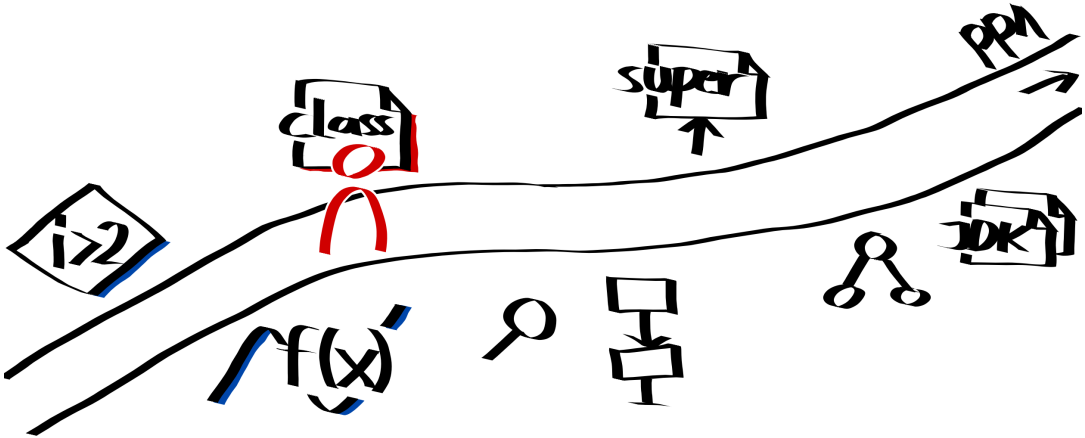
Kapitel 3: Objekte, Speicher & Klassen

VL 10: Stack & Rekursion

Wintersemester 2023/24

Stand 13. November 2023, 07:07 Uhr

Wo stehen wir gerade?

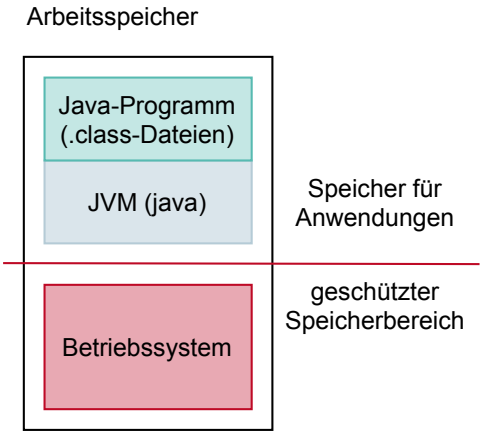


Vorlesungsfeedback

- Tempo/Schwierigkeit: genau richtig – zu hoch (n=4)
- ? Glossar
- ? Scanner als Methoden-Parameter

Speicherbereiche: Abstrakt

- Betriebssystem benötigt Speicher
  - Kernel
  - Treiber
  - ...
- JVM
  - Verwaltungsdaten
  - Implementierung von eingebauten Methoden
- Java-Programm (unser Code):
  - Verwaltungsdaten
  - Implementierung



Stack[1]

- Datenstruktur, bei der nur Zugriff auf zuletzt gespeichertes Element möglich
- Funktionsweise ähnlich wie Tablettstapel in Mensa

[1] Keller(-speicher), Stapel(-speicher)

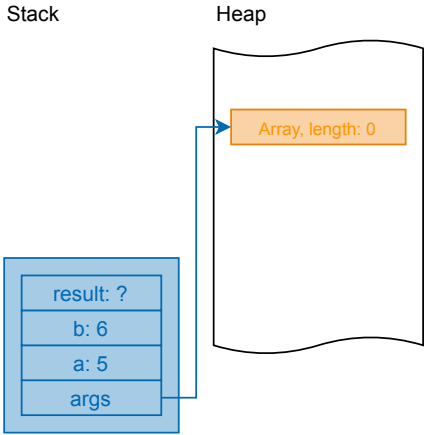
Interner JVM-Stack

- speichert Stack-Frames:
  - bei jedem Methoden-Aufruf angelegt
  - Speicherplatz für Parameter
  - Speicherplatz für lokale Variablen
  - Wo erfolgte Methodenaufruf?
  - Wo ist Stackframe des Aufrufers?
- Übergabe von Rückgabewerten

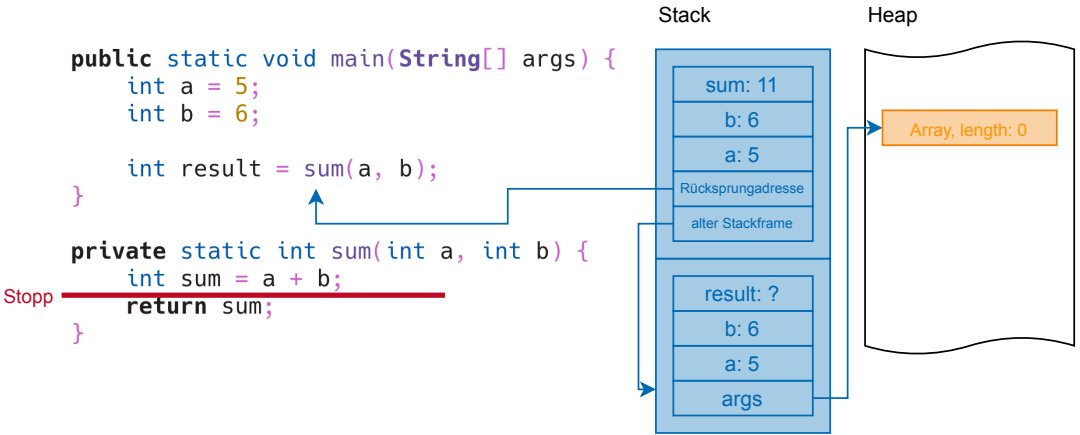
Stack vor Methodenaufruf

```
public static void main(String[] args) {
    int a = 5;
    int b = 6;
Stopp ──────────────────────────────
    int result = sum(a, b);
}

private static int sum(int a, int b) {
    int sum = a + b;
    return sum;
}
```



Stack in Methode



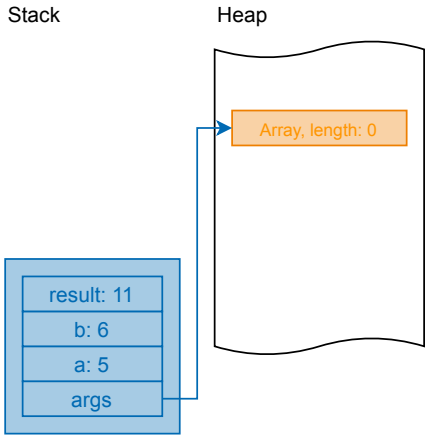
Stack nach Methodenaufruf

```
public static void main(String[] args) {
    int a = 5;
    int b = 6;

    int result = sum(a, b);
}

private static int sum(int a, int b) {
    int sum = a + b;
    return sum;
}
```

Stopp



Stack vs. Heap

Heinrich Heine Universität Düsseldorf

Was wir als Programmierer:innen wissen müssen:

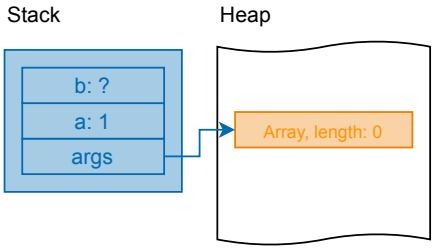
- Stack
  - Parameter & lokale Variablen
  - Methodenaufruf beansprucht Platz
  - Verlassen einer Methode gibt Platz frei
- Heap
  - Objekte und deren Eigenschaften
  - `new` fordert Speicherplatz im Heap an
  - Platz wird frei, wenn Objekt nicht mehr erreichbar (automatische *Garbage Collection*)
- für beide gilt: nur begrenzter Speicherplatz
  - Größe (typischerweise) beim JVM-Start festgelegt

Folge: Inhalt lokaler Variablen kopiert

```
3  public static void main(String[] args) {
4      int a = 1;
5      int b = a;
6
7      a = 0;
8
9      System.out.println(a);
10     System.out.println(b);
11 }
```

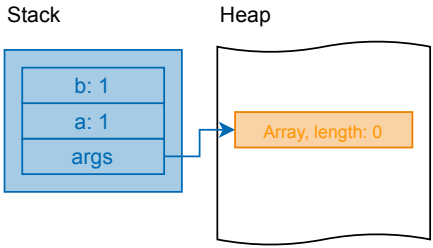
Folge: Inhalt lokaler Variablen kopiert

```
3  public static void main(String[] args) {
4      int a = 1;
5      int b = a;
6
7      a = 0;
8
9      System.out.println(a);
10     System.out.println(b);
11 }
```



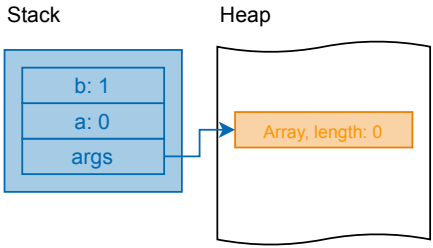
Folge: Inhalt lokaler Variablen kopiert

```
3  public static void main(String[] args) {
4      int a = 1;
5      int b = a;
6
7      a = 0;
8
9      System.out.println(a);
10     System.out.println(b);
11 }
```



Folge: Inhalt lokaler Variablen kopiert

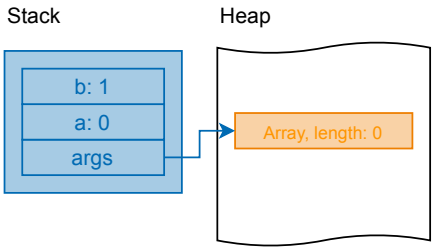
```
3  public static void main(String[] args) {
4      int a = 1;
5      int b = a;
6
7      a = 0;
8
9      System.out.println(a);
10     System.out.println(b);
11 }
```



Folge: Inhalt lokaler Variablen kopiert

```java
3  public static void main(String[] args) {
4      int a = 1;
5      int b = a;
6
7      a = 0;
8
9      System.out.println(a);
10     System.out.println(b);
11 }
```

```
% java Primitives
0
1
```



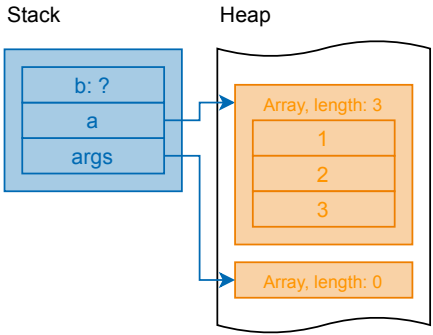
Es gibt nicht mehrere „Referenzen“ auf dieselbe lokale Variable.

Folge: Mehrere Referenzen auf dasselbe Objekt

```
3  public static void main(String[] args) {
4      int[] a = {1, 2, 3};
5      int[] b = a;
6
7      a[0] = 0;
8
9      System.out.println(a[0]);
10     System.out.println(b[0]);
11 }
```

Folge: Mehrere Referenzen auf dasselbe Objekt

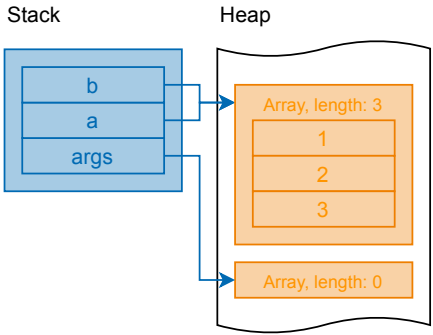
```
3  public static void main(String[] args) {
4      int[] a = {1, 2, 3};
5      int[] b = a;
6
7      a[0] = 0;
8
9      System.out.println(a[0]);
10     System.out.println(b[0]);
11 }
```



Folge: Mehrere Referenzen auf dasselbe Objekt

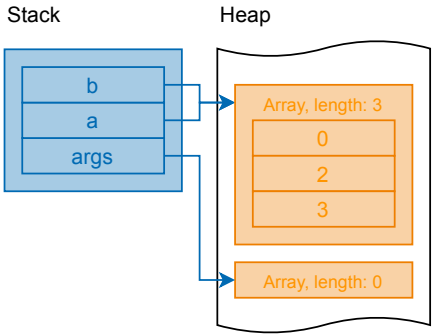
Heinrich Heine Universität Düsseldorf

```
3  public static void main(String[] args) {
4      int[] a = {1, 2, 3};
5      int[] b = a;
6
7      a[0] = 0;
8
9      System.out.println(a[0]);
10     System.out.println(b[0]);
11 }
```



Folge: Mehrere Referenzen auf dasselbe Objekt

```
3  public static void main(String[] args) {
4      int[] a = {1, 2, 3};
5      int[] b = a;
6
7      a[0] = 0;
8
9      System.out.println(a[0]);
10     System.out.println(b[0]);
11 }
```

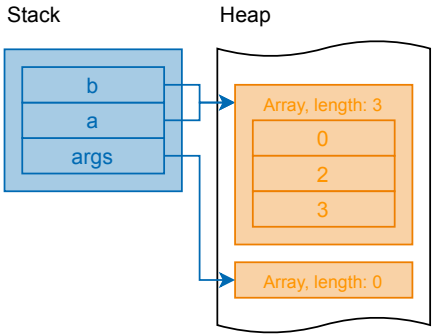


Folge: Mehrere Referenzen auf dasselbe Objekt

Heinrich Heine Universität Düsseldorf

```
3  public static void main(String[] args) {
4      int[] a = {1, 2, 3};
5      int[] b = a;
6
7      a[0] = 0;
8
9      System.out.println(a[0]);
10     System.out.println(b[0]);
11 }
```

```
% java Primitives
0
0
```



Derselbe Speicherbereich kann über verschiedene Variablen verändert werden.

= kopiert Stack-Inhalt, nicht Heap-Inhalt.

Folge: Primitive Werte des Aufrufers nicht änderbar

Heinrich Heine Universität Düsseldorf

```
3  private static void change(int value) {
4      value = 0;
5  }
6
7  public static void main(String[] args) {
8      int a = 1;
9      change(a);
10     System.out.println(a);
11 }
```

Folge: Primitive Werte des Aufrufers nicht änderbar

```
3  private static void change(int value) {
4      value = 0;
5  }
6
7  public static void main(String[] args) {
8      int a = 1;
9      change(a);
10     System.out.println(a);
11 }
```

```
% java PrimitiveCall
1
```

Ändert eine Methode einen Wert, der direkt im **Stack** steht (z. B. den Wert eines Parameters mit primitiven Datentyp), ist das für die aufrufende Methode **nicht** sichtbar

Folge: Objekte des Aufrufers änderbar

```
3  private static void change(int[] array) {
4      array[0] = 0;
5  }
6
7  public static void main(String[] args) {
8      int[] a = {1, 2, 3};
9      change(a);
10     System.out.println(a[0]);
11 }
```

Folge: Objekte des Aufrufers änderbar

Heinrich Heine Universität Düsseldorf

```
3  private static void change(int[] array) {
4      array[0] = 0;
5  }
6
7  public static void main(String[] args) {
8      int[] a = {1, 2, 3};
9      change(a);
10     System.out.println(a[0]);
11 }
```

```
% java ReferenceCall
0
```

Ändert eine Methode einen Wert, der im **Heap** steht (z. B. eine Objekt-Eigenschaft), ist das für die aufrufende Methode sichtbar.

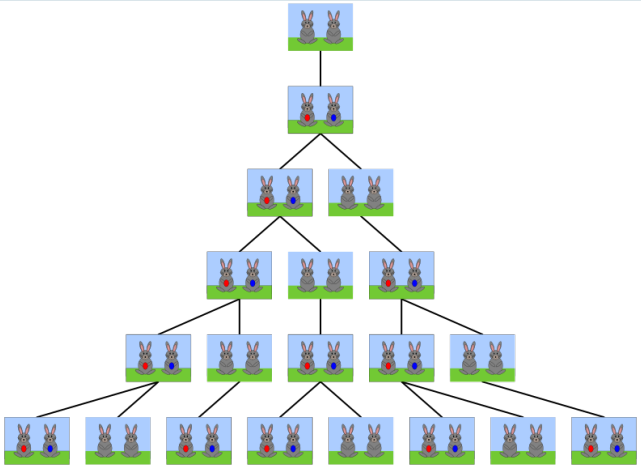
Dreieckstausch

Heinrich Heine Universität Düsseldorf

- Ziel: Tausche Wert zweier Variablen

```
5  double[] numbers = {40.5, 30.1};
6
7  if(numbers[0] > numbers[1]) {
8      // "Dreieckstausch"
9      double oldNumber0 = numbers[0];
10     numbers[0] = numbers[1];
11     numbers[1] = oldNumber0;
12 }
13
14 System.out.println(numbers[0]);
15 System.out.println(numbers[1]);
```

Vermehrung von Kaninchen[2]



[2] HB (https://commons.wikimedia.org/wiki/File:FibonacciRabbit.svg), „FibonacciRabbit“, CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0/)

Fibonacci-Folge

- 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = \begin{cases} 0 & \text{wenn } n = 0 \\ 1 & \text{wenn } n = 1 \\ F(n-1) + F(n-2) & \text{sonst} \end{cases}$$

Direkte Umsetzung in Java

```java
private static int fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

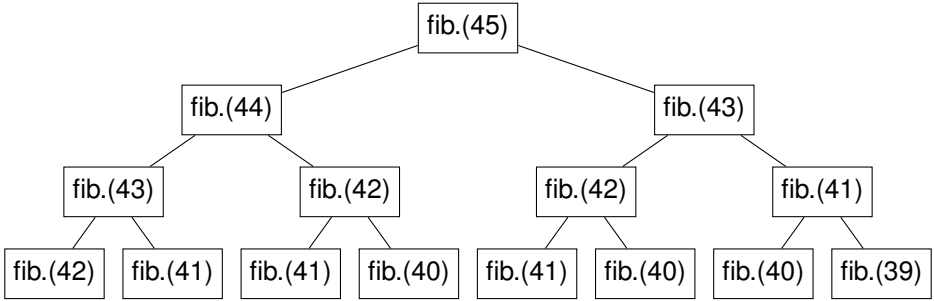
Funktioniert das?

Direkte Umsetzung in Java

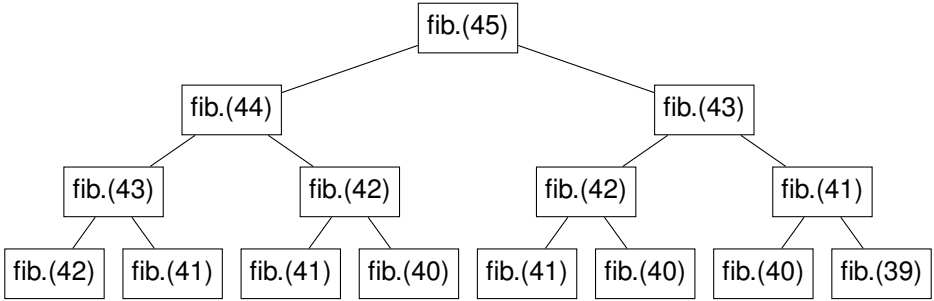
```java
private static int fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Funktioniert das?
Funktioniert das gut?

fibonacci(45)



fibonacci(45)



- sehr viele doppelte Berechnungen:
  - `fibonacci(45)` 1-mal aufgerufen
  - `fibonacci(44)` 1-mal aufgerufen
  - `fibonacci(43)` 2-mal aufgerufen
  - `fibonacci(42)` 3-mal aufgerufen
  - ...

Alternative, schnellere Umsetzung

Heinrich Heine Universität Düsseldorf

```java
private static int fibonacci(int n) {
    if (n < 2) {
        return n;
    }

    int lastFibonacci = 1;
    int secondToLastFibonacci = 0;
    for (int i = 2; i <= n; i++) {
        int newFibonacci = lastFibonacci + secondToLastFibonacci;
        secondToLastFibonacci = lastFibonacci;
        lastFibonacci = newFibonacci;
    }

    return lastFibonacci;
}
```

Und dann kamen die Mathematiker:innen ...

Heinrich Heine Universität Düsseldorf

Man kann zeigen, dass sich die n-te Fibonacci-Zahl auch direkt berechnen lässt:

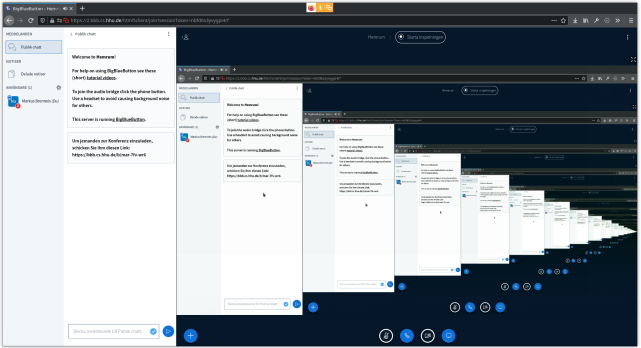
$$F(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

Rekursion

Heinrich Heine Universität Düsseldorf

Definition

Eine **rekursive** Methode ist eine Methode, die sich (direkt oder indirekt) selbst aufruft.



Wer kennt diese Funktion?

Heinrich Heine
Universität Düsseldorf

Auch bei mathematischen Funktionen anzutreffen:

$$n! = \begin{cases} 1 & \text{wenn } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

Fakultät rekursiv

direkte Umsetzung dieser mathematischen Definition:

```java
private static double factorial(double n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

$$n! = \begin{cases} 1 & \text{wenn } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

Gibt es hier ein Problem?

Heinrich Heine Universität Düsseldorf

```java
private static double factorial(double n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}

public static void main(String[] args) {
    System.out.println(factorial(11000));
}
```

Gibt es hier ein Problem?

```java
private static double factorial(double n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}

public static void main(String[] args) {
    System.out.println(factorial(11000));
}
```

```
% java FactorialRecursive
Exception in thread "main" java.lang.StackOverflowError
        at FactorialRecursive.factorial(FactorialRecursive.java:6)
        at FactorialRecursive.factorial(FactorialRecursive.java:6)
        at FactorialRecursive.factorial(FactorialRecursive.java:6)
        at FactorialRecursive.factorial(FactorialRecursive.java:6)
        at FactorialRecursive.factorial(FactorialRecursive.java:6)
        at FactorialRecursive.factorial(FactorialRecursive.java:6)
```

Kann Rekursion mehr als Schleifen?

Kann Rekursion mehr als Schleifen?

Heinrich Heine Universität Düsseldorf

- bewiesen: Rekursion und Schleifen gleich mächtig →TheoInfo
  - für manche Probleme Rekursion einfacher
  - in manchen Programmiersprachen keine Schleifen, nur Rekursion

$$n! = \prod_{i=1}^{n} i$$

```java
private static double factorial(double n) {
    double result = 1;
    for(int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Faustregel in Java: Ist ein Problem ohne große Umstände iterativ[3] lösbar, auf Rekursion verzichten.

[3] mit Schleifen, ohne Rekursion

Gibt es hier ein Problem?

Heinrich Heine
Universität Düsseldorf

```
1 private static double factorial(double n) {
2     return n * factorial(n - 1);
3 }
```

Gibt es hier ein Problem?

```java
private static double factorial(double n) {
    return n * factorial(n - 1);
}
```

Fehlende Abbruchbedingung für Rekursion

⇒ Endlosrekursion[4]

⇒ Speicherplatz für Stack geht aus (`StackOverflowError`)

[4] infiniter Regress

Lernziele

Heinrich Heine
Universität Düsseldorf

Sie können am Ende der Woche ...

- **erklären**, welche Daten im Heap/Stack gespeichert werden.
- die Semantik von = **erklären**.
- Methoden **schreiben**, die übergebene Objekte (nicht) verändern.
- rekursive Definitionen von Funktionen in Methoden **übersetzen**.
- die Funktionalität einer gegebenen rekursiven Methode **erklären**.

Wichtige Begriffe

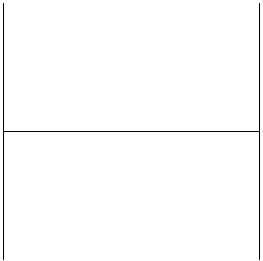
Heinrich Heine
Universität Düsseldorf

Stack
Garbage Collection
Rekursion
Stack Overflow
iterativ

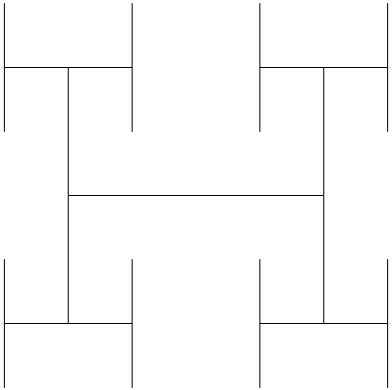
Übungsblatt

- Vokabeln
- Skytale
- Debugging: Längstes Plateau

H-Baum



H-Baum



H-Baum