DIE LIEBE ZUR REKURSION

Der Java-Stack, Funktionsaufrufe und Rekursion.
Ein wiederkehrendes Dilemma.

Florian Sihler

10. November 2022
SP, Universität Ulm

DIE LIEBE ZUR REKURSION

Der Java-Stack, Funktionsaufrufe und Rekursion.
Ein wiederkehrendes Dilemma.

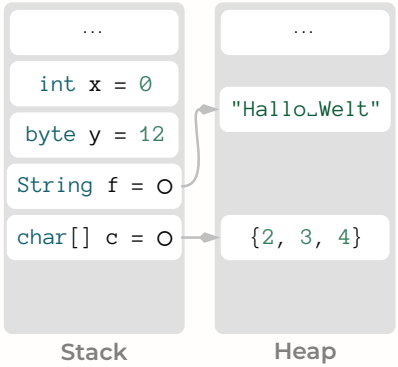
Florian Sihler

Officially supported by the Pingu-Foundation for Emotional Support. There will be light and there will be more peeps to come!

10. November 2022
SP, Universität Ulm

2 Javas' Speicherverwaltung

```
int x = 0;
byte y = 12;
String f = "Hallo_Welt";
char[] c = {2, 3, 4};
```



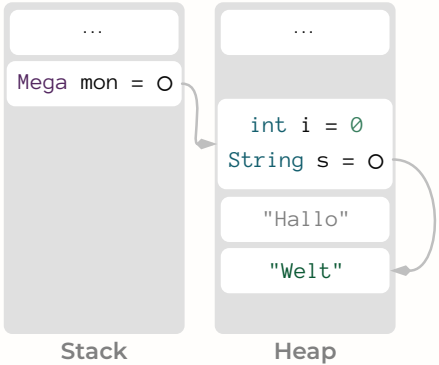
[5]: *The Java® Virtual Machine Specification [Heap & Stack]* Lindholm, 2020

3 Komplexere Konstrukte

Beachte: Alle Variablen erhalten in Java „Default-Werte“. Für komplexe Datentypen ist dies `null`.

```
class Mega {
    public int i;
    public String s;
}

Mega mon = new Mega();
mon.s = "Hallo";
mon.s = "Welt";
```

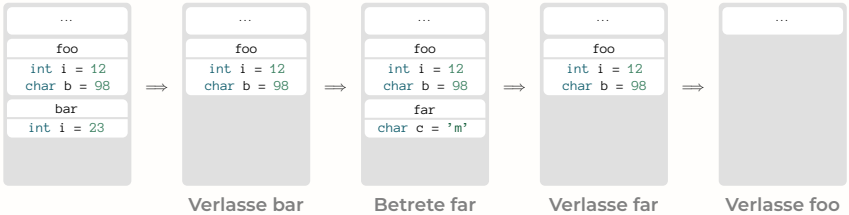


[5]: *The Java® Virtual Machine Specification [Heap & Stack]*
Lindholm, 2020

[8]: *LaTeX-Package, tikzpingus*
Sihler, 2021

4 Stack Frames

- Handhaben (lokale) Gültigkeitsbereiche.
- Werden beim Betreten auf dem Stack abgelegt.
- Werden beim Verlassen wieder vom Stack entnommen.

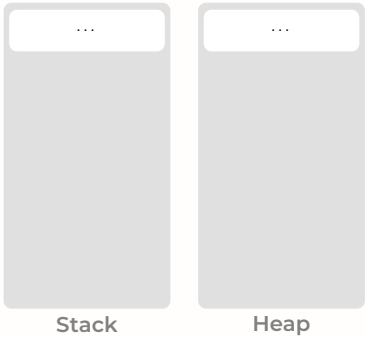


[3]: *Java Debug Interface – StackFrame (JDK 11)* , 2021

5.1 Stack Frames – Beispiel

```
void f(char c) {
  int x = 4;
  System.out.print(b(c+x));
}
String b(int n) {
  return "" + (n % 9);
}

f(b(42).charAt(0));
```



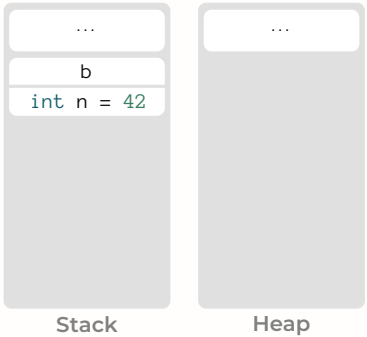
[3]: *Java Debug Interface – StackFrame (JDK 11)* , 2021

Stack Frames – Beispiel

Wir werden Rückgabemechanismen sträflich vereinfachen!

```
void f(char c) {
  int x = 4;
  System.out.print(b(c+x));
}
String b(int n) {
  return "" + (n % 9);
}

f(b(42).charAt(0));
```



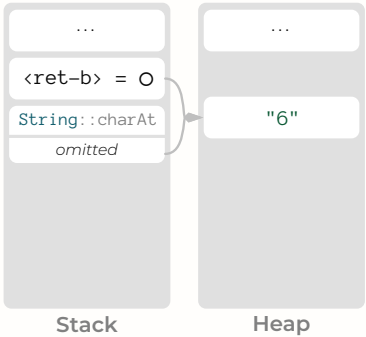
[3]: *Java Debug Interface – StackFrame (JDK 11)* , 2021

Stack Frames – Beispiel

Wir werden Rückgabemechanismen sträflich vereinfachen!

```
void f(char c) {
  int x = 4;
  System.out.print(b(c+x));
}
String b(int n) {
  return "" + (n % 9);
}

f(b(42).charAt(0));
```



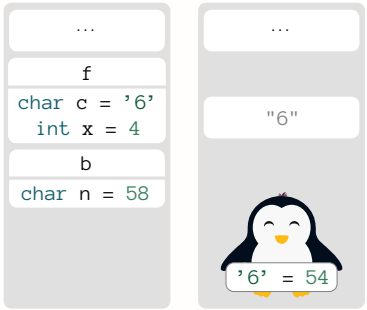
[3]: *Java Debug Interface – StackFrame (JDK 11)* , 2021

Stack Frames – Beispiel

Wir werden Rückgabemechanismen sträflich vereinfachen!

```
void f(char c) {
  int x = 4;
  System.out.print(b(c+x));
}
String b(int n) {
  return "" + (n % 9);
}

f(b(42).charAt(0));
```



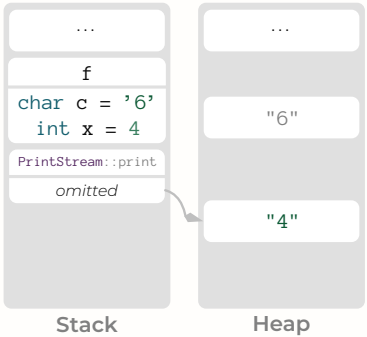
[3]: *Java Debug Interface – StackFrame (JDK 11)* , 2021

Stack Frames – Beispiel

Wir werden Rückgabemechanismen sträflich vereinfachen!

```
void f(char c) {
  int x = 4;
  System.out.print(b(c+x));
}
String b(int n) {
  return "" + (n % 9);
}

f(b(42).charAt(0));
```



[3]: *Java Debug Interface – StackFrame (JDK 11)* , 2021

Stack Frames – Beispiel

Wir werden Rückgabemechanismen sträflich vereinfachen!

```
void f(char c) {
  int x = 4;
  System.out.print(b(c+x));
}
String b(int n) {
  return "" + (n % 9);
}

f(b(42).charAt(0));
```

[3]: *Java Debug Interface – StackFrame (JDK 11)* , 2021

6 Anatomie einer Rekursion

```
1 func recursive(problem arguments) is
2     if at elemental problem then
3         solve elemental problem;
4     else
5         Split problem into smaller ones;
6         Try to solve them recursively;
7         Combine step-solutions to one;
8     end
9 end
```

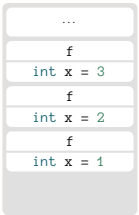
Nicht alle müssen genau dieser Struktur folgen.

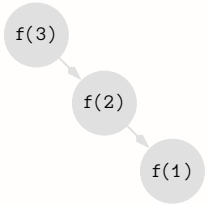
Be special. Be wild!

Stack Frames und Aufrufgraphen

- Stack Frames werden schnell unübersichtlich.
- Wir vereinfachen dies durch Aufrufgraphen.
- Knoten repräsentieren Stack Frames.

```
int f(int x) {
    return f(x - 1);
}
```

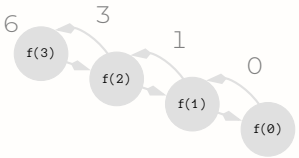




Lineare Rekursion

Lineare Rekursion wird weiter unterteilt! So beispielweise in Head- und Tailrekursiv.

```
int f(int n) {
  if (n <= 0) return n;
  else return n + f(n - 1);
}
```



Die einfachste und geläufigste Form der Rekursion.

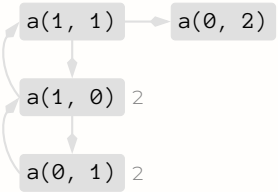
Hier handelt es sich sogar um eine *primitiv-rekursive* Funktion![2]

[2]: *About primitive recursive algorithms* Colson, 1991

Verschachtelte Rekursion

Dies ist die Ackermann Funktion!

```
int a(int m, int n) {
  if(m == 0) return n + 1;
  else if (n == 0) return a(m - 1, 1);
  else return a(m - 1, a(m, n - 1));
}
```

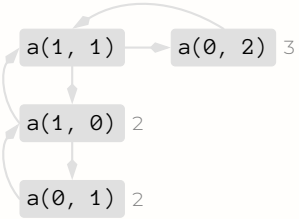


[9]: *The Ackermann function. a theoretical, computational, and formula manipulative study*
Sundblad, 1971

Verschachtelte Rekursion

Dies ist die Ackermann Funktion!

```java
int a(int m, int n) {
  if(m == 0) return n + 1;
  else if (n == 0) return a(m - 1, 1);
  else return a(m - 1, a(m, n - 1));
}
```



Verschachtelt-Rekursive Funktionen sind meist sehr schwer nachzuvollziehen.

Damit ist $a(1,1) = 3$.

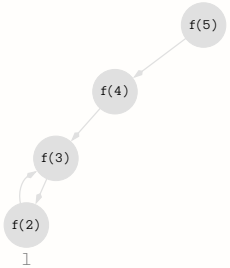
Was ist dann beispielsweise $a(4,2)$?

Nun… $a(4,2) \approx 2 \cdot 10^{19\,729}$.

[9]: *The Ackermann function. a theoretical, computational, and formula manipulative study*
Sundblad, 1971

10.1 Kaskadenförmige Rekursion

```
int f(int n) {
    if(n <= 2) return 1;
    else return f(n - 1) + f(n - 2);
}
```

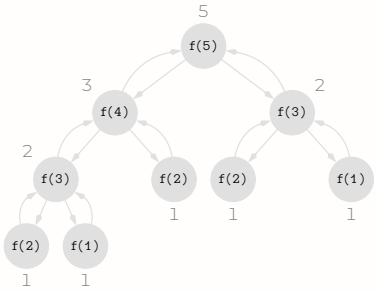


[7]: *Exploring Recursion with Fibonacci Numbers*
Rubio-Sánchez, 2007

Kaskadenförmige Rekursion

Sie wird manchmal auch „verzweigte Rekursion“ genannt.

```
int f(int n) {
    if(n <= 2) return 1;
    else return f(n - 1) + f(n - 2);
}
```



Elegant, aber meist nicht sonderlich effizient.

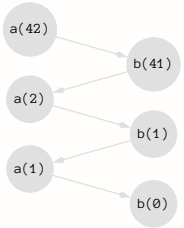
[7]: *Exploring Recursion with Fibonacci Numbers*
Rubio-Sánchez, 2007

11.1 Wechselseitige Rekursion

Sie kann durch Funktionstupel stets in „normale“ Rekursionen transformiert werden.

```
int a(int x) {
    if(x <= 0) return x + 1;
    return b(x - 1);
}

int b(int x) {
    if(x <= 0) return x;
    return a(x % 3);
}
```

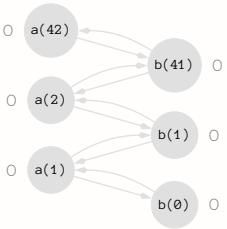


11.2 Wechselseitige Rekursion

Sie kann durch Funktionstupel stets in „normale“ Rekursionen transformiert werden.

```
int a(int x) {
    if(x <= 0) return x + 1;
    return b(x - 1);
}

int b(int x) {
    if(x <= 0) return x;
    return a(x % 3);
}
```



12 Ein paar Notizen

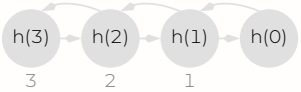
- Die gezeigten Beispiele waren möglichst minimal
- Die Varianten können kombiniert werden!
  › z.B. eine kaskadenförmige, wechselseitige Rekursion.
- Rekursion und Iteration sind meist gleich mächtig.

[6]: *From Recursion to Iteration: What Are the Optimizations?*
Liu, 1999

Head- vs. Tail-Rekursion

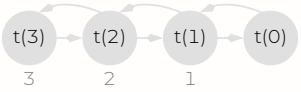
Kurzgesagt: Passiert noch etwas im rekursiven Aufstieg oder nicht (tail).

```
void h(int n) {
    if(n <= 0) return;
    h(n - 1);
    System.out.println(n);
}
```



Ausgabe: 1 → 2 → 3

```
void t(int n) {
    if(n <= 0) return;
    System.out.println(n);
    t(n - 1);
}
```



Ausgabe: 3 → 2 → 1

Disclaimer: Die Feinheiten einer Head-Rekursion werden hier vernachlässigt. Tail-Rekursionen sind interessanter, da sie sich problemlos in Iterationen transformieren lassen.

Mathematische Funktionen

- Einige mathematischen Funktionen sind rekursiv definiert.
- Die gegebene Definition lässt sich meist 1:1 übersetzen.
- Sie nimmt bereits einiges des Denkaufwands ab.

„Elementarprobleme" sind hier die nicht-rekursiven Fälle: Teile der Folge, welche oft axiomatisch gegeben sind.

$$f(n) = \begin{cases} 1, & n \leq 1 \\ n \cdot f(n-1), & n > 1 \end{cases}$$

$$f_n : \mathbb{N} \to \mathbb{N}, n \mapsto n \cdot f_{n-1}$$
$$f_1 = 1$$

```
public int f(int n) {
  if(n <= 1) return 1;
  else return n * f(n - 1);
}
```

Die Ackermann Funktion, Alternativ

```
int f(int n) {
   return phi(n, n, n);
}

int alpha(int a, int n) {
   return n <= 1 ? n : a;
}
```

$$f(n) = \varphi(n, n, n)$$

$$\varphi(a, b, 0) = a + b$$
$$\varphi(a, 0, n) = \alpha(a, n - 1)$$
$$\varphi(a, b, n) = \varphi(a, \varphi(a, b - 1, n), n - 1)$$

$$\alpha(a, n) = \begin{cases} 0, & \text{wenn } n = 0 \\ 1, & \text{wenn } n = 1 \\ a, & \text{wenn } n > 1 \end{cases}$$

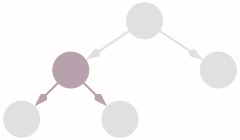
Die Ackermann Funktion, Alternativ


```
int phi(int a, int b, int n) {
    if(n == 0) return a + b;
    else if (b == 0) return alpha(a, n - 1);
    else return phi(a, phi(a, b - 1, n), n - 1);
}
```

$f(1) = 1, f(2) = 4, f(3) =$ ↯

$$\gtrsim 2 \uparrow^2 7 \approx 10^{10^{10^{19\,000}}}$$

Rekursive Strukturen



Jeder Knoten ist die Wurzel eines Teilbaums

```
class Node {
    Node left;
    Node right;
    int data;
}
```

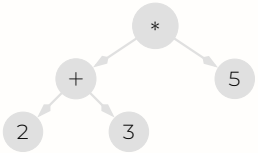
```
class Node<T> {
  Node left;
  Node right;
  T data;
}
```

Cooler mit Generics![4]

Jedes Element ist Kopf einer Teilliste

```
class Element {
    Element next;
    int data;
}
```

18.1 Einen Baum evaluieren



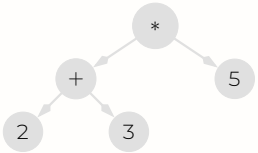
Ein Arithmetik-Baum!

```
class Node {
    Node left, right;
    int data; String op;
}
```

```
long postOrderEval(Node node) {
    if(node == null) throw new /* ... */;
    if(node.op == null) // no operation
        return node.data;
    long l = postOrderEval(node.left);
    long r = postOrderEval(node.right);
    switch(node.op) {
        case "+": return l + r;
        case "*": return l * r;
        // ... other cases, errors, ...
    }
}
```

An sich würde man die Knoten anders realisieren. Z.B. durch Vererbung.

18.2 Einen Baum evaluieren



Ein Arithmetik-Baum!

An sich würde man die Knoten anders realisieren. Z.B. durch Vererbung.

```
class Node {
    Node left, right;
    int data; String op;
}
```

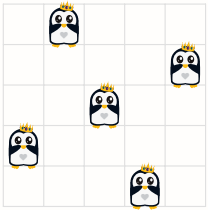
```
long postOrderEval(Node node) {
    if(node == null) throw new /* ... */;
    if(node.op == null) // no operation
        return node.data;
    long l = postOrderEval(node.left);
    long r = postOrderEval(node.right);
    switch(node.op) {
        case "+": return l + r;
        case "*": return l * r;
        // ... other cases, errors, ...
    }
}
```

Die Welt rekursiver Probleme

1. Rekursives Sortieren

2. Die Türme von Hanoi

3. Das N-Damen Problem



- Sudokus
- Wegfindung
- Hilbert-Kurven
- ...

Alles was sich in (eventuell explorative) Einzelschritte zerteilen lässt.

Rekursives Sortieren


```
void quicksort(int[] arr) {
    quicksort(arr, 0, arr.length - 1);
}

void quicksort(int[] arr, int left, int right) {
    if(low >= right) return;
    int partIndex = partition(arr, left, right);
    quicksort(arr, left, partIndex - 1);
    quicksort(arr, partIndex + 1, right);
}
```

Wählt Pivot und sortiert kleiner Pivotelement links und größer gleich rechts.

Die Türme von Hanoi

- Trivial für eine Scheibe: $A \rightarrow C$.
- Sonst:
  - Bewege $n - 1$ Scheiben auf Hilfsstapel: $A \rightarrow B$.
  - Bewege letzte Scheibe (größte) von $A \rightarrow C$.
  - Bewege $n - 1$ Scheiben vom Hilfsstapel zum Ziel: $B \rightarrow C$.

```
enum Pole {
  SOURCE,
  HELPER,
  TARGET
}
```

```
void hanoi(int disks, Pole a, Pole b, Pole c) {
   if(disks == 1) { move(a, c); }
   else {
      hanoi(disks - 1, a, c, b); // move all disks above to help pole
      move(a, c); // move last disk to target
      hanoi(disks - 1, b, a, c); // move disks from help to target pole
   }
}
```

Gibt im einfachsten Fall einfach „$a \rightarrow c$“ aus.

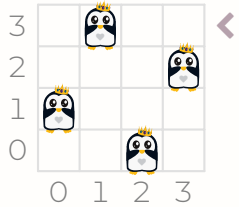
Das N-Damen Problem

```
boolean nQueens(boolean[][] board) {
  return nQueens(board, 0);
}
```

```
boolean nQueens(boolean[][] board, int row) {
    if(row == board.length) return true;
    for (int column = 0; column < board.length; column++) {
        if(!queenCanBePlacedHere(board, column, row)) continue;
        board[row][column] = true; // Try to place queen here
        if(nQueens(board, row + 1))
            return true;
        board[row][column] = false;
    }
    return false;
}
```

Wie könnte man `queenCanBePlacedHere` umsetzen?

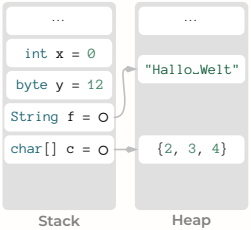
`q(board, 0)`



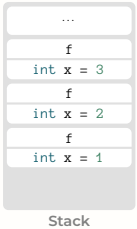
Das obligatorische „Übrigens…“

- Viele Varianten:
  - Tail-Rekursionen
  - Primitiv-Rekursive Funktionen
- Stack-Frames sind spannender und enthalten noch mehr!
- Das riesige Thema des Compilerbaus liefert mehr:
  - Optimierungen & Transformationen
  - Recursive Descent Parser
- Dynamische Programmierung!

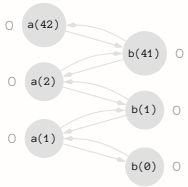
[1]: *Proper Tail Recursion and Space Efficiency* Clinger, 1998



Speicherverwaltung



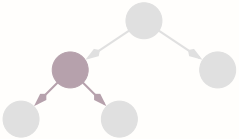
Stack Frames



Rekursionstypen

$$f(n) = \begin{cases} 1, & n \leq 1 \\ n \cdot f(n-1), & n > 1 \end{cases}$$

Mathematisch



Datenstrukturen

Divide- & Conquer

[1] William D. Clinger. „Proper Tail Recursion and Space Efficiency“. 1998

[2] Loïc Colson. „About primitive recursive algorithms“. 1991

[3] *Java Debug Interface – StackFrame (JDK 11)*. 2021

[4] *Lesson: Generics*. 2021

[5] Tim Lindholm u. a. *The Java® Virtual Machine Specification [Heap & Stack]*. 2020

[6] Yanhong A. Liu und Scott D. Stoller. „From Recursion to Iteration: What Are the Optimizations?“ Nov. 1999

[7] Manuel Rubio-Sánchez und Isidoro Hernán-Losada. „Exploring Recursion with Fibonacci Numbers“. 2007

[8] Florian Sihler. *LaTeX-Package, tikzpingus*. 2021

[9] Yngve Sundblad. „The Ackermann function. a theoretical, computational, and formula manipulative study“. 1971

Florian Sihler

Ulm, 10. November 2022

Slides and LaTeX-sources on GitHub!