Grundlagen der Programmierung

VL08: Rekursion

Prof. Dr. Samuel Kounev
Jóakim v. Kistowski
Norbert Schmitt

Inhalt

- Abgrenzung
- Definition und Beispiele
- Datenstruktur Stack
- Aufrufbaum
- Rekursion als Problemlösungsstrategie
- Modulare Programmierung

“To iterate is human, to recurse, divine.”

-- anonymous

Abgrenzung: Iteration und Rekursion

- **Iteration**
  - Verfahren, das ein Problem schrittweise durch wiederholte Anwendung desselben Verfahrens-Schritts löst. Jeder dieser Iterationsschritte liefert Ergebnisse, die im nächsten Iterationsschritt weiter verwendet werden.
  - Abbruch der Iteration, wenn die Ergebnisse eines Iterationsschrittes eine Abbruchbedingung erfüllen.
  - Im Programm: ***Schleife***
- **Rekursion**
  - Verfahren, das ein Problem durch Zurückführen auf ein einfacheres Problem löst. Für die Lösung des einfacheren Problems greift das Verfahren auf sich selbst zurück.
  - Abbruch der Rekursion, wenn das einfachere Problem gelöst werden kann, ohne dass erneut auf das Verfahren selbst zurückgegriffen werden muss.
  - Im Programm: ***Rekursive Methode***

Beispiel: Fakultätsberechnung

Java

Iterativ

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ 1 \bullet 2 \bullet \ldots \bullet n & \text{für } n > 0 \end{cases}$$

```
static long fakul(long n) {
  long f = 1;
  for (long i=1; i<=n; i++)
    f = f * i;
  return f;
}
```

Rekursiv

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n \bullet (n-1)! & \text{für } n > 0 \end{cases}$$

```
static long fakul(long n) {
  if (n == 0)
    return 1;
  else
    return n * fakul(n-1);
}
```

Abgrenzung: Iteration vs. Rekursion

- Iterativer Algorithmus
  - Bei der Ausführung werden bestimmte Teile des Algorithmus mehrfach ausgeführt
- Rekursiver Algorithmus
  - Bei der Ausführung wird in bestimmten Teilen des Algorithmus der Algorithmus selbst direkt oder indirekt aufgerufen
  - **Vorteil** von rekursiven Algorithmen: Elegant und kurz für viele Problemstellungen
  - **Nachteil** von rekursiven Algorithmen: Manchmal ineffizient
- *Die rekursive Formulierung kann immer in eine äquivalente iterative Formulierung umgewandelt werden*
  - Explizite Auswertung und Verwaltung des *Aufruf-Stacks* (siehe Folgefolien)

Rekursive Methoden

Java

- Bei der Ausführung der Methode wird die Methode selbst direkt oder indirekt wieder aufgerufen.

  direkt: Im Rumpf der Methode `m` steht ein Aufruf der Methode `m`

  Indirekt: Im Rumpf der Methode `m` steht ein Aufruf der Methode **a**
  Im Rumpf der Methode **a** steht der Aufruf der Methode **b**
  Im Rumpf der Methode **b** steht der Aufruf der Methode ...
  ...
  Im Rumpf der Methode .. steht der Aufruf der Methode `m`

- Weiteres Beispiel: Einfache Potenzfunktion (vgl. **`Math.pow`**)

$$x^k = \begin{cases} 1 & \text{für } k = 0 \\ x \bullet x^{k-1} & \text{für } k > 0 \\ 1 \;/\; x^{-k} & \text{für } k < 0 \end{cases}$$

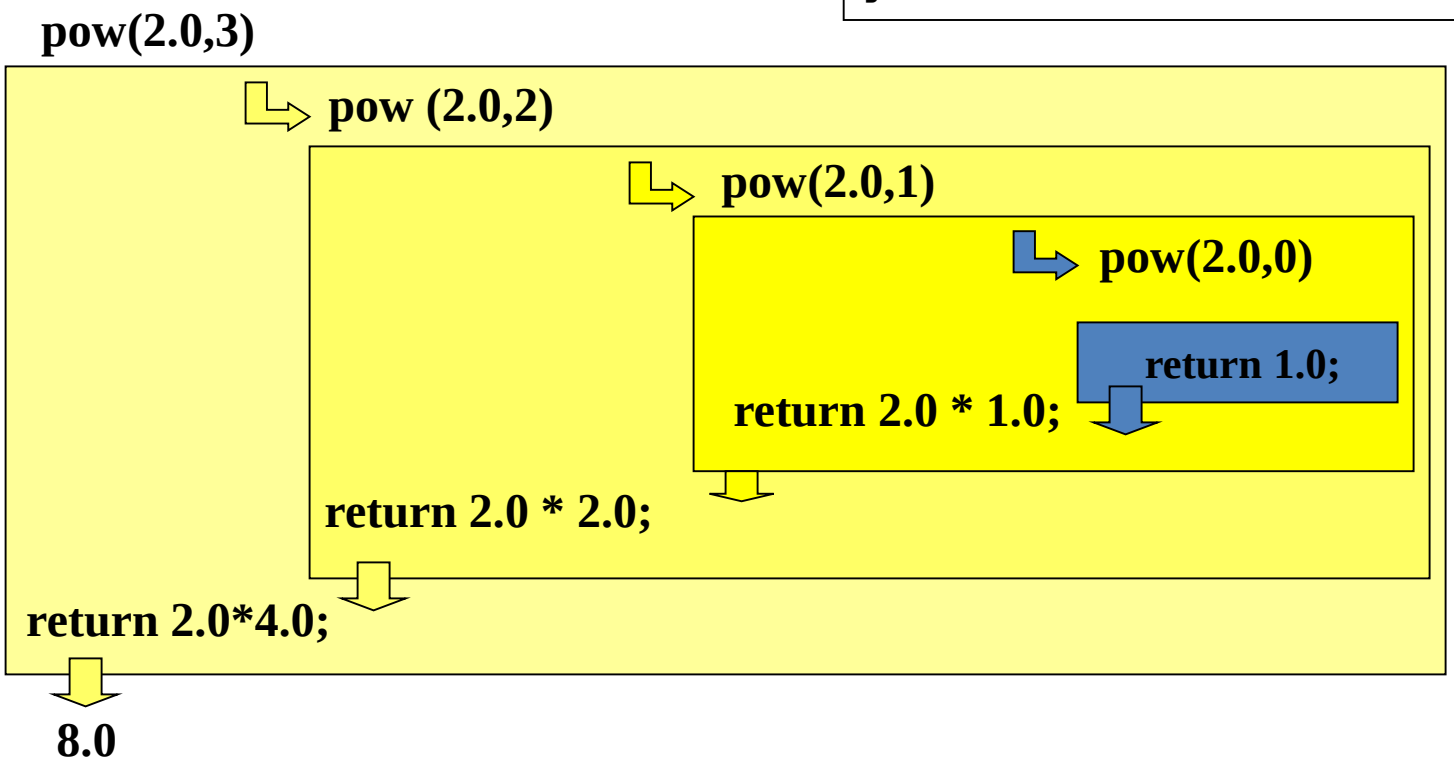
```
static double pow(double x, int k) {
  if (k == 0)
    return 1;
  else if (k > 0)
    return x * pow(x, k-1);
  else
    return 1 / pow(x, -k);
}
```

Rekursive Methoden (2)

Java

Ablauf des Methodenaufrufs

```
static double pow(double x, int k) {
  if (k == 0)
    return 1;
  else if (k > 0)
    return x * pow(x,k-1);
  else
    return 1 / pow(x,-k);
}
```



Rekursive Methoden (3)

Java

```
public static int summeVon1bis(int n) {
  return n + summeVon1bis(n-1);
}
```

Laufzeitfehler: Endlos-Rekursion!

- Wichtig!
  - Rekursionen müssen abbrechen
  - Ein oder mehrere einfache Fälle müssen gesondert behandelt werden
- Grundregel bei der Implementierung
  - Erst die nichtrekursiven einfachen Fälle implementieren
  - Danach erst den Rekursionsschritt programmieren

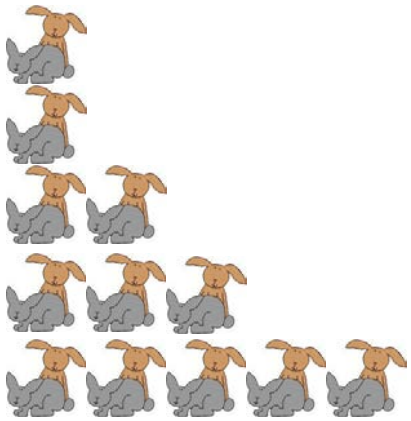
Rekursive Methoden (4)

- Fibonacci-Zahlen und Kaninchenvermehrung
- Leonardo Pisano Fibonacci formulierte 1190:

  Wenn ein neugeborenes Kaninchenpaar nach 2 Monaten ein neues Kaninchenpaar wirft und dann monatlich jeweils ein weiteres Paar und außerdem jedes neugeborene Paar sich auf die gleiche Art vermehrt, wie viele Kaninchenpaare gibt es dann nach ***n*** Monaten, wenn keines der Kaninchen vorher stirbt?

- Anzahl Kaninchenpaare nach ***n*** Monaten: **`fib(n)`**

```
fib(0) = 1
fib(1) = 1
fib(2) = 1 + 1 = 2
fib(3) = 2 + 1 = 3
fib(4) = 3 + 2 = 5
...
fib(n) = fib(n - 1) + fib(n – 2)
```



Rekursive Methoden (5)

Java

- Rekursive Fibonacci-Methode (sehr ineffizient!)

```
static long fib(long n) {
   if(n == 0)
     return 1;
   else if(n == 1)
     return 1;
   else
     return fib(n-1) + fib(n-2);
 }
```

n = 40
rekursiv fib(40) = 165580141
Laufzeit: 3157 ms

- Iterative Fibonacci-Methode

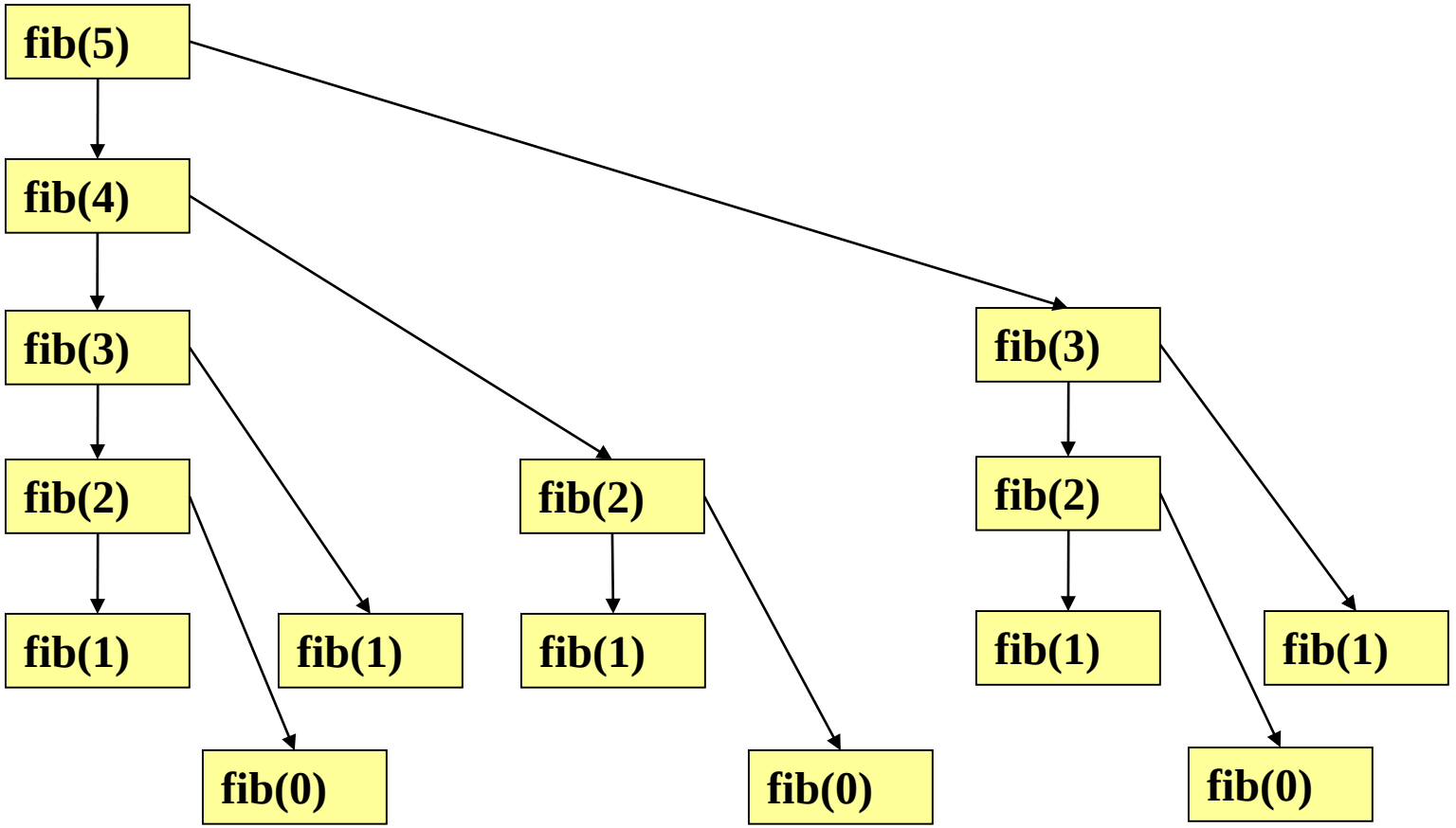
```
static long fib(long n) {
  long fN = 1, fNminus1 = 0, fNminus2;
  for(long i=0; i<n; i++) {
    fNminus2 = fNminus1;
    fNminus1 = fN;
    fN = fNminus1 + fNminus2;
  }
  return fN;
}
```

n = 40
iterativ fib(40) = 165580141
Laufzeit: 0 ms

Rekursive Methoden (6)

Java

Ablauf des rekursiven Methodenaufrufs am Beispiel **`fib(5)`**

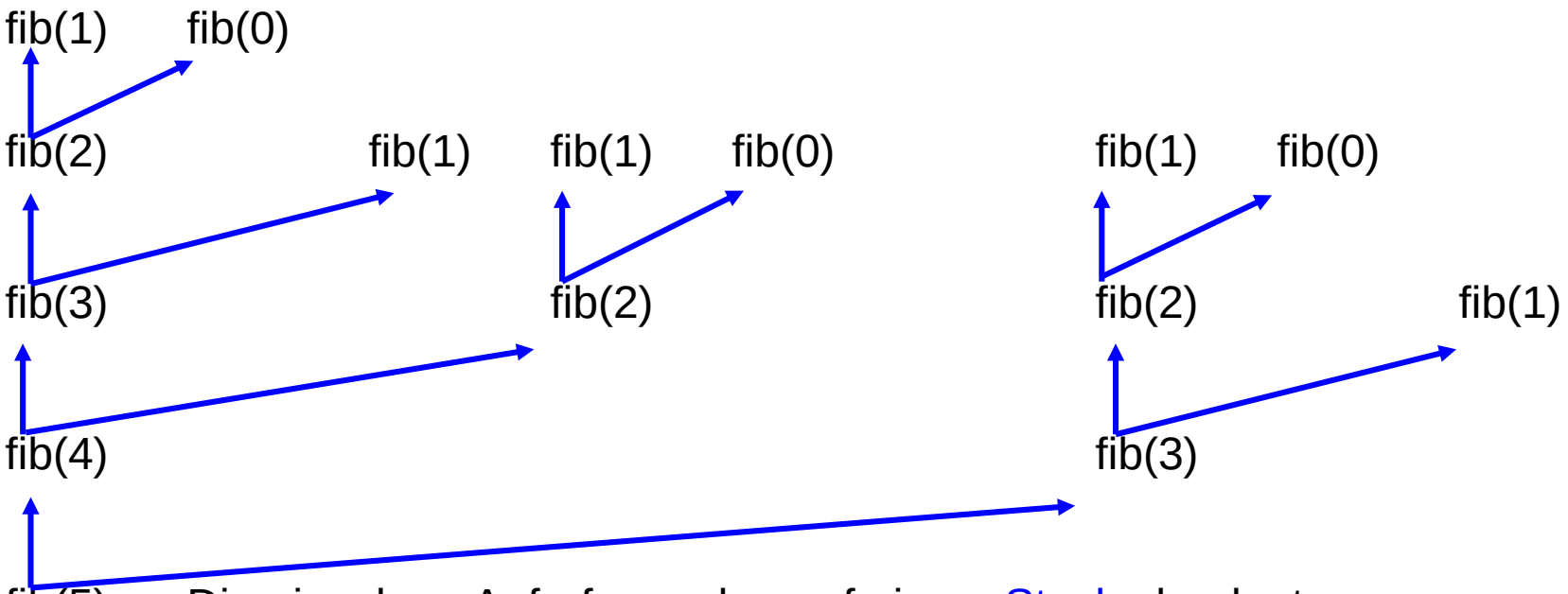


Rekursive Methoden (7)

Java

```
static long fib(long i) {
   if (i == 0) return 1;
   else if(i == 1) return 1;
      else return fib(i-1) + fib(i-2);
 }
```

Der Aufrufbaum der Methode für den Fall Fib(5) (gedreht analog zum Stack).



Die einzelnen Aufrufe werden auf einem Stack abgelegt

Rekursive Methoden (8)

- Die rekursive Formulierung eines Problems kann immer in eine äquivalente iterative Formulierung umgewandelt werden

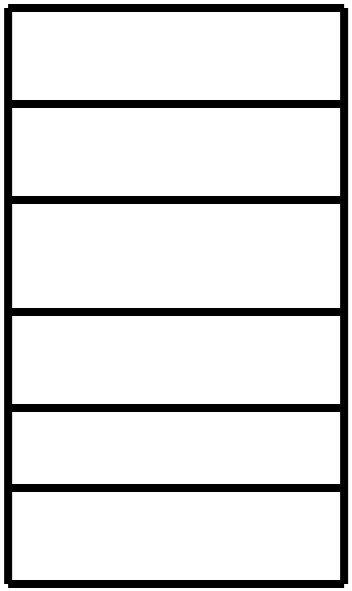
  → explizite Auswertung und Verwaltung des Prozedurstacks (Aufruf-Stacks)

  Datenstruktur: Stack

- Rekursion liefert eine wichtige allgemeine Methode zur Lösung algorithmischer Probleme

Einschub: Stack

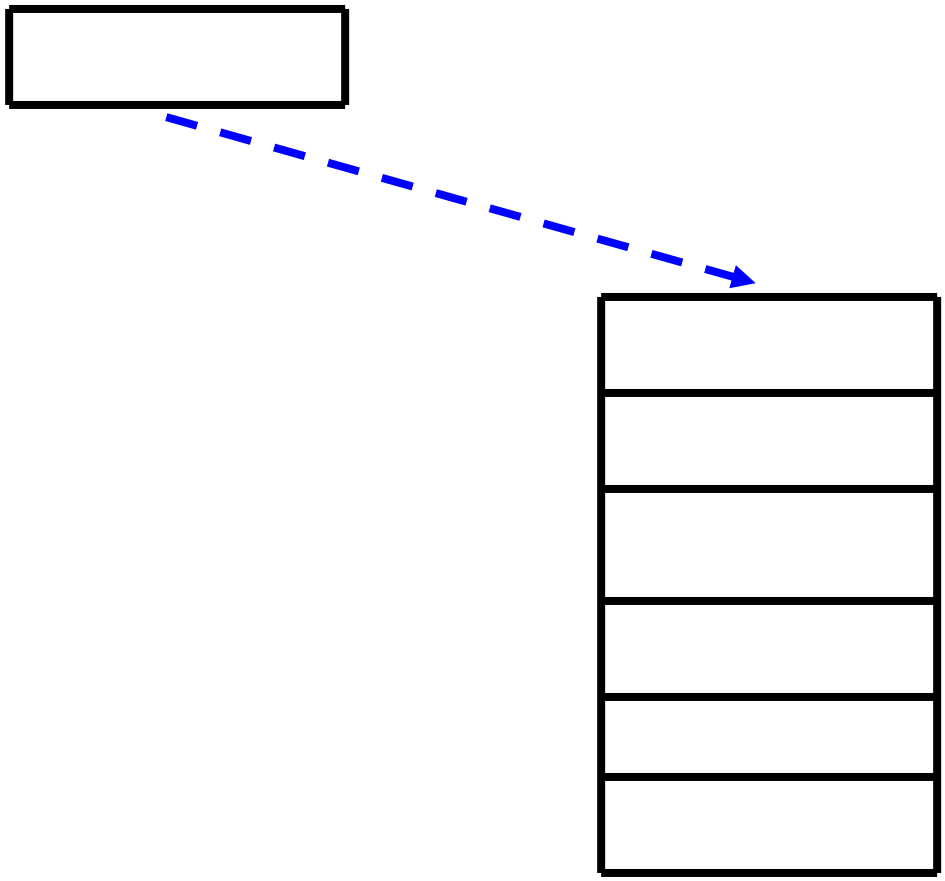
Datenstruktur: Stack (Keller, Stapel)



Einschub: Stack (2)

Datenstruktur: Stack

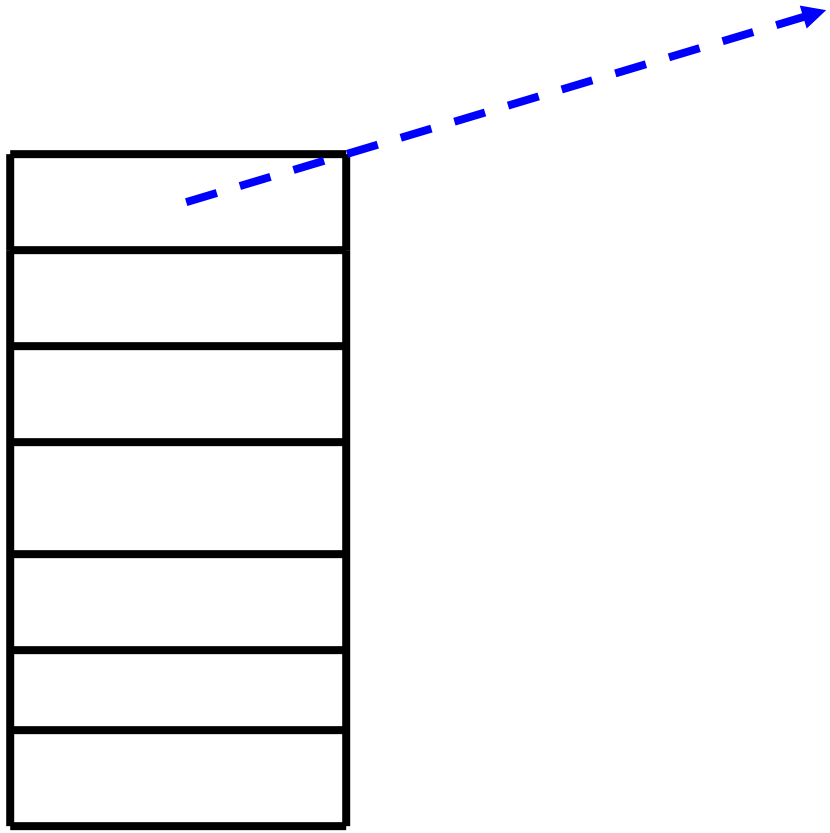
push - Ablegen eines neuen Elements auf dem Stack



Einschub: Stack (3)

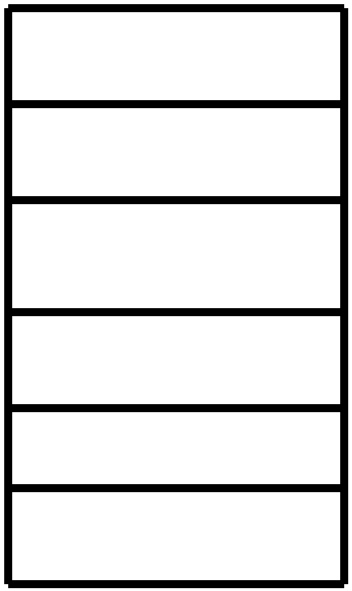
Datenstruktur: Stack

pop - Entfernen des obersten Elements vom Stack



Einschub: Stack (4)

- Weitere mögliche Operationen:
  - Erzeugen eines neuen Stack (new), Lesen des Inhalts des obersten Elements (top), Prüfen, ob der Stack leer ist (empty)



Einschub: Stack (5)

- Datenstruktur: Stack
- Ein Stack realisiert das LIFO-Prinzip: Last In First Out
- Praktische Realisierungsmöglichkeiten:
  - als Array (bei bekannter oberer Schranke für die Anzahl der Elemente
  - als linear verkettete Liste (siehe später dynamische Datenstrukturen, Listen, binäre Bäume)

Rekursion als Problemlösungsmethode


```
loese rekursiv ( Problem ) {
   if (Lösung des Problems trivial) { return Lösung }
   else {
            zerlege Problem in Teilprobleme;
            löse rekursiv ( Teilprobleme );
            setze die Lösungen der Teilprobleme
              zu einer Lösung des Gesamtproblems zusammen;
        }
}
```

Rekursion- Übung

Java

Übung: Was berechnet die folgende Methode?

```
// b muss positiv sein
// sonst unendliche Rekursion
public static int problem(int a, int b) {
  if (b == 1) return a;
  else return a + problem(a, b-1);
}
```

Methoden: Zusammenfassung

- Durch Methoden wird ausführbarer Code unter einem Namen zusammengefasst
- Dieser Code kann unter Verwendung von Parametern formuliert sein, denen später beim Aufruf der Methode Werte übergeben werden
- Die Übergabe der Parameter ist Pass-by-Value / Call-by-Value
- Methoden erlauben die Modularisierung des Programms durch die Abtrennung von Teilaufgaben, die in eigenständigen Einheiten zusammengefasst werden
- Um die Wiederverwendbarkeit (reusability) zu unterstützen sollte jede Methode möglichst nur eine einzelne, wohldefinierte Aufgabe haben (Vermeidung von Nebeneffekten) und einen Namen tragen, der die Aufgabe zum Ausdruck bringt
- Wenn dies nicht geht, so sollte man überlegen, ob man die Methode nicht in kleinere, überschaubarere zerlegt

Methoden: Zusammenfassung

- Man unterscheidet Prozeduren und Funktionen:
  - Prozeduren liefern keinen Rückgabewert
  - Funktionen liefern einen Rückgabewert

An jeder Stelle, an der ein bestimmter Wert verwendet werden kann, kann auch ein Methodenaufruf verwendet werden, der diesen Wert zurückliefert!

Regel 8 (der Goldenen Regeln der Code-Formatierung): Eine Methode mit einem Rückgabewert heißt Funktion. Eine Funktion besitzt stets nur eine einzige return-Anweisung. Diese steht in der letzten Zeile des Methodenblocks und lautet **return result;**

Methoden: Vorteile

- Unterstützung der Wiederverwendung bekannter Programmteile
- Erleichtern das Testen des Programms
- Jede Methode kann und sollte vor dem Gebrauch getestet werden
- Auch das Hauptprogramm ist testbar durch Verwendung von Testcode für die verwendeten Methoden
- Methoden können später leichter geändert werden, z.B. Ersetzung des Quelltextes im Methodenkörper durch ein effizienteres Programm
- Der definierte Zugriff auf gekapselte Daten (als `private` deklarierte Instanzvariablen) ist durch Methoden auf geregelte Weise möglich
  - Getter-Methoden: zum Auslesen der Werte solcher Variablen
  - Setter-Methoden: zum Setzen der Werte solcher Variablen
- Methoden unterstützen: Generalisierung, Vererbung, Kapselung und Polymorphie (siehe Teil Objektorientierung)

Methoden: Vorteile

- Methoden erleichtern Rekursion und damit Divide-and-Conquer-Verfahren (Teile-und-herrsche-Verfahren) und das Top-Down-Prinzip (schrittweise Verfeinerung)

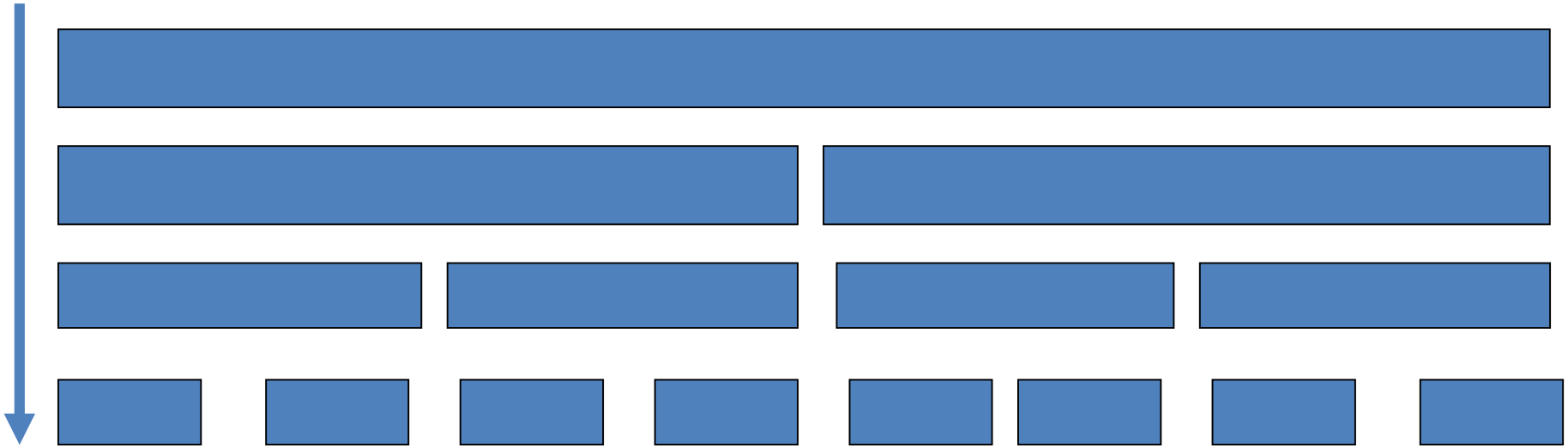
- Hierdurch wird das Paradigma der modularen Programmierung unterstützt

Divide-and-Conquer-Verfahren

- Bezeichnung für ein algorithmisches Lösungsverfahren, das sich in die beiden folgende Grundschritte gliedert:
  - **Divide-Schritt**: Das Problem wird in zwei oder mehr möglichst gleichgroße Teilprobleme derselben Art wie das Originalproblem aufgespalten, die unabhängig voneinander gelöst werden
  - **Conquer-Schritt:** Die Lösungen der Teilprobleme werden zu einer Lösung des Gesamtproblems zusammengefügt
- In der Praxis werden diese Schritte iteriert, was zu einem rekursiven Lösungsansatz führt.
- Typische Beispiele sind **Mergesort** und **Quicksort** (wie in Vorlesung ADS behandelt)

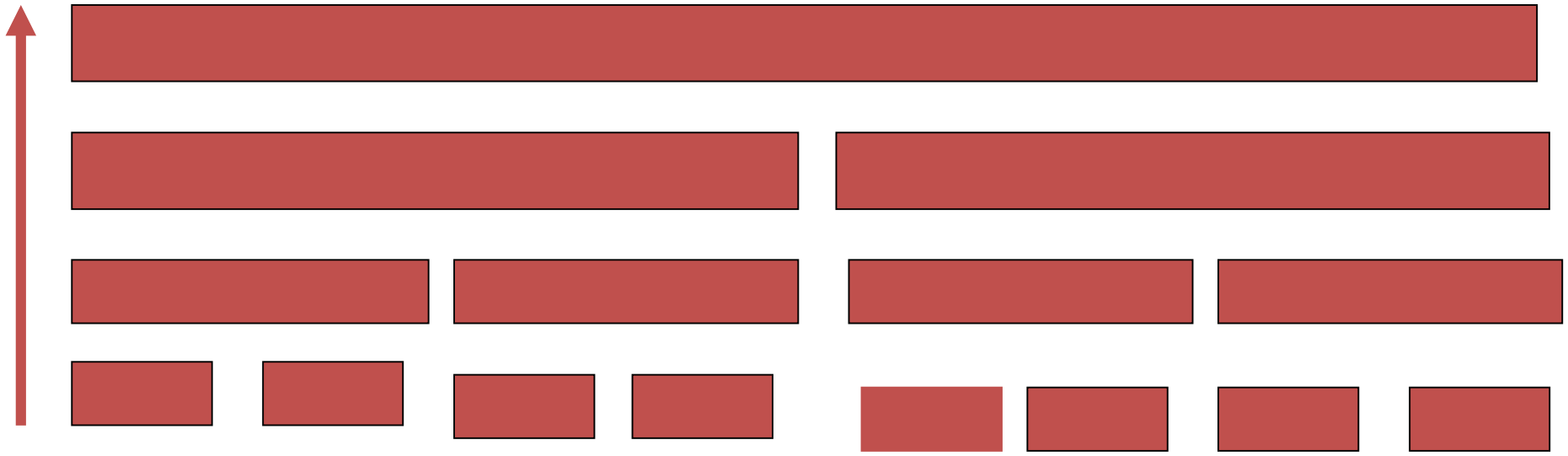
Divide-and-Conquer-Verfahren

Beispiel: Divide-Schritt



Divide-and-Conquer-Verfahren

Beispiel: Conquer-Schritt



Top-Down-Prinzip (schrittweise Verfeinerung)

- Modulare Programmierung
  - Dabei wird bei der Programmentwicklung nach dem Top-Down-Prinzip vorgegangen, indem man mit einem umfassenden abstrakten Modell beginnt und sich schrittweise verfeinernd hin zu einer konkreten Detaillierung und programmiersprachlichen Realisierung hinarbeitet

- Top-Down-Prinzip (schrittweise Verfeinerung)
  - Bei jedem Entwurfsschritt wird festgelegt, was die Untermoduln leisten sollen, nicht jedoch wie sie es leisten sollen
  - Die Funktionen der Moduln jeder Ebene werden nur durch die Funktionen der Moduln der unmittelbar darunter liegenden Ebenen realisiert
  - Es entsteht eine Hierarchie von Verfeinerungsstufen

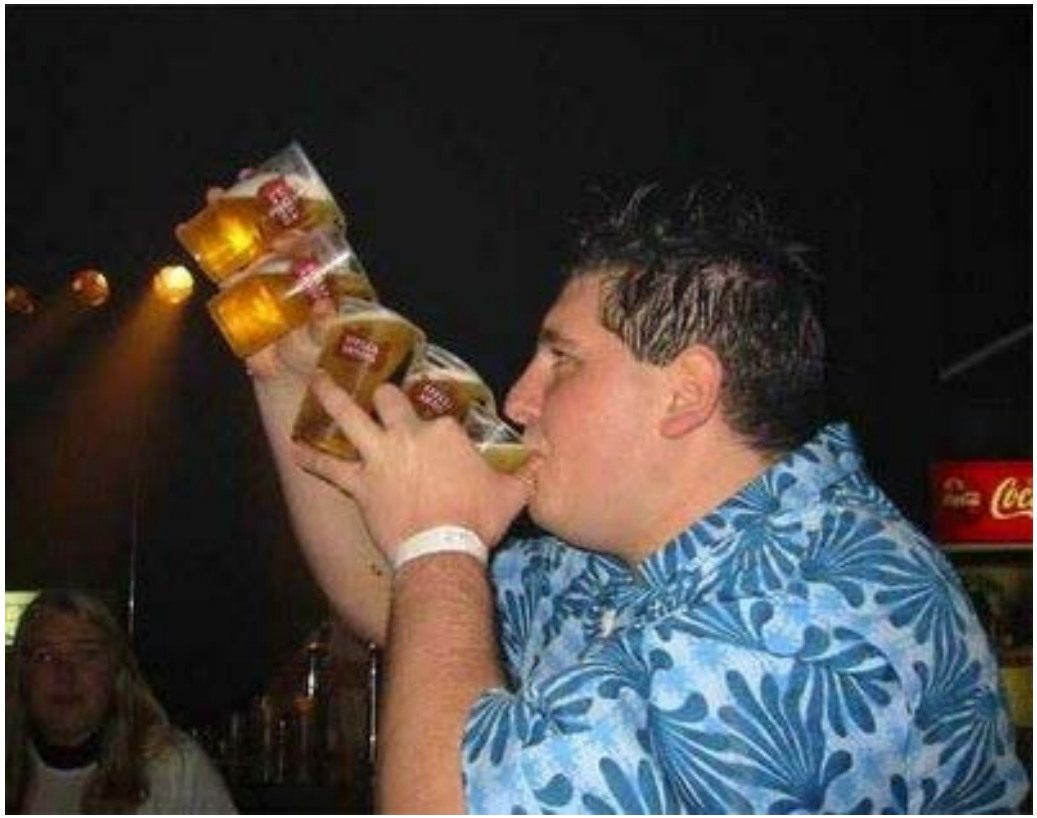
Modulare Programmierung

- Dabei wird ein Modul in der Praxis als ein in sich zusammenhängender Baustein aufgefasst, der stets folgende Eigenschaften besitzen sollte:
  - Er ist logisch oder funktional abgeschlossen
  - Wie er arbeitet oder implementiert ist, braucht außen nicht bekannt zu sein (information hiding)
  - Er besitzt klar definierte Schnittstellen nach außen hin
  - Er ist überschaubar und somit leicht testbar
  - Er sollte nur möglichst wenige andere Module verwenden
- Ein Softwaremodul ist eine geschlossene Programmeinheit, bestehend aus Konstanten, Variablen, Datentypen und Operationen, die genau definierte Funktionen ausführen und bestimmte Eigenschaften besitzen
- Ein System ist modular aufgebaut, wenn es aus abgrenzbaren Einheiten zusammengesetzt ist und wenn diese Einheiten einzeln ausgetauscht, verändert oder hinzugefügt werden können, ohne dass andere Teile des Systems hierdurch beeinflusst werden oder das System arbeitsunfähig wird

Historische Anmerkungen

- 1950 – 1970 wurde Programmieren innerhalb der Informatik als „Kunst“ bezeichnet
- Übergreifende Theorien speziell zur Entwicklung großer Programmsysteme fehlten weitgehend → Softwarekrise
- 1970 – 1990 entstand ein systematischeres strukturiertes Vorgehen
- Prozedurale und modularen Ansätze in der Programmierung waren erfolgreiche erste Schritte zu einem stärker strukturierten Vorgehen
- 1990 entstand das objektorientierte Paradigma (kommt bald in VL10)

Fragen



Danksagung

- Vorlesungsmaterialien von Prof. Dr. Detlef Seese wurden als Basis verwendet
- Unterstützung bei der technischen und inhaltlichen Gestaltung des Vorlesungsmaterials leisteten:

  Jóakim v. Kistowski

  Dietmar Ratz, Joachim Melcher, Roland Küstermann, Jana Weiner, Hagen Buchwald, Matthes Elstermann, Oliver Schöll, Niklas Kühl, Tobias Diederich