Teil III

Objektorientierung und SW-Entwicklung

K. Bothe

III: Objektorientierung und SW-Entwicklung

1. Grundkonzepte der Objektorientierung (1): abstrakte Datentypen, Objekte, Klassen
2. Objektorientierung: Grundlegende Fallbeispiele
3. Grundkonzepte der Objektorientierung (2): Klassenmethoden, Klassenvariablen
4. Komponentenarten
5. Grundkonzepte der Objektorientierung (3): Vererbung, Polymorphismus, dynamisches Binden
6. Grundkonzepte der Objektorientierung (4): Generische Klassen
7. Verkettete Strukturen: Listen
8. Grundkonzepte der Objektorientierung (5): Interface

III: Objektorientierung und SW-Entwicklung

9. Ausnahmebehandlung
10. Softwareentwicklung: Anforderungsanalyse und Problemdefinition - ein Beispiel
11. (Objektorientierte) Softwarearchitekturen
12. Vom Entwurf zur Implementation
13. Bäume: effektives Suchen und Sortieren
14. Applets
15. Ereignisse (Events): EyesApplet

    (ein Beispiel zu Applets, Events, Graphics)
16. Parallelität: Threads

1. Grundkonzepte der Objektorientierung (1):

Abstrakte Datentypen, Objekte, Klassen

Java-Beispiele:

Stack.java
Umkehrung.java
Umkehrung2.java

Schwerpunkte:

- Programmier-Paradigmen:
  imperativ und objektorientiert
- Datenabstraktion – abstrakte Datentypen – Klassen – Instanzen
- Instanzvariablen, Instanzmethoden
- Beispiel: Stack
- Klassifikation von Programmiersprachen

Vorkenntnisse?

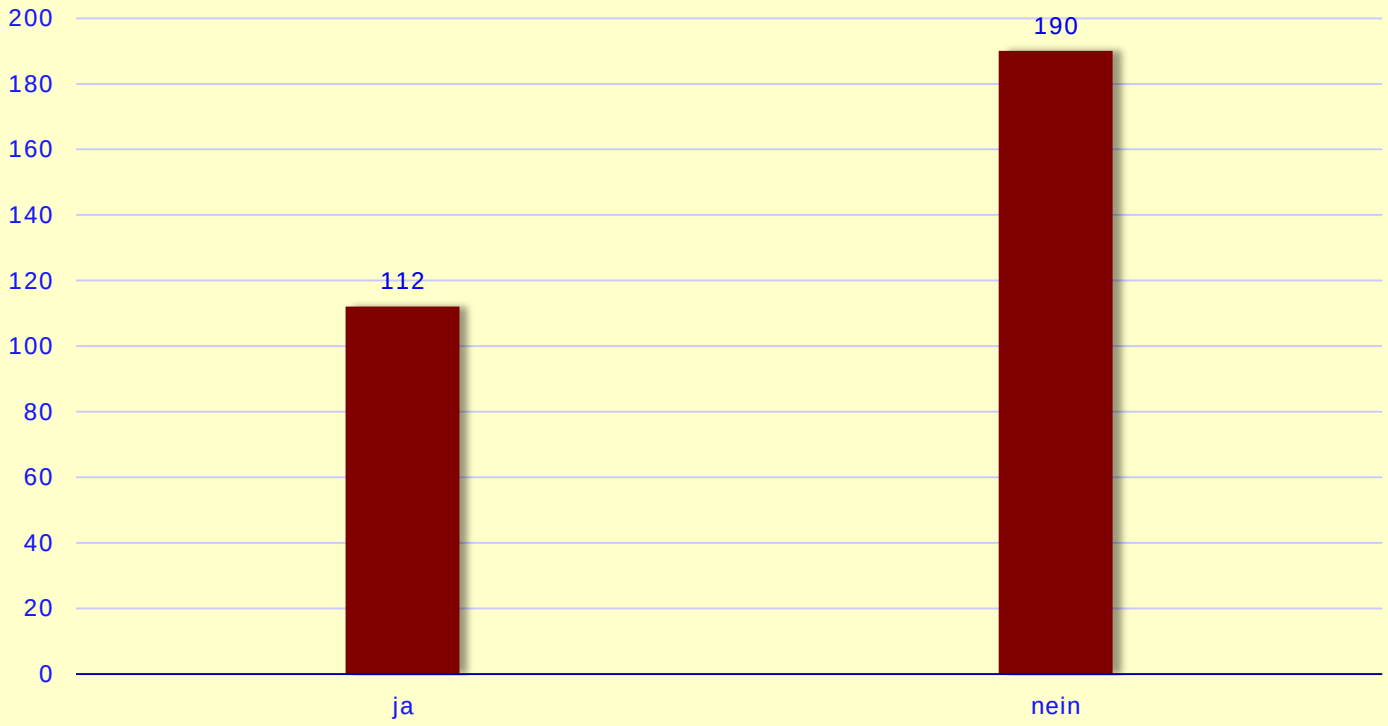
Was ist objektorientierte Programmierung?

Was sind Objekte in der Programmierung?

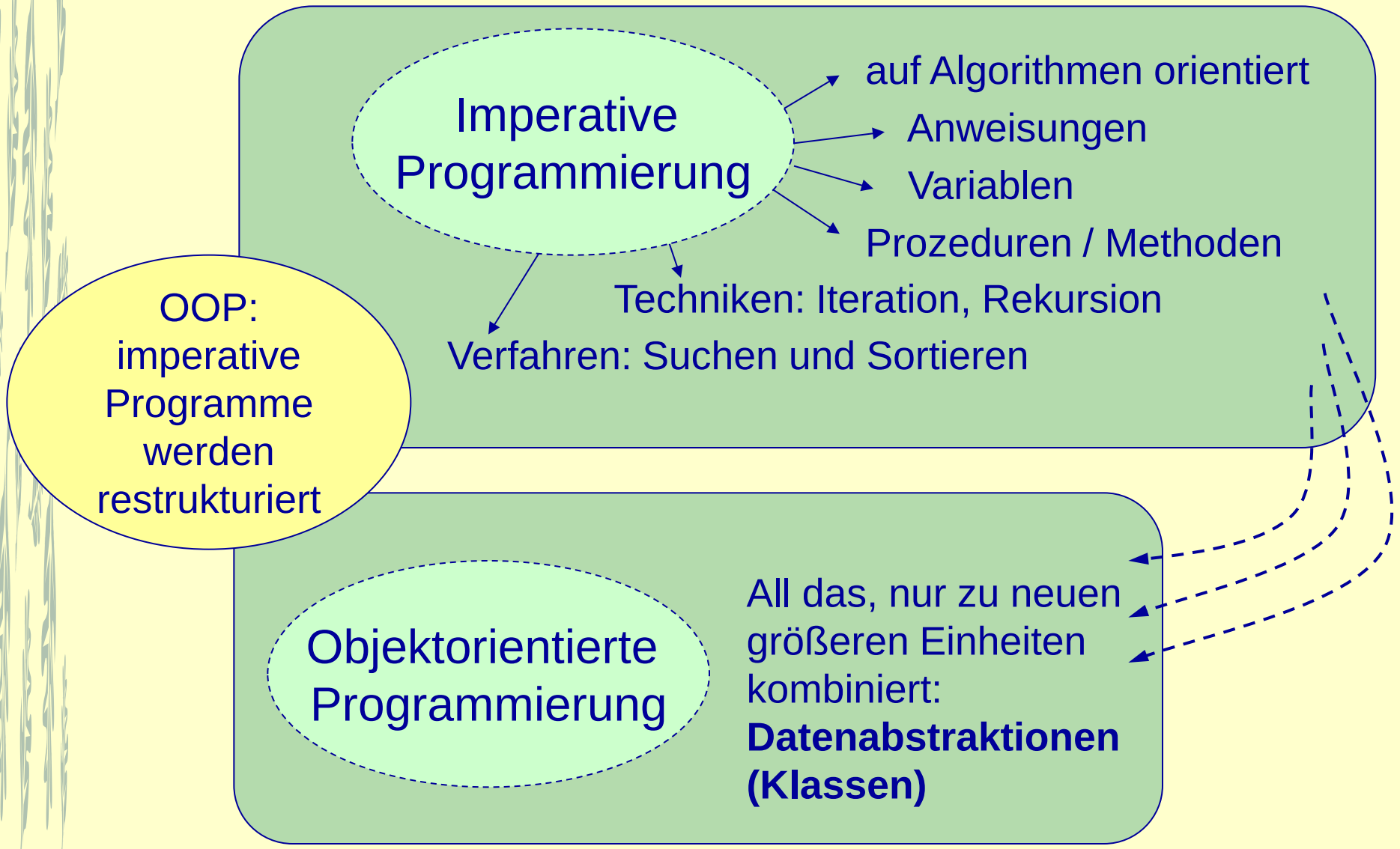
Ist objektorientierte Programmierung etwas völlig anderes – und müssen wir jetzt alles vergessen?

Vorkenntnisse: Fragebogen Oktober 2015

Wissen Sie, was objektorientierte Programmierung ist?



Imperative Programmierung:
das ‚Handwerkszeug' der objektorientierten Programmierung



OOP: imperative Programme werden restrukturiert

Imperatives Programm

```
class Zeitplan {
   private static int hour, minute;

   private static void addMinutes(int m) {
      ...
      hour = totalMinutes/60;
      ...
   }
   private static void printTime() {
      if ((hour == 0) ...
   }

   private static void includeNewEntry(...) {
      ...
   }
   public static void main (...) {
      hour = 8; minute = 30;
      addMinutes(30); printTime(); ...
   }
}
```

Objektorientiertes Programm

→ Herauslösen von problemorientierten Bausteinen (Klassen)

Zeitverwaltung

```
class Time {
   private int hour, minute;
   public Time() ...
   public addMinutes ...
   public printTime ...
   public timeInMinutes ...
   public printTimeInMinutes ...
}
```

```
class Schedule {
   public static includeNewEntry
...
   public static main ... {
     Time t1 = new Time(8,30);
     ...
  }
}
```

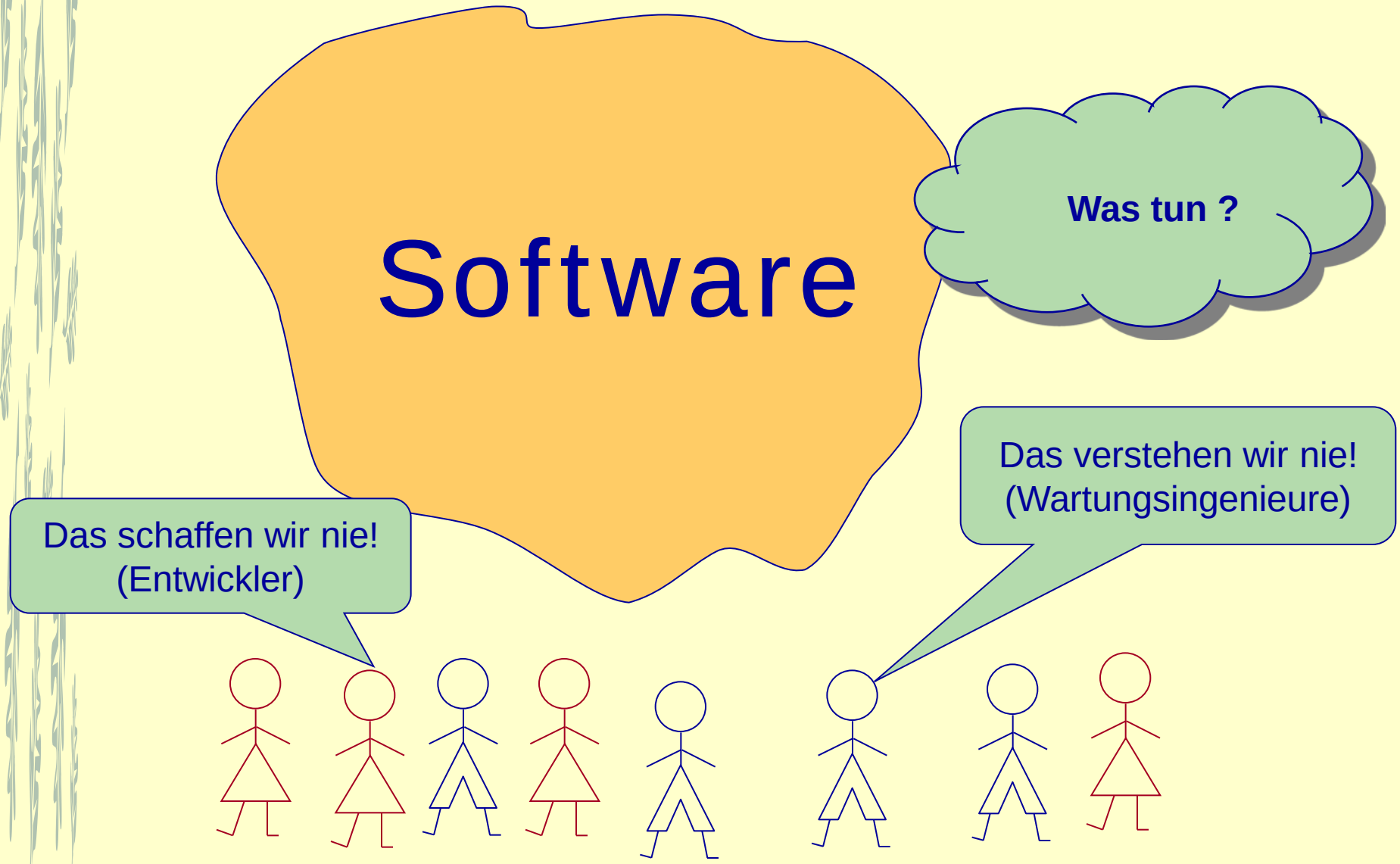
Terminkalender erzeugen

Im Wesentlichen derselbe Programmcode – nur anders strukturiert

Noch einmal:

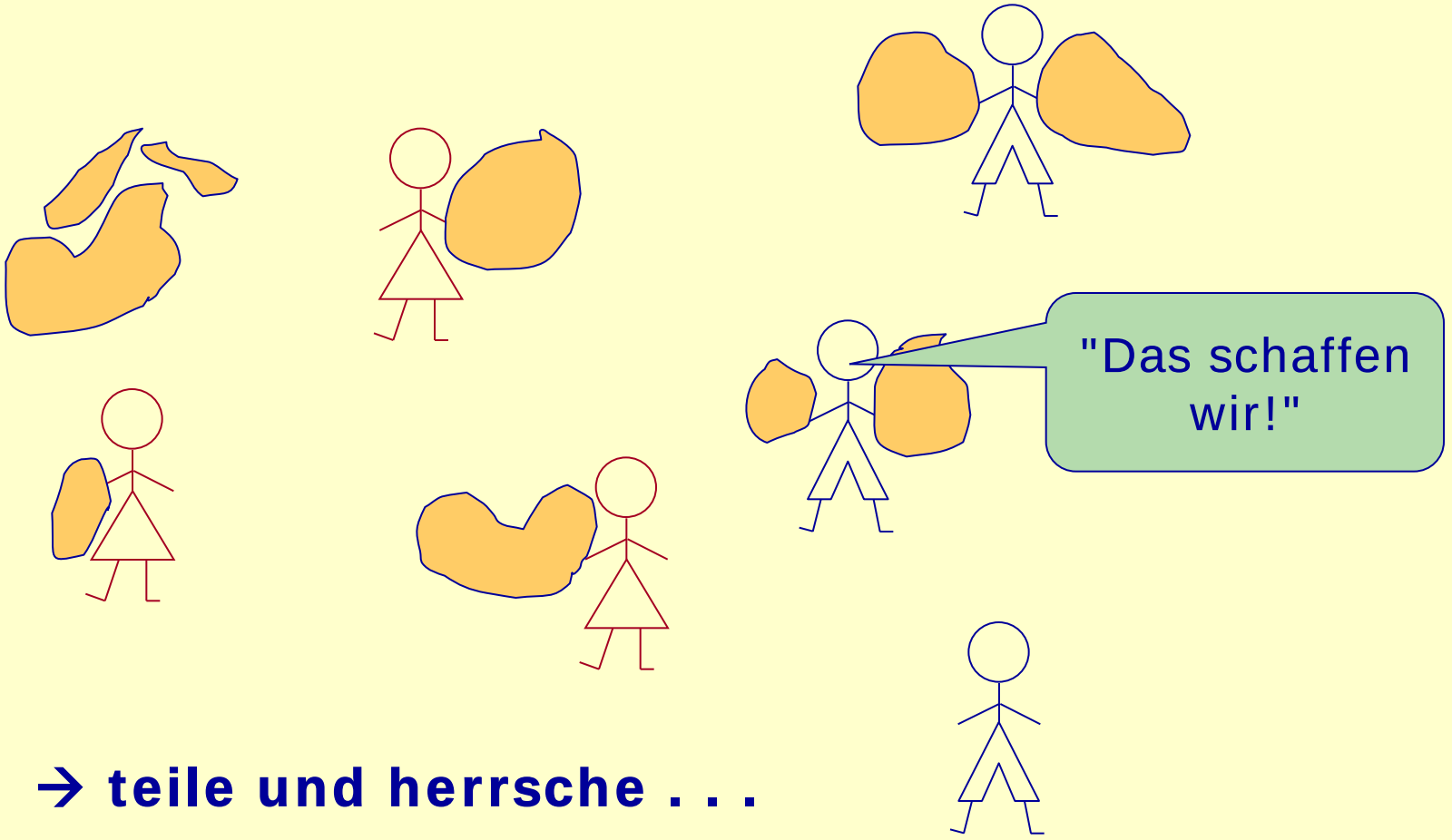
Abstraktionen in der Software-Entwicklung

Software kann sehr komplex sein...



Software kann sehr komplex sein...

➔ Zerlegung in Komponenten

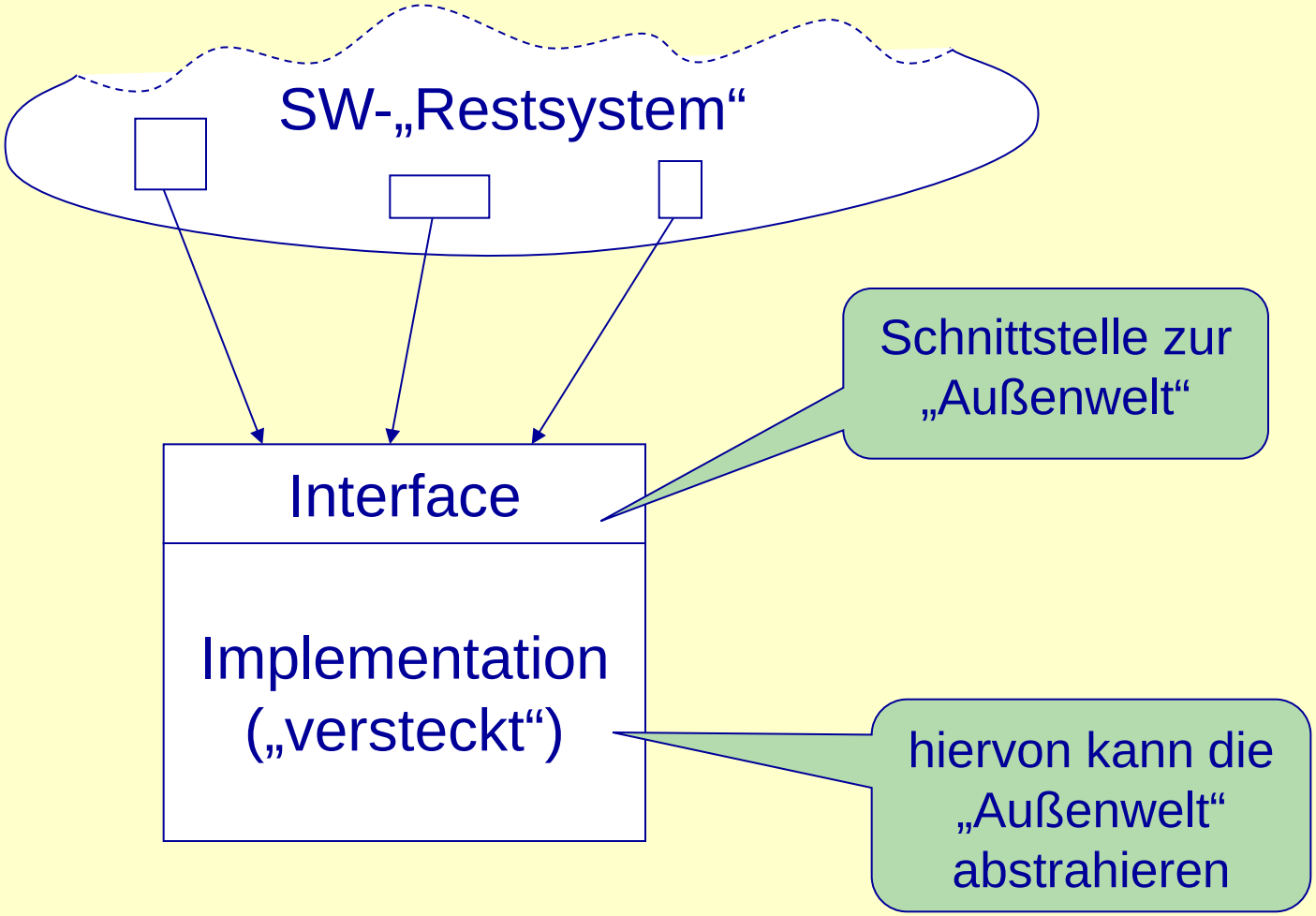


➔ teile und herrsche . . .

Komplexität von Programmen bewältigen

- **Dekomposition**:
  Zerlegung des Programms in Komponenten
- **Abstraktion**:
  Unwesentliches für die Nutzung einer Komponente weglassen
  → Komponente:
  Interface und Implementation

Komponente: Interface + Implementation



Prozedurale Abstraktion: Methoden / Prozeduren

"Restsystem":
`fak = fakultaet (10);`

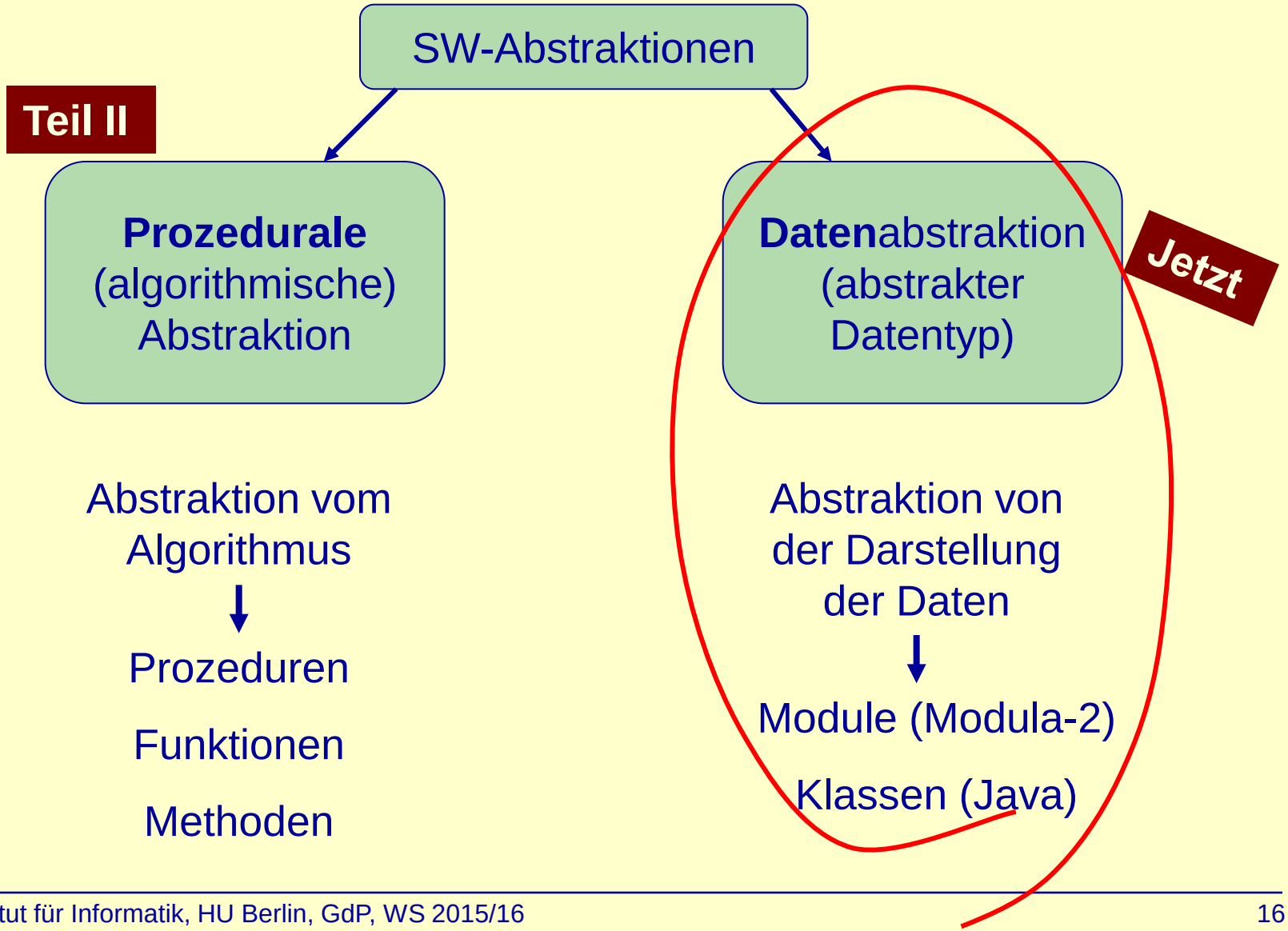
```
public static int fakultaet (int n)
```

Interface

```
{
    int x , fak = 1 ;
    for ( x = 1; x < = n ; x ++ )
        fak = fak * x ;
    return fak ;
}
```

Implementation

Grundformen von SW-Abstraktionen



Abstrakter Datentyp (Datenabstraktion)

Abstrakter Datentyp:

Einheit aus Daten und Operationen

- Operationen: dienen der Bearbeitung der Daten (Initialisieren, Ändern, Lesen, Löschen)
- Daten: vor der "Außenwelt" geschützt / "versteckt"

… wird als Klasse realisiert.

Wie arbeitet das „Restsystem“ mit den Daten? Interface eines abstrakten Datentyps

Interface:
Operationen

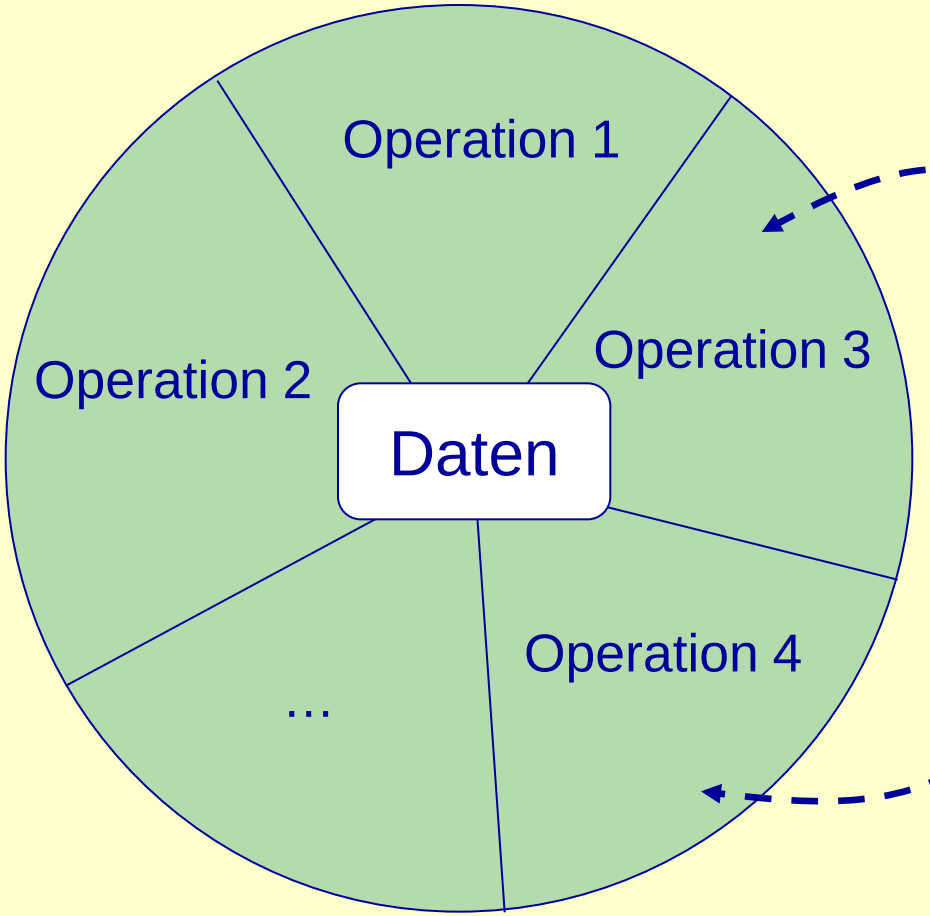
Implementation:
("versteckt")

- Art der Datendarstellung
  z.B. int[] sort;
- Implementation der Operationen
  (Algorithmen)

genauer: „Köpfe“ der Operationen, d. h. Signatur (Name, Parameter)

→ Operationen stellen für die Außenwelt die einzige Möglichkeit dar, mit den Daten zu arbeiten

Abstrakte Datentypen: versteckte Information (information hiding)



Nutzer (Restsystem)

Information hiding:
- Daten versteckt
- Daten gekapselt

Fallbeispiel:

der abstrakte Datentyp ‘Stack’

… zur Lösung der nachfolgenden Aufgaben

Programmieraufgaben

1. Einlesen einer Folge von Daten (char) und Ausgabe in umgekehrter Reihenfolge

```
Eingabe: a z d f g k
Ausgabe: k g f d z a
```

2. Überprüfung der Klammerstruktur eines Programms (paarweises Auftreten, ohne weitere Syntaxanalyse)

```
( a + ( x [ ( i + j ) ] % 12 ) {
      z [ i ] ++ ;
} ...
```

3. Überführung von Ausdrücken in Postfixform

```
  a + b * c -> a b c * +
(a + b) * c -> a b + c *
```

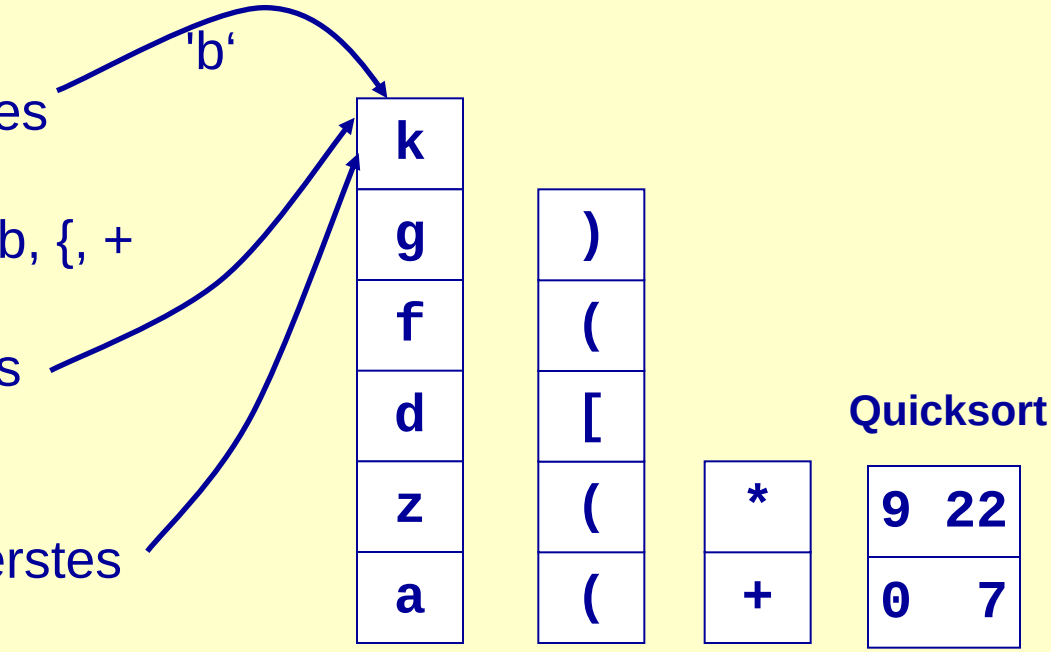
4. Auflösung der Rekursivität in Iteration

```
Türme von Hanoi, Quicksort
```

Lösungsidee: der abstrakte Datentyp 'Stack' (Keller)

Stack (Stapel, Keller):
Folge von Elementen, die nur von einer Seite her (‚von oben‘) bearbeitet werden kann (lesen, abspeichern, streichen), LIFO-Prinzip – Last In First Out

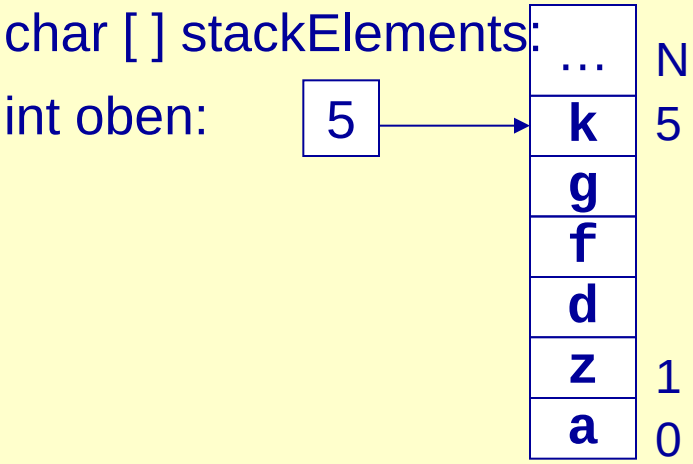
- **abspeichern:** neues oberstes Element
  z.B. von Zeichen b, {, +
- **lesen:** nur oberstes Element
- **streichen:** nur oberstes Element



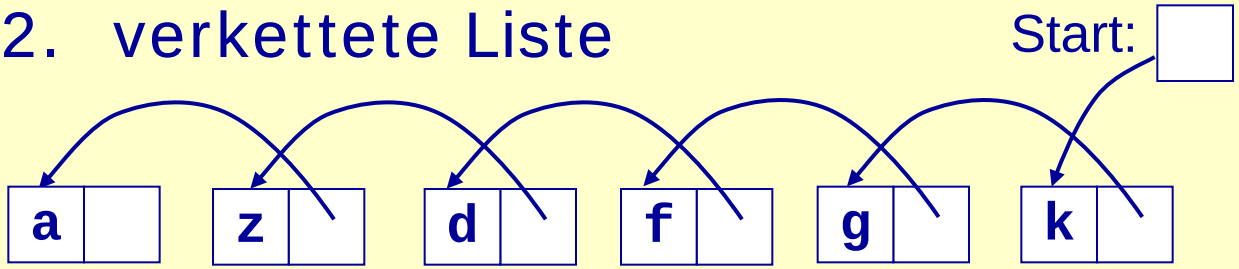
Queue (Warteschlange): FIFO-Prinzip – First In First Out

Datendarstellung für Stacks

1. Array mit Index des letzten Elements



2. verkettete Liste

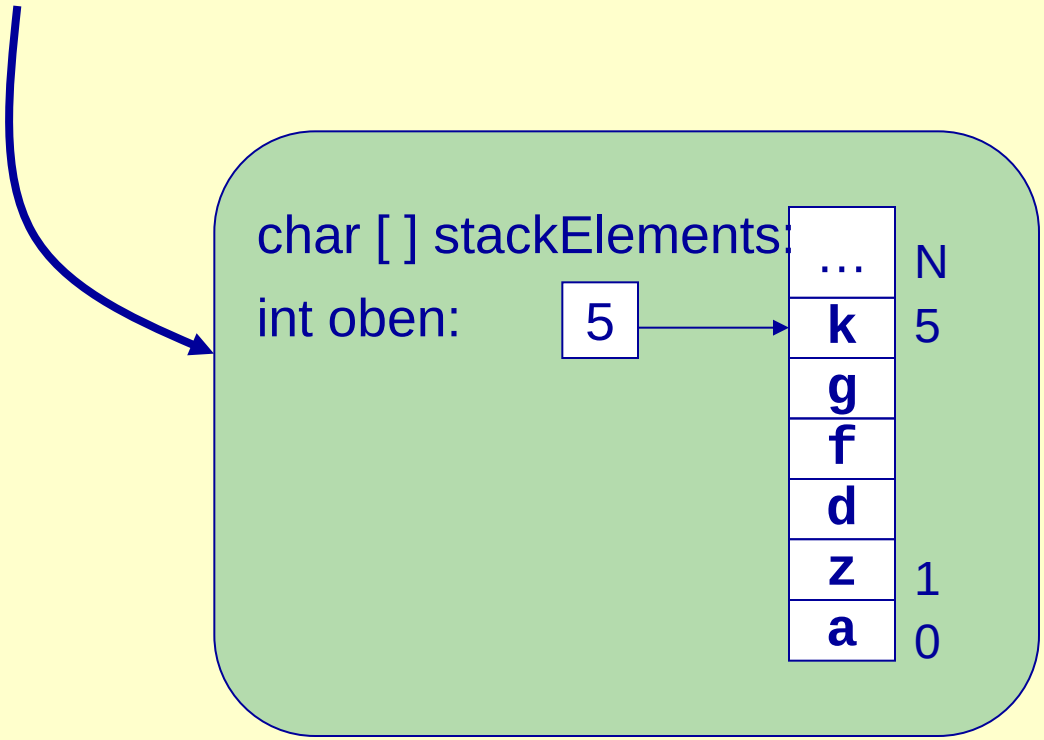


Nutzung des Stack
- unwichtig: Datendarstellung
- wesentlich: Zugriffsoperationen

Nutzersicht

- Unwichtig: Datendarstellung (Array oder Liste)
- Wesentlich: Zugriffsoperationen

Mit welchen Operationen können die Daten bearbeitet werden?



Operationen für Stacks

- erzeugen: leerer Stack (der Länge n)
- abspeichern: neues Element nach oben
- lesen: oberstes Element lesen
- streichen: oberstes Element streichen
- testen: ist der Stack leer?

Signatur: Name, Definitionsbereich, Wertevorrat

```
newStack:  int          → stack
push:      stack x char → stack
top:       stack        → char
pop:       stack        → stack
isempty:   stack        → boolean
```

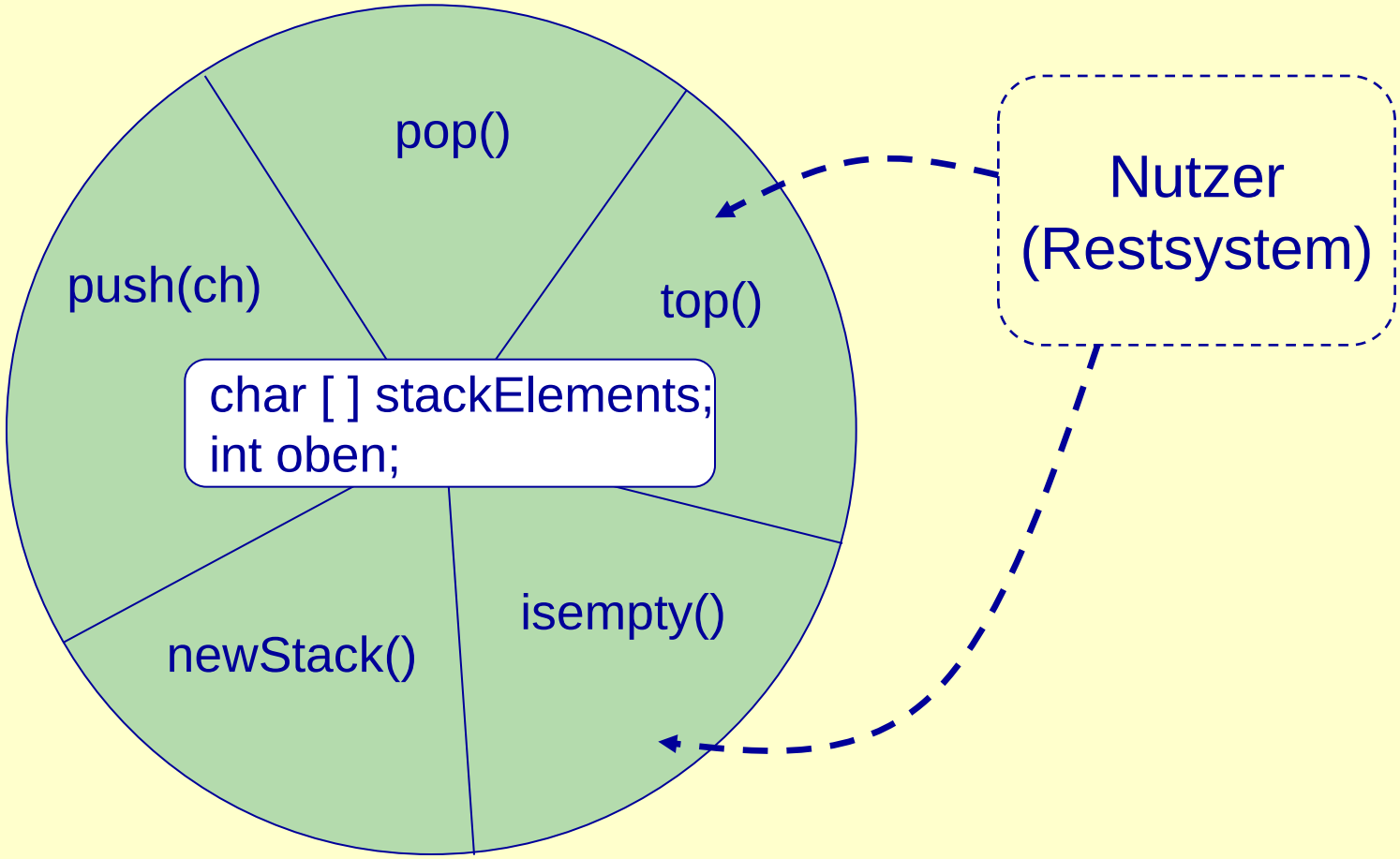
Abstrakte Datentypen in Java: Realisierung als Klassen

Abstrakter Datentyp (Datenabstraktion)

Abstrakter Datentyp:

Einheit aus versteckten Daten und Zugriffsoperationen
(Initialisieren, Ändern, Lesen, Löschen)

Abstrakter Datentyp Stack: versteckte Information (information hiding)



Abstrakte Datentypen in Java: Stack

Stack.java

```
class Stack {

    private char[] stackElements;

    private int top;  // zeigt auf oberstes Element

    public Stack(int n) {
        stackElements = new  char [n];
        top = -1;
    }

    public boolean isempty() {
        return top == -1;
    }

    public void push(char x) {
        top++; stackElements[top] = x;
    }

    public char top() {
        if (isempty()) {
          System.out.println("Stack leer");
          return ' ';
        }
        else
          return stackElements [top];
    }

    public void pop() {
        if (isempty())
          System.out.println("Stack leer");
        else
          top--;
    }
}
```

ADT → Java-Klasse

Abstrakte Datentypen in Java: wesentliche äußere Merkmale

Stack.java

```
class Stack {

    private char[] stackElements;

    private int top;  // zeigt auf oberstes Element

    public Stack(int n) {
        stackElements = new  char [n];
        top = -1;
    }

    public boolean isempty() {
        return top == -1;
    }

    public void push(char x) {
        top++; stackElements[top] = x;
    }

    public char top() {
        if (isempty()) {
          System.out.println("Stack leer");
          return ' ';
        }
        else
          return stackElements [top];
    }

    public void pop() {
        if (isempty())
          System.out.println("Stack leer");
        else
          top--;
    }
}
```

Variablen: ‚private' → ‚versteckt'

Konstruktor-methode

Methoden: ‚public' → nach ‚außen' sichbar

Variablen, Methoden:
nicht mehr ‚static'
→ Instanzvariablen, -methoden
→ Instanzen (Objekte) der Klasse erzeugen

Zum Vergleich: Imperatives Programm

```
class ZeitPlan {

   private static int hour, minute;

   private static void addMinutes (int m)
   private static int timeInMinutes ()
   private static void printTime ()
   private static void printTimeInMinutes ()
   private static void includeNewEntry
                          (int intervalInMinutes)
   public static void main (String[] args) {
        ...
        includeNewEntry(90, "V PI1");
        includeNewEntry(15, "Pause");
        includeNewEntry(90, "V ThI1");
        ...
   }
}
```

Alles „static“:
- Keine Objektorientierung
- Imperativer Stil: Variablen werden durch Methoden bearbeitet

Java-Klassen: definieren ADT

```
class Stack {

    private char[] stackElements;

    private int top;  // zeigt auf oberstes Element

    public Stack(int n) {
        stackElements = new  char [n];
        top = -1;
    }

    public boolean isempty() {
        return top == -1;
    }

    public void push(char x) {
        top++; stackElements[top] = x;
    }

    public char top() {
        if (isempty()) {
          System.out.println("Stack leer"
          return ' ';
        }
        else
          return stackElements [top];
    }

    public void pop() {
        if (isempty())
          System.out.println("Stack leer"
        else
          top--;
    }
}
```

Klasse definiert neuen (abstrakten) Typ
Typname = Klassenname

Variablen deklarieren: von diesem Typ

```
int i;
int[] sorted;
Stack s;
```

Werte des neuen Typs heißen
Objekte oder **Instanzen** der Klasse

Erzeugung eines Objekts (vgl. Arrays):

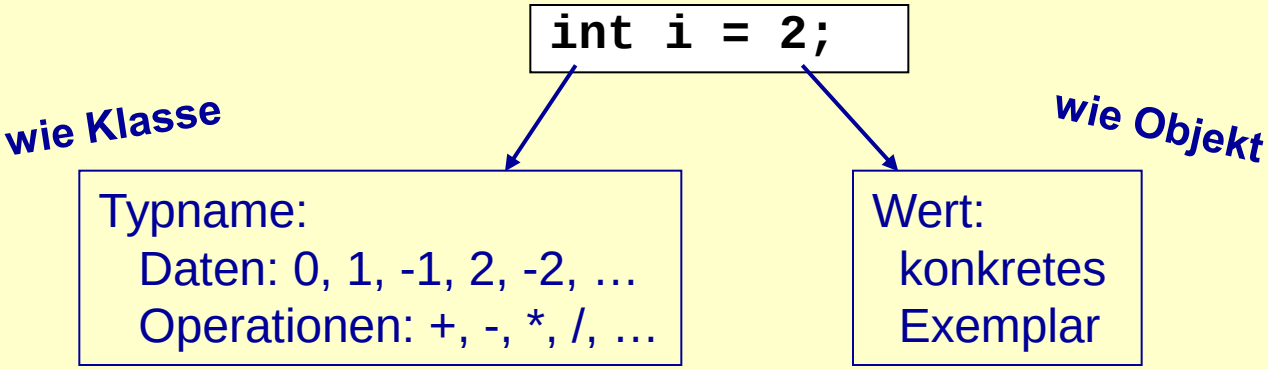
```
s = new Stack(n);
```

Klassen und Objekte

Klasse =
Beschreibung des neuen Typs
(Wie sehen Daten und Operationen des Typs aus?)

Objekt =
konkretes Exemplar des Typs; ***Instanz*** der Klasse;
existiert physisch im Speicher: new Stack(n)

Vergleiche mit vordefinierten Datentypen:



Aufbau von Klassen zur Beschreibung ADT

```
class Stack {

  private char[] stackElements;

  private int top;

  public Stack(int n) {
     ...
  }

  public void top() {
     ...
  }

  public boolean isempty() …

}
```

Instanz-Variablen: Beschreibung der **Daten** (Repräsentation)

Erzeugung und Initialisierung: Konstruktor (-Methode)

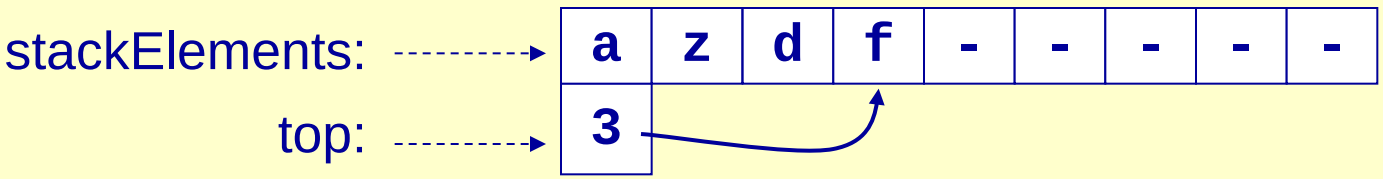
Instanz-Methoden: Beschreibung der Operationen

Datendarstellung in der Klasse Stack

```
class Stack {
   private char[] stackElements;
   private int top;
   ...
}
```

Entspricht 1. Variante (s. o.):
Array mit Verweis auf letztes Element

Beispiel für Stack der Länge 4:

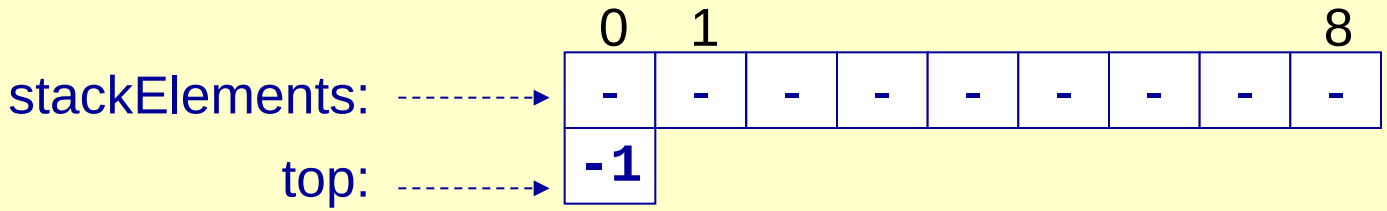


Initialisierung der Klasse Stack

Konstruktor Stack() erzeugt ein Objekt und initialisiert die (Instanz-)Variablen

```
class Stack {
   private char[] stackElements;

   private int top;

   public Stack(int n) {
      stackElements = new char [n];
      top = -1;
   }
   …
}
```

Aufrufbeispiel: s = new Stack(9);



(Instanz-) Methoden in Stack: Instanzvariablen bearbeitet

```
class Stack {
   private char[] stackElements;

   private int top;

   public boolean isempty() {
      return top == -1;
   }

   public void push(char x) {
      top++;
      stackElements[top] = x;
   }
   …
}
```

Instanzmethoden: Aufruf

Anwendung auf Objekte:

```
variable . methode ( Parameter )
```

Beispiele:

```
Stack s;
s = new Stack(9);

s.push('+');
x = s.top();
s.pop();
```

Speichere (mit ‘push‘) das Zeichen '+' in das durch die Variable s referenzierte Objekt

Klassen nutzen:

Objekte erzeugen

Umkehrung.java

Umkehrung2.java

Programmieraufgaben

1. Einlesen einer Folge von Daten (char) und Ausgabe in umgekehrter Reihenfolge

```
Eingabe: a z d f g k
Ausgabe: k g f d z a
```

2. Überprüfung der Klammerstruktur eines Programms (paarweises Auftreten, ohne weitere Syntaxanalyse)

```
( a + ( x [ ( i + j ) ] % 12 ) {
            z [ i ] ++ ;
            } ...
```

3. Überführung von Ausdrücken in Postfixform

```
   a + b * c -> a b c * +
(a + b) * c -> a b + c *
```

4. Auflösung der Rekursivität in Iteration

Anwendung des Stack: Umkehrung einer Zeichenkette

```
class Stack {

    private char[] stackElements;

    private int top;  // zeigt auf obers

    public Stack(int n) {
        stackElements = new char [n];
        top = -1;
    }

    public boolean isempty() {
        return top == -1;
    }

    public void push(char x) {
        top++; stackElements[top] = x;
    }

    public char top() {
        if (isempty()) {
          System.out.println("Stack leer
          return ' ';
        }
        else
          return stackElements [top];
    }

    public void pop() {
        if (isempty())
          System.out.println("Stack leer
        else
          top--;
    }
}
```

Umkehrung.java

```
public class Umkehrung {
  public static void main (String[] argv) {

    int n;
    char ch;
    Stack s;

    System.out.print("Groesse des Stacks: ");
    n = Keyboard.readInt();
    s = new Stack(n);

      // n Elemente einlesen und in den Stack ...
    System.out.println("Gib mindestens " + ... ");
    for (int i = 0; i < n; i++) {
      ch = Keyboard.readChar();
      s.push(ch);
    }

    System.out.println ("Umgekehrte Reihenfolge ...");
    while (!s.isempty()) {              // solange ...
        System.out.print(s.top());      // drucke ...
        s.pop();                        // oberstes ...
    }
    System.out.println();
  }
}
```

Anwendung des Stack: Umkehrung einer Zeichenkette

Umkehrung.java

```
public class Umkehrung {
  public static void main (String[] argv) {

    int n;
    char ch;
    Stack s;

    System.out.print("Groesse des Stacks: ");
    n = Keyboard.readInt();
    s = new Stack(n);

      // n Elemente einlesen und in den Stack speichern
    System.out.println("Gib mindestens " + n + " Zeichen ein:");
    for (int i = 0; i < n; i++) {
      ch = Keyboard.readChar();
      s.push(ch);
    }

    System.out.println
        ("Umgekehrte Reihenfolge der ersten " + n + " Zeichen:");
    while (!s.isempty()) {             // solange Stack nicht leer:
        System.out.print(s.top());     // drucke und streiche
        s.pop();                       // oberstes Element
    }
    System.out.println();
  }
}
```

Wirkung von Umkehrung

```
% java Umkehrung
Groesse des Stacks: 5
Gib mindestens 5 Zeichen ein:
abcde
Umgekehrte Reihenfolge der ersten 5 Zeichen:
edcba
```

Umkehrung einer Zeichenkette: Objekt erzeugen

Umkehrung.java

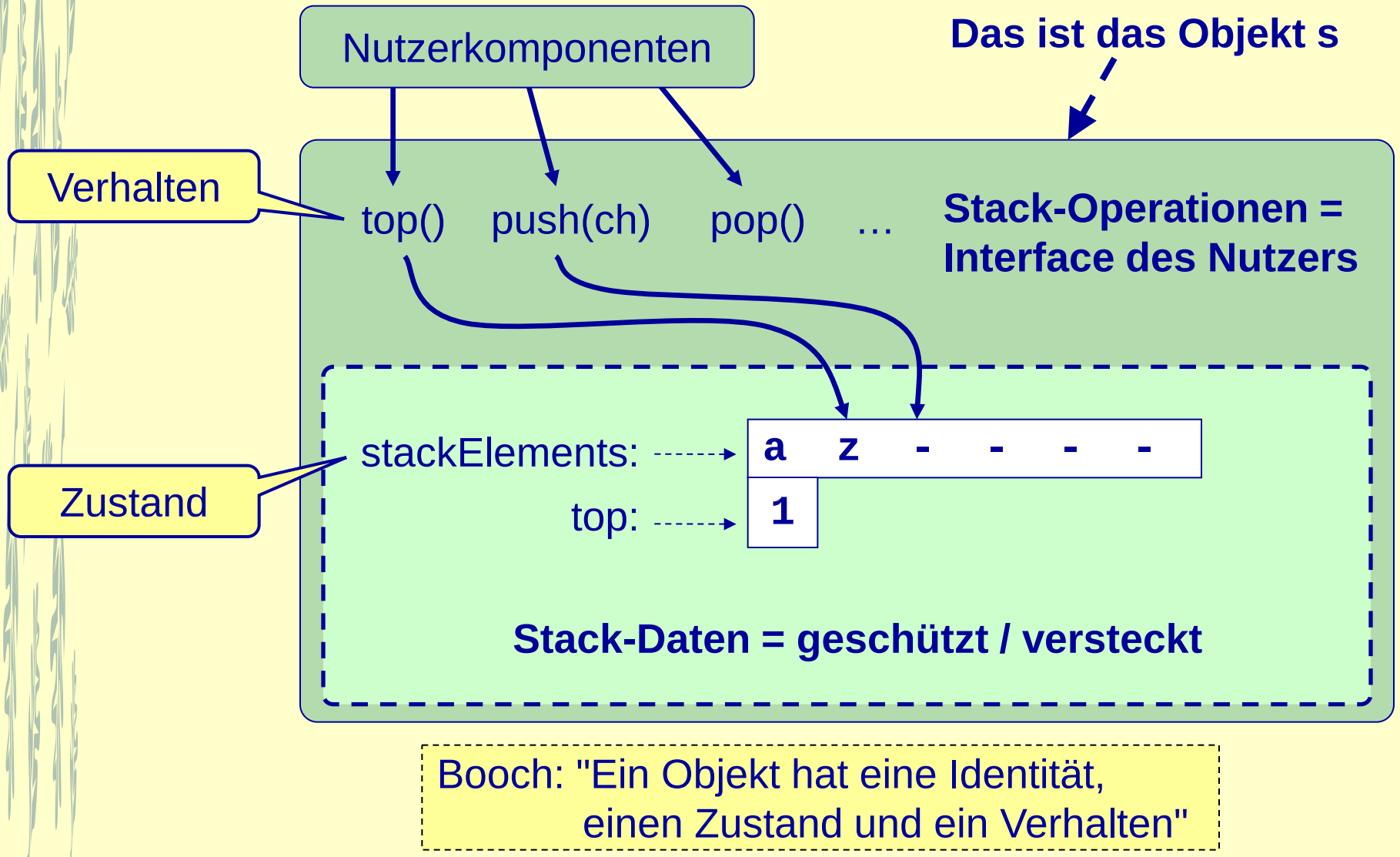
```
public static void main (String[] argv) {

    int n;
    char ch;
    Stack s;

    System.out.print("Groesse des Stacks: ");
    n = Keyboard.readInt();
    s = new Stack(n);
     ...
}
```

Umkehrung einer Zeichenkette: Speichern und Ausgeben von Elementen

Umkehrung.java

```
   // n Elemente einlesen und in dem Stack speichern
System.out.println(„Gib ... ein:“);
for (int i = 0; i < n; i++) {
   ch = Keyboard.readChar();
   s.push(ch);
}

System.out.println(„Umgekehrte Reihenfolge ... :“);
while (!s.isempty()) {
      // solange Stack nicht leer:
      // drucke und streiche oberstes Element
   System.out.println(s.top());
   s.pop();
}
```

Objekte = Einheiten aus versteckten Daten und Interface-Operationen



Booch: "Ein Objekt hat eine Identität,
einen Zustand und ein Verhalten"

Veränderungen im Objektzustand durch Methodenaufrufe (1)

Deklaration

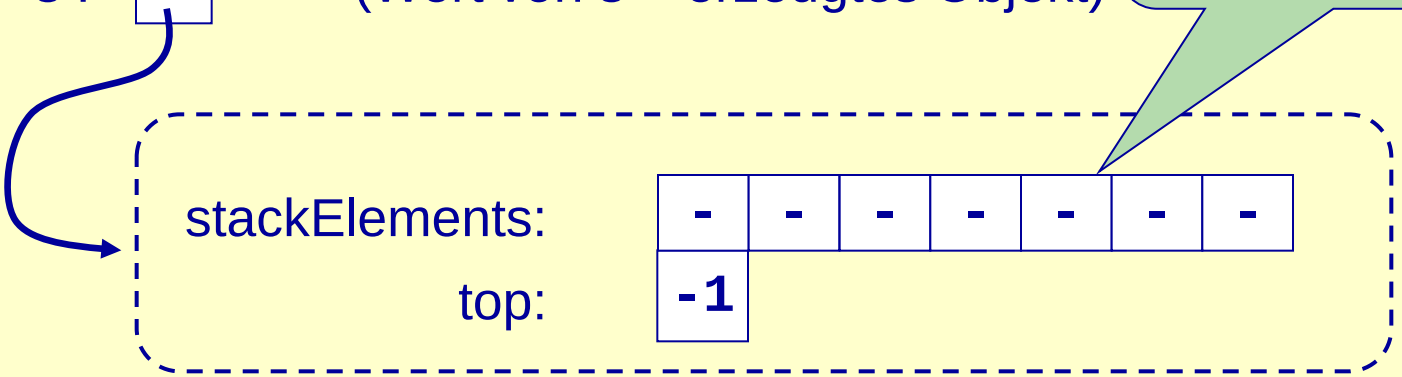
```
Stack s;
```

s :  (Speicherplatz reserviert für Zeiger, Adresse ohne Wert)

Erzeugung

```
s = new Stack(7);
```

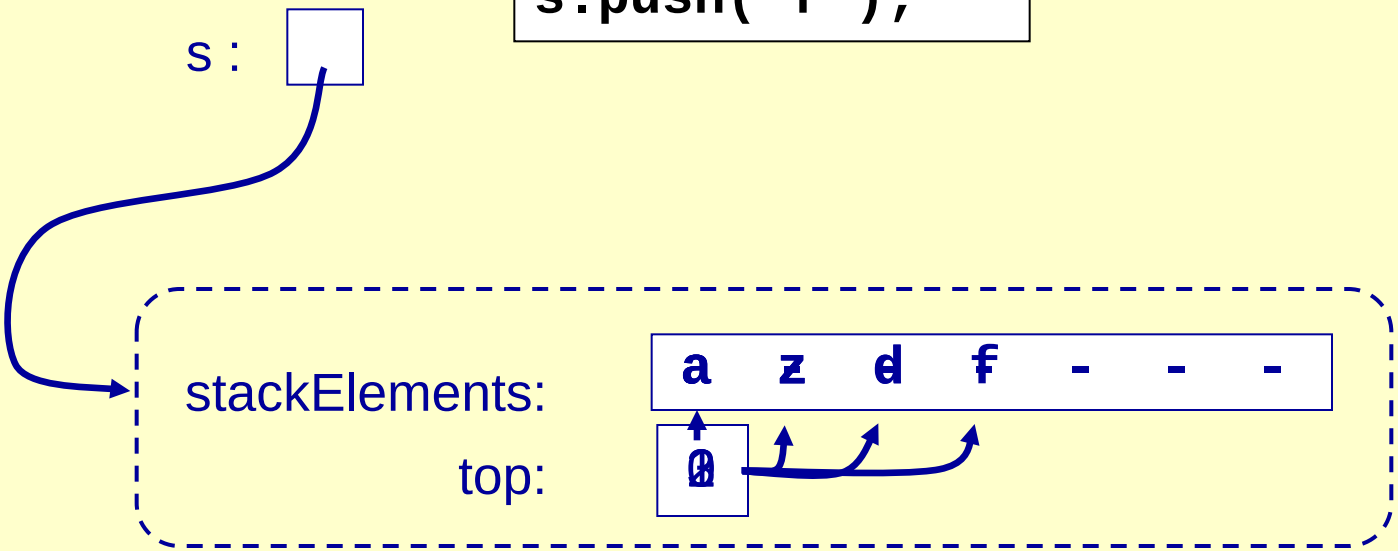
s :  (Wert von s = erzeugtes Objekt)



Veränderung im Objektzustand durch Methodenaufrufe (2)

Abspeichern von Elementen

```
s.push('a');
s.push('z');
s.push('d');
s.push('f');
```



Mehrere Instanzen der Klasse „Stack“

Umkehrung2.java

Aufgabe:

- Eingabefolge in 2 Teile (gerade/ungerade Position) aufspalten
- Jeden Teil in umgekehrter Reihenfolge ausgeben

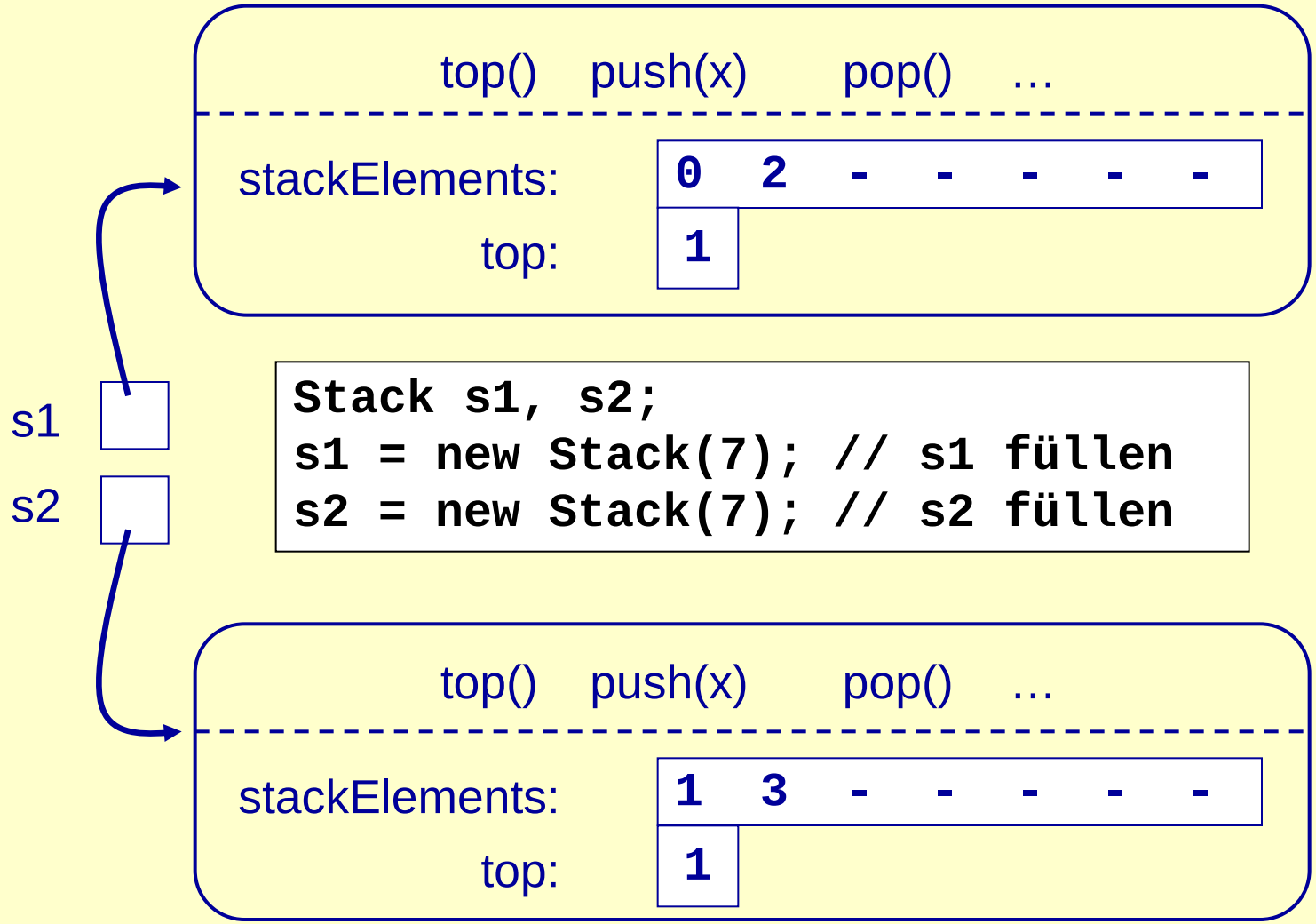
Eingabe: 0123456789

Ausgabe: 97531

86420

```
Stack s1, s2;
   ...
s1 = new Stack(n);
s2 = new Stack(n);
```

Objekte (Instanzen): jeweils mit eigenen Instanzvariablen + Instanzmethoden



(Instanz-) Methoden an Instanz gebunden

Umkehrung2.java

```
Stack s1, s2;
if (i % 2 == 0)
   s1.push(ch);
else
   s2.push(ch);

      ...

while (!s2.isempty()) {
   System.out.println(s2.top());
   s2.pop();
}
```

Methode push an s1 bzw. s2 gebunden

isempty() gehört hier zu s2

Zusätzlicher Parameter der Operationen:
Instanz (Objekt) s1 / s2 ...
(impliziter Parameter)
Pascal: push(s1, ch)

Imperative Lösung

Noch einmal:

Klassifikation von Programmiersprachen

SW-Entwicklung ohne OO?

OO erst seit ca. 1985 (Simula: 1970)

→ es muss auch imperativ gehen

```
class Stack {
   private char [] stackElements;

   private int top;

   public boolean isempty() {
      return top == -1;
   }

   public void push(char x) {
      top++;
      stackElements[top] = x;
   }
      ...
}
```

bleiben: **Variablen (Daten)**
Bearbeitungsoperationen (Prozeduren)

→ wie in Pascal und C

Variante in C

```
char [] stackElements;

int top;

boolean isempty() {
     return top == -1;
}

void push(char x) {
     top++;
     stackElements[top] = x;
}
. . .
```

Variante in Pascal

```
VAR
   stackElements:
      ARRAY [0 .. 10] OF CHAR;
   top: INTEGER;

PROCEDURE isempty(): BOOLEAN;
   BEGIN ... END;

PROCEDURE push (x: char x)
   BEGIN ... END;
      ...
```

Konsequenzen: ohne OO (Pascal, C)

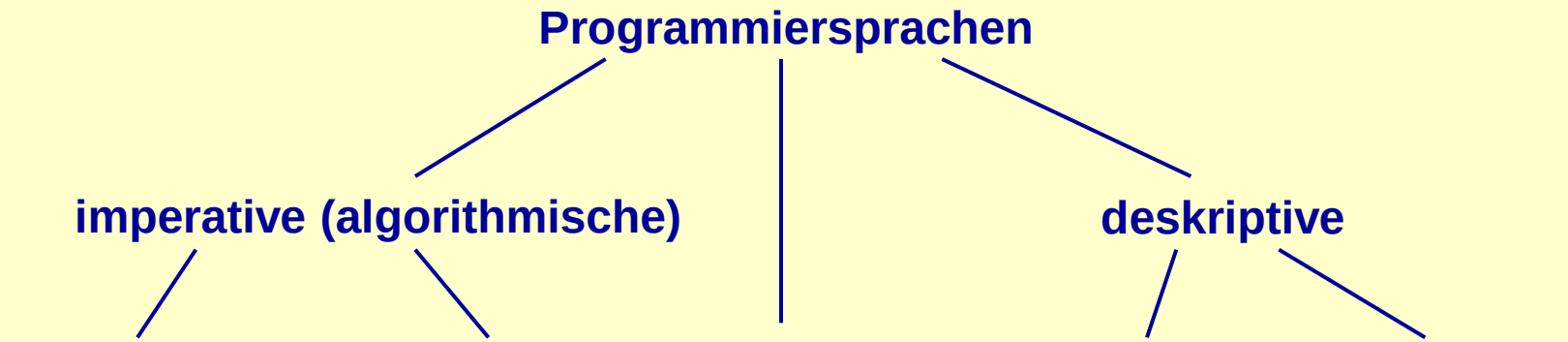
```
char [] stackElements;
int top;

boolean isempty() {
      return top == -1;
}

void push(char x) {
      top++;
      stackElements[top] = x;
}
```

C

Probleme?

- Keine Sichtbarkeitsangaben (Daten ungeschützt)
- Semantische Einheit (Daten + Operationen) nicht mehr hervorgehoben als syntaktische Einheit
- Nur **ein** Objekt gebildet (**ein** Stack)
  → mehrere Objekte: mehrere Variablen vereinbaren, Operationen mit zusätzlichem Parameter

Wiederholung: Klassifikation von Programmiersprachen

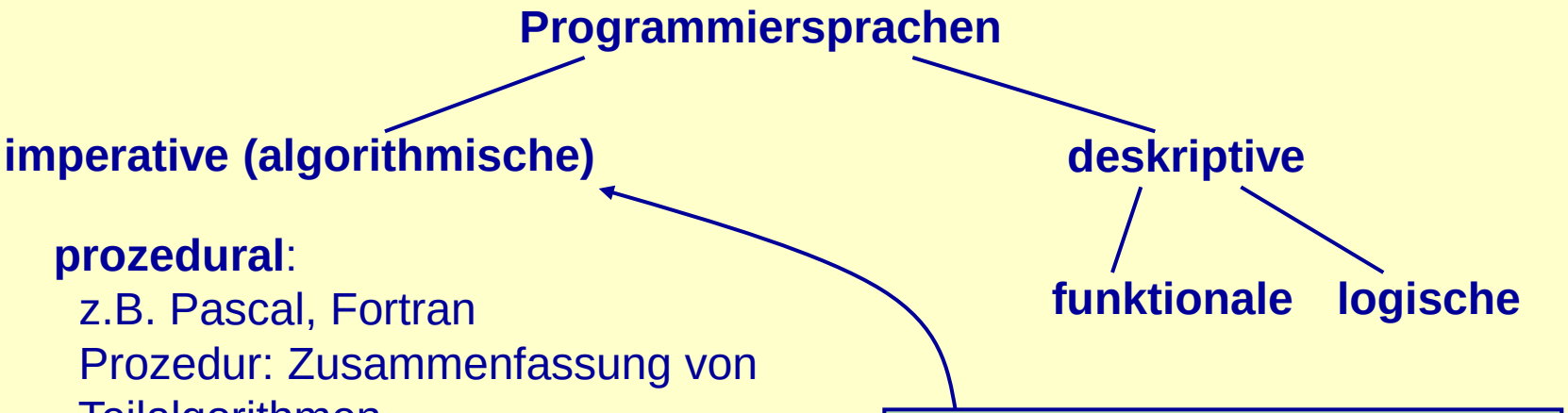


maschinennah	höhere	objektorientierte	funktionale	logische
Assembler-sprachen	Pascal, Ada, C, Fortran, Cobol, Basic	Simula 67, C++, Delphi, Java	Haskell, Lisp, Scheme	Prolog, Prolog II

Imperative und objektorientierte Sprachen mit gleichen Grundkonzepten:
- Variable
- Anweisung
- Algorithmus

nur zu unterschiedlichen Einheiten zusammengefasst

Alternative Klassifikation



prozedural:
z.B. Pascal, Fortran
Prozedur: Zusammenfassung von Teilalgorithmen

modular:
z.B. Modula-2, Ada
Modul: Zusammenfassung von verwandten Daten und Prozeduren

objektorientiert:
z.B. Java, C++
Klasse: Zusammenfassung verwandter Daten und Prozeduren zur Typbildung (ADT)

Zerlegungsarten
von Programmen innerhalb des imperativen Paradigmas
(keine neuen Paradigmen – es geht in allen Fällen um Algorithmenformulierung)

Objektorientierung ist Variante imperativer Programmierung

Imperativer Ansatz: Algorithmus ohne/mit ADT