

# 4. Komponentenarten

Java-Beispiele:

Java.lang.Math (Java-API)

Stack.java (aus: III.1)

Time.java (aus: III.2)

Keyboard.java (aus: II.3)

TimeC.java (aus: III.3)

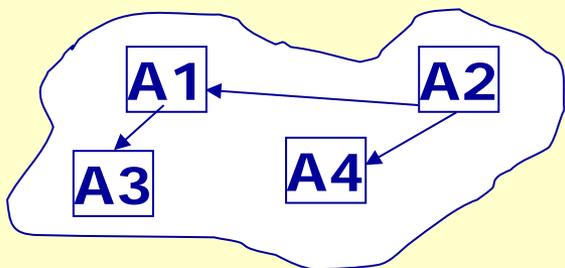
# Schwerpunkte

## Ziele:

- Systematisierung: Aufbau von Klassen
- Beherrschung des Klassenkonzepts vertiefen
- Unterschiedliche Einsatzfelder von Klassen aufzeigen
- Erkennen:  
Software-Systeme in der Praxis sind aus Klassen unterschiedlicher Art aufgebaut, d.h. es gibt keine "reinen" objekt-orientierten Systeme

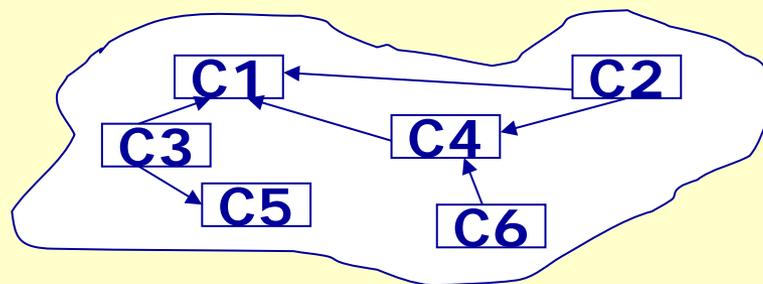
# Imperative und objekt-orientierte Programm-Strukturen: im Ideal

## Imperative Struktur:



Komponenten  $A_i$  sind imperativ:  
→ realisieren Algorithmen,  
z.B. ZeitPlan.java, Hanoi.java

## Objekt-orientierte Struktur:



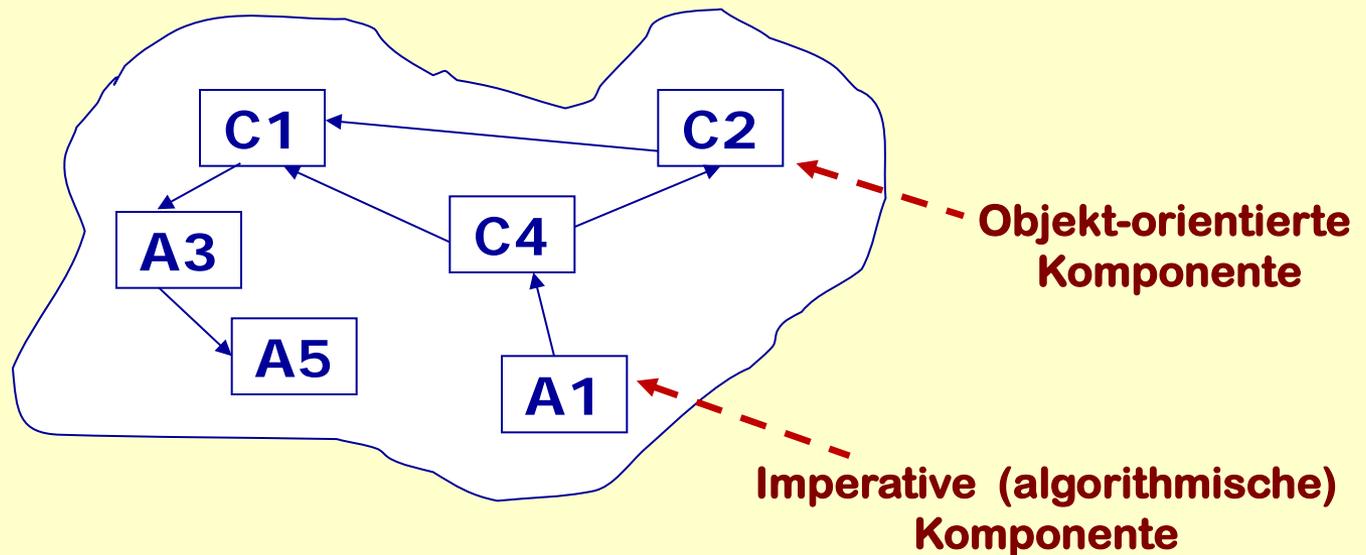
Komponenten  $C_i$  sind objekt-orientiert:  
→ realisieren abstrakte Datentypen,  
z.B. Time.java, Stack.java

# Imperative und objekt-orientierte Programm-Strukturen: Wirklichkeit

## Realität:

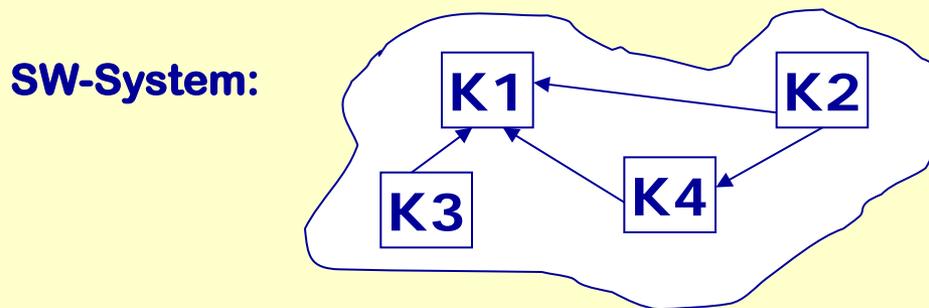
- "Reine imperative" bzw. "reine objekt-orientierte" System-Strukturen sind selten bzw. nicht sinnvoll bzw. unmöglich
- Normalerweise sind die Systeme eine Mischung unterschiedlichster Komponentenarten

## Reale Strukturen:



# Problem: sinnvolle Komponentenarten

**Klasse** = Komponente eines objektorientierten SW-Systems  
(in Java, C++, Smalltalk, Simula 67, Eiffel ...)



Welche Komponentenarten möglich / sinnvoll ?

Das Java-Klassenkonzept (auch: C++ u.a.)  
spiegelt nur ungenügend relevante Arten  
von Softwarekomponenten wider.

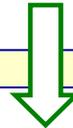
# Klassen und Komponentenarten

**Klasse: eine Syntax**

Sammlung von Variablen und Methoden

static/non-static

public/private



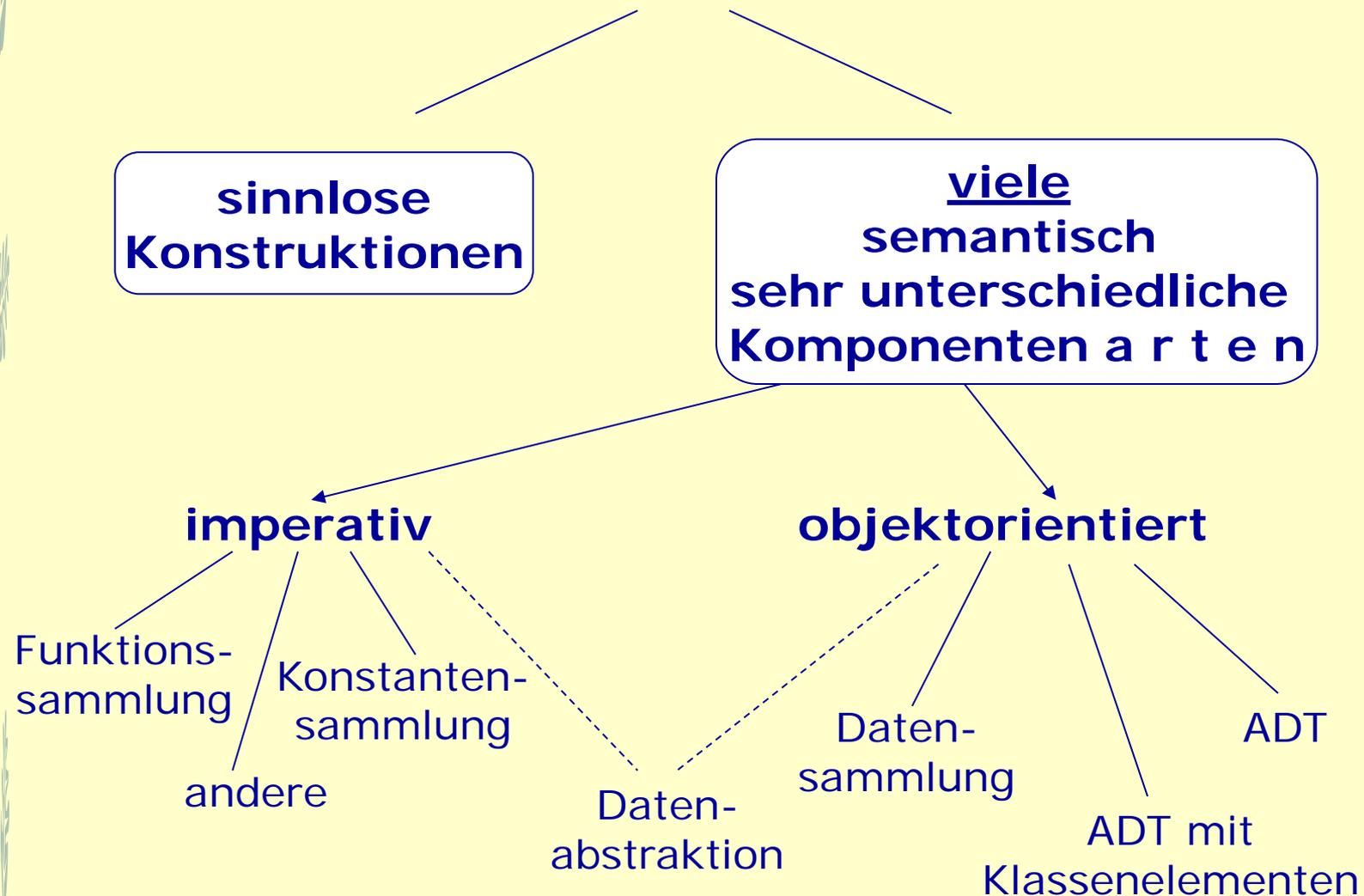
**Viele Komponentenarten**

mit höchst unterschiedlichen Eigenschaften realisierbar  
(z.B. ADT, imperative Komponenten ...).

**Ziel dieses Kapitels:**

- Ordnung hineinbringen (Klassifikation)
- Orientierung und methodische Hilfe:
  - Lesen und Verstehen von Programmen
  - Entwickeln von Programmen

# Klassen und realisierbare Komponentenarten: Überblick



# Klassen und realisierbare Komponentenarten: Überblick

Es folgt die Behandlung  
verschiedener  
Komponentenarten:

- Prinzip
- Syntaktische Realisierung  
in Java
- Nutzung
- Beispiele

Funk-  
sammlung

andere  
sammlung

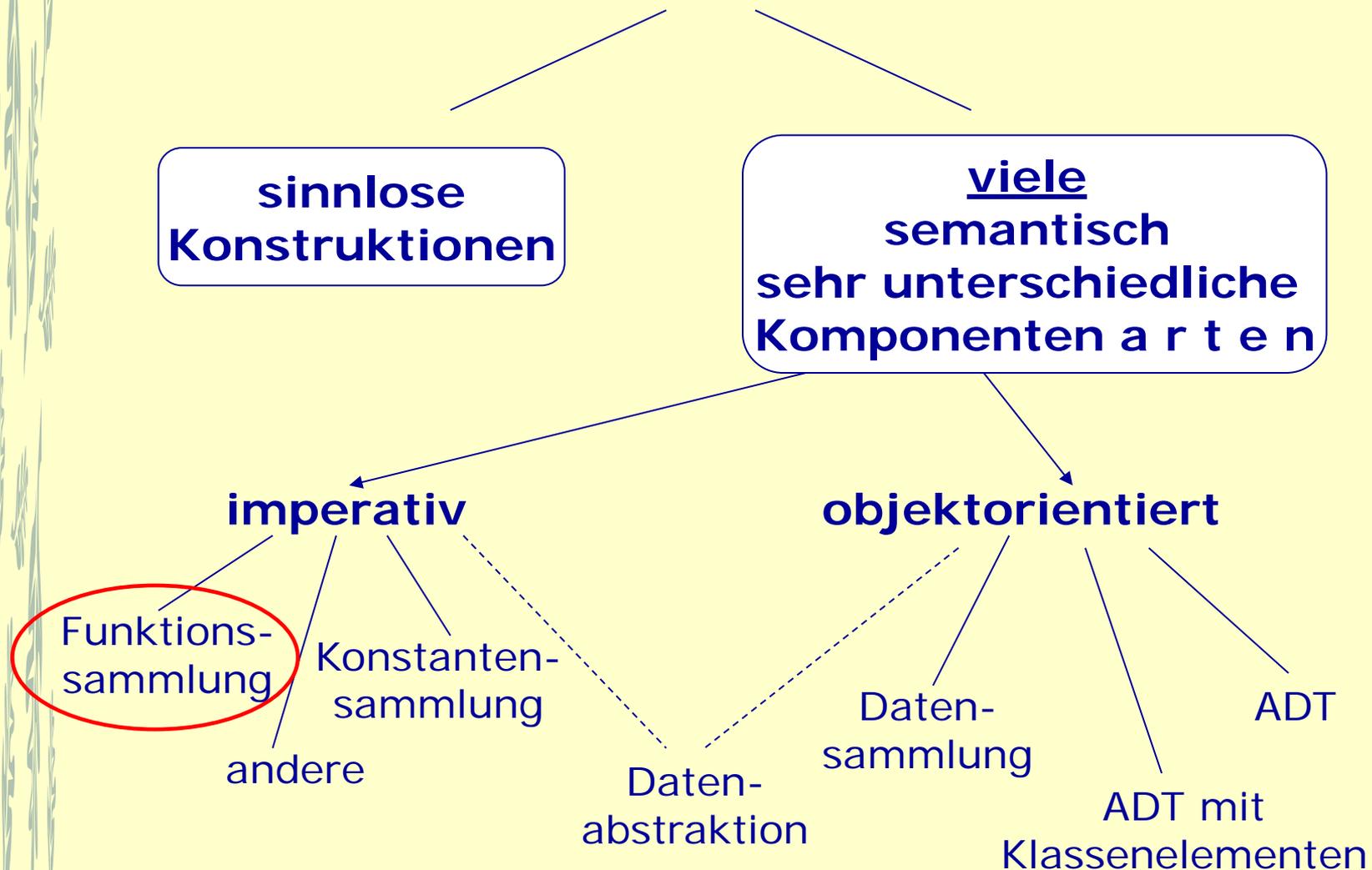
Daten-  
abstraktion

Daten-  
sammlung

ADT mit  
Klassenelementen

ADT

# Klassen und Komponentenarten: Funktionssammlung



# Komponentenart: Funktionssammlung

Menge von verwandten Funktionen  
ohne gemeinsame Daten  
(evtl. einige Konstanten)

- Daten können nicht dem Austausch von Information zwischen den Funktionen dienen (vgl. Stack)
- also: Menge unabhängiger Algorithmen

## Java-Klasse:

nur 'public' 'static'-Methoden  
(nur *Klassenmethoden*)  
(+ einige 'final' 'static' - Variablen)

→ **Nutzung: Instanzenbildung nicht sinnvoll**

- **Beispiel:** mathematische Funktionen im Java-API

java.lang

## Class Math

# API-Klasse java.lang.Math

[java.lang.Object](#)

└ [java.lang.Math](#)

public final class Math  
extends [Object](#)

The class Math contains methods

Unlike some of the numeric methods, Math methods do not return exact results. This relaxation permits

By default many of the Math methods use specific native libraries or micro-processor implementations still must conform

The quality of implementation of Math methods is measured in terms of floating-point values bracketed by the exact result. The error bound cited is for the worst-case error. For exact results, such a method is

many floating-point methods to be correctly rounded. Instead, for the Math class, a larger error bound of 1 or 2 ulps is allowed for certain methods. Informally, with a 1 ulp error bound, when the exact result is a representable number, the exact result should be returned as the computed result; otherwise, either of the two floating-point values which bracket the exact result may be returned. For exact results large in magnitude, one of the endpoints of the bracket may be infinite. Besides accuracy at individual arguments, maintaining proper relations between the method at different arguments is also important. Therefore, most methods with more than 0.5 ulp errors are required to be *semi-monotonic*: whenever the mathematical function is non-decreasing, so is the floating-point approximation, likewise, whenever the mathematical function is non-increasing, so is the floating-point approximation. Not all approximations that have 1 ulp accuracy will automatically meet the monotonicity requirements.

Since:

JDK1.0

### Field Summary

static double [E](#)

The

static double [PI](#)

The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

public final class java.lang.Math

The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Zwei Konstanten:

static double E

static double PI

## Method Summary

static double	<a href="#">abs</a> (double a) Returns the absolute value of a double value.
static float	<a href="#">abs</a> (float a) Returns the absolute value of a float value.
static int	<a href="#">abs</a> (int a) Returns the absolute value of an int value.
static double	<a href="#">acos</a> (double a) Returns the arccosine of a value.
static double	<a href="#">asin</a> (double a) Returns the arcsine of a value.
static double	<a href="#">atan</a> (double a) Returns the arctangent of a value.
static double	<a href="#">ceil</a> (double a) Returns the smallest integer greater than or equal to a value.
static double	<a href="#">cos</a> (double a) Returns the cosine of a value.
static double	<a href="#">exp</a> (double a) Returns Euler's constant e raised to the power of a value.
static double	<a href="#">floor</a> (double a) Returns the largest integer less than or equal to a value.
static double	<a href="#">log</a> (double a) Returns the natural logarithm of a value.
static double	<a href="#">max</a> (double a, double b) Returns the greater of two double values.
static float	<a href="#">max</a> (float a, float b) Returns the greater of two float values.
static int	<a href="#">max</a> (int a, int b) Returns the greater of two int values.
static double	<a href="#">min</a> (double a, double b) Returns the smaller of two double values.
static float	<a href="#">min</a> (float a, float b) Returns the smaller of two float values.
static int	<a href="#">min</a> (int a, int b) Returns the smaller of two int values.
static double	<a href="#">pow</a> (double a, double b) Returns the value of the first argument raised to the power of the second argument.

## Java-API:

```
public final class Java.lang.Math
```

## Auswahl von Methoden:

```
static double abs(double a)  
static float abs(float a)  
static int abs(int a)  
static double asin(double a)  
static double cos(double a)  
static double log(double a) // Basis e  
static double pow(double a, double b)
```

```
static double pow(double a, double b)
```

```
static double
```

```
static long
```

```
static int
```

```
static double
```

```
static double
```

```
static double
```

## public static final double E

The double that is closer than any other to e, the base of the natural logarithm

## public static final double PI

The double that is closer than any other to pi,

...

### Field Detail

E

```
public static final double E
```

The double value

See Also:

[Constant Field Va](#)

PI

```
public static final d
```

The double value

See Also:

[Constant Field Va](#)

### Method Detail

sin

```
public static double sin(double a)
```

Returns the trigono

- If the argu
- If the argu

The computed res

Parameters:

a - an angle, in ra

Returns:

the sine of the argument.

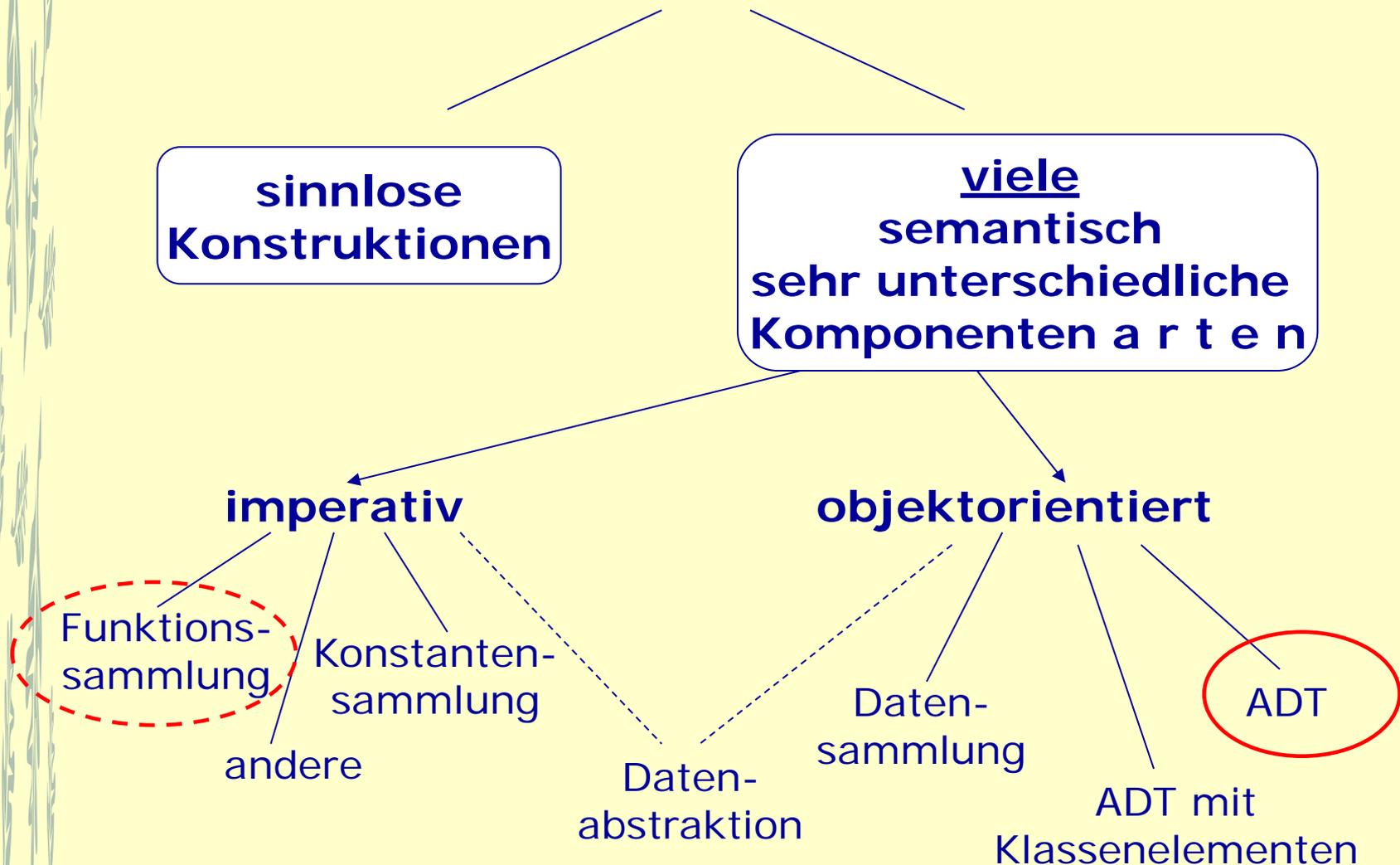
Nutzung: ohne Instanzenbildung

Math.sin(1.2); Math.PI;

cos

```
public static double cos(double a)
```

# Klassen und Komponentenarten: abstrakter Datentyp



# Komponentenart: ADT

## Ideal einer OO-Komponente:

- Menge von verwandten Funktionen mit gemeinsamen versteckten Daten
- Neuer nutzerdefinierter Typ definiert: beliebig viele Instanzenbildungen

## Java-Klasse:

- 'private' non-static – Variablen (Instanz)
- 'public' non-static – Methoden (Instanz)

non-static  
= 'static' fehlt

→ **Nutzung: beliebig viele Instanzen (Objekte) des Typs können erzeugt werden**

- **Beispiele:** Stack, Time, HashTable
- **Abweichungen:** (vgl. Abschnitt III.2)

gefährlich!

- public-Variablen (schneller Zugriff)
- private-Methoden (Hilfsfunktionen)

# Beispiel für ADT: Klasse Time

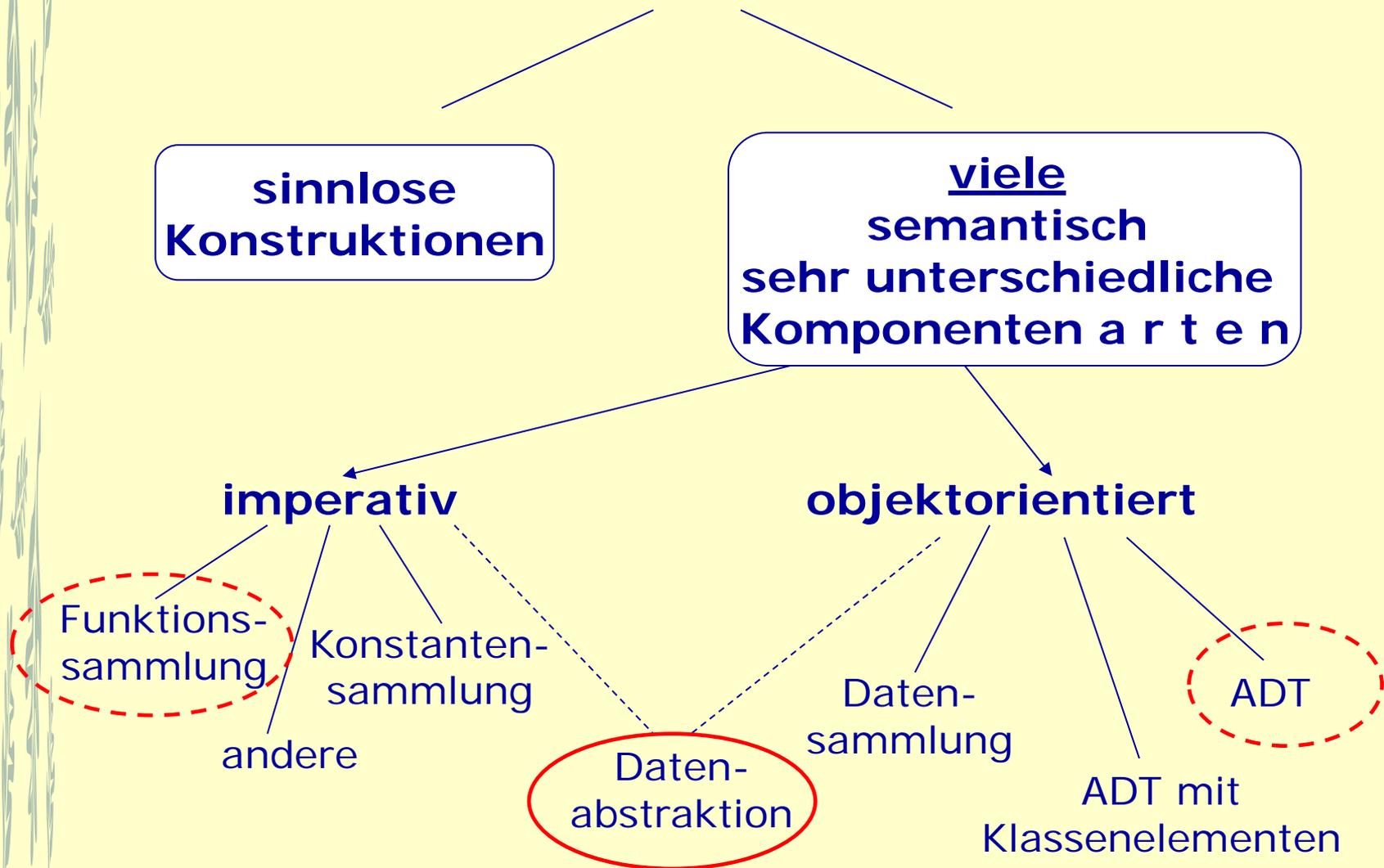
```
class Time {  
    // alles non-static  
    private int hour, minute;  
    public Time() ...  
    public addMinutes ...  
    public printMinutes ...  
    public timeInMinutes ...  
    public printTimeInMinutes ...  
}
```

Time.java

```
class Schedule {  
    public main ...  
        Time t1 = new Time(8,30);  
        Time t2 = new Time();  
        Time t3, t4;  
    ...  
}
```

Nutzung:  
Instanzbildung

# Klassen und Komponentenarten: Datenabstraktion



# Komponentenart: Datenabstraktion

- Wie ADT:  
Menge von verwandten Funktionen mit  
gemeinsamen versteckten Daten
- Ohne die Möglichkeit der Instanzenbildung  
(nur ein Exemplar benötigt  
→ nur ein Exemplar möglich)  
→ Spezialfall eines ADT

**Java-Klasse:** alles Klassenelemente  
'private' 'static' – Variablen  
'public' 'static' - Methoden

→ **Nutzung: keine Instanzenbildung**

- **Beispiel: Keyboard**  
(auch: Stack, Time können so realisiert werden)

# Keyboard/Tastatur: Anwendungsprogramm benötigt nur ein Exemplar

```
class Keyboard {  
    private static boolean iseof = false;  
    private static char c;  
    ...  
    private static BufferedReader input = ...  
    public static int readInt () {  
        if (iseof) return 0;  
        System.out.flush();  
        ... }  
    public static char readChar () ...  
    public static double readDouble (  
    public static String readString (  
    public static boolean eof () ...  
}
```

Keyboard.java

```
class Temperature {  
    ... main ...  
    tempFahr = Keyboard.readDouble();  
}
```

Vorteil?

→ Vorteil: keine Instanzenbildung in der Anwendung nötig

# Stack als Datenabstraktion: nur eine Instanz?

Hier:  
ADT Stack

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

    public void 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--;
    }
}
```

Stack.java

Was ist zu  
ändern?

# Stack als Datenabstraktion: nur eine Instanz

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

    public static void init(int n) {
        stackElements = new char [n];
        top = -1;
    }
    public static boolean isempty() {
        return top == -1;
    }
    public static void push(char x) {
        top++;
        stackElements[top] = x;
    }
    public static char top() {
        if (isempty()) {
            System.out.println("Stack leer");
            return ' ';
        }
        else return stackElements [top];
    }
    public static void pop() {
        if (isempty()) System.out.println("Stack leer");
        else top--;
    }
}
```

# Stack als Datenabstraktion (1): nur e i n e Instanz

```
class Stack {
```

Variablen: existieren nur einmal

```
private static char [] stackElements;  
private static int top;
```

nicht Konstruktor Stack()

```
public static void init(int n) {  
    stackElements = new char [n];  
    top = -1;  
}
```

Methoden: nur einmal

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

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

# Stack als Datenabstraktion (2): nur e i n e Instanz

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

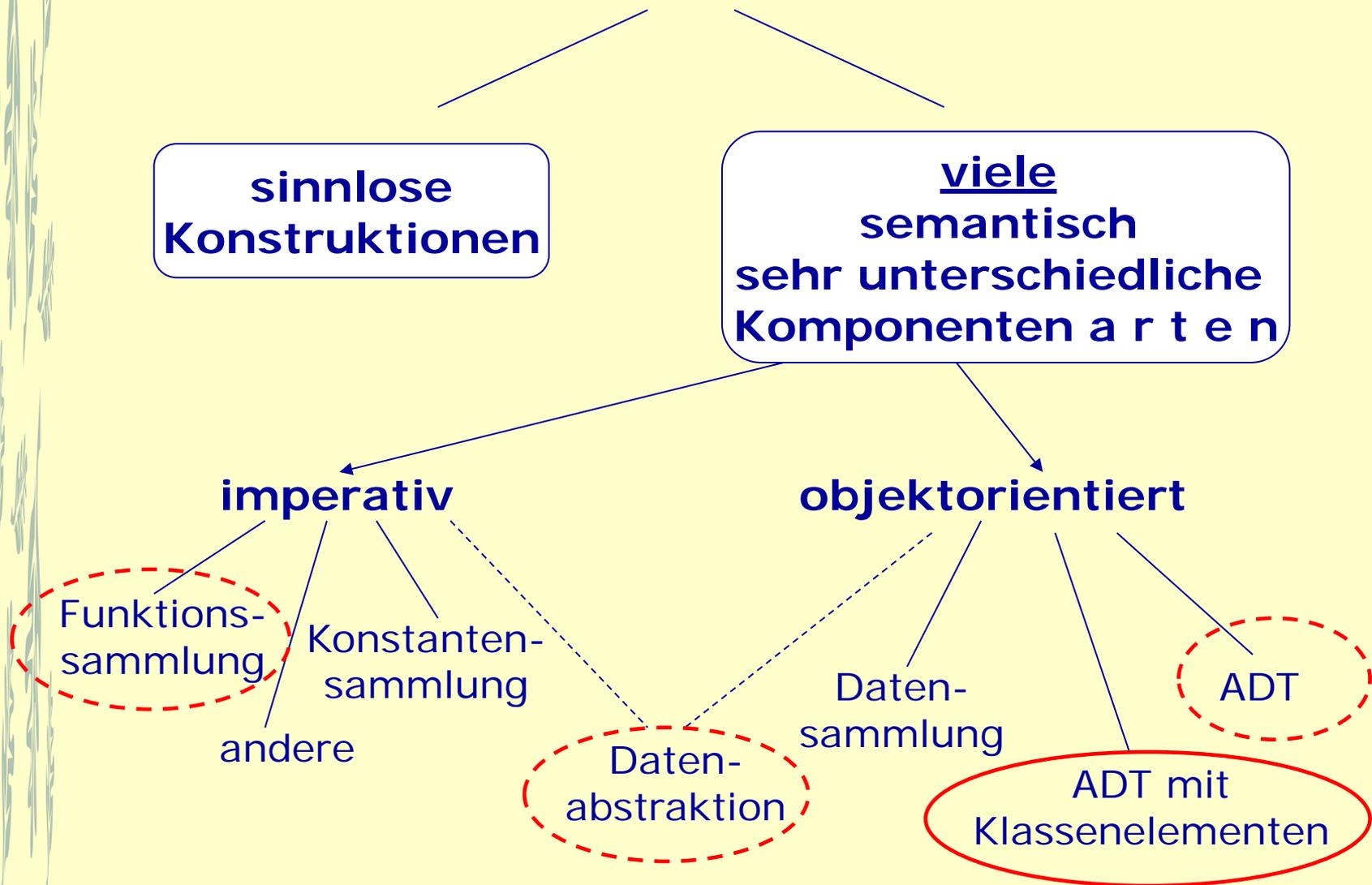
public static void pop() {
    if (isempty())
        System.out.println("Stack leer");
    else
        top--;
}
}
```

# Anwendung: Stack als Datenabstraktion

```
public static void main (String argv[]) {  
  
    int n;  
    char ch;  
    // moeglich - aber Unsinn :  
    // Instanzenbildung von Stack:  
    // Stack s = new Stack ();  
    ...  
    Stack.init(n);  
    ...  
    for (int i = 0; i < n; i++) {  
        ch = Keyboard.readChar();  
        Stack.push(ch);  
    }  
    while (!Stack.isEmpty()) {  
        System.out.print(Stack.top());  
        Stack.pop();  
    }  
}
```

Wieso ?

# Klassen und Komponentenarten: ADT mit Klassenelementen



# Komponentenart: ADT mit Klassenelementen

Wie ADT: Menge von verwandten Funktionen mit gemeinsamen versteckten Daten, aber:

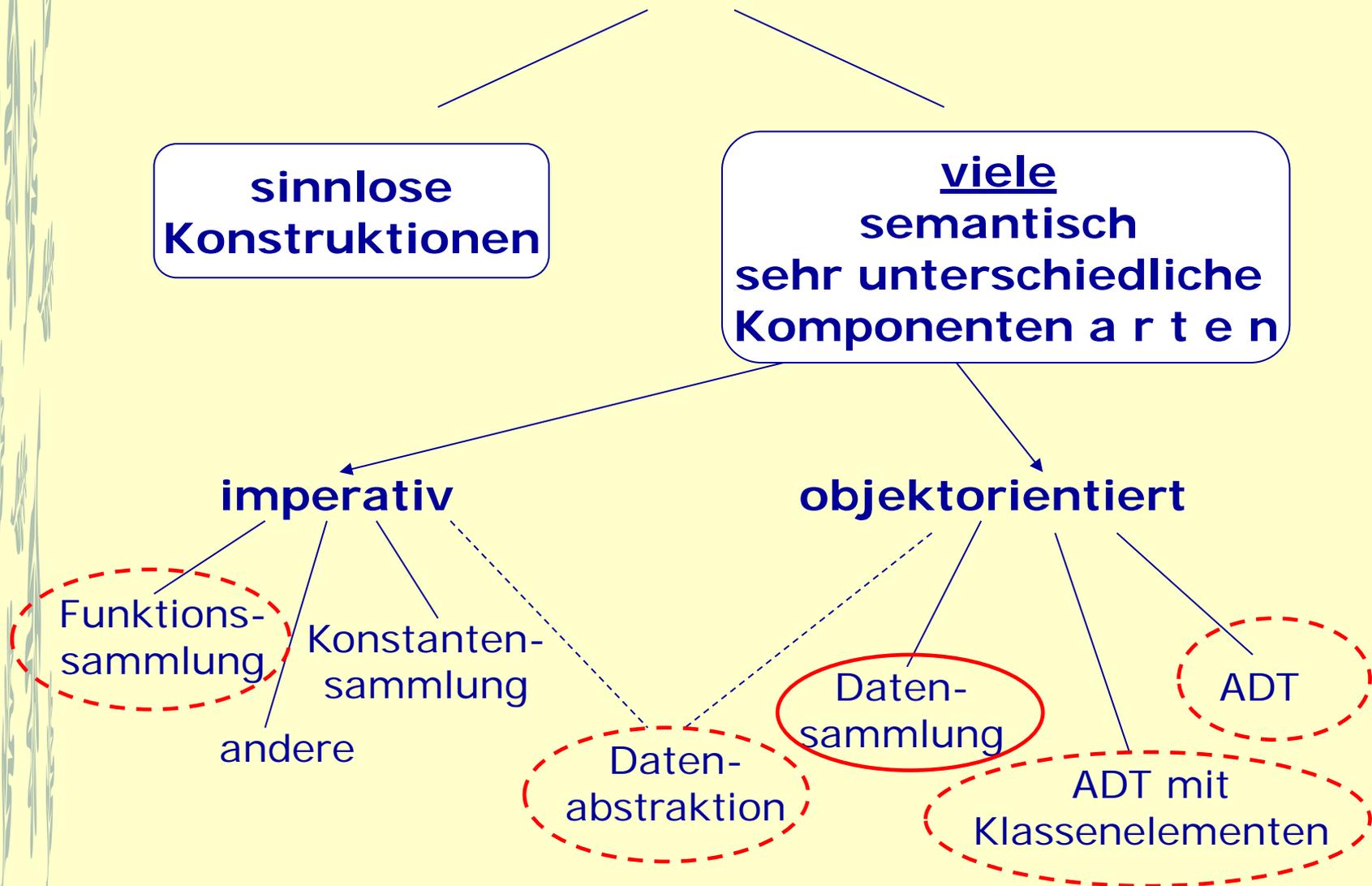
- einige Variablen / Methoden nur einfach (keine Instanzenbildung)
- beliebig viele Instanzen (Objekte / Variablen) des Typs

→ Variablen als gemeinsamer Speicher aller Instanzen des Typs

## Java-Klasse:

- 'private' non-static oder static – Variablen
- 'public' non-static oder static – Methoden
- **Beispiel:** Time mit Klassenvariablen 'noonHour' ... und Klassenmethode switchTimeFormat()  
→ TimeC (Kapitel III.3)

# Klassen und Komponentenarten: Datensammlung



# Komponentenart: Datensammlungen (Datenklassen)

Zusammenfassung von sichtbaren Daten  
zu neuem Typ

## Java-Klasse:

keine Methoden  
nur nicht-statische public Daten

→ **Nutzung: Instanzenbildung**

- **Beispiel:** Punkt3D  
→ wie Pascal-Record (bzw. C-Struct)

# Beispiel: Datensammlung

Punkte im dreidimensionalen Raum bestehen aus drei Werten:

x-, y-, z- Koordianten

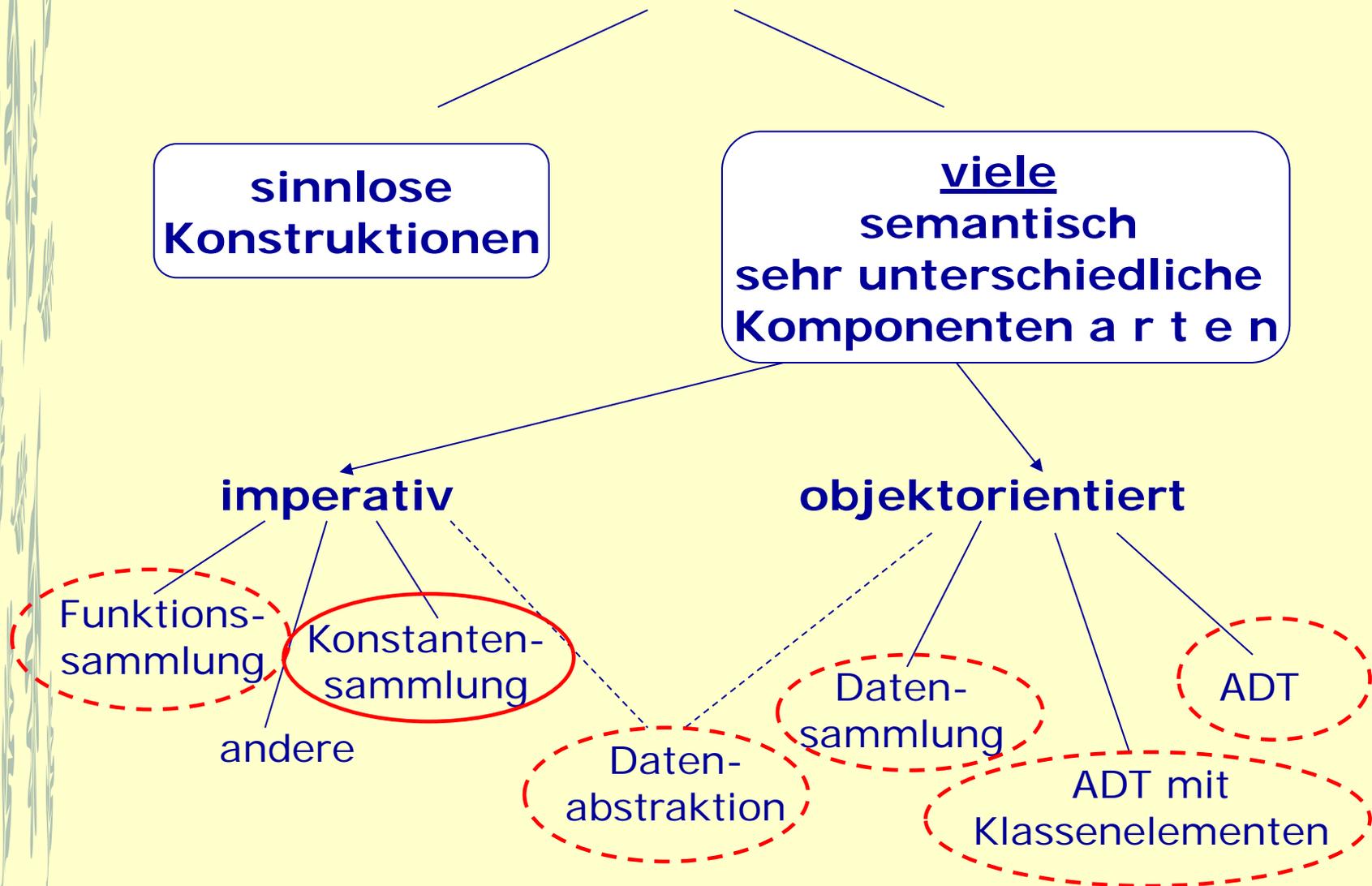
```
class Punkt3D {  
    double x, y, z;  
}
```

sichtbar im selben Paket

```
class Punkt3D {  
    public double x, y, z;  
}
```

sichtbar in allen Paketen

# Klassen und Komponentenarten: Konstantensammlung



# Komponentenart: Konstantensammlung

Zusammenfassung  
von sichtbaren Konstanten

## Java-Klasse:

- keine Methoden
- nur public final static Variablen

→ Nutzung: keine Instanzenbildung

- **Beispiel:** Drucksteuerzeichen als Konstanten

# Beispiele: Konstantensammlung

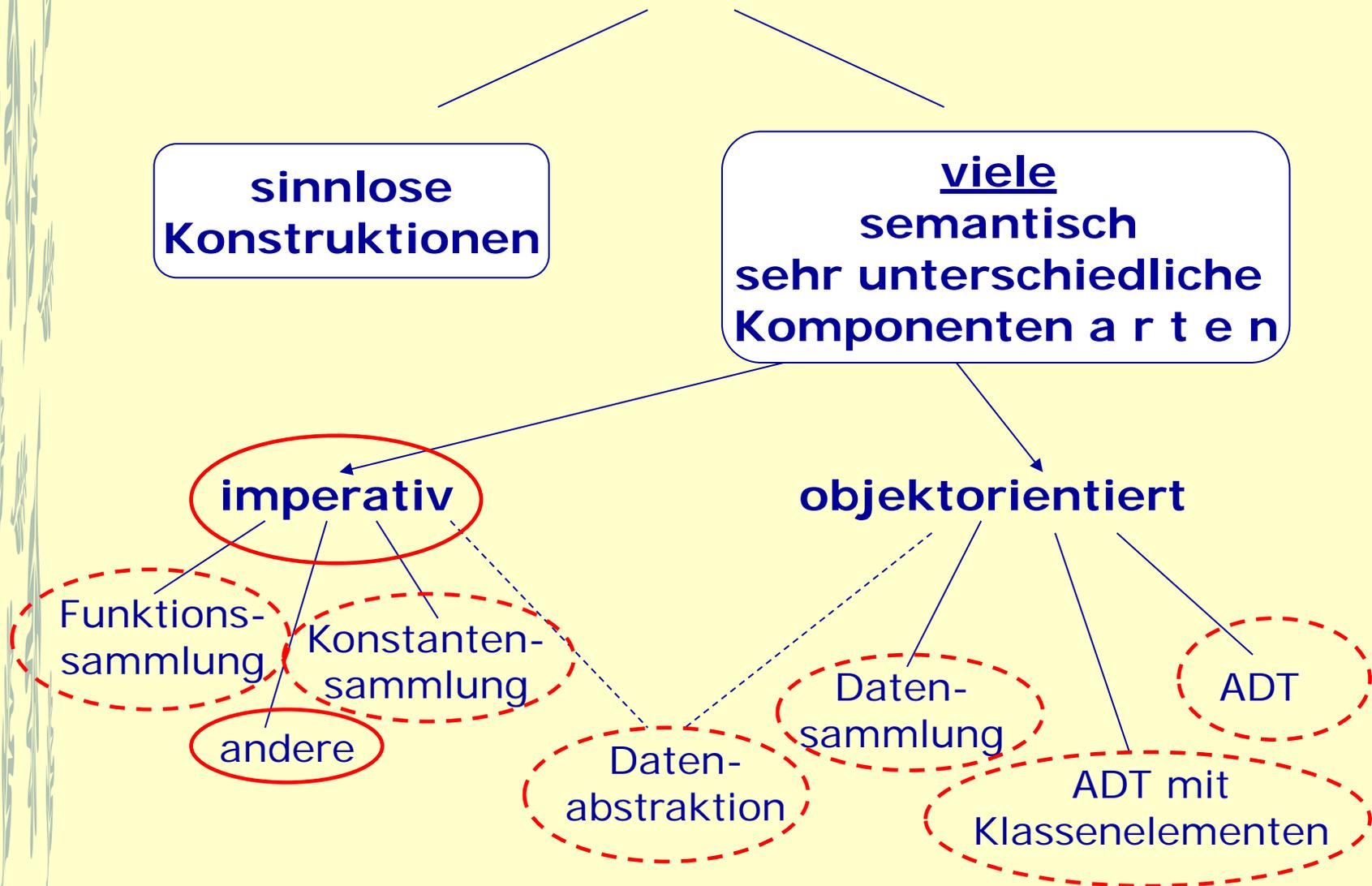
```
class IO_Constants {  
  
    public final static char LF = '\n',  
                           FF = '\f',  
                           CR = '\r';  
  
    public final static int DRUCK_BREITE = 80;  
  
}
```

C, C++: .h-Files:

```
#define LF '\n'  
#define FF '\f'  
#define CR '\r'  
#define DRUCK_BREITE 80
```

In C, C++: Namen nicht typgebunden → reine Textersetzung

# Klassen und Komponentenarten: imperative Komponenten



# Komponentenart: imperative Komponente

Komponente,  
für die Instanzenbildung **n i c h t** sinnvoll ist

## Java-Klasse:

Daten und Methoden nur 'static'

→ **Nutzung:**  
**keine Instanzenbildung von Klassen**

**Beispiele: Teil II der Vorlesung**  
– es kommt auf Algorithmen an

- Sortierverfahren
- Hanoi.java
- s.o.: Funktionssammlungen
- **Main-Klasse:** enthält main()
  - **niemals Instanzenbildung**
  - **Main-Klasse immer imperative Komponente**
  - **Beginn des Algorithmus**

# Beispiel: main-Klasse ohne Instanzenbildung

```
public class Echo {  
  
    public static void main(String args[]) {  
  
        for (int i=0; i < args.length; i++)  
            System.out.println(args[i] + " ");  
  
        System.out.print("\n");  
    }  
}
```

**Instanzenbildung sinnlos**

# Beispiel: Instanziierung einer main-Klasse

(Quelle: J. Bishop, Java lernen, Addison-Wesley, 2001)

```
class Hallo {  
  
    Hallo() {  
        System.out.println("Hallo!");  
    }  
  
    public static void main (String[] args) {  
        new Hallo();  
    }  
}
```

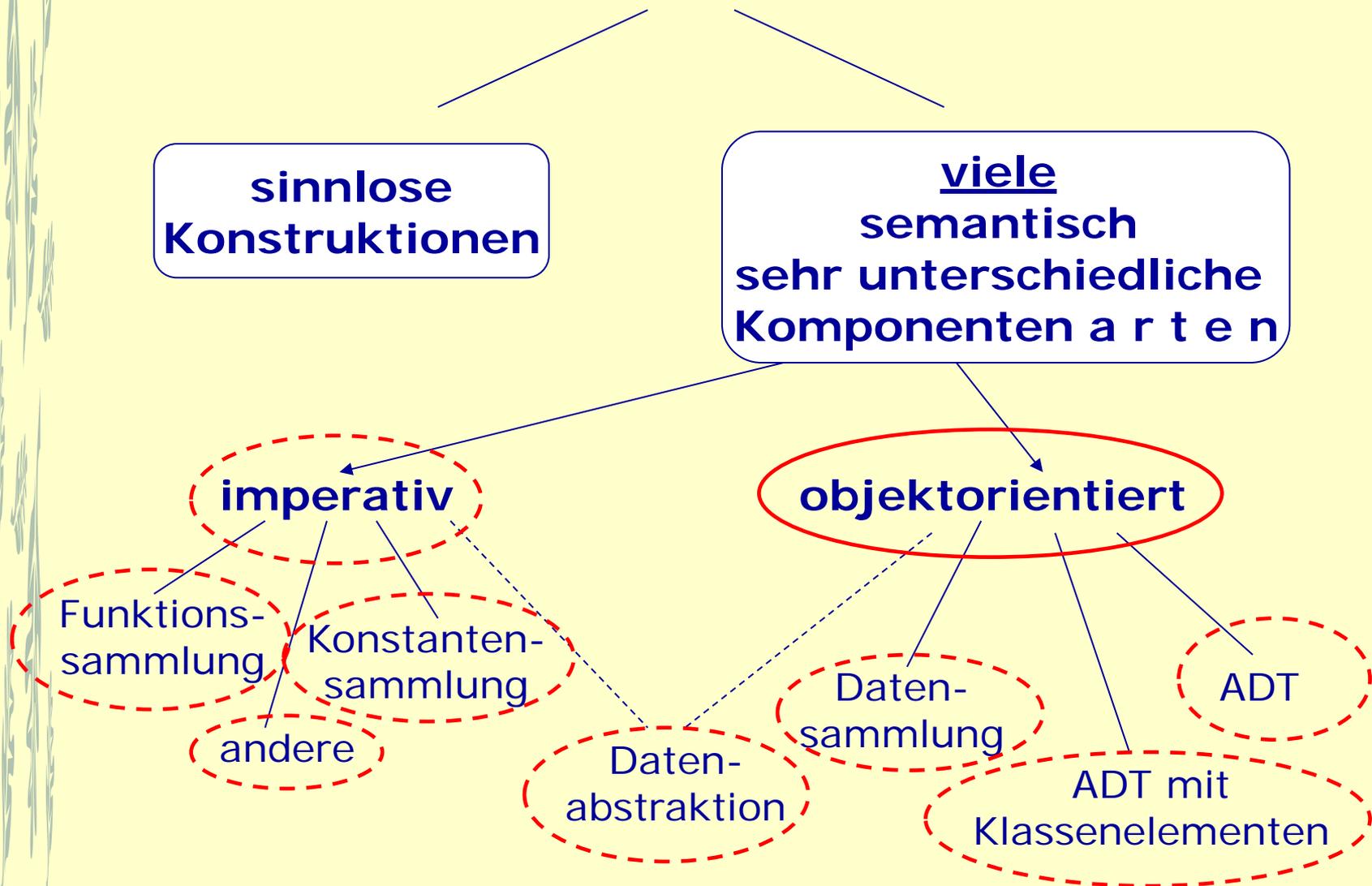
Bewertung dieses Stils ?

Wie sieht das erzeugte  
Objekt aus ?

Anm.: auch in: D. Bell, M. Parr: Java for Students,  
Prentice-Hall 2001 / auch in Deutsch

**Buch bis zur 3. Auflage nicht zu empfehlen:  
nur Applet-orientiert, rezepthaft,  
keine Rekursion u.v.a.**

# Klassen und Komponentenarten: objektorientierte Komponente



# Komponentenart: objektorientierte Komponente

Komponente, für die Instanzenbildung  
sinnvoll (notwendig) ist

## Java-Klasse:

enthält non-static Elemente  
(Instanz-Elemente: Variablen oder Methoden)

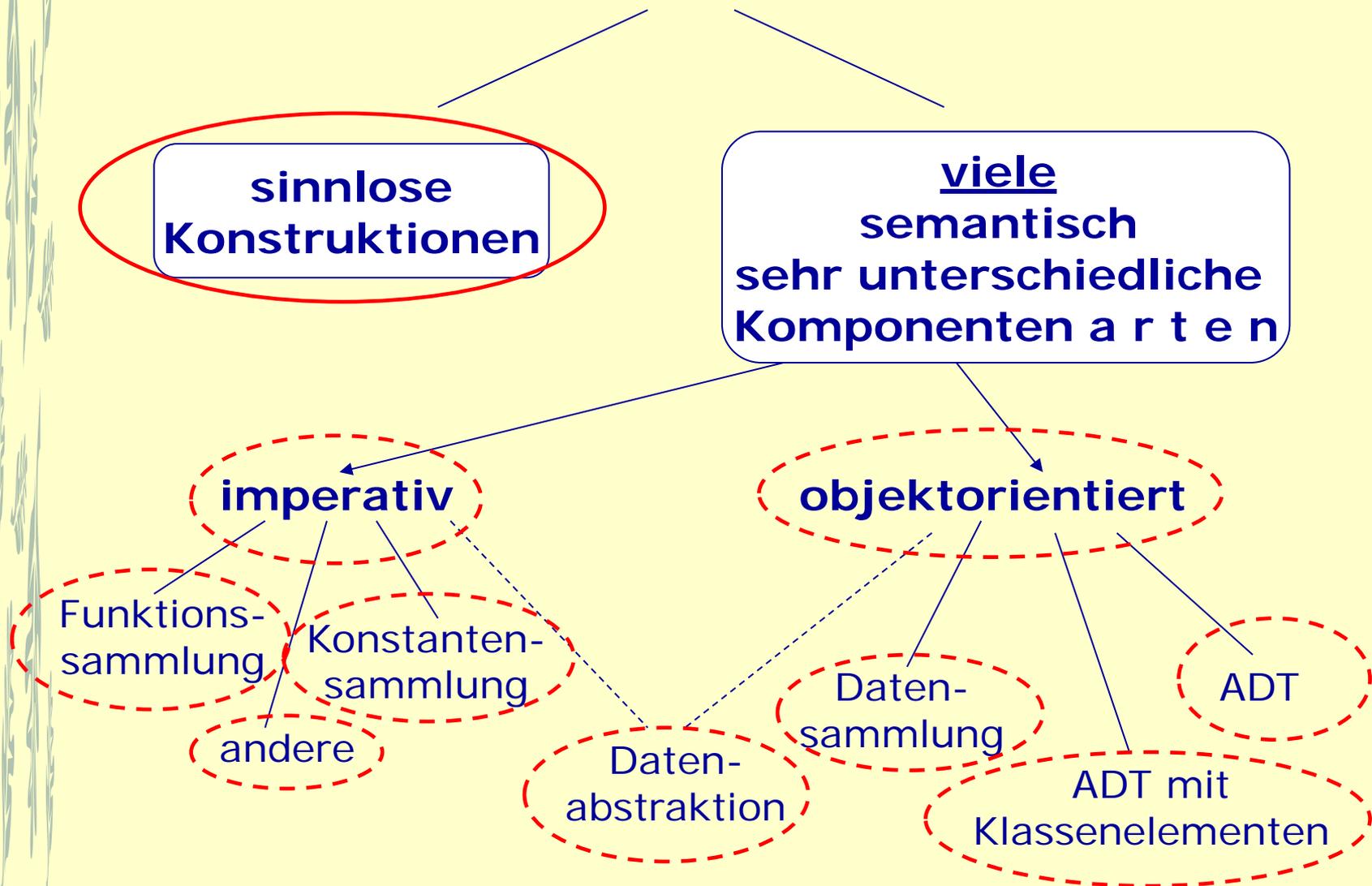
### → Nutzung:

Instanzenbildung nötig, um Variablen zu erzeugen  
(Speicherplatz) bzw. Methoden nutzbar zu machen

### • Beispiele:

- ADT
- ADT mit Klassenelementen
- Datensammlung

# Klassen und Komponentenarten: sinnlose Konstruktionen



# Komponentenarten: sinnlose Klassenkonstruktionen

- trotz syntaktischer Korrektheit der Klasse

Compiler meldet keinen Fehler

- Alle Elemente (Daten, Methoden): 'private'
- (alle) Daten: 'public'; alle Methoden: 'private'
- Alle Elemente der Klasse C: 'static'  
+ Instanzenbildung new C()
- Variablen: 'private', non-static; alle Methoden: 'static'
- Alle Variablen: 'static'; alle Methoden: non-static
- Weitere?

# Beispiel: alle Elemente 'private'

```
class Time {  
    private int hour, minute;  
    private final int noonHour = 12;  
    private addMinutes (int m) {  
        ...  
    }  
}
```

Niemand  
erreichbar

# Beispiel: (alle) Daten 'public' alle Methoden 'private'

```
class Time {  
    public int hour, minute;  
    public final int noonHour = 12;  
    private addMinutes (int m) {  
        hour = ...  
    }  
}
```



Nutzlose  
Methoden

# Beispiel: alle Elemente 'static' + Instanzenbildung

```
class Time {  
  
    public static int hour, minute;  
    public final static int noonHour = 12;  
  
    public static addMinutes (int m) {  
        hour = ...  
    }  
}
```



Leere  
Instanzen

```
class Apply {  
  
    Time t1, t2;  
    public static void main(...) {  
        t1 = new Time ();  
        t2 = new Time ();  
    }  
}
```

# Beispiel: Variablen 'private', non-static alle Methoden 'static'

```
class Time {  
  
    private int hour, minute;  
  
    private final int noonHour = 12;  
  
    public static Time() {  
        ...  
    }  
  
    public static addMinutes (int m) {  
        hour = ...  
    }  
  
}
```

Keine Bearbeitung der  
Instanzvariablen  
möglich

ok ?

Klassenmethoden (static) können Instanzvariablen nicht bearbeiten

# Beispiel: Variablen 'static' alle Methoden non-static

```
class Time {  
  
    private static int hour, minute;  
    private final static int noonHour = 12;  
  
    public addMinutes (int m) {  
        hour = ...  
    }  
}
```

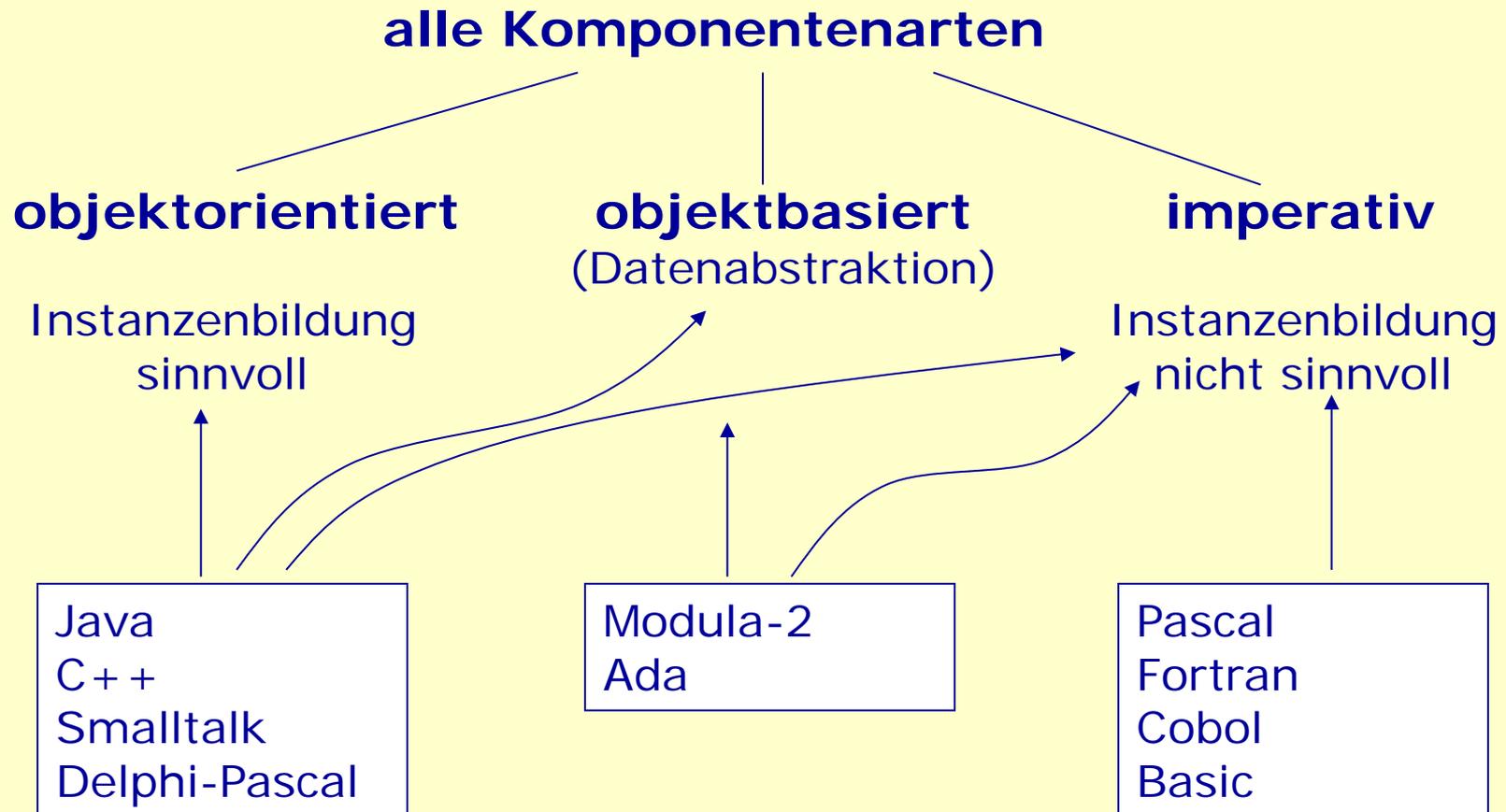
Instanzen  
ohne Daten

```
Time t1, t2;  
t1 = new Time ();  
t2 = new Time ();  
t1.addMinutes(20);  
t2.addMinutes(10);
```

Methoden aller  
Instanzen identisch

# Komponentenarten und Programmiersprachen

Problem: Welche Programmiersprache kann welche Komponentenarten unterstützen?



# Anwendung: Komponentenarten (Klassifikation)

- Erkennen der Komponentenart aufgrund syntaktischer Merkmale (private / public – static / non-static)

```
class Counter {  
    private int x;  
    public Counter () {  
        x = 0;  
    }  
    public void count () {  
        x++;  
    }  
}
```

Bedeutung?

Art?

Was fehlt der Klasse?

- Selbstständige Entwicklung sinnvoller Komponenten = die "Kunst" der oo-Programmierung (viel Erfahrung !!)