Java training

Recursion & recursivity

Session overview

- Recursion - general overview
- Recursion in Java

Recursion

- **Recursion** - programming technique → a method calls itself, to solve a problem
  - A method that uses this technique - **recursive**
- Many programming problems can be solved *only* by recursion
  - Some problems, solvable by other techniques, are better solved by recursion
- The core parts:
  - The *base case*: returns a value without making subsequent recursive calls
    - Done for one or more input values, for which the algorithm can be evaluated
  - The *reduction step*: relates the value of a function on 1+ other input values
    - The sequence of input values *must converge* to the *base case*

Recursion example

Classical example / problem - calculating the factorial of an integer

- Factorial of an integer n - the product of all the integers from 1 to n
  - The factorial of 5 is 120: 5 * 4 * 3 * 2 * 1
- The factorial number:
  - For 1 - is 1
  - For a number *n*: *n* * (the factorial of n−1)
- Recursivity - the definition includes:
  - The factorial method itself
  - The end condition

```
private long factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n-1);
}
```

$n! = n * (n-1) * (n-2) * \ldots \times 2 \times 1$

+ Hands-on →

Recursion - potential problems

- **Missing base case** → missing the base case: may run infinitely

```
public static double harmonic(int n) {
    return harmonic(n-1) + 1.0/n;
}
```

The base case is missing

- **Missing convergence** → defining the recursive call to solve a problem that is not smaller than the original problem

```
public static double harmonic(int n) {
    if (n == 1) return 1.0;
    return harmonic(n) + 1.0/n;
}
```

+ Hands-on →

Recursion - potential problems (continued)

- **Excessive memory consumption** - if a recursive method calls itself for an excessive number of times, the required stack memory may be too high
  - → may lead to a `StackOverflowError`

```
public static double harmonic(int n) {
    if (n == 0) return 0.0;
    return harmonic(n-1) + 1.0/n;
}
```

- **Excessive recomputation** - some recursion algorithms may re-invoke some (already computed) computations again
  - Solution → caching the already computed values

Dynamic programming

- General approach for implementing recursive algorithms → *divide et impera*
  - Dividing the problem into several smaller subproblems
  - Store the solutions of those subproblems
  - Use the stored solutions to solve the initial (big) problem
- Two main approaches:
  - **Top-down** dynamic programming - storing (caching) the result of each solved subproblem + reusing it the next time it would be needed
  - **Bottom up** dynamic programming - computing solutions for all subproblems, starting with the simplest + gradually building solutions for more complicated

Top down dynamic programming - Fibonacci

```java
private static long[] fib = new long[92]; // the 93rd → overflow

public static long fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    // return the cached value (if it was previously computed)
    if (fib[n] > 0) return fib[n];

    // compute and cache a value
    fib[n] = fibonacci(n-1) + fibonacci(n-2);
    return fib[n];
}
```

? Hands-on →

Bottom-up dynamic programming - Fibonacci

```
public static long fibonacci(int n) {
    long[] fib = new long[n+1];
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i <= n; i++) {
        fib[i] = fib[i-1] + fib[i-2];
    }

    return fib[n];
}
```

? Hands-on →

Q&A session

1. You ask, I answer
2. I ask, you answer