Übungen zu

Middleware

Wintersemester 2003/04

Teil I: Java

A Übersicht

A.1 Organisatorisches

- Tafelübungen
  - Dienstag 12.30 - 14.00 Uhr Raum 2.038
  - Mittwoch 12.30 - 14.00 Uhr Raum 0.031
- Rechnerübungen
  - Mittwoch 14.00 - 16.00 Uhr Raum 00.159
  - Donnerstag 14.00 - 16.00 Uhr Raum 00.156
  - Rechnerraum ist reserviert
  - Betreuung (nur) bei Bedarf
- Ansprechpartner
  - Meik Felser Raum 0.042 felser@informatik.uni-erlangen.de
  - Andreas Weißel Raum 0.045 weissel@informatik.uni-erlangen.de

A.1 Organisatorisches

- Projektverzeichnis
  - `/proj/i4mw/`*<loginname>*
- Übungsaufgaben müssen abgegeben werden
- Abgabe mittels Abgabeprogramm
  - `/proj/i4mw/pub/abgabe` *`aufgabe1`*
- Übungsaufgaben bauen aufeinander auf

A.2 Inhalt

- Teil I: Java
  - automatische Tests mit JUnit
  - Fehlerbehandlung, Ein-/Ausgabe, Threads
  - Sockets
  - Serialization, Classloader
  - RMI
- Teil II: CORBA
  - IDL
  - CORBA Programmieren in Java
- Teil III: JINI
- Teil IV: .NET
  - C#
  - ".NET Remoting"

B Überblick über die 1. Übung

- Java Überblick
- OO Grundlagen und Konzepte mit Java
  - ◆ Abstraktion, Kapselung, Objekte
  - ◆ Klassen, Methoden, Variablen, Konstruktoren
  - ◆ Referenzen, Aufrufsemantik, Gleichheit, Identität
  - ◆ Vererbung (Ersetzungsprinzip), Überladen, Überschreiben
  - ◆ dynamisches Binden, Typenermittlung
  - ◆ abstrakte Klassen
  - ◆ Interfaces
  - ◆ Modularität: Packages & Sichtbarkeitsattribute
  - ◆ statische Elemente
  - ◆ Konstanten (final Methoden, final Klassen)
  - ◆ innere Klasssen

B Überblick über die 1. Übung

- Ausgewählte Kapitel des Java Laufzeitsystems
  - ◆ Fehlerbehandlung (Exceptions)
  - ◆ Ein-/Ausgabesystem (Streams)

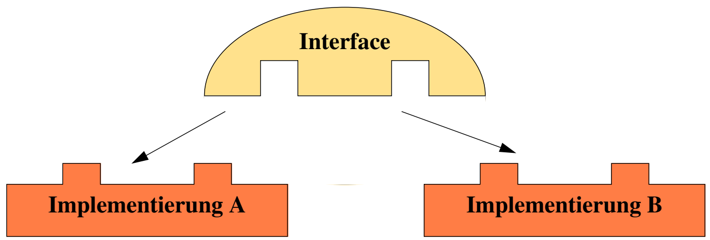
B.1 OO-Grundlagen mit Java

- Fundamentale Konzepte der objektorientierten Programmierung:

Abstraktion
und
Kapselung

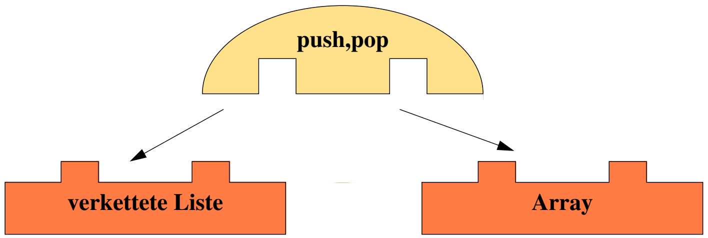
1 Abstraktion

- Trennung von:
  - ◆ Schnittstelle (interface): Was kann getan werden?
  - ◆ Implementierung: Wie wird es gemacht?



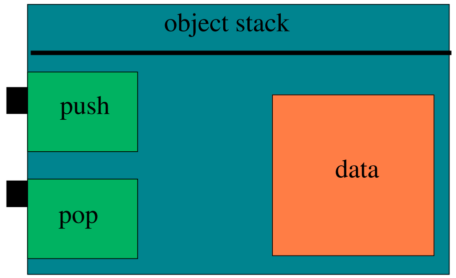
1 Abstraktion

- Beispiel: stack
  - Interface: push, pop
  - Implementierung: verkettete Liste, Array



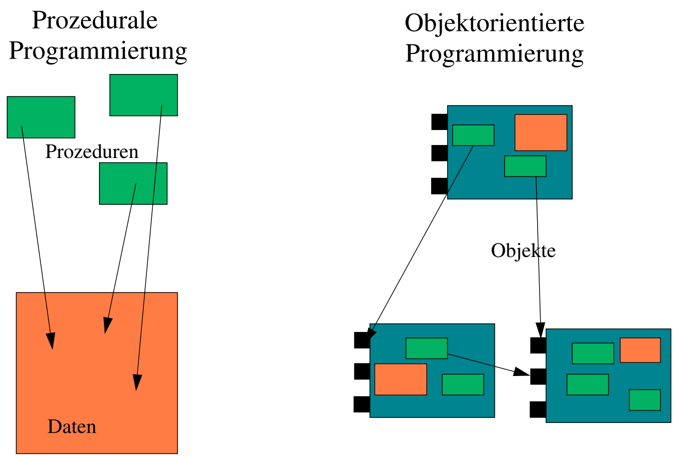
2 Kapselung

- Objekte: Gekapselte Datenstruktur, bestehend aus:
  - Daten (Instanzvariablen, Attribute)
  - Methoden (Operationen)



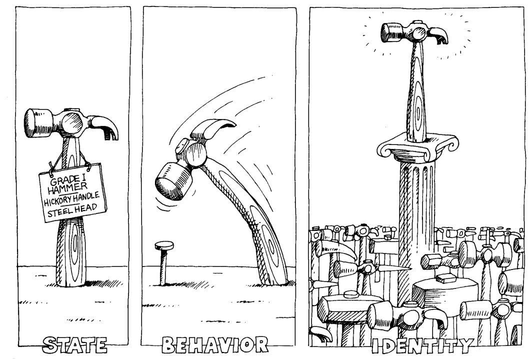
- Kapselung unterstützt die Bildung von Abstraktionen.

2 Kapselung



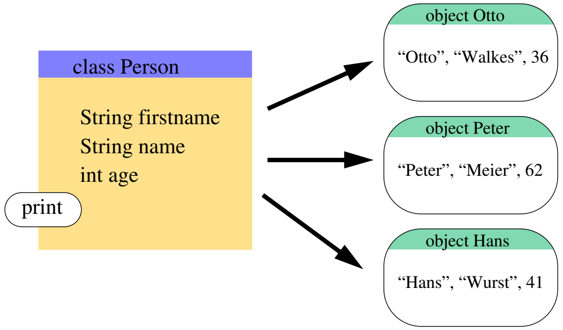
B.2 Objekte

1 Eigenschaften von Objekten



2 Klassen

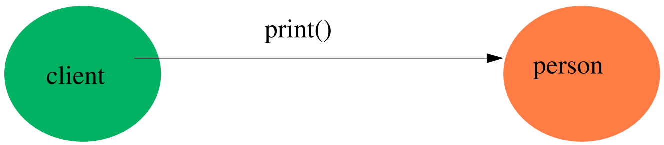
- Objekte sind *Instanzen* einer Klasse.
- Die Klasse bestimmt die interne Struktur und die Schnittstelle eines Objekts.
- Beispiel:



3 Nachrichten / Methoden

- Objekte kommunizieren mit Hilfe von Nachrichten.
  - Jedes Objekt legt sein eigenes Verhalten selbst fest.
  - Die Objektsemantik ist nicht über das ganze Programm verteilt.
- Nachricht = Methodenaufruf an einem Objekt

```
person.print()
```



- objektorientiertes Programm: mehrere Objekte kommunizieren miteinander, um eine bestimmte Aufgabe zu erfüllen.

4 Überladen

- Methoden mit unterschiedlichen Parametern können den gleichen Namen haben.
- Beispiel:

```
class Date {
    ...
    void print(PrintStream stream) { stream.println(...); }
    void print() { print(System.out); }
}
```

- Hinweis: Überladen funktioniert nur mit Parametern nicht mit dem Typ des Rückgabewerts:

```
class Income {
    ...
    int computeIncome() { ... }
    float computeIncome() { ... } // Error !!
}
```


5 Objekt-Initialisierung

- Erzeugen eines Objekts bedeutet Reservierung von Speicher.
- Dieser Speicher muss initialisiert werden.
- Eine Möglichkeit:
  - Explizites Aufrufen einer Initialisierungsmethode.
  - Nachteil: fehleranfällig.

6 Konstruktoren

- Konstruktoren dienen der Initialisierung des Objekts.
- Name des Konstruktors = Name der Klasse.
- Der Konstruktor wird automatisch nach der Objekterzeugung aufgerufen.

6 Konstruktoren (2)

- Mehrere Konstrukturen sind möglich
- Aufruf eines anderen Konstruktors mit `this(...)`:

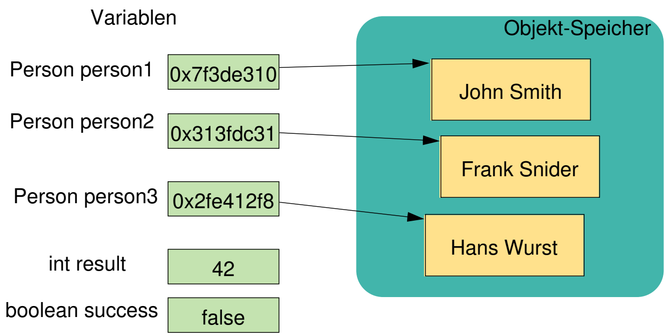
```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    Person(String name) {
        this(name,18);
    }
    ...
}
```


7 Objekte und Referenzen

- Java-Variablen bezeichnen keine Objekte, sondern Referenzen.

```
Person p;      // Deklaration einer Referenz auf ein Objekt
               // der Klasse Person
p.print();     // Fehler: Methodenaufruf an einer null-
```



8 Zuweisungen

- = weist einer Variable eine Referenz zu
- == vergleicht zwei Referenzen
- Einer Variable eines primitiven Datentypes kann keine Referenz zugewiesen werden.
- Einer Variable, welche eine Objektreferenz ist, kann niemals der Wert eines primitiven Datentypes zugewiesen werden.
- Beispiel:

```
Person p;      // Deklaration einer Referenz-Variablen
int i=42;    // Deklaration und Initialisierung einer
               // Variable eines primitiven Datentypes
p = i;        // Fehler: Zuweisung zwischen Referenz und
               // primitiven Datentyp
```


9 Aufrufsemantik von Methoden

- Objekt-Parameter werden als Referenz übergeben.
- Primitive Datentypen (int, float, etc.) werden als Wert übergeben.
- Beispiel:

```
void meth(int a, Person k) {
    a = 5;              // a: passed by value
    k.setAge(25);       // k: passed by reference
}
```

10 Gleichheit und Identität

- Unterschied zwischen *gleichen Objekten* und *identischen Objekten*:

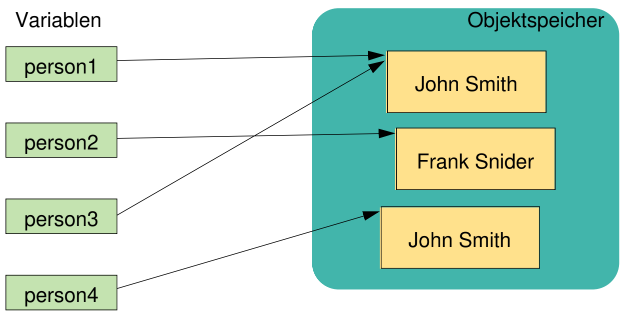
```
class Date {
  int day, month, year;
  Date(int day, int month, int year) {
    this.day = day; this.month = month; this.year = year;
  }
}
...
Date d = new Date(1,3,98);
Date d1 = new Date(1,3,98);
Date d2 = d;
```

◆ d und d1 sind gleich

◆ d und d2 sind identisch

11 Identität und Referenzen

- Identität: gleiche Referenz
- Gleichheit: Inhalt der referenzierten Objekte ist gleich



- Welche Personen sind identisch, welche gleich?

12 Testen von Gleichheit und Identität

- Identität kann mit dem Operator == getestet werden:

```
if (d == d1) { ... }
```

- Gleichheit kann mit der Methode **`equals`** getestet werden:

```
if (d.equals(d1)) { ... }
```

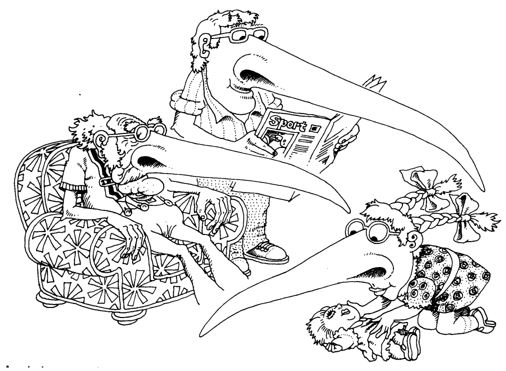
- Die Methode **`equals`** muss selbst implementiert werden:

```
class Date {
   ...
  public boolean equals(Object o) {
    if (! (o instanceof Date)) return false;
    Date d = (Date)o;
   return d.day == day && d.month == month && d.year == year;
  }
}
```


B.3 Vererbung

- Definition von Objekten durch Verweise auf andere Objekte.
  ◆ Beispiel:
    - Definition eines Tieres: Ein Ding welches atmet und isst.
    - Definition einer Kuh: Ein Tier, dass “muuuu” macht und Milch gibt.
    - Kuh *erbt* die Eigenschaften “atmen” und “essen” von Tier.

1 Vererbung in der echten Welt



2 Vererbung in Java

- Vererbung: Definition eines neuen Objekts auf der Basis einer spezialisierten Definition eines existierenden Objekts.
- Neue Klassen können von existierenden Klassen *abgeleitet* werden.
- Beispiel: Ein Kunde ist eine Person.

```
class Customer extends Person {
    int number;
    ...
}
```

- **Customer** ist eine *Unterklasse (subclass)* von **Person**
- **Person** ist die *Oberklasse (superclass)* von **Customer**.

3 Unterklassen

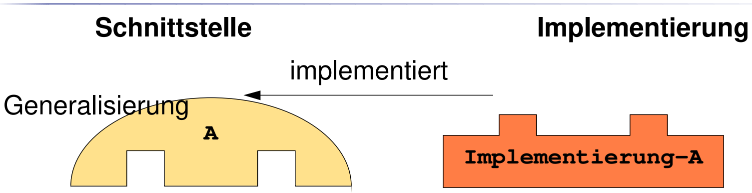
- Unterklassen erben
  - ◆ Zustand (Instanzvariablen) und
  - ◆ Verhalten (Methoden) von der Oberklasse.
- Unterklassen können:
  - ◆ neue Instanzvariablen einführen
  - ◆ neue Methoden einführen
  - ◆ geerbte Instanzvariablen verdecken (vorsicht!)
  - ◆ geerbte Methoden überschreiben

4 Das Ersetzungsprinzip

- Wenn ein Objekt der Oberklasse erwartet wird, kann immer auch ein Objekt einer Unterklasse verwendet werden.
  - ◆ wichtigster Typ des Polymorphismus
- Vererbung ist Spezialisierung ("ist ein" Relation)
- alles was für die Generalisierung gilt muss auch auf die Spezialisierung zutreffen.
  - ◆ Die Spezialisierung muss alle Anforderungen der Generalisierung erfüllen.

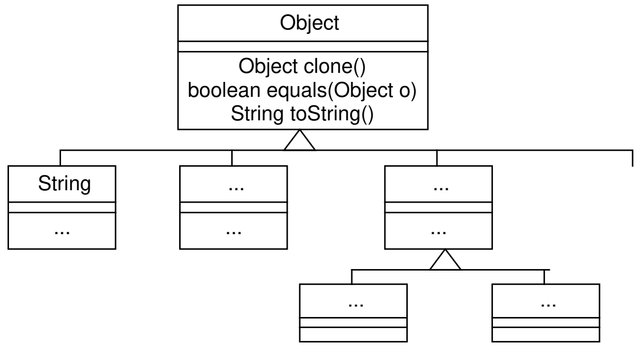
→ Abstraktion

5 Vererbung ist Spezialisierung



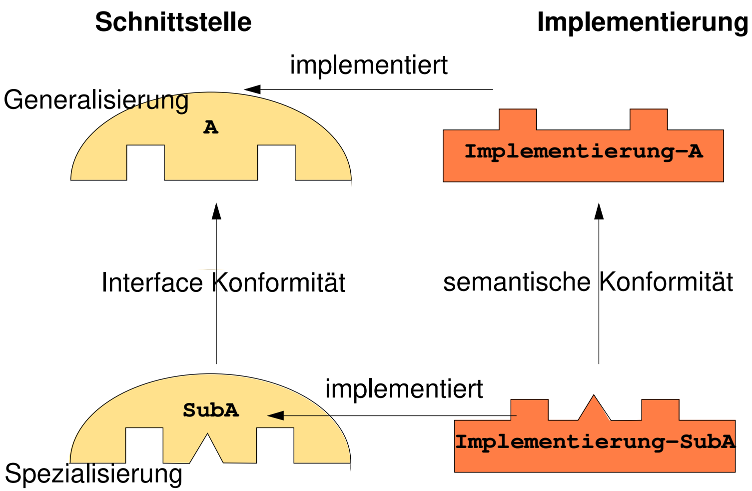
6 Vererbung in Java

- Klassen mit einfacher Vererbung
- Baum Hierarchie mit der Klasse `Object` als Basisklasse aller anderen Klassen



- primitive Typen (`int`,`float`,...) sind außerhalb des Klassenbaumes

B.3 Vererbung ist Spezialisierung



7 Die Klasse Object

- Die Klasse `Object`: Basisklasse aller anderen Klassen

```
class Person extends Object {... }
```

  - `extends Object` kann wegfallen

```
class Person {... }
```

- Stellt Basisfunktionalität zur Verfügung, zum Beispiel:
  - `boolean equals(Object o)` // Test auf Gleichheit
    - Standardimplementierung vergleicht die Referenzen
    - Jede Klasse sollte eine eigene Implementierung bereitstellen
  - `String toString()` // Stringdarstellung eines Objekts
    - Standardimplementierung: Klassenname und Objekt ID
    - Jede Klasse sollte eine eigene Implementierung bereitstellen

8 Überschreiben

- Unterklassen können für geerbte Methoden eine neue Implementierung bereitstellen.
- Die neue Implementierung *überschreibt* die geerbte Implementierung:

```
class Person {
    String name;
    ...
    void print() {
        System.out.println("Person: " + name);
    }
}
class Customer extends Person {
    int number;
    ...
    void print() {
        System.out.println("Person: " + name);
        System.out.println("Customer number: " + number);
    }
}
```


8 Überschreiben (2)

- Die Implementierung der Oberklasse kann mit **`super.`*`method`*`()`** aufgerufen werden.
- Beispiel:

```
class Customer extends Person {
    int number;
    ...
    void print() {
        super.print();
        System.out.println("Customer number: " + number);
    }
}
```


9 Überladen vs. Überschreiben

- Um eine Methode zu überschreiben müssen die Typen der Parameter und des Rückwerts exakt übereinstimmen, ansonsten wird die Methode überladen.
- Um dem Ersetzungsprinzip gerecht zu werden würde es ausreichen, wenn:
  - die Typen der Parameter von einer Oberklasse der ursprünglichen Parameter sind
  - der Typ des Rückgabewertes von einer abgeleiteten Klasse des ursprünglichen Rückgabetyps ist.
- Java unterstützt das jedoch nicht!!!

9 Überladen vs. Überschreiben (2)

- häufiger Fehler:

```
class Object {
    boolean equals(Object o) { ... }
    ...
}

class Customer {
    int number;
    boolean equals(Customer c) { return number == c.number;}
    ...
}
```

`equals` *von Object wird nicht überschrieben*

10 Dynamisches Binden

- Die Methoden werden erst bei einem Aufruf gebunden.
- *dynamischer Typ*: Typ/Klasse des referenzieren Objekts
  (die Klasse, die bei **new** verwendet wurde)
- *statischer Typ*: Typ der Referenz
- Durch den statischen Typ wird festgelegt, welche Methoden aufgerufen werden können.
- Der dynamische Typ legt fest, welche Methode verwendet wird:

```
Customer c = new Customer("Max", 1234);
Person p = c;      // dynamischer Typ von p ist Customer,
                   // statischer Typ ist Person

c.print();
p.print(); // obwohl die Referenz p den Typ Person hat, wird
                 // die print() Methode von Customer verwendet
```


11 Sichtbarkeit und Vererbung

- Die Sichtbarkeit von Methoden darf in Unterklassen nicht eingeschränkt werden (Ersetzbarkeit!):.

```
class Person {
    public String getName() { ... }
}
class Customer extends Person {
    private String getName() { ... }
}
         Error
```


12 Konstruktoren und Vererbung

- Konstruktoren werden **nicht** vererbt
- Aufrufen eines Konstruktors der Oberklasse mittels **super(...)**
- **super(...)** muss die erste Anweisung in einem Konstruktor sein
- Falls die erste Anweisung nicht **super(...)** ist, so fügt der Compiler automatisch eine **super()** Anweisung ein.
- Standardkonstruktor:
  - wird vom Compiler erzeugt, falls *kein* Konstruktor definiert wird
  - enthält eine **super()** Anweisung

12 Konstruktoren und Vererbung (2)

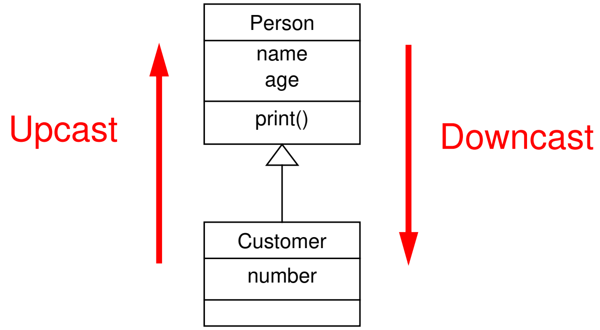
- Beispiel:

```
class Customer extends Person {
    int number;
    Customer(String name, int number) {
        super(name);
        this.number = number;
    }
    ...
}
```

13 Typenkonvertierung: Upcast

- Typenkonvertierung von einer Unterklassen zur Oberklasse (*upcast*) erfolgt automatisch:

```
Person p = new Customer(...);
```



13 Typenkonvertierung: Downcast

- Typenkonvertierung von der Oberklasse zu einer Unterklasse *(downcast*) explizit mittels Cast-Opertor:

```
Customer c = new Customer(...)
Person p = c;                 // implizite Typenkonvertierung
Customer c2 = (Customer) p;   // explizite Typenkonvertierung
```

- Wenn das Objekt und die Variable nicht typkonform sind, wird eine **`ClassCastException`** generiert:

```
Customer c = new Customer(...)
Person p = c;                 // implizite Typenkonvertierung
Employee e = (Employee) p;
                 // erzeugt eine ClassCastException zur Laufzeit
```


14 Typ Ermittlung

- **`instanceof`** Operator:

```
Customer c = new Customer(...)
Person person = c;     // upcast
if (person instanceof Employee) {
    Employee employee = (Employee) person;
    ...
} else if (person instanceof Customer) {
    Customer customer = (Customer) person;
    ...
}
```

- **`instanceof`** mit der Oberklasse des dynamischen Typs ist ebenfalls **`true`**:

```
person instanceof Person
```


15 Die Klasse Class

- Die Klasse **`Class`**: Die Klasse aller Klassen.
- Ein **`Class`** Objekt repräsentiert eine Klasse oder ein Interface.
- Mit Hilfe eines **`Class`** Objekts können neue Instanzen erzeugt werden:

```
Customer c = new Customer();
...
Class aClass = c.getClass();
System.out.println("Class of c is:"+aClass);

Object o = aClass.newInstance();  // ein neues Customer Objekt

Person p = (Person) o;
```

- Ein **`Class`** Objekt kann aus dem Namen einer Klasse generiert werden:

```
Class aClass = Class.forName("Employee");
```

B.4 Packages

- Klassen lassen sich in Pakete (packages) zusammenfassen.
- Paket = Programm-Modul
  - ◆ mit eindeutigem Namen (z.B. **`java.lang`** oder **`java.awt.image`**)
  - ◆ enthält eine oder mehreren Klassen
- Pakete partitionieren den Namensraum.

1 Schlüsselwort package

- Packages werden mit **`package`** deklariert:

```
package test;
public class TestClass ...
```

- **`package`** muss die erste Anweisung in einer Datei sein.
- Hierarchien von Packages sind möglich:

```
package test.unittest1;
```


2 Schlüsselwort: import

- Klassen anderer Packages können mit **`import`** verwendet werden:

```
import java.util.*; // use all classes from package java.util
import java.io.File;// use class File from package java.io
```

- Bei der Suche von Klassen wird der Package-Name als Verzeichnis verwendet.
- Beispiel:
  - ◆ Package **`bank`** mit Klasse **`Customer`**
  - ◆ kann verwendet werden mit **`import bank.Customer`**
  - ◆ Bytecode-Datei wird gesucht als **`bank/Customer.class`**
- Package **`java.lang.*`** wird automatisch importiert.
- Zugriff auf Klassen ohne import: durch Verwendung des vollständigen Klassennamens, inklusive Package.

2 Schlüsselwort: import - Beispiel

Datei
`editor/shapes/X.java`

```
package editor.shapes;
public class X {
    public void test() {
        System.out.println("X");
    }
}
```

Datei
`editor/filters/Y.java`

```
package editor.filters;
import editor.shapes.*;
class Y {
    X x;
    void test1() { x.test();}
}
```

Datei
`editor/filters/Z.java`

```
package editor.filters;
class Z {
    editor.shapes.X x;
    void test1() { x.test();}
}
```

3 Packages und der CLASSPATH

- Klassen werden mit Hilfe der Umgebungsvariable CLASSPATH gesucht.
- Beispiel:
  - ◆ Package **`bank`** mit Klasse **`Customer`**
  - ◆ wird verwenden mit **`import bank.Customer;`**
  - ◆ Compiler und Interpreter suchen die Bytecode-Datei: **`bank/Customer.class`**
  - ◆ **`CLASSPATH`** enthält **`/proj/test:/tmp`**
  - ◆ gesucht wird nach:
    - **`/proj/test/bank/Customer.class`**
    - **`/tmp/bank/Customer.class`**
  - ◆ beim kompilieren kann man das Zielverzeichnis angeben:
    - **`javac -d /proj/test Customer.java`**

B.5 Sichtbarkeitsattribute

- *Kapselung* ist eines der Grundprinzipien objektorientierter Programmierung.
- Kapselung wird zum verstecken unnötiger Information verwendet (*information hiding*).

4 Standard Java Pakete

- **`java.lang`**: fundamentale Java Klassen (Thread, String,...)
- **`java.io`**: Ein- / Ausgabe Unterstützung (Files, Streams,...)
- **`java.net`**: Netzwerk Unterstützung (Sockets, URLs, ...)
- **`java.awt`**: GUI Unterstützung (Abstract Windowing Toolkit)
- **`java.applet`**: Applet Unterstützung
- **`java.util`**: Hilfsklassen (Random) und Datenstrukturen (Vector, Stack)
- **`java.rmi`**: Remote Method Invocation
- **`java.security`**: kryptographische Unterstützung
- Nähere Informationen in der API Documentation:

`http://www4/Services/Doc/Java/jdk-1.4/docs/api/index.html`

1 Sichtbarkeitsattribute - Klassen

- Eine Klasse kann öffentlich (**`public`**) oder nicht öffentlich sein.

```
public class X { ... }

class X { ... }
```

- **`public`** Klassen sind außerhalb des Package verfügbar.
- Klassen ohne **`public`**-Deklaration (private Klassen) sind nur innerhalb desselben Package sichtbar.
- Eine public-Klasse muss in einer eigenen Datei deklariert werden.
  Dateiname := Klassenname + ".java"
  Beispiel: Klasse **`X`** muss in der Datei **`X.java`** definiert werden.

2 Sichtbarkeitsattribute - Klassenelemente

- Sichtbarkeitsattribute für Methoden und Variablen:
  - public, default, protected, private
- Wirkung:
  - **`public`**: global sichtbar
  - default: innerhalb des gleichen Packages sichtbar
  - **`protected`**: innerhalb des gleichen Packages und in Unterklassen sichtbar.
  - **`private`**: nur innerhalb der gleichen Klasse sichtbar
- Sichtbarkeitsattribute müssen bei *jeder* Methode bzw. Instanzvariable extra angegeben werden.

3 Kapselung

- ohne Kapselung:

```
class Person {
    public String name;// "name" can be modified/read globally
}
```

- besser:

```
class Person {
    private String name;     // only Person can access "name"
    public String getName() {            // access method
        return name;
    }
}
```


2 Sichtbarkeitsattribute - Klassenelemente (2)

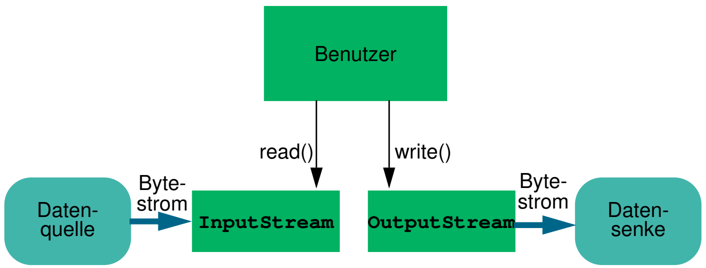
- Übersicht:

	Sichtbarkeitsattribute			
sichtbar in	**`public`**	**`protected`**	default	**`private`**
gleiche Klasse	ja	ja	ja	ja
gleiches Package	ja	ja	ja	nein
Unterklassen	ja	ja	nein	nein
andere Packages	ja	nein	nein	nein

B.6 Das Java Ein-/Ausgabesystem

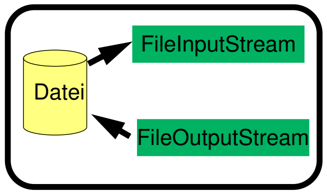
- Grundlegendes Konzept: Ströme (Streams)
  - Byteströme (InputStream/OutputStream)
  - Zeichenströme (Reader/Writer)

1 Bytesröme



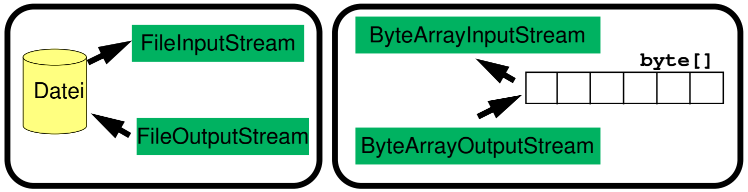
2 Spezialisierungen von Strömen

- Wo kommen die Daten her, wo gehen sie hin?



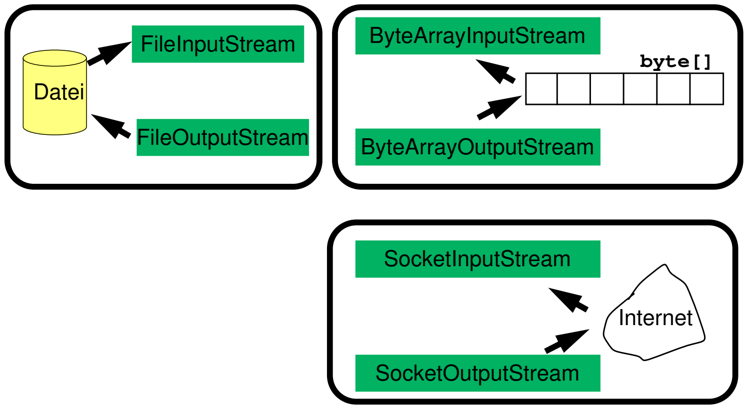
2 Spezialisierungen von Strömen (2)

- Wo kommen die Daten her, wo gehen sie hin?



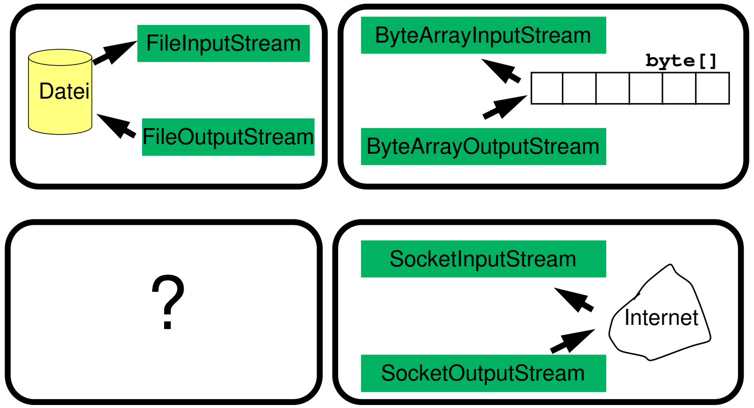
2 Spezialisierungen von Strömen (3)

- Wo kommen die Daten her, wo gehen sie hin?

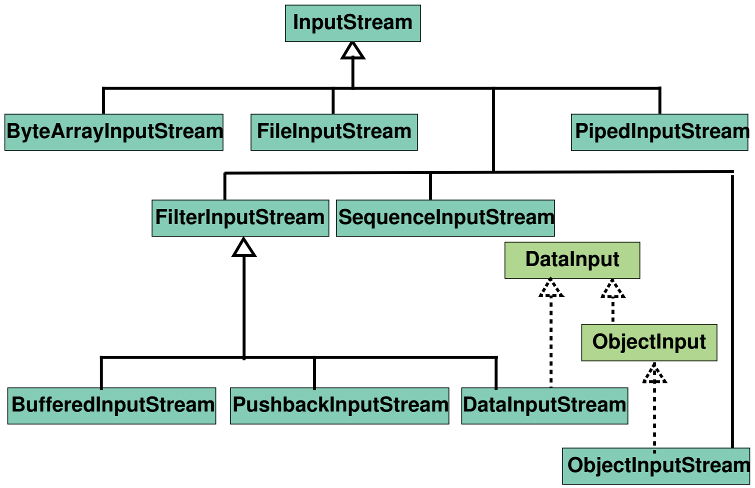


2 Spezialisierungen von Strömen (4)

- Wo kommen die Daten her, wo gehen sie hin?



3 Klassendiagram der Eingabeströme



4 FileInputStream

- Aus einer Datei lesen:

```
import java.io.*;

public class InTest {
  public static void main (String argv[]) throws IOException {
    FileInputStream f = new FileInputStream ("/tmp/test");
    byte buf[] = new byte[4];
    f.read(buf);
  }
}
```


4 FileOutputStream

- In eine Datei schreiben:

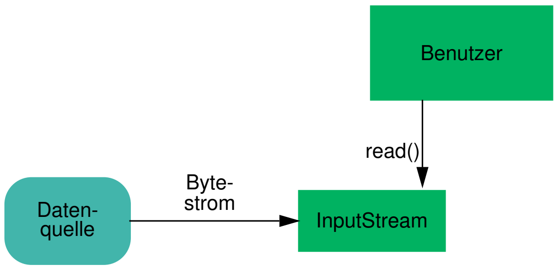
```
import java.io.*;

public class OutTest {
  public static void main (String argv[]) throws IOException {
    FileOutputStream f = new FileOutputStream ("/tmp/test");
    byte buf[] = new byte[4];
    for (byte i=0; i < buf.length; i++) buf[i]=i;
    f.write(buf);
  }
}
```

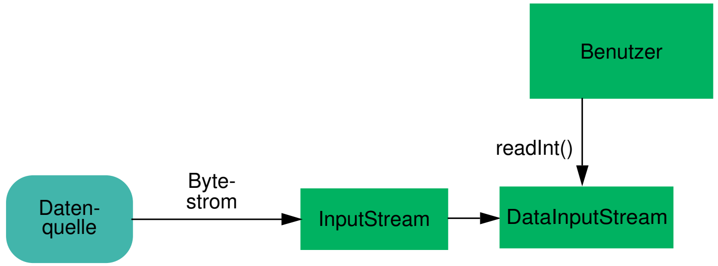
5 Kombinieren von Strömen

- Aus einfachen Strömen “komfortable” Ströme generieren
- Der komfortable Strom umhüllt den einfachen Strom
- ➜ Decorator Design-Pattern

5 Kombinieren von Strömen (2)



5 Kombinieren von Strömen (3)



6 DataInputStream

- **`InputStream`** ist relativ unkomfortabel
- **`DataInputStream`** wird verwendet um eine *binäre Darstellung* der Daten zu lesen (int, float,...)
- Ein **`DataInputStream`** kann aus jedem **`InputStream`** erzeugt werden:

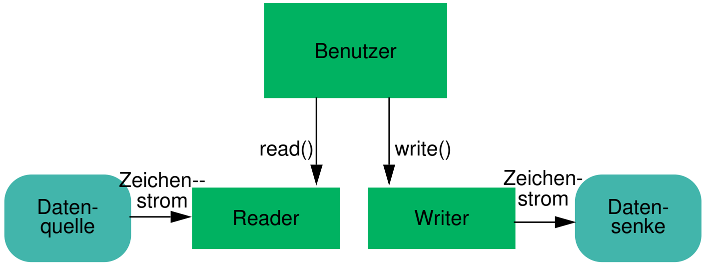
```
InputStream in = new FileInputStream ("/tmp/test");
DataInputStream dataIn = new DataInputStream(in);
float f = dataIn.readFloat();
```

- **`readLine()`** kann verwendet werden um ganze Zeilen zu lesen:

```
for(;;) {
    String s = dataIn.readLine();
    System.out.println(s);
}
```

7 Reader/Writer

- Zeichenströme zur Ein- und Ausgabe (**`Reader, Writer`**)



- Zeichenströme enthalten Unicodezeichen (16 bit)

B.6 Das Java Ein-/Ausgabesystem

8 FileReader

- Wird verwendet um aus einer Datei zu lesen
- Konstruktoren:
  - **`FileReader(String fileName)`**
  - **`FileReader(File file)`**
  - **`FileReader(FileDescriptor fd)`**
- Keine weitere Methoden (nur die von **`InputStreamReader`** geerbten)
- Was ist ein **`InputStreamReader`**?

B.6 Das Java Ein-/Ausgabesystem

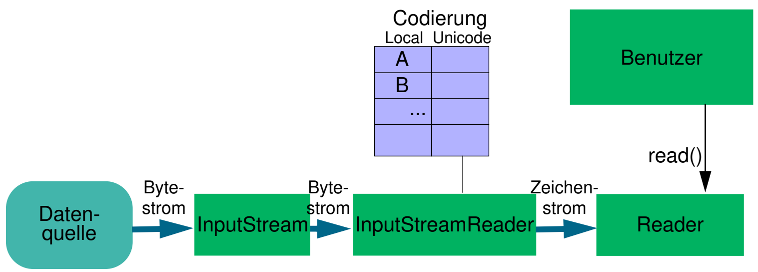
8 Reader

- wichtige Methoden:
  - **`int read()`**
    liest ein Zeichen und gibt es als **`int`** zurück
  - **`int read(char buf[])`**
    liest Zeichen in ein Array. Liefert die Anzahl der gelesenen Zeichen zurück oder -1 falls ein Fehler aufgetreten ist
  - **`int read(char buf[], int offset, int len)`**
    liest **`len`** Zeichen in den Puffer **`buf`**, beginnend ab **`offset`**
  - **`long skip(long n)`**
    überspringt **`n`** Zeichen
  - **`void close()`**
    schließt den Strom

B.6 Das Java Ein-/Ausgabesystem

9 Byte- und Zeichenströme

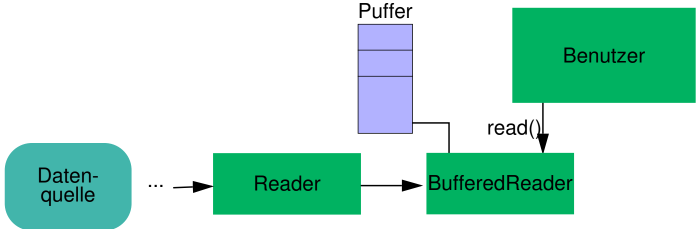
- Umwandeln von Byteströmen in Zeichenströme mit Hilfe einer Codierung



- einige Codierungen: "Basic Latin", "Greek", "Arabic", "Gurmukhi"

10 Gepufferte Ein-/Ausgabe

- Lesen/Schreiben von einzelnen Zeichen kann teuer sein.
- Umrechnung der Zeichenkodierung kann teuer sein.
- Falls möglich **`BufferedReader`**, **`BufferedWriter`** verwenden.
- **`BufferedReader`** kann aus jedem anderen Reader erzeugt werden.
- Wichtige Methoden von **`BufferedWriter`**: **`void flush()`**:
  Leert den Puffer - schreibt den Puffer zum ungepufferten Writer:



10 Gepufferte Ein-/Ausgabe (2)

- **`BufferedReader`** kann ganze Zeilen lesen: **`String readLine()`**

```
BufferedReader in = new BufferedReader(new FileReader("test.txt"));
String line = in.readLine();


BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String line = in.readLine();
```


11 PrintWriter

- Kann von jedem **`OutputStream`** oder **`Writer`** erzeugt werden.
- **`println(String s)`**: schreibt den String und das/die EOL Zeichen.
- Beispiel: Datei einlesen und auf der Standardausgabe ausgeben:

```
import java.io.*;

public class CopyStream {
    public static void main(String a[]) throws Exception {
        BufferedReader in = new BufferedReader(
                                 new FileReader("test.txt"));
        PrintWriter out = new PrintWriter(System.out);
        for(String line; (line = in.readLine())!=null;) {
            out.println(line);
        }
        out.close();
    }
}
```


12 FileWriter

- **`FileWriter`** wird verwendet um Zeichen in eine Datei zu schreiben.
  - ◆ Nachdem das Schreiben beendet ist sollte **`close()`** aufgerufen werden!