Aufgabe 1 – Kontrollstrukturen

Aufgabe 1.1 – Gerade oder ungerade

Schreibe ein Programm, das eine ganze Zahl einliest und überprüft, ob sie gerade oder ungerade ist.

Anforderungen:

- Verwende eine if-else-Struktur.
- Gib eine passende Meldung aus, z. B.:
- „Die Zahl 7 ist ungerade."

Tipp: Eine Zahl ist gerade, wenn zahl % 2 == 0.

Aufgabe 1.2.1 – Notenbewertung mit switch

Schreibe ein Programm, das eine Note (1–5) einliest und eine textuelle Bewertung ausgibt.

Beispielausgabe:

1 → „Sehr gut"
2 → „Gut"
3 → „Befriedigend"
4 → „Ausreichend"
5 → „Nicht bestanden"

Verwende eine **switch-Struktur** und eine **default-Klausel** für ungültige Eingaben.

Aufgabe 1.2.2

Erweitere dein Programm aus Aufgabe 1.2.1 so, dass auch „Komma-Noten" (z. B. 1,3; 2,7; 3,3 …) textuell bewertet werden.
Beispiel:

- 1,0 – 1,3 → „Sehr gut"
- 1,7 – 2,3 → „Gut"
- 2,7 – 3,3 → „Befriedigend"
- 3,7 – 4,0 → „Ausreichend"
- 5,0 → „Nicht bestanden"

Anforderungen:

- Lies eine Note als Kommazahl ein.
- Gib die textuelle Bewertung der Note aus.
- Verwende dazu eine switch-Struktur, die nicht bei jedem case ein break; verwendet.
  → Überlege dir, wie du mehrere Fälle zusammenfassen kannst, um Code zu sparen.
- Gib bei ungültigen Noten eine Fehlermeldung aus.

Aufgabe 2 – for-Schleife vs. Rekursion

Schreibe ein Programm, das die **Fakultät einer Zahl n (n!)** berechnet, also:
$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$n!=n×(n−1)×(n−2)×…×1

Erstelle dazu zwei Methoden:

a. **Iterative Methode:**
   Berechne die Fakultät mit einer **for-Schleife**.

- Gib bei jedem Schleifendurchlauf aus,
  wie oft die Schleife bisher ausgeführt wurde
  und wie der aktuelle **Zwischenstand der Berechnung** ist.

b. **Rekursive Methode:**
   Berechne die Fakultät mit einer **rekursiven Methode**, die sich **selbst aufruft**, bis eine Abbruchbedingung erreicht ist.
   - Gib **vor dem rekursiven Aufruf** aus, was die Methode gerade tut (z. B. „Berechne Fakultät von n …“)
   - Gib **nach dem Aufruf** aus, wenn sie das Teilergebnis wieder zurückgibt (z. B. „Ergebnis für n = …“)

So kannst du beobachten, dass:

- die **Schleife** den Ablauf **von oben nach unten** in fester Reihenfolge durchläuft,
- die **Rekursion** das Problem **schrittweise zerlegt** und die Ergebnisse **beim Zurückkehren** zusammensetzt.

Erklärung:

Eine **for-Schleife** wiederholt einen Codeblock **mehrfach innerhalb derselben Methode**, solange eine Bedingung erfüllt ist.
Eine **rekursive Methode** ruft **sich selbst** auf und löst das Problem in **kleineren Teilaufgaben**, bis eine **Abbruchbedingung** erreicht wird.

Die Rekursion nutzt den **Call Stack** (Aufrufstapel), um sich zu merken, wo sie fortfahren muss.
Dadurch läuft sie **in zwei Phasen** ab:

1. **Zerlegung** (Vorwärtsphase – Aufrufe nach unten)
2. **Zusammensetzung** (Rückwärtsphase – Rückkehr der Ergebnisse)

Fehlt die Abbruchbedingung, führt das zu einem **„StackOverflowError“**, weil sich die Methode unendlich oft selbst aufruft.