

## 2. Objektorientierung: grundlegende Fallbeispiele und weitere Konzepte

Java-Beispiele:

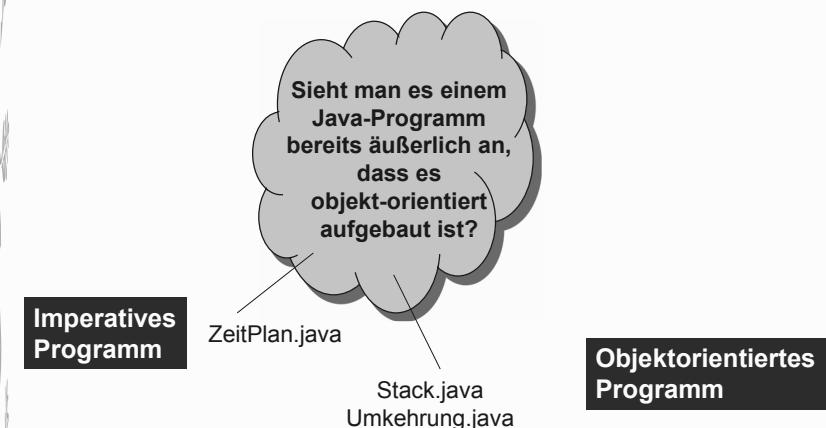
KlammerStruktur.java  
Time.java  
Schedule.java

## Schwerpunkte

- ▶ **Wiederverwendbarkeit** von ADT (Klassen):
  - Überprüfung von Klammerstrukturen mit einem Stack
  - Auflösung von Rekursion mit einem Stack
- ▶ **Vergleich** eines imperativen mit einem oo Programm:  
ZeitPlan.java, Time.java, Schedule.java
- ▶ **Neue Konzepte der OO**:  
Überladen (Overloading),  
this – das aktuelle Objekt,  
private Methoden, public Variablen,  
Alias-Problem,  
Gleichheit und Identität, Kopie von Objekten,  
lexikographische Ordnung
- ▶ ADT als Java-API-Klassen: **String**
- ▶ **Datenklassen**: Pascal-Records, C++-Struct

## Äußere Merkmale objektorientierter Programme in Java

## Wodurch unterscheiden sich in Java oo- und imperativ strukturierte Programme?



## Objektorientiert

```
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--;
    }
}
```

Sieht man es einem Java-Programm bereits äußerlich an, dass es objekt-orientiert aufgebaut ist?

## Imperativ

```
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)
    static void main
    (String[] args) {
        includeNewEntry(90, "V P11");
        includeNewEntry(15, "Pause");
        includeNewEntry(90, "V Th11");
    }
}
```

## Wodurch unterscheiden sich in Java oo- und imperativ strukturierte Programme?

### Objektorientierte Struktur:

Programm mit Klassen, die ADT realisieren

ADT = Einheit aus versteckten Daten und Interface-Operationen

ZeitPlan.java  
Stack.java  
Umkehrung.java

- 1 class Stack {  
    private char[] stackElements;  
    public void push (char x) { ... }  
}
- 2 Variablen und Methoden nicht ,static'  
→ Instanzen erzeugen:  
Instanzmethoden, -Variablen

## Objektorientierte Komponente

```
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--;
    }
}
```

1 ADT = Einheit aus versteckten Daten und Interface-Operationen

2 Variablen und Methoden nicht ,static'  
→ Instanzen erzeugen:  
Instanzmethoden, -Variablen

**imperativ**

```
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) {
        ...
    }
}
```

## Wiederverwendbarkeit

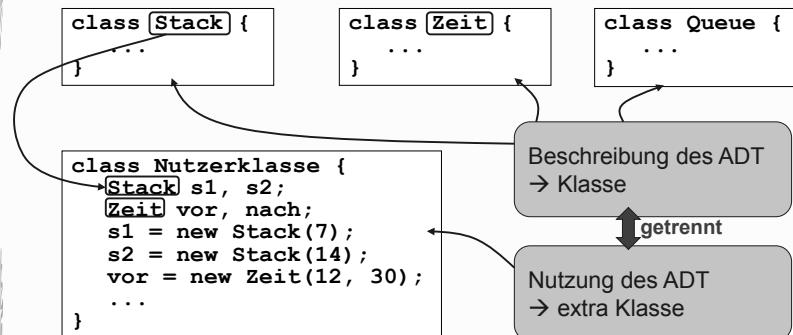
KlammerStruktur.java

Wesentliches Qualitätsmerkmal von Software  
Besonders durch OO unterstützt

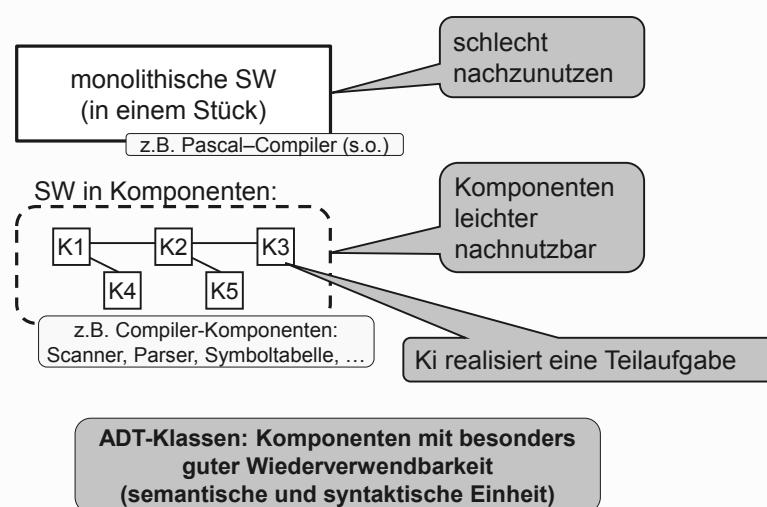
# OO: Zwang zur Komponentenbildung

Klasse: neuer (nutzerdefinierter, abstrakter) Typ

je Typ/ADT – eine Klasse  
→ Zwang zur Komponentenbildung



## Wiederverwendbarkeit von Software: eines der wichtigsten Qualitätskriterien



## Imperatives Programm: als Einkomponentenprogramm möglich

```
PROGRAM pascalcompiler;
CONST c1 ... (100 LOC)
TYPE t1 ... (500 LOC)
VAR v1, v2 ... (500 LOC)
PROCEDURE P1 ... PROCEDURE P2 ... (9000 LOC)
BEGIN
    ...
END.
```

Pascal

```
class pascalcompiler {
    final static int c1 ...
    static char[] v1, v2;
    static int p1(...) {...}
    static void p2 ...
    main(...) {
        ...
    }
}
```

Java

LOC = Lines Of Code

Wiederverwendung von Programmteilen schwierig

## Programmieraufgaben

- 1. Einlesen einer Folge von Daten (int/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 ] ++ ;
} ...
```

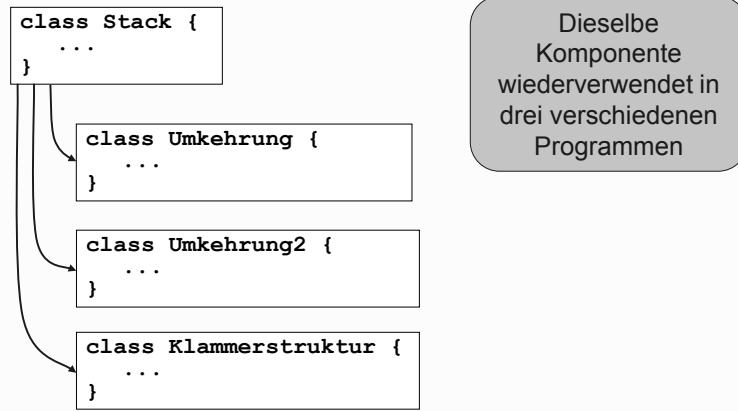
- 3. Überführung von Ausdrücken in Postfixform

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

- 4. Auflösung der Rekursivität in Iteration

# Neue Anwendung des Stack: Klammerstruktur überprüfen

Klassen (unabhängige Komponenten):  
→ sehr gute Wiederverwendbarkeit (vgl. SW-Qualität)



**KlammerStruktur.java**

```

class KlammerStruktur {

    public static void main(String[] args) {

        final int N = 100;
        char [] eingabe = new char [N];           // Eingabe
        int j = 0;                                // gefüllt bis zur Länge j-1
        int i = 0;                                // Index: durchläuft das Eingabeprogramm
        boolean ok = true;                         // zu Beginn: kein Fehler
        Stack s = new Stack(20);                   // Klammerstack

        System.out.println("Ausdruck mit Klammern eingeben: (), [], () (Ende: .):");
        do {
            eingabe[j] = Keyboard.readChar(); j++;
        } while ( eingabe[j-1] != '.' ); // Eingabe endet mit '.'

        while ((i < j) && (ok)) { // solange noch Eingabzeichen vorhanden und kein Fehler aufgetreten ist
            switch (eingabe[i]) {
                case '(': // öffnende Klammer: abspeichern
                case ')': s.push(eingabe[i]); break;
                case '[': if (!s.isEmpty() && s.top() == ']') {
                    s.pop();
                } else ok = false; break;
                case ']': if (!s.isEmpty() && (s.top() == '[')) {
                    s.pop();
                } else ok = false; break;
                default: break; // keine Klammer
            }
            i++;
        } // Ende-Test: alle Zeichen der Eingabe erfasst und ...

        if ((i==j) && ok && s.isEmpty())
            System.out.println("Klammerstruktur ok!");
        else System.out.println("Klammerstruktur falsch!");
    }
}

```

# Klammerstruktur überprüfen

- Aufgabe:

In einem Programm treten Paare von Klammern  $() - [] - \{ \}$  auf, die auch ineinander verschachtelt sein können. Man überprüfe die Klammerstruktur (ohne Syntaxanalyse).

- korrekte Strukturen:

```
(a + b[i] * (c - d))
{ a = arr[i] - (c + d) }
```

- fehlerhafte Strukturen:

```
((a && b) - c)      (Anzahl)
((a[i+]))          (falsche Paare)
```

# Algorithmus (Klammerstruktur)

- Verarbeite Eingabe zeichenweise (Zyklus):

- Zeichen ist öffnende Klammer:  
→ abspeichern
- Zeichen ist schließende Klammer:
  - Test: ist die zuletzt abgespeicherte Klammer die zugehörige öffnende Klammer?
    - ja: ok + streiche öffnende Klammer
    - nein: Fehler → stop
  - Sonstige Zeichen: ignorieren/überlesen

$(a + b[i] * (c - d))$

- Ende:  
positiv:

- Gesamte Eingabe fehlerfrei verarbeitet **und**
- Keine abgespeicherten Klammern mehr (keine öffnende Klammer zu viel)

negativ: sonst

Wieso ist ein Stack hilfreich  
- Was deutet im Algorithmus auf einen Stack hin?

## Algorithmus benötigt Stack

- Verarbeite Eingabe zeichenweise (Zyklus):
  - Zeichen ist öffnende Klammer:  
→ abspeichern
  - Zeichen ist schließende Klammer:
    - Test: ist die zuletzt abgespeicherte Klammer die zugehörige öffnende Klammer?
      - ja: ok + streiche öffnende Klammer
      - nein: Fehler → stop
    - Sonstige Zeichen: ignorieren/überlege
- Ende:
  - positiv:
    - Gesamte Eingabe fehlerfrei verarbeitet **und**
    - Keine abgespeicherten Klammern mehr  
(keine öffnende Klammer zu viel)
  - negativ: sonst

## Implementation: Klamertest

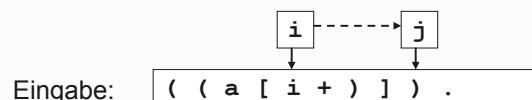
```

aktuelles Zeichen
switch (eingabe[i]) {
  case '(':           // öffnende Klammern
  case '{':           s.push(eingabe[i]);
  case '[':           break;
  case ')': if (!s.isEmpty() &&
                s.top() == '(')
    s.pop();
    else ok = false;
    break;
  ...
  default: break;    // keine Klammer
}

```

KlammerStruktur.java

## Implementation: Ende-Test



```

// Ende-Test positiv:
// - alle Zeichen der Eingabe erfasst
// - keine Fehler entdeckt
// - keine offnende Klammer zu viel
if ((i == j) && ok && s.isEmpty())
  System.out.println("Klammerstruktur ok!");
else
  System.out.println("Klammerstruktur falsch!");

```

## Auflösung von Rekursivität mit Stacks

Weitere Nachnutzung des Stack

## Programmieraufgaben

- ✓ 1. Einlesen einer Folge von Daten (int/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  
(Effizienz; nicht: Übersichtlichkeit)

**Übungsblatt 5**

## Hanoi: rekursiv → iterativ (mit Stack)

### Grundprinzip:

- in einem (Zyklus-)Schritt:
  - löse immer **das** aktuelle Problem und merke die später zu lösenden Probleme (in einem **Stack**)
- ein Problem:
  - bewege n Scheiben von 'start' über 'hilfe' nach 'ziel'
- im Stack:
  - das aktuelle Problem ist 'oben' gespeichert
- Algorithmus:

```
while (!isempty(problemStack)) {
    bearbeite das oberste Problem
}
```

n = 1 → drucken  
n > 1 → speichere drei neue Probleme

## Hanoi-Programm

```
static void bewege (
    int n,
    char start,
    char hilfe,
    char ziel) {

    if (n == 1)
        System.out.println
            (" von " + start + " nach " + ziel);
    else {
        bewege(n-1, start, hilfe, ziel);
        System.out.println
            (" von " + start + " nach " + ziel);
        bewege(n-1, hilfe, start, ziel);
    }
}
```

Ohne Rekursion möglich (verbesserte Effizienz)

→ mit Stack

1 Problem → 3 Teilprobleme (mit 2 rekursiven Aufrufen)

## Beispiel: 'Problem-Stack'

bewege(5, 'A', 'B', 'C')

Dynamische Aufruffreihenfolge von Stack-Operationen:

Stackentwicklung  
Anfang:



push()

top()



pop()

push()

push()

1. Zyklusschritt:



top()

pop()

push()

push()

2. Zyklusschritt:

4 Stacks oder ein Stack mit Basistyp „Record/Struct“

## Vergleich:

imperatives – objektorientiertes Programm

Objektorientierte Variante eines existierenden imperativen Programms

OO-Variante des  
imperativen Programms  
ZeitPlan.java entwickeln –  
was heißt das?

Teil II: ZeitPlan.java (→ Quellcode)

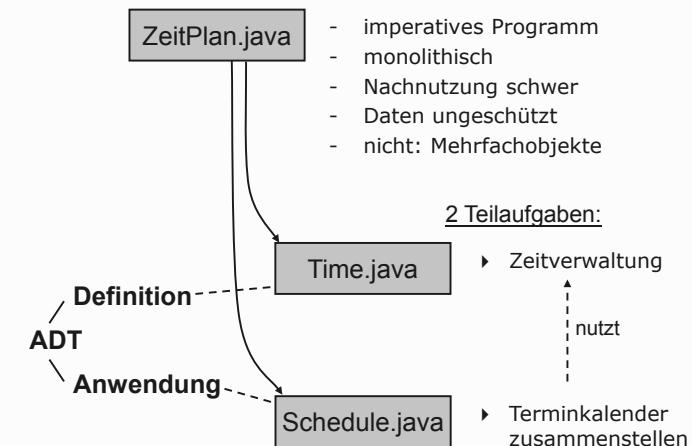
Jetzt:

Time.java  
Schedule.java

illustriert auch neue Konzepte:  
Overloading, Alias,  
Gleichheit und Identität u.a.

## Vergleich: imperatives und objektorientiertes Programm

Terminkalender-Programm



## Beispielanwendung

```
% java ZeitPlan
Terminkalender: zeitangabe + Text
-----
8:30AM      V PI1
10:00AM     Pause
10:15AM      V THI1
11:45AM     Pause
12:15PM      U PI1
Letzte (aktuelle) Tageszeit in Minuten:
1:45PM = 825. Minute des Tages
```

## Implementation:

Imperatives Programm Zeitplan.java

### Idee

→ **Verwaltung der aktuellen Zeit:**  
2 globale Variablen (hour, minute)

→ **Methoden (Algorithmen) für Teilaufgaben:**

- die aktuelle Zeit erhöhen (addMinutes)
- die Uhrzeit in Minuten umrechnen (timeInMinutes)
- Druck der Uhrzeit nach den englischen Konventionen (printTime)
- Druck der Uhrzeit und der entsprechenden Minutenzahl (printTimeInMinutes)
- Verarbeitung eines neuen Eintrags (includeNewEntry)
- Hauptalgorithmus (main)

# Imperative Programmstruktur:

eine Klasse – zwei Aufgaben

```
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(); ...  
    }  
}
```

Ziel: Klasse Time  
→ separate Klasse  
→ nutzbar,  
nicht nur bei  
Zeitplanerstellung  
→ mehrere  
Terminkalender,  
mehrere Zeitobjekte

Terminkalender  
zusammenstellen  
(Zeitverwaltung  
genutzt)

# Imperative Programmstruktur: Nachteile

Programm Zeitplan.java

## Einkomponentenprogramm:

- nicht möglich:  
Nachnutzung der Zeitverwaltung  
(ADT 'Time') in anderer Anwendung  
(z. B. Simulationsaufgabe)
- nicht möglich:  
mehrere Zeitobjekte  
(z. B. für mehrere Terminkalender)
- Daten **nicht** geschützt (versteckt):  
in main() möglich: minute = 65;

## Zeitplan: imperativ → objektorientiert

```
class Zeitplan {  
    ...  
}  
  
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, t2, t3, t4;  
        ...  
    }  
}
```

eine imperativ  
Komponente

Time.java  
Schedule.java

ADT 'Time'

Anwendung  
von 'Time':  
4 Terminkalender  
zusammenstellen  
→ 4 Zeitobjekte  
benötigt

## Aufruf der Terminplanung

```
% java Schedule  
erster Plan:  
8:30AM   P1  
10:00AM  Pause  
10:15AM  T11  
11:45AM  Pause  
noon     Ma1  
1:40PM   Pause  
1:55PM   Ma2  
letzte (aktuelle) Tageszeit in Minuten:  
2:15PM   = 855. Minute des Tages  
  
zweiter Plan:  
midnight Nebenfach  
1:40AM   Proseminar  
letzte (aktuelle) Tageszeit in Minuten:  
3:10AM   = 190. Minute des Tages  
  
dritter Plan:  
2:15PM   Sport  
3:55PM   Freizeit  
letzte (aktuelle) Tageszeit in Minuten:  
7:15PM   = 1155. Minute des Tages  
  
vierter Plan:  
noon     zweites Nebenfach  
1:40PM   frei  
letzte (aktuelle) Tageszeit in Minuten:  
5:00PM   = 1020. Minute des Tages
```

4 Pläne –  
4 Zeitobjekte

## Konzepte:

Overloading

this

Alias-Problem

Kopie von Objekten

Gleichheit und Identität

am Beispiel von  
Time.java, Schedule.java

## Überladen von Methoden (Overloading)

```
class Time {  
    private int hour, minute;  
    // die aktuelle Zeit  
  
    public Time(int h, int m) {  
        hour = h; minute = m;  
    }  
  
    public Time() {  
        hour = 0; minute = 0;  
    }  
    ...  
}
```

Time.java

Warum zwei Konstruktoren (Unterschied)?

### Overloading (Überladen):

- Derselbe Name für unterschiedliche Methoden
  - Unterschiedliche Anzahl von Parametern oder unterschiedliche Parametertypen
- Beim Aufruf klar: Welche Methode ist gemeint.

## Methoden mit zwei Objekten derselben Klasse

Vergleich: Liegt ein Time-Objekt vor einem anderen?

das aktuelle Objekt, zu dem die Methode gehört

Time.java

```
public boolean before (Time t) {  
    return ((hour < t.hour) ||  
            (hour == t.hour) &&  
            (minute < t.minute));  
}
```

Variable des aktuellen Objekts

Variable des Parameterobjekts

aktueller Objekt

Parameter-Objekt

Aufruf: `t4.before(t1);`

## this: das aktuelle Objekt

Methode after():

Das aktuelle Zeitobjekt liegt nach dem anderen Zeitobjekt t2, wenn t2 vor dem aktuellen Zeitobjekt liegt.

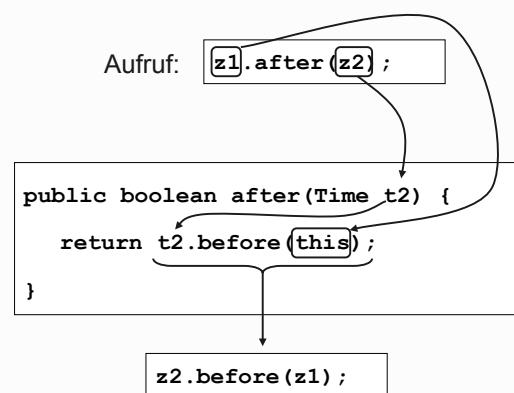
```
public boolean after(Time t2) {  
    return t2.before( this );  
}
```

Das aktuelle Objekt

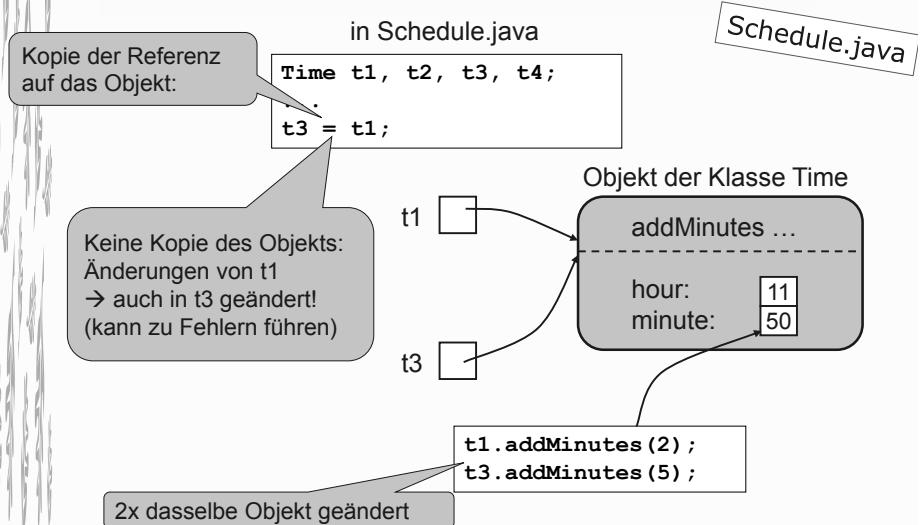
Problem:  
das aktuelle Objekt muss als Parameter übergeben werden,  
hat aber keinen Namen

Time.java

## this: Aufrufbeispiel



## Alias-Problem: zwei Namen für dasselbe Objekt



## Aufruf der Terminplanung

```

% java Schedule
erster Plan:      t1
8:30AM   PI1
10:00AM  Pause
10:15AM  TI1
11:45AM  Pause
noon     Ma1
1:40PM   Pause
1:55PM   Ma2
letzte (aktuelle) Tageszeit in Minuten:
2:15PM   = 855. Minute des Tages

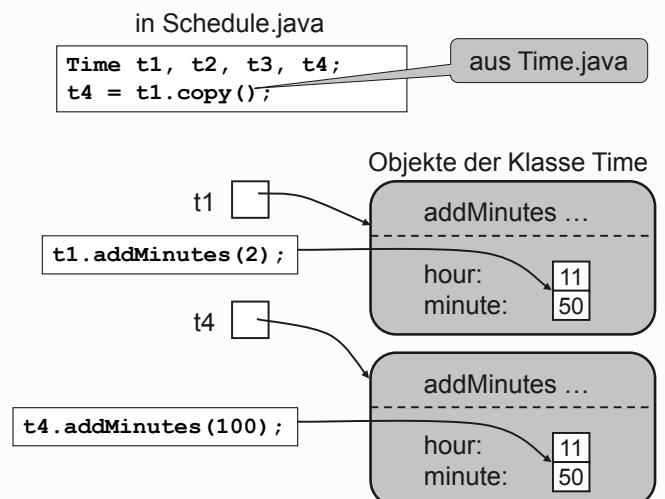
zweiter Plan:
midnight Nebenfach
1:40AM   Proseminar
letzte (aktuelle) Tageszeit in Minuten:
3:10AM   = 190. Minute des Tages

dritter Plan:      t3
2:15PM   Sport
3:55PM   Freizeit
letzte (aktuelle) Tageszeit in Minuten:
7:15PM   = 1155. Minute des Tages

vierter Plan:      t4
noon     zweites Nebenfach
1:40PM   frei
letzte (aktuelle) Tageszeit in Minuten:
5:00PM   = 1020. Minute des Tages

```

## Explizite Kopien von Objekten

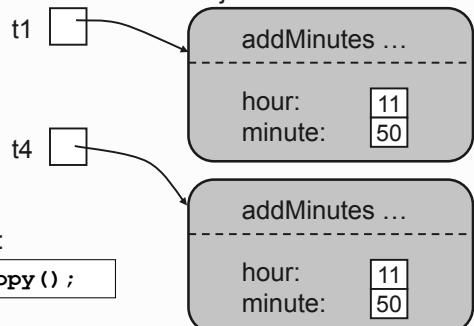


## Kopie von Objekten: Implementation

in Time.java:

```
public Time copy () {  
    return new Time(hour,minute);  
}
```

Objekte der Klasse Time



in Schedule.java:

```
t4 = t1.copy();
```

## Gleichheit von Objekten

in Schedule.java

```
t3 = t1;  
t4 = t1.copy();  
  
if (t1 == t3) ...  
if (t1 == t4) ...
```

`==` testet auf Identität der Objekte



## public-Variablen, private-Methoden

ADT in Java (Grundform):

- private Variablen
- public Methoden

- **public-Variablen:** sichtbar für die Außenwelt  
→ bequemerer Zugriff (Vorsicht!)
- **private-Methoden:** Hilfsfunktionen (unsichtbar ...)  
→ korrekter Gebrauch (keine Gefahr)

```
public int hour;  
private int timeInMinutes () {  
    int totalMinutes = (60*hour + minutes) % (24*60);  
    ...  
}
```

in Klasse Schedule: `h1 = t2.hour;`

Nur zur Demonstration – hier nicht gerechtfertigt!

## Datenklassen:

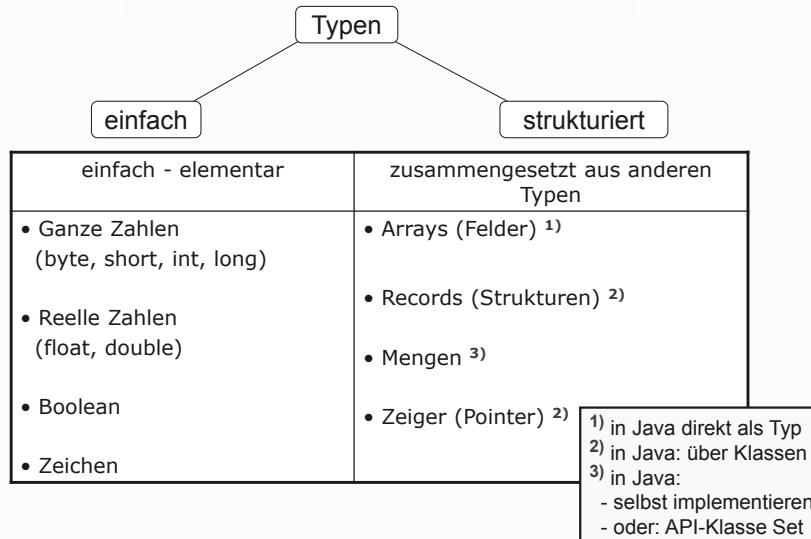
Pascal-Records

C-struct

**Array:** Sammlung von Elementen desselben Typs

**Record/Struktur:** Sammlung von Elementen beliebiger / unterschiedlicher Typen

## Klassifikation von Typen in Programmiersprachen



## Datenklassen: Klassen als Sammlung von Daten

Beispiel:

- Punkte im dreidimensionalen Raum bestehen aus drei Werten:  
x-, y- und z-Koordinaten

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

Spezialfall eines ADT:

- nur (sichtbare) Daten – keine Operationen

→ sinnlos: 

```
private double x, y, z;
```

## Punkt3D-Beispiel

Klassendeklaration

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

Nutzung der Klasse: Variablen deklarieren

```
Punkt3D mitte = new Punkt3D();
Punkt3D ecke = new Punkt3D();
mitte.x = 0; mitte.y = 0; mitte.z = 0;
ecke.x = 3; ecke.y = 5; ecke.z = 10;
```

mitte

ecke

## Records (Pascal) und Struct-Type (C++) in Java als Datenklassen

Pascal

```
TYPE Punkt3D =
  RECORD
    x, y, z: REAL
  END;
VAR mitte: Punkt3D;
... mitte.x := 0;
```

C / C++

```
typedef struct {
    double x, y, z;
} Punkt3D;

Punkt3D mitte;
mitte.x = 0;
```

## Anwendung: Hanoi-Programm

Problem-Stack:

4 Einzelstacks mit Basistypen int, char, char, char



Ein Einzelstack mit Basistyp 'Problem':

```
class Problem {  
    public int n;  
    public char start, hilfe, ziel;  
}  
  
class ProblemStack {  
    private Problem[] stackelements;  
    private int top;  
    // Problemstack(),  
    // push(), pop(), top(), isempty()  
}
```

## Bedeutung der Klasse ?

```
class C1 {  
  
    private int a, b;  
  
    public C1(...) {  
        ...  
    }  
    // weitere Methoden  
}
```

Datenrepräsentation (int a, b) könnte bedeuten:

- komplexe Zahlen a + bi
- rationale Zahlen a / b
- Tageszeit: a - Stunden  
b - Minuten
- Datum: Tag a / Monat b
- Punkt (a, b) der Ebene

Entscheidend ist nicht, welche Daten in der Klasse zusammengefasst sind, sondern das, was mit den Daten passieren soll:

**Die Operationen bestimmen die Semantik der Klasse.**

## API-Klasse

### java.lang.String

## Java-API: Vielzahl klassischer ADT

ADT = Einheit versteckter Daten + Zugriffsoperationen

- String: Zeichenketten
- Hashtable: offene Hashtechnik
- Applet-Klasse (Java-Programme in Webseiten)
- Grafik-Programmierung: Graphics
- GUI (graphic user interface): Label, TextField, Button, Checkbox, ...
- Ein- und Ausgabe: FileInputStream, FileOutputStream, ...
- ...

## Klasse java.lang.String

Erzeugung:

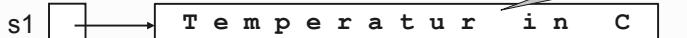
```
String s1 = "Temperatur in C";
String s2 = new String("Tabelle");
char[] cArr = {'S','t','r','i','n','g'};
byte[] bArr = {83,116,114,105,110,103};
String s3 = new String(cArr);
String s4 = new String(bArr);
```

(ca. 10 überladene Konstruktoren)

Besonderheit:  
Erzeugung ohne  
Konstruktor

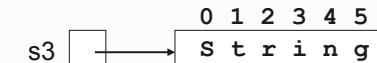
ASCII

s1



## String: Methoden (1)

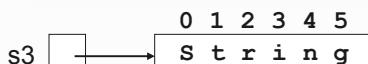
- Keine Methoden zur Veränderung von Strings



- Abfrage des String-Inhalts

	return-Wert
s3.charAt(4)	-> 'n'
s3.substring(3)	-> "ing"
s3.substring(3,5)	-> "in" halboffenes Intervall [3,5]
s3.length()	-> 6
s3.indexOf('n')	-> 4
s3.startsWith("St")	-> true
s3.endsWith("inge")	-> false

## String: Methoden (2)



- Vergleich: lexikographische Ordnung

int compareTo(String s)

return-Wert

s3.compareTo("String")	0	"String" = "String"
s3.compareTo("Ttring")	< 0	"String" < "Ttring"
s3.compareTo("Aaaaaaa")	> 0	"String" > "Aaaaaaa"
s3.compareTo("String1")	< 0	"String" < "String1"
s3.compareTo("S")	> 0	"String" > "S"

c0 c1 c2 c3 ...

Vergleichsrichtung: Unicode-Werte  
(erster verschiedener Wert)

d0 d1 d2 d3 ...

## String: Methoden (3)

- viele weitere Methoden

z.B. Erzeugung neuer Strings (return-Typ)

- Umwandlung in Großbuchstaben
- neue Kopie erzeugen (+ Ersetzung an bestimmten Stellen)
- ...

→ API-Dokumentation

- vgl. auch Klasse 'StringBuffer'

wie String + Änderungsoperationen