8. Grundkonzepte der Objektorientierung (5):

Interfaces

Java-Beispiele:

ScheduleInt.java
ScheduleAbstr.java
UmkehrungNU.java
KeyboardIApp.java
Druck.java

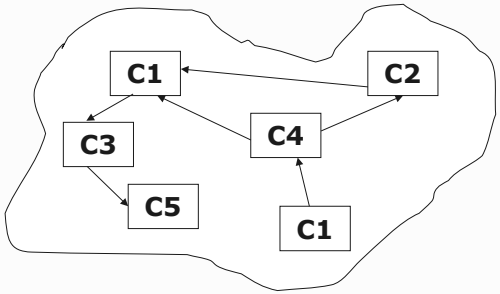
Schwerpunkte

- Aufgaben des Interface
- Java-Interface als Abstraktion
- Implementation von Interfaces: *implements*
- Interface: wirkt wie neuer Typ
  (Variablen vereinbaren u.a.)
- Interface: Spezialfall abstrakter Klassen
- Interface: erlaubt …
  (eingeschränkte) *Mehrfachvererbung* in Java
- Typische Anwendungsfälle

Interface:

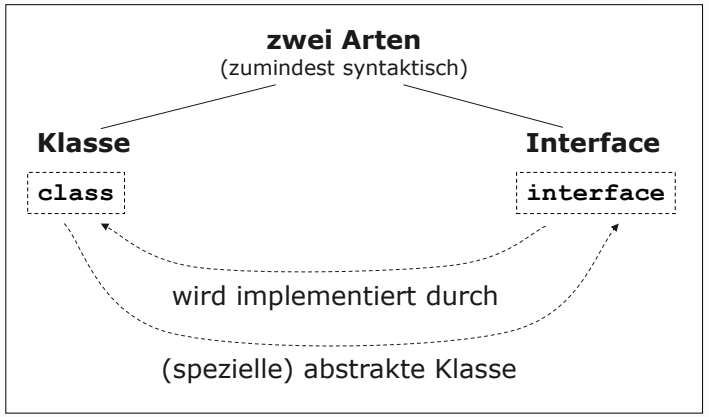
Überblick und Grundprinzip

Java-Programm: Menge von Komponenten



Komponente Ci: Klasse oder Interface

Komponenten in Java



Klassenkonzept kann Interface modellieren

Interface benötigt Klassenkonzept

Also: man kommt (fast) ohne das Interface aus

Java-Syntax: Komponentenarten

Satzsymbol

Quelltextdatei ::=
[Paketfestlegung]
{Import}
{Typdeklaration}

Typdeklaration ::=
Klassendeklaration |
Schnittstellendeklaration .

Die beiden Arten von Komponenten

Beispiel: volle Form der Quelltextdatei

```
package demo;

import java.util.*

class C1   {
   ...
}

interface I1   {
   ...
}
```

Definition einer Komponente C1/I1 kann voraussetzen:

- package: Zuordnung zu einem Paket
  (Sammlung von Klassen mit ähnlichen Aufgaben)

- import: Paket angegeben, aus dem Klassen verwendet werden sollen

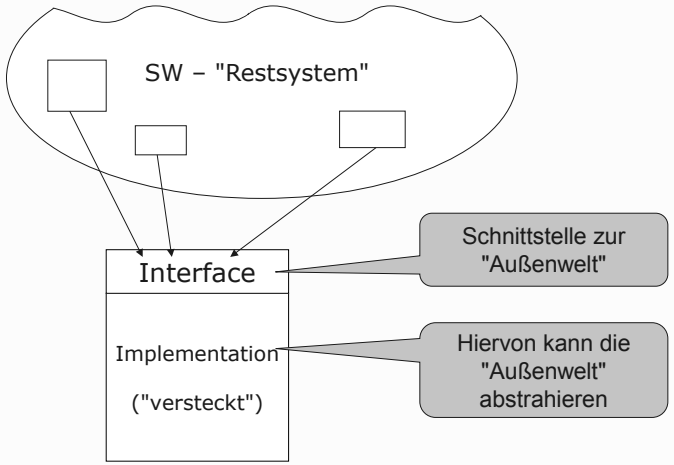
Rolle von Java-'interface'-Komponenten

- **Inhaltlich-logische Funktion:**
  - **Trennung Interface - Implementation**

- **Technische Funktion:**
  - **eingeschränkte Mehrfachvererbung**

SW-Komponenten sind Abstraktionen

Komponente: Interface + Implementation



Java-Klasse: Wo ist die Abstraktion?
(Was vom Code ist das Interface – also nach außen für den Nutzer sichtbar bzw. wichtig?)

```
class Time   {
   private int hour, minute;
   ...
   public void addMinutes (int m)   {
      int totalMinutes =
         (60*hour + minute + m) % (24 * 60);
      if (totalMinutes < 0)
         totalMinutes = totalMinutes + 24 * 60;
      hour = totalMinutes / 60;
      ...
   }
   ...
}
```

Java-Klasse ist Mischung aus Interface und Implementation: Abstraktionsprinzip nicht umgesetzt

- Interface - Implementation

Was wir eigentlich benötigen: Aufteilung der Klasse in Interface + Implementation

```
class Time{
   private int hour, minute;

   public void addMinutes (int m)   {
      int totalMinutes =
        (60*hour + minute + m) % (24 * 60);
      if (totalMinutes < 0)
         totalMinutes = totalMinutes + 24 * 60;
      hour = totalMinutes / 60;
         ...
   } ...
}
```

Implementation von Time

```
private int hour, minute;

public void addMinutes (int m)   {
  int totalMinutes =
    (60*hour + minute + m) % (24 * 60);
  if (totalMinutes < 0)
     totalMinutes = totalMinutes + 24 * 60;
  hour = totalMinutes / 60;
     ...
} ...
}
```

Interface von Time

```
interface Time   {
   public void addMinute (int m);
   // weitere Methoden
}
```

Nur das interessiert den Nutzer der Klasse

C++-Klasse: eine Abstraktion ?

```
class Time   {

   public:
        Time ();
        Time (int h, int m);
        void addMinutes (int m);
        void printTime ();
        ...

   private:
        int hour, minute;
}
```

Konsequenz: Klassen in C++ sind bessere Abstraktionen als Klassen in Java

Aber: Java-Interface überwindet diesen Mangel

Extra File in C++: Implementation aller Methoden

- Interface - Implementation

Interface in Java:

- Definition
- Implementation
- Anwendung
- Vergleich mit abstrakten Klassen

Klasse → Interface + Implementation

```
class Time{
   private int hour, minute;

   public void addMinutes (int m)   {
    int totalMinutes =
     (60*hour + minute + m) % (24 * 60);
    if (totalMinutes < 0)
      totalMinutes = totalMinutes + 24 * 60;
    hour = totalMinutes / 60;
        ...
   } ...
}
```

bisherige Technik

```
interface TimeI   {
   public void addMinutes (int m);
   public void substractMinutes (int m);
   public void printTime ();
   public void printTimeInMinutes ();
}
```

alternative Technik

```
class Time implements TimeI  {
   private int hour, minute;

   public void addMinutes (int m)   {
    int totalMinutes =
     (60*hour + minute + m) % (24 * 60);
    if (totalMinutes < 0)
      totalMinutes = totalMinutes + 24 * 60;
    hour = totalMinutes / 60;
        ...
   } ...
}
```


Java–Interface: eine (fast) "saubere" Abstraktion

ScheduleInt.java

```
interface TimeI   {
   public void addMinutes (int m);
   public void substractMinutes (int m);
   public void printTime ();
   public void printTimeInMinutes ();
}
```

- Nur die Köpfe der Methoden
- Keine Daten + keine Algorithmen
- Was fehlt dem Nutzer noch?

Konstruktoren (im Interface nicht erlaubt):

```
TimeI(int h, int m);
TimeI();
```

Grund: es können keine Instanzen eines Interface erzeugt werden.

Implementation eines Interface durch Klassen

Nutzer

ScheduleInt.java

```
interface TimeI   {
   public void addMinute (int m);
   public void substractMinutes (int m);
   public void printTime ();
   public void printTimeInMinutes ();
}
```

Beachte: Größenverhältnis Interface - Implementation

```
class Time implements TimeI  {
   private int hour, minute;

   public Time (int h, int m)   {...}
   public void addMinutes (int m)   {
       int totalMinutes =
          (60*hour + minute + m) % (24 * 60);
       if (totalMinutes < 0)
          totalMinutes = totalMinutes + 24 * 60;
       hour = totalMinutes / 60;
       ...
   }
   ...
}
```

Nutzer muss auch von implementierender Klasse etwas kennen: Name, Konstruktoren

Hinzugefügt:
- private Variablen
- Konstruktor
- Körper der Methoden

Anwendung eines Interface

```
interface TimeI   {
   public void addMinute (int m);
   public void substractMinutes (int m);
   public void printTime ();
   public void printTimeInMinutes ();
}
```

```
class Time implements TimeI  {
   private int hour, minute;
   public Time (int h, int m)
   public void addMinutes (int m)
   ...
}
```

ScheduleInt.java

Interface wie neuer (nutzerdefinierter) Typ

```
public static void main (...) {
   TimeI t1;

   t1 = new Time(8,30);
   t1.addMinutes(30);
}
```

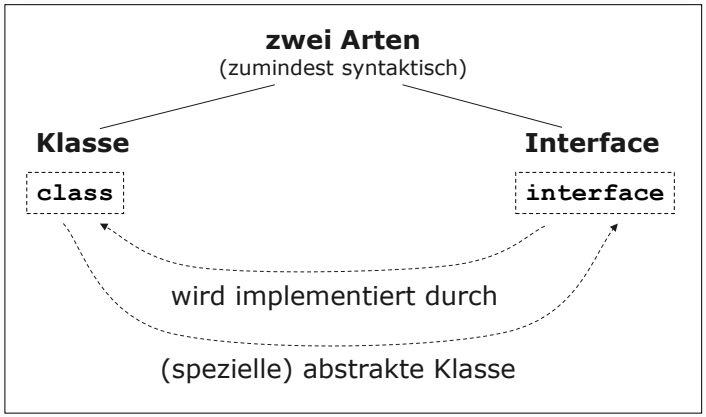
Variablen vom Interface-Typ 'TimeI'

Erlaubte Werte: Objekte von ALLEN implementierenden Klassen (Kompatibilitätsregel)

Konstruktor identifiziert DIE ausgewählte implementierende Klasse

Deshalb: Konstruktor nicht Teil des Interface

Interface: Spezialfall abstrakter Klassen



Mit abstrakten Klassen kann Interface modelliert werden

Interface: Spezialfall abstrakter Klassen

ScheduleAbstr.java

```
abstract class TimeI   {
   public abstract void addMinutes (int m);
   public abstract void substractMinutes (int m);
   public abstract void printTime ();
   public abstract void printTimeInMinutes();
}

class Time extends TimeI   {
   private int hour, minute;
   public void addMinutes (int m);   {
       ...
   }
   ...
}
```

Klasse wird erweitert

Interface wird implementiert

Interface und abstrakte Klasse:
logisch identisch – technische Unterschiede

Rolle von Java-'interface'-Komponenten

- **Inhaltlich-logische Funktion**
  - Trennung Interface - Implementation

```
public static void main (...)
{
   TimeI t1;
   t1 = new Time(8,30);
   ...
}
```

Nutzerkomponente

Als eine logische Einheit auffassen:

```
interface TimeI (...)
```

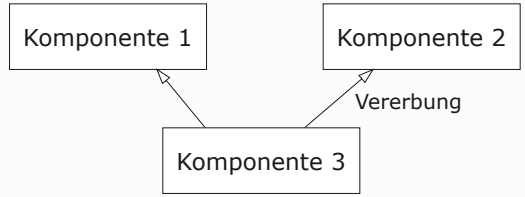
```
class Time implements TimeI{...}
```

- **Technische Funktion:**
  - eingeschränkte Mehrfachvererbung

Mehrfachvererbung in Java durch Interface-Komponenten

Interface: Mehrfachvererbung

für Klassen verboten – für Interface erlaubt:




```
(abstract) class K1 ...
(abstract) class K2 ...
class K3 extends K1, K2 ...
```

falsch

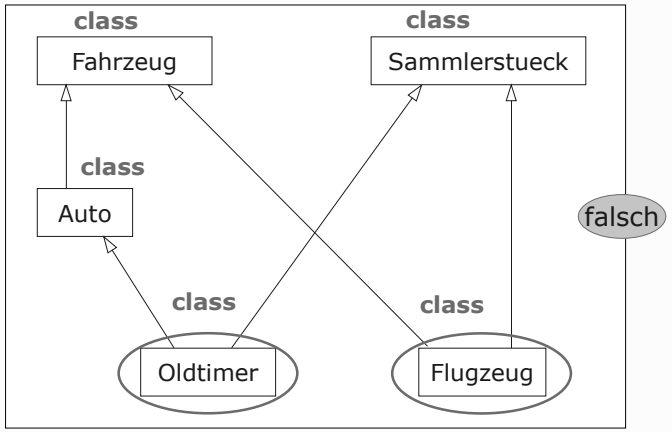
```
interface I1 ...
interface I2 ...
class K3 implements I1, I2
...
```

ok

```
interface I1 ...
(abstract)class K2 ...
class K3 implements I1
          extends K2 ...
```

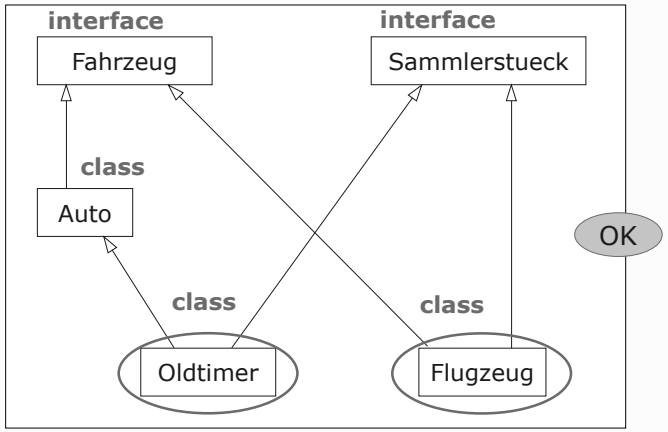
ok

Beispiel: Hierarchie mit Mehrfachvererbung (1)



Man darf nicht von 2 Klassen erben.

Beispiel: Hierarchie mit Mehrfachvererbung (2)



Man darf nicht von 2 Klassen erben.

Beispiel: Vererbungshierarchie in Java

```
interface Fahrzeug ...

interface Sammlerstueck ...

class Auto implements Fahrzeug ...
```

Erbt alle Elemente (Variablen, Methoden) von 'Auto' und 'Sammlerstueck'

```
class Oldtimer    extends Auto
                  implements Sammlerstueck
```

Erbt alle Elemente von 'Fahrzeug' und Sammlerstueck'

```
class Flugzeug    implements Fahrzeug,
                             Sammlerstueck
```


Beispiel: volle Form

(nach: Krüger: Go To Java 2)

```
interface Fahrzeug   {
   public int kapazitaet();
   public double verbrauch();
}
```

```
interface Sammlerstueck   {
   public double wert();
   public boolean ausgestellt();
}
```

```
class Auto implements Fahrzeug {
  public String name;
  private int anzSitze;
  private double spritVerbrauch;
  public int kapazitaet()
        { return anzSitze; }
  public double verbrauch()
        { return spritVerbrauch;}
}
```

```
class Oldtimer extends Auto
      implements Sammlerstueck {
  private double sammlerWert;
  private boolean ausst;
  public double wert ()
        { return sammlerWert; }
  public ausgestellt ()
        { return ausst; }
}
```

```
class Flugzeug
        implements Fahrzeug, Sammlerstueck ...
```


Interface: Welche Komponentenart modellierbar?

Vgl. III.4 Komponentenarten

Nicht alle Komponentenarten erlaubt

viele semantisch sehr unterschiedliche Komponenten a r t e n

imperativ

Funktions-sammlung

Konstanten-sammlung

andere

objektorientiert

Daten-abstraktion

Daten-sammlung

ADT

ADT mit Klassenelementen

Interface: Welche Komponentenart modellierbar?

Vgl. III.4 Komponentenarten

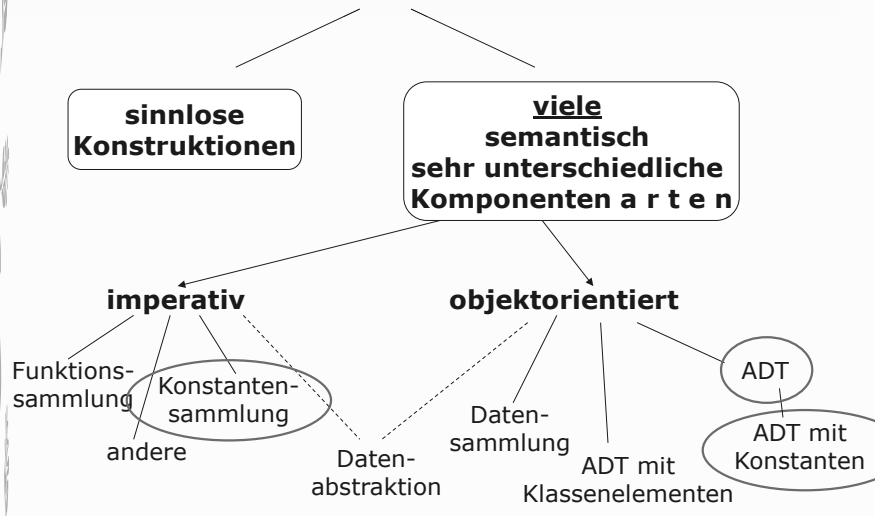
```
interface Fahrzeug   {
   public int kapazitaet();
   public double verbrauch();
   static final int anzahl = 100000;
}
```

Festlegung (gilt auch implizit für Interface):
Methoden: public, abstract, non-static
Variablen: public, static, final

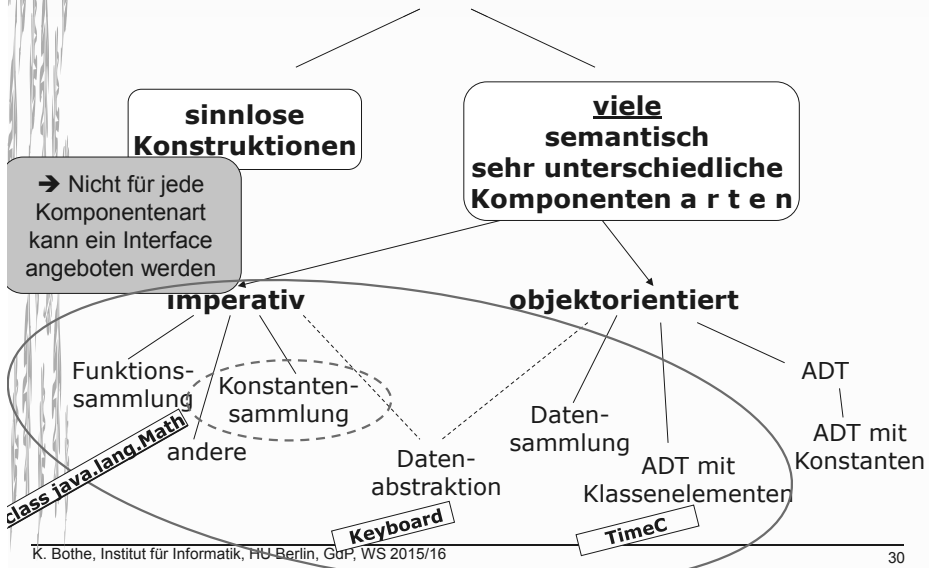
→ Instanzmethoden + (Klassen-)Konstanten

→ ADT, ggf. mit Konstanten

Interface: modellierbare Komponentenarten



Interface: nicht modellierbare Komponentenarten



Java-Interface: Methoden mit Interface-Parametern

```
interface TimeI   {
   public void addMinutes (int m);
   public void printTime ();
   ...
   public boolean before (TimeI t);
}

class Time implements TimeI  {
   private int hour ...
   public boolean before ( TimeI t1 )   {
        Time t = (Time) t1;
        return ((hour < t.hour)  || ...)
   }
```

Dieselbe Signatur: Interface-Typ auch in Klasse

Typ-Transformation: Parameter hat Interface-Typ
allgemeiner Typ (Interface) → spezieller Typ
(nachfolgend wird ein Time-Objekt vorausgesetzt: t.hour)

Cast-Operation kann Laufzeitfehler erzeugen:
aktueller Parameter einer anderen implementierenden Klasse übergeben

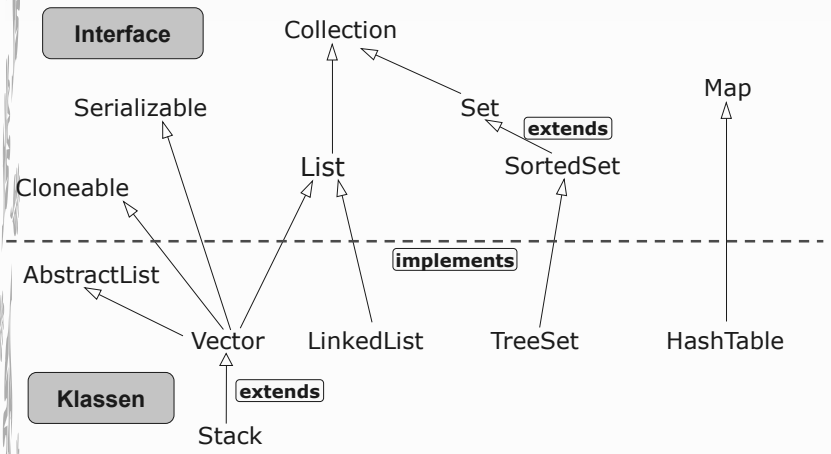
ClassCastException

Interface: wichtige Anwendungen

Interface: Anwendungsfälle

- **E i n Interface eines ADT**
  - z w e i (mehrere) Implementationen:
  - UmkehrungNU.java: beschränkt und unbeschränkt
- **Interface für Datenabstraktionen:**
  - KeyboardIApp.java
- **Import gemeinsamer Konstanten von Klassen**
- **Algorithmen mit 'Funktionsparametern':**
  - Druck.java: Druck beliebiger Funktionen
- **Java-API mit Vielzahl von Interfaces:**
  List – Set – Collection ...

Java-API: Vielzahl von Interfaces



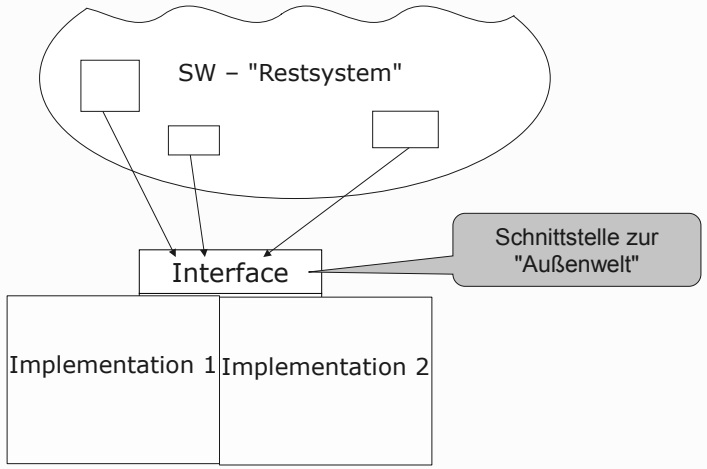

```
public class Vector
extends AbstractList
implements List, Cloneable, Serializable
```

Mehrfachvererbung durch Vielzahl von Interfaces im Java-API

Interface: Anwendungsfälle

- **E i n Interface eines ADT**
  - z w e i (mehrere) Implementationen:
  - UmkehrungNU.java: beschränkt und unbeschränkt
- **Interface für Datenabstraktionen:**
  - KeyboardIApp.java
- **Import gemeinsamer Konstanten von Klassen**
- **Algorithmen mit 'Funktionsparametern':**
  - Druck.java: Druck beliebiger Funktionen
- **Java-API: List – Set – Collection ...**

Bei alleiniger Nutzung des Interface kann die Implementation leicht ausgewechselt werden



E i n Interface eines ADT – zwei (mehrere) Implementationen:

```
interface Stack   {
   public boolean isempty();
   public void push(char x);
   public char top();
   pulic void pop();
}
```

UmkehrungNU.java

Vorteil:

Sehr gute Modifizierbarkeit, falls die Anwendung nur mit den Informationen aus dem Interface arbeitet

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

beschränkte Größe

```
class StackU implements Stack  {
   private class Zelle {...}
   private Zelle top;
   ...
}
```

unbeschränkt

Anwendung: nur Interface bekannt + Konstruktor

```
interface Stack   {
   public boolean isempty();
   public void push(char x);
   public char top();
   pulic void pop();
}
```

UmkehrungNU.java

Besonderheit von UmkehrungNU.java: Erst dynamisch wird die implementierende Klasse ausgewählt → dynamische Bindung von Methoden zur Laufzeit

Anwendung:

```
Stack s;
s = new StackN(n);
      oder
s = new StackU();

while (!s.isempty()) {
    System.out.print(s.top());
    s.pop();
}
```

Auswechseln der Implementation: Nur der Konstruktor ist auszuwechseln

Methoden aus dem Interface

Algorithmus korrekt für beliebige Implementationen:
- Begrenzte Stacks
- Unbegrenzte Stacks

Interface: Anwendungsfälle

- **E i n Interface eines ADT**
  - z w e i (mehrere) Implementationen:
  - UmkehrungNU.java: beschränkt und unbeschränkt
- **Interface für Datenabstraktionen:**
  - KeyboardIApp.java
- **Import gemeinsamer Konstanten von Klassen**
- **Algorithmen mit 'Funktionsparametern':**
  - Druck.java: Druck beliebiger Funktionen
- **Java-API: List – Set – Collection ...**

Klasse 'Keyboard`: eine Datenabstraktion

```
import java.io.*;

class Keyboard {

// Author: M. Dennis Mickunas,
// June 9, 1997 Primitive Keyboard
// input of integers, reals,
// strings, and characters.

static boolean iseof = false;
static char c;
static int i;
static double d;
static String s;

/* WARNING: THE BUFFER VALUE IS SET
TO 1 HERE TO OVERCOME ** A KNOWN BUG
IN WIN95 (WITH JDK 1.1.3 ONWARDS)*/

static
   = ne

InputSt
```

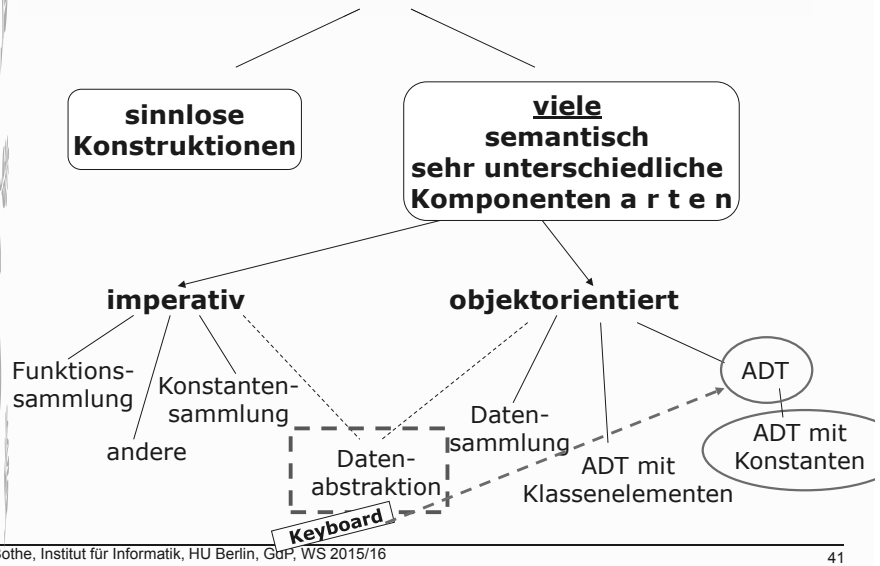
```
public static int readInt () {
   if (iseof) return 0;
   System.out.flush();
   try {
         s = input.readLine();
   }
   catch (IOException e) {
         System.exit(-1);
   }
   if (s==null) {
         iseof=true;
         return 0;
   }
   i = new
      Integer(s.trim()).intValue();
   return i;
}

public static char readChar () {
   if (iseof) return (char)0;
   System.out.flush();
   ...
```

Klasse realisiert Abstraktionsprinzip nicht: Kein Interface für Nutzer erkennbar

Realisierung als Interface möglich?

Interface: modellierbare Komponentenart



Interface für Datenabstraktionen

Interface für den Nutzer

Datenabstraktion nicht durch 'interface' realisierbar → ADT

KeyboardIAppl.java

```
interface KeyboardI   {
   public int readInt();
   public char readChar();
   public double readDouble();
   public String readString();
   public boolean eof();
}
```

jetzt: Instanzmethoden → ADT

Unwichtige Implementations-details

```
class Keyboard implements KeyboardI  {
   // 110 Zeilen Implementation
}
```

Anwendung: technisch andere Realisierung → Objekt erzeugt

```
Keyboard kb = new Keyboard();

jn = kb.readChar();
n = kb.readInt();
```

vgl. `Keyboard.readChar()`

Interface: Anwendungsfälle

- **E i n Interface eines ADT**
  - z w e i (mehrere) Implementationen:
  - UmkehrungNU.java: beschränkt und unbeschränkt
- **Interface für Datenabstraktionen:**
  - KeyboardIApp.java
- **Import gemeinsamer Konstanten von Klassen**
- **Algorithmen mit 'Funktionsparametern':**
  - Druck.java: Druck beliebiger Funktionen
- **Java-API: List – Set – Collection ...**

Import gemeinsamer Konstanten

Konstanten: Steuerzeichen

```
interface Steuerzeichen   {
   public static final char BS = '\b',
         HT = '\t', LF = '\n',
         FF = '\f', CR = '\r';
}
```

'implements' – ein adäquater Begriff?

Konstanten verfügbar in Ausgabe1 / Ausgabe2

```
class Ausgabe1 implements Steuerzeichen  {
   // Methoden der Ausgabe 1
}
```

```
class Ausgabe2 implements Steuerzeichen  {
   // Methoden der Ausgabe 2
}
```

dieselbe Technik: include-Files in C / C++

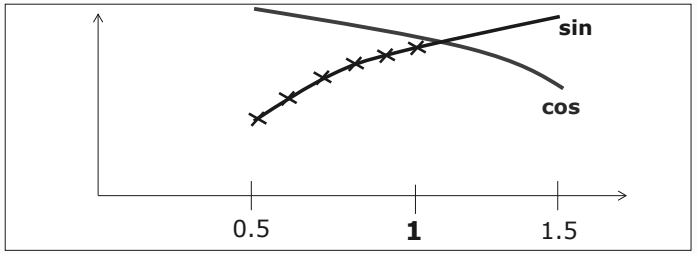
Interface: Anwendungsfälle

- **E i n Interface eines ADT**
  - z w e i (mehrere) Implementationen:
  - UmkehrungNU.java: beschränkt und unbeschränkt
- **Interface für Datenabstraktionen:**
  - KeyboardIApp.java
- **Import gemeinsamer Konstanten von Klassen**
- **Algorithmen mit 'Funktionsparametern':**
  - Druck.java: Druck beliebiger Funktionen
- **Java-API: List – Set – Collection ...**

Algorithmen mit 'Funktionsparametern'

Problem:
- Algorithmen: arbeiten mit (beliebigen) mathematischen Funktionen
- Funktion ist Parameter des Algorithmus

```
druckeKurve(0.5, 1.5, 0.1, sin);
druckeKurve(0.5, 1.5, 0.1, cos);
```
Schrittweite



Parameterarten: Werte-, Referenz-, Funktionsparameter (in Java nur indirekt)

Algorithmen mit 'Funktionsparametern´: Umsetzung in Programmiersprachen

- **Pascal:** Funktionsparamter (spezielle Parameterart)

```
function druckeKurve (x0: real; x1: real;
       delta: real;
       function F(x: real): real )  ...
```

Aufruf:

```
druckeKurve(0.5, 1.5, 0.1, sin);
druckeKurve(0.5, 1.5, 0.1, cos);
```

- **C, C++:** Zeiger auf Funktionen als Parameter
- **Java:** Interface oder abstrakte Klassen *)

*) Funktionsparameter existieren nicht
→ Parameter in Java: elementare Typen oder Objekte
→ Funktion in Interface "verpacken" (Wrapper für Funktion)

Beispiel: Algorithmen mit 'Funktionsparametern' (1)

Druck.java

```
interface Function   {
   double apply (double x);
}
```

allgemeinstes Schema einer einstelligen reellwertigen Funktion

```
class SinFunction implements Function   {
   public double apply (double x)   {
       return Math.sin(x);
   }
}
```

2 Spezialisierungen: Sinus, Cosinus

```
class CosFunction implements Function   {
   public double apply (double x)   {
       return Math.cos(x);
   }
}
```

Beispiel: Algorithmen mit 'Funktionsparametern' (2)

Druck.java

```
interface Function   {
   double apply (double x);
}
```

Nutzer: arbeitet nur mit allgemeinem Schema (interface)

```
static void druckeKurve (double x0, double x1,
                         double delta, Function f)   {
   System.out.print( f.apply(x0) ); ...
}
```

Anwendung: konkrete Funktionen 'sin', 'cos'

```
public static void main (...)   {
   SinFunction sinus   = new SinFunction();
   CosFunction cosinus = new CosFunction();
   druckeKurve(0, Math.PI/2, 0.1, sinus);
   druckeKurve(0, Math.PI/2, 0.1, cosinus);
}
```

Technisch anders als in Pascal, C, ...:
Hier wird Objekt übergeben, das passende Methode umfasst (Wrapper)

Druck aller Funktionswerte von x0 bis x1

(Verbesserung von Druck.java)

```
static void druckeKurve (double x0,
   double x1, double delta, Function f )   {

   for (int x = x0; x <= x1; x = x + delta)
      System.out.print( f.apply(x) );
      ...
}
```

Bisher in Druck.java:
Nur Ausgabe von 2 Funktionswerten
– keine graphische Veranschaulichung