TECHNISCHE HOCHSCHULE LÜBECK

Programmieren II

Informatik (B. Sc.)
2. Semester

Unit 01

Einführung in die rekursive Programmierung

Prof. Dr. Nane Kratzke

Units

Rekursion

01 Einführung in die rekursive Programmierung
02 Sequenzbasierte Rekursionen
03 Rekursive Datenstrukturen

Funktionale Programmierung

04 Einführung in die funktionale Programmierung
05 Funktionale Programmierung mit Streams
06 Thinking in Filter - Map - Reduce

Generische Datentypen <T>

07 Einführung in Gen. Datentypen (Type Erasure)
08 Bounded Types

Graphical User Interfaces

09 Einführung in Swing (Typvertreter)
10 Model - View - Controller (MVC)
11 MVC an einer Beispielanwendung

TECHNISCHE HOCHSCHULE LÜBECK

Unit 01

Einführung in die Rekursive Programmierung

Rekursion

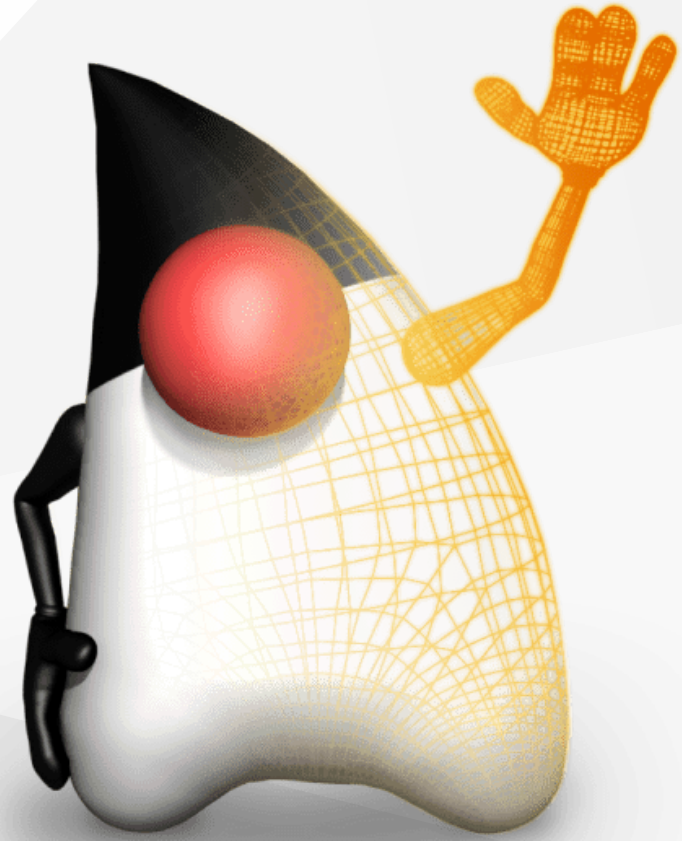
- Definition und Prinzipien der Rekursion
- Unterschiede zwischen rekursiven und iterativen Ansätzen

Wirkungsweise

- Aufbau und Struktur rekursiver Methoden
- Call Stack

Übungsaufgaben

- Beispiele für rekursive Methoden
- Formulierung rekursiver Methoden für einfache Probleme



Rekursion

Basisfall • Rekursiver Mindset • Iteration

Definition und Prinzipien der Rekursion

Was ist Rekursion?

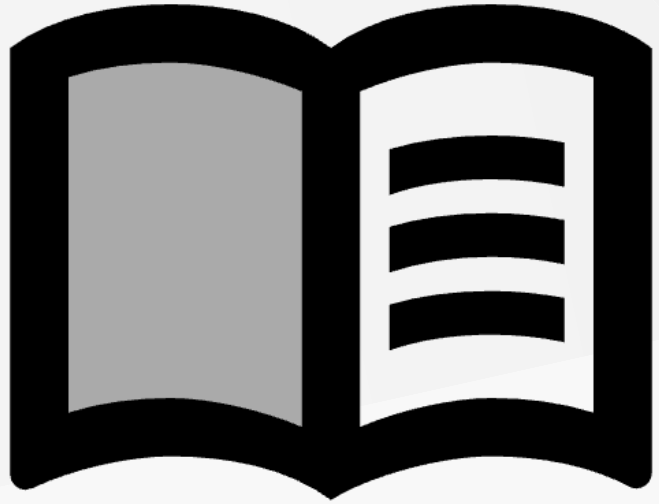
- Eine Funktion (oder in Java eine Methode), die sich selbst aufruft.
- Ermöglicht die Lösung von Problemen durch schrittweise Zerlegung in kleinere Teilprobleme.

Warum ist Rekursion wichtig?

- Entwicklung analytischer Fähigkeiten zur Problemlösung.
- Das Konzept der Rekursion hat eine immense Bedeutung in der Informatik und ihre Anwendung in verschiedenen Algorithmen (z.B. `quicksort`).

Mathematische Grundlagen:

- Viele mathematische Funktionen sind rekursiv definiert (z. B. Fakultät, Fibonacci).
- Die mathematische Perspektive fördert das Verständnis der rekursiven Denkweise bei der Problemlösung informatischer Probleme.



Basisfall *(oder auch Abbruchbedingung)*

Definition

Der Basisfall ist eine Bedingung in einer rekursiven Methode, die sicherstellt, dass die Rekursion zu einem Ende kommt. Er definiert den **einfachsten Fall** des Problems, der direkt gelöst werden kann, ohne eine weitere Rekursion auszulösen.

Bedeutung des Basisfalls:

- Ohne einen Basisfall würde eine rekursive Methode sich unendlich oft selbst aufrufen, was zu einem **Stack Overflow** führen würde.
- Der Basisfall bietet einen konkreten Wert oder eine Bedingung, um die Rekursion zu stoppen.

Beispiel (Fakultät)

Definition der Fakultät (mathematisch):

$$\text{fak}(n) = \begin{cases} 1 & \text{falls } n = 0 \text{ (Basisfall)} \\ n \times \text{fak}(n-1) & \text{falls } n > 0 \end{cases}$$

Realisierung der Fakultät (Java):

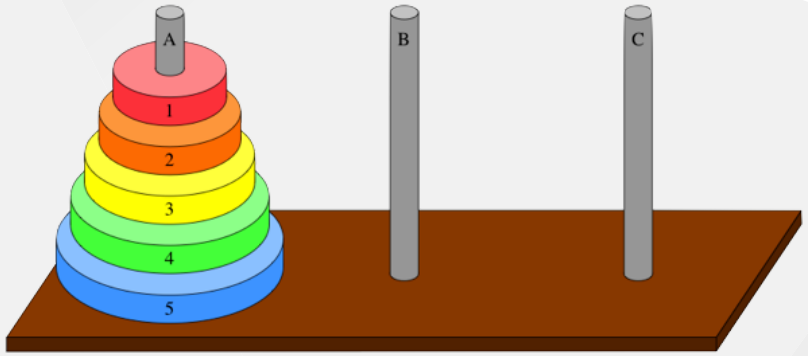
```java
public static int fak(int n) {
    if (n == 0) {  // Basisfall
        return 1;
    }
    return n * fak(n - 1);
}
```

Die Geschichte von den Mönchen *(aka "Die Türme von Hanoi")*

Die Legende *(frei erfunden von Eduard Lucas, 1883)*

- Ein antiker Tempel in **Benares** im Mittelpunkt der Welt *(wo auch sonst)*
- Die Mönche dieses Tempels, haben die Aufgabe, einen Turm aus **64 goldenen Scheiben** *(na klar, was sonst)* zu bewegen.
- Die Mönche müssen die Aufgabe lösen, bevor die **Welt endet** *(der Einsatz muss schon stimmen in einer guten Geschichte)*
- Wenn die letzte Scheibe bewegt wird, wird das Universum nicht in Auflösung übergehen *(Happy End!)*.

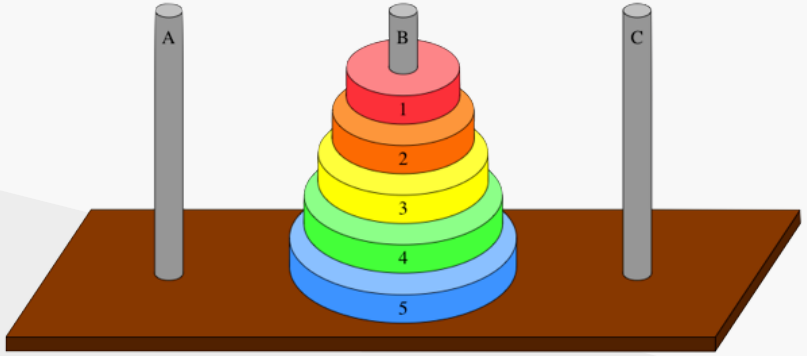
Ausgangszustand



Die Regeln:

- In jedem Zug darf nur je eine Scheibe auf einen anderen Stab gelegt werden.
- Es darf niemals eine größere Scheibe auf einer kleineren liegen.
- Ziel: Die gesamte Anordnung auf einen anderen Stab bewegen, unter der Bedingung, dass die Regeln beachtet werden.

Gewünschter Zielzustand



Bildquelle: https://de.khanacademy.org

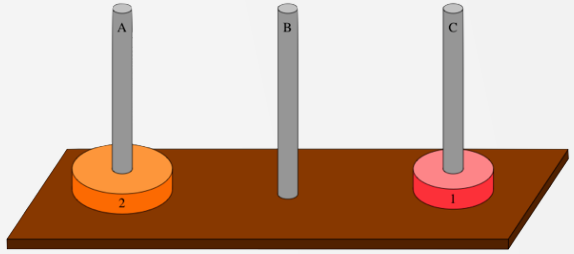
Wie löst man so etwas wie die Türme von Hanoi?

Bei der Implementierung rekursiver Methoden ist es wichtig, zuerst den Basisfall zu identifizieren, bevor man sich auf die rekursive Lösung konzentriert. Die Auswahl eines geeigneten Basisfalls kann oft das gesamte Design der rekursiven Methode positiv beeinflussen.

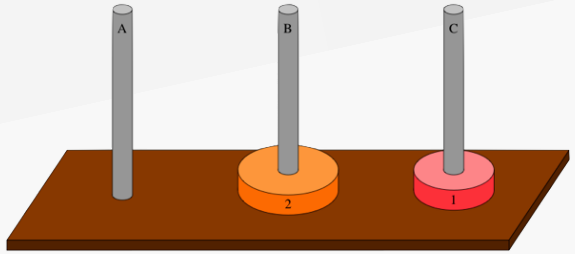
Die Idee des Abst (ältester und weisester Mönch) - sei **dreist**

- Der älteste Mönch gibt die Aufgabe, die obersten $n-1$ Scheiben auf den Zielplatz (B) zu versetzen, dem zweitältesten Mönch.
- Der älteste Mönch muss dann nur noch nach Abschluss des zweitältesten Mönchs die kleineste, letzte und verbleibende Scheibe auf Stab B versetzen **(trivial)**.
- Der zweitälteste Mönch delegiert seinen Teil der Aufgabe analog an den drittältesten Mönch, usw.
- Dieser Prozess setzt sich fort, wobei jeder Mönch die Verantwortung für das Verschieben einer verringerten Anzahl von Scheiben an den nächsten Mönch weitergibt.

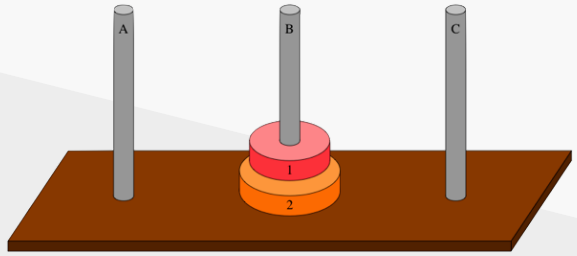
Veranschaulichung für $n = 2$



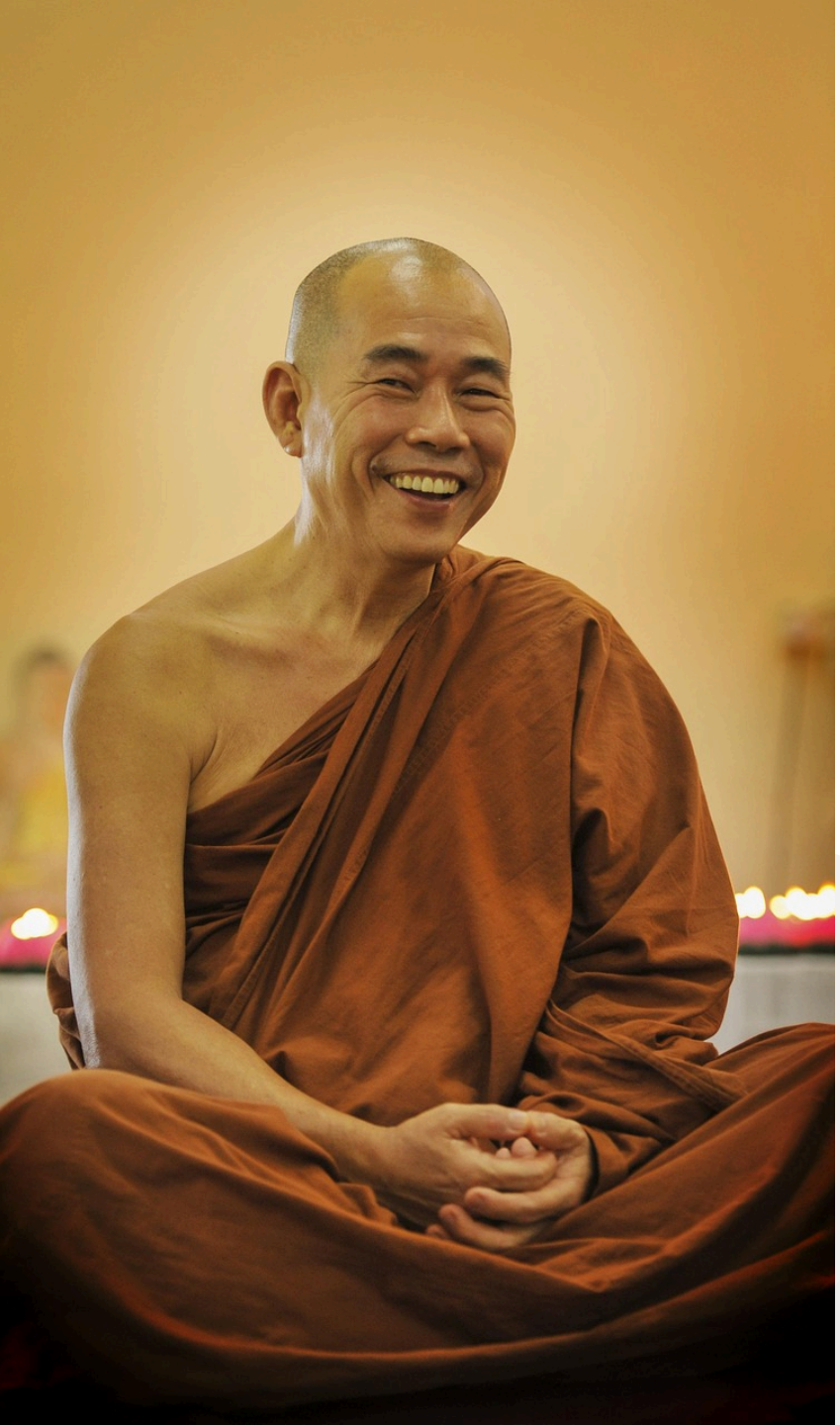
Verschiebe kleine Scheibe von A nach C



Verschiebe große Scheibe von A nach B



Verschiebe kleine Scheibe von C nach B



Die Lehre für das rekursive Programmieren

Sei dreist - sei wie der älteste Mönch.

Rekursiver Mindset!

- Was ist der triviale Teil des Problems (Basisfall)?
- Nur den Teil löse ich!
- Ich ziehe meinen trivialen Teil vom Gesamtproblem ab und
- übergebe dann den Rest des kleineren Problems anderen zur Lösen (sei faul!).

```java
/* Lösung der Türme von Hanoi in Java */

public static void bewege(int i, Stack a, Stack b, Stack c) {
    if (i == 0) return;         // Rekursionsabbruch
    bewege(i - 1, a, c, b);     // Bewege die oberen i-1 Scheiben
    c.push(a.pop());            // Bewege die i-te Scheibe
    bewege(i - 1, b, a, c);     // Bewege die i-1 Scheiben auf den Zielstab
}
```

Rekursionen vs Iterationen

Rekursionen sind eigentlich nur Iterationen in einem anderen Gewand.

- **Rekursion**: Eine Methode ruft sich selbst zur Lösung eines Problems auf.
- **Iteration**: Eine bedingte Wiederholung eines Codesegments (zum Beispiel durch Schleifen)
- Beide Techniken verwenden eine Abbruchbedingung zur Steuerung der Ausführung, um unendliche Wiederholungen zu vermeiden.

Rekursive Implementierung der Fakultät:

```java
public static int fakRekursiv(int n) {
    if (n == 0) return 1; // Basisfall
    return n * fakRekursiv(n - 1);
}
```

Iterative Implementierung der Fakultät:

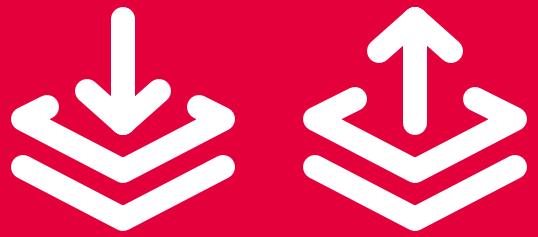
```java
public static int fakIterativ(int n) {
    int result = 1;
    // Iteration
    for (int i = 1; i <= n; i++) result *= i;
    return result;
}
```

Mathematische Rekursion

$$\text{fak}_{rek}(n) = \begin{cases} 1 & \text{falls } n = 0 \text{ (Basisfall)} \\ n \times \text{fak}(n-1) & \text{falls } n > 0 \end{cases}$$

Mathem. Iteration mit $\prod$ (Produkt, analog zu $\sum$ Summe)

$$\text{fak}_{it}(n) = \prod_{k=1}^{n} k = 1 \times 2 \times \ldots \times k \times \ldots \times (n-1) \times n$$



Wirkungsweise

It's the call stack - stupid!

Kochrezept für einfache Rekursionen (Zählparameter)

1. Methodensignatur (Methodenkopf):
   Jede rekursive Methode beginnt mit einer definierten Signatur, die den Rückgabetyp, den Methodennamen und die Parameter enthält. *Nicht anders als bei anderen Methoden auch.*
   Beispiel: `public static int fakultaet(int n)`
2. Basisfall (Abbruchbedingung):
   Der Basisfall ist entscheidend, um eine endliche Rekursion zu gewährleisten. Er definiert den Punkt, an dem die Methode nicht mehr rekursiv aufgerufen wird. Wird normalerweise zu Beginn der Methode geprüft.
   Beispiel: `if (n == 0) return 1; // Basisfall`
3. Rekursiver Aufruf:
   Der rekursive Aufruf erfolgt innerhalb der Methode nach dem Basisfall. Für den Aufruf muss das Problem kleiner gemacht werden, d.h. auf den Basisfall zulaufen (oft $n \Rightarrow n-1$)
   Beispiel: `return n * fakultaet(n - 1); // Rekursiver Aufruf`

```java
public static int fakultaet(int n) {
    if (n == 0) return 1; // Basisfall
    return n * fakultaet(n - 1); // Rek. Aufruf
}
```

```java
public static int fakultaet(int n) {
  return n == 0 ? 1 : n * fakultaet(n - 1);
  // Kompaktere Notation mit bedingter Auswertung
}
```

Call Stack

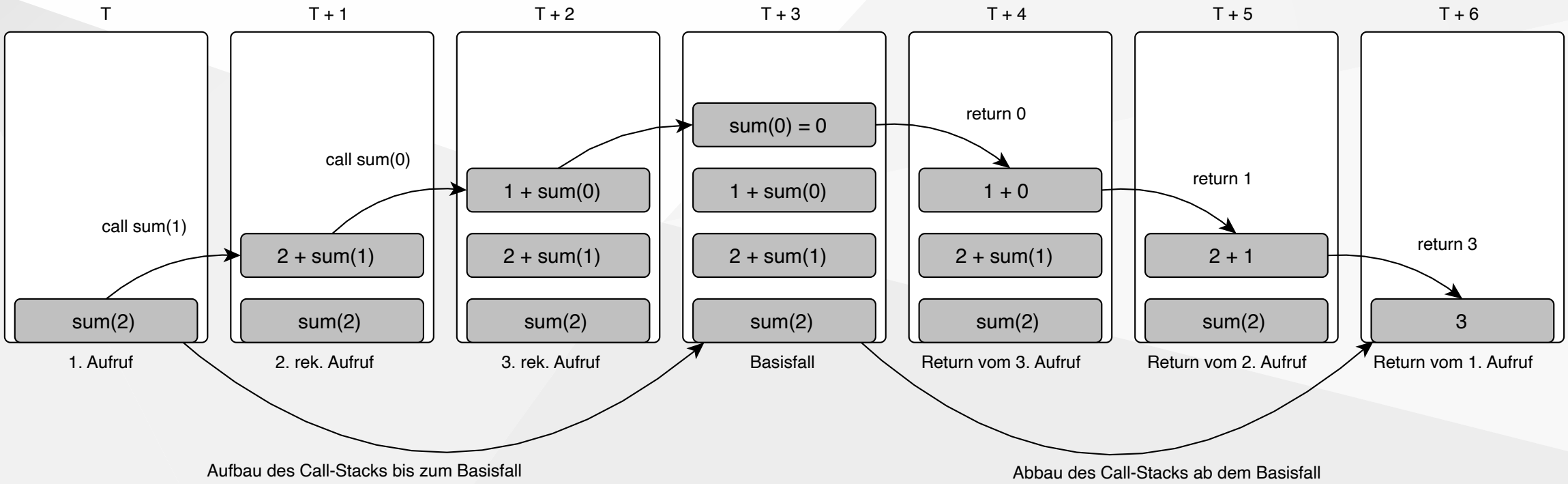
Der Call Stack ist ein grundlegendes Konzept in Programmiersprachen, das beschreibt, wie die Parameter bei Aufrufen von Methoden (oder Funktionen) verwaltet werden.

1. Methodenaufruf: Wenn eine Methode aufgerufen wird, wird ein neuer Stack Frame erstellt, der Eingangsparameter, lokale Variablen und Informationen zur Rückkehradresse speichert.
2. Weitere Methodenaufrufe: Bei weiteren Methodenaufrufen wird ein zusätzlicher Stack Frame für jede neue Methode oben auf den Stapel gelegt.
3. Basisbedingung erreicht: Wenn die Methode die Basisbedingung erreicht (z.B. bei rekursiven Aufrufen), gibt sie einen Wert zurück.
4. Rückgabe und Stapelabbau: Nach der Rückgabe wird der oberste Stack Frame entfernt, und der Programmfluss kehrt zur vorhergehenden Methode zurück. Der Rückgabewert wird an den Aufrufer übergeben.
5. Leerer Stapel: Wenn alle Methoden abgeschlossen sind und der letzte Stack Frame entfernt wurde, ist die Berechnung abgeschlossen.

Beispiel: Rekursive Summe

```java
public static int sum(int n) {
    if (n == 0) return 0; // Basisfall: Summe von 0 ist 0
    return n + sum(n - 1); // Rekursive Berechnung
}
```

Visualisierung des zeitlichen Verlaufs des Call Stacks: `sum(2) => 3`



Übungsaufgaben

Zähle Ziffern • Quersumme • Zähle Ziffern kleiner n • Fibonacci

1. Aufgabe: Ziffern einer Dezimalzahl zählen

Schreiben Sie eine **rekursive Methode**, die die Anzahl an Stellen einer Dezimalzahl bestimmt.

```
int n = count(123); // => 3
n = count(-1); // => 1
n = count(0); // => 1
n = count(424242); // => 6
n = count(-424242); // => 6
```

Lösung:

>>> YOUR-TURN <<<

Ergänzung:

Geben Sie dabei auch den Call Stack auf der Konsole aus, um die Aufruftiefe ihrer Rekursion nachvollziehen zu können. *(Wer vorher mit* `String.length()` *gearbeitet hat, hat nun vermutlich ein Problem ;-)*

2. Aufgabe: Quersumme einer ganzen Zahl

Schreiben Sie eine **rekursive Methode**, die die Quersumme einer ganzen Zahl (Dezimalnotation) bestimmt.

```
int n = querSumme(123); // => 6
n = querSumme(-1); // => 1
n = querSumme(0); // => 0
n = querSumme(424242); // => 18
n = querSumme(-424242); // => 18
```

Lösung:

>>> YOUR-TURN <<<

3. Aufgabe: Zähle Stellen kleiner n in einer ganzen Zahl

Schreiben Sie eine **rekursive Methode**, die die Anzahl der Stellen einer ganzen Zahl (Dezimalnotation) bestimmt, die echt kleiner als n sind.

```
int n = smaller(12345, 3); // => 2
n = smaller(12345, 2); // => 1
n = smaller(-12345, 10); // => 5
n = smaller(424242, 3); // => 3
```

Variante:

Schreiben Sie eine **rekursive Methode**, die die das Verhältnis der Anzahl der Stellen einer ganzen Zahl (Dezimalnotation) bestimmt, die echt kleiner als n sind, zur Gesamtstellenanzahl.

```
double r = smallerRatio(12345, 3); // => 0.4
r = smallerRatio(12345, 2); // => 0.2
r = smallerRatio(-12345, 10); // => 1.0
r = smallerRatio(424242, 3); // => 0.5
```

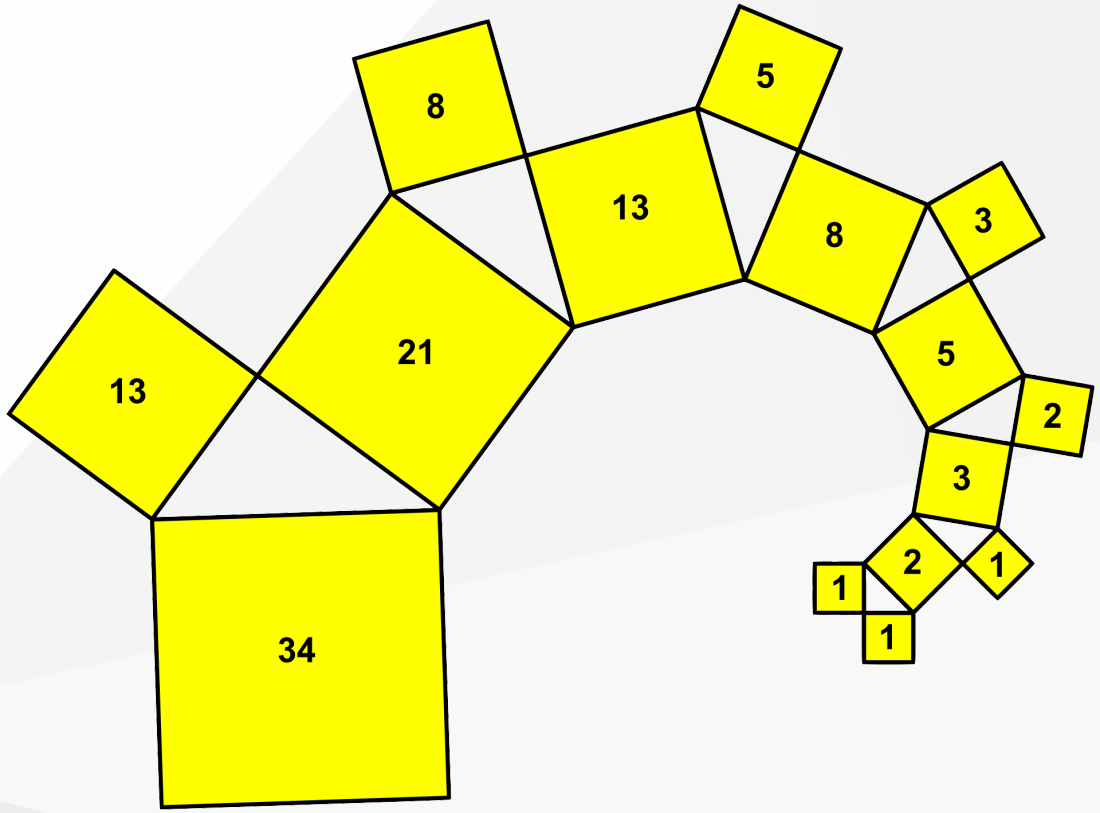
4. Aufgabe: Fibonacci

Recherchieren Sie nun, wie eine Zahl der Fibonacci-Folge berechnet werden kann und implementieren Sie diese in Java.

Geben Sie dabei auch den Call Stack auf der Konsole aus, um die Aufrufstruktur nachvollziehen zu können.

Lösung:

>>> YOUR-TURN <<<



Bildquelle: Wikipedia

Zusammenfassung

- Rekursion
  Eine Programmiertechnik, bei der eine Methode sich selbst aufruft, um ein Problem zu lösen, indem das Problem schrittweise in kleinere Teilprobleme zerlegt wird bis ein trivialer Basisfall erreicht wird.
- Basisfall
  Der Basisfall in der Rekursion ist der spezifische Zustand, bei dem die rekursive Methode nicht mehr sich selbst aufruft, sondern einen unmittelbaren Wert zurückgibt, um den Rekursionsprozess zu beenden.
- Call Stack
  Der Call Stack verwaltet die aktiven Methodenaufrufe eines Programms. Jeder Aufruf erhält einen Stack Frame, der Informationen wie Parameter, lokale Variablen und Rücksprungadressen umfasst, um Rückgaben zu ermöglichen.
- Rekursiver Mindset
  *Sei wie die **Mönche** im Tempel von Benares - faul aber clever!*

TECHNISCHE HOCHSCHULE LÜBECK

Kontakt

Prof. Dr. Nane Kratzke

Technische Hochschule Lübeck

Mail: nane.kratzke(at)th-luebeck.de

Code Strong!

Alle Icons stammen von iconify.design • Bilder von Pixabay

Lösungsvorschlag zur 1. Aufgabe (Ziffern einer Dezimalzahl zählen)

```java
public class Main {

    public static int count(int n) {
        System.out.println("Callstack: count(" + n + ")");
        if (n < 0) return count(Math.abs(n));
        if (n < 10) return 1;
        return 1 + count(n / 10);
    }

    public static void main(String[] args) {
        int n = count(123); // => 3
        n = count(-1); // => 1
        n = count(0); // => 1
        n = count(424242); // => 6
        n = count(-424242); // => 6

        System.out.println(n);
    }
}
```

Lösungsvorschlag zur 2. Aufgabe (Quersumme)

```java
public class Main {

    public static int querSumme(int n) {
        System.out.println("Callstack: querSumme(" + n + ")");
        if (n < 0) return querSumme(Math.abs(n));
        if (n < 10) return n;
        return n % 10 + querSumme(n / 10);
    }

    public static void main(String[] args) {
        int n = querSumme(123); // => 6
        n = querSumme(-1); // => 1
        n = querSumme(0); // => 1
        n = querSumme(424242); // => 18
        n = querSumme(-424242); // => 18

        System.out.println(n);
    }
}
```

Lösungsvorschlag zur 3. Aufgabe (Stellen kleiner als)

```java
public class Main {
  public static int smaller(int n, int t) {
    if (n < 0) return smaller(Math.abs(n), t);
    if (n < 10) return n < t ? 1 : 0;
    return (n % 10 < t ? 1 : 0) + smaller(n / 10, t);
  }

  public static double smallerRatio(int n, int t) {
    return (double)smaller(n, t) / smaller(n, 10); // Rückführung auf smaller
  }

  public static void main(String[] args) {
    int n = smaller(12345, 3); // => 2
    System.out.println(n);

    double r = smallerRatio(12345, 3); // => 0.4
    System.out.println(r);
  }
}
```

Lösungsvorschlag zur 4. Aufgabe (Fibonacci)

Beachten Sie den Anstieg der Callstack-Länge bei größer werdendem i.

```java
public class Main {

  public static int fib(int n) {
    System.out.println("Callstack: fib(" + n + ")");
    if (n == 1) return 1;
    if (n == 2) return 1;
    return fib(n - 1) + fib(n - 2);
  }

  public static void main(String[] args) {
    for (int i = 1; i < 10; i++) {
      System.out.println("Berechne fib(" + i + ")");
      System.out.println(fib(i));
    }
  }
}
```