Technische Hochschule Rosenheim
Technical University of Applied Sciences

Object-oriented programming
Chapter 6 – Recursion

Prof. Dr Kai Höfig

The term recursion in programming

- In programming, recursion is a method (function) that calls itself again either directly or indirectly (via intermediate calls of other methods).
- Usually the recursion-controlling parameter values that are passed become smaller with each recursive call (self-call).
- Often, the calculation of the value of a function f(n) ("big problem") is reduced to the calculation of the value of a function f(n−1) ("smaller problem") until trivial problems such as the calculation of f(1) or f(0) arise
  - direct recursive call: f(5)→f(4)→f(3)→f(2)...
  - indirect recursive call: f(5)→g(5)→h(5)→f(4)→g(4)...

Example:
Faculty iteratively and recursively

Technische Hochschule Rosenheim
Technical University of Applied Sciences

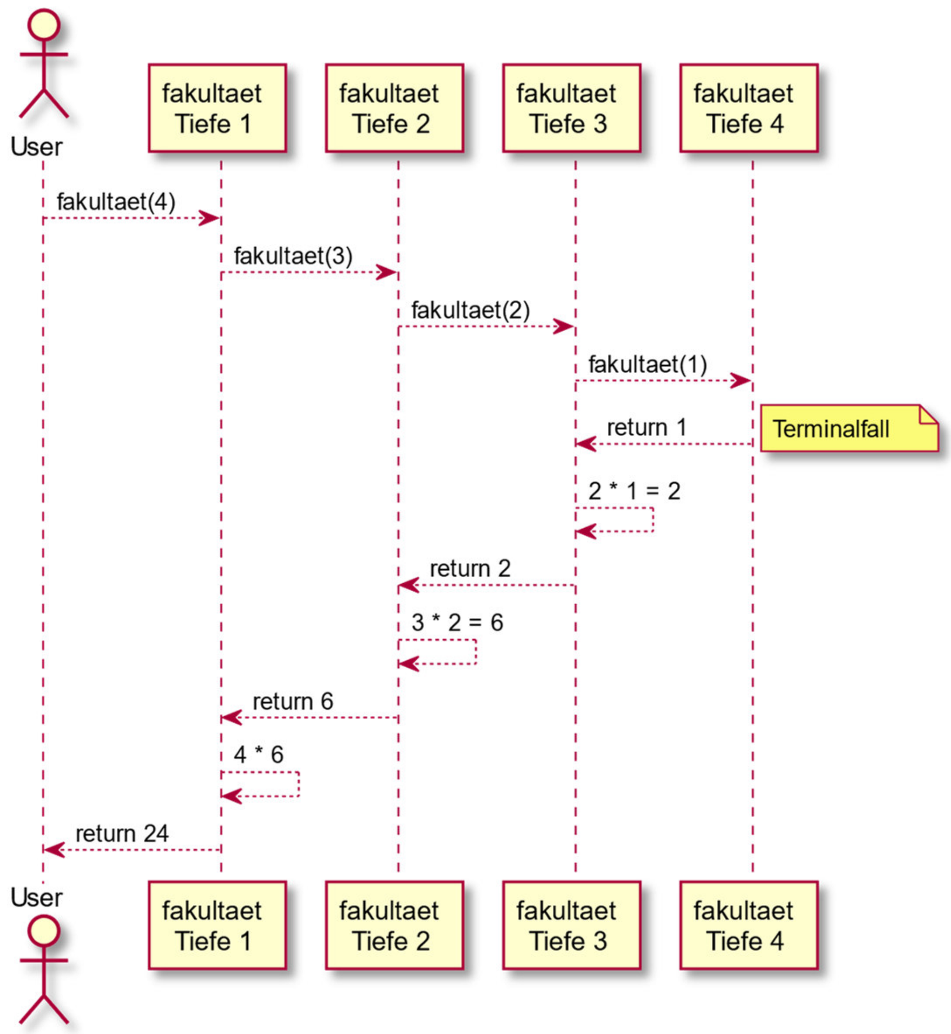
$$n! = \begin{cases} 1 & \text{für n} = 1 \text{ (terminal)} \\ n \cdot (n-1)! & \text{für n} > 1 \text{ (rekursiv)} \end{cases}$$

```java
static int facultyIT(int n) {
    int facu =1;
    // iterative calculation
    for(int i = 1; i<=n; i++)
     {
        facu *= i;
     }
    return facu;
}
```

```java
static int facultyRC(int n) {
    if (n == 1) {
        // rule 1: base (terminal)
        return 1;
     } else {
        // rule 2: recursive
        return n * facultyRC(n - 1);
     }
}
```

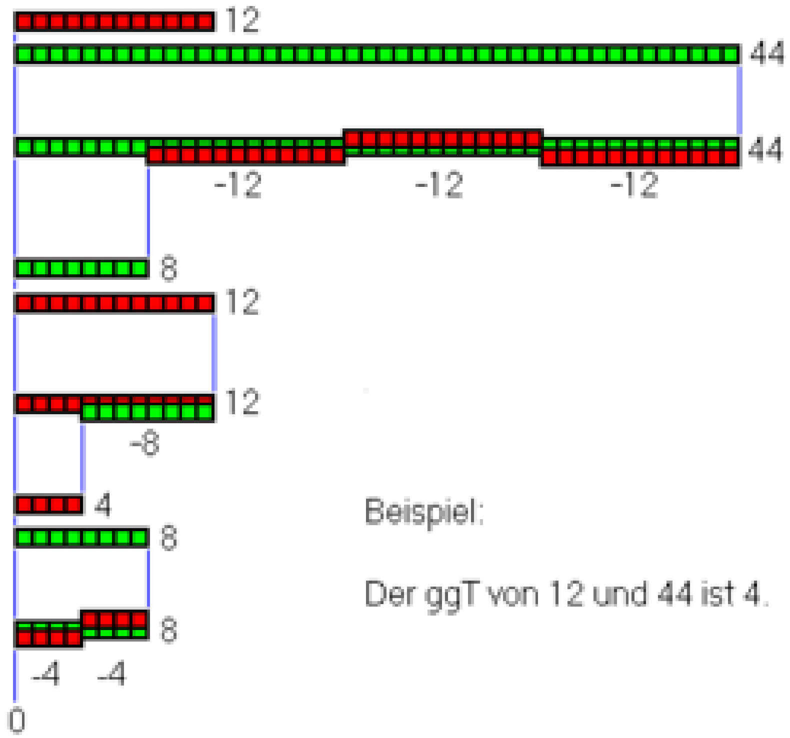
Recursion for faculty schematically

```java
static int facultyRC(int n) {
    if (n == 1) {
        // rule 1: base (terminal)
        return 1;
    } else {
        // rule 2: recursive
        return n * facultyRC(n - 1);
    }
}
```



Greatest common divisor (GCD) iteratively according to Euclid

- **Euclidean algorithm**:
  - We are looking for the common *measurement* for lengths a and b. It must be possible to subtract the two lengths from each other until the *common measurement* remains.




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

    return a;
}
```

Recursion cooking recipe

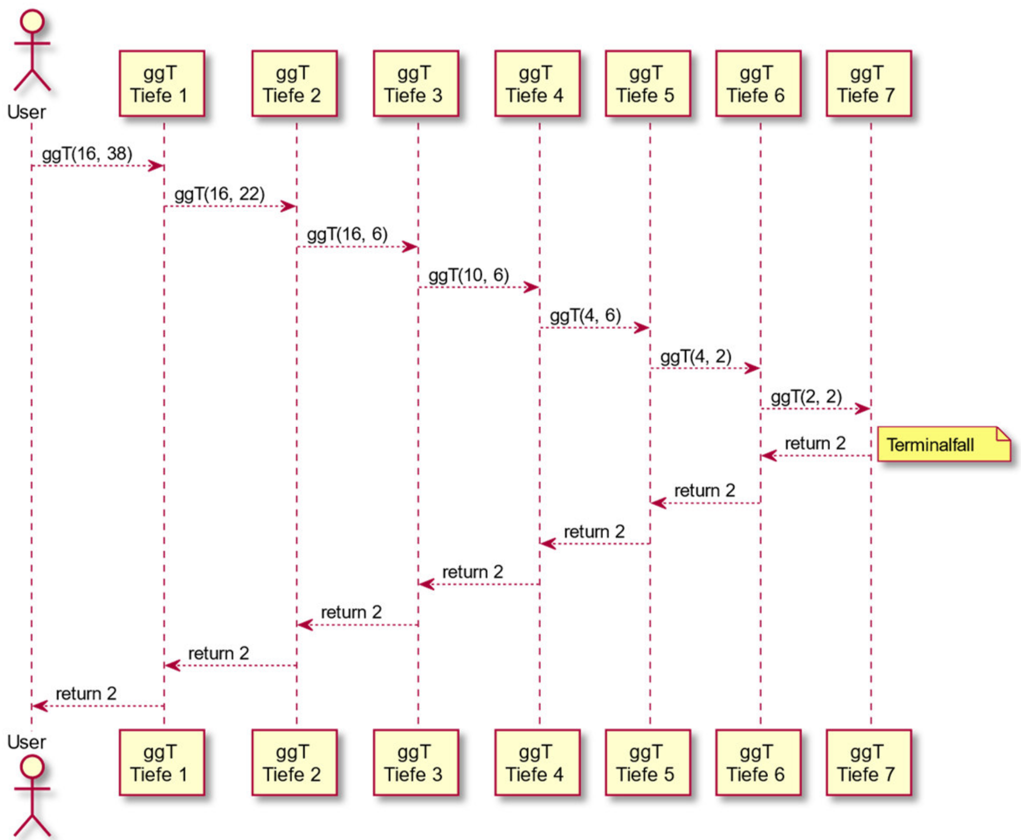
1. Determine base cases (terminating cases). When is the solution trivial?

2. Determine recursive cases. How can I break the problem down into a smaller one?

3. Structure the recursion: do I need a helper method, what must the signature look like, how must the arguments be changed for recursive calls?

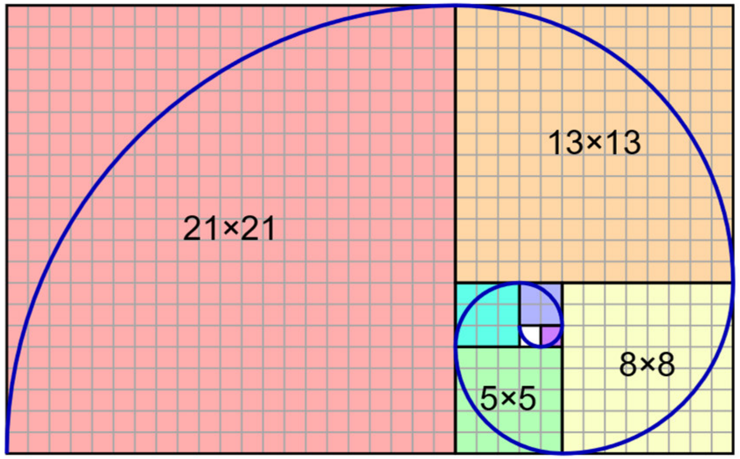
```
// not valid Java...
int recursive(...) {
   if (BaseCase) {
      return /* fixed value */
   } else {
      // recursive case: call recursive at least once!
      return recursive(/* changed arguments*/);
    }
}
```

Greatest common divisor (GCD) recursively schematically




```
static int gcdRC(int a, int b) {
    // base condition (termination cc
    if (b == 0)
        return a;
    // recursive case
    if (a > b)
        return gcdRC(a-b, b);
    return  gcdRC(a, b-a);
}
```

Fibonacci



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

```
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 as simple recursion

```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);
}
```

However, this simple implementation has one disadvantage: in the recursive case, the method is called twice. Even just one call of fib(70) already takes several seconds to several minutes to calculate.

```
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 with 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;
    // already calculated?
  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 with helper function

- A further optimisation of the above recursion would be to take a closer look at the rule:

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

- Accordingly, a value always depends exactly on its two predecessors.
- These we can now also "carry along" as arguments in a helper function.

```java
static int fibBetter(int n) {
  // initialise base cases (terminating cases)
  return fibHelper(n, 0, 1);
}

private static int fibHelper(int n, int a, int b) {
  if (n == 0)     return a;
  else if (n == 1) return b;
    // adjusted parameters!
  else return fibHelper(n-1, b, a+b);
}
```

Palindrome

```
static boolean isPalindromeIT(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 isPalindromeRC(String s) {
  if (s.length() < 2)
    // spaces and single characters are always palindromes
    return true;
  else if (s.charAt(0) != s.charAt(s.length() - 1))
    return false;  // Oops.
  else
    // assuming that first and last match,
        // what about the rest?
    return isPalindromeRC(
                s.substring(1, s.length() - 1));
}
```

Recursion for lists

- If we now want to determine the size of the list, then we must look at the base and recursive cases again.

  - A list that has no first element is empty.
  - If there is a first element, we can then ask it how long it is.
  - With one element, it is at least 1 long; if there is a `next` successor, we have to add the length of the successor as well.

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

  public int size() {
    if (first == null)  return 0; // base case (terminating
case) 1
    else return first.size();     // helper method!
  }

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

Recursion for trees

- Here we can define the size recursively, for example:
  - Base case (terminating case): if there is no root node, then the tree is empty.
  - Recursive case: if there is a root node, then the tree size is at least 1 (base case), plus the size of the left and right subtree (recursion, if there is a left and right subtree).

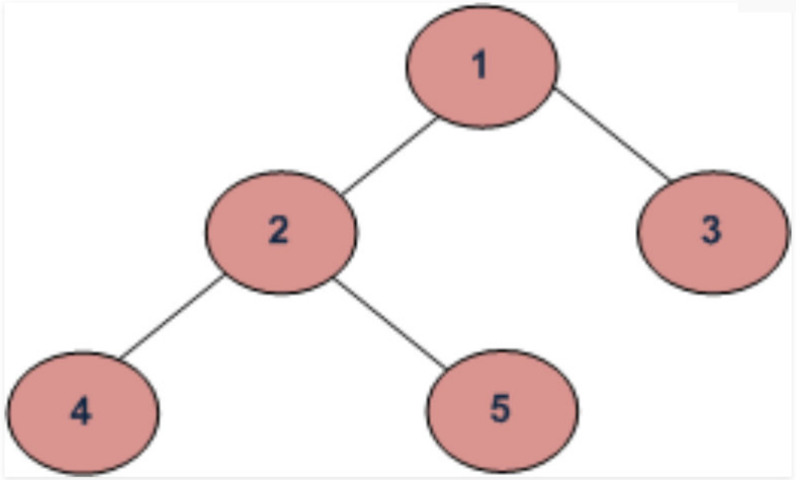
```
public class Tree<T extends Comparable<T>> {
  class Element {
    T value;
        Element left, right;
        Element(T value) { this.value = value; }
    int size() {
      return 1 +
                    (left == null ? 0 : left.size()) +
                    (right == null ? 0 : right.size());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

    Element root;

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

Tree traversals

```
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.
```

Types of recursion

- **Linear** recursion: exactly one recursive call, e.g. Faculty.
- **Repetitive** recursion (tail recursion): special case of linear recursion, in which the recursive call is the last code statement. These tail recursions can be directly converted into an iterative loop (and vice versa). Example: improved implementation of the Fibonacci function.
- **Cascade-like** recursion (tree recursion): multiple recursive calls occur in a branch of case differentiation, which result in an avalanche-like growth of the function calls. Example: simple implementation of the Fibonacci function.
- **Mutual** recursion: a method f() calls a method g(), which in turn calls f() again.

Summary

- A recursive method is a method that calls itself again; characteristics include the absence of `for` and `while`, as well as clear `if-else` instructions, which differentiate between base case and recursive case.

- In cascade-like (tree) recursions, i.e. more than one recursive call per run, caches can make the calculation considerably more efficient, depending on the specific problem.

- Repetitive recursion is desirable, as this can effectively be implemented as a `for` or `while` loop.

- For the above, we often need variables that encode the intermediate results in the recursive call.