

Rekursion

Einführung in die Programmierung 1

Sommersemester 21

TU Wien

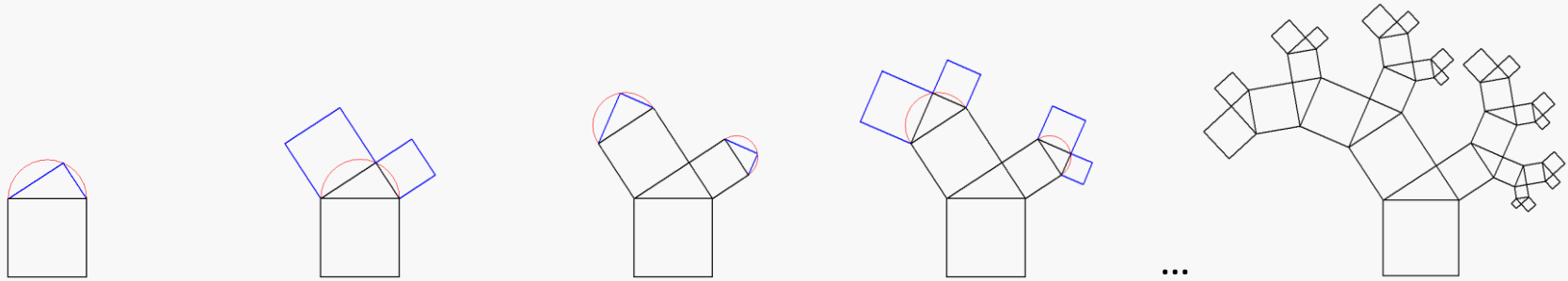
Überblick

- Einführung
- Lineare Rekursion
- Verzweigte Rekursion
- Verschachtelte Rekursion
- Wechselseitige Rekursion
- Terminierung
- Rekursion und Stack-Speicher
- Vergleich von Rekursion und Iteration

Einführung

Rekursion

- Allgemein
 - Abstrakter Vorgang, bei dem Regeln auf ein Produkt, das sie hervorgebracht haben, von neuem angewendet werden



<https://de.wikipedia.org/wiki/Pythagoras-Baum>

- Grundprinzip für Problemlösen
 - Zurückführen einer allgemeinen Aufgabe auf eine **einfachere gleichartige** Aufgabe

Rekursion (ein einfaches Beispiel)

- Berechnung der Summe der Zahlen von 1 bis n
- **Iterativ** ist die Summe definiert durch

$$\text{sum}(n) = 1 + 2 + \dots + n$$

- **Rekursiv** ist die Summe definiert durch

$$\text{sum}(n) = \begin{cases} 1 & \text{falls } n = 1 \\ \text{sum}(n-1) + n & \text{sonst } (n > 1) \end{cases}$$

Basisfall

Rekursionsschritt

Rekursionsschritt

- **Selbstaufufruf** (Zerlegung auf kleineres Problem)
 - Jeder Aufruf muss in irgendeiner Form einen Beitrag zum Erreichen der Abbruchbedingung leisten (Fortschritt)
- **Problem lösen**
 - Ergebnis des Selbstaufufrufs verwenden (in Operationen),
 - Selbstaufufruf entsprechend parametrisieren, usw.

$$\text{sum}(n) = \begin{cases} 1 & \text{falls } n = 1 \\ \text{sum}(\mathbf{n - 1}) + \mathbf{n} & \text{sonst } (n > 1) \end{cases}$$

Basisfall

- Beschreibt triviales Problem
 - Kann sofort „gelöst“ werden
 - Z. B. Rückgabe eines sinnvollen Wertes
 - Kein Selbstaufruf

$$\text{sum}(n) = \begin{cases} \mathbf{1} & \textbf{falls } n = 1 \\ \text{sum}(n - 1) + n & \textit{sonst } (n > 1) \end{cases}$$

Einfache Klassifikation

- Klassifikation nach der Anzahl und Anordnung der rekursiven Aufrufe in einem Rekursionsschritt
 - Lineare Rekursion
 - Verzweigte Rekursion
 - Verschachtelte Rekursion
 - Wechselseitige Rekursion

Lineare Rekursion

Lineare Rekursion

- In jedem Rekursionsschritt gibt es genau einen rekursiven Aufruf
- Beispiel rekursive Summenberechnung

$$\text{sum}(n) = \begin{cases} 1 & \text{falls } n = 1 \\ \text{sum}(n-1) + n & \text{sonst } (n > 1) \end{cases}$$

Basisfall

Rekursionsschritt

- Realisierung in Java
 - Eine Methode die sich selbst wieder aufruft

Beispiel (Summe, rekursiv und iterativ)

```
public class SumTest {  
  
    // num >= 1  
    private static int sumRecursive(int num) {  
        if (num <= 1) {  
            return 1;  
        } else {  
            return sumRecursive(num - 1) + num;  
        }  
    }  
}
```

Rekursiver Aufruf

```
    // num >= 1  
    private static int sumIterative(int num) {  
        int sum = 1;  
        for (int i = 2; i <= num; i++) {  
            sum += i;  
        }  
        return sum;  
    }  
}
```

...

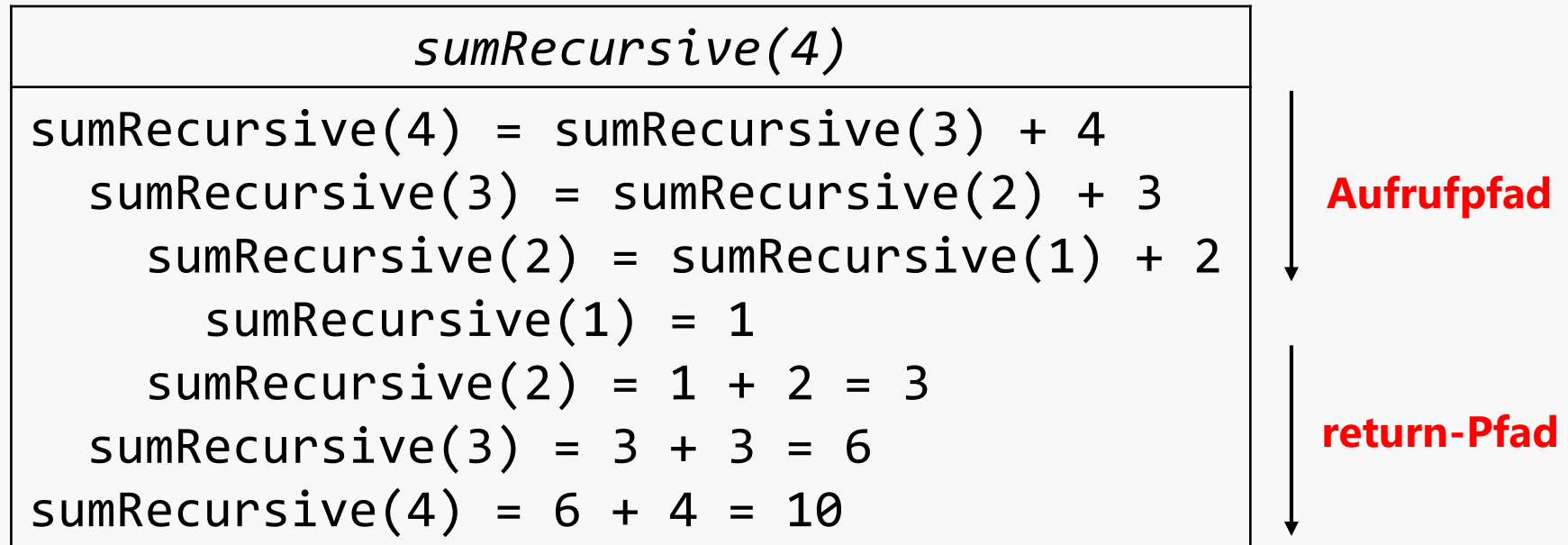
```
...  
public static void main(String[] args) {  
    System.out.println(sumIterative(10));  
    System.out.println(sumRecursive(10));  
    System.out.println(sumIterative(100));  
    System.out.println(sumRecursive(100));  
}
```

}

55
55
5050
5050

Rekursive Implementierung – Ablauf

- Ablauf der Rekursion für $n = 4$



Berechnung erfolgt
im return-Pfad

Position des rekursiven Aufrufes

```
public class RecursiveOutputTest {  
  
    // n >= 1  
    private static void output(int n) {  
        if (n < 1) {  
            System.out.print("Numbers ");  
        } else {  
            System.out.print(n + " ");  
            output(n - 1);  
        }  
    }  
  
    // n >= 1  
    private static void output2(int n) {  
        if (n < 1) {  
            System.out.print("Numbers ");  
        } else {  
            output2(n - 1);  
            System.out.print(n + " ");  
        }  
    }  
}
```

Ausgabe im Aufrufpfad

Ausgabe im return-Pfad

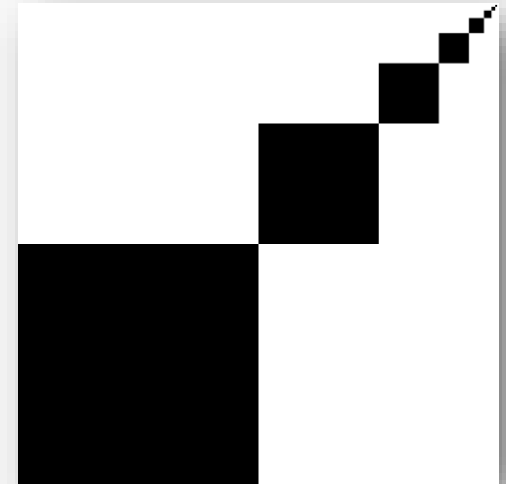
```
...  
public static void main(String[] args) {  
    output(5);  
    System.out.println();  
    output2(5);  
}
```

5 4 3 2 1 Numbers
Numbers 1 2 3 4 5

Rekursion und grafische Ausgabe

```
public class RecursiveSquares {  
  
    private static void drawSquares(double x, double y, double length) {  
        if (length > 1) {  
            double halfLength = length / 2.0;  
            StdDraw.filledSquare(x + halfLength, y + halfLength, halfLength);  
            drawSquares(x + length, y + length, halfLength);  
        }  
    }  
  
    public static void main(String[] args) {  
        int size = 500;  
        StdDraw.setCanvasSize(size, size);  
        StdDraw.setScale(0, size);  
        drawSquares(0, 0, size / 2.0);  
    }  
}
```

Basisfall (nicht zeichnen) ist hier implizit



Rekursion – Rückgabe verwenden

```
public class SqueezeString {
```

Entfernt rekursiv alle Vorkommnisse vom Zeichen c aus dem String s und liefert einen neuen String zurück

```
    // s != null
```

```
    private static String squeeze(String s, char c) {  
        if (s.isEmpty()) {  
            return s;  
        }  
        if (s.charAt(0) == c) {  
            return squeeze(s.substring(1), c);  
        }  
        return s.charAt(0) + squeeze(s.substring(1), c);  
    }  
}
```

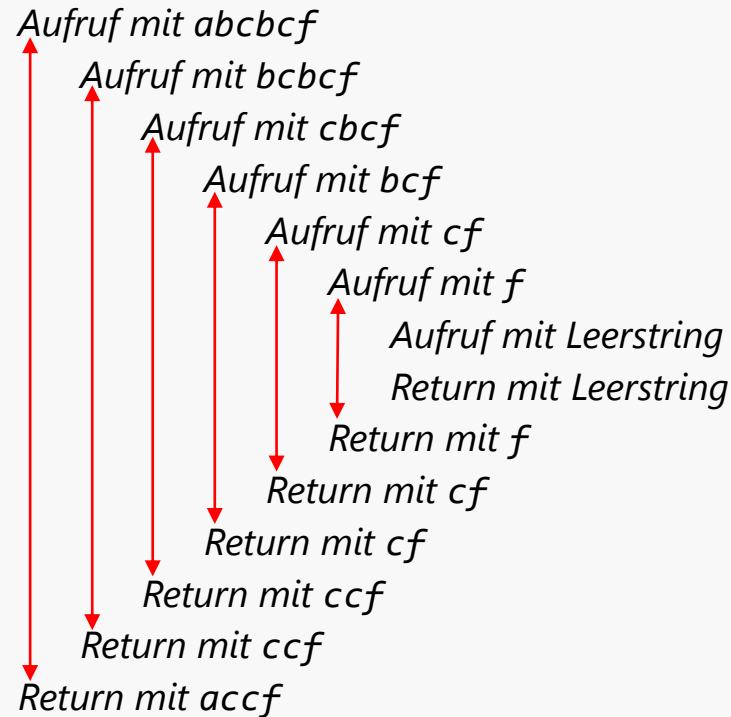
```
    public static void main(String[] args) {  
        System.out.println(squeeze("abcbcbdbf", 'b'));  
        System.out.println(squeeze("Hallo EP1 World", 'l'));  
        System.out.println(squeeze("aaaa", 'a'));  
        System.out.println(squeeze("a b c d", ' '));  
    }  
}
```

accdf
Hao EP1 Word

abcd

Rekursion – Rückgabe verwenden (Beispiel)

- Aufruf mit String *abcbcf* und Zeichen *b*
- Ablauf



```
// s != null
private static String squeeze(String s, char c) {
    if (s.isEmpty()) {
        return s;
    }
    if (s.charAt(0) == c) {
        return squeeze(s.substring(1), c);
    }
    return s.charAt(0) + squeeze(s.substring(1), c);
}
```


Endrekursion (tail recursion)

- Jeder rekursive Aufruf ist die letzte Aktion der Methode
- Rekursiver Aufruf ersetzt damit den Originalaufruf mit angepassten Argumenten

Beispiel für Endrekursion (ggT-Berechnung)

```
public class GCDTest {  
  
    // a > 0 && b > 0  
    private static int gcdRecursive(int a, int b) {  
        if (a == b) {  
            return a;  
        } else if (a > b) {  
            return gcdRecursive(a - b, b);  
        } else {  
            return gcdRecursive(a, b - a);  
        }  
    }  
  
    // a > 0 && b > 0  
    private static int gcdIterative(int a, int b) {  
        while (a != b) {  
            if (a > b) a -= b;  
            else b -= a;  
        }  
        return a;  
    }  
    ...  
}
```

```
...  
public static void main(String[] args) {  
    System.out.println(gcdIterative(24, 16));  
    System.out.println(gcdRecursive(24, 16));  
    System.out.println(gcdIterative(4399, 166));  
    System.out.println(gcdRecursive(4399, 166));  
}  
}
```

8
8
83
83

Endrekursion mit zusätzlichen Parametern

- Rekursiver Aufruf in einer Berechnung ist nicht endrekursiv
 - Beispiel Summenberechnung
- So ein Aufruf kann aber umgeschrieben werden
 - Zusätzliche überladene Methode
 - Zusätzliche Parameter bei dieser Methode
 - Akkumulierende Parameter

Endrekursion – Beispiel (Summenberechnung)

- Beispiel mit 2 Methoden

- Eine Methode mit ursprünglicher Signatur (nur ein Parameter)
- Eine überladene Methode mit zusätzlichem Parameter

```
private static int sumRecursive2(int sum, int n) {  
    if (n <= 1) {  
        return sum;  
    } else {  
        return sumRecursive2(sum + n, n - 1);  
    }  
}
```

Endrekursive Form der
Summenberechnung

```
// num >= 1  
private static int sumRecursive2(int num) {  
    return sumRecursive2(1, num);  
}
```

Methode mit
ursprünglicher Signatur

Endrekursion – Ablauf

- Ablauf der Rekursion für $n = 4$

<i>Endrekursion sumRecursive2(4)</i>	
sumRecursive2(4)	= sumRecursive2(1, 4)
	= sumRecursive2(5, 3)
	= sumRecursive2(8, 2)
	= sumRecursive2(10, 1)
	= 10

Berechnung im Aufrufpfad - im return-Pfad wird nur das fertig berechnete Ergebnis zurückgegeben

Sinn der Endrekursion

- Endrekursive Methoden lassen sich schnell in iterative überführen
- Manche Compiler erkennen Endrekursion und entfernen diese automatisch
 - Daher rekursiv programmieren und die Umsetzung dem Compiler überlassen
 - Iteration kann effizienter sein bzw. für größere Datenmenge noch funktionieren

Verzweigte Rekursion

Verzweigte Rekursion

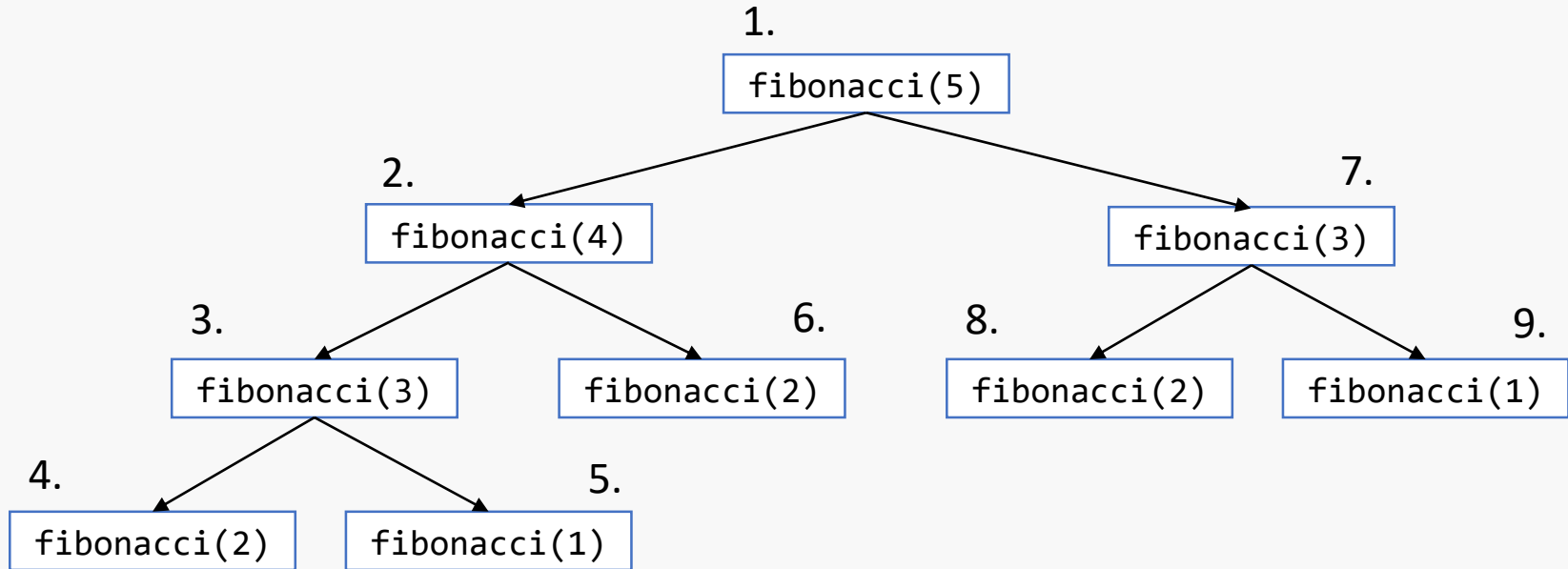
- Jeder nichtterminierende rekursive Aufruf erzeugt zwei oder mehrere Aufrufe
 - Es entsteht ein Aufrufbaum
- Einfaches Beispiel – Fibonacci-Zahlen
 - $\text{fibonacci}(1) = 1$
 - $\text{fibonacci}(2) = 1$
 - $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$ für $n > 2$

Beispiel (Fibonacci-Zahlen)

```
public class FibonacciTest {  
  
    // n >= 1  
    private static long fibonacci(int n) {  
        if (n <= 2) {  
            return 1;  
        } else {  
            return fibonacci(n - 1) + fibonacci(n - 2);  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.println(fibonacci(1));  
        System.out.println(fibonacci(3));  
        System.out.println(fibonacci(10));  
    }  
}
```

1
2
55

Aufrufbaum für fibonacci (5)



- Anzahl der Aufrufe wächst schnell an!

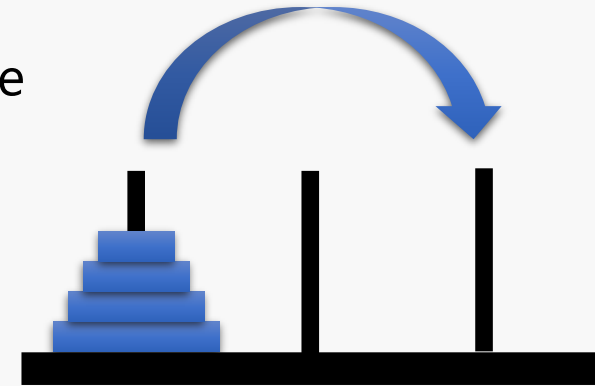
Beispiel (Binomialkoeffizient)

$$\text{binom}(n, k) = \begin{cases} 1 & \text{falls } k = 0 \text{ oder } n = k \\ \text{binom}(n - 1, k - 1) + \text{binom}(n - 1, k) & \text{sonst} \end{cases}$$

```
// 0 <= k <= n
private static long binomialCoefficient(int n, int k) {
    if (k == 0 || n == k) {
        return 1;
    }
    return binomialCoefficient(n - 1, k - 1) + binomialCoefficient(n - 1, k);
}
```

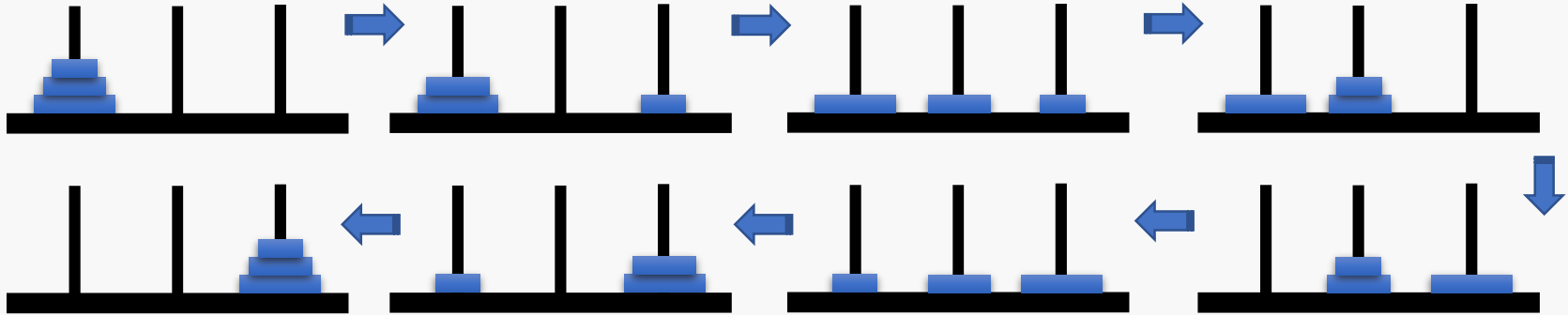
Türme von Hanoi

- Ausgangssituation
 - Turm mit n Scheiben auf dem linken Stab von drei Stäben
 - Jede Scheibe liegt auf einer größeren Scheibe
- Aufgabe
 - Den Turm auf den rechten Stab bringen
- Einschränkungen
 - Nur die oberste Scheibe darf bewegt werden
 - Es darf nicht eine Scheibe auf eine kleinere Scheibe gelegt werden



Türme von Hanoi – Ablauf

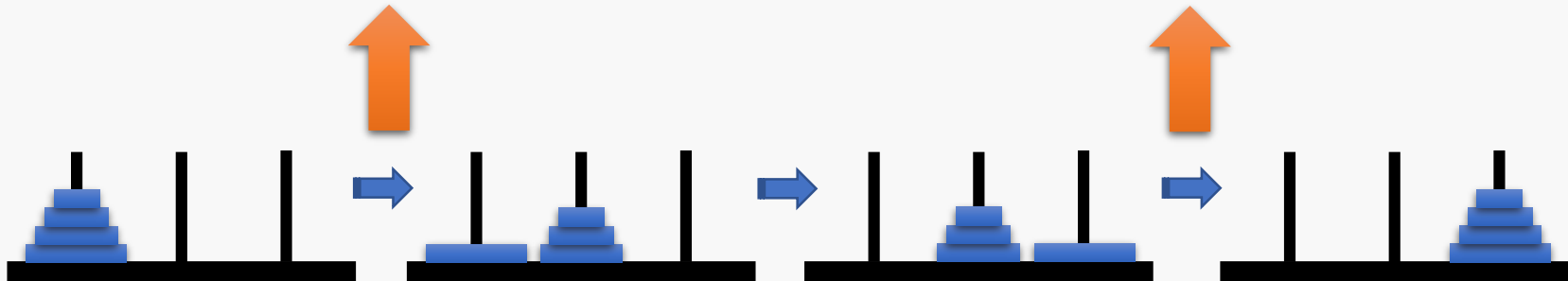
- $n = 3$



- $n = 4$

$n=3$

$n=3$



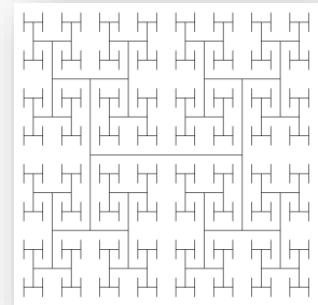
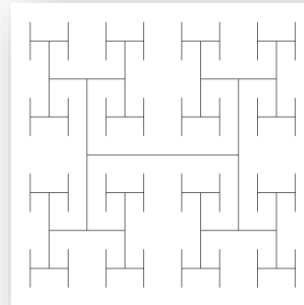
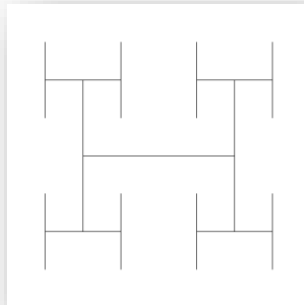
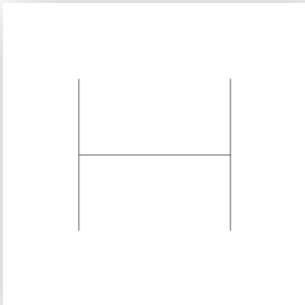
Türme von Hanoi – Implementierung

```
public class TowersOfHanoi {  
  
    private static void solveHanoi(int n, String a, String b, String c) {  
        if (n > 0) {  
            solveHanoi(n - 1, a, c, b);  
            System.out.println(a + " --> " + c);  
            solveHanoi(n - 1, b, a, c);  
        }  
    }  
  
    public static void main(String[] args) {  
        solveHanoi(3, "Tower 1", "Tower 2", "Tower 3");  
    }  
}
```

Tower 1 --> Tower 3
Tower 1 --> Tower 2
Tower 3 --> Tower 2
Tower 1 --> Tower 3
Tower 2 --> Tower 1
Tower 2 --> Tower 3
Tower 1 --> Tower 3

Verzweigte Rekursion und Grafik (H-Baum)

- H-Baum n-ter Ordnung (Htree.java in TUWEL)
 - Ein H mit drei Linien zeichnen
 - Rekursionsschritt
 - Vier H-Bäume für $n - 1$ zeichnen
 - Jeweils mit den Spitzen des H-Baums für n verbinden
 - Jeder H-Baum für $n - 1$ ist nur halb so groß wie der H-Baum für n (Basisfall, d. h. nichts zeichnen, wenn $n \leq 0$)
- H-Baum der Ordnung 1, 2, 3, 4



Verschachtelte Rekursion

Verschachtelte Rekursion

- Argumente bei einem rekursiven Aufruf sind selbst wieder Rekursionsaufrufe
- Vorhergehender Aufruf muss abgeschlossen sein, bevor ein weiterer gestartet werden kann

Beispiel (Moduloberechnung)

```
public class ModuloTest {  
  
    // a >= 0, b > 0  
    private static int modulo(int a, int b) {  
        if (a < b) {  
            return a;  
        } else if (a < 2 * b) {  
            return a - b;  
        } else {  
            return modulo(modulo(a, 2 * b), b);  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.println(modulo(3, 5));  
        System.out.println(modulo(14, 4));  
        System.out.println(modulo(426374, 485));  
    }  
}
```

3
2
59

Beispiel (Ackermann-Funktion)

```
public class AckermannTest {  
  
    // m >= 0, n >= 0  
    private static long ackermann(long m, long n) {  
        if (m == 0) {  
            return n + 1;  
        } else if (n == 0) {  
            return ackermann(m - 1, 1);  
        } else {  
            return ackermann(m - 1, ackermann(m, n - 1));  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.println(ackermann(1, 0));  
        System.out.println(ackermann(2, 1));  
        System.out.println(ackermann(2, 2));  
        System.out.println(ackermann(3, 3));  
    }  
}
```

2
5
7
61

Ackermann-Funktion – Verwendung

- Benchmarks
 - Anzahl der Aufrufe wächst extrem schnell
- Analyse von Algorithmen
- Wichtige Funktion in der theoretischen Informatik

Wechselseitige Rekursion

Wechselseitige Rekursion

- Der rekursive Aufruf erfolgt indirekt
 - Es können auch mehr als zwei Methoden an der Rekursion beteiligt sein
 - Wird auch als indirekte Rekursion oder alternierende Rekursion bezeichnet
- Beispiel
 - Methode a ruft Methode b auf
 - Methode b ruft wieder Methode a auf
- Hinweis
 - Beim Lösen eines Problems P werden rekursive Lösungen für Subprobleme verwendet, die selbst nicht kleinere Instanzen von P sind

Beispiel (gerade/ungerade überprüfen)

```
public class EvenOddTest {  
  
    private static boolean even(int n) {  
        if (n == 0) {  
            return true;  
        }  
        return odd(n - 1);  
    }  
  
    private static boolean odd(int n) {  
        if (n == 0) {  
            return false;  
        }  
        return even(n - 1);  
    }  
  
    public static void main(String[] args) {  
        System.out.println(even(1));  
        System.out.println(even(2));  
        System.out.println(odd(3));  
        System.out.println(odd(4));  
    }  
}
```

false
true
true
false

Terminierung

Rekursion und Induktion

- Vollständige Induktion
 - Beweisverfahren, das in der Mathematik häufig eingesetzt wird
 - Bildet die Basis für Rekursion (ähnliches Verfahren)
- Induktion
 - Um eine Aussage $A(n)$ für alle natürlichen Zahlen $n \geq m$ zu beweisen, genügen zwei Beweisschritte
 1. *Induktionsanfang*
 - Beweise $A(m)$
 2. *Induktionsschritt*
 - Beweise $A(n+1)$ unter der Voraussetzung, dass $A(n)$ mit $n \geq m$ gilt
 - Es muss also gezeigt werden, dass (für $n \geq m$) gilt: $A(n) \Rightarrow A(n+1)$
 - Sehr oft wird für m die Zahl 0 oder 1 gewählt

Reguläre Terminierung

- Beendigung eines Programms, einer Schleife oder eines rekursiven Methodenaufrufs
 - ohne Fehlermeldung
 - ohne fehlerhaftes Programmverhalten
 - Überlauf des Wertebereichs eines Parameters
- Beispiele für Methoden, die (meist) nicht regulär terminieren

```
private static void endless() {  
    endless();  
}  
  
private static void overflow(int x) {  
    if (x * x > 0) {  
        overflow(x * x);  
    }  
}
```

Fundiertheit und Fortschritt

- Fundiertheit

- Eine rekursive Methode ist **fundiert**, wenn es eine erreichbare Abbruchbedingung gibt
- Hängt mit dem Induktionsanfang zusammen

- Fortschritt

- Methodenausführung erzielt einen **Fortschritt**, wenn sie uns näher an die Abbruchbedingung bringt
- Hängt mit dem Induktionsschritt zusammen

Reguläre Terminierung

- Reguläre Terminierung setzt Fundiertheit und einen Fortschritt um einen Mindestbetrag in jedem Rekursionsschritt voraus

Beispiele (Fundiertheit und Fortschritt)

```
private static void endless() {  
    endless();  
}
```

Nicht fundiert

```
private static void overflow(int x) {  
    if (x * x > 0) {  
        overflow(x * x);  
    }  
}
```

Fundiert, aber
Fortschritt fehlerhaft

Terminierung analysieren

- Vereinfachte Annahmen
 - Stack-Speicher geht nicht aus
 - Kein arithmetischer Überlauf
- Zunächst geht es nur um die theoretisch korrekte Implementierung
 - Fundiertheit
 - Fortschritt

Beispiele (Terminierung **theoretisch** analysieren)

```
private static int f(int x) {  
    return x <= 0 ? 0 : f(x - 2) + 3;  
}
```

Terminiert immer

```
private static int g(int x) {  
    return x == 0 ? 0 : g(x + 2) + 3;  
}
```

Terminiert nur für gerade
Parameterwerte ≤ 0

```
private static int h(int x) {  
    return x < 0 ? 0 : h(x / 2) + 3;  
}
```

Terminiert nur für negative
Parameterwerte

```
private static int k(int x) {  
    return x <= 0 ? 0 : k(x / 2) + 3;  
}
```

Terminiert immer

Praktisches Beispiel (Fakultät)

- Fakultät
$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

- Kompakte Implementierung

```
// n >= 0
private static long factorial(int n) {
    return n <= 0 ? 1 : n * factorial(n - 1);
}
```

- Fundiertheit und Fortschritt
 - Terminiert theoretisch
 - Aber praktisch (Java) nur korrekte Werte für $0 \leq n < 21$

Fakultät – redundanter Basisfall

- Redundante Implementierung

```
private static long factorialRedundant(int n) {  
    return n <= 0 || n == 1 ? 1 : n * factorialRedundant(n - 1);  
}
```

- Terminierung
 - Ähnlich zu vorheriger Implementierung
- Aber
 - Ein Basisfall zu viel
 - Kann auch mit ursprünglichem Basisfall berechnet werden
 - Redundanter Basisfall sollte vermieden werden
 - Bringt nicht viel und verwirrt

Fakultät – falscher Basisfall

- Abgekürzte Implementierung

```
private static long factorialMissingBaseCase(int n) {  
    return n == 1 ? 1 : n * factorialMissingBaseCase(n - 1);  
}
```

- Terminierung
 - Terminiert für $n > 0$
 - Funktioniert nicht für $n \leq 0$ (theoretisch Endlosrekursion)

Fakultät – kein Basisfall

- Kein Basisfall

```
private static long factorialNoBaseCase(int n) {  
    return n * factorialNoBaseCase(n - 1);  
}
```

- Terminierung
 - Nicht fundiert
 - Ruft sich immer wieder auf (theoretisch Endlosrekursion)

Fakultät – falscher rekursiver Aufruf

- Beispiel Fakultät

```
private static long factorialNoConvergence(int n) {  
    return n <= 0 ? 1 : factorialNoConvergence(n) * n;  
}
```

- Terminierung
 - Fundiert
 - Fehlerhafter Fortschritt

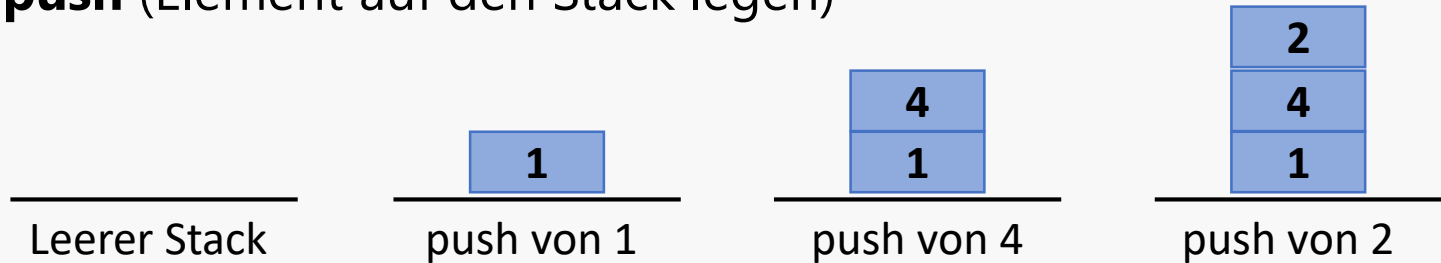
Endlosrekursion

- Bei vielen bisherigen Beispielen
 - Rekursion terminiert nicht
 - Methode ruft sich immer wieder auf
- Praxis in Java
 - Führt zu einem Fehler (StackOverflowError)

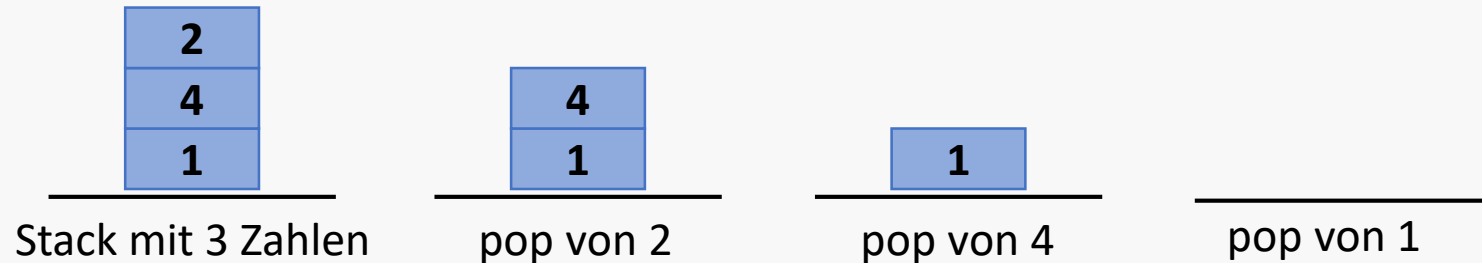
Rekursion und Stack-Speicher

Stack

- Einfache Datenstruktur
- Operationen (z. B. hier für einen Stack von Zahlen jeweils nach der Operation)
 - **push** (Element auf den Stack legen)



- **pop** (oberstes Element vom Stack nehmen)



JVM-Stack und Methoden-Frames (1)

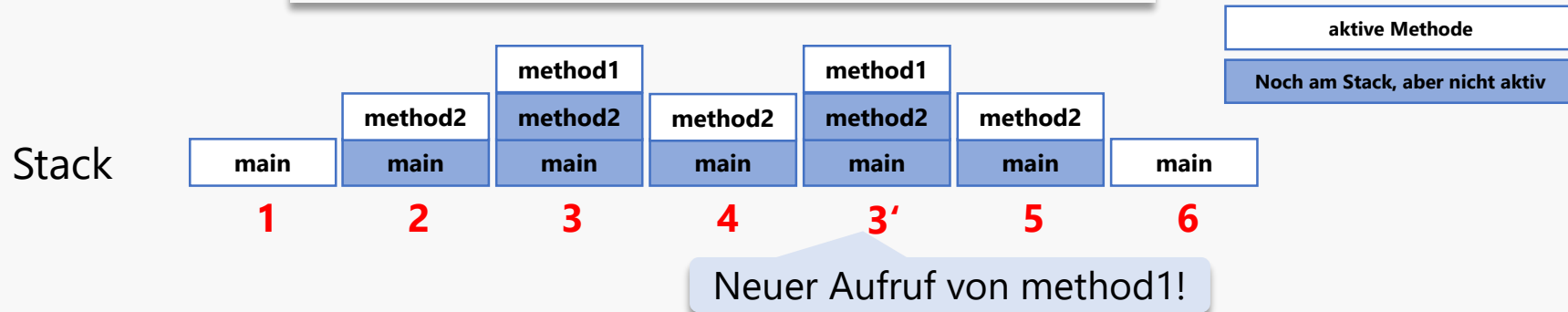
- JVM-Stack
 - Beim Ablauf eines Programms
- Stack-Frame wird beim Aufruf einer Methode erzeugt
 - Speichert Parameter, lokale Variablen, Zwischenresultate und weitere Daten einer Methode
 - Wird oben auf den Stack gelegt und wird der aktuelle (aktive) Frame

JVM-Stack und Methoden-Frames (2)

- Bei Beendigung einer Methode
 - Java Virtual Machine nimmt den Frame vom Stack und verwirft ihn (z. B. gehen lokale Daten verloren)
 - Der Frame der vorhergehenden Methode (aufrufenden Methode) wird der aktuelle (aktive) Frame

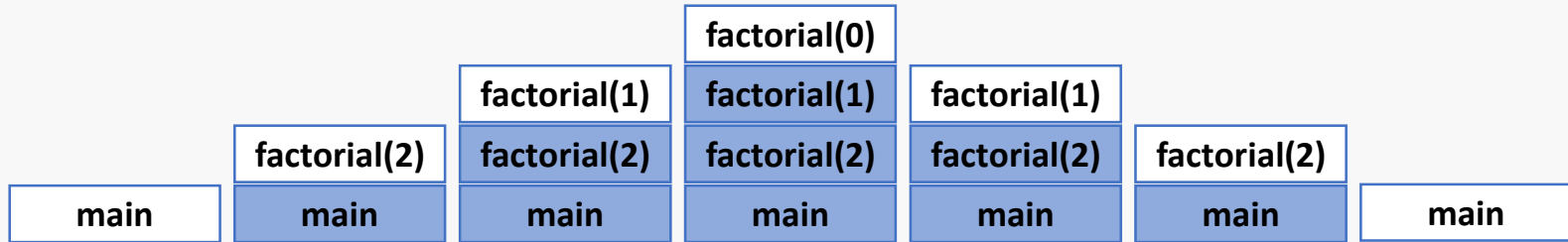
Stack bei Methodenaufrufen

```
public class Tester {  
    public static void main(String[] args) {  
        1 ... method2(); ... 6  
    }  
  
    private static void method2() {  
        2 ... method1(); ... 4 ... method1(); ... 5  
    }  
  
    private static void method1() {  
        ... 3 ...  
    }  
}
```

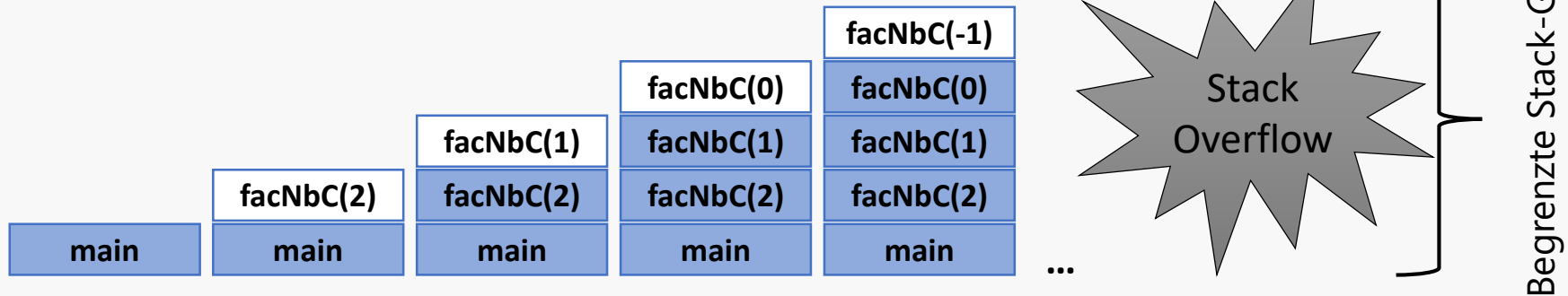


Stack und Rekursion (1)

- Aufruf von `factorial(2)` in `main`



- Aufruf von `factorialNoBaseCase(2)` in `main`
 - Führt zu Endlosrekursion



Stack und Rekursion (2)

- Beispiel aus Teil 1

```
private static int sumRecursive(int num) {  
    if (num <= 1) {  
        return 1;  
    } else {  
        return sumRecursive(num - 1) + num;  
    }  
}
```

- Terminierung

- Fundiert
- Fortschritt korrekt

- Praktisches Problem

- Zu viele rekursive Aufrufe bei großen Werten von n führen wiederum zu einem Stack-Overflow (Iteration dann vorteilhaft)

Vergleich von Rekursion und Iteration

Rekursion und Iteration

- Rekursion und Iteration sind gleich mächtig
- Rekursion lässt sich in Iteration umformen und umgekehrt
 - Das sagt noch nichts über die Lesbarkeit, den Umfang und die Effizienz des Ergebnisses aus
- Entrekursivierung (Rekursion zu Iteration)
 - Einfach bei Endrekursion und linearer Rekursion
 - Meist schwieriger (umfangreicher) bei verzweigter Rekursion
 - Schwieriger bei verschachtelter Rekursion

Beispiel (Rekursion – Iteration)

- Beispiel ggT-Berechnung

```
private static int gcdRecursive(int n, int m) {  
    if (n % m > 0) {  
        return gcdRecursive(m, n % m);  
    }  
    return m;  
}
```

```
private static int gcdIterative(int n, int m) {  
    while (n % m > 0) {  
        int tempN = m;  
        int tempM = n % m;  
        n = tempN;  
        m = tempM;  
    }  
    return m;  
}
```

Endrekursive Version

Iterative Version
(lange Version)

Transformationsschema für Endrekursion

```
type method(param1, param2, ..., result) {  
    if (condition) {  
        // Code ...  
        return method(expression1, expression2, ..., expressionX);  
    }  
    else  
        return result;  
}
```



```
type method(param1, param2, ..., result) {  
    while (condition) {  
        // Code ...  
        temp1 = expression1;  
        temp2 = expression2;  
        ...  
        tempX = expressionX;  
        param1 = temp1;  
        param2 = temp2;  
        ...  
        result = tempX;  
    }  
    return result;  
}
```


Endrekursion und Compiler

- Umformung kann automatisch erfolgen
- Manche Compiler erkennen solche Konstellationen
 - Auflösung vom Compiler abhängig
 - Nicht bei Oracle-Compiler für Java

Türme von Hanoi

- Einfache rekursive Implementierung

```
private static void solveHanoi(int n, String a, String b, String c) {  
    if (n > 0) {  
        solveHanoi(n - 1, a, c, b);  
        System.out.println(a + " --> " + c);  
        solveHanoi(n - 1, b, a, c);  
    }  
}
```

- Iterative Version (alternativer iterativer Ansatz)

```
private static void solveHanoiIterative(int n) {  
    int twoN = 1 << n;  
    for (int x = 1; x < twoN; x++) {  
        System.out.println((1 + (x & x - 1) % 3) + " --> " +  
                           (1 + ((x | x - 1) + 1) % 3));  
    }  
}
```

Rekursion und Iteration im Vergleich (1)

- Innerhalb eines Methodenrumpfs können mehrere rekursive Aufrufe erfolgen
 - Es ist aufwendig (aber möglich), solche rekursiven Methoden in Schleifen umzuformen
- Bestimmte Aufgaben lassen sich mit Rekursion kürzer formulieren

Rekursion und Iteration im Vergleich (2)

- Bei der Rekursion können Daten weitergereicht werden
 - Durch Parameter
 - Durch Rückgabewerte, die wiederum in Berechnungen einfließen können
- In einer Schleife können Daten dagegen nur von der vorigen zur nächsten Iteration fließen
 - Zur Kompensation sind häufig zusätzliche Variablen für Zwischenergebnisse notwendig
 - Komplexere Struktur durch zusätzliche Variablen

Rekursion und Iteration im Vergleich (3)

- Stack-Speicher ist begrenzt
 - Reicht oft aus
 - Geht bei sehr tiefen Rekursionen (sehr viele Aufrufebenen, großes n) aus
- Schleifen gehen mit diesem Speicher (zumindest in Java) weniger verschwenderisch um
 - Sind bei sehr großem n vorteilhaft

Rekursion und Iteration im Vergleich (4)

- Falsche oder fehlende Abbruchbedingung
 - Weder rekursive Methoden noch Schleifen terminieren auf normale Weise
- Rekursive Methoden brechen (zumindest in Java) irgendwann ab
 - Stack-Overflow
- Schleifen können ohne Fehlermeldung endlos laufen
 - Aufgrund der Fehlermeldungen sind solche Fehler bei Rekursion etwas leichter zu identifizieren als bei Schleifen

Rekursion und Iteration im Vergleich (5)

- Rekursion führt zu vielen kleinen Methoden, Schleifen zu eher größeren
- Bei Schleifen ist die Größe der Methoden besser steuerbar

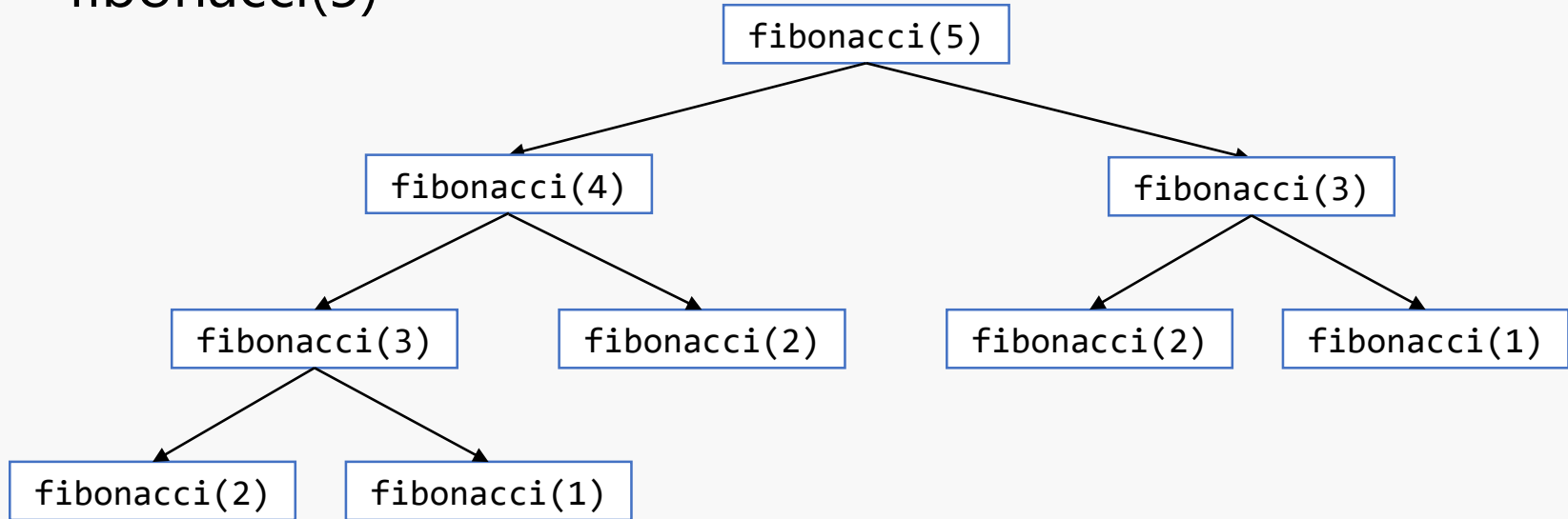
Rekursion und Iteration im Vergleich (6)

- Rekursion kann sehr ineffizient sein!
- Beispiel Fibonacci-Zahlen (Wiederholung)

```
private static long fibonacci(int n) {  
    if (n <= 2) {  
        return 1;  
    } else {  
        return fibonacci(n - 1) + fibonacci(n - 2);  
    }  
}
```


Aufrufbaum (Wiederholung)

- fibonacci(5)



- Wie sieht das für fibonacci(50) aus?
 - Wie viele Aufrufe müssen dort durchgeführt werden?

Test (Zeit messen, Anzahl der Aufrufe zählen)

```
public class FibonacciTester {
    private static long counter;
    // n >= 1
    private static long fibonacci(int n) {
        counter++;
        return n <= 2 ? 1 : fibonacci(n - 1) + fibonacci(n - 2);
    }
    public static void main(String[] args) {
        long start, end;
        long result;
        for (int i = 1; i <= 50; i++) {
            counter = 0;
            start = System.nanoTime();
            result = fibonacci(i);
            end = System.nanoTime();
            System.out.print("fibonacci(" + i + ") = " + result + " -> " + (end - start) / 1000000000.0
                + " seconds for " + counter + " method calls\n");
        }
    }
}
```

Ausgabe des Testers

...

fibonacci(40) = 102334155 -> 0.258819501 seconds for 204668309 method calls

fibonacci(41) = 165580141 -> 0.4036408 seconds for 331160281 method calls

fibonacci(42) = 267914296 -> 0.6514712 seconds for 535828591 method calls

fibonacci(43) = 433494437 -> 1.0455163 seconds for 866988873 method calls

fibonacci(44) = 701408733 -> 1.611623599 seconds for 1402817465 method calls

fibonacci(45) = 1134903170 -> 2.611942001 seconds for 2269806339 method calls

fibonacci(46) = 1836311903 -> 4.2654961 seconds for 3672623805 method calls

fibonacci(47) = 2971215073 -> 6.8933712 seconds for 5942430145 method calls

fibonacci(48) = 4807526976 -> 11.134899901 seconds for 9615053951 method calls

fibonacci(49) = 7778742049 -> 18.0095478 seconds for 15557484097 method calls

fibonacci(50) = 12586269025 -> 29.3449949 seconds for 25172538049 method calls

Testergebnisse

- Anzahl der Methodenaufrufe ist fix für ein bestimmtes n
 - Nimmt stark zu, führt aber zu keinem Abbruch!
 - Aufruftiefe ist nicht sehr groß
- Zeitangaben sind keine fixen Werte
 - Auf einem konkreten Rechner (mit Intel Core i7 9750H) gemessen
 - Können leicht schwanken
 - Zeit für `fibonacci(n)` \approx Zeit für `fibonacci(n-1)` + Zeit für `fibonacci(n-2)`

fibonacci(100)?

- Wie lange dauert fibonacci(100)?
 - ca. 26 000 Jahre (eine sehr grobe Abschätzung!)
- Wie kann das rekursiv und effizient implementiert werden?

Fibonacci-Zahlen endrekursiv

- Zusätzliche überladene Methode

```
private static long fibonacci2(int n, long current, long next) {  
    if (n <= 1) {  
        return current;  
    } else {  
        return fibonacci2(n - 1, next, current + next);  
    }  
}
```

```
private static long fibonacci2(int n) {  
    return fibonacci2(n, 1, 1);  
}
```

Beispiel für Ablauf

- fibonacci2(10)
 - Aufrufe von fibonacci2(int n, long current, long next) mit folgenden Werten

n	current	next
10	1	1
9	1	2
8	2	3
7	3	5
6	5	8
5	8	13
4	13	21
3	21	34
2	34	55
1	55	89

Zusammenfassung

- Lineare Rekursion
 - Endrekursion
- Verzweigte Rekursion
- Verschachtelte Rekursion
- Wechselseitige Rekursion
- Terminierung
- Rekursion und Stack-Speicher
- Rekursion vs. Iteration