1
        else return first.size();     // Hilfsmethode!
    }

    class Element {
        T value;
        Element next;
        int size() {
            if (next == null) return 1; // Terminalfall 3a
            else return 1 + next.size();
        }
    }
    // ...
}
```

Rekursion für Bäume

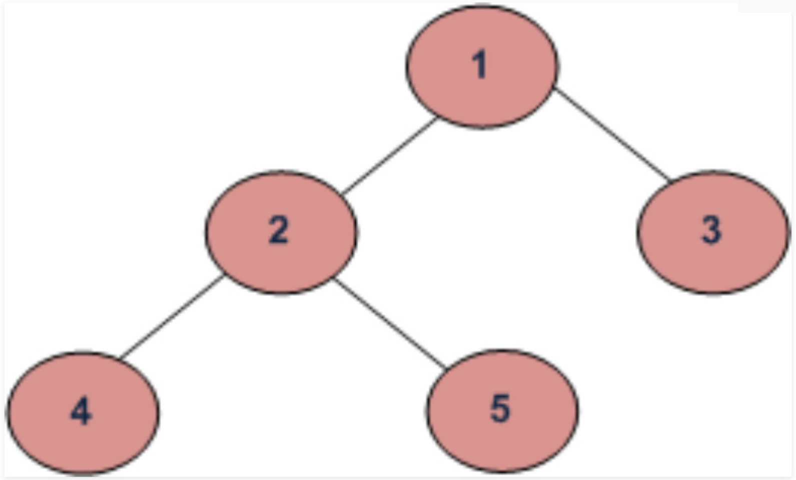
- Hier können wir z.B. die Größe (size) rekursiv definieren:

  - Terminalfall: Gibt es keinen Wurzelknoten, so ist der Baum leer.
  - Rekursionsfall: Gibt es einen Wurzelknoten, so ist die Baumgröße mind. 1 (Terminalfall), sowie zusätzlich die Größe des linken und rechten Teilbaums (Rekursion, sofern vorhanden).

```java
public class Baum<T extends Comparable<T>> {
    class Element {
        T value;
        Element left, right;
        Element(T value) { this.value = value; }
        int size() {
            return 1 +
                    (left == null ? 0 : left.size()) +
                    (right == null ? 0 : right.size());
        }
    }

    Element root;

    int size() {
        if (root == null) return 0;
        else return root.size();
    }
}
```

Baum Traversierungen

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```




```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Depth First Traversals:

(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```

Arten der Rekursion

- **Lineare** Rekursion: genau ein rekursiver Aufruf, z.B. Fakultät.

- **Repetetive** Rekursion (Rumpfrekursion, engl. *tail recursion*): Spezialfall der linearen Rekursion, bei der der rekursive Aufruf die letzte Rechenanweisung ist. Diese Rumpfrekursionen können direkt in eine iterative Schleife umgewandelt werden (und umgekehrt). Beispiel: verbesserte Implementierung der Fibonacci Funktion.

- **Kaskadenartige** Rekursion: in einem Zweig der Fallunterscheidung treten mehrere rekursive Aufrufe auf, was ein lawinenartiges Anwachsen der Funktionsaufrufe mit sich bringt. Beispiel: einfache Implementierung der Fibonacci Funktion.

- **Verschränkte** Rekursion: Eine Methode f() ruft eine Methode g(), die wiederum f() aufruft.

Zusammenfassung

- Eine rekursive Methode ist eine Methode, die sich selbst wieder aufruft; charakteristisch sind die Abwesenheit von `for` und `while`, sowie klare `if-else` Anweisungen, welche Terminal- von Rekursionsfall unterscheiden.
- Bei kaskadenartigen Rekursionen, also mehr als ein rekursiver Aufruf pro Durchlauf, können je nach Problemstellung Caches die Berechnung enorm effizienter gestalten.
- Repetitive Rekursion ist wünschenswert, da diese effektiv als `for` bzw. `while` Schleife realisiert werden könnten.
- Für obige braucht man oft Variablen, welche die Zwischenergebnisse im rekursiven Aufruf codieren.