

# **Programmierung II**

## **Thema 1: Einführung, Klassen, Objekte, JUnit**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

# Einordnung an der FH LU

Bachelor Studiengang  
Wirtschaftsinformatik  
an der FH LU

Master Studiengang  
Wirtschaftsinformatik  
an der FH LU

1. Sem.: Programmierung I  
Grundkenntnisse Java



2. Sem.: Programmierung II  
Objektorientierung mit Java



3. Sem.: Datenbanken  
DB-Zugriff mit Java (JDBC)



4. Sem.: Webanwendungen:  
Java Server Pages



Anwendungsentwicklung /  
Java EE

- Objektorientierung
  - Klassen und Objekte
  - Vererbung (extends)
  - Interfaces (Mehrfachvererbung: implements)
- Weitere ausgewählte Kapitel aus Java
  - JUnit
  - Exceptions
  - Collections
  - Generics
  - Threads
  - Diverses, z.B. JavaDocs, JAR-Files
- Optionale Themen – je nach der zur Verfügung stehenden Zeit
  - evtl. Visual Editor
  - evtl. Java-Netzwerkprogrammierung

# Anmerkung

- Das Java-Universum wächst ständig
  - Die Sprache selbst wächst mit jedem JDK,
  - Zusätzliche Pakete für unterschiedliche Anwendungsbereiche.
- Wir werden bei weitem nicht jedes Java-Thema behandeln können.
- Wir werden die Themen, die wir behandeln, nicht in allen Feinheiten behandeln können.
- Unser Ziel ist es, Ihnen so viele Grundlagen mitzugeben, dass Sie die nicht behandelten Punkte anhand von Dokumentation und Büchern selbst erarbeiten können.

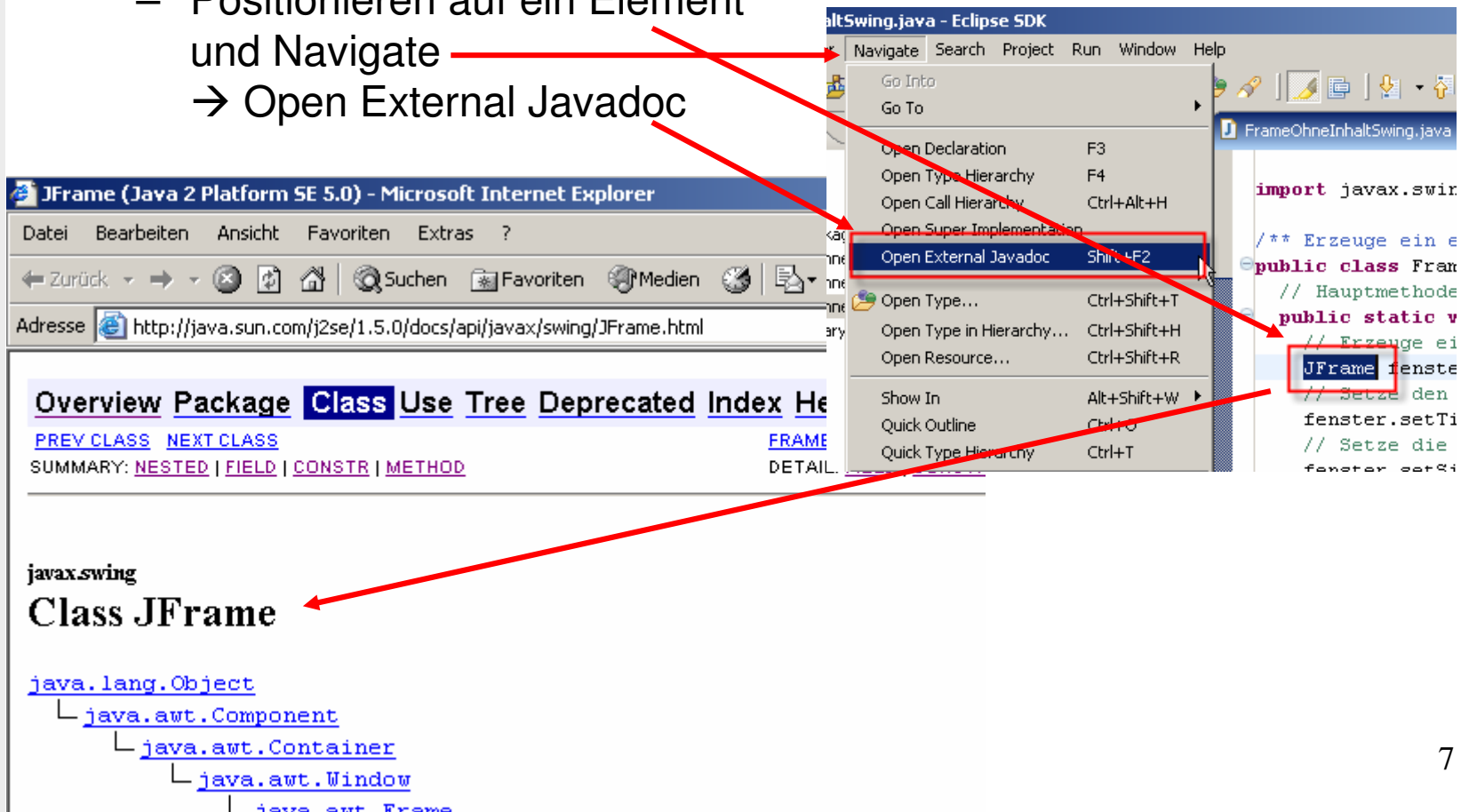
# Literatur

- Vgl. Literaturliste unter <http://www.roeckle.de>
  - K.G.Deck, H.Neuendorf: Java-Grundkurs für Wirtschaftsinformatiker, Vieweg, 2007
  - Eclipse 3 für Java-Programmierung, RRZN / Herdt-Verlag
  - Java 2 JDK 5 / JDK 6 – Grundlagen Programmierung, RRZN / Herdt-Verlag
  - Java ist auch eine Insel - Galileo Computing, <http://www.galileocomputing.de/openbook/javainsel6/>
  - K.Echtle, M.Goedicke: Lehrbuch der Programmierung mit Java, dpunkt, 2000
  - F.Jobst: Programmieren in Java, 5. Aufl., Hanser
  - H.Müller, F.Weichert: Vorkurs Informatik, Teubner
  - H.Mössenböck: Sprechen Sie Java? - 3. Aufl., dpunkt, 2005
  - D.Ratz, J.Scheffler, D.Seese: Grundkurs Programmieren in Java, Bd.1: Der Einstieg in Programmierung und Objektorientierung, 2. Auflage, Hanser, 2004
  - D. Ratz, J.Scheffler, D.Seese: Grundkurs Programmieren in Java, Bd.2: Programmierung kommerzieller Systeme, Hanser, 2003
  - R.Schiedermeier: Programmieren mit Java, Pearson

- Die Dokumentation der Java Bibliothek liegt in Form von API-Spezifikationen vor.
- Im Internet abrufen unter
  - [java.sun.com/api](http://java.sun.com/api) → Standard Edition, J2SE 1.5.0 bzw.
  - [java.sun.com/api](http://java.sun.com/api) → Standard Edition, Java SE 6 → Core API Docs → 5.0 oder 6 (English)
- oder runterladen, z.B. unter
  - [java.sun.com/api](http://java.sun.com/api) → Popular Downloads → JDK 6.0 → Java SE 6 Documentation
  - entpacken und öffnen über `\docs\api\index.html`
- Manchmal auch im Intranet
  - O-Platte

# API-Spezifikation

- In Eclipse kann die API-Spez. ggfs. kontextsensitiv aufgerufen werden:
  - Positionieren auf ein Element und Navigate → Open External Javadoc



The screenshot illustrates the process of opening the external Javadoc for the `JFrame` class in Eclipse. A context menu is open over the `JFrame` class in the source code, with the 'Open External Javadoc' option (labeled `Shift+F2`) selected. A red arrow points from the text 'Open External Javadoc' in the list above to this menu item. Another red arrow points from the 'Navigate' menu item in the context menu to the 'JFrame' class in the source code. A third red arrow points from the 'Open External Javadoc' menu item to the Javadoc page displayed in the Microsoft Internet Explorer browser window. The browser window shows the 'Class JFrame' page for the `javax.swing` package, with a class hierarchy tree on the left.

**Class JFrame**

- [java.lang.Object](#)
  - [java.awt.Component](#)
    - [java.awt.Container](#)
      - [java.awt.Window](#)
        - [java.awt.Frame](#)

# Klassen und Objekte

➔ Jetzt gehts los mit Programmierung 2:

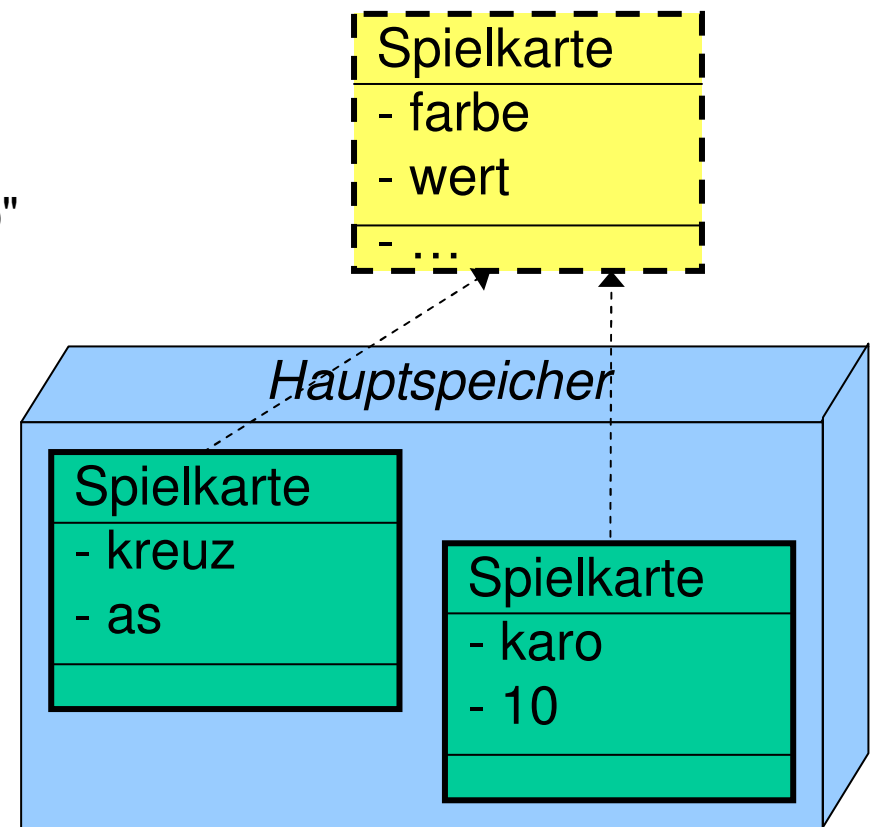
- Grundlage der Objektorientierung ➔ Klasse.

## Klasse

- existiert als Modell (Vorlage, Schablone) im **Katalog**
- gibt's nur einmal für einen "Typ"  
➔ sie definiert einen Typ

## Objekte

- existieren während der Ausführung eines Programms konkret als Bits und Bytes im **Hauptspeicher**
- mehrere Objekte eines Typs
  - sind möglich
  - haben dieselbe Struktur





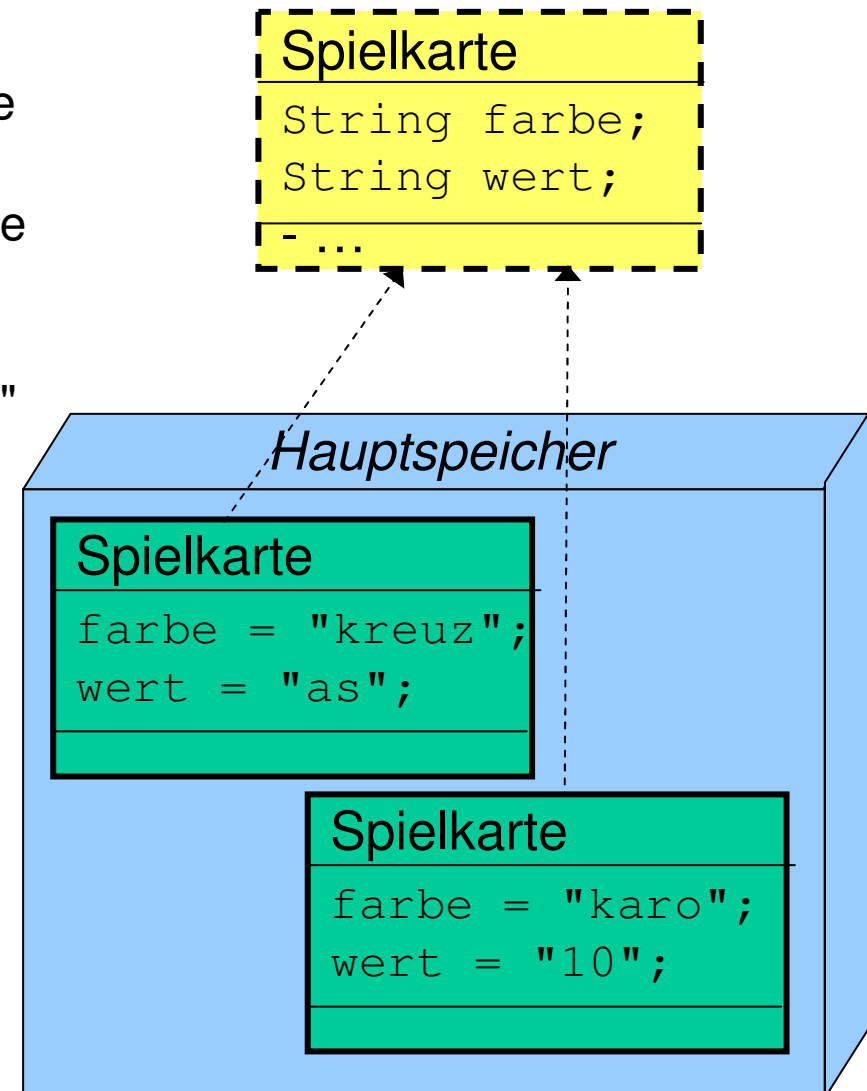
# Klassen und Objekte

## Klasse

- legt die Struktur für alle Objekte dieses Typs fest und zwar
- als (Objekt-)Variablen mit Name und Datentyp
- Objektvariablen heißen auch "Attribute" oder "Eigenschaften"

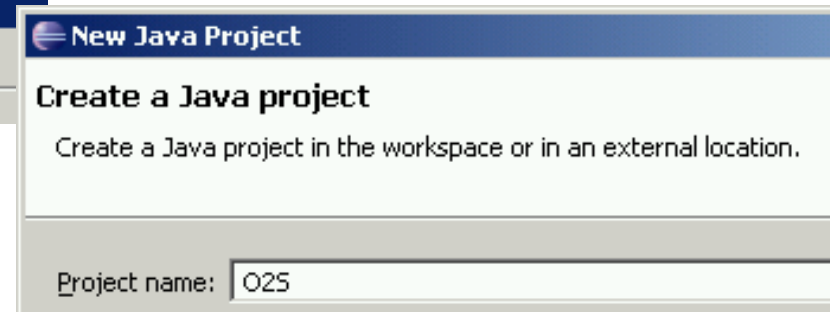
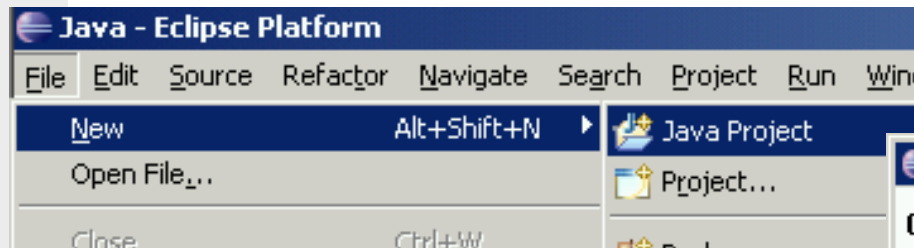
## Objekte

- Verschiedene Objekte einer Klasse (eines Typs) haben
  - dieselben Eigenschaften (Arten von Eigenschaften)
  - aber evtl. unterschiedliche Werte für diese Eigenschaften



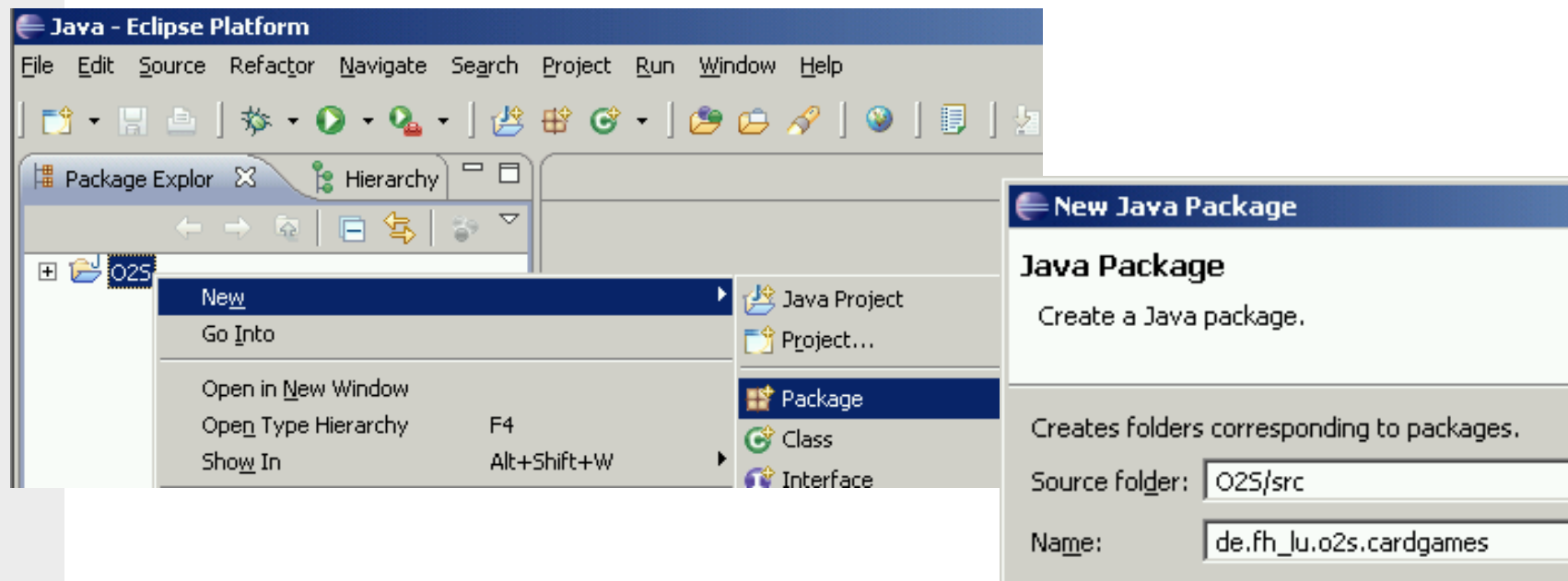
# Klasse "Spielkarte" in Java

- Aufgabe 1:
  - Entwickeln Sie die dargestellte Klasse "Spielkarte" in Java.  
Achtung: Klassenname im Singular.
  - Legen Sie dazu ein neues Eclipse-Projekt O2S und darin ein Package `de.fh_lu.o2s.cardgames` (Kleinbuchstaben) an
- Schritt 1: Neues Projekt "O2S" in Eclipse:
  - File → New → Java Project...
  - Project name: "O2S"
  - Finish



# Klasse "Spielkarte" in Java

- Schritt 2: Neues Package "de.fh\_lu.o2s.cardgames" im Projekt O2S
  - Rechter Mausklick auf das Projekt O2S
  - New → Package
  - Name: "de.fh\_lu.o2s.cardgames"
  - Finish



# Klasse "Spielkarte" in Java

- Anmerkungen:
  - Paketnamen sollen ausschließlich aus Kleinbuchstaben bestehen.
  - Sie sind aufgebaut umgekehrt wie Internet-Domänen:
    - Top-Level-Qualifier, z.B. "de"
    - Domain-Qualifier, z.B. "fh\_lu"
    - Evtl. Subdomain-Qualifier, z.B. "o2s"
    - Package-Name, z.B. "cardgames"
  - Bindestrich '-' ist nicht erlaubt, Underscore '\_' ist möglich
  - Im Dateisystem wird die Paketstruktur in Form von Verzeichnissen aufgebaut, z.B. src\de\fh\_lu\o2s\cardgames
  - Darin werden die Java-Dateien gespeichert.

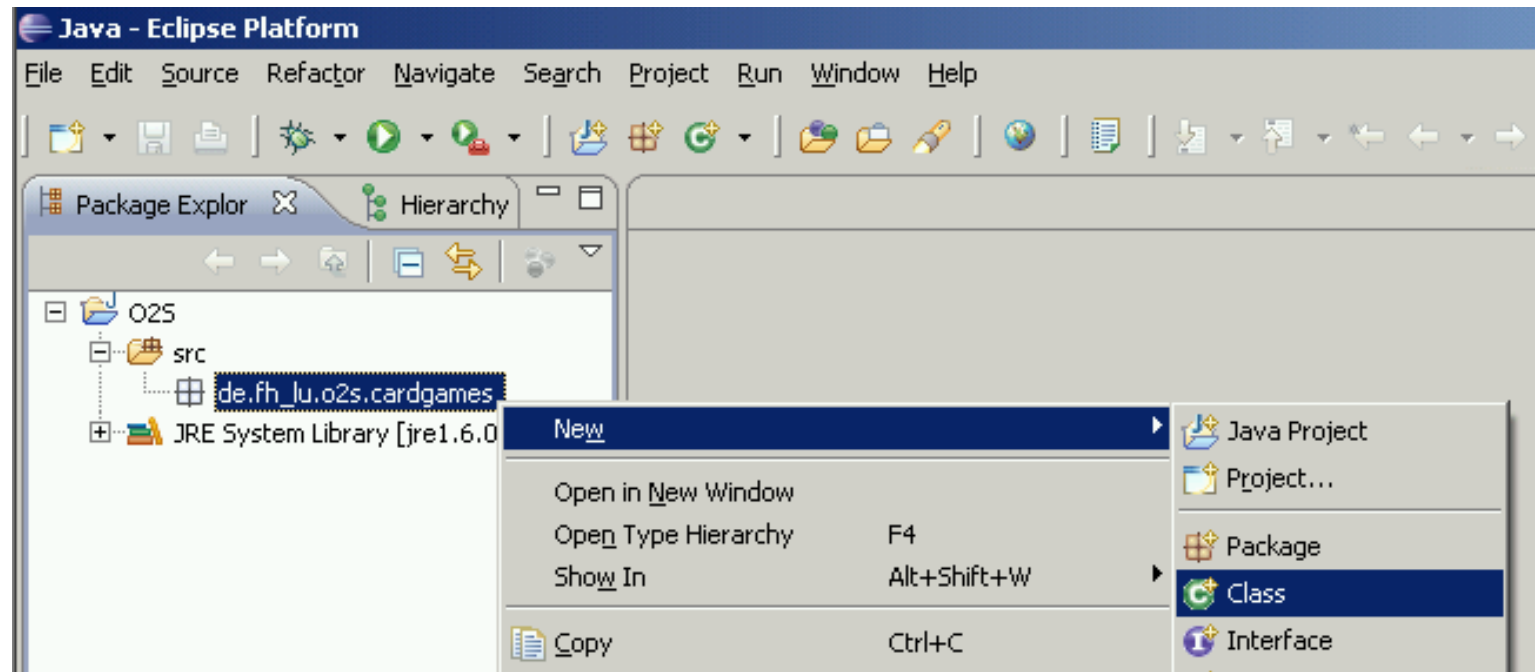
Creates folders corresponding to packages.

Source folder:

Name:

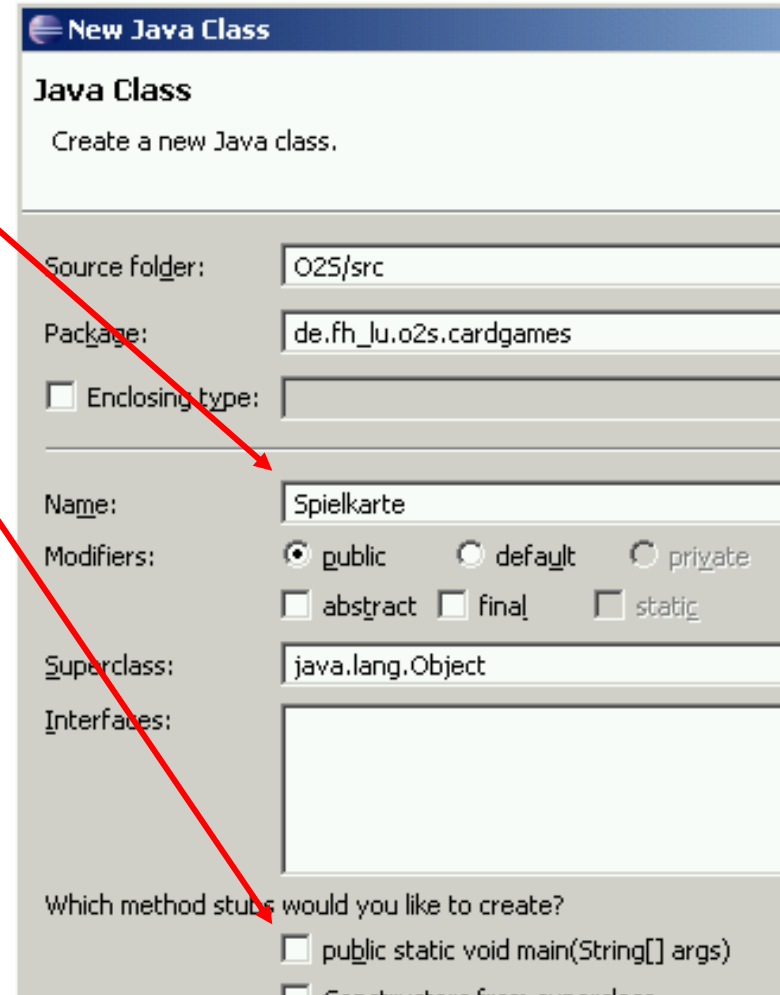
# Klasse "Spielkarte" in Java

- Schritt 3: Neue Klasse "Spielkarte" im Package `de.fh_lu.o2s.cardgames`
  - Rechter Mausklick auf das Package `de.fh_lu.o2s.cardgames`
  - New → Class



# Klasse "Spielkarte" in Java

- Schritt 3 (Forts.): Neue Klasse "Spielkarte" ...
  - Name: "Spielkarte"
  - keine "main"-Methode  
→ Finish
- Anmerkung:
  - Eine Klasse mit einer main-Methode ist eine Applikation / Anwendung / ein Programm
  - Unsere Klasse Spielkarte ist **kein Programm** sondern nur eine Vorlage (Schablone, Modell, Template) für Objekte vom Typ Spielkarte.



**New Java Class**

Java Class  
Create a new Java class.

Source folder: O2S/src

Package: de.fh\_lu.o2s.cardgames

☐ Enclosing type:

Name: Spielkarte

Modifiers: ☒ public ☐ default ☐ private  
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

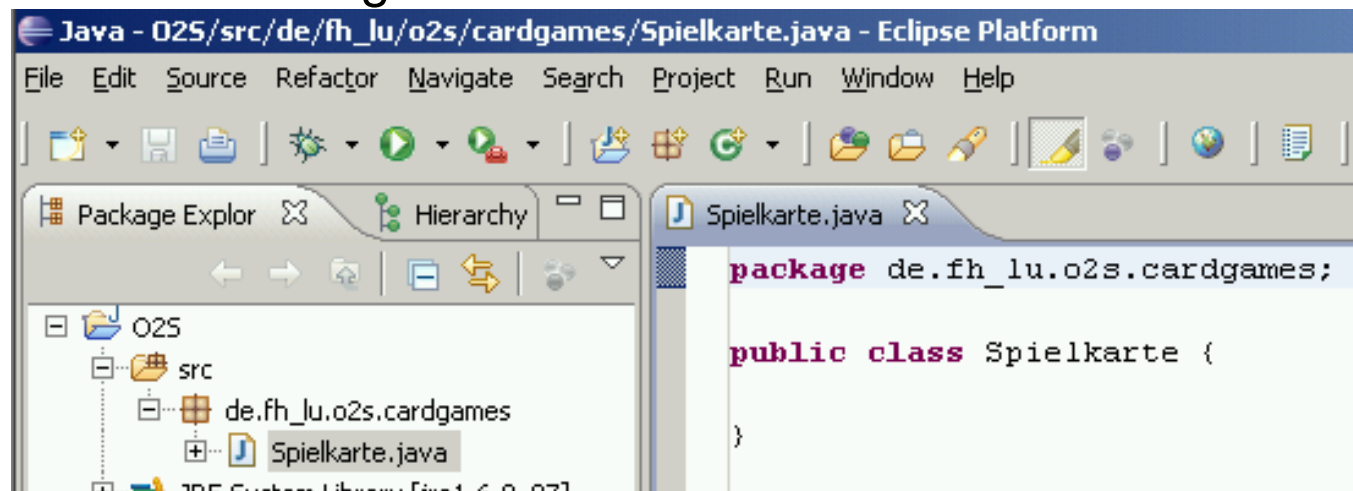
Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)  
☐ Constructors from superclass

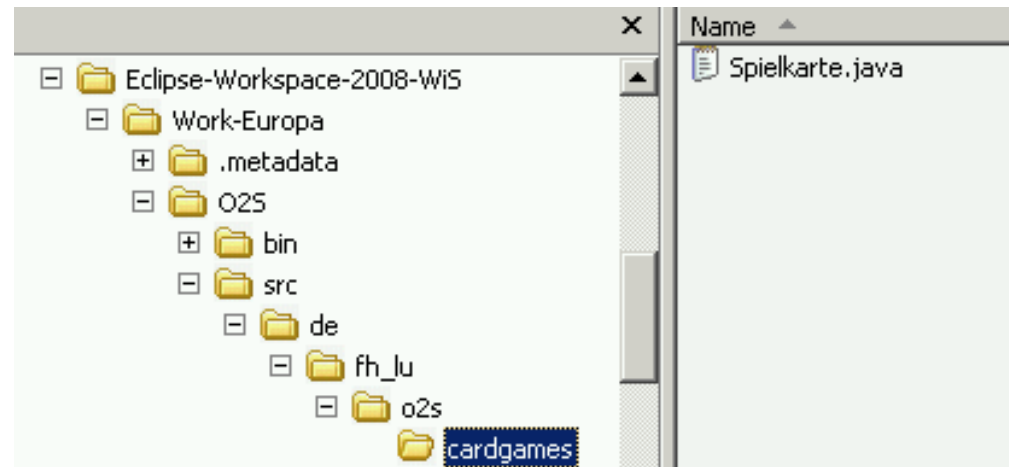
# Klasse "Spielkarte" in Java

- Zwischenergebnis



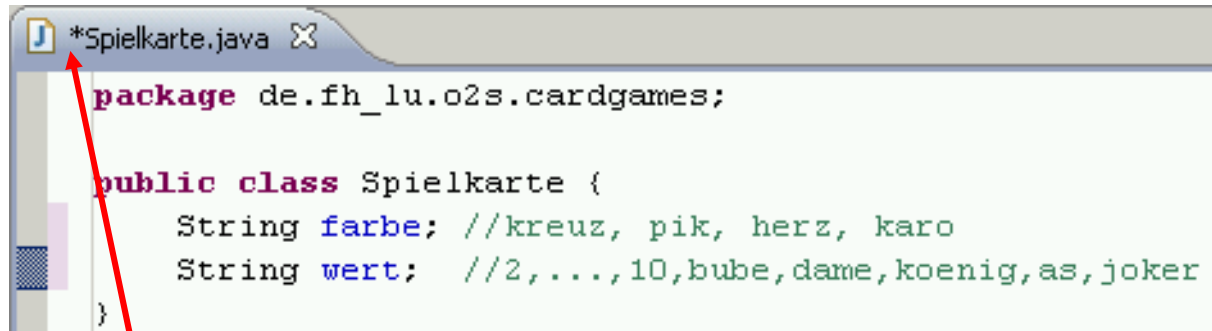
- Anmerkung

- Am Anfang jeder Klasse steht, zu welchem Package sie gehört.
- Der Code wird als <Name>.java im Dateisystem gespeichert.



# Klasse "Spielkarte" in Java

- Schritt 4: Attribute `farbe` und `wert`
  - Im Code eintragen, dabei Datentyp jedes Attributs angeben.
  - Objektvariablen beginnen immer mit einem Kleinbuchstaben.
  - Wenn die Namen der Objektvariablen nicht zusammengesetzt sind, bestehen sie nur aus Kleinbuchstaben.



```
*Spielkarte.java X
package de.fh_lu.o2s.cardgames;

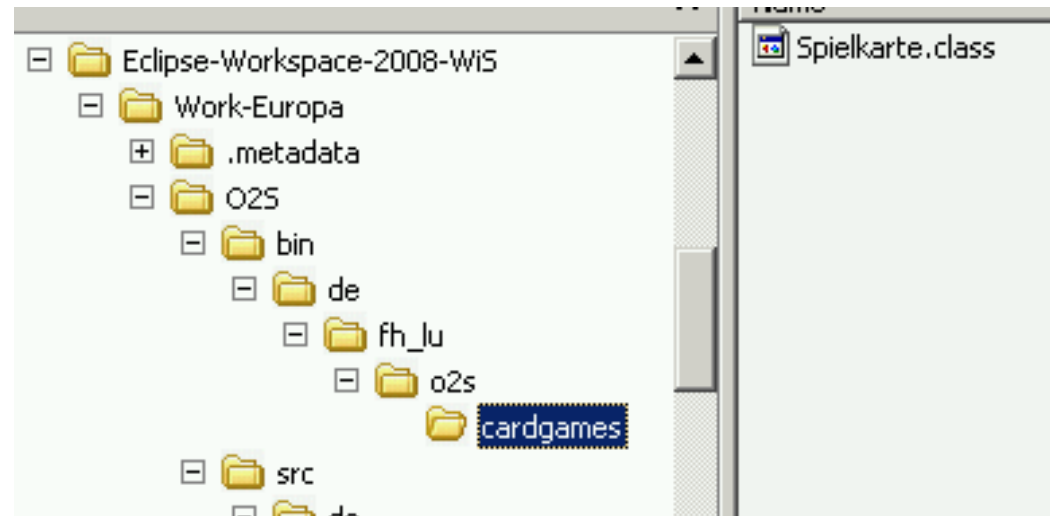
public class Spielkarte {
    String farbe; //kreuz, pik, herz, karo
    String wert;  //2,...,10,bube,dame,koenig,as,joker
}
```

- das Sternchen neben dem Klassennamen bedeutet, dass der Code verändert wurde.

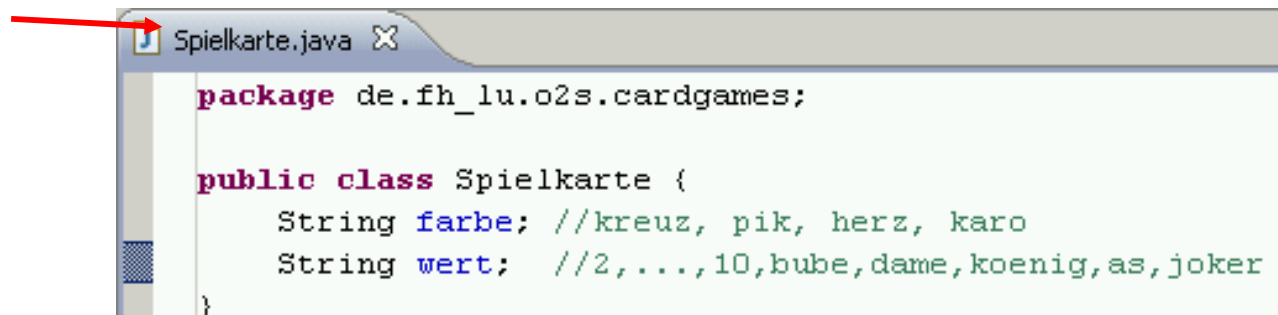


# Klasse "Spielkarte" in Java

- Schritt 4 (Forts.):
  - In Eclipse wird mit Strg + S gespeichert, dabei wird die compilierte Klasse als <Name>.class im Dateisystem gespeichert.



- der Code wird außerdem auf Fehler geprüft und compiliert.
- das Sternchen verschwindet.



# Klasse "AppTestSpielkarte"

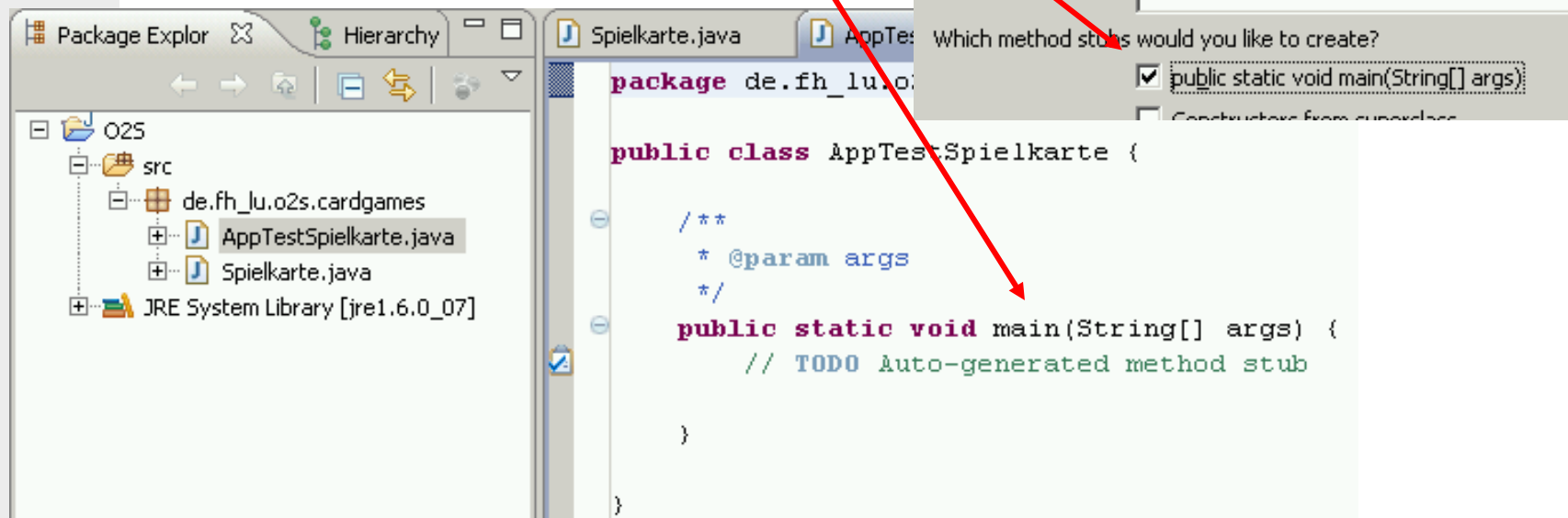
- Anmerkungen:
  - Bis dahin gibt es nur die Klasse und noch kein Objekt
  - Eine Klasse implementiert einen (selbst erzeugten) Datentyp, vgl.
    - elementare Datentypen `int`, `char`, etc.
    - in Java bereits vorhandene Klassen, z.B. `String`, `StringBuffer`, etc.
- Schritt 5: Testen der Klasse mit einer Applikation
  - Wir erzeugen eine Applikation `AppTestSpielkarte` im selben Package
  - **Erinnerung: Eine Applikation ist eine Klasse mit einer main-Methode, also fast wie in Schritt 3...**
  - Rechter Mausklick auf das Package `de.fh_lu.o2s.cardgames`
  - New → Class
  - Name: "AppTestSpielkarte"
  - Aktivieren: "main"-Methode
  - Finish

Which method stubs would you like to create?

☒ `public static void main(String[] args)`

# Klasse "AppTestSpielkarte"

- Schritt 5 (Forts.):
  - Aktivieren: "main"-Methode  
→ Finish



Source folder: O25/src

Package: de.fh\_lu.o2s.cardgames

☐ Enclosing type:

Name: AppTestSpielkarte

Modifiers: ☒ public ☐ default ☐ private

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☒ public static void main(String[] args)

☐ Constructors from superclass

```
package de.fh_lu.o2s.cardgames;

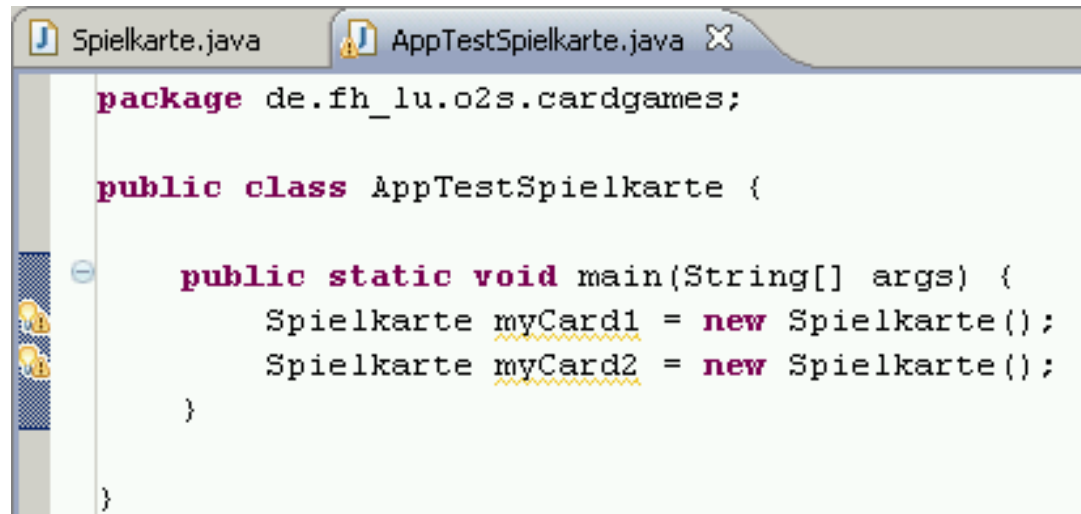
public class AppTestSpielkarte {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }

}
```

# Klasse "AppTestSpielkarte"

- Schritt 5 (Forts.):
  - Innerhalb der main-Methode
  - zwei Objekte vom Typ Spielkarte erzeugen mit new Spielkarte()
  - **und** gleichzeitig je einer Variablen vom Typ Spielkarte zuweisen.



```
package de.fh_lu.o2s.cardgames;

public class AppTestSpielkarte {

    public static void main(String[] args) {
        Spielkarte myCard1 = new Spielkarte();
        Spielkarte myCard2 = new Spielkarte();
    }
}
```

- Anmerkung:
  - Die Variablen myCard1 und myCard2 "leben" in der main-Methode der Applikation AppTestSpielkarte und
  - sind deshalb lokale Variablen und keine Objektvariablen.

# Klasse "AppTestSpielkarte"

- Schritt 5 (Forts.):
  - Setzen von farbe = "kreuz" und wert = "as" innerhalb des Spielkarten-Objekts myCard1,
  - Analog: Verwandeln von myCard2 in eine Karo 10.

```
public static void main(String[] args) {  
    Spielkarte myCard1 = new Spielkarte();  
    Spielkarte myCard2 = new Spielkarte();  
    myCard1.farbe = "kreuz";  
    myCard1.wert = "as";  
    myCard2.farbe = "karo";  
    myCard2.wert = "10";  
}
```

- Zugreifen auf farbe und wert und Ausgabe auf der Konsole

```
public static void main(String[] args) {  
    Spielkarte myCard1 = new Spielkarte();  
    Spielkarte myCard2 = new Spielkarte();  
    ...
```

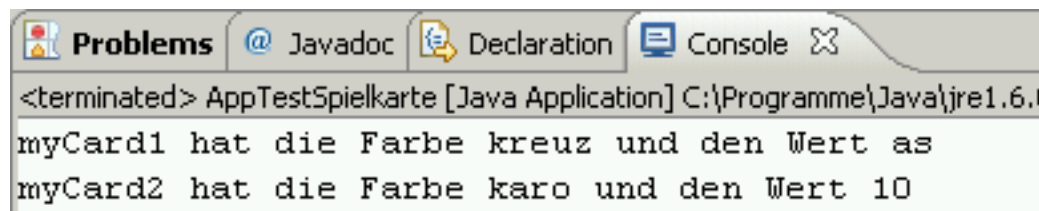
```
    System.out.println("myCard1 hat die Farbe " + myCard1.farbe +  
        " und den Wert " + myCard1.wert);  
    System.out.println("myCard2 hat die Farbe " + myCard2.farbe +  
        " und den Wert " + myCard2.wert);  
}
```

# Klasse "AppTestSpielkarte"

- Schritt 6: Ausführen von AppTestSpielkarte
  - Rechter Mausklick auf die Klasse AppTestSpielkarte
  - Run As → 2 Java Application



- Ausgabe des Programms auf der Konsole prüfen:

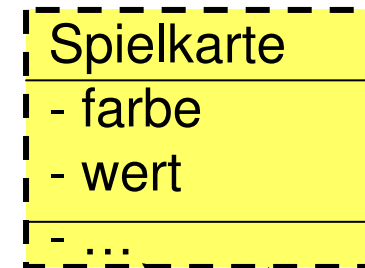


# Erinnerung

Diese Situation haben wir nun programmiert:

## Klasse

- existiert als Modell (Vorlage, Schablone) im **Katalog**
- gibt's nur einmal für einen "Typ"  
→ sie definiert einen Typ

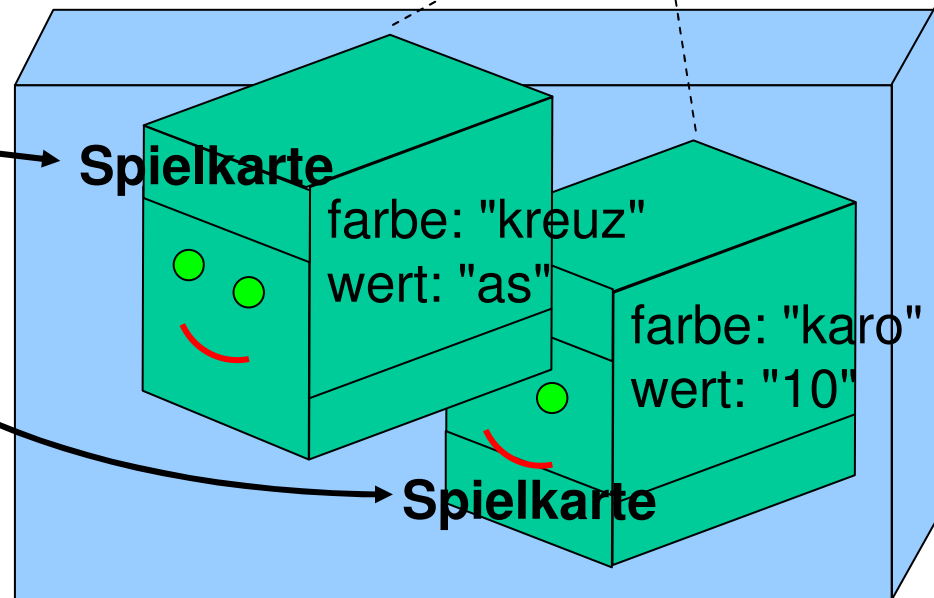


## Objekte in der Applikation

Spielkarte myCard1

Spielkarte myCard2

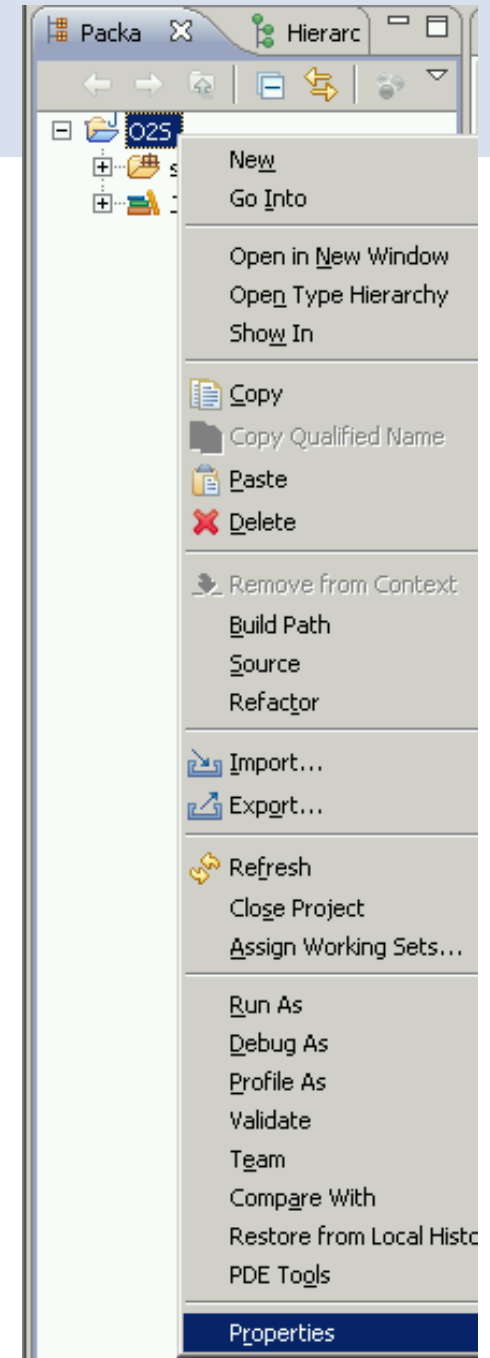
- Ein Programm findet ein Objekt im Hauptspeicher über eine Variable bzw. einen Pointer (einen Link, eine **Referenz**).
- Die Referenz hat einen Datentyp, der zum Objekttyp (Klasse) passen muss.



**Hauptspeicher**

# Test mit JUnit

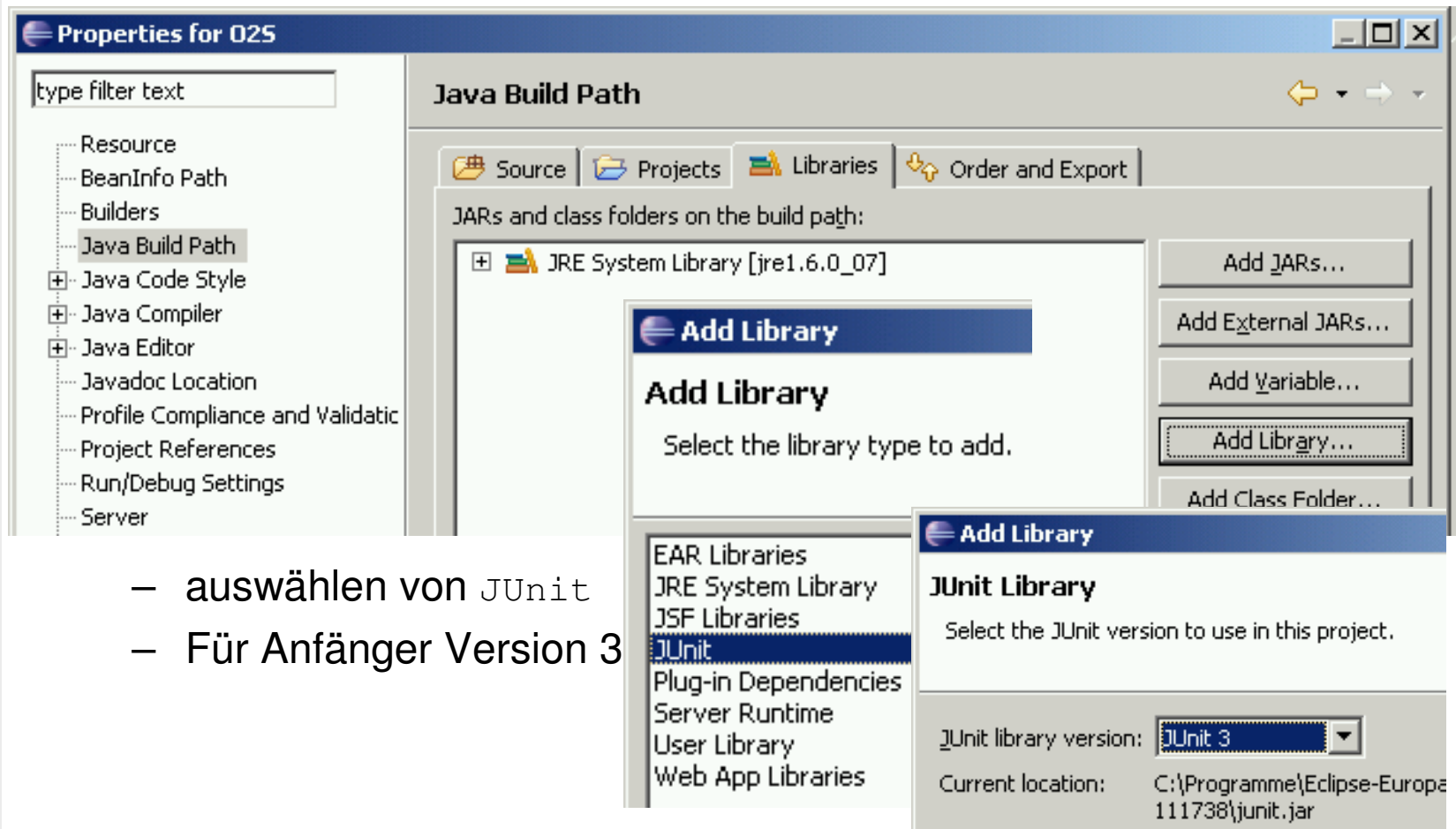
- Aufgabe 2:
  - Entwickeln Sie mit **JUnit** einen Test für Ihre Klasse Spielkarte, der
    - ein Spielkartenobjekt anlegt,
    - die Variablen füllt,
    - das Spielkartenobjekt anzeigt
- Schritt 0: Vorbereitung
  - In Eclipse für Java EE (früher: Eclipse WTP) wird JUnit mit ausgeliefert,
  - muss aber noch ins jeweilige Projekt eingebunden werden:
  - Rechter Mausklick auf das Projekt O2S → Properties





# Testen mit JUnit

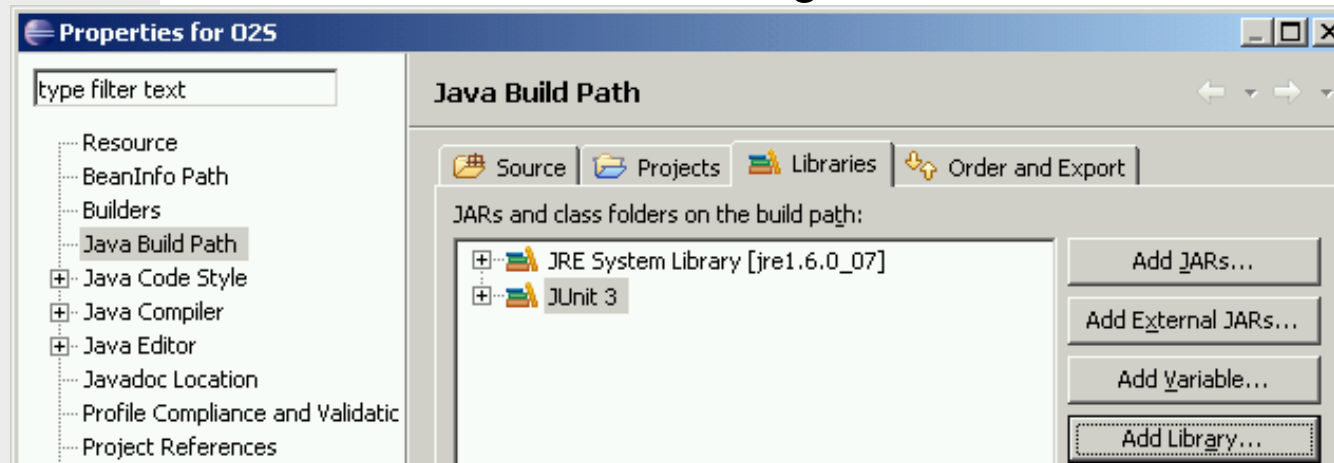
- Vorbereitung (Forts.):
  - Java Build Path → Libraries → Add Library...



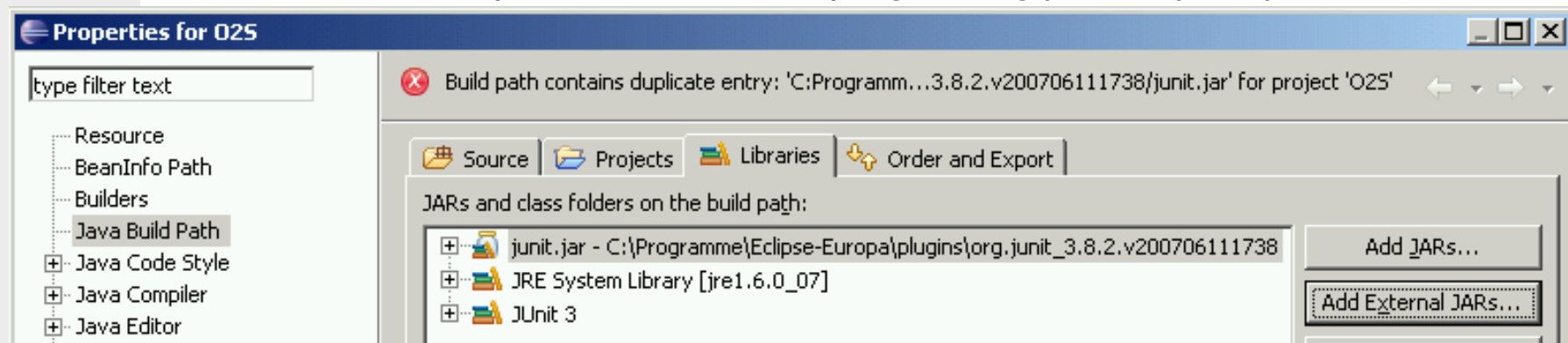
- auswählen von JUnit
- Für Anfänger Version 3

# Testen mit JUnit

- Ende der Vorbereitung:

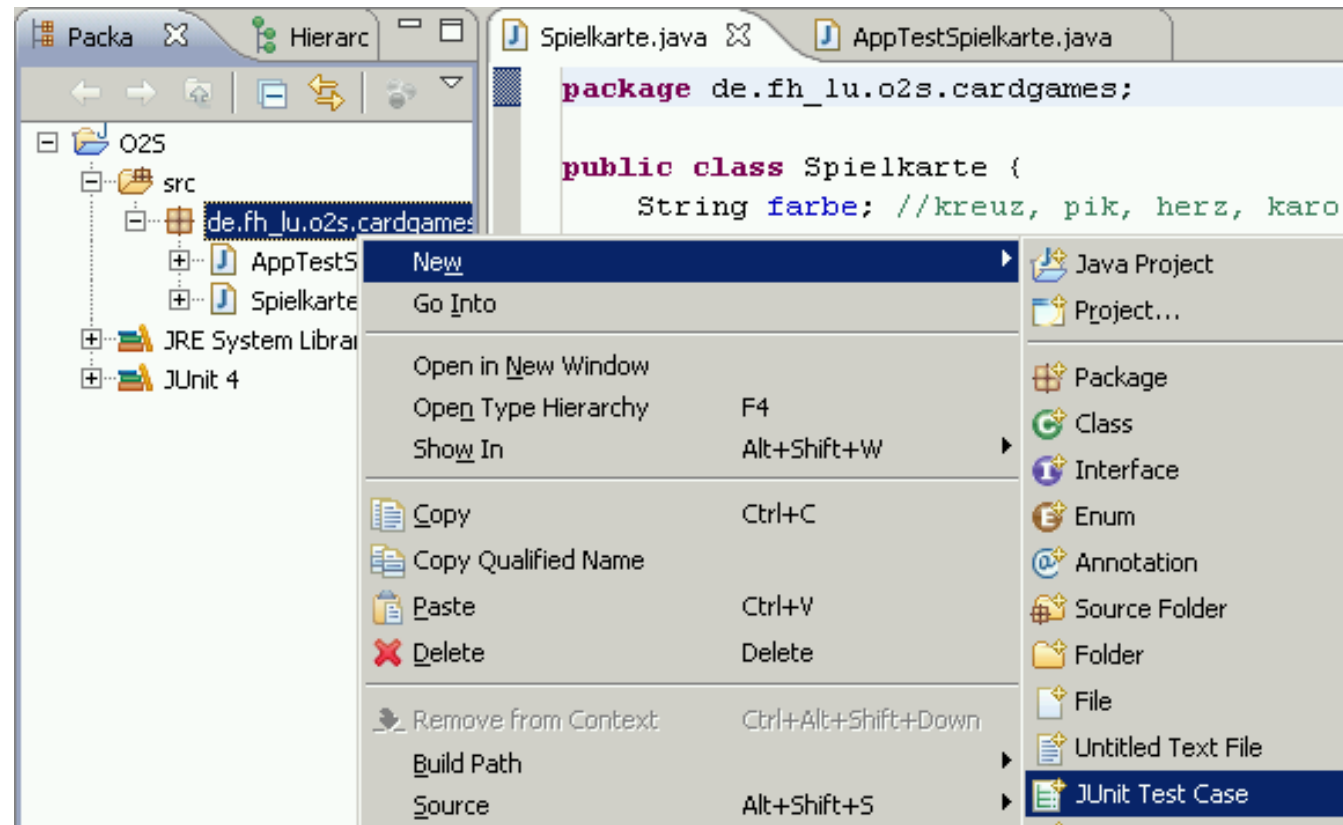


- Anmerkung:
  - Man könnte JUnit auch mit Add External JARS... einbinden
  - aus <Eclipse-Verzeichnis>\plugins\org.junit...\junit.jar



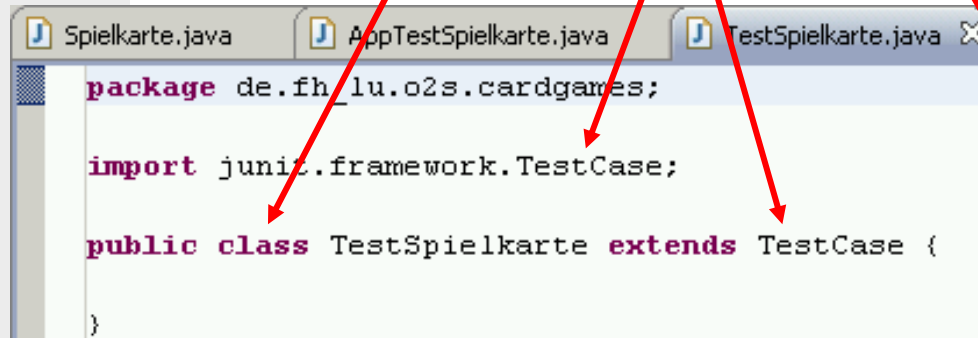
# Testen mit JUnit

- Schritt 1: Einen „JUnit-Test“ anlegen mit
  - Rechter Mausklick auf das Package `de.fh_lu.o2s.cardgames`
  - New → JUnit Test Case



# Testen mit JUnit

- Schritt 1 (Forts.):
  - Name des Tests angeben
  - Angeben, welche Klasse getestet werden soll.
  - Bisheriges Ergebnis ist eine Testklasse, die in das JUnit Test-Framework eingebettet ist.

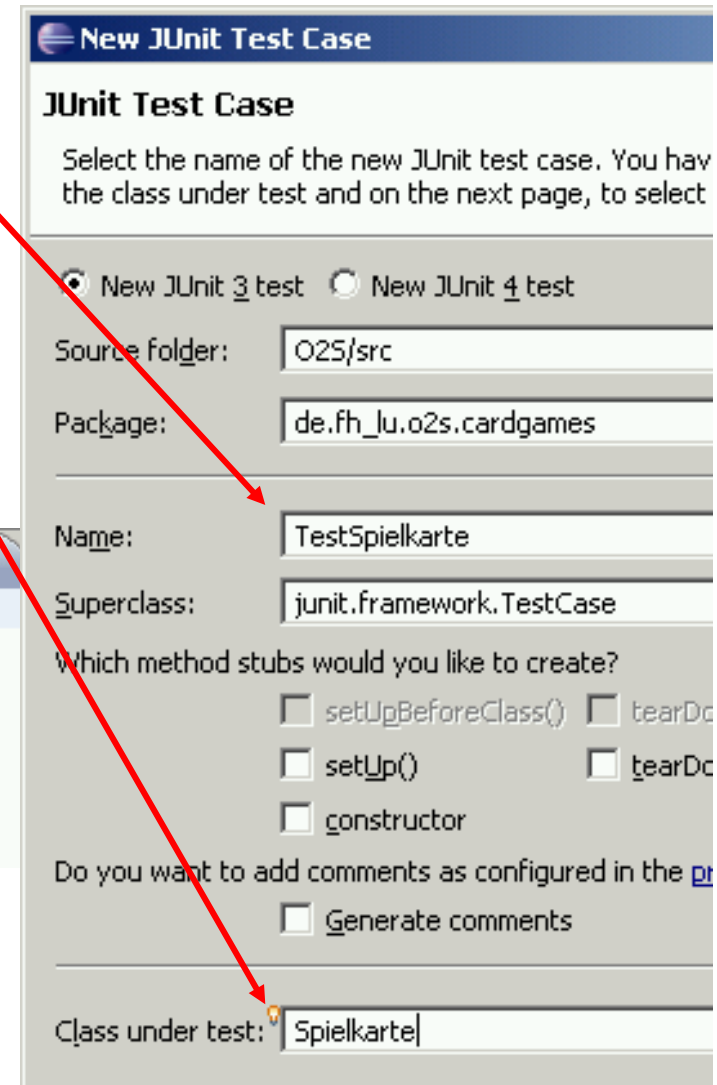


```
package de.fh_lu.o2s.cardgames;

import junit.framework.TestCase;

public class TestSpielkarte extends TestCase {

}
```



**New JUnit Test Case**

JUnit Test Case

Select the name of the new JUnit test case. You have the class under test and on the next page, to select

☒ New JUnit 3 test ☐ New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownClass()

☐ setUp() ☐ tearDown()

☐ constructor

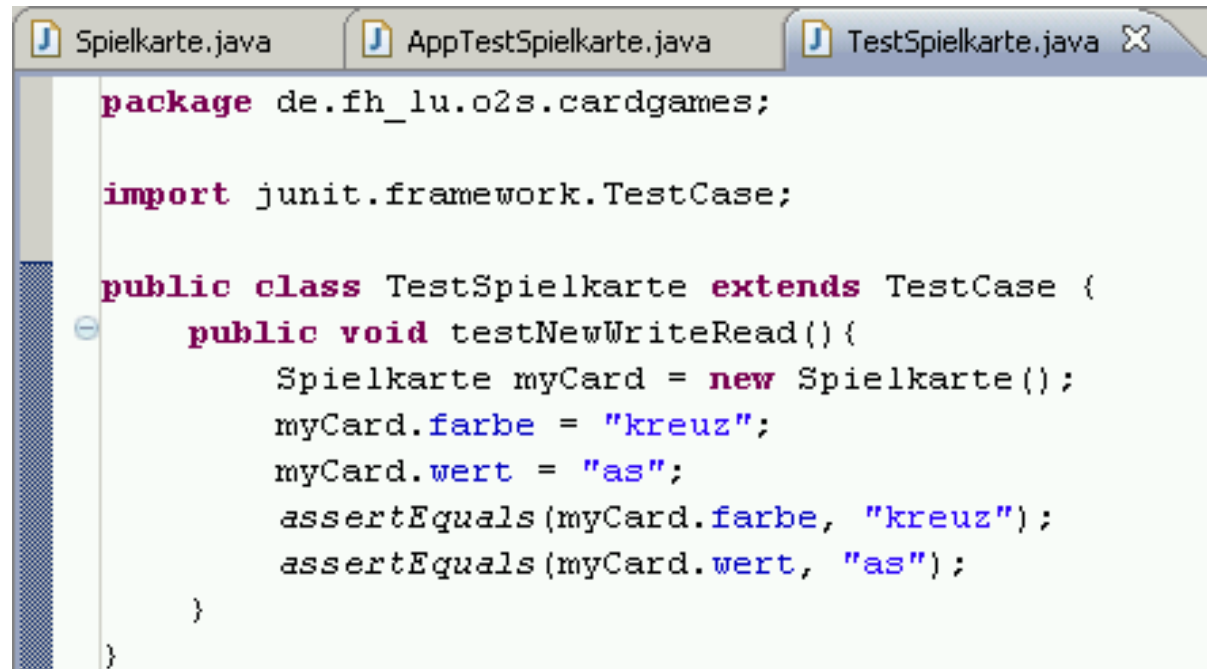
Do you want to add comments as configured in the preferences?

☐ Generate comments

Class under test:

# Testen mit JUnit

- Schritt 2: Testcode schreiben:
  - Wir benötigen eine Testmethode. Diese nennen wir `testNewWriteRead()`
  - Der folgende Code
    - erzeugt – analog zu `AppTestSpielkarte` – eine neue Spielkarte,
    - weist dieser Attributwerte für "farbe" und "wert" zu,
    - und prüft, ob diese Werte wieder gelesen werden können:



```
package de.fh_lu.o2s.cardgames;

import junit.framework.TestCase;

public class TestSpielkarte extends TestCase {
    public void testNewWriteRead() {
        Spielkarte myCard = new Spielkarte();
        myCard.farbe = "kreuz";
        myCard.wert = "as";
        assertEquals(myCard.farbe, "kreuz");
        assertEquals(myCard.wert, "as");
    }
}
```

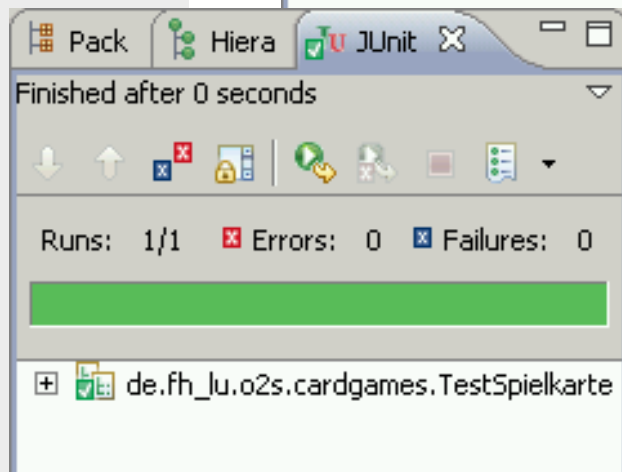
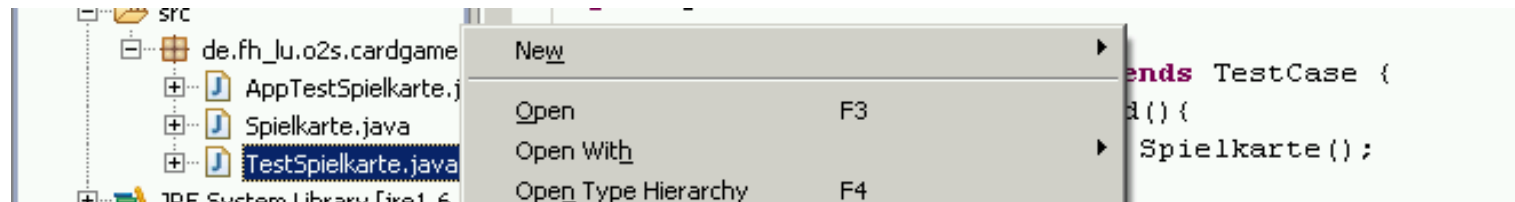
# Testen mit JUnit

- Anmerkung:
  - Die Methode `assertEquals()` stammt aus dem JUnit-Test-Framework
  - Sie vergleicht die angegebenen Werte, z.B. `myCard.farbe` und `"kreuz"`
  - Der wahre Nutzen erschließt sich aber erst bei der Ausführung des Tests

```
public void testNewWriteRead(){  
    Spielkarte myCard = new Spielkarte();  
    myCard.farbe = "kreuz";  
    myCard.wert = "as";  
    assertEquals(myCard.farbe, "kreuz");  
    assertEquals(myCard.wert, "as");  
}
```

# Testen mit JUnit

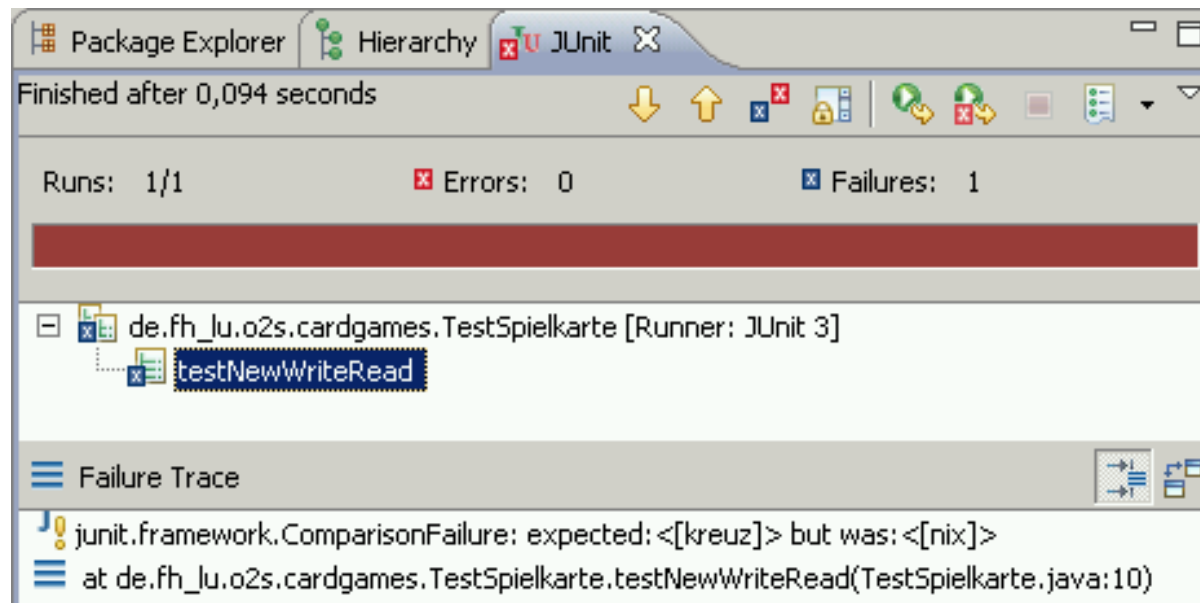
- Schritt 3: Test ausführen:
  - Rechter Mausklick auf die Testklasse "TestSpielkarte"
  - Run As → JUnit Test



- Beachte im linken Fensterbereich das JUnit-Ergebnis-Feld:
- Der Test war erfolgreich (grün):
  - Gesamter Code fehlerfrei durchgelaufen,
  - alle Assertions (Behauptungen) erfüllt.

# Testen mit JUnit

- Schritt 4: Gegenprobe
    - Wir ändern den Code, so dass enthalten ist:
- ```
myCard.farbe = "kreuz";  
myCard.wert = "as";  
assertEquals(myCard.farbe, "nix");  
assertEquals(myCard.wert, "gar nix");
```
- Test-Durchführung wieder mit Run As → JUnit Test
  - Ergebnis: Fehler incl. aussagekräftiger Fehlermeldung:





# Anmerkungen

- Vorteile
  - Der Test wird einmal angelegt und kann immer wieder durchgeführt werden: „Regressionstest“
  - Dadurch lohnt es sich, den Test gründlich zu entwickeln
  - Der Test kann bereits vor der Entwicklung der Software angelegt werden: „test-driven Softwareentwicklung“
- Nachteile
  - Aufwand zur Testfallentwicklung,
  - Unvollständige Testfälle können zu falscher Sicherheit führen.

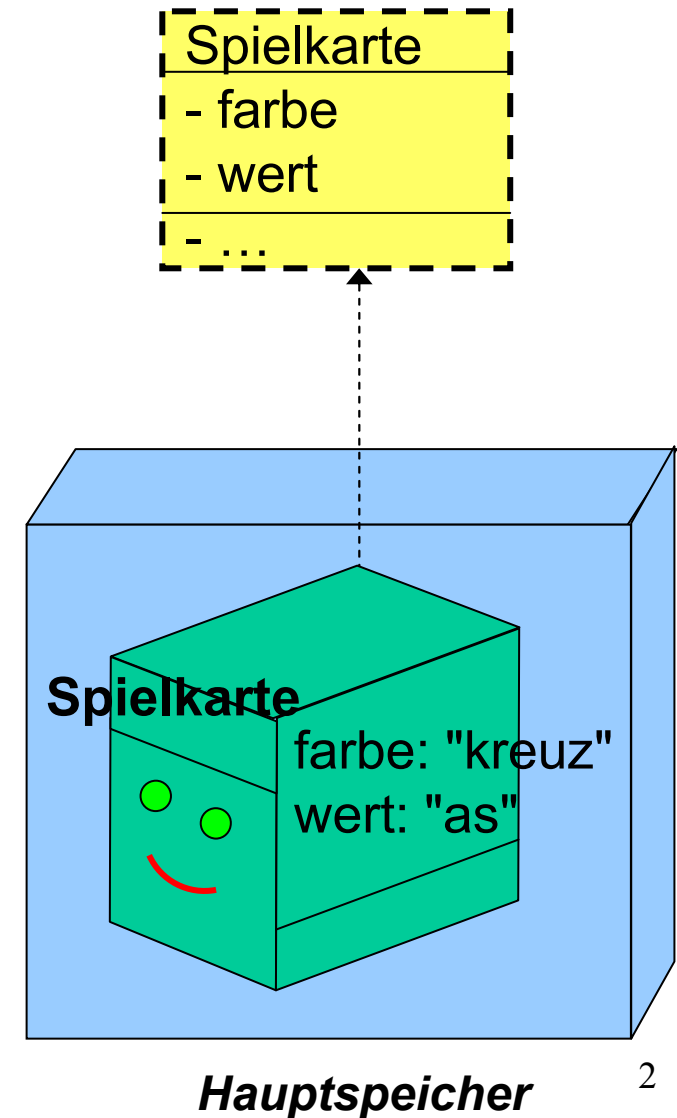
# **Programmierung II**

## **Thema 2: Methoden**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

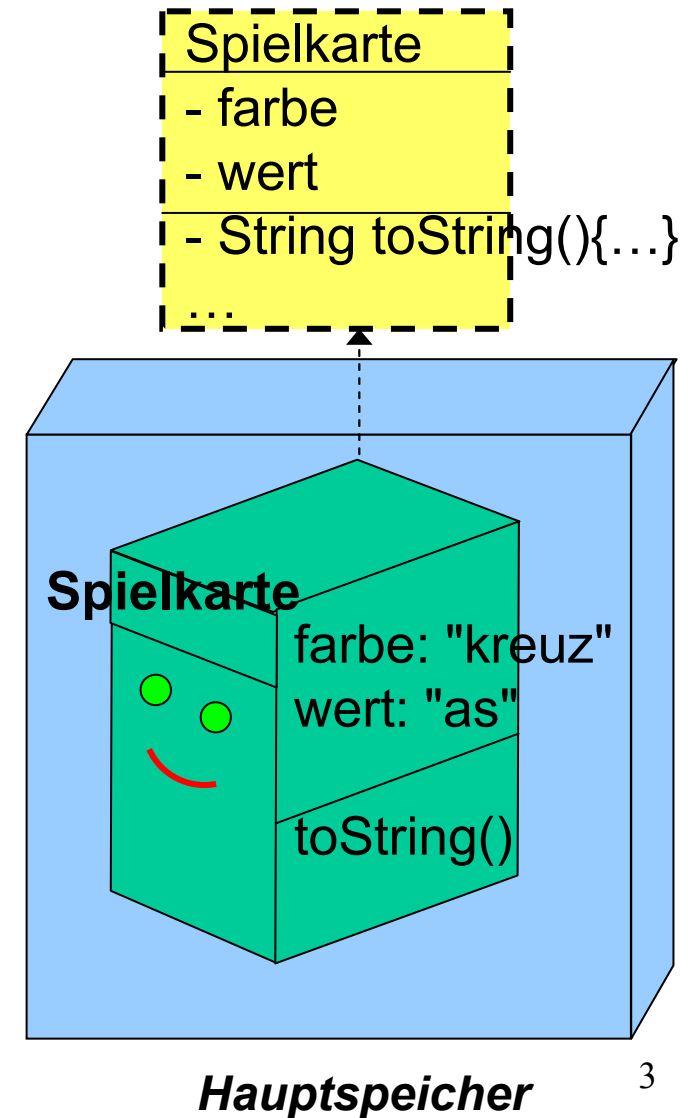
# Methoden

- Bis jetzt:
  - beschreibt die Klasse `Spielkarte`  
Spielkarten-Objekte mit bestimmten Eigenschaften (Attributen)
- Ab jetzt: Methoden
  - Objekte können Fähigkeiten besitzen.
  - Objekte vom selben Typ (derselben Klasse) haben auch dieselben Fähigkeiten.
  - Fähigkeiten werden als **Methoden** in der jeweiligen Klasse implementiert.



# toString()

- Aufgabe 1:
  - Sorgen Sie dafür, dass alle Spielkartenobjekte befähigt werden, eine Beschreibung von sich selbst zu erzeugen und zurückzugeben.
- Lösungsansatz:
  - Erweitere die Klasse Spielkarte um eine Methode `toString()` mit Rückgabewert `String`,
  - Implementiere als Beschreibung, dass zum Beispiel für ein Kreuz As ausgegeben wird:  
"Spielkarte mit Farbe Kreuz und Wert As"
  - Für Farbe und Wert greift die Methode auf die Attribute zu.



# toString()

- Schritt 1:
  - Bauen Sie in der Klasse `Spielkarte`
    - einen Methodenrumpf `toString{...}`
    - mit Sichtbarkeit `public` und
    - Rückgabewert `String`



```
package de.fh_lu.o2s.cardgames;

public class Spielkarte {
    String farbe; //kreuz, pik, herz, karo
    String wert;  //2,...,10,bube,dame,koenig,as,joker

    public String toString(){
    }
}
```

- Anmerkung:
  - Bis jetzt ist die Methode noch fehlerhaft, weil sie behauptet, sie gebe einen `String` zurück, es aber (noch) nicht tut.
  - Fehlermeldung: - This method must return a result of type `String`

## toString()

- Schritt 2:
  - Erzeugen Sie innerhalb der Methode einen geeigneten String und geben Sie diesen zurück:

```
public String toString(){  
    String myString = "Spielkarte mit Farbe " + farbe +  
                      " und Wert " + wert;  
    return myString;  
}
```

- Anmerkungen:
  - Hier werden einfach 4 Teilstrings aneinandergehängt, nämlich
    - "Spielkarte mit Farbe " – ein konstanter String – beachte die Leerzeichen
    - farbe – das Attribut, das die Farbe des jeweiligen Spielkarten-Objekts enthält
    - " und Wert " – ein konstanter String
    - wert – das Attribut, das den Wert des jeweiligen Spielkarten-Objekts enthält
  - Das Ergebnis wird mit `return` an den jeweiligen Aufrufer der Methode zurückgegeben.

# Test von toString()

- Test 1: Mit AppTestSpielkarte, Erinnerung:
  - Wenn die Applikation AppTestSpielkarte ausgeführt wird, werden zwei Spielkarten-Objekte myCard1 und myCard2 erzeugt und deren Attribute mit Werten gefüllt:

```
public class AppTestSpielkarte {  
  
    public static void main(String[] args) {  
        Spielkarte myCard1 = new Spielkarte();  
        Spielkarte myCard2 = new Spielkarte();  
        myCard1.farbe = "kreuz";  
        myCard1.wert = "as";  
        myCard2.farbe = "karo";  
        myCard2.wert = "10";  
    }  
}
```

- Ansatz:
  - Wir wollen die Applikation AppTestSpielkarte so erweitern, dass an myCard1 und myCard2 jeweils die Botschaft toString() geschickt wird.

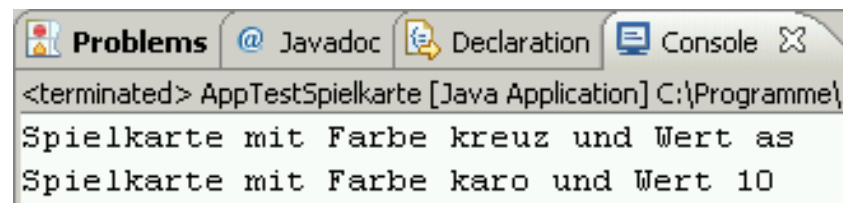
# Test von `toString()`

- Anmerkung:
  - Der "Aufruf einer Methode" wird auch "Senden einer Botschaft" genannt seit die Objektorientierung – vor ca. 25 Jahren – erfunden wurde.

- Test 1: Mit AppTestSpielkarte, Schritt 1:

```
System.out.println(myCard1.toString());  
System.out.println(myCard1.toString());
```

- `myCard1` und `myCard2` erhalten den Methodenaufruf (die Botschaft) `toString()`,
- führen die Methode aus,
- geben den Rückgabewert (den String, der sie selbst beschreibt) zurück an den Aufrufer, nämlich die Applikation `AppTestSpielkarte`.
- Diese gibt den String mit `System.out.println(...)` auf der Konsole aus.



Problems @ Javadoc Declaration Console X

```
<terminated> AppTestSpielkarte [Java Application] C:\Programme\  
Spielkarte mit Farbe kreuz und Wert as  
Spielkarte mit Farbe karo und Wert 10
```



# Test von toString()

- Test 2: Mit JUnit, Erinnerung:
  - In der JUnit-Testklasse TestSpielkarte ist bis jetzt eine Test-Methode `testNewWriteRead()` definiert:

```
public class TestSpielkarte extends TestCase {  
    public void testNewWriteRead(){  
        Spielkarte myCard = new Spielkarte();  
        myCard.farbe = "kreuz";  
        myCard.wert = "as";  
        assertEquals(myCard.farbe, "kreuz");  
        assertEquals(myCard.wert, "as");  
    }  
}
```

- Ansatz:
  - Wir wollen in dieser Testklasse eine Test-Methode `testToString()` definieren, die die Erzeugung des beschreibenden Strings testet.

# Test von toString()

- Test 2: Mit JUnit, Schritt 1:

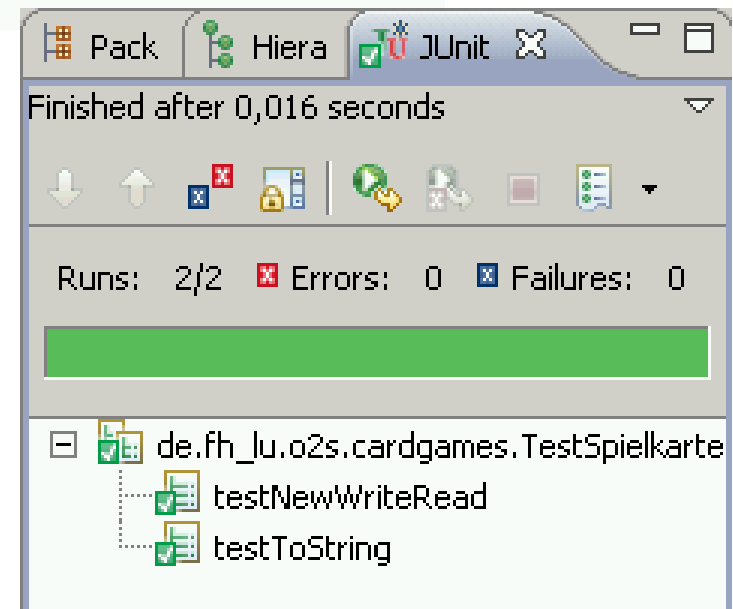
```
public class TestSpielkarte extends TestCase {  
    ...
```

```
    public void testToString() {  
        Spielkarte myCard = new Spielkarte();  
        myCard.farbe = "kreuz";  
        myCard.wert = "as";  
        assertEquals(myCard.toString(),  
            "Spielkarte mit Farbe kreuz und Wert as");  
    }  
}
```

- Anmerkung:

- Hier wird zurzeit nur eine einzige Wertekombination ("kreuz", "as") getestet...

- Testausführung durch
  - Rechter Mausklick auf die Testklasse "TestSpielkarte"
- Run As → JUnit Test



# Warum???

- Wir haben der Klasse Spielkarte – und damit allen Spielkarten-Objekten – eine Methode `toString()` gebaut,
  - weil wir dann **nur einmal** einen beschreibenden String programmieren müssen und andere Programme immer darauf zugreifen können.
- Wir haben `toString()` innerhalb der Klasse Spielkarte gebaut,
  - weil die Klasse Spielkarte am Besten weiß, wie ihre Objekte dargestellt werden sollten.

# Anmerkungen zu `toString()`

- Anmerkungen:
  - `toString()` ist ein Standard in Java. Es wird davon ausgegangen, dass alle Objekte aller Klassen eine `toString()`-Methode haben.
  - Wenn ein Objekt keine `toString()` Methode hat, kann man ihm trotzdem eine `toString()`-Botschaft schicken und erhält dann einen technischen Wert, nämlich eine Referenz auf das Objekt im Hauptspeicher.
  - Seit der Java SE Version 1.5 bzw. Java SE Version 5, wird ein Objekt `obj` automatisch in einen String verwandelt, wenn
    - ein Aufruf `System.out.println(obj);` erfolgt,
    - das Objekt an einen String angehängt werden soll, z.B.  
`"Mein Objekt ist " + obj;`
  - In den genannten Fällen wird dem Objekt `obj` implizit (intern) die `toString()`-Methode geschickt.

- In unterschiedlichen Situationen
  - werden ggfs. unterschiedliche String-Darstellungen benötigt.
- Für eine Spielkarte
  - würde sich z.B. einfach die Darstellung "Kreuz As" oder "Karo 10" anbieten.

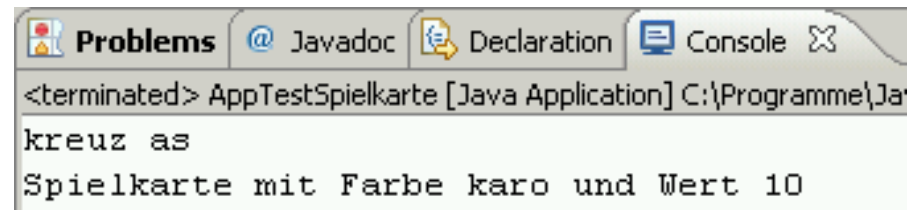
- Dafür wäre eine zusätzliche Java-Methode `toStringKurz()` sinnvoll:

```
public String toStringKurz(){  
    return farbe + " " + wert;  
}
```

- In einer Applikation müsste diese Methode dann explizit aufgerufen werden:

```
System.out.println(myCard1.toStringKurz());  
System.out.println(myCard2);
```

- Denn der implizite Aufruf, z.B. `System.out.println(myCard2)` ruft weiterhin die Standard-Methode `toString()` auf:



```
<terminated> AppTestSpielkarte [Java Application] C:\Programme\Ja  
kreuz as  
Spielkarte mit Farbe karo und Wert 10
```

# Methodenaufruf

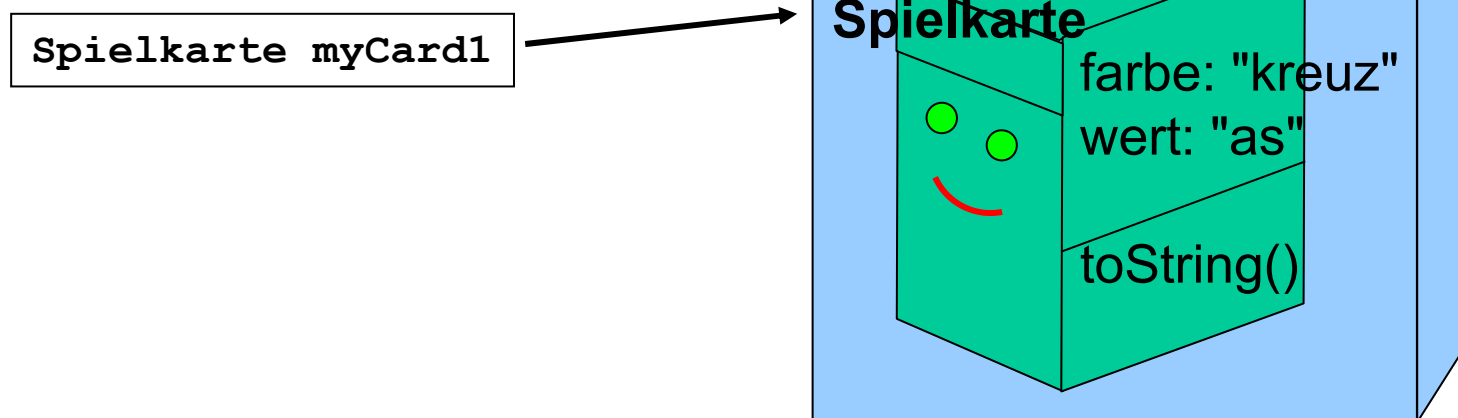
- Zum einem Methodenaufruf gehören immer zwei:
  - Der Aufrufer einer Methode,
  - ein Objekt, das diese Methode besitzt (Zielobjekt).
- Beispiel: 

```
System.out.println(myCard1.toString());
```

  - Aufrufer ist die Applikation `AppTestSpielkarte`
  - Aufgerufen wird die Methode `toString()`
  - Zielobjekt des Aufrufs ist das Spielkarten-Objekt, das mit  

```
Spielkarte myCard1 = new Spielkarte();
```

im Hauptspeicher erzeugt wurde und auf das die Variable `myCard1` zeigt.

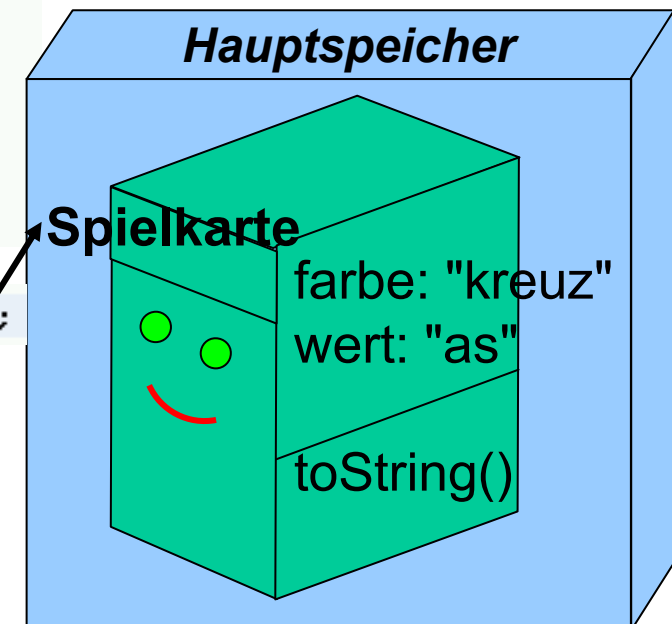


# Genauer

- Im genannten Beispiel ist der Aufrufer
  - die `main()`-Methode der Applikation `AppTestSpielkarte`
- Allgemeiner:
  - Es gibt ein aufrufendes Objekt, hier:  
die Applikation `AppTestSpielkarte` und
  - eine aufrufende Methode, hier: `main()`

```
public class AppTestSpielkarte {  
  
    public static void main(String[] args) {  
        Spielkarte myCard1 = new Spielkarte();  
        Spielkarte myCard2 = new Spielkarte();  
        myCard1.farbe = "kreuz";  
        myCard1.wert = "as";  
        ...  
        System.out.println(myCard1.toString());  
        ...  
    }  
}
```

`Spielkarte myCard1`



## druckDich()

- Aufgabe: Entwickeln Sie in der Klasse Spielkarte eine Methode `druckDich()`, mit
  - Sichtbarkeit `public` und
  - ohne Rückgabewert (`void`),
  - die nur den Befehl enthält, die Spielkarte auf der Konsole auszugeben.
- Lösung:

```
public void druckDich(){  
    System.out.println(this.toString());  
}
```



## druckDich()

- In der Applikation `AppTestSpielkarte`
  - kann `druckDich()` folgendermaßen genutzt werden:

```
//      System.out.println(myCard1.toString());  
myCard1.druckDich();
```

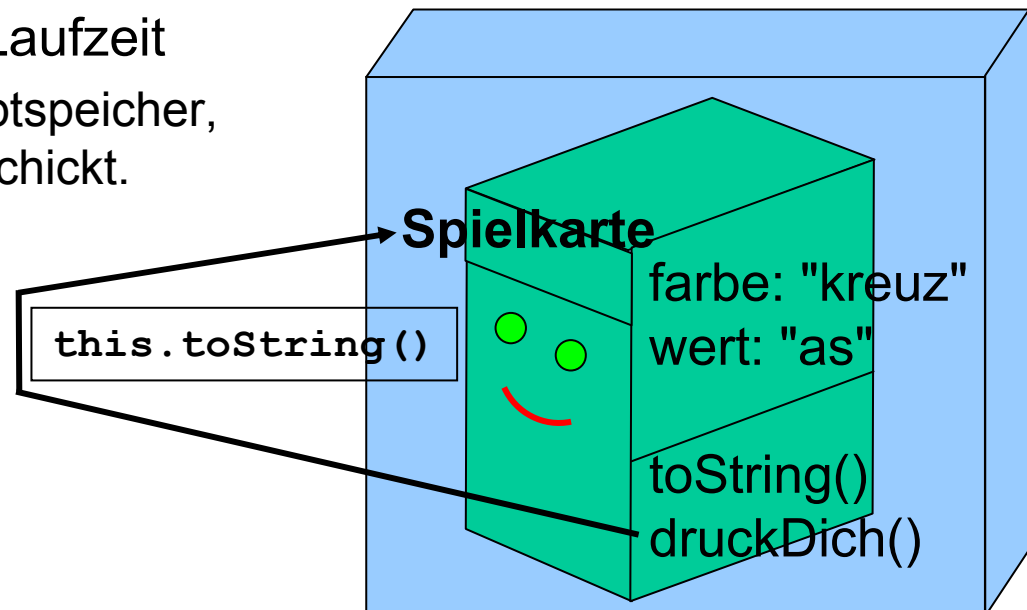
- Anmerkung:
  - Zur Anzeige einer Spielkarte `sk` kann jetzt anstatt jeweils `System.out.println(sk.toString())` zu schreiben auch kürzer (!) `sk.druckDich()` aufgerufen werden.
  - Schließlich weiß die Spielkarte am besten, wie sie sich auszugeben hat, oder?

# this

- Die Objektreferenz `this` bedeutet,

```
public void druckDich(){  
    System.out.println(this.toString());  
}
```

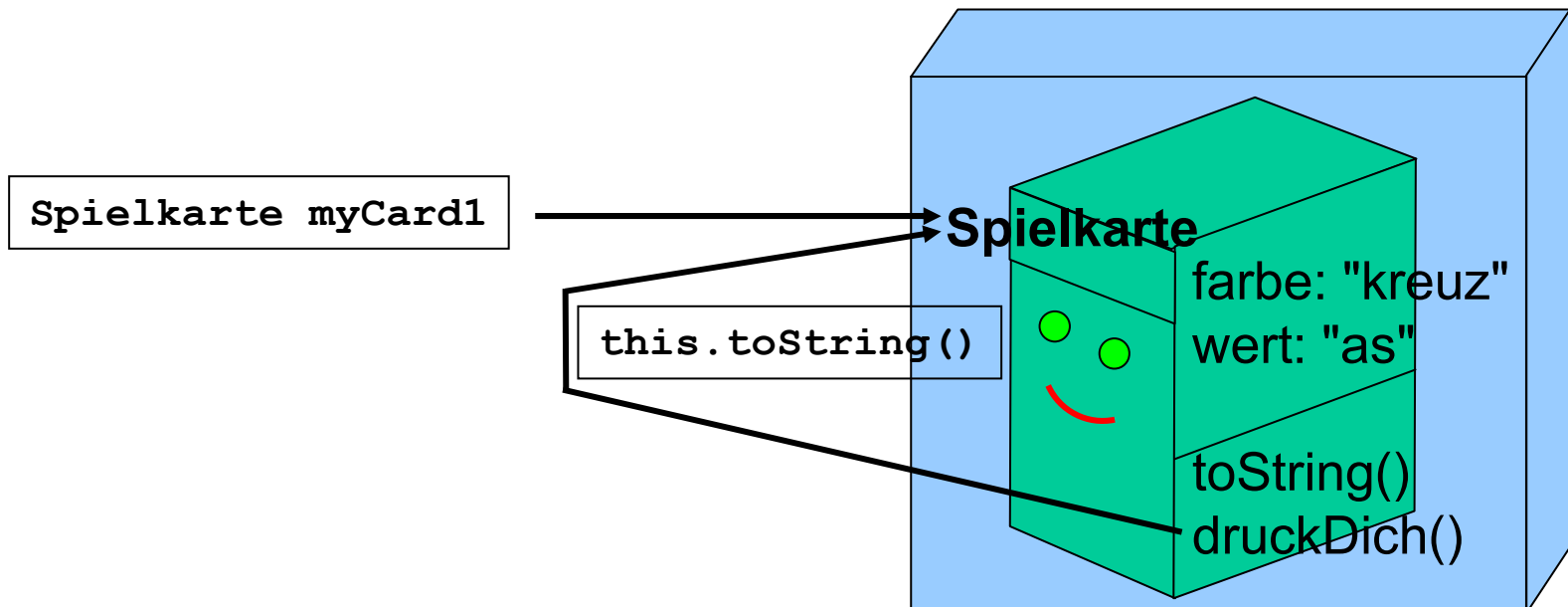
- dass das aufrufende Objekt die Botschaft, also den Methodenaufruf an sich selbst schickt.
  - dass also das Zielobjekt dasselbe Objekt ist wie das aufrufende Objekt.
- Wenn im Programmcode einer Klasse `this` verwendet wird, bezeichnet dies zur Laufzeit
    - das Objekt im Hauptspeicher, das die Botschaft schickt.



# Methodenaufruf

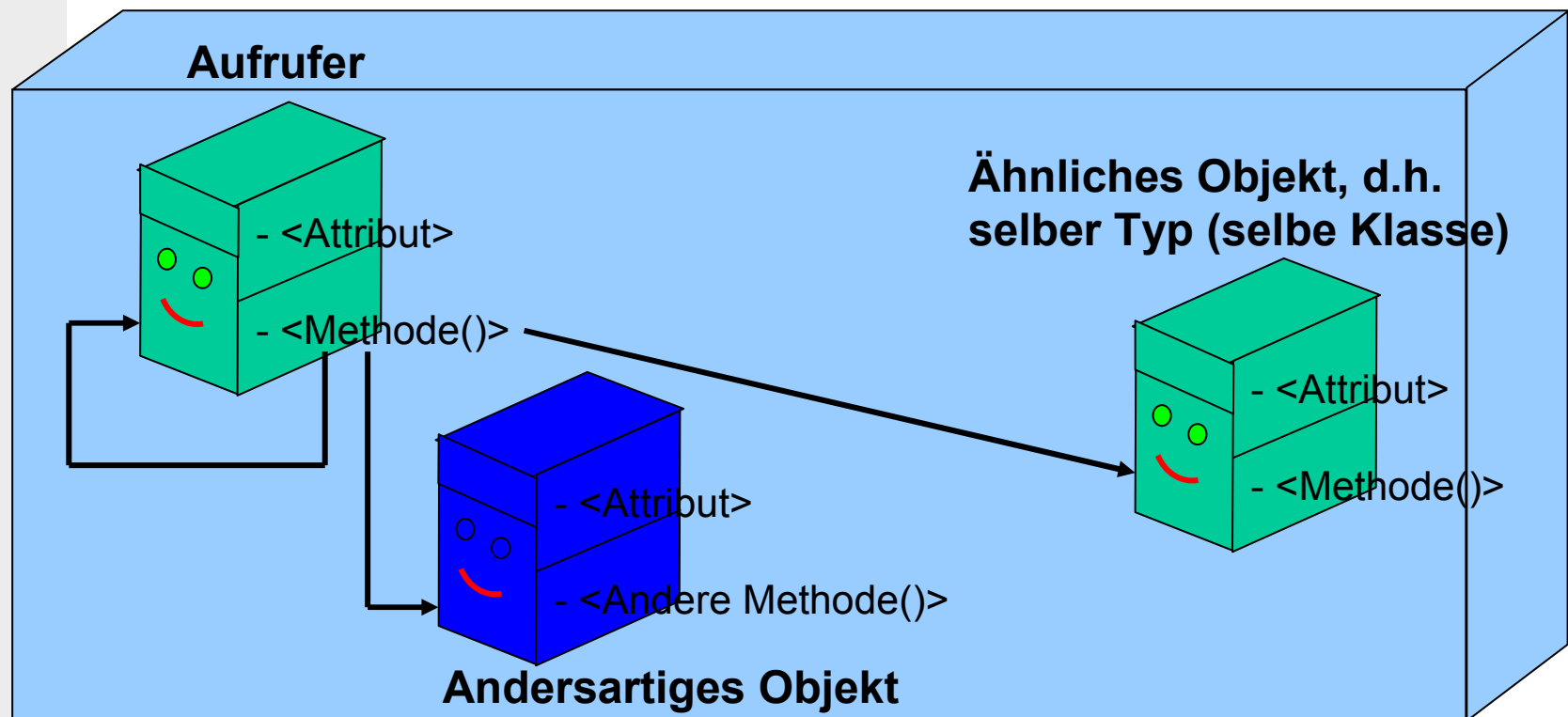
- Hierbei erfolgen die Methodenaufrufe

| Methode     | Aufrufer:<br>Methode / Objekt          | Zielobjekt                                                      |
|-------------|----------------------------------------|-----------------------------------------------------------------|
| druckDich() | main()-Methode in<br>AppTestSpielkarte | myCard1, bzw. das Spielkarten-<br>Objekt, auf das myCard1 zeigt |
| toString()  | druckDich()-<br>Methode in<br>myCard1  | myCard1,<br>adressiert durch <b>this.toString()</b>             |



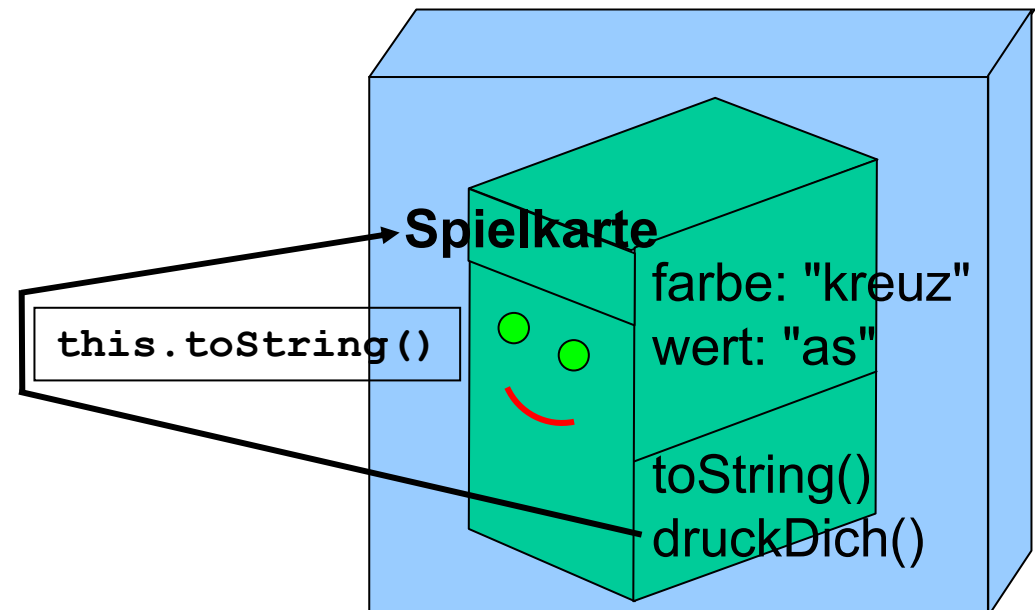
# Genauer (2)

- Innerhalb eines Methodenaufrufs ist der Aufrufer
  - die main()-Methode einer Applikation (s.o.) oder
  - eine Methode eines Objekts, die eine (andere) Methode eines (anderen) Objekts aufruft.
  - Es kann aber auch eine Methode des selben Objekts aufgerufen werden oder sogar dieselbe Methode (Rekursion).



# this weglassen?

- Wenn man einen Methodenaufruf programmiert,
  - ohne hinzuschreiben, welches Objekt die Botschaft erhalten soll,
  - z.B. `System.out.println(toString());`
  - dann geht die Botschaft automatisch an `this`
- Trotzdem ist es sehr empfehlenswert,
  - `this` jedesmal hinzuschreiben, damit immer ganz klar ist, wer der Empfänger der Botschaft sein soll.



# get-Methoden

- Erinnerung:
  - Eine Methode entspricht einer bestimmten Fähigkeit gleichartiger Objekte.
  - Die Fähigkeit wird in der Klasse der Objekte implementiert.
- Eine get-Methode
  - entspricht der Fähigkeit, einen Wert zu liefern.
  - Allgemeiner: ... der Fähigkeit, ein Objekt zurückzugeben.
- Beispiel: Spielkarten sollten in der Lage sein,
  - ihre Farbe zu sagen,
  - ihren (aufgedruckten) Wert zu sagen, z.B. "7", "Bube", etc.
  - ihren Punktwert zu sagen (König = 4, Bube = 2, etc.)

# get-Methoden

- Aufgabe: Entwickeln Sie in der Klasse Spielkarte
  - eine Methode `getFarbe()`, die die Farbe der Spielkarte zurückgibt,
  - eine Methode `getWert()`, die den aufgedruckten Wert der Spielkarte zurückgibt (s.o.),
  - eine Methode `getPunktwert()`, die den Wert der Spielkarte in Punkten zurückgibt (s.o.).
- Lösungsansatz:
  - Alle Methoden haben Sichtbarkeitsbereich `public` und keinen Eingabeparameter.
  - `getFarbe()` und `getWert()` haben den Rückgabedatentyp `String`, `getPunktwert()` hat den Rückgabedatentyp `int`.
  - `getFarbe()` und `getWert()` ermitteln den Rückgabewert, indem sie einfach auf die Attribute `farbe` / `wert` zugreifen.
  - Zur Ermittlung des jeweiligen Punktwertes ist ein bisschen Code erforderlich.

# get-Methoden

- Lösung:

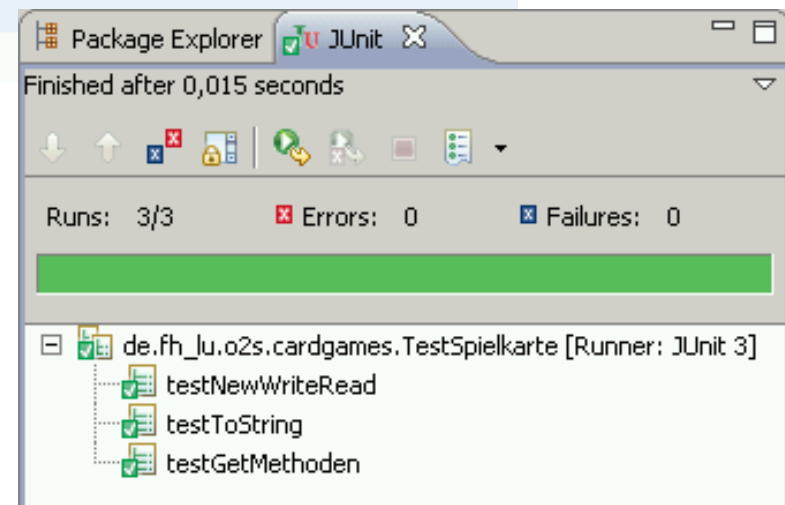
```
public String getFarbe(){
    return this.farbe;
}
public String getWert(){
    return this.wert;
}
public int getPunktWert(){
    int myVal = 0;
    String myWert = this.getWert();
    if (myWert == "2") myVal = 2;
    else if (myWert == "3") myVal = 3;
    else if (myWert == "4") myVal = 4;
    else if (myWert == "5") myVal = 5;
    else if (myWert == "6") myVal = 6;
    else if (myWert == "7") myVal = 7;
    else if (myWert == "8") myVal = 8;
    else if (myWert == "9") myVal = 9;
    else if (myWert == "10") myVal = 10;
    else if (myWert == "Bube") myVal = 2;
    else if (myWert == "Dame") myVal = 3;
    else if (myWert == "König") myVal = 4;
    else if (myWert == "As") myVal = 11;
    return myVal;
}
```



- Test mit JUnit: Neue Testmethode in unserer Testklasse TestSpielkarte

```
public class TestSpielkarte extends TestCase {  
    ...  
    public void testGetMethoden(){  
        Spielkarte myCard = new Spielkarte();  
        myCard.farbe = "kreuz";  
        myCard.wert = "as";  
        assertEquals(myCard.getFarbe(), "kreuz");  
        assertEquals(myCard.getWert(), "as");  
        assertEquals(myCard.getPunktwert(), 11);  
    }  
}
```

- Testausführung durch
  - Rechter Mausklick auf die Testklasse "TestSpielkarte"
  - Run As → JUnit Test



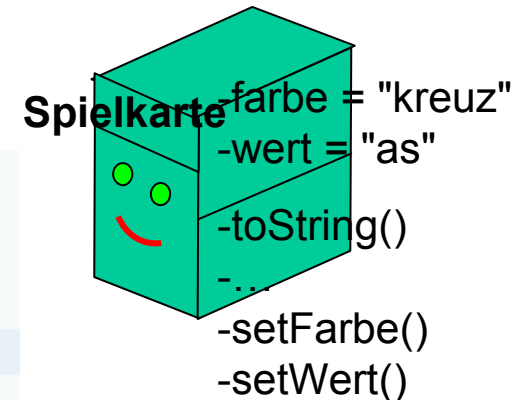
# set-Methoden

- set-Methoden
  - verändern den (inneren) Zustand eines Objekts,
  - z.B. indem sie den Wert eines Attributs ändern.
- Aufgabe: Entwickeln Sie in der Klasse Spielkarte
  - eine Methode `setFarbe(String farbe)`, die die Farbe der Spielkarte ändert,
  - eine Methode `setWert(String wert)`, die den aufgedruckten Wert der Spielkarte ändert,
- Lösungsansatz:
  - Beide Methoden haben
    - den Sichtbarkeitsbereich `public`,
    - den Rückgabedatentyp `void` und
    - als Eingabeparameter den neu zu setzenden String.
  - Zur Implementierung ist der Eingabeparameter in das jeweilige Attribut zu übernehmen.

# set-Methoden

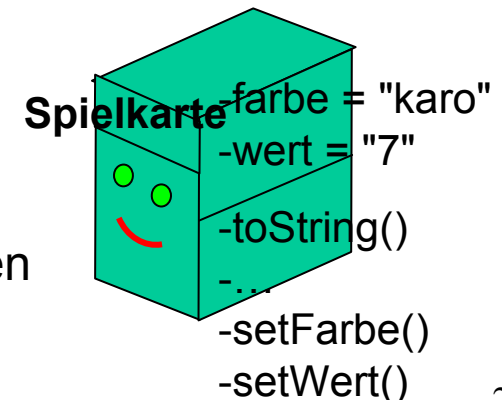
## ■ Lösung:

```
public void setFarbe(String farbe){  
    this.farbe = farbe;  
}  
public void setWert(String wert){  
    this.wert = wert;  
}
```



## ■ Anmerkung:

- In `setFarbe(...)` kommen zwei Variablen des Namens "farbe" vor:
  - `farbe` als Eingabeparameter – diese Variable gilt nur innerhalb dieser einen Methode, dies ist eine lokale Variable
  - `this.farbe` bezeichnet das Attribut des Objekts – diese Variable gilt innerhalb der gesamten Spielkarte (dieses einen Spielkarten-Objekts).
- In einem anderen Spielkarten-Objekt gelten aber andere Werte für `this.farbe` und `this.wert`.

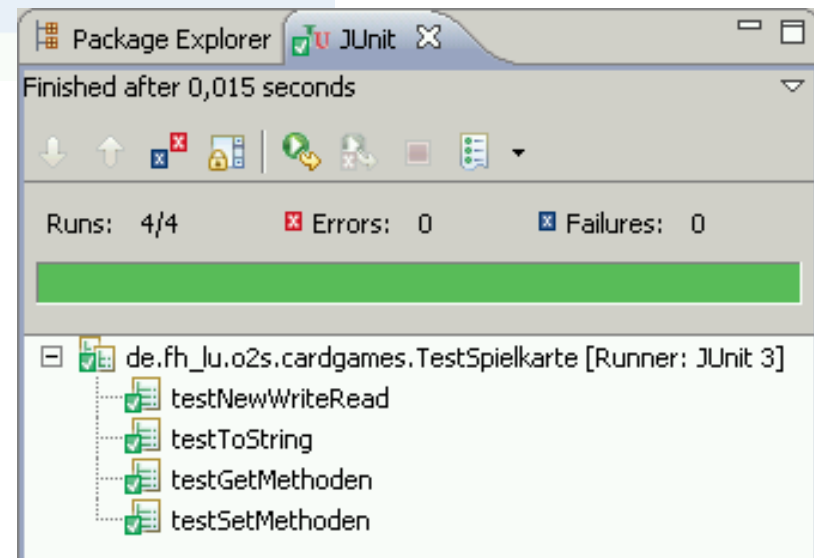


# Test

- Test mit JUnit: Neue Testmethode in unserer Testklasse TestSpielkarte

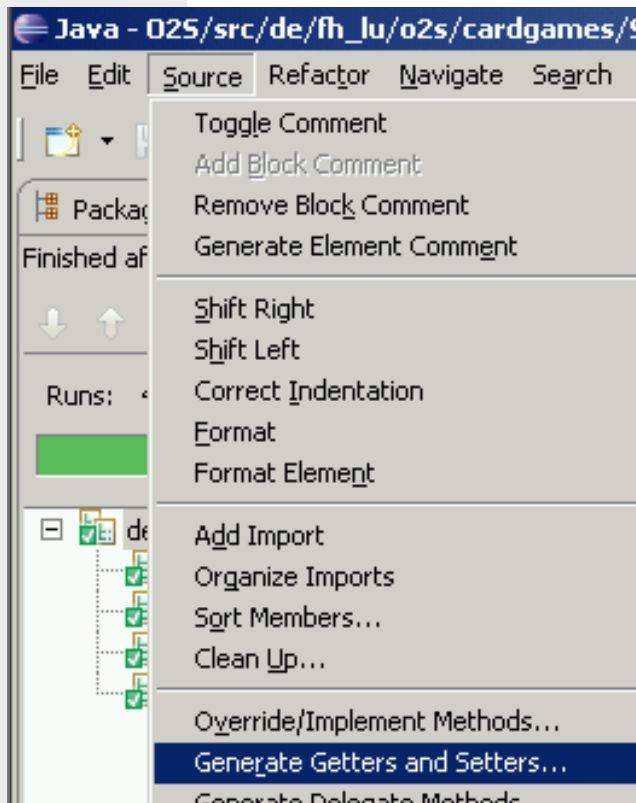
```
public class TestSpielkarte extends TestCase {  
    ...  
    public void testSetMethoden() {  
        Spielkarte myCard = new Spielkarte();  
        myCard.setFarbe("Kreuz");  
        myCard.setWert("As");  
        assertEquals(myCard.getFarbe(), "Kreuz");  
        assertEquals(myCard.getWert(), "As");  
    }  
}
```

- Testausführung durch
  - Rechter Mausklick auf die Testklasse "TestSpielkarte"
  - Run As → JUnit Test



# Standard-Methoden get-/set-

- Soweit get- und set-Methoden tatsächlich nur dazu dienen,
  - Werte aus Attributen zu lesen bzw.
  - Attributwerte zu setzen,
  - spricht man von Standard-Methoden.



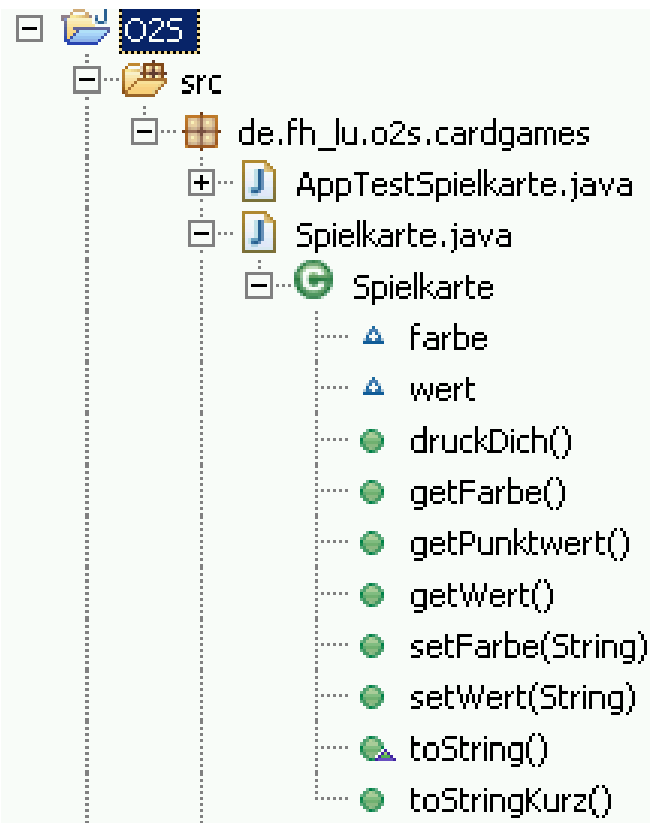
- Diese sind in vielen Klassen definiert, deshalb unterstützt Eclipse deren Implementierung
  - mit einer eigenen Funktion, nämlich im Menü Source
    - Generate Getters and Setters...

# Namenskonventionen

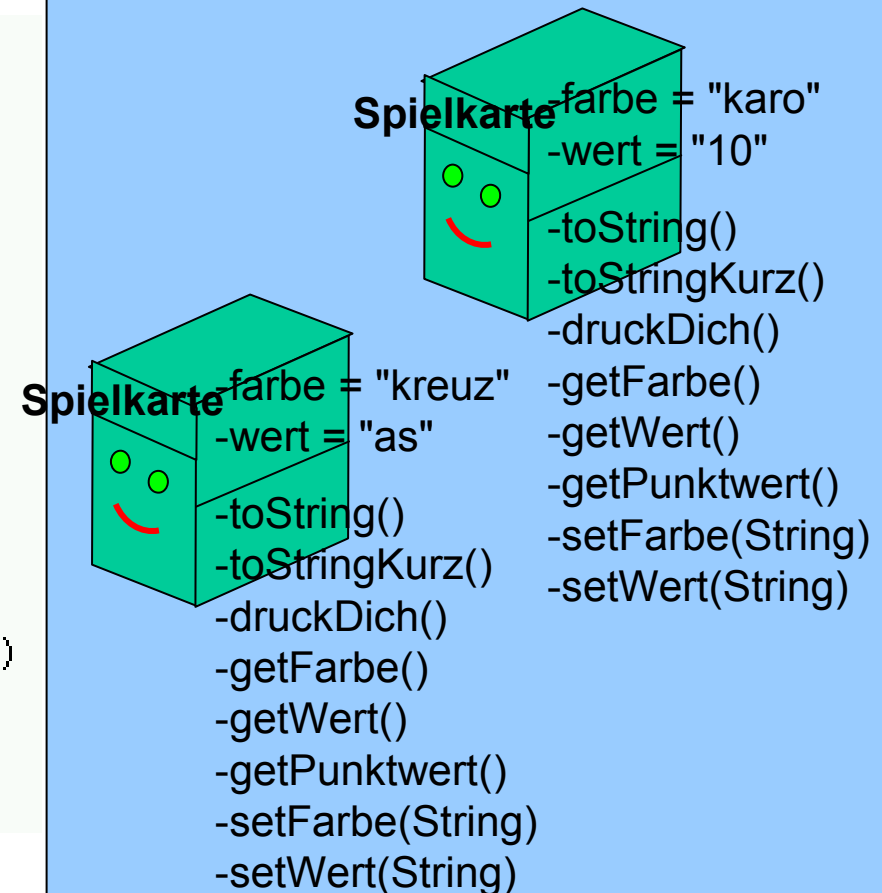
- Mit Kleinbuchstaben beginnen
  - Attribute und Methodennamen, z.B. `farbe` / `getFarbe()`.
- Mit Großbuchstaben beginnen
  - Klassennamen, z.B. `Spielkarte`.
- Namen sollen "sprechend" sein.
- beim Zusammensetzen von Namen werden in der Mitte Großbuchstaben verwendet, z.B. `getFarbe()`
  - Alternative: Manchmal auch `get_farbe()` oder `get-farbe()`
- Lokale Variablen oder Parameter dürfen genauso heißen wie Attribute.
  - Dadurch werden die Attribute überdeckt.
  - Durch Nutzung von `this` kann trotzdem auf die Attribute zugegriffen werden, vgl. "set-Methoden".

# Zwischenstand

Klasse (Modell,  
Vorlage, Schablone)



Objekte (existieren konkret  
im Hauptspeicher)



# class vs. record / struct

- In prozeduralen Programmiersprachen (3rd generation language, 3GL)
  - konnten bereits Datenstrukturen als Zusammenfassung von Variablen definiert werden,
  - Z.B. in Pascal: record, in C: struct
- Getrennt davon
  - konnten funktionale Blöcke definiert werden: Unterprogramme / Prozeduren / Funktionen
- Bei Änderung der Daten mussten immer alle funktionalen Blöcke auf einen Änderungsbedarf geprüft werden.
- Die Besonderheit der Objektorientierung besteht darin, dass Daten und Funktionen in Klassen zusammengefasst werden:
  - Daten in Form von Attributen
  - Methoden, die mit den Daten arbeiten / auf die Daten wirken
- Im Idealfall wirken sich Datenänderungen nur auf die Methoden einer oder weniger Klassen aus.



# Geheimnisprinzip

- Ein objektorientiertes System funktioniert durch
  - Erzeugung von Objekten im Hauptspeicher (new ...)
  - Aufruf von Methoden dieser Objekte
- Jedes Objekt
  - hat "öffentliche" Fähigkeiten (Methoden), die von anderen Objekten genutzt werden dürfen.
  - kann "private" Fähigkeiten haben, die nicht von anderen Objekten genutzt werden dürfen
- Die öffentliche Menge von Methoden wird genannt
  - „Protokoll“,
  - API (Application Programmer's Interface)
  - oder einfach „Interface“ bzw. "öffentliches Interface" der Klasse.

# Geheimnisprinzip

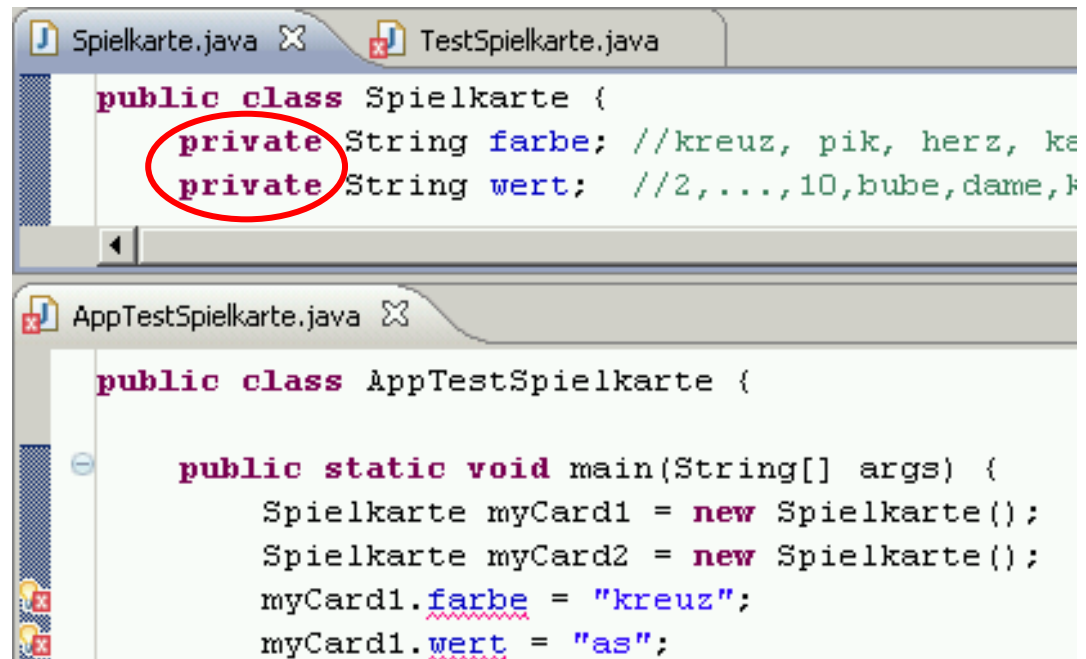
- Die Attribute (Daten / Werte / Eigenschaften) von Objekten sind in der Regel nicht öffentlich, denn
  - wenn Attribute von außen zugreifbar sein sollen, dann kann eine Fähigkeit zum Zugriff implementiert werden → get-Methode
  - wenn Attribute von außen änderbar sein sollen, dann kann eine Fähigkeit zur Änderung implementiert werden → set-Methode
  - wenn sich an der Datenstruktur etwas ändert,
    - würden Programme, die direkt auf die Attribute zugreifen, ins Leere laufen.
    - Wenn ein direkter Zugriff auf Attribute ausgeschlossen ist, reicht es, bei Änderungen darauf zu achten, dass das öffentliche Interface der Klasse weiterhin funktioniert.

# Geheimnisprinzip

- Jedes Attribut und jede Methode hat eine Sichtbarkeit.
  - "public" bedeutet, dass von jedem Java-Objekt darauf zugegriffen werden kann
  - "private" bedeutet, dass nur von Objekten derselben Klasse darauf zugegriffen werden kann.
  - Wenn keine Sichtbarkeit angegeben wird, können alle Objekte, deren Klasse im selben Package definiert ist, darauf zugreifen.

- Im Beispiel  
Spielkarte:

- AppTestSpielkarte und TestSpielkarte funktionieren dann nicht mehr.



```
public class Spielkarte {  
    private String farbe; //kreuz, pik, herz, ka  
    private String wert; //2,...,10,bube,dame,k  
}  
  
public class AppTestSpielkarte {  
  
    public static void main(String[] args) {  
        Spielkarte myCard1 = new Spielkarte();  
        Spielkarte myCard2 = new Spielkarte();  
        myCard1.farbe = "kreuz";  
        myCard1.wert = "as";  
    }  
}
```

# Geheimnisprinzip

- Das öffentliche Interface der Klasse `Spielkarte` besteht damit aus den Methoden `toString()`, `toStringKurz()`, `druckDich()`, `getFarbe()`, `setFarbe(String)`, `getWert()`, `setWert(String)`, `getPunktwert()`
- Damit `AppTestSpielkarte` und `TestSpielkarte` wieder laufen, müssen wir zum Zugriff auf die Attribute die get- und set-Methoden verwenden.

```
public class AppTestSpielkarte {  
  
    public static void main(String[] args) {  
        Spielkarte myCard1 = new Spielkarte();  
        Spielkarte myCard2 = new Spielkarte();  
        myCard1.setFarbe("kreuz");  
        myCard1.setWert("as");  
    }  
}
```

# JUnit-Test

- Für `TestSpielkarte` nutzen wir ein weiteres Feature von JUnit.
- Aufgabe:
  - Reparieren Sie Ihren JUnit-Test `TestSpielkarte`
  - Nutzen Sie außerdem die `setUp()`-Methode um Ihren JUnit-Test zu vereinfachen.
- Beobachtung:
  - In jeder Testmethode wird zunächst der selbe Code ausgeführt, um eine Spielkarte zu erzeugen und mit Werten auszustatten

```
public void testGetMethoden(){  
    Spielkarte myCard = new Spielkarte();  
    myCard.farbe = "kreuz";  
    myCard.wert = "as";  
    assertEquals(myCard.getFarbe(), "kreuz");  
    assertEquals(myCard.getWert(), "as");  
    assertEquals(myCard.getPunktwert(), 11);  
}  
}
```

- Lösungsansatz:
  - Diesen Code fassen wir in einer eigenen Methode setUp() innerhalb unserer Testklasse zusammen.
  - Dabei greifen wir nicht mehr direkt auf die Attribute der Spielkarte zu, sondern verwenden die set-Methoden.
  - Anschließend durchdenken wir die einzelnen Tests.
- Lösung, Schritt 1:
  - Damit myCard in allen Testmethoden verwendet werden kann, darf dies keine lokale Variable von setUp() sein, sondern muss ein Attribut der Testklasse sein.

```
public class TestSpielkarte extends TestCase {  
    Spielkarte myCard;  
    public void setUp(){  
        myCard = new Spielkarte();  
        myCard.setFarbe("Kreuz");  
        myCard.setWert("As");  
    }  
}
```

- Lösung, Schritt 2:
  - Die Testmethode `testNewWriteRead()` wird überflüssig, weil die dort getesteten Zugriffe nicht mehr funktionieren.
  - Die Testmethode `testSetMethoden()` wird mit der `setUp()`-Methode überflüssig, weil dort die set-Methoden bereits verwendet und damit auch getestet werden.
  - Die Testmethoden `testToString()` und `testGetMethoden()` können auf die assert-Methoden reduziert werden:

```
public void testToString(){
    assertEquals(myCard.toString(),
        "Spielkarte mit Farbe Kreuz und Wert As");
}

public void testGetMethoden(){
    assertEquals(myCard.getFarbe(), "Kreuz");
    assertEquals(myCard.getWert(), "As");
    assertEquals(myCard.getPunktWert(), 11);
}
```

# Kapselung

- Man spricht von „Kapselung“, wenn
  - Attribute und Funktionalitäten so in Klassen zusammengefasst sind, dass daraus sinnvolle Einheiten entstehen,
  - Fähigkeiten von Objekten derart als Methoden zur Verfügung gestellt werden, dass es nicht mehr nötig ist, in die innere Struktur der Objekte einzugreifen,
  - Attribute und private Methoden auch als `private` gekennzeichnet sind und damit der Eingriff in die innere Struktur der Objekte auch technisch verhindert wird.



# Überladen von Methoden

- Aufgabe: Entwickeln Sie in Ihrer Klasse Spielkarte
  - eine Methode toString(boolean kurz) mit booleschem Eingabeparameter kurz, so dass
    - für kurz == true die Methode toStringKurz() ausgeführt wird,
    - für kurz == false die normale Methode toString() ausgeführt wird.
- Lösung:
  - Die Methode benötigt eine Sichtbarkeit (public) und einen Rückgabetyp (String).

```
public String toString(boolean kurz){  
    String retString;  
    if (kurz == true){  
        retString = this.toStringKurz();  
    }else{  
        retString = this.toString();  
    }  
    return retString;  
}
```

oder (gleichwertig)

```
public String toString(boolean kurz){  
    if (kurz) return this.toStringKurz();  
    else return this.toString();  
}
```

# Überladen von Methoden

- Anmerkung:
  - Als **Signatur einer Methode** bezeichnet man den Namen und die Parameterdatentypen.
- Wenn in einer Klasse zwei Methoden mit demselben Namen vorkommen,
  - müssen sie eine unterschiedliche Signatur, also unterschiedliche Eingabeparameter haben und
  - man spricht vom **Überladen der Methode**.
- Beobachtung: Wir haben zwei `toString()`-Methoden mit unterschiedlicher Signatur:
  - `public String toString() {...}`
  - `public String toString(boolean kurz) {...}`

# Konstruktor

- Problem:
  - Wenn wir aufrufen `Spielkarte myCard = new Spielkarte();`
  - dann erhalten wir zunächst ein Objekt vom Typ `Spielkarte` ohne Farbe und Wert, d.h. `farbe` und `wert` sind beide null
  - Das ist semantisch sinnlos!
- Ziel:
  - Ein Objekt soll nach der Erzeugung sofort sinnvoll sein.
- Aufgabe:
  - Eine Methode soll bereitgestellt werden, die das Objekt vom ersten Moment an sinnvoll macht.
  - Diese soll bei der Erzeugung des Objekts automatisch aufgerufen werden.

- Lösungsansatz: Konstruktor
  - Heißt genau wie die Klasse selbst,
  - Wird bei `new ...` ausgeführt.
  - beginnt auch mit einem Großbuchstaben, obwohl es eine Methode ist,
  - wird ohne Rückgabewert definiert; gibt immer das Objekt selbst zurück.
  - Kann mit oder ohne Eingabeparameter definiert werden, z.B.
    - `public Spielkarte(){...}`
    - `public Spielkarte(String farbe, String wert){...}`
  - Wenn die Spielkarte von Anfang sinnvoll sein soll, müssen Farbe und Wert angegeben werden, wir benötigen also einen Konstruktor mit Eingabeparametern.

# Konstruktor

- Lösung:

```
public class Spielkarte {  
    private String farbe; //kreuz, pik, herz, karo  
    private String wert;  //2,...,10,bube,dame,koen  
  
    public Spielkarte(String farbe, String wert){  
        this.farbe = farbe;  
        this.wert = wert;  
    }  
}
```

- Test:
  - Wir benutzen den Konstruktor in unserem JUnit-Test
  - und zwar da wo eine Spielkarte erzeugt wird: In der setUp()-Methode:

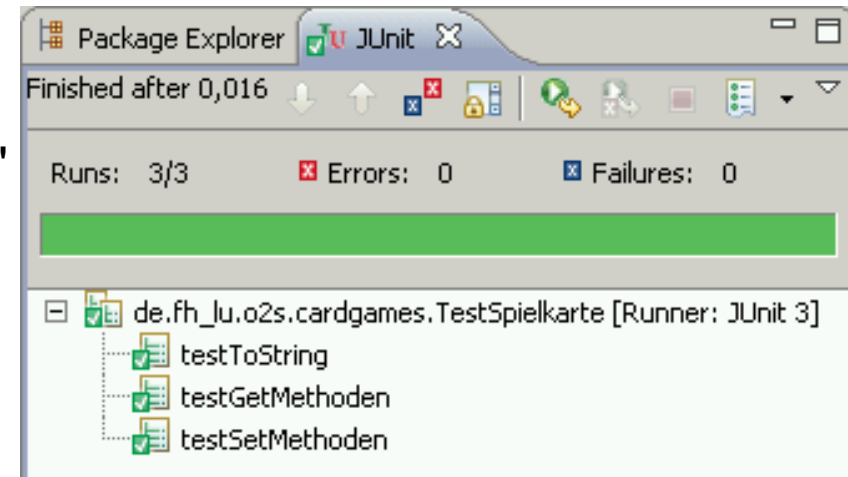
```
public class TestSpielkarte extends TestCase {  
    Spielkarte myCard;  
    public void setUp() {  
        myCard = new Spielkarte("Kreuz", "As");  
        // myCard.setFarbe("Kreuz");  
        // myCard.setWert("As");  
    }  
}
```

- Allerdings haben wir dadurch den Test der set-Methoden auskommentiert,
  - wir benötigen also eine neue Test-Methode testSetMethoden():

```
public void testSetMethoden() {  
    myCard.setFarbe("Karo");  
    myCard.setWert("10");  
    assertEquals(myCard.getFarbe(), "Karo");  
    assertEquals(myCard.getWert(), "10");  
}
```

# Test

- Testausführung durch
  - Rechter Mausklick auf die Testklasse "TestSpielkarte"
  - Run As → JUnit Test



- Anmerkung:
  - Unsere AppTestSpielkarte läuft jetzt auch nicht mehr:

```
public class AppTestSpielkarte {  
  
    public static void main(String[] args) {  
        Spielkarte myCard1 = new Spielkarte();  
        Spielkarte myCard2 = new Spielkarte();  
        myCard1.getFarbe("Kreuz");  
    }  
}
```

# Konstrukturen

- Beobachtung:
  - Bei der Erzeugung eines Objekts mit `new ...` wird immer (!) versucht, einen Konstruktor mit passenden Eingabeparametertypen zu finden.
  - Bei `new Spielkarte();` wird versucht, einen Konstruktor ohne Eingabeparameter zu finden → Fehler!
- Warum hat es vorhin funktioniert?
  - Wenn in einer Klasse überhaupt kein Konstruktor definiert ist,
    - dann denkt sich das System implizit (automatisch) einen Konstruktor ohne Parameter ("leerer Konstruktor" bzw. "trivialer Konstruktor")
    - dieser Konstruktor enthält keine Funktionalität
  - Wenn in einer Klasse mindestens ein Konstruktor definiert ist,
    - dann gibt es diese Automatik nicht mehr.

```
public class AppTestSpielkarte {  
  
    public static void main(String[] args) {  
        Spielkarte myCard1 = new Spielkarte();  
        Spielkarte myCard2 = new Spielkarte();  
        myCard1.getFarbe("Kreuz");  
    }  
}
```



# AppTestSpielkarte

- Was können wir tun, um den Fehler bei AppTestSpielkarte zu beheben?

```
public class AppTestSpielkarte {  
  
    public static void main(String[] args) {  
        Spielkarte myCard1 = new Spielkarte();  
        myCard1.setFarbe("kreuz");  
        myCard1.setWert("as");  
    }  
}
```

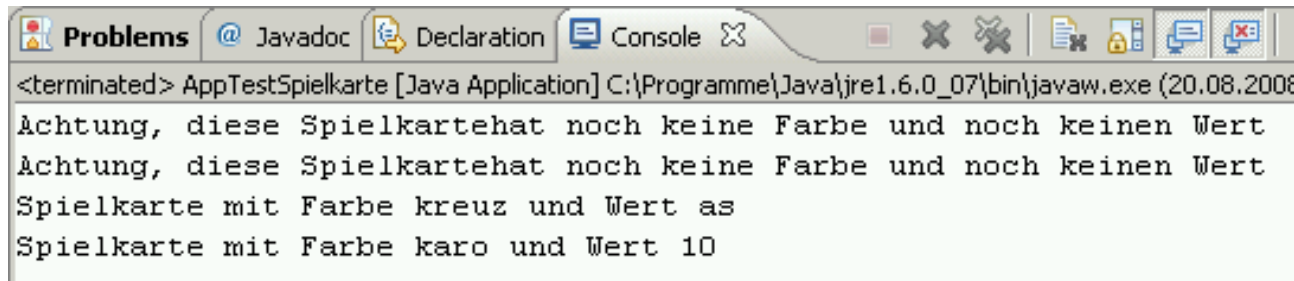
1. Wir könnten die Test-Applikation umstellen, so dass der Konstruktor mit zwei Parametern genutzt wird.
  2. Wir entwickeln einen Konstruktor `Spielkarte()` ohne Parameter
- Aufgabe: Implementieren Sie die 2. Option
  - Lösungsansatz:
    - Der leere Konstruktor benötigt keine Funktionalität
    - Wir sollten aber eine Warnung ausgeben, weil die Benutzung des leeren Konstruktors eigentlich unerwünscht ist.

# AppTestSpielkarte

- Lösung:

```
public class Spielkarte {  
    private String farbe; //kreuz, pik, herz, karo  
    private String wert;  //2,...,10,bube,dame,koenig,as,joker  
    public Spielkarte(){  
        System.out.println("Achtung, diese Spielkarte" +  
            "hat noch keine Farbe und noch keinen Wert");  
    }  
    public Spielkarte(String farbe, String wert){  
        this.farbe = farbe;  
    }  
}
```

- Die Ausführung von AppTestSpielkarte ergibt dann:

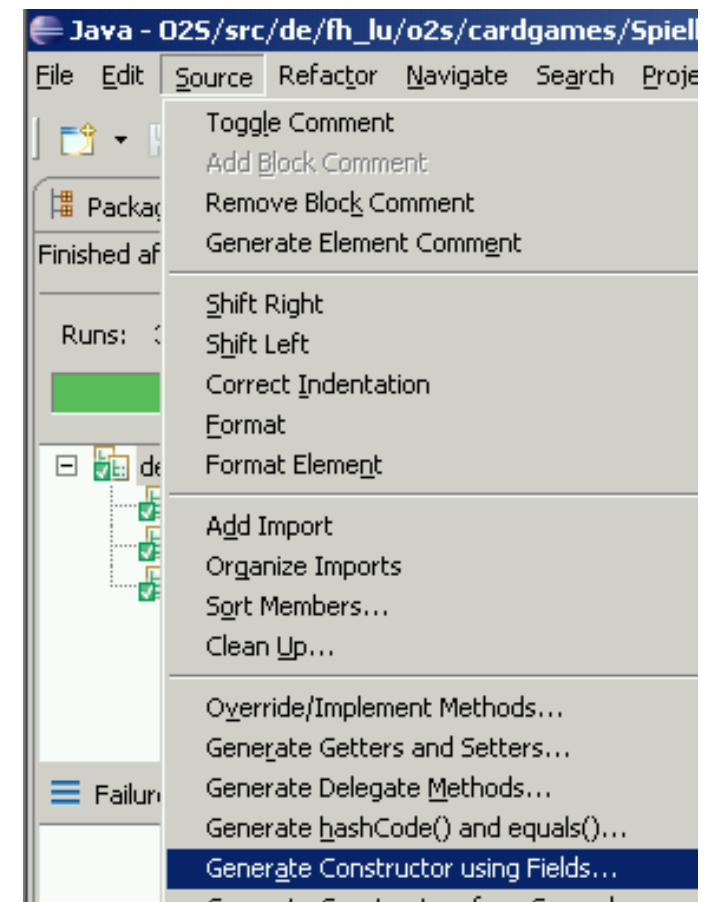


The screenshot shows a Java IDE window with the 'Console' tab selected. The title bar indicates the application is 'AppTestSpielkarte [Java Application]' running on 'C:\Programme\Java\jre1.6.0\_07\bin\javaw.exe (20.08.2008)'. The console output displays the following lines:

```
<terminated> AppTestSpielkarte [Java Application] C:\Programme\Java\jre1.6.0_07\bin\javaw.exe (20.08.2008)  
Achtung, diese Spielkartehat noch keine Farbe und noch keinen Wert  
Achtung, diese Spielkartehat noch keine Farbe und noch keinen Wert  
Spielkarte mit Farbe kreuz und Wert as  
Spielkarte mit Farbe karo und Wert 10
```

# Anmerkungen

- Die Klasse Spielkarte hat jetzt zwei Konstruktoren mit unterschiedlichen Parametern,
  - also unterschiedlichen **Signaturen**
  - der Konstruktor wurde **überladen**.
- Konstruktoren wie  
`Spielkarte(String, String)`,
  - die nur Parameter in Attribute kopieren, sind in vielen Klassen definiert,
- Deshalb unterstützt Eclipse deren Implementierung
  - mit einer eigenen Funktion, nämlich im Menü Source
    - Generate Constructor using Fields...



# Exkurs: Destruktoren

- In C++ gibt es auch noch Destruktoren, die z.B. `~Spielkarte()` heißen.
- Diese gibt es in Java nicht. Stattdessen kann jede Klasse eine Methode `public void finalize(){...}` enthalten.
- Kurz bevor ein Objekt dieser Klasse endgültig aus dem Hauptspeicher geputzt wird, wird dann dessen `finalize()`-Methode aufgerufen – falls vorhanden.
- Frage:
  - Wann wird eigentlich ein Objekt aus dem Hauptspeicher geputzt?

# Garbage Collection

- Etwas ausholen...
  - Bei objektorientierter Entwicklung werden laufend Objekte erzeugt (`new ...`) und im Hauptspeicher abgelegt.
  - Meistens wird zusätzlich eine Variable erzeugt, die auf das Objekt im Hauptspeicher zeigt.
  - Auf ein Objekt können auch mehrere (viele) Variablen zeigen.
  - Wenn auf ein Objekt keine Variable mehr zeigt, dann ist es nicht mehr erreichbar und kann gelöscht werden.

# Garbage Collection

- ... und löschen
  - Bei C++ ist es die Aufgabe des Programmierers, zu kontrollieren, wann ein Objekt nicht mehr gebraucht wird, und dies explizit zu löschen (`free()`).
  - Dies ist extrem fehleranfällig und führt bei großen Programmen dazu, dass Speicher nicht mehr freigegeben wird und deshalb der Hauptspeicher irgendwann voll ist mit "Müll".
  - Besser ist deshalb ein sogenannter „Garbage Collector“, der automatisch prüft, welche Objekte nicht mehr erreichbar sind und diese automatisch löscht.
  - Programmierer und Benutzer müssen sich nicht darum kümmern.
  - Vor dem automatischen Löschen eines Objekts wird diesem noch einmal die `finalize()`-Methode geschickt, damit es ggfs. noch eine letzte Aktion erledigen kann, z.B. das Schließen einer Datei oder einer Datenbankverbindung.

# Exkurs: init-Methoden

- Erinnerung: Konstruktoren sollen dafür sorgen, dass
  - ein Objekt nach seiner Erzeugung
  - einen "vernünftigen" Ausgangszustand einnimmt.
- Es ist denkbar, dass
  - ein Objekt später in den Ausgangszustand zurückversetzt werden soll,
  - ohne dass ein neues Objekt erzeugt werden soll, es kann also kein Konstruktoraufruf erfolgen.
- Dafür haben sich init()-Methoden etabliert (zur "Initialisierung").
  - Diese haben häufig dieselbe Funktionalität wie ein Konstruktor, z.B.

```
public void init(String farbe, String wert){  
    this.farbe = farbe;  
    this.wert = wert;  
}
```

# Exkurs: init-2

- In komplexeren Beispielen ist es üblich, dass dieselbe Funktionalität mit verschiedenen Methoden / Signaturen zur Verfügung gestellt wird, z.B.

```
public void init(String farbe, String wert){  
    this.setFarbe(farbe);  
    this.setWert(wert);  
}  
public void init(String farbe){  
    this.init(farbe, null);  
}  
public void init(){  
    this.init(null, null);  
}
```

- Dies vereinfacht den Zugriff durch einen Benutzer der Klasse und damit die Wiederverwendbarkeit.



# Exkurs init-3

- Der Konstruktor

`Spielkarte(String farbe, String wert)`

und die Methode

`init (String farbe, String wert)`

haben also dieselbe Funktionalität.

- Der Konstruktor kann deshalb die `init`-Methode aufrufen, anstatt die Funktionalität selbst ebenfalls zu programmieren:

```
public Spielkarte(String farbe, String wert){  
    this.init(farbe, wert);  
}
```

- Der leere Konstruktor könnte ebenfalls die `init`-Methode aufrufen:

```
public Spielkarte(){  
    this.init(null, null);  
}
```

# Horizontaler Konstruktoraufruf

- Wenn ein Konstruktor eine Funktionalität
  - vollständig implementiert hat,
  - die ein weiterer Konstruktor derselben Klasse nutzen möchte,
- dann kann der weitere Konstruktor den ersten Konstruktor aufrufen,

- dieser Aufruf erfolgt mit `this(...)` und den richtigen Aufrufparametern, z.B.

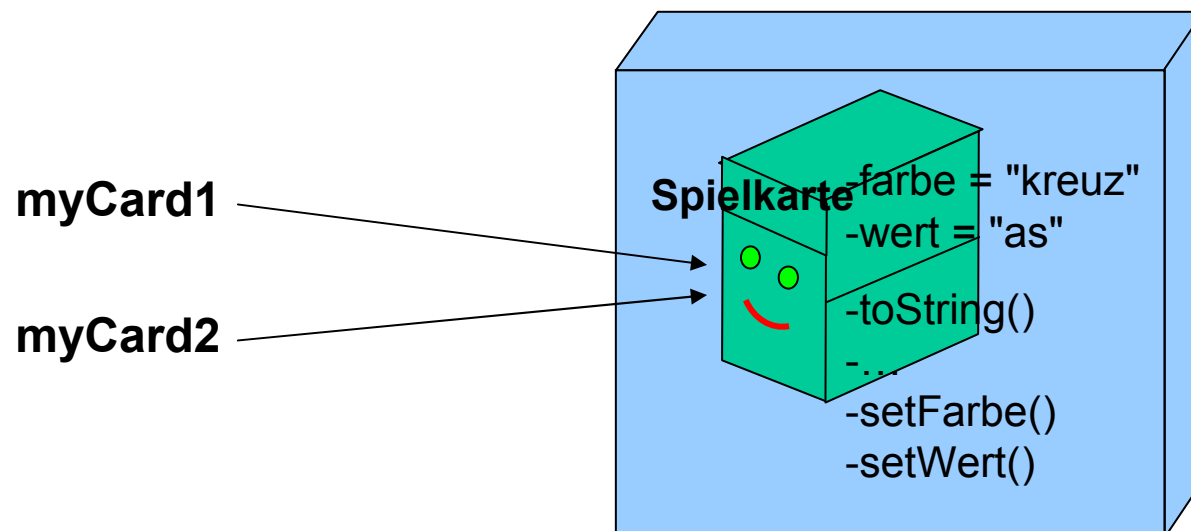
```
Spielkarte(String farbe){//Definition Spielkarte(String)
    this(farbe, dummy);//Aufruf Spielkarte(String, String)
}
```

- Anmerkung:
  - Falls eine init-Methode vorhanden ist, könnte natürlich auch diese aufgerufen werden, z.B.

```
Spielkarte(String farbe){
    this.init(farbe, dummy);
}
```

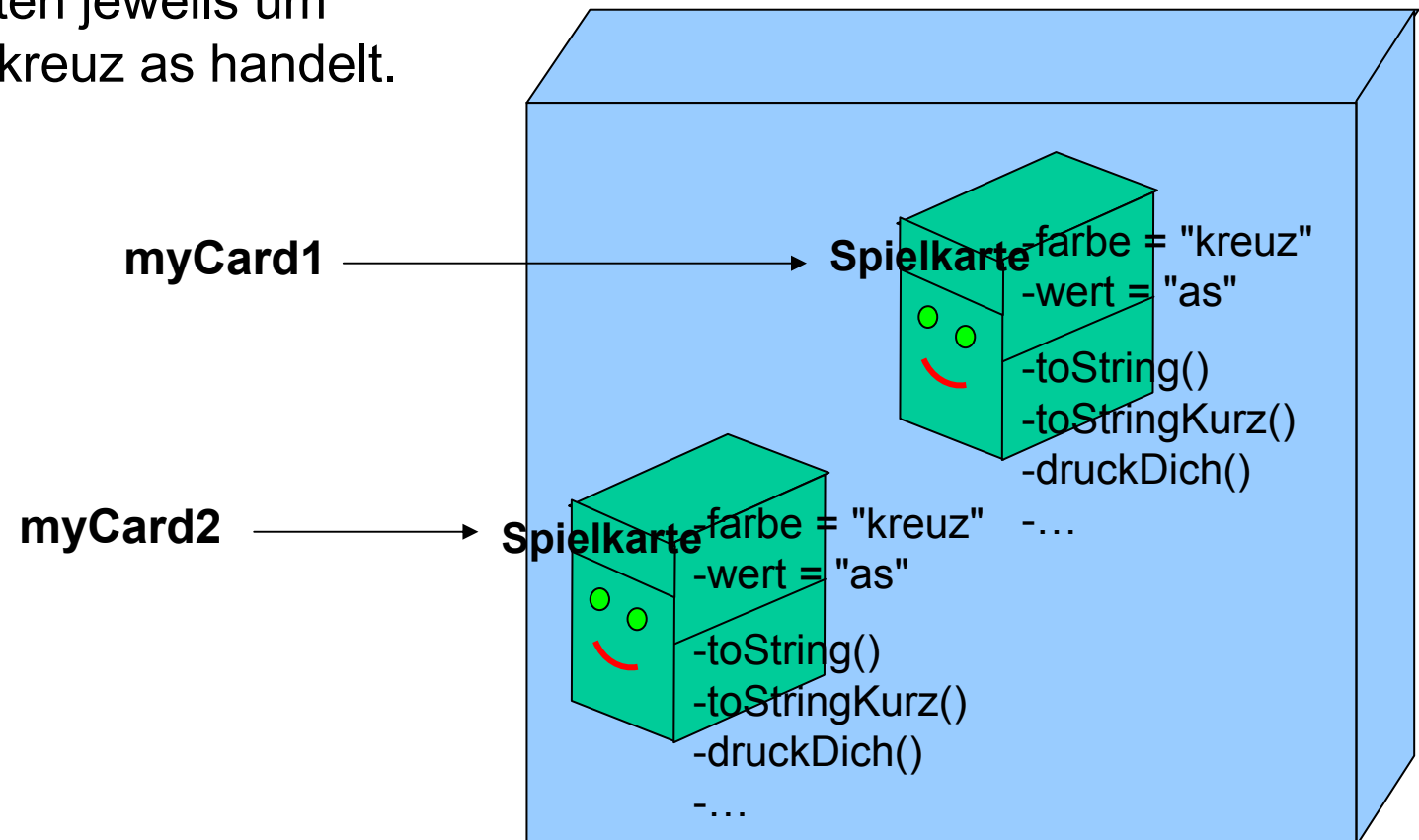
# Gleichheit

- Der Vergleichsoperator heißt "=="
  - und nicht etwa "=", das ist nämlich der Zuweisungsoperator
- Bei `if (zahl == 0) <tu was>;` muss deshalb "==" stehen.
- Wenn Objekte verglichen werden, bedeutet `obj1 == obj2`, dass es sich um das gleiche Objekt handelt, also um die gleiche Speicherstelle im Hauptspeiche.
- Beispiel: `(myCard1 == myCard2)` ist hier `true`.



# Gleichheit

- Bei den folgenden Spielkarten handelt es sich NICHT um dasselbe Objekt.
- `(myCard1 == myCard2)` ist `false`, obwohl es sich bei beiden Karten jeweils um ein kreuz as handelt.



# Gleichheit

- Dasselbe gilt bei Strings.
- Bei den folgenden Strings handelt es sich NICHT um dasselbe Objekt:
  - `String myString1 = "Hans";`  
`String myString2 = "Hans";`  
`System.out.println(myString1 == myString2)`
  - **müsste** `false` liefern (liefert aber `true`)
  - denn der Compiler in Eclipse hat da was "optimiert"...
- Aber
  - `String myString1 = "Hans";`  
`String myString2 = "Hansi";`  
`myString2 = myString2.substring(0,4);`  
`System.out.println(myString1 == myString2)`
  - **liefert tatsächlich** `false`.

# Objektvergleich

- Sollen Objekte verglichen werden, z.B. die Strings "Hans" und "Hans", dann brauchen wir eine Vergleichsmethode.

- Für Strings funktioniert

```
System.out.println(myString1.equals(myString2))
```

und sollte auch immer verwendet werden!

- Wie können wir Spielkarten vergleichen?

```
Spielkarte myCard1 = new Spielkarte("Kreuz", "As");
```

```
Spielkarte myCard2 = new Spielkarte("Kreuz", "As");
```

```
System.out.println(myCard1 == myCard2) liefert false
```

- Aufgabe: Entwickeln Sie in der Klasse Spielkarte
  - eine Methode `equals()`, die zwei Spielkarten vergleicht und
    - `true` zurückgibt, wenn die Spielkarten in Farbe und Wert übereinstimmen und
    - andernfalls `false` zurückgibt

# Objektvergleich

- Lösungsansatz:
  - Eingabeparameter muss eine Spielkarte sein,
  - Rückgabetyp ist boolean,
  - Sichtbarkeit ist öffentlich
  - Die Übereinstimmung in Farbe und Wert wird als Stringvergleich implementiert.
- Lösung:

```
public boolean equals(Spielkarte otherCard) {  
    if (!this.getFarbe().equals(otherCard.getFarbe())) {  
        return false;  
    } else if (!this.getWert().equals(otherCard.getWert())) {  
        return false;  
    } else return true;  
}
```

**oder (gleichwertig)**

```
public boolean equals(Spielkarte otherCard) {  
    return (this.getFarbe().equals(otherCard.getFarbe())  
        && this.getWert().equals(otherCard.getWert()));  
}
```

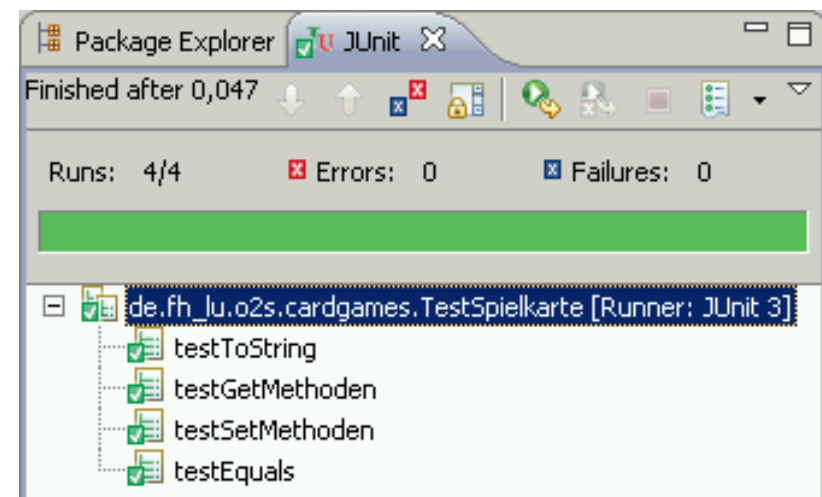
# Test

- Test mit JUnit: Neue Testmethode in unserer Testklasse

TestSpielkarte

```
public void testEquals(){  
    Spielkarte myCard1 = new Spielkarte("Kreuz", "As");  
    assertTrue(myCard.equals(myCard1));  
    assertFalse(myCard == myCard1);  
}
```

- Beachte: assertEquals() verwendet intern "==",
    - wir müssen deshalb mit assertTrue() arbeiten.
    - Gleichzeitig ermöglicht uns assertFalse() eine Gegenprobe.
  - Testausführung durch
    - Rechter Mausklick auf die Testklasse "TestSpielkarte"
- Run As → JUnit Test





# Klassendiagramm

- In einem Klassendiagramm der UML werden die Klassen etwa so dargestellt wie auf diesen Folien, vgl. Vorlesung „Modellierung“.
- Allerdings werden dabei nur die Klassen (Vorlagen / Schablonen) dargestellt und nicht die einzelnen Objekte.

| Spielkarte                                                                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| - private farbe<br>- private wert                                                                                                                              |
| - Spielkarte (String, String)<br>- toString()<br>- druckDich()<br>- <set...-Methoden><br>- <get...-Methoden><br>- init(String, String)<br>- equals(Spielkarte) |

# **Programmierung II**

## **Thema 3: Fallstudie Kartenspiel**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

# Fallstudie

- Wir erzeugen u.a. Klassen `Kartenspiel` und `Kartenstapel` und schauen uns ein fertiges BlackJack Spiel an.
- Ziele:
  - Wiederholung von Java-Programmierkonstrukten, speziell Arrays, Methoden, Konstruktoren
  - Üben der objektorientierten Denkweise, speziell der Kapselung von Daten und Fähigkeiten
  - Einführung des Datentyps `Stapel` als Klasse

# Ein Kartenspiel in Java







- In der klassischen Denkweise könnte man sagen:  
Ein Kartenspiel ist eine bestimmte Anzahl von Spielkarten, z.B. in einem Array
- In der objektorientierten Denkweise gilt:  
Alles ist ein Objekt, also ist auch ein Kartenspiel ein Objekt
- ➔ Objekt zu sein bedeutet (nach Vorlesung 1), es hat Daten (Attribute) und Fähigkeiten (Methoden)
  - Daten eines Kartenspiels sind z.B. die einzelnen Spielkarten
  - Was sind die Fähigkeiten eines Kartenspiels?

# Daten

- Daten eines Kartenspiel

- Die einzelnen Spielkarten. Wie werden diese gespeichert? Z.B. in einem Array.
- Der Übersicht halber sollen die Karten zweidimensional gespeichert werden.

- Wir benutzen deshalb ein
- Attribut `kartenAA` vom Typ `Spielkarte[][]` (zweidimensionales Array)

|       | 2                                                                                     | 3                                                                                     |     | As                                                                                    |
|-------|---------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|-----|---------------------------------------------------------------------------------------|
| Kreuz |  2 |  3 | ... |  A |
| Pik   |  2 |  3 | ... |  A |
| ...   | ...                                                                                   | ...                                                                                   | ... | ...                                                                                   |

- Zur Unterstützung:









- Attribute `farbeA` und `werteA` vom Typ `String[]`, in denen die einzelnen Farben bzw. Werte als Konstanten gespeichert sind:
  - `String[] farbenA = {"kreuz", "pik", "herz", "karo"};`
  - `String[] werteA = {"2", "3", ..., "as"};`

Darstellung in UML →



# Fähigkeiten

- Interessante Fähigkeiten
  - zurückgeben der Anzahl an Karten im Kartenspiel  
→ Methode `getKartenAnzahl()`
  - zurückgeben des Gesamtpunktwerts aller Karten  
→ Methode `getPunktwert()`
- Standardfähigkeiten
  - Konstruktor `Kartenspiel()`, der bei der Erzeugung eines Kartenspiels aufgerufen wird und das `kartenAA` füllt
  - get-/set-Methoden: Zunächst nur `getKartenAA()`
  - Ausgabe-Methode: `toString()`

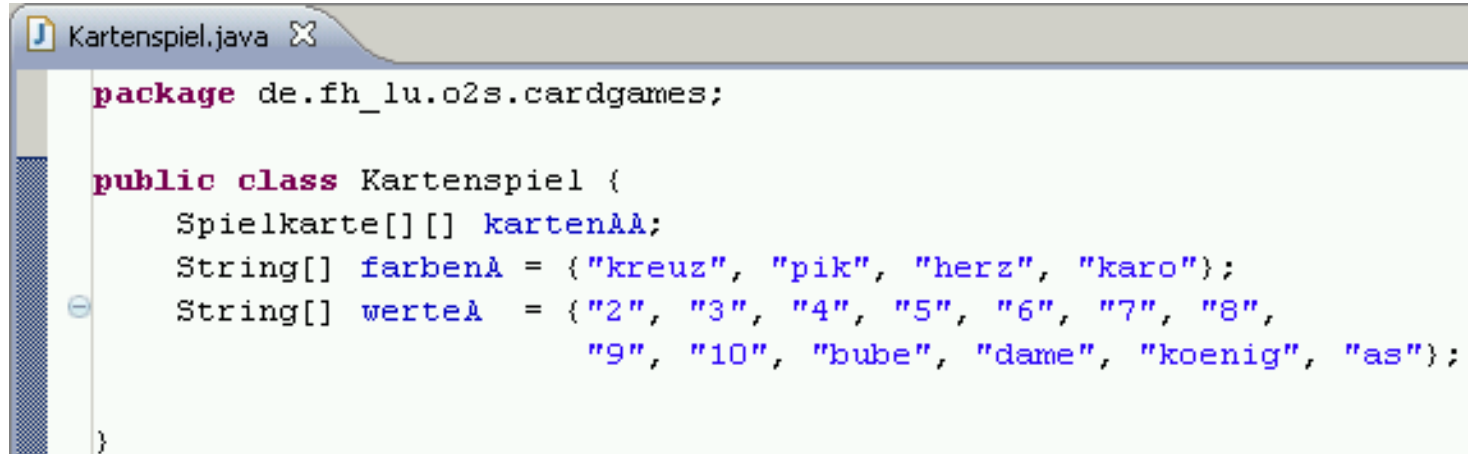
| Kartenspiel                                                                           |                                             |
|---------------------------------------------------------------------------------------|---------------------------------------------|
|  | <code>kartenAA : Spielkarte[][]</code>      |
|  | <code>farbenA : String[]</code>             |
|  | <code>werteA : String[]</code>              |
| <hr/>                                                                                 |                                             |
|  | <code>Kartenspiel()</code>                  |
|  | <code>getKartenAA() : Spielkarte[][]</code> |
|  | <code>toString() : String</code>            |
|  | <code>getKartenAnzahl() : Integer</code>    |
|  | <code>getPunktwert() : Integer</code>       |

# Kartenspiel

- Aufgabe: Implementieren Sie die Klasse Kartenspiel wie oben beschrieben
  - im Package `de.fh_lu.o2s.cardgames` Ihres Eclipse-Workspace
- Lösungsansatz: Vorgehen wie oben beschrieben
- Anmerkung:
  - Mit den Mitteln aus Programmierung 1 sollten Sie bereits in der Lage sein, dies selbst zu tun.

# Kartenspiel

- Lösung, Schritt 1:
  - Neue Klasse ohne main()-Methode
  - Definition der genannten Attribute wie oben

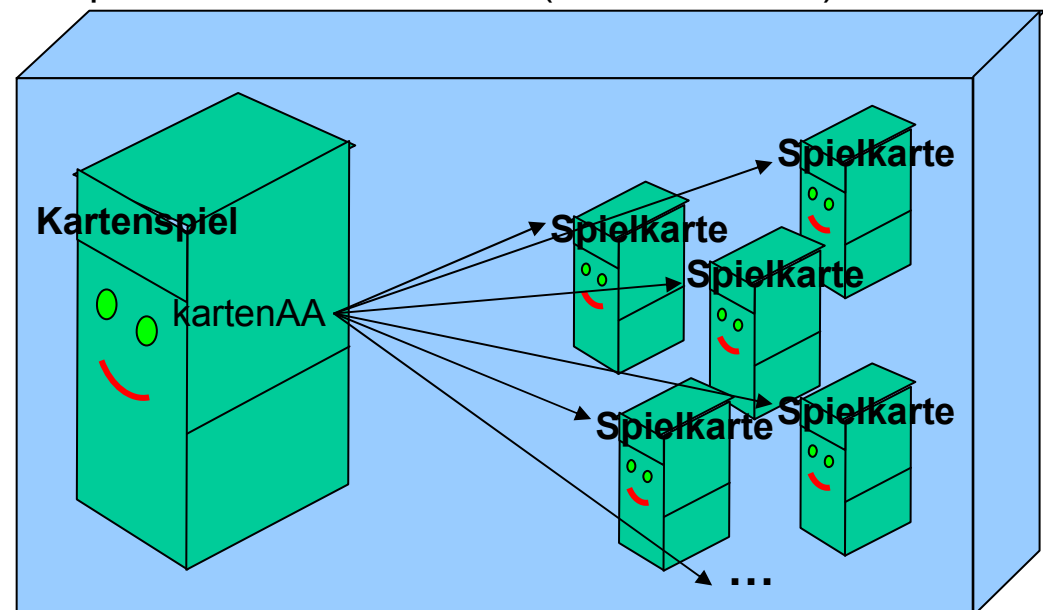


```
Kartenspiel.java ✕  
  
package de.fh_lu.o2s.cardgames;  
  
public class Kartenspiel {  
    Spielkarte[][] kartenAA;  
    String[] farbenA = {"kreuz", "pik", "herz", "karo"};  
    String[] werteA = {"2", "3", "4", "5", "6", "7", "8",  
                      "9", "10", "bube", "dame", "koenig", "as"};  
}
```



# Kartenspiel

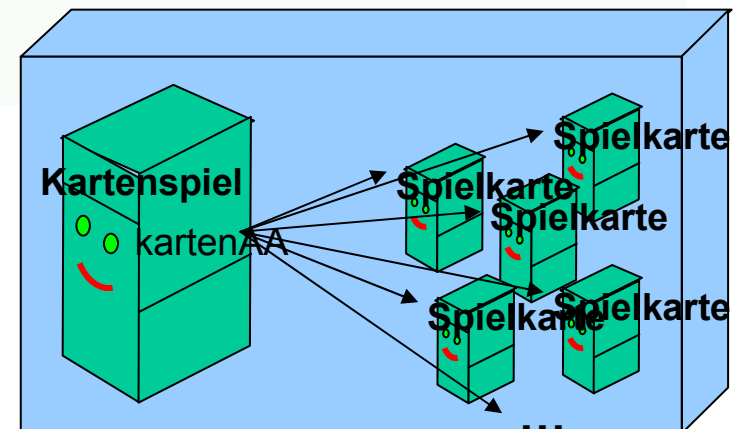
- Lösung, Schritt 2: Implementierung des Konstruktors:
    - Sichtbarkeit `public`, kein Rückgabewert, keine Parameter
    - Das Array `kartenAA` muss in der richtigen Größe erzeugt werden:
      - Zweidimensionales Array mit Platz für `farbenA.length * werteA.length` Spielkarten
- ```
public Kartenspiel(){  
    kartenAA = new Spielkarte[farbenA.length][werteA.length];
```
- Anschließend muss der Konstruktor
    - alle Spielkarten als Objekte im Hauptspeicher erzeugen und
    - in `kartenAA` für jede Spielkarte eine Referenz (einen Verweis) einfügen.



# Kartenspiel

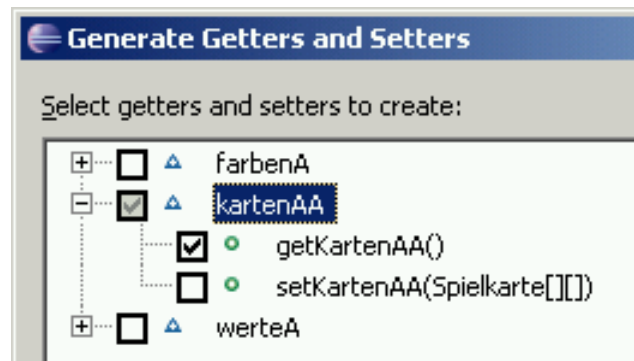
- Lösung, Schritt 2: Implementierung des Konstruktors:
  - Dies erfolgt mit einer Doppelschleife über Farben und Werte, in der
    - alle Spielkarten mit dem Konstruktor `Spielkarte(String, String)` erzeugt werden. Als Parameter werden Farbe und Wert aus der Schleife genutzt.
    - Die jeweilige Spielkarte wird an der richtigen Stelle in `kartenAA` eingetragen. Die Indizes dafür ergeben sich aus der Doppelschleife

```
public Kartenspiel(){  
    kartenAA = new Spielkarte[farbenA.length][werteA.length];  
    for(int f = 0; f < farbenA.length; f++){  
        for(int w = 0; w < werteA.length; w++){  
            kartenAA[f][w] = new Spielkarte(farbenA[f], werteA[w]);  
        }  
    }  
}
```



# Kartenspiel

- Lösung, Schritt 3: Standard-Methode `getKartenAA()`
  - Die Standard-Methode `getKartenAA()` können wir selbst bauen mit Rückgabedatentyp `Spielkarte[][]` und Sichtbarkeit `public`
  - oder uns von Eclipse erzeugen lassen:
    - Source → Generate Getters and Setters...
    - Dazu muss der Cursor innerhalb des Codes der Klasse stehen,
    - idealerweise klicken Sie zuerst auf das Attribut `kartenAA`.



→ OK

- Ergebnis (in beiden Fällen):

```
public Spielkarte[][] getKartenAA() {  
    return kartenAA;  
}
```

# Kartenspiel

- Lösung, Schritt 4: Weitere get-Methoden
  - `getKartenAnzahl()` und `getPunktWert()` haben beide die Sichtbarkeit `public` und den Rückgabedatentyp `int`.
  - Die Kartenanzahl kann nach der Größe des Spielkartenarrays berechnet werden: `farbenA.length * werteA.length`
  - Der PunktWert kann berechnet werden, indem die PunktWerte aller Spielkarten aufaddiert werden. Hierfür brauchen wir wieder eine Doppelschleife über `kartenAA`.

```
public int getKartenAnzahl(){
    return farbenA.length * werteA.length;
}
public int getPunktWert(){
    int myPunktWert = 0;
    for(int f = 0; f < farbenA.length; f++){
        for(int w = 0; w < werteA.length; w++){
            myPunktWert += kartenAA[f][w].getPunktWert();
        }
    }
    return myPunktWert;
}
```

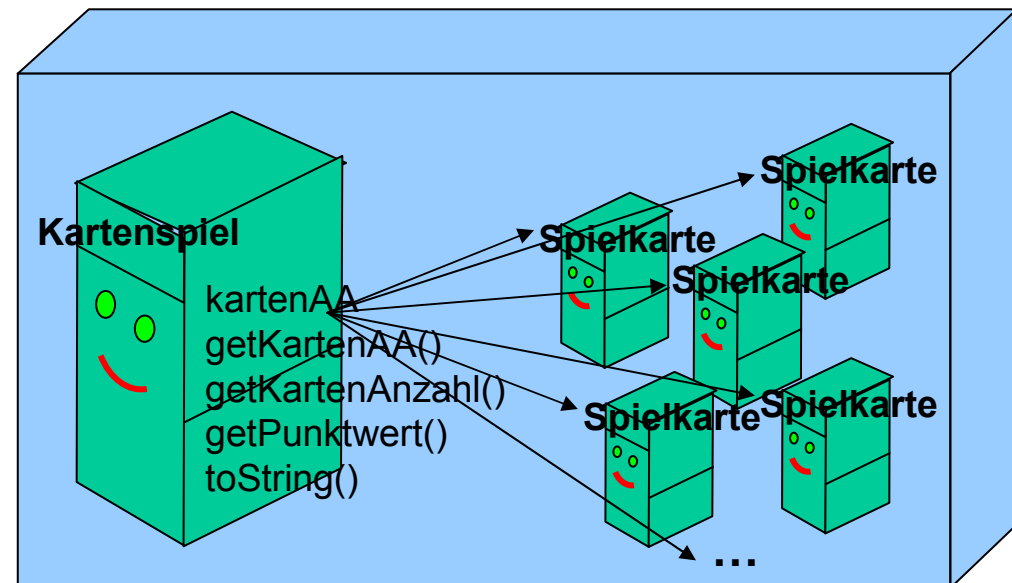
# Kartenspiel

- Lösung, Schritt 5: Ausgabemethode `toString()`
  - Grundidee:
    - Sichtbarkeit ist `public`, Rückgabedatentyp ist `String`
    - Zuerst kommt eine Zeile "dies ist ein Kartenspiel mit  $x$  Karten und einem Gesamtpunkt看t von  $y$ "
    - Danach werden alle Spielkarten untereinander angezeigt.
  - Um alle Spielkarten anzuzeigen, brauchen wir wieder eine Doppelschleife.

```
public String toString(){
    String myString = "Dies ist ein Kartenspiel mit " +
                      this.getKartenAnzahl() +
                      " Karten und einem Gesamtpunkt看t von " +
                      this.getPunkt看t() +
                      " Punkten\n";
    for(int f = 0; f < farbenA.length; f++){
        for(int w = 0; w < werteA.length; w++){
            myString += kartenAA[f][w].toString() + "\n";
        }
    }
    return myString;
}
```

# Anmerkung

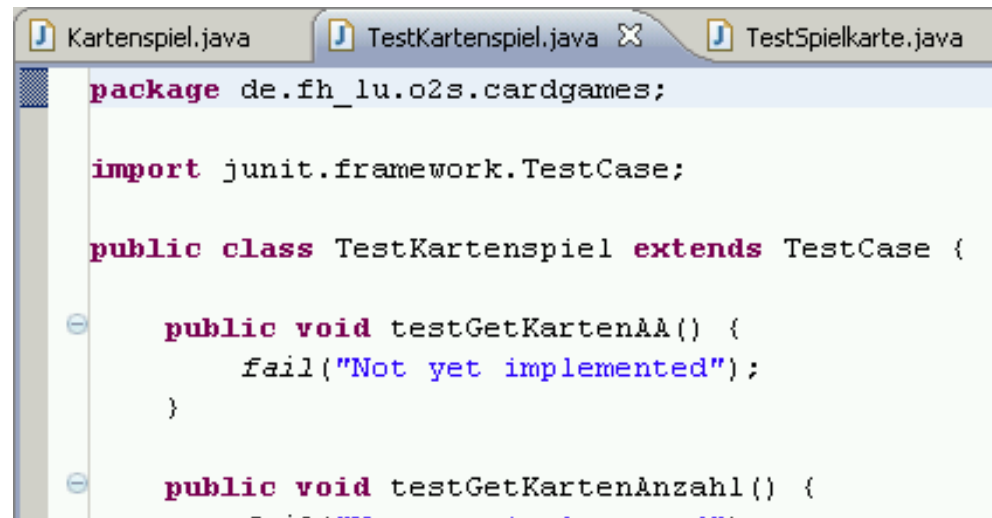
- Sowohl bei `getPunktwert()` als auch bei `toString()`
  - haben wir auf die entsprechende Funktionalität in `Spielkarte` zurückgegriffen.
  - Damit brauchten wir dies nicht noch einmal programmieren.
  - und eine `Spielkarte` kennt ihren Punktwert viel besser als ein `Kartenspiel` diesen kennt.



- Bei der Implementierung
  - mussten wir die Klasse Kartenspiel von innen betrachten, z.B. die Datenspeicherung im Array
- Zum Test
  - betrachten wir die Klasse Kartenspiel nur noch von außen, d.h. wir nutzen nur die public-Methoden
- Aufgabe:
  - Implementieren Sie jetzt einen JUnit-Test, der die Klasse `Kartenspiel` testet.
- Lösungsansatz:
  - Wir testen die Methoden `getKartenAA()`, `getKartenAnzahl()`, `getPunktwert()`, `toString()`, und verwenden eine `setUp()`-Methode, in der der Konstruktor `Kartenspiel()` benutzt wird.

# Test

- Lösung, Schritt 1:
  - Rechtsklick auf das Package `de.fh_lu.o2s.cardgames`
  - New → JUnit Test Case
  - Name: *TestKartenspiel*, Class under Test: *Kartenspiel*, Next >
  - Auswahl der Methoden `getKartenAA()`, `getKartenAnzahl()`, `getPunktwert()`, `toString()` → Finish.
- Ergebnis ist eine Klasse `TestKartenspiel` mit Test-Methoden für alle der genannten Methoden von `Kartenspiel`



```
package de.fh_lu.o2s.cardgames;

import junit.framework.TestCase;

public class TestKartenspiel extends TestCase {

    public void testGetKartenAA() {
        fail("Not yet implemented");
    }

    public void testGetKartenAnzahl() {
        fail("Not yet implemented");
    }
}
```



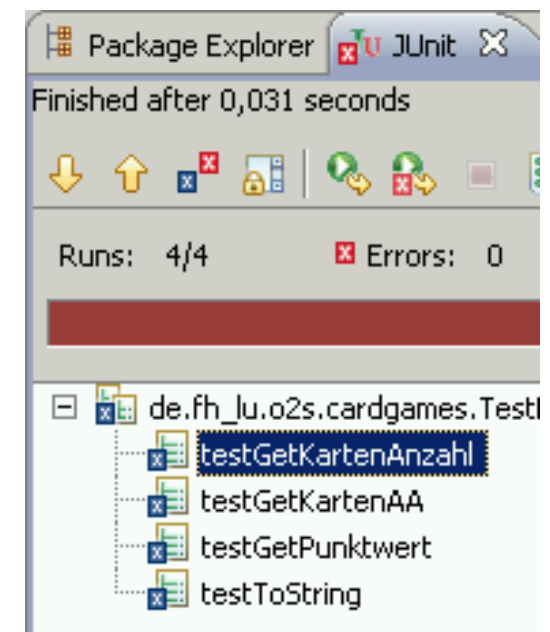
# Test

- Lösung, Schritt 2:

- Attribut `myKartenspiel` und zusätzliche Methode `setUp()`, in der ein Kartenspiel erzeugt und in diesem Attribut gespeichert wird.

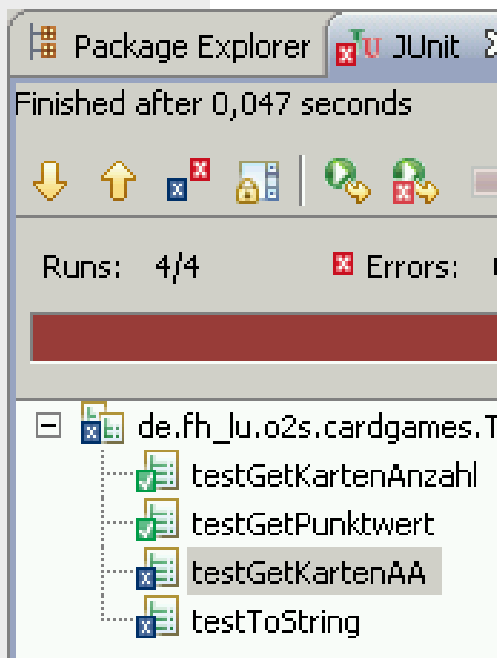
```
public class TestKartenspiel extends TestCase {  
    Kartenspiel myKartenspiel;  
    public void setUp(){  
        myKartenspiel = new Kartenspiel();  
    }  
}
```

- Ausführung des Tests durch Rechtsklick auf die Klasse  
TestKartenspiel → Run As → JUnit Test
- Ergebnis: Alle vorhandenen Test-Methoden schlagen fehl, weil die Inhalte noch nicht implementiert sind.



# Test

- Lösung, Schritte 3 und 4:
  - Implementierung der Test-Methode `testGetKartenAnzahl()`, in der wir sicherstellen, dass als Ergebnis immer 52 herauskommt, denn wir haben konstant 4 Farben und 13 Werte.
  - Implementierung der Test-Methode `testGetPunktwert()`, in der wir sicherstellen, dass als Ergebnis immer 216 herauskommt.
- Testausführung: Beide Methoden funktionieren



```
public class TestKartenspiel extends TestCase {
    Kartenspiel myKartenspiel;
    public void setUp() {
        myKartenspiel = new Kartenspiel();
    }
    public void testGetKartenAnzahl() {
        assertEquals(myKartenspiel.getKartenAnzahl(), 52);
    }
    public void testGetPunktwert() {
        assertEquals(myKartenspiel.getPunktwert(), 216);
    }
}
```

- Lösung, Schritt 5:
  - Zum Test von `getKartenAA()` testen wir nur, dass der Rückgabewert tatsächlich ein Spielkarten-Doppelarray ist

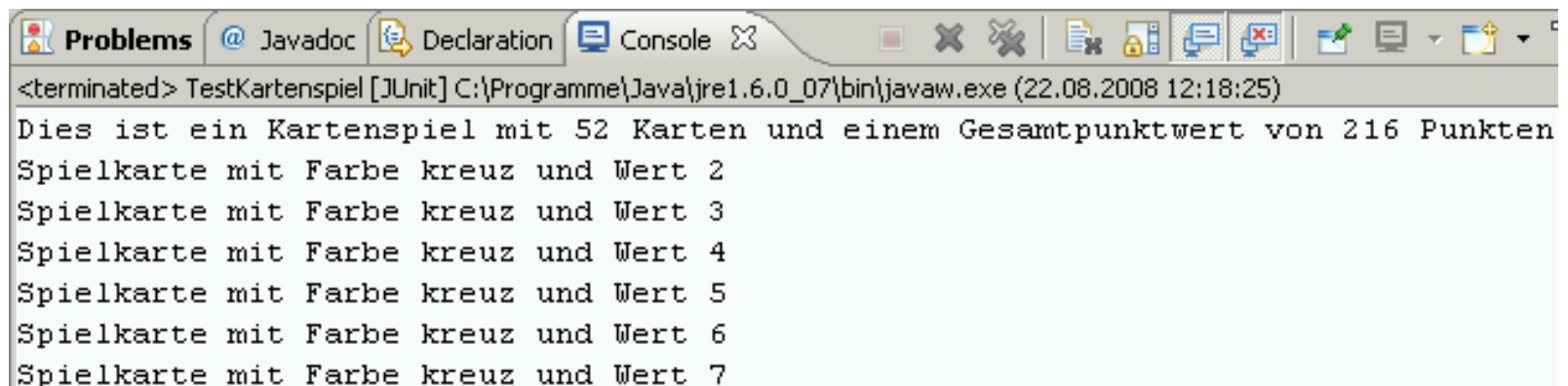
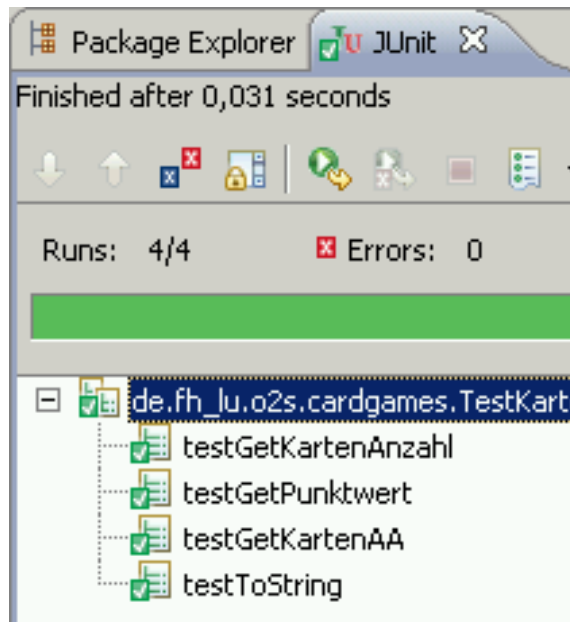
```
public void testGetKartenAA() {  
    assertTrue(myKartenspiel.getKartenAA() instanceof Spielkarte[][]);  
}
```

- Schritt 6:
  - Zum Test von `toString()` lassen wir das Ergebnis der Methode in der Konsole ausgeben, prüfen die Ausgabe manuell und verzichten auf eine Assertion.

```
public void testToString() {  
    System.out.println(myKartenspiel.toString());  
}
```

# Ergebnis

- Testausführung und Konsolenausgabe



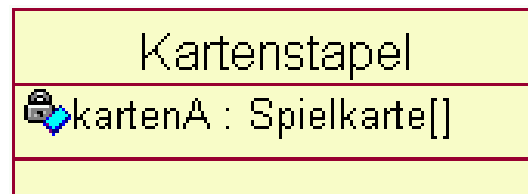
# Beobachtung

- Das `Kartenspiel`, das wir eben gebaut haben ist immer vollständig und immer gleich sortiert.
  - Wenn wir Karten verteilen
    - bleibt ein unvollständiges Spiel übrig: Ein Karten***stapel***.
    - Dieser Kartenstapel sollte nicht sortiert, sondern zufällig gemischt sein.
  - Außerdem
    - hat jeder Spieler einige Karten auf der Hand  
→ Ebenfalls ein Kartenstapel.
- Wir brauchen Objekte vom Typ `Kartenstapel`

Karo König
...
Pik 4
Kreuz 10
Karo 6
Herz Bube
Herz 2
Kreuz As

# Kartenstapel als Objekt

- Ziel: Wir implementieren eine Klasse `Kartenstapel`
- Aufgabe: Beantworten Sie in Form eines Klassendiagramms
  - die Daten eines Kartenstapels und
  - die Fähigkeiten eines Kartenstapels.
- Lösungsansatz (zunächst die Daten):
  - Die Karten eines Kartenstapels liegen in einer bestimmten Reihenfolge vor,
    - ➔ Zur Realisierung reicht deshalb ein einfaches Array von Spielkarten
- Lösung, Schritt 1: Klassendiagramm:



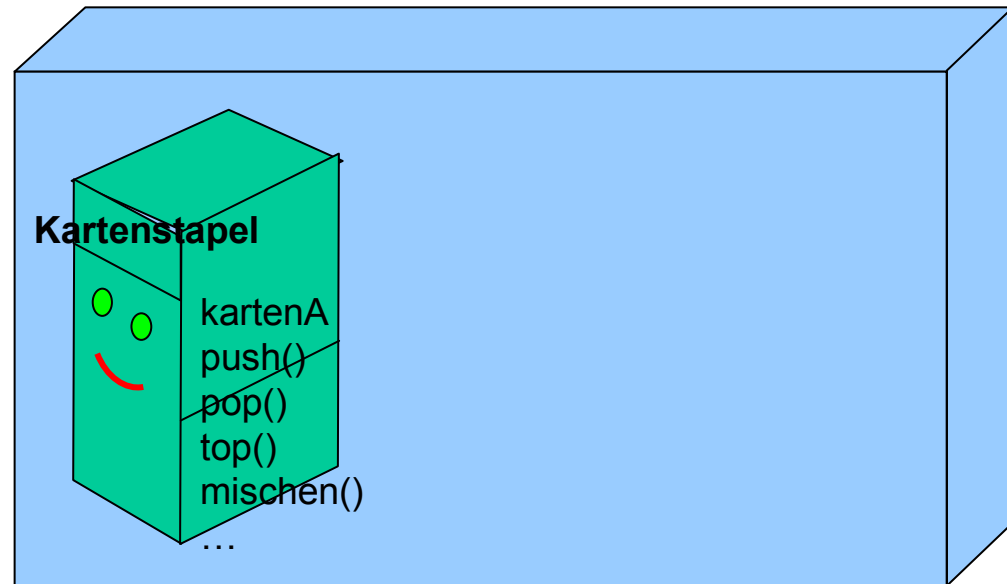
# Kartenstapel als Objekt

- Lösung, Schritt 2: Erzeugung eines Kartenstapels
  - Es sollte möglich sein, aus einem `Kartenspiel` einen `Kartenstapel` zu erzeugen
  - Konstruktor mit einem `Kartenspiel` als Parameter
  - Weitere Möglichkeiten, die evtl. sinnvoll sein könnten:
    - Erzeugung eines leeren Kartenstapels (0 Karten),
    - Erzeugung eines Kartenstapels mit genau einer Spielkarte,
    - Erzeugung eines Kartenstapels, in dem eine bestimmte Anzahl (int) von Spielen zusammengestapelt wird.
  - drei weitere Konstruktoren mit entsprechenden Parametern (überladen)



# Anmerkungen

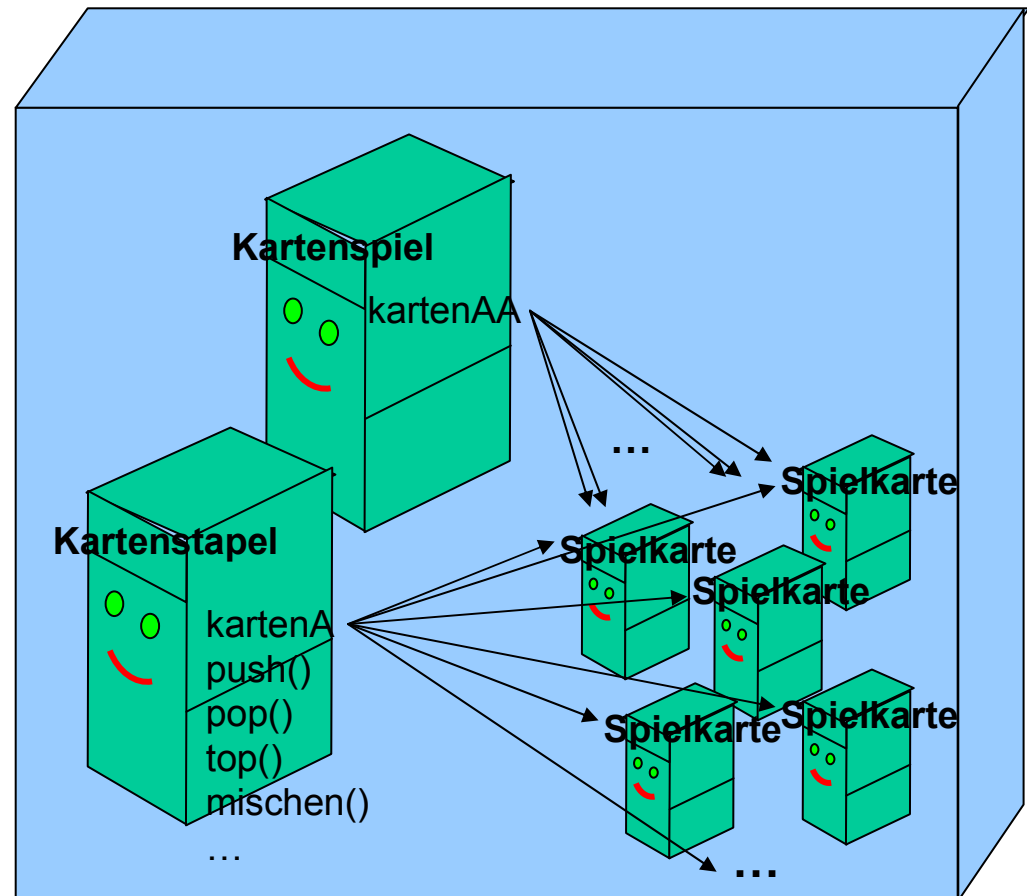
- Aus der dargestellten Klasse Kartenstapel können wir dann
  - mit `Kartenstapel myStapel1 = new Kartenstapel();` einen leeren Kartenstapel im Hauptspeicher erzeugen





# Anmerkungen

- um ein Kartenspiel zu stapeln,
  - brauchen wir zuerst ein Kartenspiel:  
`Kartenspiel mySpiel1 = new Kartenspiel();`
  - um dieses dann aufzustapeln:  
`Kartenstapel myStapel2 = new Kartenstapel(mySpiel1);`
- Die gestapelten Objekte vom Typ `Spielkarte` werden dabei
  - von dem Konstruktor der Klasse `Kartenstapel` erzeugt und anschließend auf dem Stapel abgelegt, indem jeweils eine Referenz in `kartenA` eingefügt wird.




# Kartenstapel als Objekt

- Lösung, Schritt 3: Zugriff auf den Kartenstapel mit den typischen Stapelmethoden
  - `push(Spielkarte)` – Legt eine Karte oben auf dem Stapel ab
  - `pop()` – Hebt eine Karte oben vom Stapel ab
  - `top()` – Liefert die oberste Karte des Stapels, ohne sie abzuheben.
  - `empty()` – Liefert `true`, wenn der Stapel keine Karten enthält, andernfalls `false`.



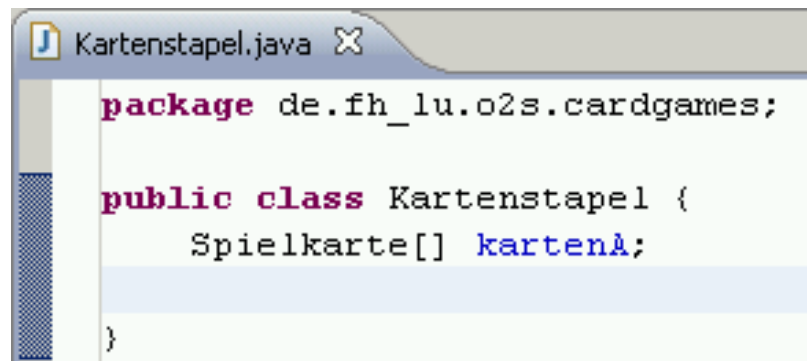
# Kartenstapel als Objekt

- Lösung, Schritt 4: Weitere interessante Methoden
  - `mischen()` – Mischt den Kartenstapel
  - `addKartenspiel(Kartenspiel)` – Stapelt ein weiteres Kartenspiel dazu
  - `addKartenstapel(Kartenstapel)` – Stapelt einen weiteren Kartenstapel auf den aktuellen Kartenstapel oben drauf
  - `getKartenAnzahl()` und `getPunktwert()` – Wie beim Kartenspiel
  - `toString()` – Anzeigemethode wie üblich

Kartenstapel	
	<code>kartenA : Spielkarte[]</code>
	<code>Kartenstapel(Kartenspiel)</code>
	<code>Kartenstapel()</code>
	<code>Kartenstapel(Spielkarte)</code>
	<code>Kartenstapel(int)</code>
	<code>push(Spielkarte) : void</code>
	<code>pop() : Spielkarte</code>
	<code>top() : Spielkarte</code>
	<code>empty() : boolean</code>
	<code>mischen() : void</code>
	<code>addKartenspiel(Kartenspiel) : void</code>
	<code>addKartenstapel(Kartenstapel) : void</code>
	<code>getKartenAnzahl() : int</code>
	<code>getPunktwert() : int</code>
	<code>toString() : String</code>

# Implementierung Kartenstapel

- Aufgabe: Implementieren Sie die beschriebene Klasse Kartenstapel
- Anmerkung:
  - Wir implementieren hier nur einen Teil der genannten Methoden, der Rest dient als Übungsaufgabe.
- Lösung, Schritt 1:
  - Neue Java-Klasse Kartenstapel ohne main()-Methode
  - kartenA als Attribut vom Typ Spielkarten-Array



```
Kartenstapel.java X
package de.fh_lu.o2s.cardgames;

public class Kartenstapel {
    Spielkarte[] kartenA;
}
```

# Implementierung Kartenstapel

- Lösung, Schritt 2: Konstruktor mit Parameter vom Typ `Kartenspiel`
  - Sichtbarkeit `public`, kein Rückgabedatentyp, ein Parameter vom Typ `Kartenspiel`
  - Wir müssen sämtliche Karten des Kartenspiels in unser Array `kartenA` übernehmen.
  - Dafür müssen wir `kartenA` zunächst in der richtigen Größe erzeugen. Diese ermitteln wir, indem wir bei dem Kartenspiel-Objekt die Methode `getKartenAnzahl()` aufrufen.

```
public Kartenstapel(Kartenspiel spiel){  
    kartenA = new Spielkarte[spiel.getKartenAnzahl()];  
  
}
```

# Implementierung Kartenstapel

- Lösung, Schritt 2, Fortsetzung:
  - Wir schicken dem Kartenspiel die Methode `getKartenAA()` und erhalten dafür alle Spielkarten des Kartenspiels in Form eines Doppelarrays.
  - Wir durchlaufen das Doppelarray und fügen alle Karten in das Array `kartenA` unseres Kartenstapels ein. Den hierfür nötigen Index pflegen wir von Hand.

```
public Kartenstapel(Kartenspiel spiel){
    kartenA = new Spielkarte[spiel.getKartenAnzahl()];
    int index = 0;
    Spielkarte[][] kartenAA = spiel.getKartenAA();
    for (int i = 0; i < kartenAA.length; i++){
        for (int j = 0; j < kartenAA[0].length; j++){
            kartenA[index] = kartenAA[i][j];
            index++;
        }
    }
}
```

# Implementierung Kartenstapel

- Lösung, Schritt 3: push(Spielkarte)
  - Sichtbarkeit `public`, Rückgabedatentyp `void`.
  - Um eine weitere Spielkarte auf unseren Stapel zu legen, müssen wir unser Array `kartenA` um einen Platz vergrößern.
  - Das geht aber nicht, wir müssen also
    - ein neues, größeres Array erzeugen,
    - die bereits vorhandenen Karten in das neue Array kopieren,
    - die zusätzliche Karte in das neue Array einfügen,
    - das neue Array im Attribut `kartenA` speichern.

```
public void push(Spielkarte newCard){  
    Spielkarte[] kartenNeu = new Spielkarte[kartenA.length + 1];  
    System.arraycopy(kartenA, 0, kartenNeu, 0, kartenA.length);  
    kartenNeu[kartenA.length] = newCard;  
    kartenA = kartenNeu;  
}
```

# Anmerkungen

```
public void push(Spielkarte newCard){  
    Spielkarte[] kartenNeu = new Spielkarte[kartenA.length + 1];  
    System.arraycopy(kartenA, 0, kartenNeu, 0, kartenA.length);  
    kartenNeu[kartenA.length] = newCard;  
    kartenA = kartenNeu;  
}
```

- Die Parameter des Befehls `System.arraycopy(...)` sind
  - Quell-Array,
  - Position, ab der aus dem Quell-Array kopiert werden soll,
  - Ziel-Array,
  - Position, ab der in das Ziel-Array kopiert werden soll,
  - Anzahl von Objekten, die kopiert werden sollen.



# Anmerkungen

```
public void push(Spielkarte newCard){  
    Spielkarte[] kartenNeu = new Spielkarte[kartenA.length + 1];  
    System.arraycopy(kartenA, 0, kartenNeu, 0, kartenA.length);  
    kartenNeu[kartenA.length] = newCard;  
    kartenA = kartenNeu;  
}
```

- Um die Indizes zu prüfen, macht man am besten ein Beispiel:
  - Wenn der Stapel seither 5 Spielkarten hat, dann gilt **kartenA.length = 5** und die Indizes dafür sind 0,1,2,3,4
  - Der neue Stapel muss  $5 + 1 = 6$  Karten aufnehmen, also `kartenNeu = new Spielkarte[6]`
  - Mit `System.arraycopy()` wird ab Index 0 kopiert (Quelle und Ziel) und es werden alle 5 vorhandenen Spielkarten kopiert. Diese befinden sich dann in `kartenNeu` auf den Indizes 0,1,2,3,4
  - `kartenNeu` hat 6 Plätze. Der oberste Index, wo die neue Karte hinkopiert werden muss, ist deshalb 5
- Ergebnis: Alles stimmt!

# JUnit Test

- Wir können nun einen Test für den Kartenstapel implementieren,
  - obwohl der Kartenstapel noch nicht ganz fertig ist.
  - Der Test wird also zunächst fehlschlagen und erst im Laufe der Entwicklung von Kartenstapel erfolgreich werden.
  - Das heißt dann **test-driven development**
- Dieser Test
  - wird Ihnen als Klasse `TestKartenstapel` zur Verfügung gestellt.
  - Kopieren Sie diesen bitte in Ihr Package  
`de.fh_lu.o2s.cardgames`
- Übungsaufgabe:
  - Beenden Sie die Implementierung der Klasse Kartenstapel und
  - verifizieren Sie Ihre Lösung anhand des vorgegebenen Tests  
`TestKartenstapel`

# Black Jack – Spielregeln I

- entnommen aus **Ratz, Scheffler, Seese: Grundkurs Programmieren in Java**, Band 1:
- Von Las Vegas über Monte Carlo bis zum Casino in Baden-Baden gehört das Kartenspiel Black Jack zum Standardprogramm. Auch wenn sich die Regeln im Detail von Haus zu Haus unterscheiden, folgen sie doch alle dem folgenden Grundprinzip:
- Ein oder mehrere Spieler spielen gegen die Bank (den Croupier) und versuchen, eine höhere Punktzahl zu erhalten als das Haus. Zu Anfang erhalten alle Spieler und der Croupier eine offen liegende Karte. Danach erhalten alle Spieler eine zweite offene, der Croupier eine verdeckt liegende Karte. Man versucht, die ideale Punktzahl von 21 Punkten zu erreichen. Hat man mit seinem Blatt diese überboten, so hat man verloren. Asse zählen 11 Punkte (es gibt auch Spielregeln, in denen das As nur einen Punkt zählt), sonstige Bilder 10 Punkte. Die anderen Karten zählen ihren aufgedruckten Wert.
- Der Spieler bzw. die Spielerin kann vom Croupier weitere Karten fordern („bleiben“) oder sich mit seinem Blatt zufrieden geben („danke“). Er sollte versuchen, so nahe wie möglich an die 21 Punkte heranzukommen, darf die Grenze aber nicht überschreiten.
- Sind alle Spieler fertig, kann auch der Croupier Karten nehmen. Er muss so lange Karten nehmen, wie er höchstens 16 Punkte hat. Hat er mehr als 16 Punkte, darf er keine weiteren Karten nehmen.
- Hat ein Spieler oder eine Spielerin bereits mit den ersten beiden Karten 21 Punkte erreicht, bezeichnet man dies als „Black Jack“. In diesem Fall darf der Croupier keine weiteren Karten nehmen; er hat also auch nur zwei Karten auf der Hand.
- Der Spieler bzw. die Spielerin gewinnt, wenn er bzw. sie nicht über 21 liegt und mehr Punkte als der Croupier hat. Haben Spieler und Croupier die gleiche Anzahl von Punkten, handelt es sich um ein Unentschieden („Egalité“).
- Wir wollen auf dem Computer nun ein solches Black-Jack-Spiel für einen Spieler und den Croupier realisieren.

# Black Jack Methoden

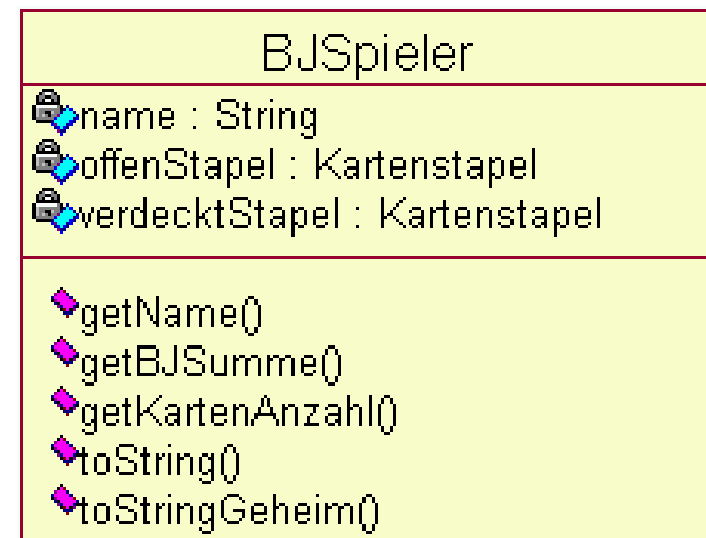
- Da beim Black Jack andere Punktwerte gelten, brauchen wir zusätzlich zu `getPunktWert()`
  - eine Methode `getBJWert()` in der Klasse `Spielkarte` und
  - eine Methode `getBJSumme()` in der Klasse `Kartenstapel`

```
public int getBJWert() {  
    int myVal = 0;  
    String myWert = this.getWert();  
    if (myWert.equals("bube")  
        || myWert.equals("dame")  
        || myWert.equals("koenig")) myVal = 10;  
    else if (myWert.equals("as")) myVal = 11;  
    else myVal = this.getPunktWert();  
    return myVal;  
}
```

```
public int getBJSumme() {  
    int pw = 0;  
    for (Spielkarte card : kartenA) {  
        pw += card.getBJWert();  
    }  
    return pw;  
}
```

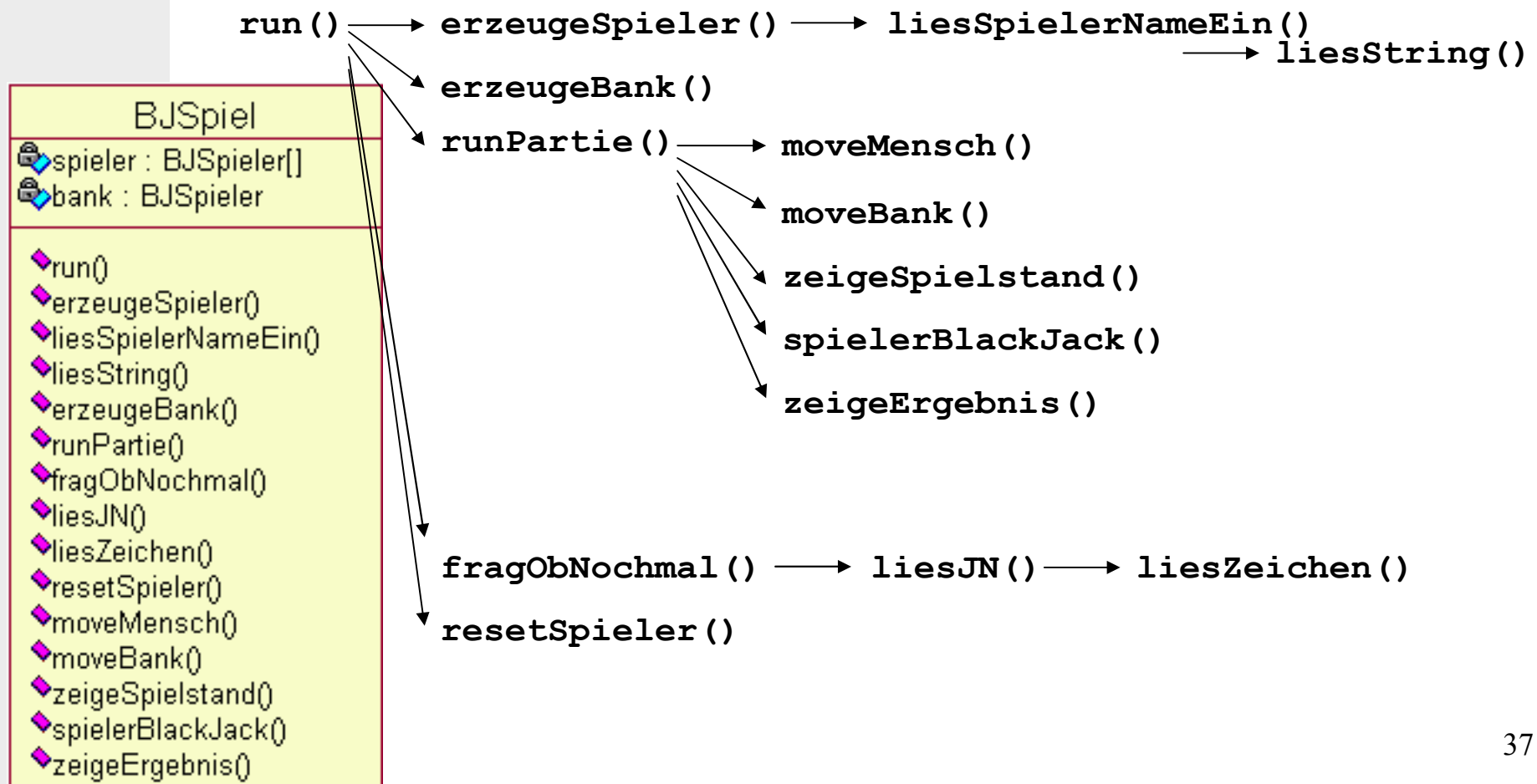
# Black Jack Spieler

- Ein Black Jack Spieler hat Daten
  - seinen Namen
  - einen Stapel offen liegender Karten
  - einen weiteren Stapel von Karten verdeckt auf der Hand
- und Fähigkeiten: Er kann
  - seinen Namen sagen,
  - seine Black Jack Summe sagen,
  - sagen wieviele Karten er hat,
  - sich selbst vollständig als String darstellen
  - sich selbst soweit als String darstellen, wie es auch Gegenspieler sehen dürfen



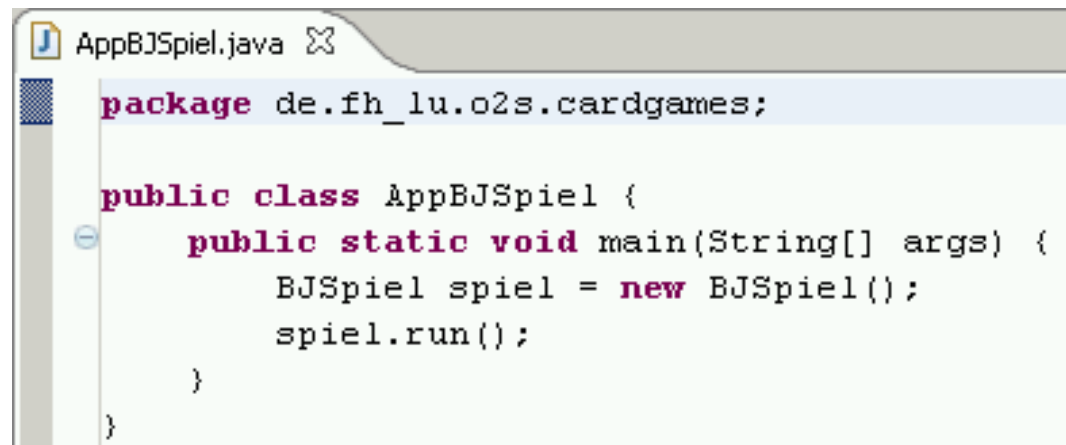
# Black Jack Spiel

- Die Funktionalität des Spiels wird von einer run()-Methode gesteuert:



# Black Jack Spiel

- BJSpiel
  - hat aber keine `main()`-Methode.
  - Stattdessen wird das Spiel über eine Applikation `AppBJSpiel` gestartet



```
AppBJSpiel.java ✕  
  
package de.fh_lu.o2s.cardgames;  
  
public class AppBJSpiel {  
    public static void main(String[] args) {  
        BJSpiel spiel = new BJSpiel();  
        spiel.run();  
    }  
}
```

- Genauer betrachten und ausführen / spielen:
  - Das Spiel wird Ihnen im Quelltext zur Verfügung gestellt.

# **Programmierung II**

## **Thema 4: Vererbung, Teil 1**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

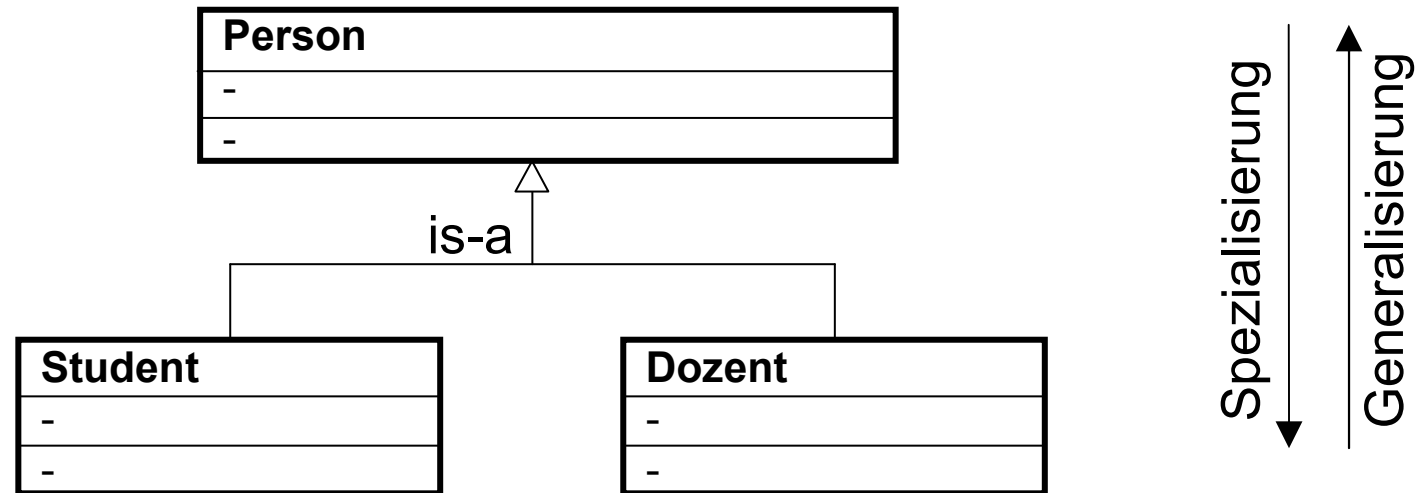


# Vererbung

- Mit Vererbung soll
  - der Zusammenhang zwischen Objekttypen, der in der richtigen Welt gilt, auch im Programmcode erhalten bleiben,
  - Codeverdopplung vermieden werden,
  - die Wiederverwendung von Programmteilen und Komponenten unterstützt werden.
- Wichtige Begriffe, die in diesem Kapitel vorgestellt werden, sind
  - Generalisierung und Spezialisierung,
  - Polymorphie und dynamisches Binden,
  - Casting.

# Beispiel

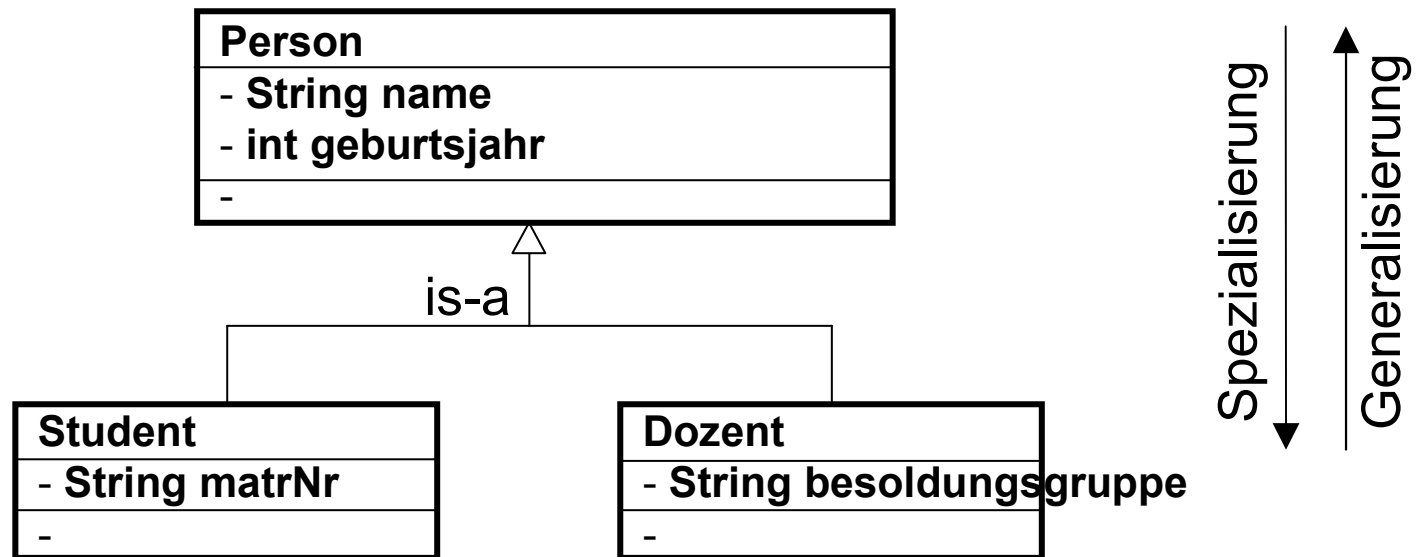
- Hochschulangehörige:



- Generalisierung:
  - Ein Student "is-a" Person
  - Ein Dozent "is-a" Person
- Spezialisierung:
  - Eine Person kann entweder Person, Student oder Dozent sein.
- In Java:
  - Person wird "Oberklasse" von Student und Dozent.
  - Student und Dozent werden "Unterklassen" von Person

# Beispiel

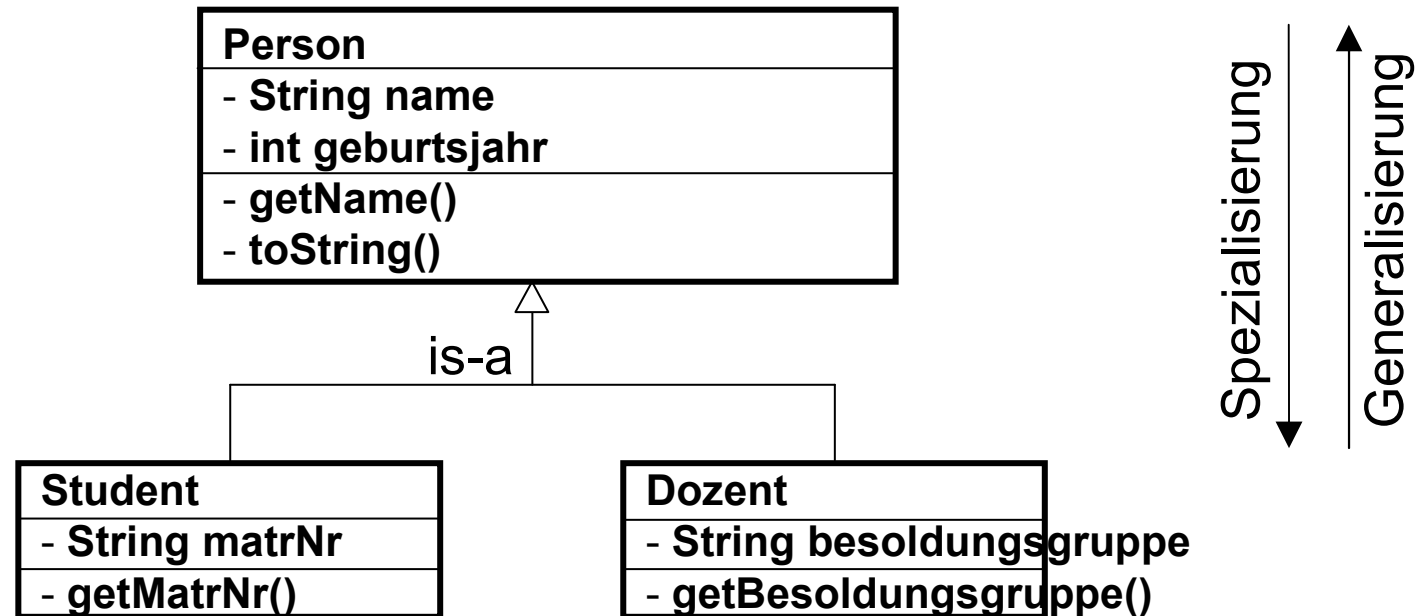
- Eigenschaften (Daten):



- Jede Eigenschaft der Oberklasse gilt – ohne dass man sie hinschreiben muss – automatisch auch für die Unterklasse
  - Person: name, geburtsjahr
  - Student: name, geburtsjahr, matrNr
  - Dozent: name geburtsjahr, besoldungsgruppe

# Beispiel

- Fähigkeiten (Methoden):



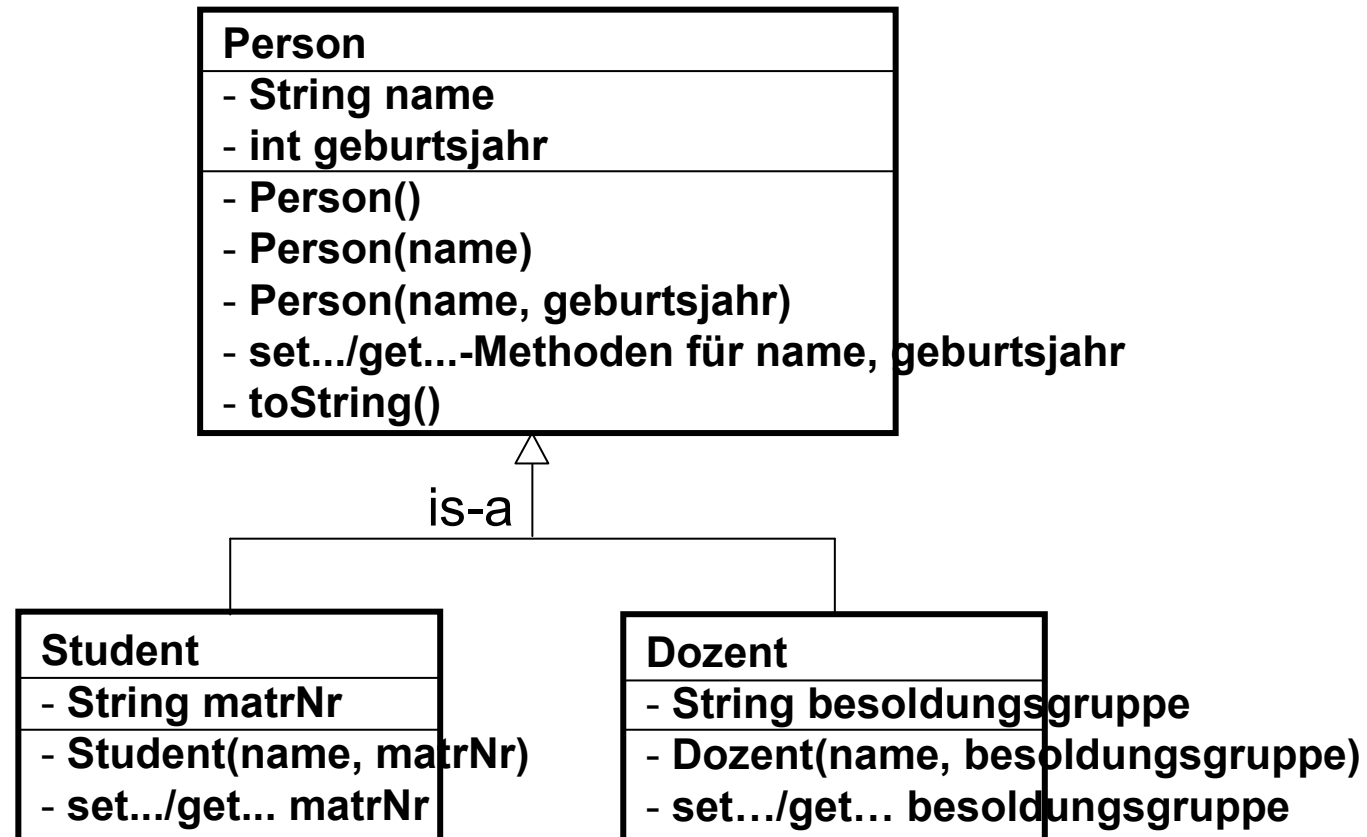
- Jede Fähigkeit der Oberklasse gilt automatisch auch für die Unterklasse.

# Beispiel

- Prinzip:
  - Eine Person hat die Eigenschaft `name`,
  - ein Student ist auch ("is-a") eine Person,
  - also muss ein Student auch die Eigenschaft `name` haben.
- bzw.
  - Eine Person kann `getName()` beantworten.
  - Ein Student ist auch ("is-a") eine Person.
  - Also kann ein Student auch `getName()` beantworten.
- Unterklassen können zusätzliche Fähigkeiten und Eigenschaften haben.

# Beispiel

- Vorläufiges Modell:

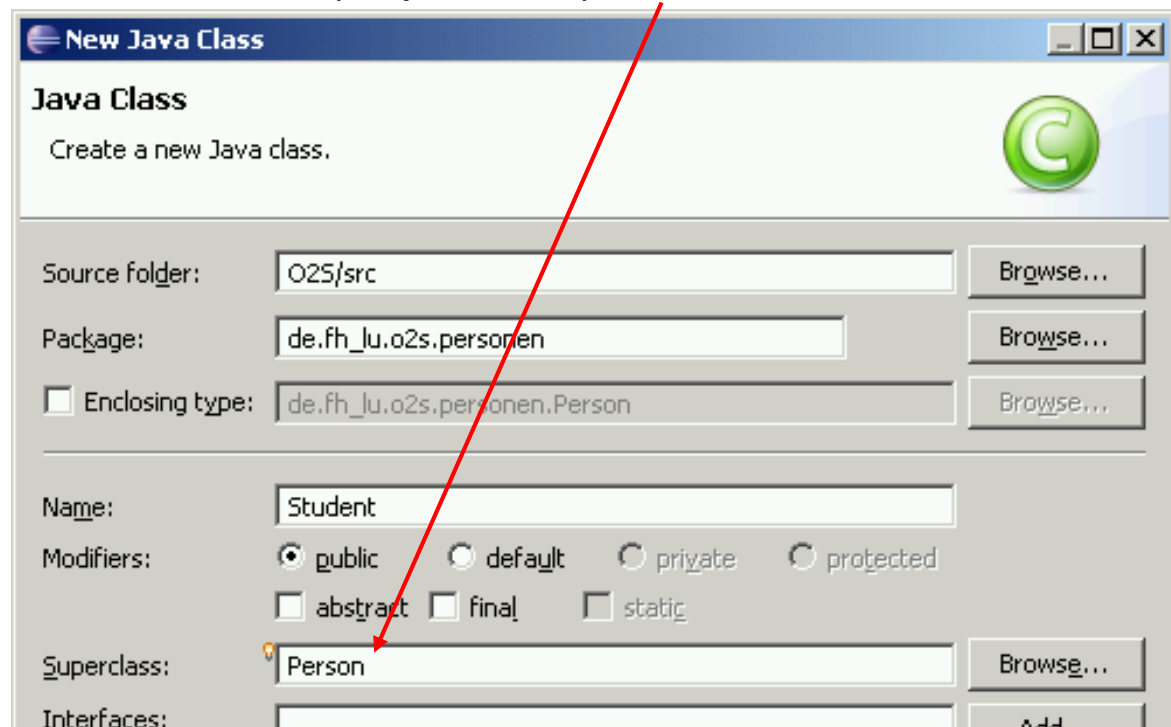


# Beispiel

- Anmerkung:
  - Zur Verwaltung von Mitgliedern einer Hochschule können später evtl. weitere Personenarten hinzugefügt werden (z.B. Alumni, Hochschulrat, Verwaltung).
  - In den Klassen Person, Student, Dozent sollen ausdrücklich weibliche Hochschulangehörige mit eingeschlossen sein.
- Aufgabe 1:
  - Implementieren Sie die dargestellte Klassenstruktur in einem Package `de.fh_lu.o2s.personen`

# Beispiel

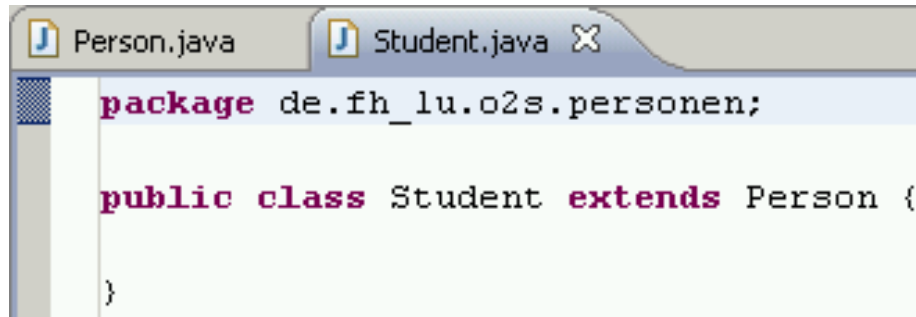
- Lösung, Schritt 1: Student als Unterklasse von Person
  - Legen Sie das Package `de.fh_lu.o2s.personen` an wie in Vorlesung 1.
  - Entwickeln Sie die Klasse `Person` (Fingerübung) oder kopieren Sie sie von anderswo, z.B. aus den Daten von Übung 1.
  - Legen Sie eine neue Klasse `Student` an und geben Sie bei der Erzeugung die Oberklasse (Superclass) `Person` an.





# Beispiel

- Lösung, Schritt 1, Forts.: Ergebnis in Java
  - Student extends Person, d.h.
    - Student erweitert Person bzw.
    - Student ist Unterklasse von Person:



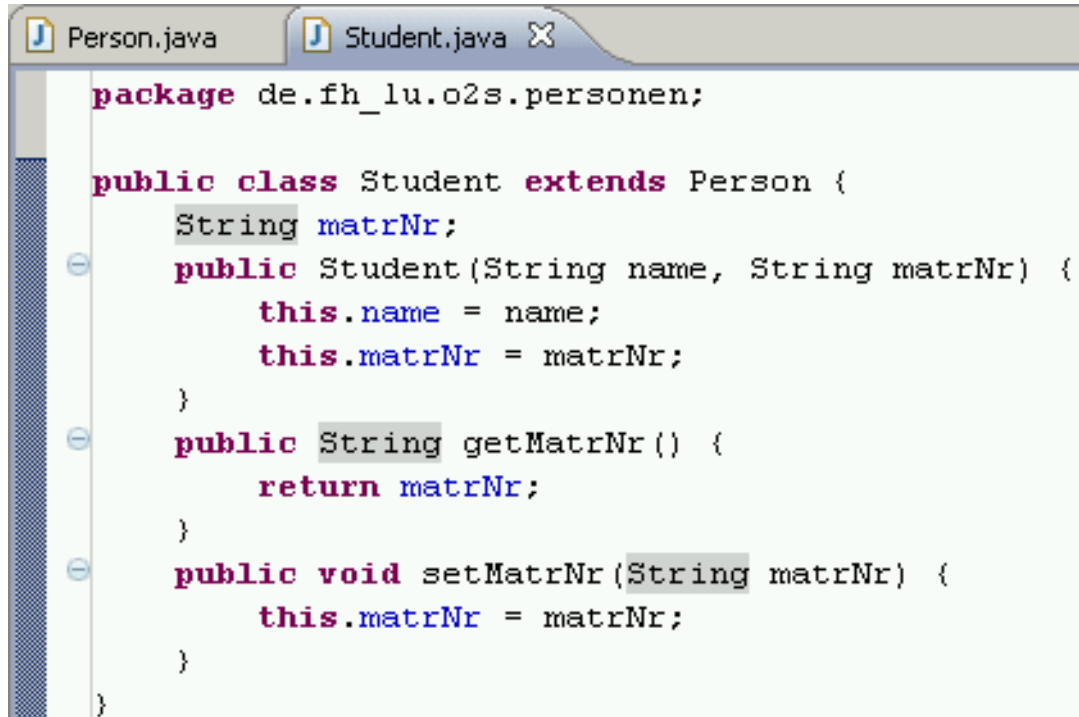
```
Person.java Student.java X
package de.fh_lu.o2s.personen;

public class Student extends Person {

}
```

# Student

- Lösung, Schritt 2: Vollständige Klasse `Student`



```
package de.fh_lu.o2s.personen;

public class Student extends Person {
    String matrNr;
    public Student(String name, String matrNr) {
        this.name = name;
        this.matrNr = matrNr;
    }
    public String getMatrNr() {
        return matrNr;
    }
    public void setMatrNr(String matrNr) {
        this.matrNr = matrNr;
    }
}
```

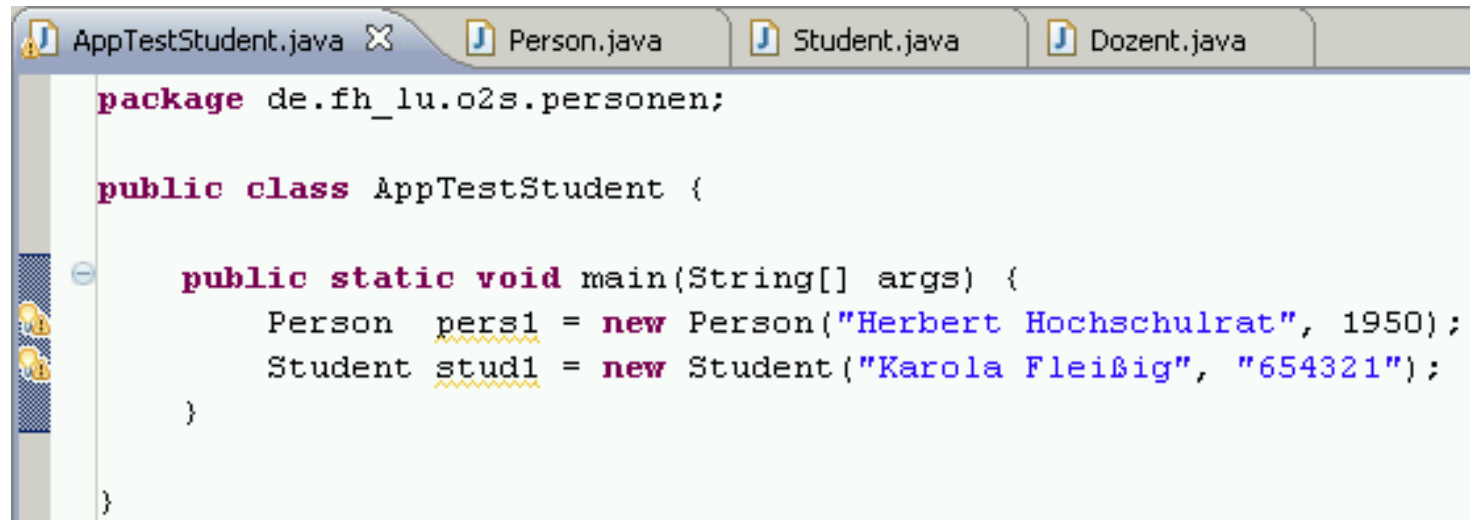
- Zusätzlich besitzen Objekte vom Typ `Student` alle Attribute und Methoden von `Person`. Sie werden geerbt.
- Lösung, Schritt 3: Klasse `Dozent`: Übungsaufgabe oder aus den Materialfiles kopieren

# Testapplikation

- Aufgabe 2: Implementieren Sie eine Applikation `AppTestStudent`, in der Sie
  - eine Person erzeugen, die kein Student ist
  - eine Studentin erzeugen
  - die Methoden `setGeburtsjahr()`, `getName()`, `getMatrNr()` und `toString()` sinnvoll einsetzen.
- Lösungsansatz:
  - Konstruktor `Person(name, geburtsjahr)` verwenden
  - Konstruktor `Student(name, matrNr)` verwenden
  - Der Studentin mit `setGeburtsjahr()` ein Geburtsjahr zuweisen
  - Beide Objekte mit `toString()` anzeigen
  - `name` und `matrnr` von beiden Objekten separat anzeigen

# Testapplikation

- Lösung, Schritt 1:
  - Klasse `AppTestStudent` mit `main()`-Methode,
  - Zwei Objekte erzeugen mit den genannten Konstruktoren



```
AppTestStudent.java X Person.java Student.java Dozent.java

package de.fh_lu.o2s.personen;

public class AppTestStudent {

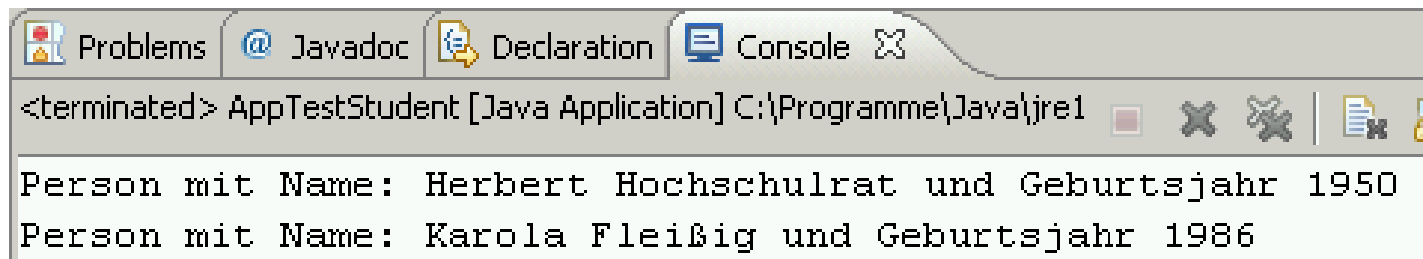
    public static void main(String[] args) {
        Person pers1 = new Person("Herbert Hochschulrat", 1950);
        Student stud1 = new Student("Karola Fleißig", "654321");
    }
}
```

# Testapplikation

- Lösung, Schritt 2:
  - Geburtsjahr zuweisen und beide Objekte anzeigen.

```
public static void main(String[] args) {  
    Person pers1 = new Person("Herbert Hochschulrat", 1950);  
    Student stud1 = new Student("Karola Fleißig", "654321");  
    stud1.setGeburtsjahr(1986);  
    System.out.println(pers1.toString());  
    System.out.println(stud1.toString());  
}
```

- Applikation ausführen: Run As → Java Application
- Konsolenausgabe:



The screenshot shows the Eclipse IDE's console window. The title bar includes tabs for Problems, Javadoc, Declaration, and Console. The console output shows the application has terminated and displays the toString() output for both objects: "Person mit Name: Herbert Hochschulrat und Geburtsjahr 1950" and "Person mit Name: Karola Fleißig und Geburtsjahr 1986".

```
<terminated> AppTestStudent [Java Application] C:\Programme\Java\jre1  
Person mit Name: Herbert Hochschulrat und Geburtsjahr 1950  
Person mit Name: Karola Fleißig und Geburtsjahr 1986
```

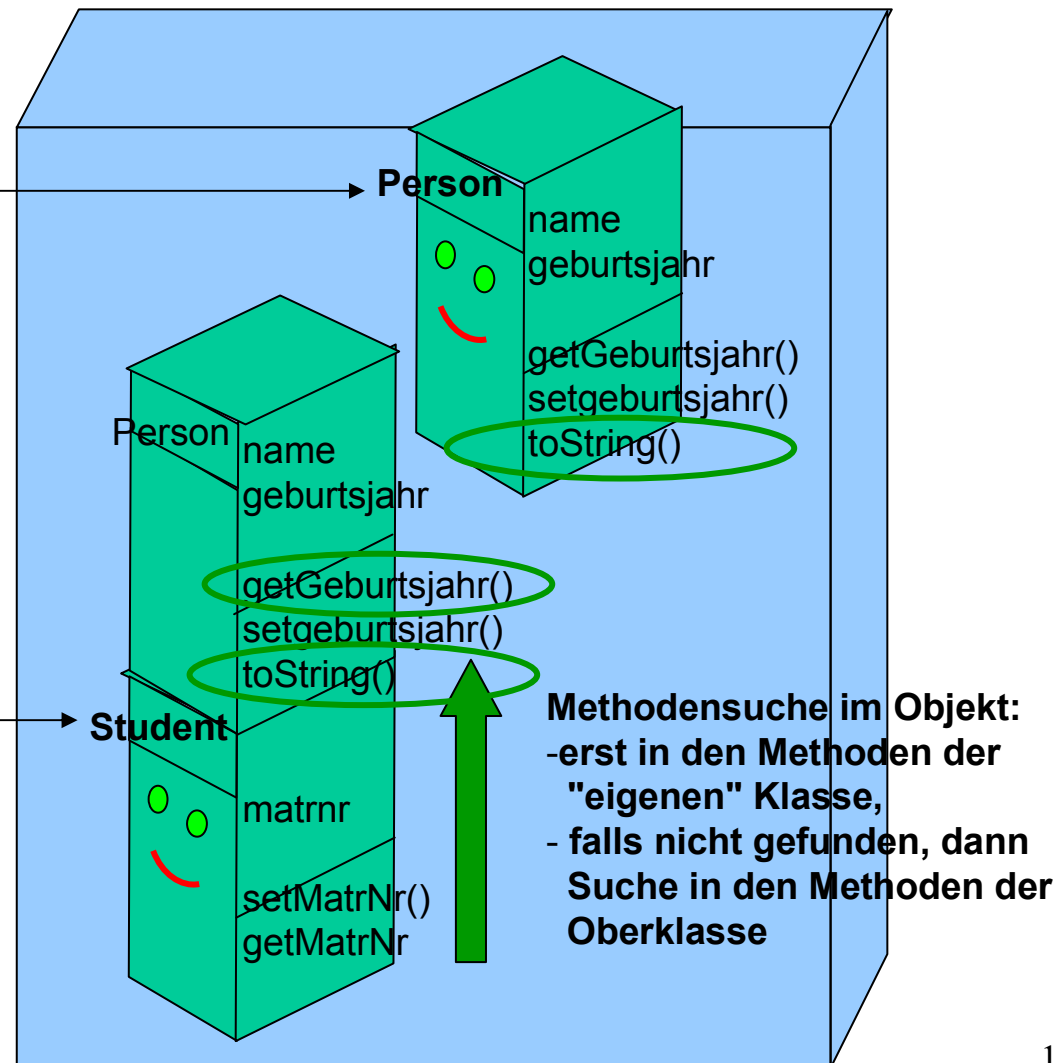
# Im Hauptspeicher

## Person pers1

- pers1.toString(): ok

## Student stud1

- stud1.setGeburtsjahr(): in Student  
nicht gefunden, Suche in Person: ok  
- toString(): Ebenso



# Testapplikation

- Lösung, Schritt 3, erster Versuch:
  - getName() und getMatrNr() verwenden

```
public static void main(String[] args) {  
    Person pers1 = new Person("Herbert Hochschulrat", 1950);  
    Student stud1 = new Student("Karola Fleißig", "654321");  
    stud1.setGeburtsjahr(1986);  
    System.out.println(pers1.toString());  
    System.out.println(stud1.toString());  
    System.out.println(pers1.getName());  
    System.out.println(pers1.getMatrNr());  
    System.out.println(stud1.getName());  
    System.out.println(stud1.getMatrNr());  
}
```

The method getMatrNr() is undefined for the type Person

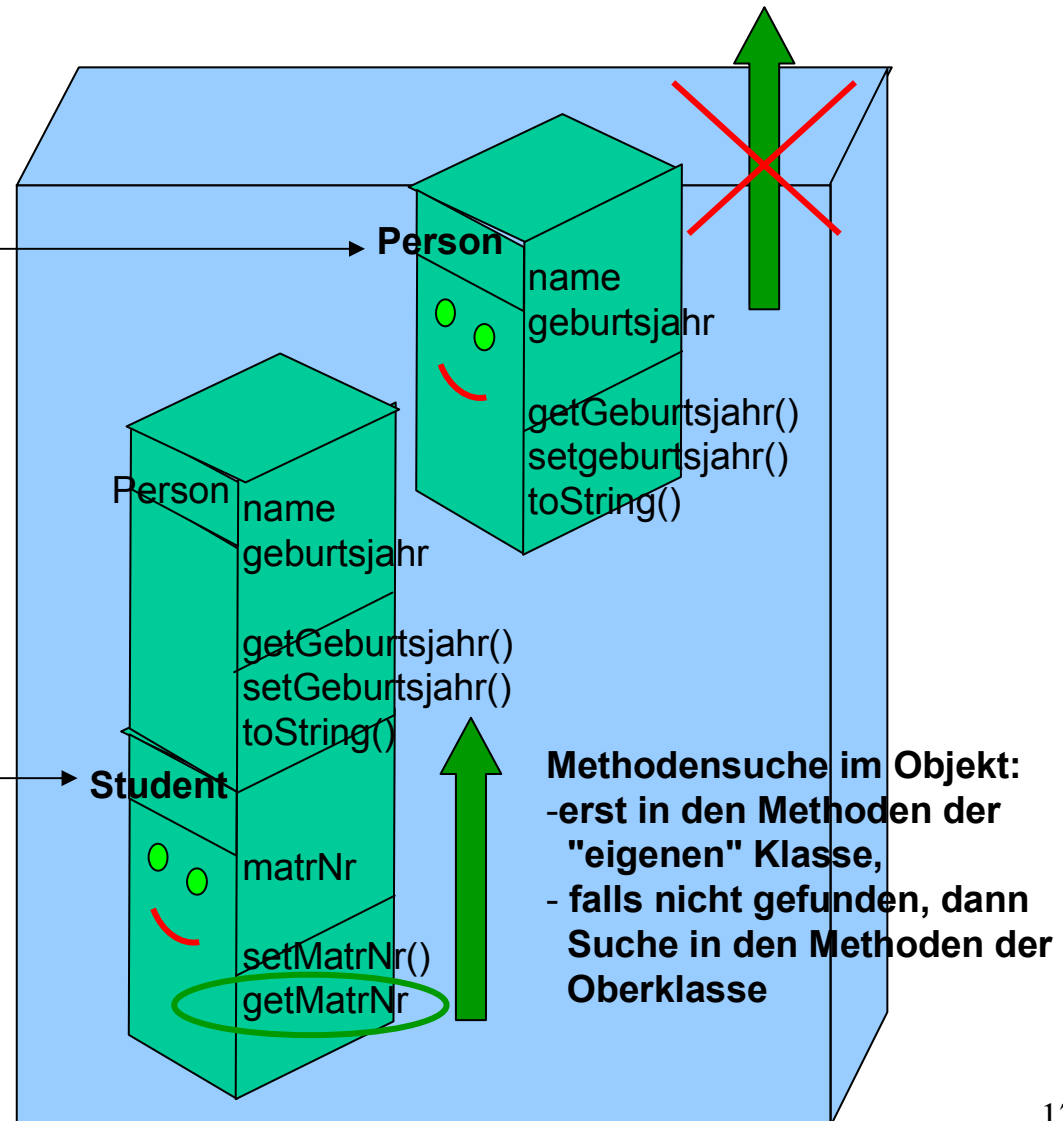
# Im Hauptspeicher

## Person pers1

pers1.toString(): ok  
pers1.getMatrNr(): **geht nicht**

## Student stud1

- stud1.setGeburtsjahr(): in Student nicht gefunden, Suche in Person: ok
- toString(): Ebenso
- pers1.getMatrNr(): Sofort ok



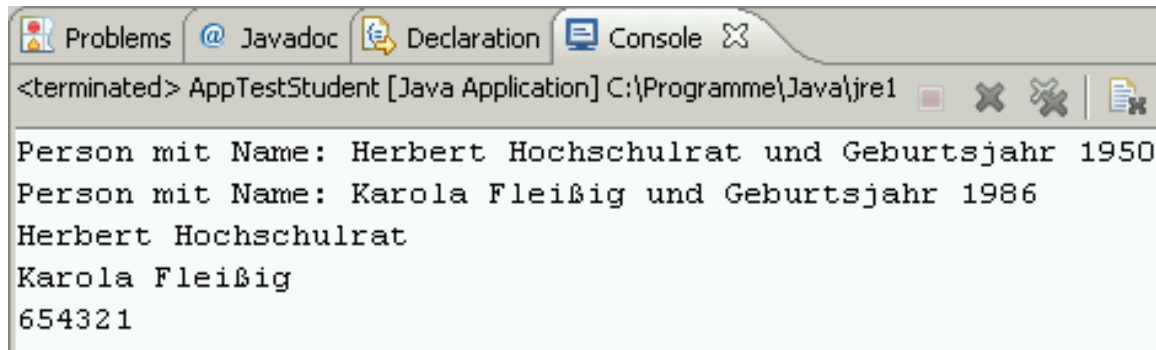


# Testapplikation

- Lösung, Schritt 3, korrigiert (Fehler auskommentiert):
  - getName() und getMatrNr() verwenden

```
public static void main(String[] args) {  
    Person pers1 = new Person("Herbert Hochschulrat", 1950);  
    Student stud1 = new Student("Karola Fleißig", "654321");  
    stud1.setGeburtsjahr(1986);  
    System.out.println(pers1.toString());  
    System.out.println(stud1.toString());  
    System.out.println(pers1.getName());  
    // System.out.println(pers1.getMatrNr());  
    System.out.println(stud1.getName());  
    System.out.println(stud1.getMatrNr());  
}
```

- Konsolenausgabe:



```
<terminated> AppTestStudent [Java Application] C:\Programme\Java\jre1  
Person mit Name: Herbert Hochschulrat und Geburtsjahr 1950  
Person mit Name: Karola Fleißig und Geburtsjahr 1986  
Herbert Hochschulrat  
Karola Fleißig  
654321
```

# Personen anzeigen

- Personen und Studenten werden mit derselben – in der Klasse `Person` definierten – Methode `toString()` angezeigt:

```
public String toString(){  
    return "Person mit Name: " + name +  
           " und Geburtsjahr " + geburtsjahr;  
}
```

- Ergebnis

```
Person mit Name: Herbert Hochschulrat und Geburtsjahr 1950  
Person mit Name: Karola Fleißig und Geburtsjahr 1986
```

- Beobachtung:
  - Die Methode unterscheidet nicht zwischen `Person` und `Student`
  - Sie weiß nichts von Matrikelnummern
  - ➔ Die Methode ist für Studenten eigentlich ungeeignet

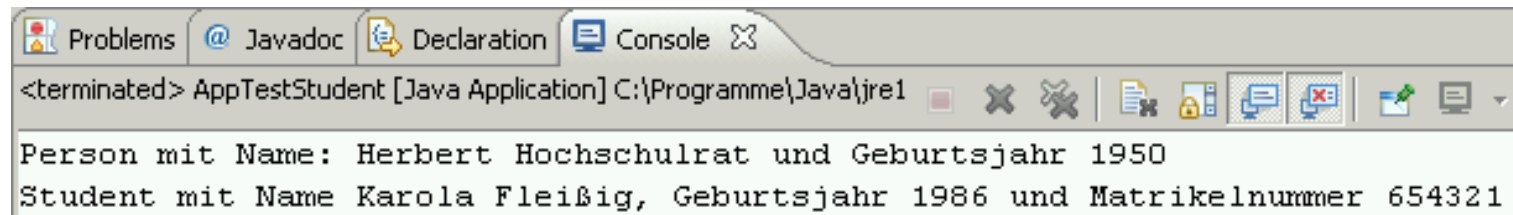
# Studenten anzeigen

- Aufgabe:
  - Implementieren Sie in der Klasse Student eine eigene Methode `toString()`, die auch die Matrikelnummer anzeigt.

- Lösung:

```
public String toString(){  
    return "Student mit Name " + this.getName() +  
        ", Geburtsjahr " + this.getGeburtsjahr() +  
        " und Matrikelnummer " + this.getMatrNr();  
}
```

- Test:
  - Durch Ausführung der unveränderten Applikation `AppTestStudent`:



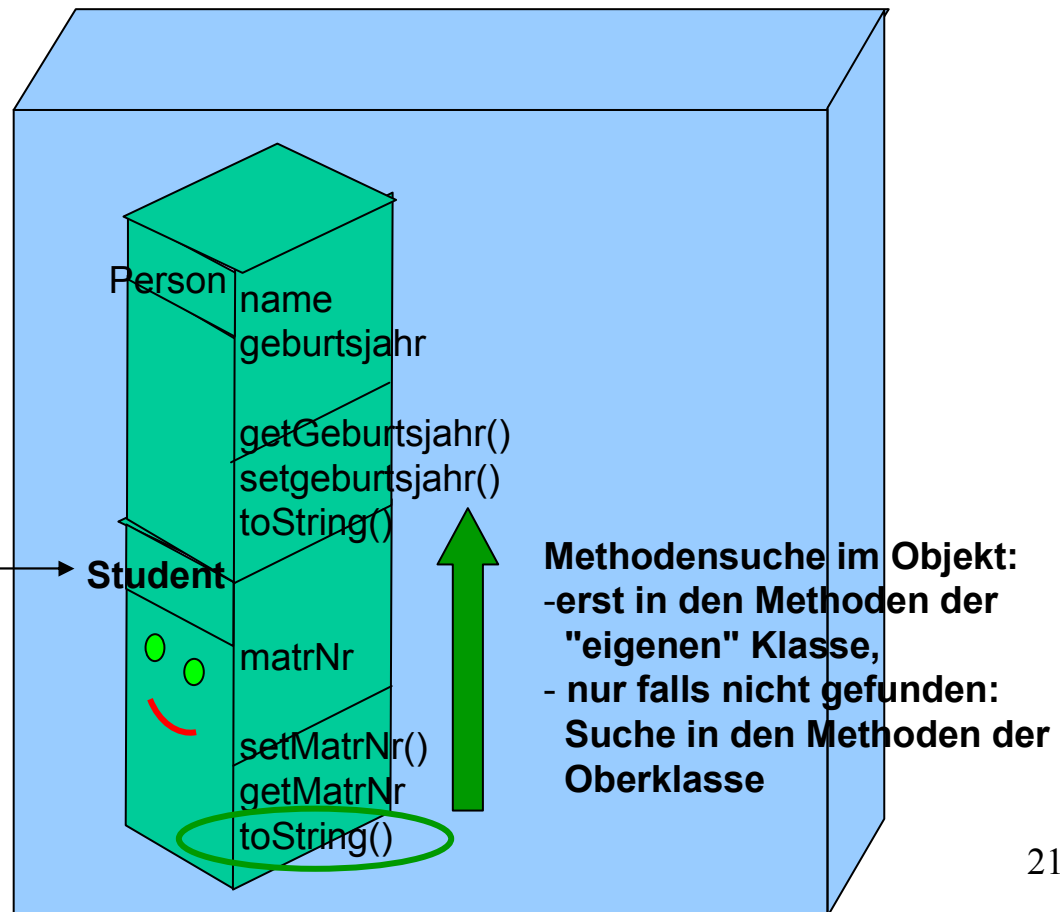
```
<terminated> AppTestStudent [Java Application] C:\Programme\Java\jre1  
Person mit Name: Herbert Hochschulrat und Geburtsjahr 1950  
Student mit Name Karola Fleißig, Geburtsjahr 1986 und Matrikelnummer 654321
```

# Studenten anzeigen

- Beobachtung:
  - Obwohl die Applikation nicht verändert wurde, wurde die neue toString()-Methode ausgeführt.

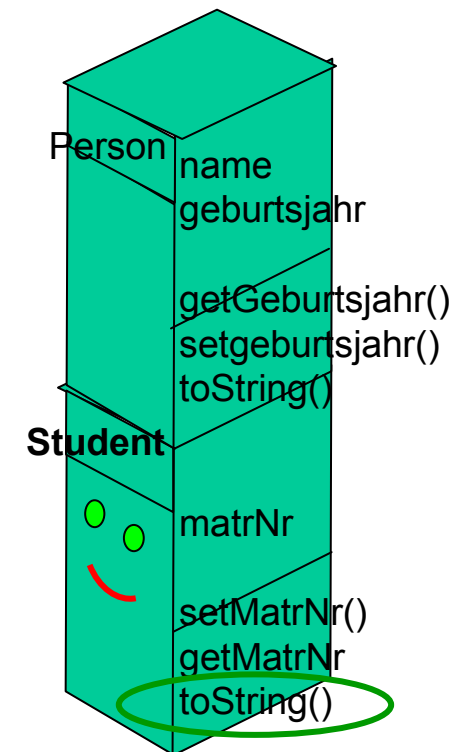
## Student stud1

- stud1.setGeburtsjahr(): in Student nicht gefunden, Suche in Person: ok
- toString(): Ebenso
- pers1.getMatrNr(): Sofort ok



# Methoden überschreiben

- Erklärung:
  - Die Methode `toString()` (aus der Klasse `Person`) wurde in der Unterklasse `Student` überschrieben.
  - In beiden Klassen hat `toString()` denselben Methodenkopf.
- Erinnerung:
  - Gleichheit von Methodenköpfen:
    - derselbe Methodename **und**
    - dieselben Parametertypen –  
bei `toString()`: Jeweils keine Parameter
  - Überladen von Methoden:
    - derselbe Methodename
    - unterschiedliche Parametertypen
- Anmerkung:
  - Überschreiben geht nur in Unterklassen
  - Überladen geht auch in derselben Klasse



# Gleichheit von Methoden

- Aufgabe 3: Experimentieren Sie in der Klasse `Student` mit einer `toString()`-Methode, die
  - keine Rückgabe liefert (`void`), sondern
  - den erzeugten String direkt in die Konsole schreibt.
- Lösung:

```
public void toString() {  
    System.out.println("Student mit Name " + this.getName() +  
        ", Geburtsjahr " + this.getGeburtsjahr() +  
        " und Matrikelnummer " + this.getMatrNr());  
}
```

The return type is incompatible with Person.toString()  
Press 'F2' for focus.

# Gleichheit von Methoden

- Beobachtung:
  - Geht nicht, weil der Rückgabedatentyp in `Student.toString()`  
`public void toString() {`  
derselbe sein muss wie in `Person.toString()`  
`public String toString() {`
- Erklärung:
  - Zum Überschreiben gehören identische Methodenköpfe
  - der Rückgabedatentyp gehört auch zum Methodenkopf
- Folgerung:
  - Wenn Namen und Parametertypen gleich sind, dann **muss** auch der Rückgabedatentyp gleich sein:
    - In Person:  
`public String toString() {`
    - In Student:  
`public String toString() {`

# Sichtbarkeit von Methoden

- Aufgabe 4: Experimentieren Sie
  - in der Klasse `Student` mit einer `toString()`-Methode, die
  - die Sichtbarkeit `private` hat (Zugriff nur durch Objekte derselben Klasse)

- Lösung:

```
private String toString(){  
    return "Student mit Name " + this.getName() +  
           ", Geburtsjahr " + this.getGeburtsjahr() +  
           " und Matrikelnummer " + this.getMatrNr();  
}
```

Cannot reduce the visibility of the inherited method from Person

Press 'F2' for focus.



- Beobachtung:
    - Geht nicht, weil die Sichtbarkeit in `Student.toString()` **nicht geringer** sein darf als in `Person.toString()`
    - `private toString()` geht auch in `Person` nicht, weil es noch eine Oberklasse `Object` gibt, in der `toString()` bereits mit Sichtbarkeit `public` definiert ist.
- ➔ Später mehr dazu.

# Codeverdopplung vermeiden

- Beobachtung:
  - In `Person.toString()` und `Student.toString()` wird teilweise derselbe Code implementiert:

```
public String toString(){// In Person
    return "Person mit Name: " + this.getName() +
           ", Geburtsjahr " + this.getGeburtsjahr();
}

public String toString(){// In Student
    return "Student mit Name " + this.getName() +
           ", Geburtsjahr " + this.getGeburtsjahr() +
           ", Matrikelnummer " + this.getMatrNr();
}
```

- Aufgabe 5:
  - Passen Sie die Methode `Student.toString()` so an, dass kein Code mehr doppelt verwendet wird.
- Lösungsansatz:
  - Von `Student.toString()` auf den Code aus `Person.toString()` zugreifen.

# Codeverdopplung vermeiden

- Lösung:

```
public String toString(){// In Person
    return "Person mit Name: " + this.getName() +
           ", Geburtsjahr " + this.getGeburtsjahr();
}

public String toString(){// In Student
    return super.toString() +
           ", Matrikelnummer " + this.getMatrNr();
}
```

- Erklärung:

- Das Schlüsselwort `super` bezeichnet wie `this` das aufrufende Objekt, aber
  - bei Zugriff mit `this` wird zuerst in den eigenen Methoden gesucht,
  - bei Zugriff mit `super` sind die eigenen Methoden verboten und es wird sofort in der Oberklasse (in den Oberklassen) gesucht.
- Hier liefert `super.toString()` den Rückgabewert der Methode `Person.toString()`, an den dann noch die Matrikelnummer angehängt wird.
- Ausführung mit `AppTestStudent`:

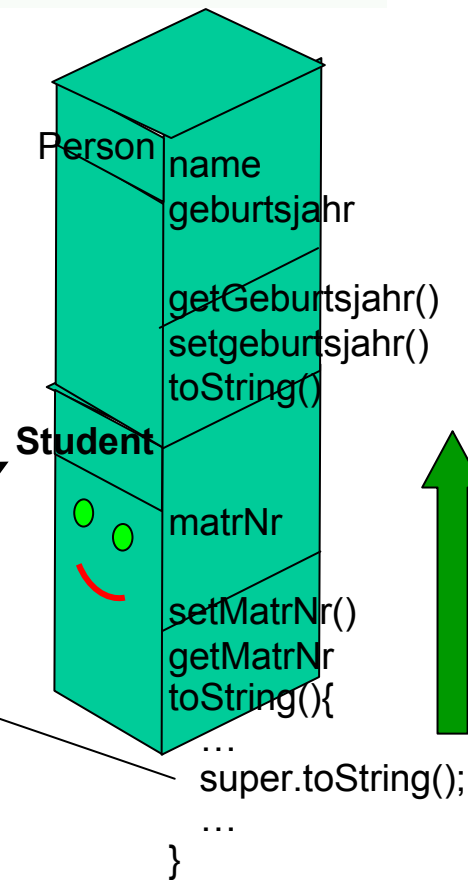
```
Person mit Name: Herbert Hochschulrat, Geburtsjahr 1950
Person mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321
```

# super

```
public String toString(){// In Student  
    return super.toString() +  
        ", Matrikelnummer " + this.getMatrNr();  
}
```

## Student stud1

- stud1.toString(): In Student gefunden
- super.toString()
- Aufruf geht an dasselbe Objekt
- Die Methodensuche beginnt erst bei den Methoden der Oberklasse



Methodensuche  
bei Aufruf mit  
**super**

Methodensuche  
bei direktem  
Aufruf oder bei  
Aufruf mit **this**

Bei Aufruf mit  
**super** sind diese  
verboten

# Codeverdopplung vermeiden

- Beobachtung:
  - Die bisherige Lösung behauptet, ein Student sei eine Person.
  - Das ist zwar nicht falsch (Student is-a Person), aber wir können das noch besser!

```
public String toString(){// In Person
    return "Person mit Name: " + this.getName() +
        ", Geburtsjahr " + this.getGeburtsjahr();
}

public String toString(){// In Student
    return super.toString() +
        ", Matrikelnummer " + this.getMatrNr();
}
```

```
Person mit Name: Herbert Hochschulrat, Geburtsjahr 1950
```

```
Person mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321
```

- Aufgabe 6:
  - Passen Sie die Methode `Person.toString()` so an, dass immer der richtige Klassenname ausgegeben wird.

# getClass().getName()

- Lösungsansatz:
  - Jedes Objekt kann den eigenen Klassennamen (als String) zurückliefern, und zwar

- incl. Package-Name mit `getClass().getName()`

```
de.fh_lu.o2s.personen.Person mit Name: Herbert Hochschulrat, Gek  
de.fh_lu.o2s.personen.Student mit Name: Karola Fleißig, Geburtsj
```

- ohne Package-Name mit `getClass().getSimpleName()`

```
Person mit Name: Herbert Hochschulrat, Geburtsjahr 1950  
Student mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321
```

- Lösung:

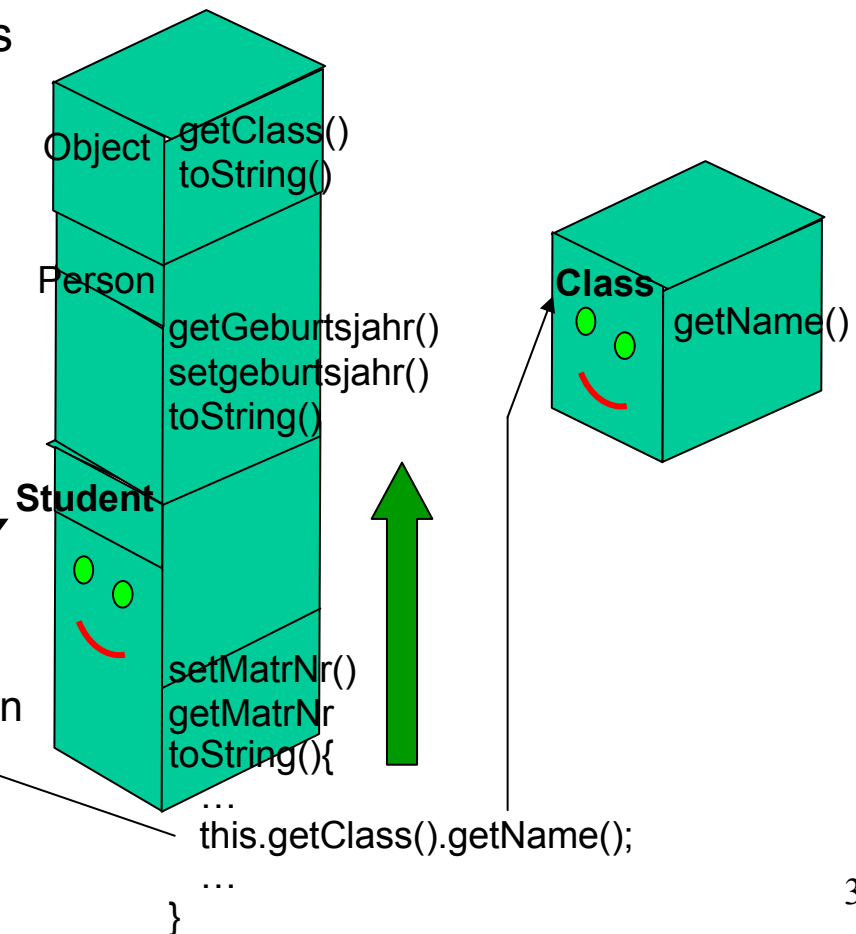
```
public String toString(){// In Person  
    return this.getClass().getSimpleName() +  
           " mit Name: " + this.getName() +  
           ", Geburtsjahr " + this.getGeburtsjahr();  
}
```

# getClass().getName()

- **Genauer:**
  - Jedes Objekt liefert bei `getClass()` seine Klasse und
  - diese liefert bei `getName()` bzw. `getSimpleName()` Ihren Namen.
  - `getClass()` wird übrigens von der Oberklasse `Object` geerbt.

## Student stud1

- `stud1.toString()`: In `Student` gefunden
  - `this.getClass()`
  - Aufruf geht an dasselbe Objekt
  - `getClass()` wird erst in `Object` gefunden
  - Rückgabedatentyp "Class"
  - Methodenaufruf `getSimpleName()` im `Class`-Objekt



# Weiteres Beispiel

- Aufgabe 7:
  - Implementieren Sie in `Person` eine Methode `druckDich()`, die die Person auf die Konsole schreibt.
  - Verwenden Sie diese Methode in Ihrer `AppTestStudent` und verfolgen Sie die ausgelösten Methodenaufrufe.
- Lösungsansatz: Die Methode `druckDich()`
  - erhält Sichtbarkeit `public` und Rückgabedatentyp `void`
  - ruft `toString()` auf
  - schreibt den zurückgegebenen String mit `System.out.println()` auf die Konsole.



# druckDich()

- Lösung: `Person.druckDich()` und `AppTestStudent`

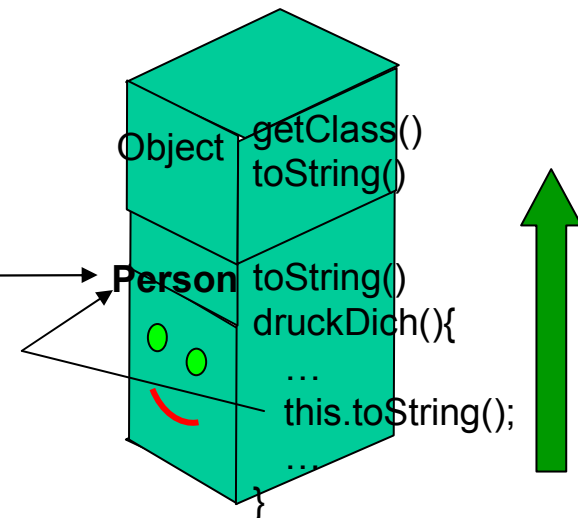
```
public void druckDich(){  
    System.out.println(this.toString());  
}
```

```
pers1.druckDich();  
stud1.druckDich();
```

```
Person mit Name: Herbert Hochschulrat, Geburtsjahr 1950  
Student mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321
```

## Person pers1

- `pers1.druckDich()`: In `Person` gefunden
  - `this.toString()`
  - Aufruf geht an dasselbe Objekt
  - `toString()` wird in `Person` gefunden
  - Rückgabedatentyp `String`
  - Anzeige mit `System.out.println(...)`;



# druckDich()

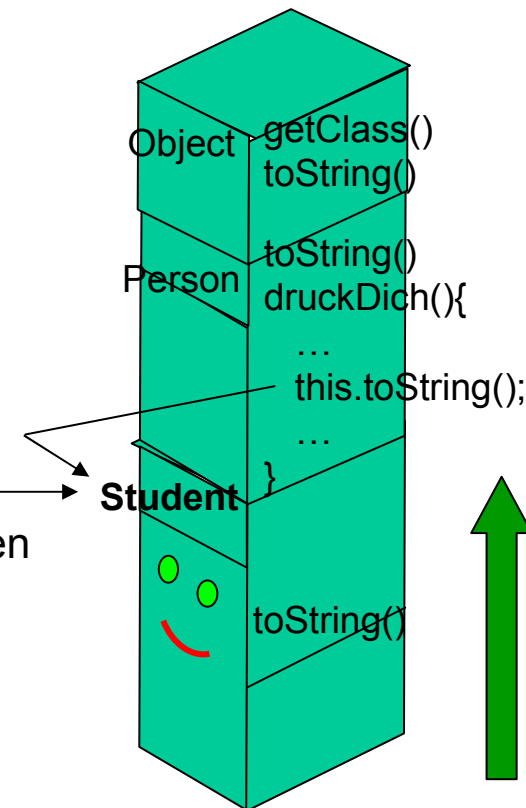
- Lösung: Student.druckDich() und AppTestStudent

```
public void druckDich(){  
    System.out.println(this.toString());  
}
```

```
pers1.druckDich();  
stud1.druckDich();
```

## Student stud1

- stud1.druckDich(): In Student nicht gefunden
  - aber in Person gefunden
  - this.toString()
  - Aufruf geht an dasselbe Objekt (stud1)
  - toString() wird in Student gefunden
  - Rückgabetyp String
  - Anzeige mit System.out.println(...);



# druckDich()

```
public void druckDich(){  
    System.out.println(this.toString());  
}
```

```
pers1.druckDich();  
stud1.druckDich();
```

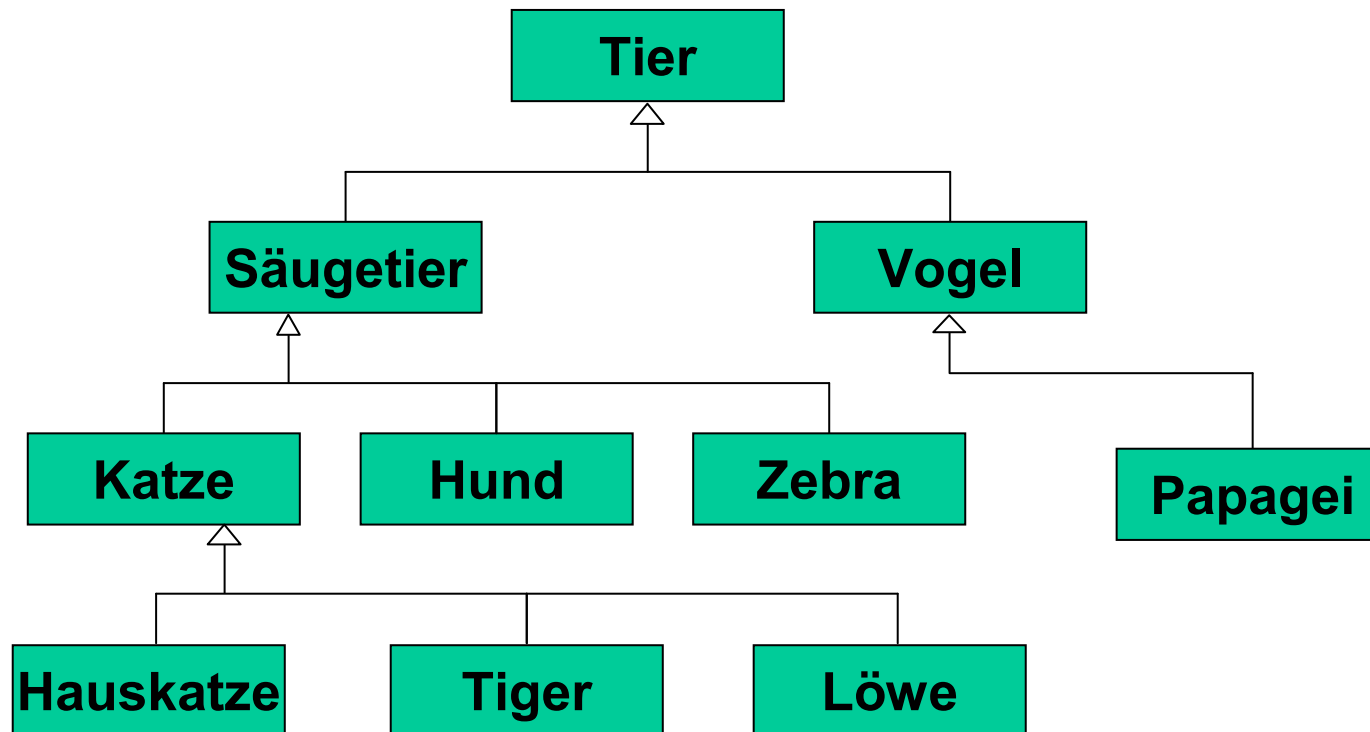
- Beobachtung zu `stud1.druckDich()`:
  - Obwohl die Methode `druckDich()` in `Person` definiert ist, ruft `this.toString()`
    - nicht die `toString()`-Methode aus `Person` auf,
    - sondern die `toString()`-Methode aus `Student`
- Erklärung:
  - `this` bezeichnet das aktuelle Objekt.
  - Innerhalb von `druckDich()` ist das das Objekt, dem die Methode `druckDich()` geschickt wurde,
  - in unserem Fall also das Objekt, auf das die Referenz `stud1` zeigt.
  - Das ist ein Objekt vom Typ `Student`, führt also auf den Aufruf `toString()` die eigene `toString()`-Methode aus.

# Dynamisches Binden

- Unter Binden versteht man
  - die Zusammenführung von Methodenaufrufen und Methodencode durch den Compiler bzw. Linker
- Die Methode `druckDich()` kann zur Laufzeit
  - entweder `Person.toString()` oder
  - `Student.toString()` aufrufen.
- Das Binden kann also erst zur Laufzeit gemacht werden
  - Dynamisches Binden (oder "spätes Binden")
- Man spricht auch von Polymorphie (Vielgestaltigkeit), weil eine Referenz, z.B. `this`, unterschiedliche Objekttypen bezeichnen kann.

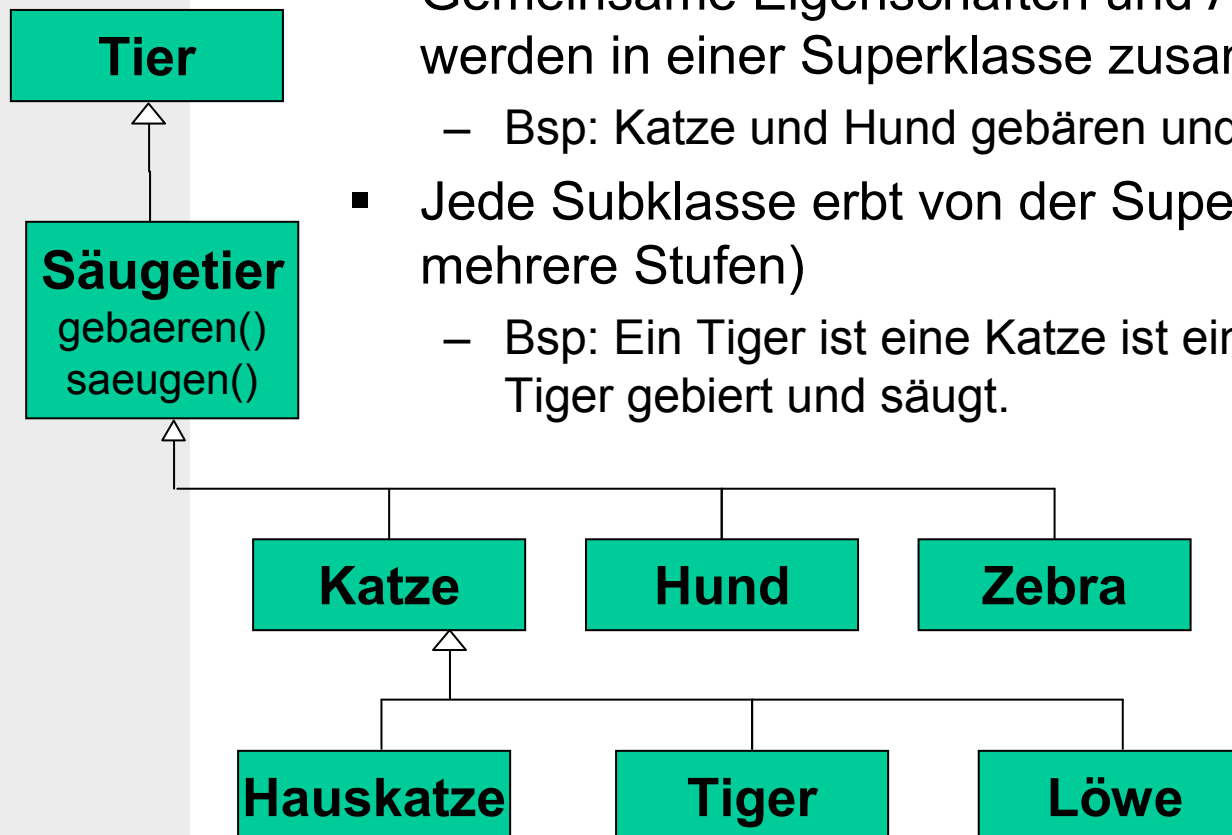
# Beispiel für Klassenhierarchien

- Vererbung kann über mehrere Stufen hinweg gehen  
→ Es entsteht eine Vererbungshierarchie, z.B.



# Grundprinzip der Vererbung

- Jede Objektart wird durch eine eigene Klasse (Typ) realisiert
  - Bsp: Katze oder Hund
- Gemeinsame Eigenschaften und Aktionsmöglichkeiten werden in einer Superklasse zusammengefasst
  - Bsp: Katze und Hund gebären und säugen → Säugetier
- Jede Subklasse erbt von der Superklasse (auch über mehrere Stufen)
  - Bsp: Ein Tiger ist eine Katze ist ein Säugetier: auch ein Tiger gebiert und säugt.



# Überschreiben von Methoden

- In einer Klassenhierarchie macht es oft Sinn, in verschiedenen Subklassen die gleiche Methode zu definieren
  - Bsp: Methode „friss()“ → man kann diese Methode für alle Tiere des Zoos aufrufen
- Wenn Methode in der Superklasse definiert wird, erben die Subklassen automatisch
  - Bsp: Art des Fressens unwichtig → Methode „friss()“ bei Tier implementieren
- Methoden können in der Subklasse überschrieben werden (müssen aber nicht)
  - Bsp: Ist für eine Tierart spezielles Fressverhalten wichtig → Methode „friss()“ in der Subklasse implementieren

# Gleichheit von Methoden

- Überschreiben bedeutet
  - „gleicher Name, gleiche Parametertypen“
  - dann muss auch der Rückgabetyt gleich sein.

- Bsp.:

```
public void friss(String futter){// In Tier
    this.incrementFutter();
}
public void friss(Futter futter){// In Tier überladen: ok
    this.incrementFutter();
}
public boolean friss(){// In Tier überladen: ok
    boolean aufgegessen = false;
    return aufgegessen;
}
public void friss(String futter){// In Tiger überschrieben: ok
    this.incrementMuchFutter();
}
public boolean friss(Futter futter){// In Katze nicht ok
    ... // Rückgabetyt nicht kompatibel mit der Elternmethode
}
```

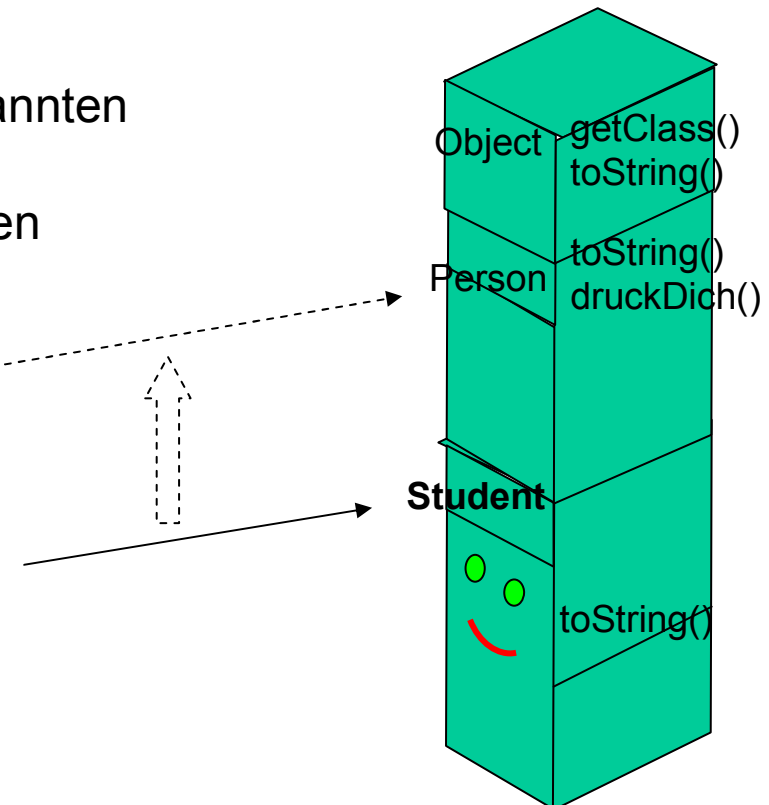


# Polymorphie

- Aufgabe 8: Ermitteln Sie experimentell, ob Sie
  - 1. einer Referenz (Variable) vom Typ `Person` auch einen `Student` zuweisen können,
  - 2. einer Referenz vom Typ `Student` auch eine `Person` zuweisen können.
- Lösungsansatz:
  - In `AppTestStudent` die genannten Variablen anlegen und die entsprechenden Zuweisungen ausprobieren

**Fall 1: `Person pers2 = stud1`**

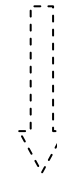
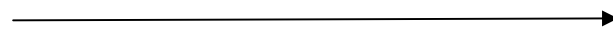
**Student stud1**



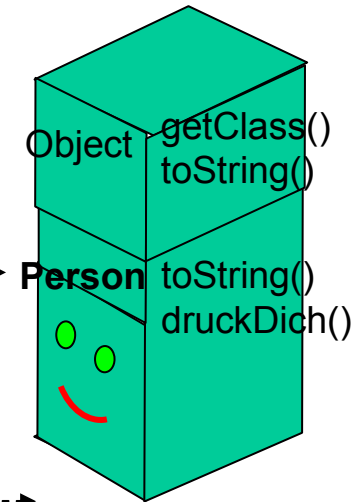
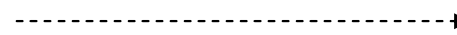
# Polymorphie

- Lösungsansatz, Forts.:

**Person pers1**



**Fall2: Student stud2 = pers1**



- Lösung durch Experiment:

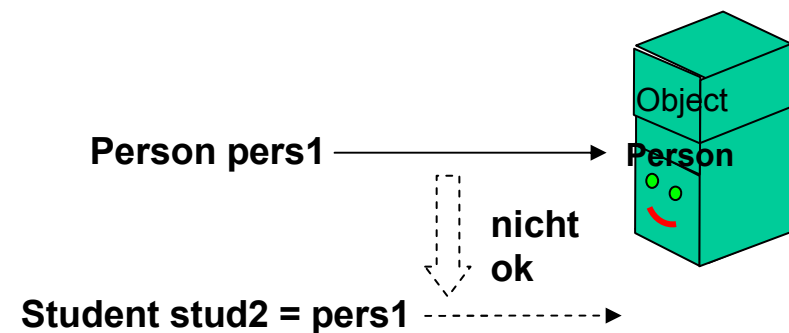
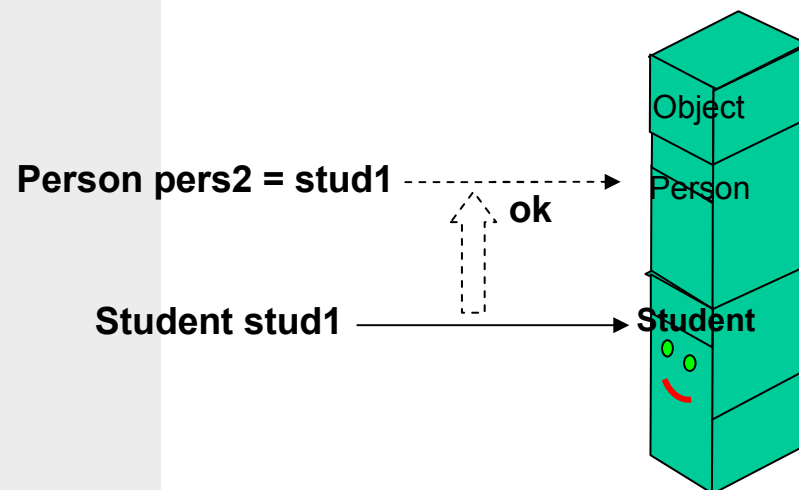
```
Person pers2 = stud1;  
Student stud2 = pers1;
```

Type mismatch: cannot convert from Person to Student  
Press 'F2' for focus.

# Polymorphie

- Erklärung 1:
  - Eine Variable vom Typ `Person` darf auf ein Objekt vom Typ `Student` zeigen, weil ein `Student` auch eine `Person` ist (`Student is-a Person`)
  - Eine Variable vom Typ `Student` darf nicht auf ein Objekt vom Typ `Person` zeigen, weil eine `Person` evtl. kein `Student` ist, bzw.

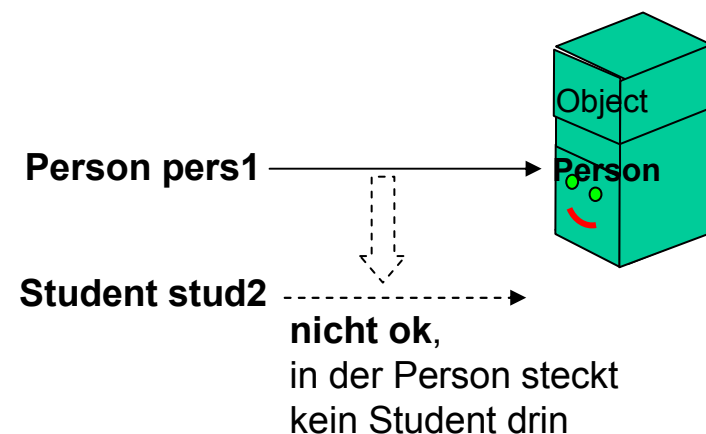
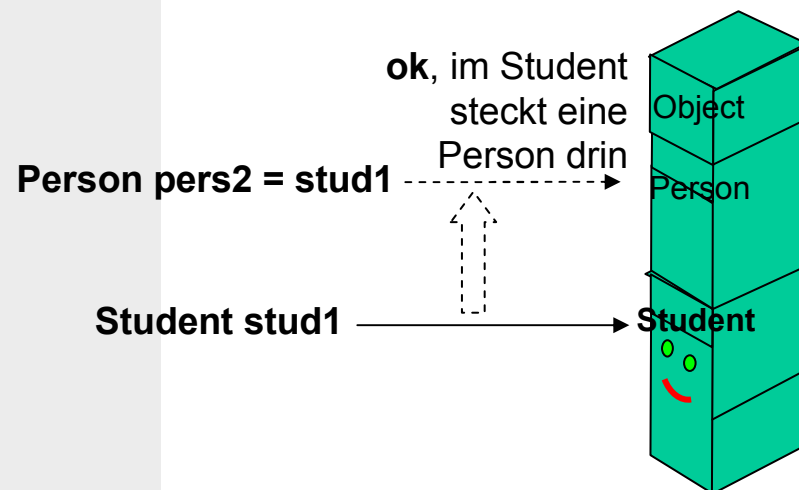
```
Person pers2 = stud1;  
Student stud2 = pers1;
```



# Polymorphie

- Erklärung 2:
  - Eine Variable vom Typ `Person` darf auf ein Objekt vom Typ `Student` zeigen, weil in dem `Student` eine `Person` drinsteckt
  - Eine Variable vom Typ `Student` darf nicht auf ein Objekt vom Typ `Person` zeigen, weil in der `Person` evtl. kein `Student` drinsteckt.

```
Person pers2 = stud1;  
Student stud2 = pers1;
```



# Polymorphie

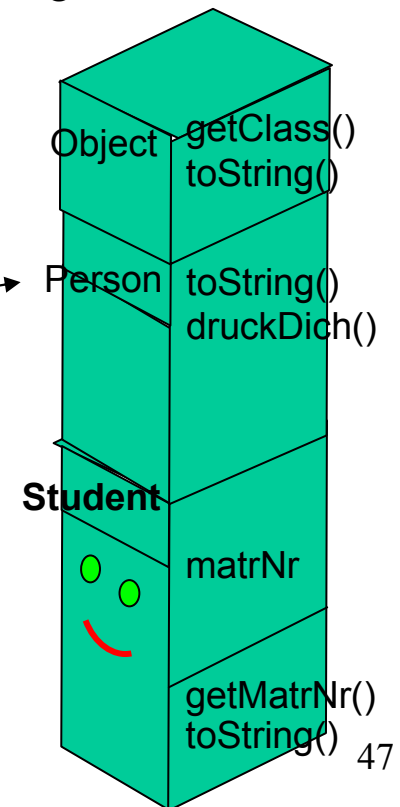
- Anmerkung:
  - Die Variable vom Typ `Person` kann also auf Objekte vom Typ `Person` oder einer Unterklasse von `Person` zeigen.
  - Dies ist das Standardbeispiel für Polymorphie: Eine Referenz kann auf unterschiedliche Objekttypen zeigen.

# Dynamisches Binden

- Aufgabe 9: Gehen Sie von der (erlaubten) Situation aus, dass
  - eine Referenz vom (Referenz-)Typ `Student` und
  - eine Referenz vom (Referenz-)Typ `Person`
  - auf dasselbe Objekt vom (Objekt-)Typ `Student` zeigen.
- Ermitteln Sie experimentell,
  - welche `toString()` Methode ausgeführt wird,
  - wenn Sie die Botschaft `toString()` über die unterschiedlichen Referenzen an dasselbe Objekt schicken.

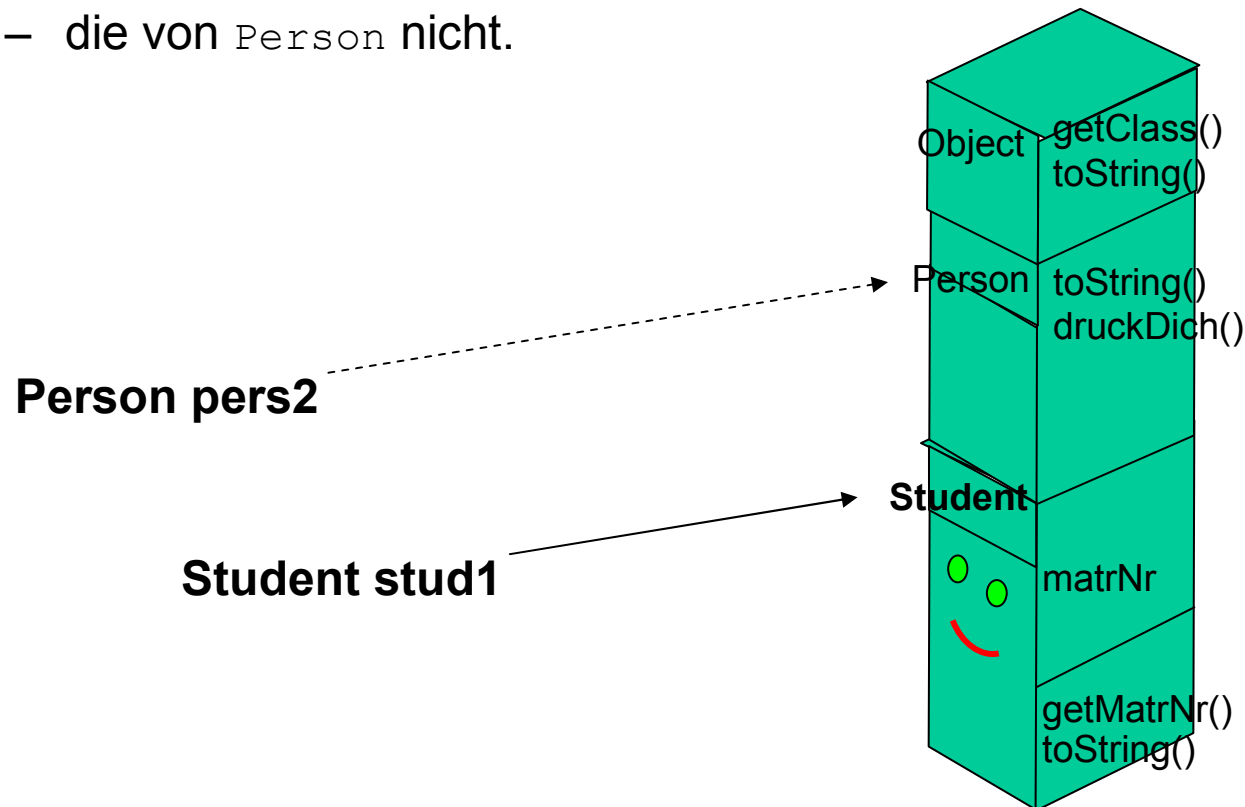
**Person pers2**

**Student stud1**



# Dynamisches Binden

- Lösungsansatz:
  - In `AppTestStudent` beide Methodenaufrufe ausprobieren,
  - die `toString()`-Methode von `Student` müsste eine Matrikelnummer ausgeben,
  - die von `Person` nicht.



# Dynamisches Binden

- Lösung: Java-Code

```
System.out.println(stud1.toString());  
System.out.println(pers2.toString());
```

- Konsol-Output:

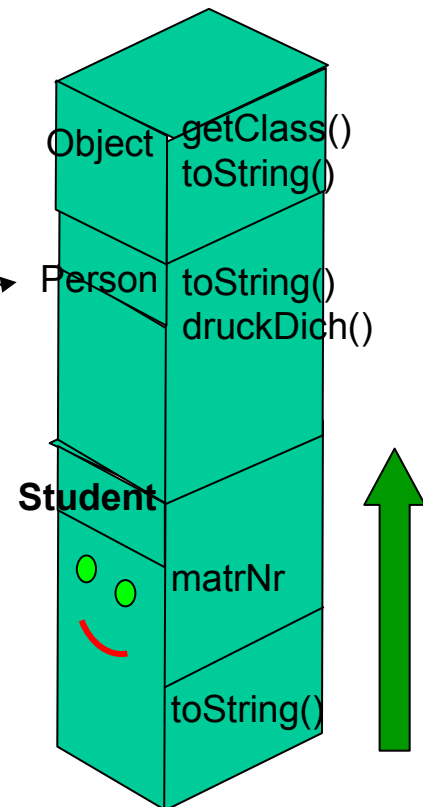
```
Student mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321  
Student mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321
```

- Beobachtung:

- Die Methodensuche beginnt in beiden Fällen immer ganz unten

**Person pers2**

**Student stud1**

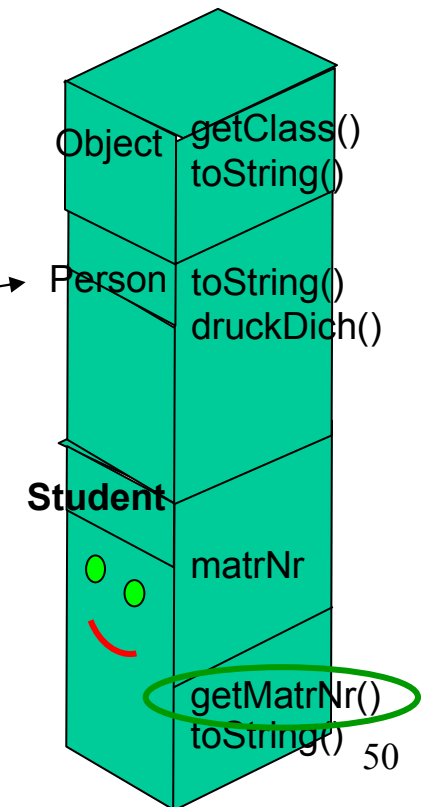




# Zulässige Methodenaufrufe

- Aufgabe 10: Gehen Sie nun von der Situation aus, dass
  - Sie nur eine Referenz vom (Referenz-)Typ `Person` haben, die
  - auf ein Objekt vom (Objekt-)Typ `Student` zeigt.
- Ermitteln Sie experimentell,
  - ob Sie die Objektmethode `getMatrNr()` ausführen können, indem Sie die `getMatrNr()` Botschaft an Ihre `Person`-Referenz schicken.

**Person pers2**



# Zulässige Methodenaufrufe

- Lösungsansatz:
  - In `AppTestStudent` ausprobieren,
- Lösung:
  - Es geht nicht.

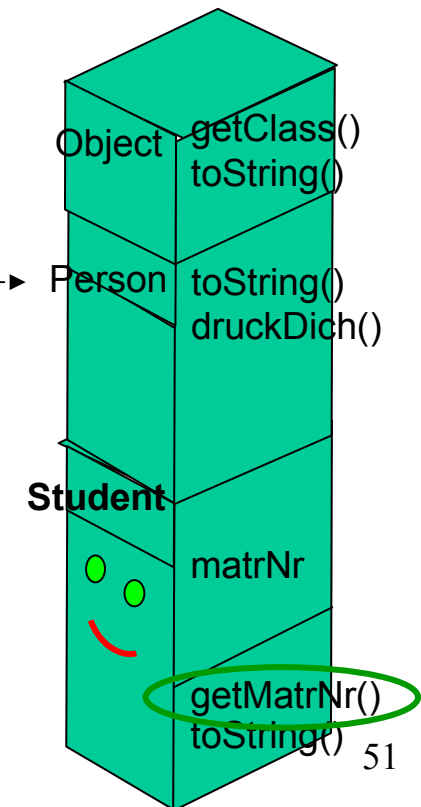
```
System.out.println(pers2.getMatrNr());
```

The method `getMatrNr()` is undefined for the type `Person`  
Press 'F2' for focus.

**Person pers2**

→ Folgerung zu Aufgabe 9 und 10:

1. Damit eine Methode ausgeführt wird, muss bereits der Referenztyp sie kennen,
2. es wird aber die Methode des Objekttyps ausgeführt

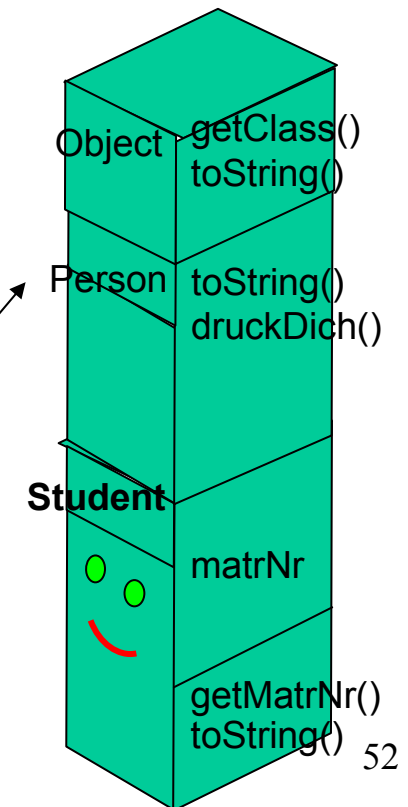


# Casting

- Aufgabe 11:
  - Gehen Sie von derselben Situation aus wie in der vorigen Aufgabe.
  - Finden Sie einen Weg, die Methode `getMatrNr()` **des** `Student`-Objekts trotzdem aufzurufen.
- Lösungsansatz:
  - Wir brauchen eine Referenz vom (Referenz-)Typ `Student`.
  - Wir probieren es mit Casting:  

```
Student stud3 = (Student) pers2.
```
- Anmerkung:
  - Ohne Casting geht es nicht (vgl. Aufgabe 8)

**Person pers2**



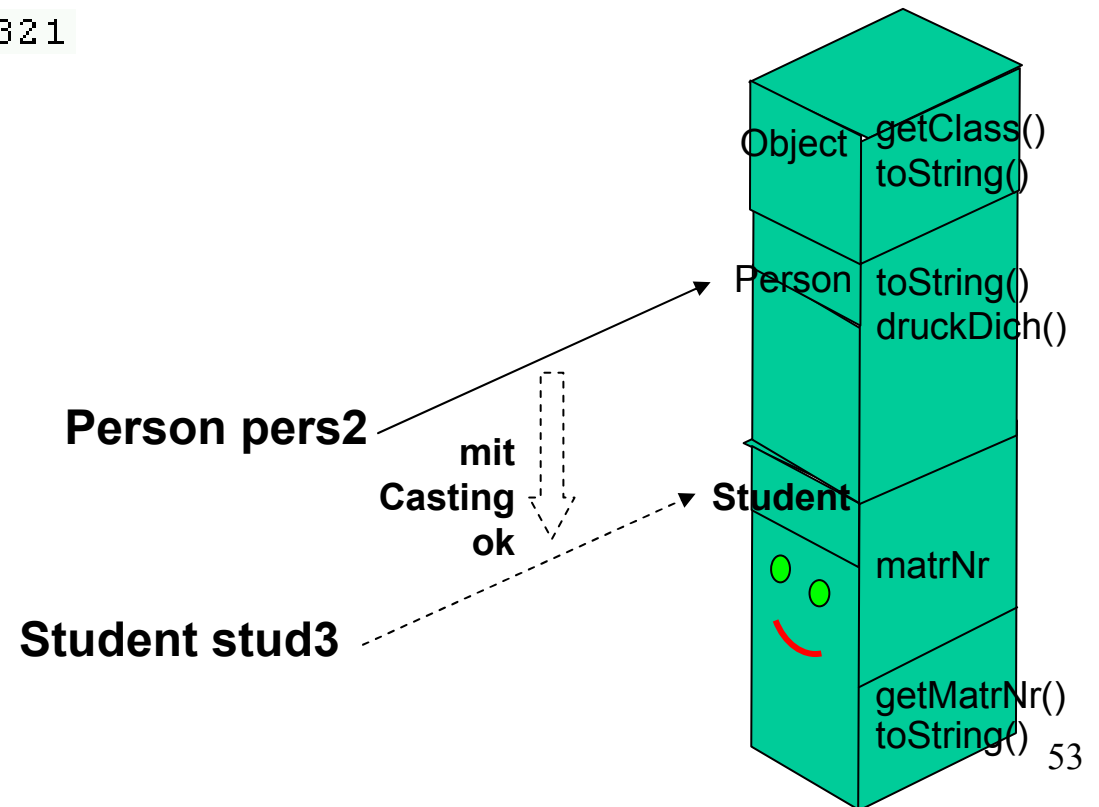
# Casting

- Lösung, Java-Code:

```
Student stud3 = (Student)pers2;  
System.out.println("Matrikelnummer: " + stud3.getMatrNr());
```

- Konsol-Output:

```
Matrikelnummer: 654321
```



# Casting

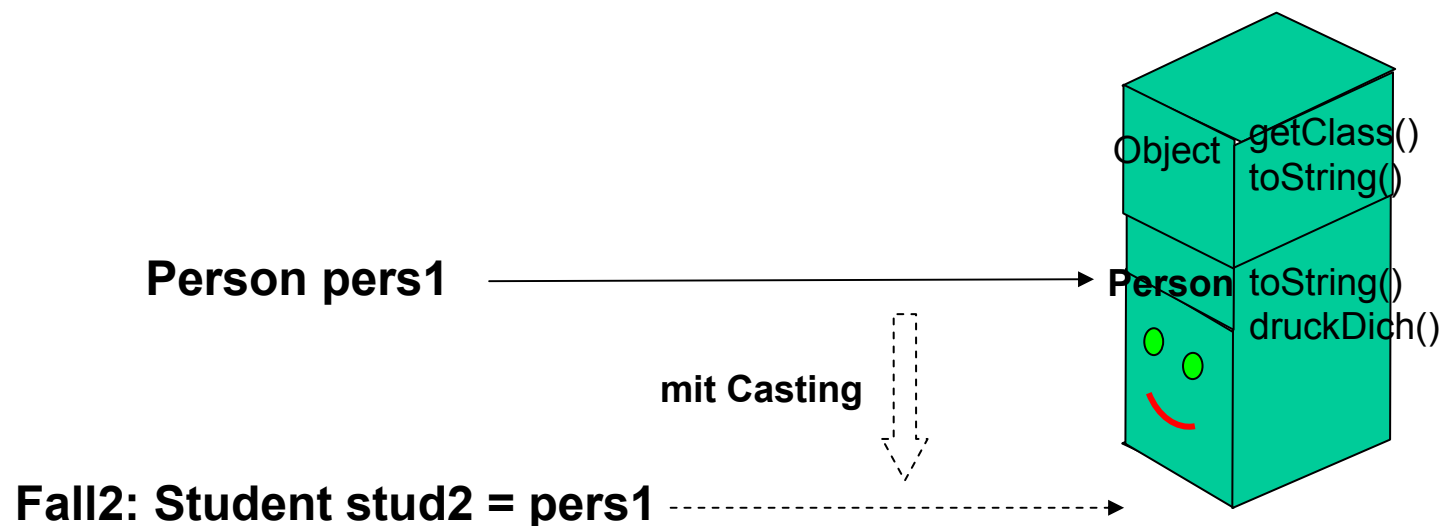
- Aufgabe 12:
  - Versuchen Sie, eine Person, die kein Student ist, nach Student zu casten

- Lösung, Java-Code: 

```
Student stud4 = (Student) pers1;  
stud4.druckDich();
```

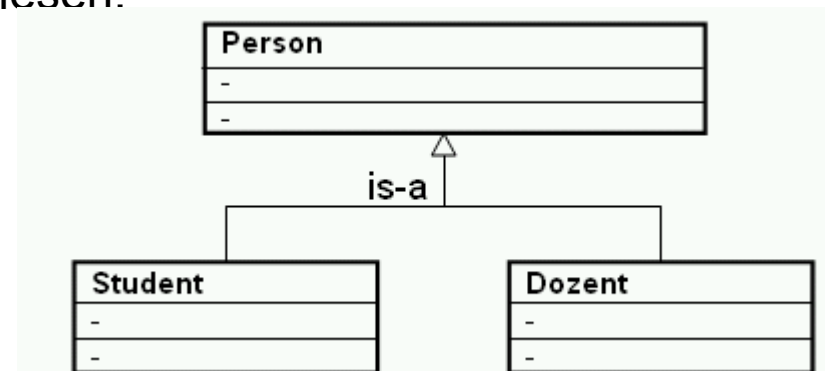
- Konsol-Output:

```
Exception in thread "main" java.lang.ClassCastException: de.fh_lu.o2s.personen.  
Person cannot be cast to de.fh_lu.o2s.personen.Student  
at de.fh_lu.o2s.personen.AppTestStudent.main(AppTestStudent.java:22)
```



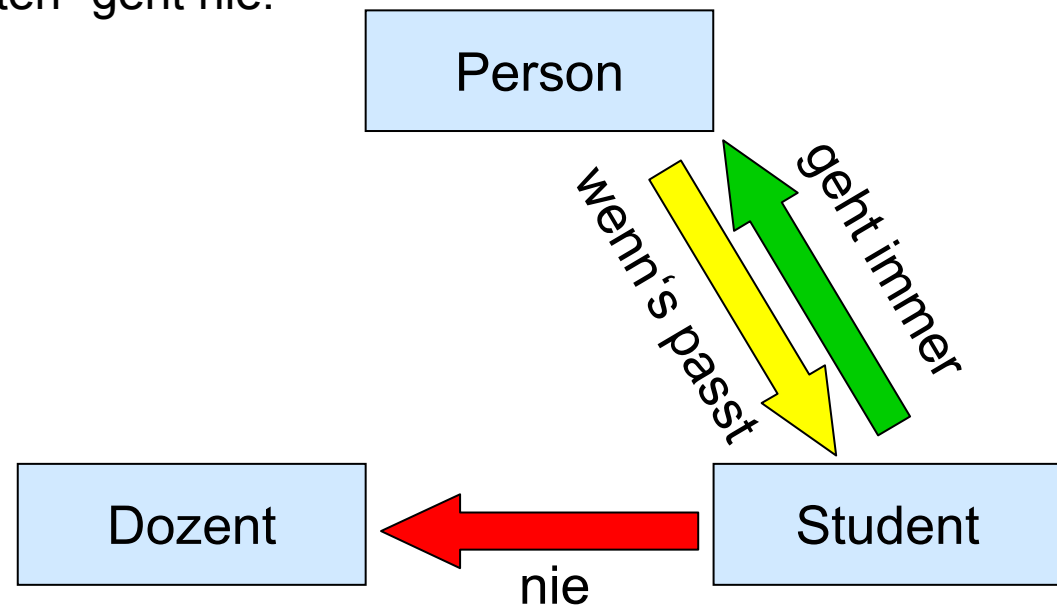
# Casting

- Erklärung:
  - Der Compiler akzeptiert den **Versuch**, die `Person pers1` nach `Student` zu casten,
  - denn die `Person` könnte ja in Wirklichkeit ein `Student`-Objekt sein.
  - Zur Laufzeit erkennt das System, dass `pers1` tatsächlich "nur" ein `Person`-Objekt ist und
  - liefert einen Fehler (`ClassCastException`)
- Anmerkung:
  - Ein Versuch, einen `Dozent` nach `Student` zu casten würde bereits vom Compiler abgewiesen.



# Casting

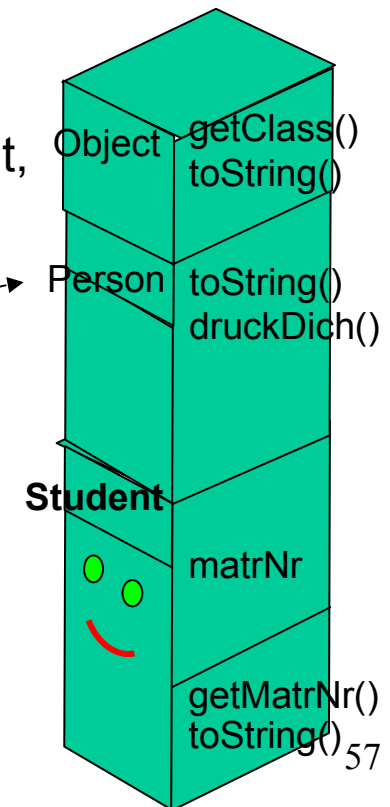
- Folgerung: Casting ist innerhalb einer Vererbungskette möglich:
  - "Aufwärts-Casten" (Objekt „verkleinern“) geht immer
  - "Abwärts-Casten" führt zur Laufzeit zu einer `ClassCastException`, wenn der Objekttyp nicht stimmt.
  - "Seitwärts-Casten" geht nie.



# instanceof

- Aufgabe 13:
  - Führen Sie in den Lösungen zu Aufgabe 11 und 12 vor dem Casting jeweils eine Prüfung durch, ob pers2 bzw. pers1 überhaupt ein Student ist.
- Lösungsansatz:
  - Die Bedingung `(pers2 instanceof Student)` liefert `true`, wenn das Objekt, auf das pers2 zeigt, vom Typ `Student` ist, andernfalls `false`.

**Person pers2**

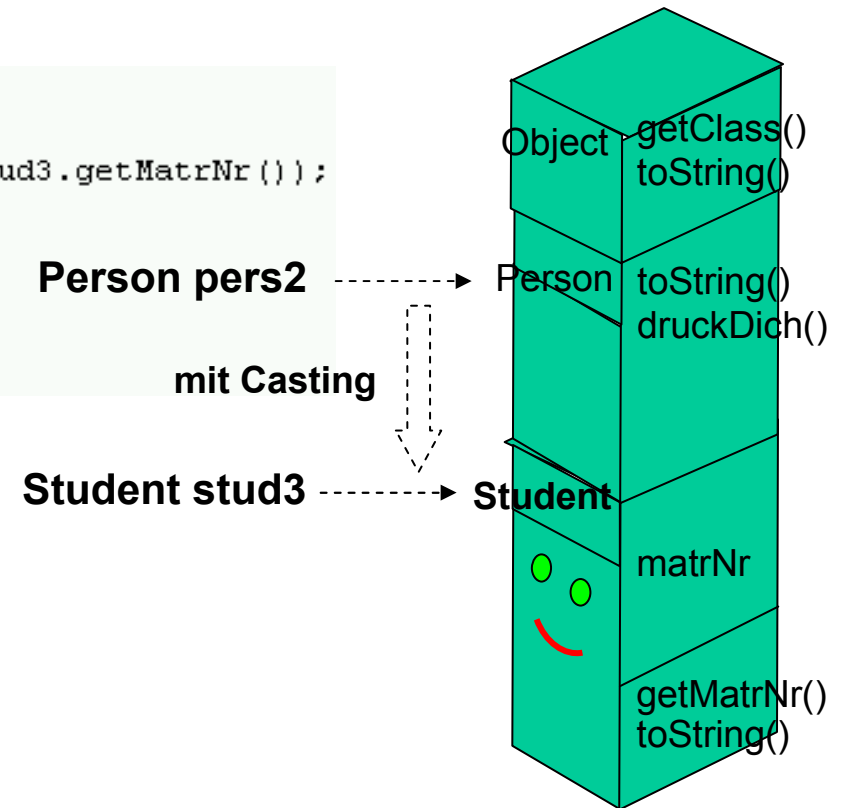
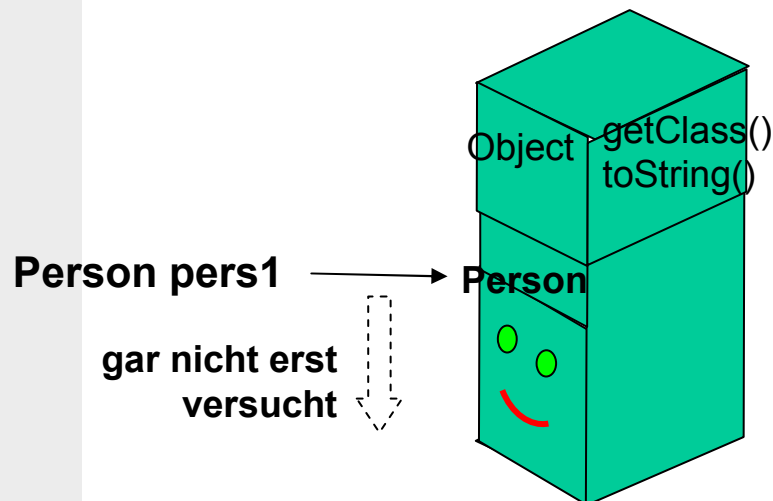




# instanceof

## ■ Lösung:

```
if (pers2 instanceof Student){
    Student stud3 = (Student)pers2;
    System.out.println("Matrikelnummer: " + stud3.getMatrNr());
}
if (pers1 instanceof Student){
    Student stud4 = (Student) pers1;
    stud4.druckDich();
}
```



# Array von Objekten

- Aufgabe 14:
  - Erzeugen Sie in `AppTestStudent` auch noch ein Objekt vom Typ `Dozent` und
  - fassen Sie dieses `Dozent`-Objekt sowie `pers1` und `stud1` in einem Array zusammen.
  - Implementieren Sie dann eine Schleife über das Array, das all diese Objekte auf der Konsole anzeigt.
- Lösungsansatz:
  - Das `Dozent`-Objekt wird erzeugt mit `new (...)`
  - Das Array braucht einen Datentyp.
  - Da `pers1` weder `Student` noch `Dozent` ist, kommt als Datentyp nur `Person` (oder `Object`) in Frage.
  - Wie in Java 1 wird das Array erzeugt und die Objekte eingetragen.
  - Die Schleife durchläuft das Array und schickt jedem Objekt die `toString()`-Methode.

# Array von Objekten

- Lösung, Java-Code:

```
Dozent doz1 = new Dozent("Uwe Klug", "C2");  
Person[] persA = new Person[3];  
persA[0] = pers1;  
persA[1] = stud1;  
persA[2] = doz1;  
for (int index = 0; index < persA.length; index++){  
    System.out.println(persA[index]);  
}
```

- Konsol-Output:

```
Person mit Name: Herbert Hochschulrat, Geburtsjahr 1950  
Student mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321  
Dozent mit Name: Uwe Klug, Geburtsjahr 0
```

- Beobachtung:

- Die Klasse `Dozent` hat (noch) keine eigene `toString()`-Methode, deshalb wird zur Anzeige die `toString()`-Methode von `Person` verwendet.

# Array von Objekten

- Aufgabe 15:
  - Implementieren Sie die Schleife aus Aufgabe 14 als `for each`-Schleife und
  - fügen Sie noch eine Prüfung hinzu, die immer bevor ein `Dozent`-Objekt angezeigt wird, "Achtung der Dozent kommt" in die Konsole schreibt.
- Lösungsansatz:
  - Die `for each`-Schleife geht mit  
`for(Datentyp variable:array){...}`
  - Die Prüfung kann erfolgen
    - mit `instanceof`
    - mit Prüfung auf den Klassennamen

# Array von Objekten

- Lösung, Java-Code:

```
for (Person myPers: persA) {  
    if (myPers instanceof Dozent) {  
        System.out.println("Achtung, der Dozent kommt");  
    }  
    System.out.println(myPers);  
}
```

- Konsol-Output:

```
Person mit Name: Herbert Hochschulrat, Geburtsjahr 1950  
Student mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321  
Achtung, der Dozent kommt  
Dozent mit Name: Uwe Klug, Geburtsjahr 0
```

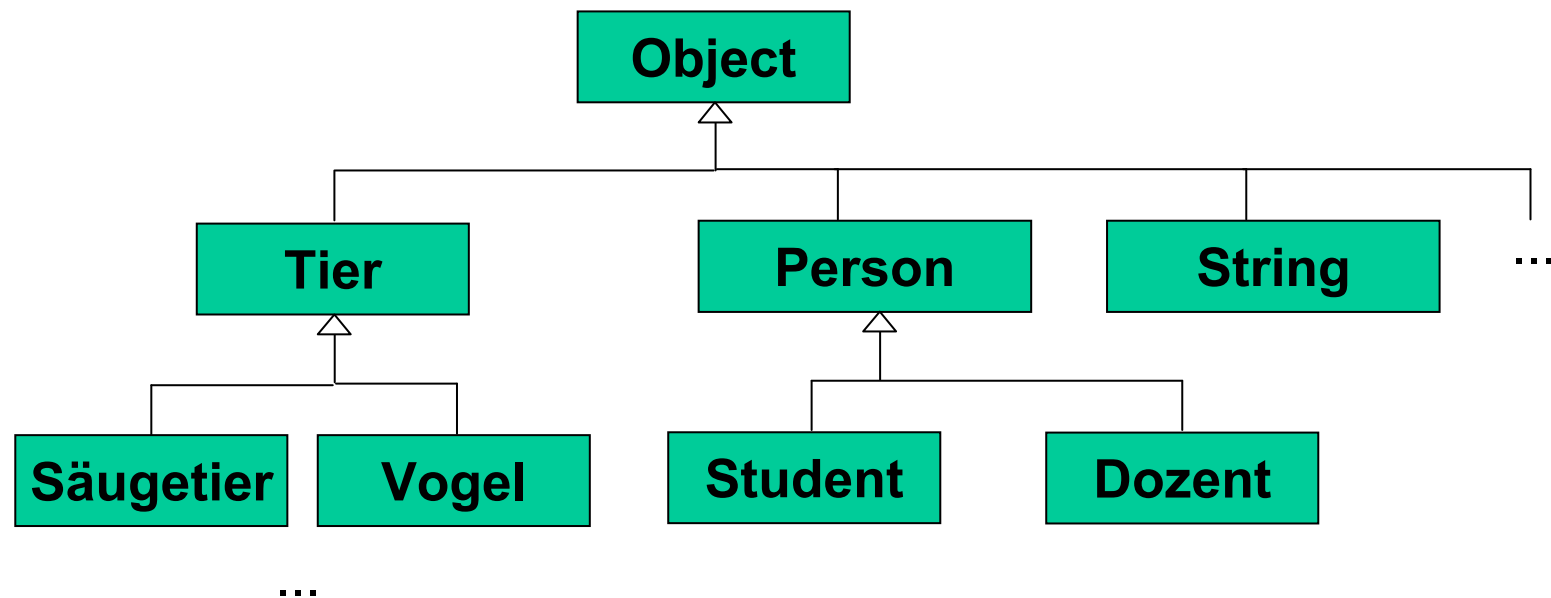
# **Programmierung II**

## **Thema 5: Vererbung, Teil 2**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

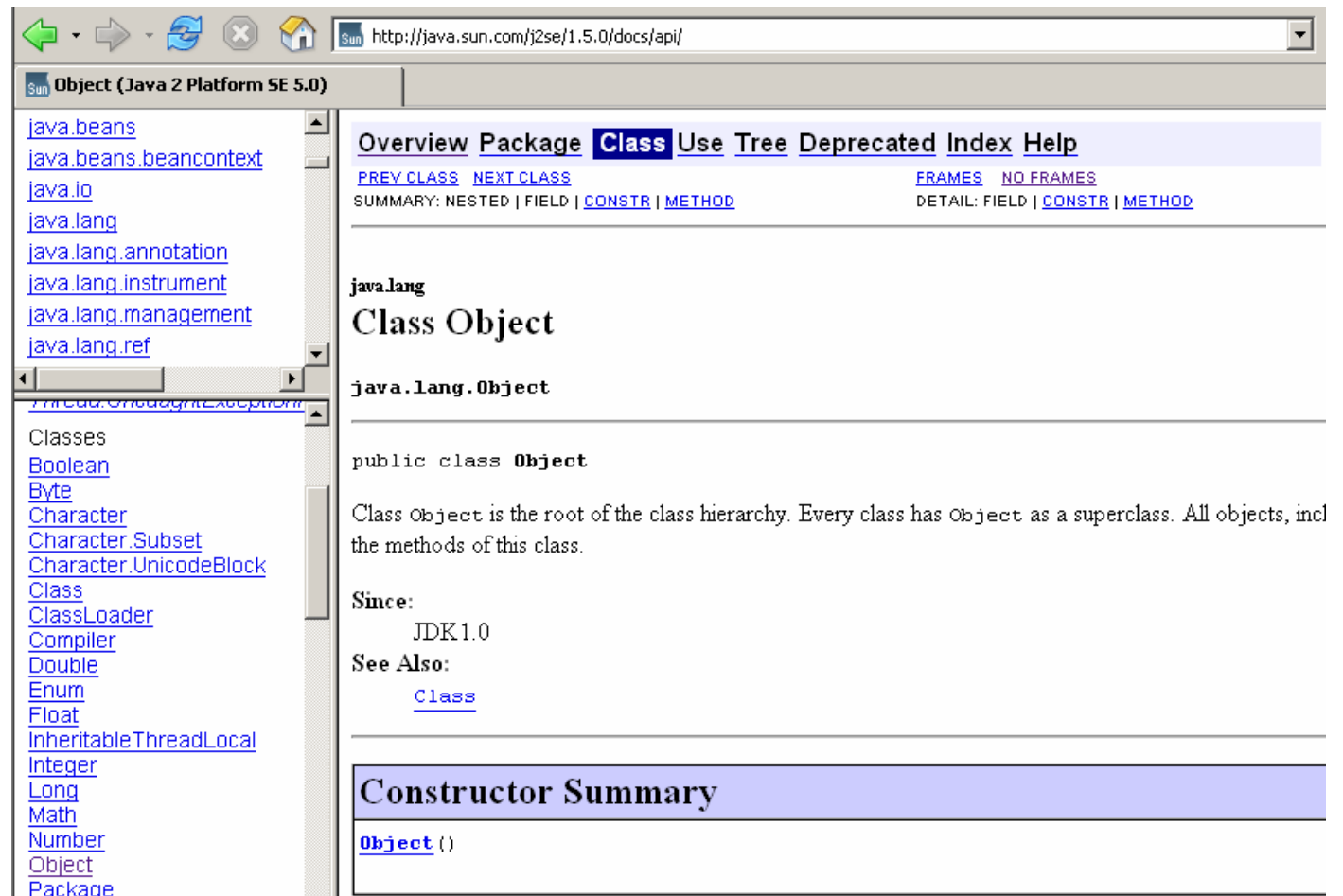
# Klassenhierarchie

- An der Spitze der Klassenhierarchie steht die Klasse `Object`
- Wenn bei der Definition einer neuen Klasse keine Oberklasse angegeben wird, wird die Klasse automatisch direkt unter `Object` angelegt.
- Hierarchie anschauen in Eclipse mit  
Navigate → Open Type Hierarchy (F4)



# Klassenhierarchie

- Folgerung:
  - Alle Klassen erben von `Object`, wo wir aber keine zusätzliche Funktionalität implementieren dürfen.



The screenshot shows the Java API documentation for the `Object` class in the `java.lang` package. The browser address bar shows `http://java.sun.com/j2se/1.5.0/docs/api/`. The left sidebar lists various Java packages and classes, with `Object` selected. The main content area displays the class name `Object` and its description: "Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including instances of the class itself, inherit from Object." It also shows the constructor `Object()` and a section for the constructor summary.

Object (Java 2 Platform SE 5.0)

Overview Package **Class** Use Tree Deprecated Index Help

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#) DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

`java.lang`  
**Class Object**

`java.lang.Object`

`public class Object`

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including instances of the class itself, inherit from Object.

Since:  
JDK 1.0

See Also:  
[Class](#)

**Constructor Summary**

[Object](#) ()



# Methoden von `Object`

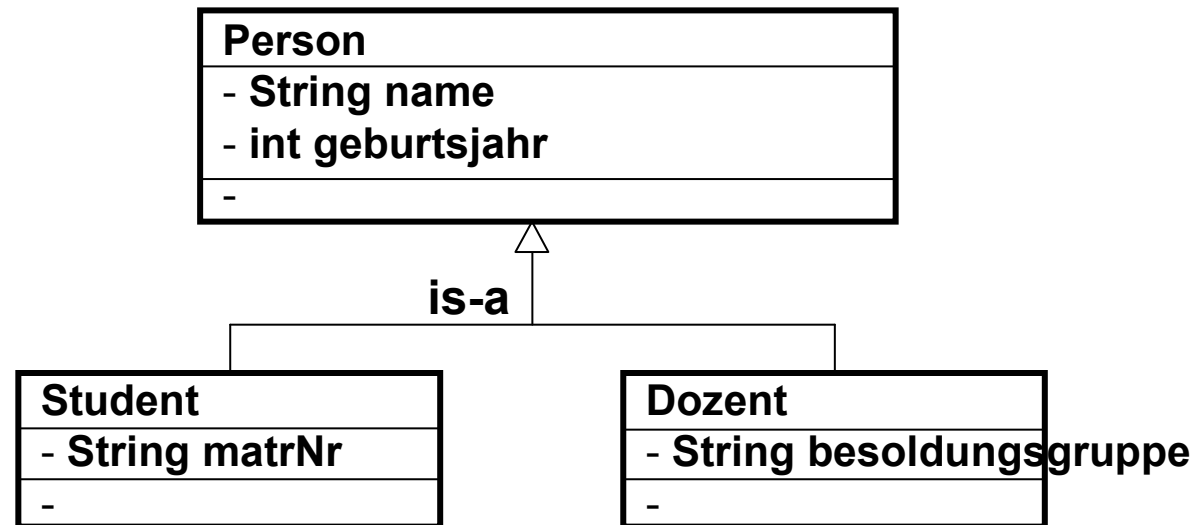
Method Summary			
protected <code>Object</code>	<code>clone()</code>	Creates and returns a copy of this object.	
boolean	<code>equals(Object obj)</code>	Indicates whether some other object is "equal to" this one.	
protected void	<code>finalize()</code>	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.	
<code>Class&lt;? extends Object&gt;</code>	<code>getClass()</code>	Returns the runtime class of an object.	
int	<code>hashCode()</code>	Returns a hash code value for the object.	
void	<code>notify()</code>	Wakes up a single thread that is waiting on this object.	void <code>wait()</code> Causes current thread to wait until another thread calls the <code>notify()</code> method for this object.
void	<code>notifyAll()</code>	Wakes up all threads that are waiting on this object.	void <code>wait(long timeout)</code> Causes current thread to wait until either the specified amount of time has elapsed, or another thread calls the <code>notify()</code> method for this object, or a specified amount of time has elapsed.
<code>String</code>	<code>toString()</code>	Returns a string representation of the object.	void <code>wait(long timeout, int nanos)</code> Causes current thread to wait until another thread calls the <code>notify()</code> method for this object, or some other thread interrupts the current thread, or some other thread calls the <code>wait()</code> method for this object, or some other thread calls the <code>wait()</code> method for this object, or some other thread calls the <code>wait()</code> method for this object.

# toString()

- In `Object` wird z.B. `toString()` implementiert, deshalb versteht jedes Objekt in Java `toString()`.
- In `Object` kann `toString()` aber nicht wissen, wie z.B. ein Kartenspiel als `String` darzustellen ist, deshalb
  - liefert die `toString()`-Methode von `Object` nur die Adresse des Objekts im Hauptspeicher und
  - man muss `toString()` in Subklassen überschreiben
- Diese Konstruktion hat zwei Folgerungen:
  1. Die Nutzung von `toString()` bringt nie einen „method-not-found“-Fehler
  2. Die Methode `toString()` aus `Object` legt für alle Subklassen den Rückgabewert fest.
- Erinnerung: Das Überschreiben von Methoden (mit gleichen Parametern) ist nur möglich mit demselben Rückgabewert.

# Konstrukturen

- Erinnerung:



- Aufgabe 1:

- Vergeben Sie an alle Attribute von `Person` die Sichtbarkeit `private`.
- Implementieren Sie dann in `Student` einen Konstruktor, der die Parameter `name (String)`, `geburtsjahr (int)` und `matrNr (String)` akzeptiert und damit die entsprechenden Attribute füllt.
- Implementieren Sie einen JUnit-Test für die Klasse `Student`, in dem Sie nur den Konstruktor testen.

# Konstrukturen

- Lösung, Schritt 1: Klasse Person
  - Der Vollständigkeit halber Attribute und Konstruktoren

```
public class Person {  
    private String name;  
    private int geburtsjahr;  
    public Person() {  
        super();  
    }  
    public Person(String name) {  
        super();  
        this.name = name;  
    }  
    public Person(String name, int geburtsjahr) {  
        super();  
        this.name = name;  
        this.geburtsjahr = geburtsjahr;  
    }  
}
```

# Konstrukturen

- Lösung, Schritt 2: Erster Versuch des Konstruktors in `Student`


```
public class Student extends Person {  
    private String matrNr;  
    public Student(String name, int geburtsjahr, String matrNr) {  
        this.name = name;  
        this.geburtsjahr;  
        this.matrNr = matrNr;  
    }  
}
```

The field `Person.name` is not visible  
Press 'F2' for focus.

- schlägt fehl, denn
  - `Student` darf nicht auf das Attribut `name` zugreifen, weil dieses in `Person` `private` ist.
- Lösungsansatz:
  - Die Klasse `Person` soll sich selbst um `name` und `geburtsjahr` kümmern
  - ➔ wir rufen den Konstruktor der Klasse `Person` auf.

# Konstruktoren

- Lösung, Schritt 3: Zweiter Versuch des Konstruktors in `Student`

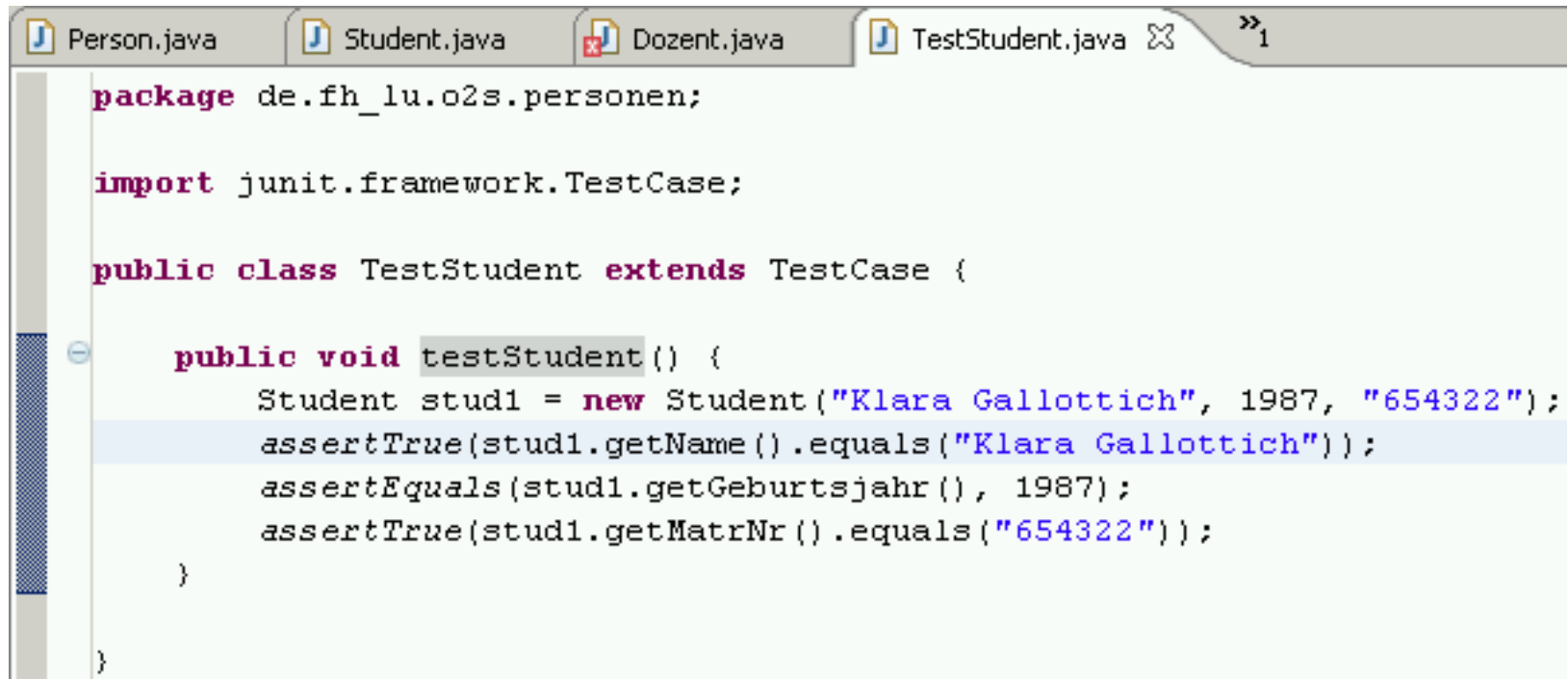


```
public class Student extends Person {  
    private String matrNr;  
    public Student(String name, int geburtsjahr, String matrNr) {  
        super(name, geburtsjahr);  
        this.matrNr = matrNr;  
    }  
}
```

- Anmerkungen:
  - Der Konstruktor der Oberklasse wird aufgerufen mit `super (...)`
  - Da es nur eine Oberklasse gibt, muss der Klassenname nicht angegeben werden.
  - `Person` hat drei Konstruktoren aber anhand der Parametertypen wird der richtige herausgefunden, in diesem Fall `Person(String name, int geburtsjahr)`.
  - Weil dieser Konstruktor `public` ist, darf er aufgerufen werden und schreibt die Parameter in die Attribute, obwohl die Attribute `private` sind.

# Konstrukturen

- Lösung, Schritt 4: JUnit-Test anlegen
  - Neuer JUnit Test Case mit Name `TestStudent`, der zu testenden Klasse `Student` und dem Konstruktor als zu testende Methode,
  - In der Testmethode Erzeugung eines geeigneten Student-Objekts und Assertions mit Abfrage der Attribute.



```
Person.java Student.java Dozent.java TestStudent.java »1
package de.fh_lu.o2s.personen;

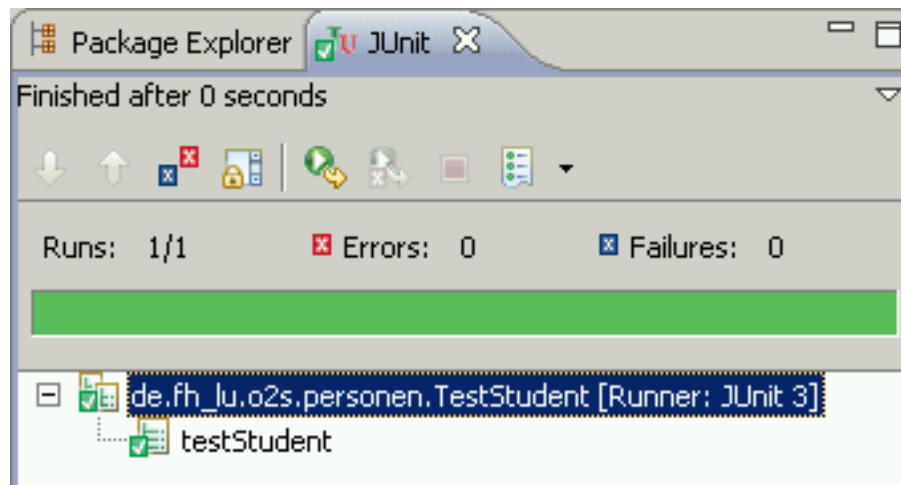
import junit.framework.TestCase;

public class TestStudent extends TestCase {

    public void testStudent() {
        Student stud1 = new Student("Klara Gallottich", 1987, "654322");
        assertTrue(stud1.getName().equals("Klara Gallottich"));
        assertEquals(stud1.getGeburtsjahr(), 1987);
        assertTrue(stud1.getMatrNr().equals("654322"));
    }
}
```

# Konstrukturen

- Lösung, Schritt 5: JUnit-Test ausführen
  - Rechter Mausklick auf `TestStudent` → Run As → JUnit Test



- Ergebnis:
  - Der Test ist erfolgreich,
  - Die Attribute müssen korrekt eingetragen worden sein.



# Konstrukturen

- Anmerkungen:
  - Jeder Konstruktor ruft als allererstes einen anderen Konstruktor auf.
  - Meistens ist das ein Konstruktor der Oberklasse, manchmal aber auch ein anderer Konstruktor derselben Klasse
  - Ausnahme: `Object()`
  - Wenn ein solcher Aufruf nicht ausdrücklich hingeschrieben wird, wird automatisch `super()` ausgeführt. Die folgenden Codes sind also gleichwertig:

```
public Person(String name, int geburtsjahr) {  
    super();  
    this.name = name;  
    this.geburtsjahr = geburtsjahr;  
}
```

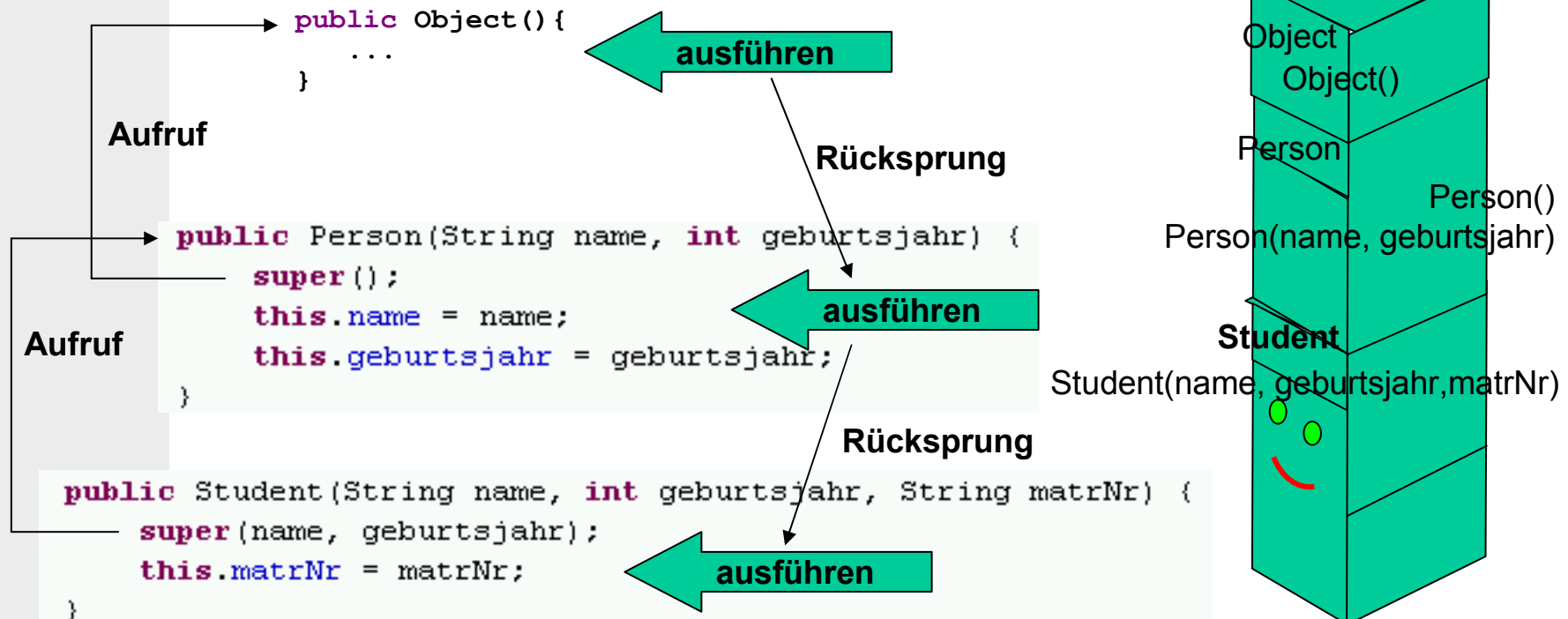
```
public Person(String name, int geburtsjahr) {  
    this.name = name;  
    this.geburtsjahr = geburtsjahr;  
}
```

# Konstruktoren

- Begründung: Jede Klasse weiß selbst am Besten,
  - wie ihre Attribute zu füllen sind
  - ob sie noch andere Vorbereitungen treffen muss, z.B. eine Datenbankverbindung öffnen, etc.
- Dies gilt
  - auch gegenüber Unterklassen,
  - speziell wenn diese von jemand anderem geschrieben wurden.
  - Z.B. ist `Person` eine Unterklasse von `Object` und wir haben keine Ahnung, was `Object()` tun muss, um ein Objekt im Hauptspeicher anlegen zu können.
- Anmerkung:
  - Der Aufruf des Konstruktors der Oberklasse muss immer am Anfang des Konstruktors stehen.

# Konstrukturen

- Folgerung:
  - Wenn ein Objekt im Hauptspeicher neu angelegt wird, z.B. mit `new Student(...)`, dann wird
  - als erstes immer der Code von `Object()` ausgeführt.
  - Anschließend der Konstruktoren-Code in der Klassenhierarchie abwärts.



# Konstrukturen

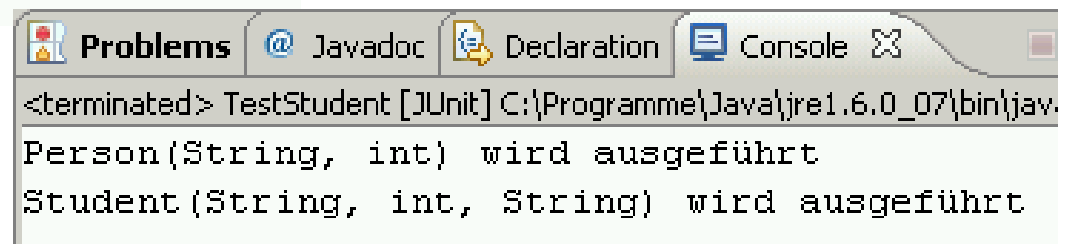
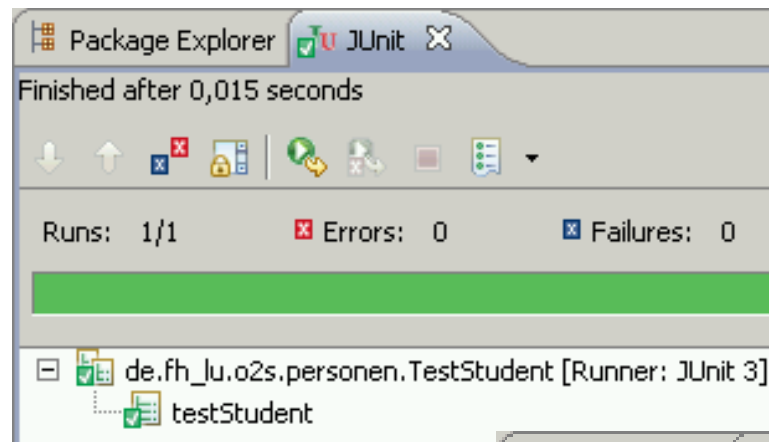
- Aufgabe 2:
  - Implementieren Sie in den Konstruktoren von `Person` und `Student` Konsolausgaben und weisen Sie damit die Ausführungsreihenfolge nach.
- Lösung, Schritt 1: Konstruktoren

```
public Person(String name, int geburtsjahr) {  
    super();  
    System.out.println("Person(String, int) wird ausgeführt");  
    this.name = name;  
    this.geburtsjahr = geburtsjahr;  
}
```

```
public Student(String name, int geburtsjahr, String matrNr) {  
    super(name, geburtsjahr);  
    System.out.println("Student(String, int, String) wird ausgeführt");  
    this.matrNr = matrNr;  
}
```

# Konstrukturen

- Lösung, Schritt 2: Ausführung
  - Wiederholung des JUnit Test incl. `new Student(...)` und Prüfung der Konsolaausgabe



- Anmerkung:
  - Leider können wir in `Object()` keine Konsolaausgabe hinzufügen.

# Exkurs: `this (...)`

- Exkurs:
  - Anstatt eines Konstruktors der Oberklasse kann auch ein Konstruktor der eigenen Klasse ausgeführt werden.
  - Die Aufrufsyntax dafür lautet `this (...)` anstatt `super (...)`.
- Anmerkung:
  - Dadurch dass dann der aufgerufene Konstruktor einen Konstruktor der Oberklasse aufruft, endet die Aufrufkette am Ende wieder bei `Object ()`.
  - Wenn sich ein Konstruktor selbst aufruft (oder Konstruktoren einer Klasse sich gegenseitig (rekursiv) aufrufen), streikt der Compiler.

```
public Student(String name, int geburtsjahr) {  
    this(name, geburtsjahr);  
    System.out.println("Konstruktor wurde ausgeführt");  
}
```

Recursive constructor invocation Student(String, int)  
Press 'F2' for focus.

# Exkurs: `this (...)`

- Beispiel:

```
public Student(String name, int geburtsjahr) {  
    super(name, geburtsjahr);  
    System.out.println("Student(String, int) wird ausgeführt");  
}  
public Student(String name, int geburtsjahr, String matrNr) {  
    this(name, geburtsjahr);  
    System.out.println("Student(String, int, String) wird ausgeführt");  
    this.matrNr = matrNr;  
}
```



- Konsol-Output:

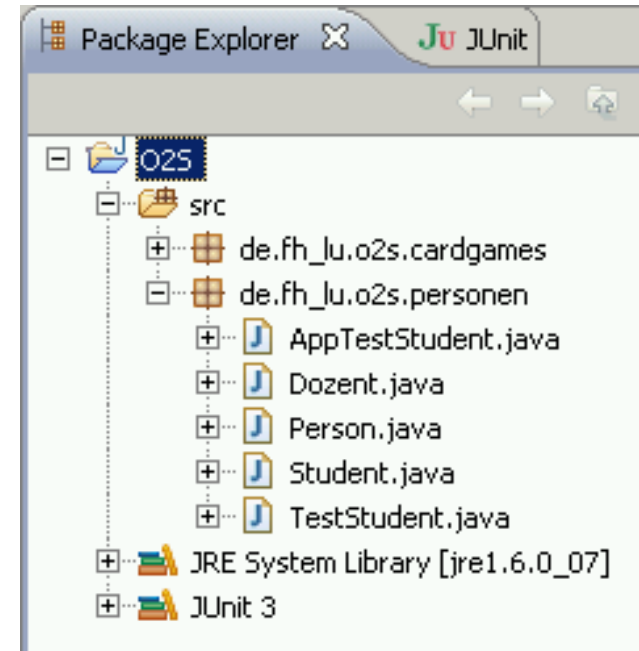
```
Person(String, int) wird ausgeführt  
Student(String, int) wird ausgeführt  
Student(String, int, String) wird ausgeführt
```

# Packages

- Ein Package ist
  - eine Sammlung von Klassen.
  - Jede Klasse ist einem Package zugeordnet.
  - Default ist das (default package)
  - Im Java-Code der Klasse steht das Package in der ersten Zeile:

```
package de.fh_lu.o2s.personen;
```

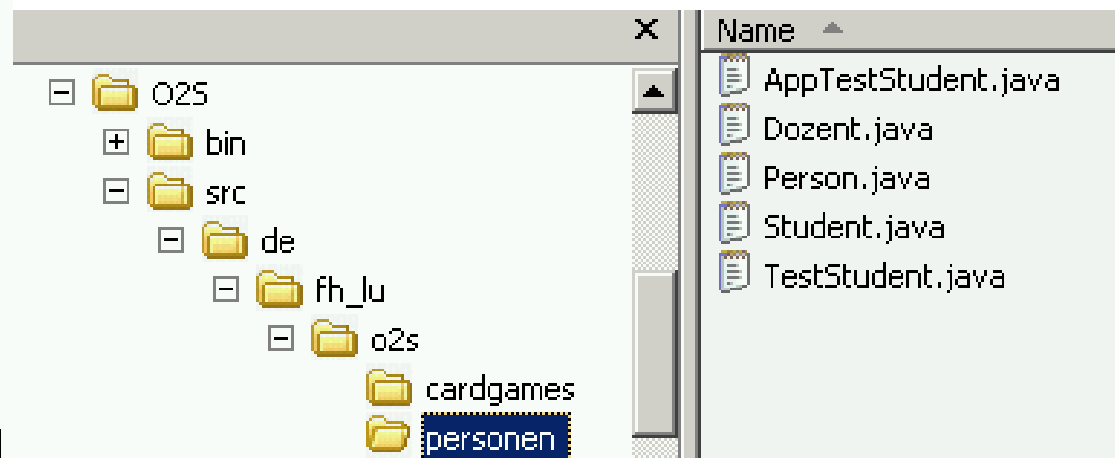
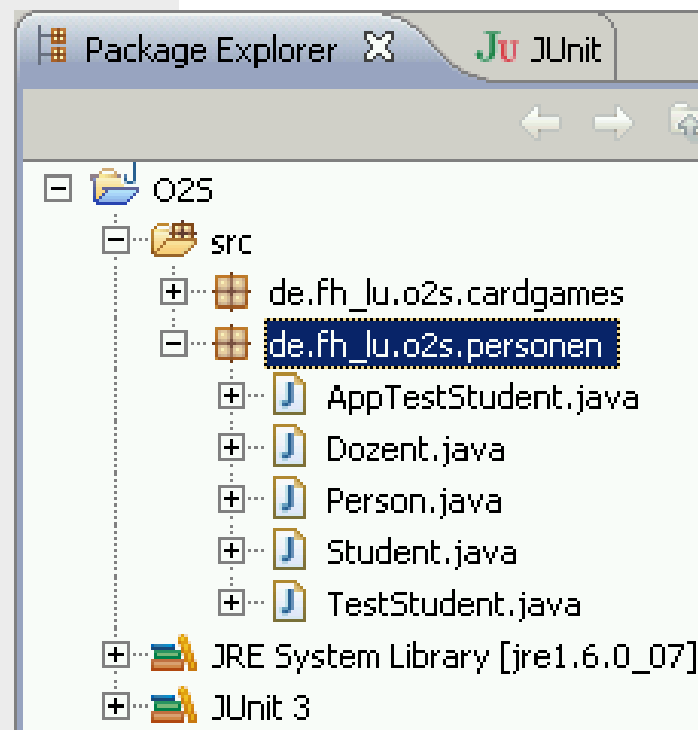
```
public class Student extends Person{  
    private String matrNr;  
    public Student () {  
        ...  
    }  
}
```





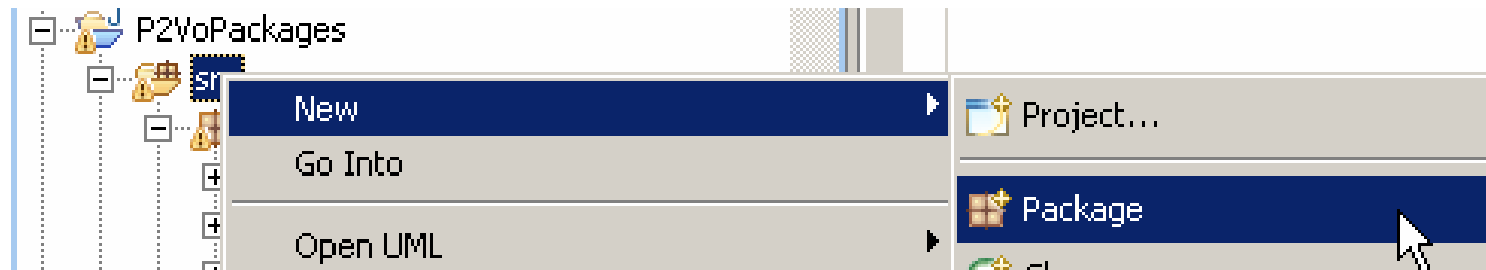
# Dateiverwaltung

- Eclipse legt pro Package einen Ordner mit allen Klassen an
- Wenn der Packagename einen Punkt enthält,
  - wird eine Ordnerstruktur erzeugt:  
`de.fh_lu.o2s.personen` → `de\fh_lu\o2s\personen`

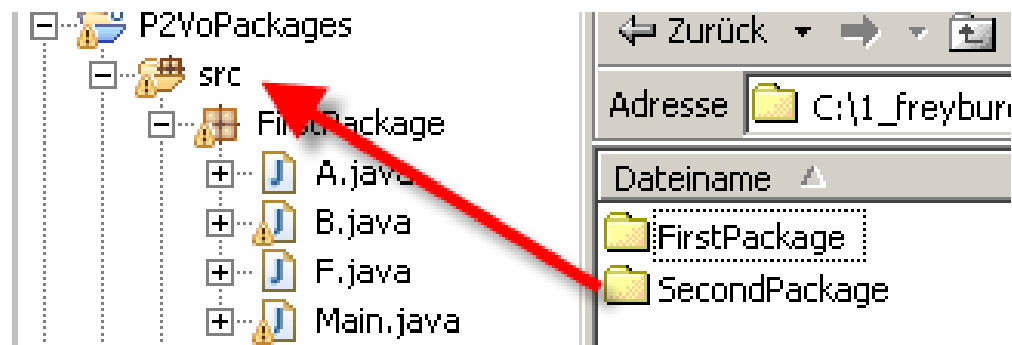


# Packages

- Neues Package anlegen



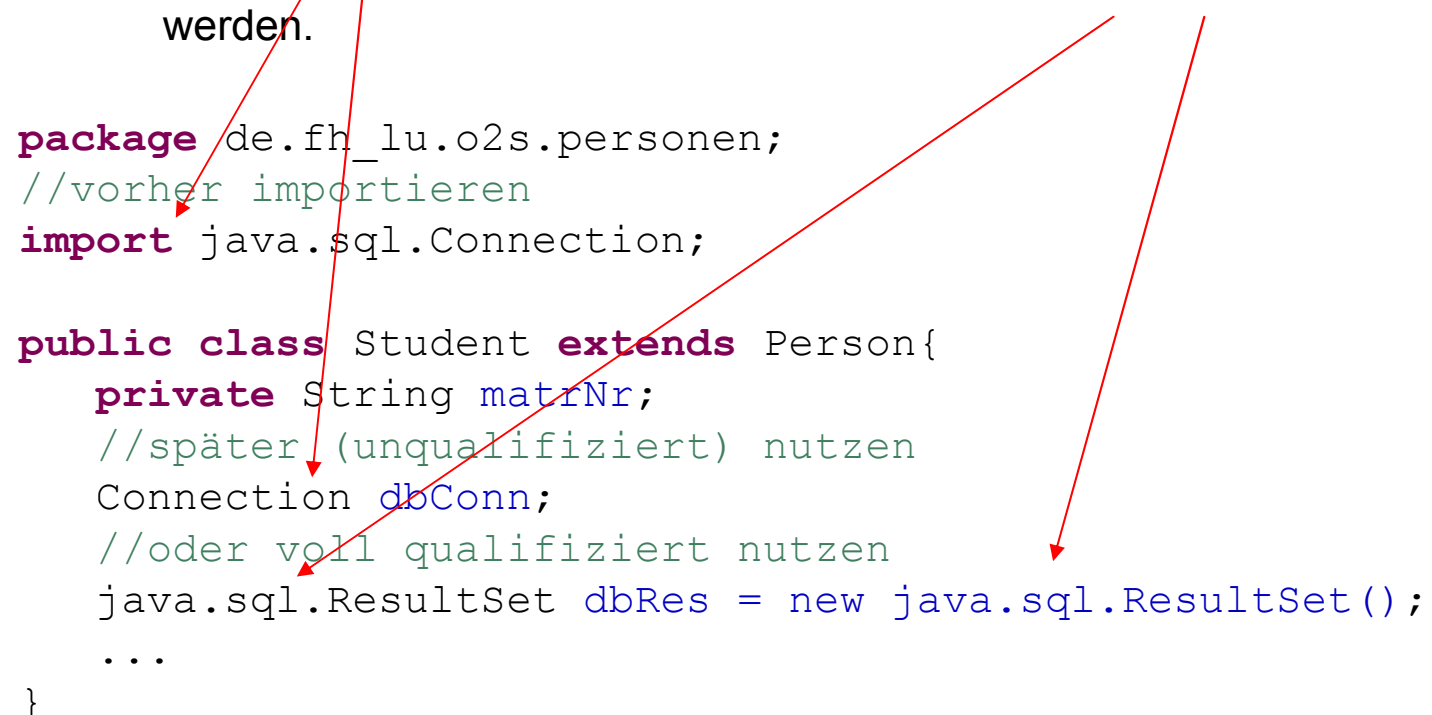
- ...oder einfach Ordner drag & drop, (anschließend Refresh Workspace)



# Packages

- Wenn in einer Klasse
  - Objekte genutzt werden sollen,
  - deren Typen (Klassen) in einem anderen Package definiert sind,
  - dann müssen diese Typen (Klassen)
    - vorher importiert (und dann einfach genutzt) oder
    - „voll qualifiziert“ werden, d.h. immer wenn der Klassenname verwendet wird, muss das gesamte Package dazugeschrieben werden.

```
package de.fh_lu.o2s.personen;  
//vorher importieren  
import java.sql.Connection;  
  
public class Student extends Person{  
    private String matrNr;  
    //später (unqualifiziert) nutzen  
    Connection dbConn;  
    //oder voll qualifiziert nutzen  
    java.sql.ResultSet dbRes = new java.sql.ResultSet();  
    ...  
}
```



# Packages

- Nur `java.lang`, also die Sprache (language) selbst steht immer automatisch zur Verfügung und muss nicht importiert werden.
- Wenn alle Klassen eines Packages importiert werden sollen:  
`import java.sql.*;`
- Alle Klassen mehrerer Packages können nicht auf einmal importiert werden:  
`import java.*; //geht nicht`

# Zugriffsrechte/Sichtbarkeit

- Standardmäßig kann auf ein Attribut oder eine Methode von allen Methoden in allen Klassen desselben Packages aus zugegriffen werden (sonst nicht „sichtbar“ / „visible“):
  - `String name;`
  - `String getName(){return this.name;}`
- Mit *modifiern* kann dies geändert werden.
- Bsp.: Zugriff für alle Methoden aller Klassen mit `public`:
  - `public String name;`
  - `public void getName(){return this.name;}`
  - Wenn Objekte aus anderen Packages angesprochen werden sollen, müssen die entsprechenden Klassen vorher importiert werden.
- Bsp.: Zugriff nur durch Methoden derselben Klasse mit `private`:
  - `private String name;`
  - `private void getName(){return this.name;}`

# Zugriffsrechte/Sichtbarkeit

- Letzter modifier: protected
- Auf Attribute mit protected kann von allen Methoden aus Klassen desselben Packages und aus Subklassen zugegriffen werden:
  - **protected** String name;
  - **protected** void getName(){return this.name;}

**public – zugreifbar aus allen Klassen**

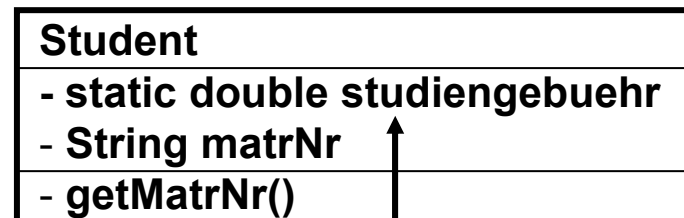
**protected – zugreifbar aus allen Klassen desselben Package und aus allen Subklassen**

**package (kein Modifier) – zugreifbar aus allen Klassen desselben Package**

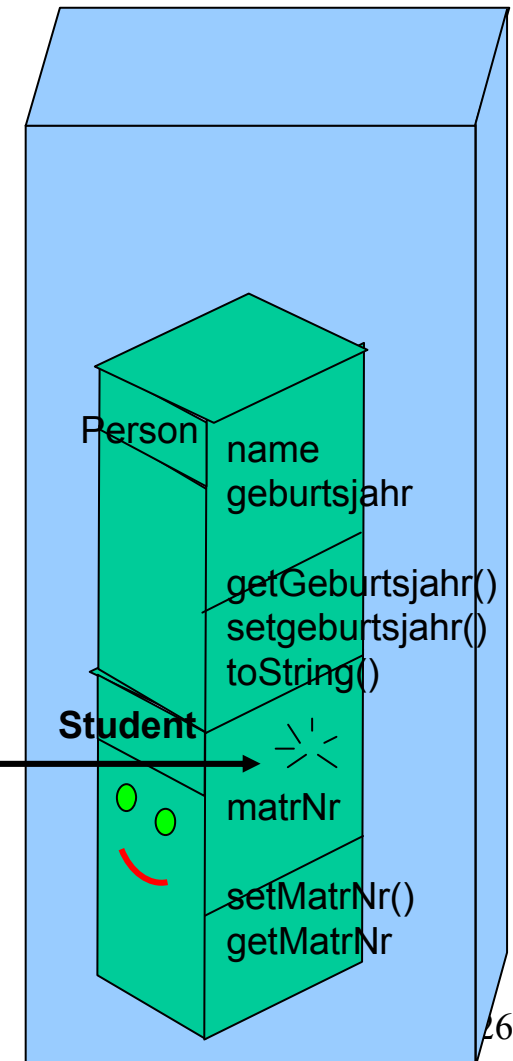
**private – zugreifbar nur aus derselben Klasse**

# Schlüsselwort static

- Attribute oder Methoden einer Klasse können als `static` gekennzeichnet werden.
- Bsp.: Attribut "Studiengebuehr"
  - Attribut von Student,
  - aber für alle Studenten gleich



- Keine Eigenschaft eines Objekts (im Hauptspeicher), sondern der ganzen Klasse (im Klassenkatalog)



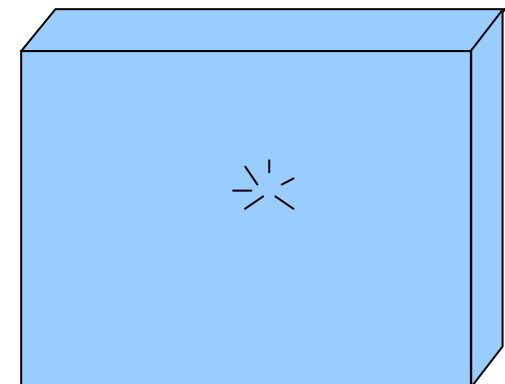
# Schlüsselwort static

- Bsp.: Methode main()

```
public static void main(String[] args) {  
    Person pers1 = new Person("Herbert Hochschulrat", 1950);  
    Student stud1 = new Student("Karola Fleißig", "654321");  
    stud1.setGeburtsjahr(1986);  
}
```

- Keine Methode eines Objekts (im Hauptspeicher), sondern der ganzen Klasse (im Klassenkatalog)
- Folgerung:
  - `main()` existiert bereits dann, wenn im Hauptspeicher noch kein Objekt der Klasse (mit `new ...`) erzeugt wurde.

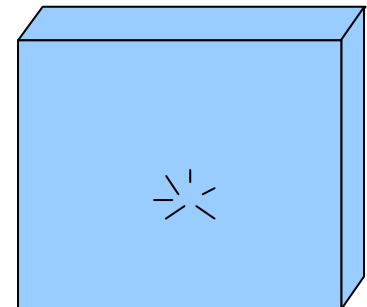
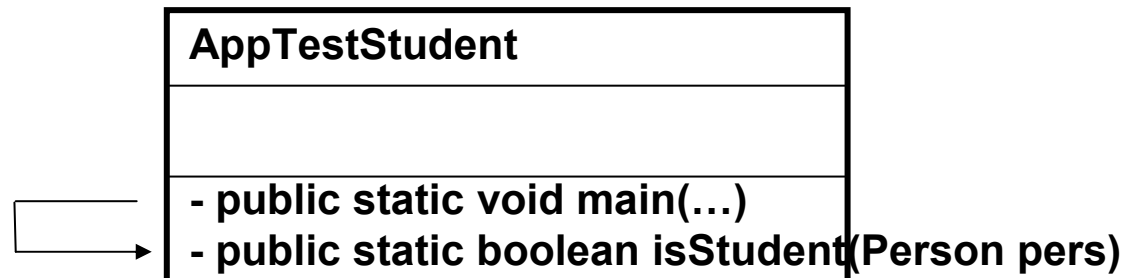
AppTestStudent
- public static void main(...)





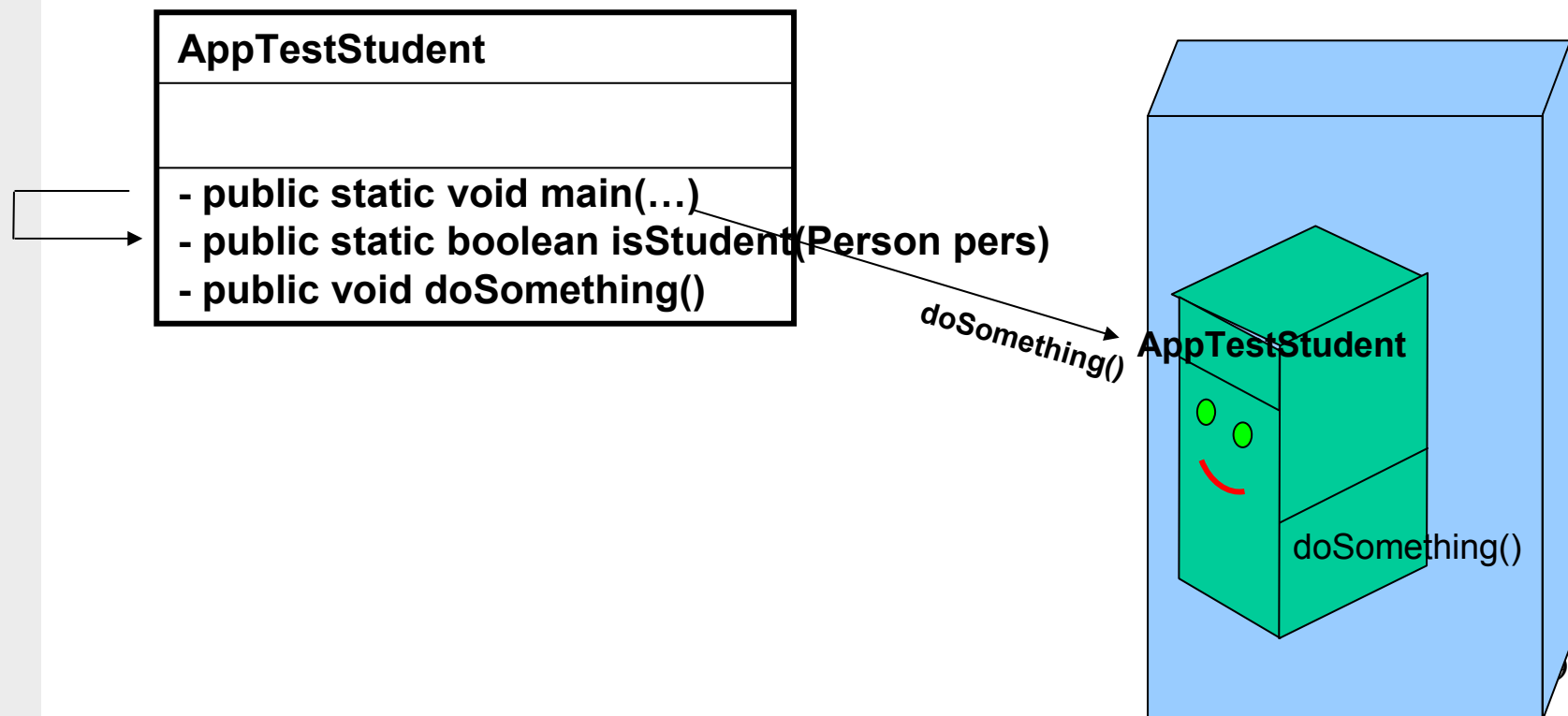
# Zusammenspiel

- Aufgabe 3: Implementieren Sie in Ihrer Applikationsklasse `AppTestStudent`
  - eine statische Methode `isStudent(...)`, die für jede `Person` prüfen kann, ob es sich dabei um einen `Student` handelt und
  - eine nicht-statische Methode `doSomething()`, die nur die Konsolausgabe "doSomething() wurde gestartet" erzeugt.
  - Rufen Sie beide Methoden aus Ihrer `main()`-Methode heraus auf.
- Lösungsansatz, Teil 1:
  - `isStudent()` kann direkt von `main()` aufgerufen werden, weil beide Methoden `static` sind, also beide im Klassenkatalog stehen.
  - Ein Objekt im Hauptspeicher wird nicht benötigt.



# Zusammenspiel

- Lösungsansatz, Teil 2:
  - als nicht-statische Methode "lebt" `doSomething()` nur im Inneren eines Objekts.
  - Da sie in der Klasse `AppTestStudent` definiert wurde, muss also ein Objekt vom Typ `AppTestStudent` erzeugt werden.



# Zusammenspiel

- Lösung, Schritt 1: `isStudent (...)`

- In `AppTestStudent`:

```
public static boolean isStudent(Person pers){  
    return (pers instanceof Student);  
}
```

- Lösung, Schritt 2: `doSomething ()`

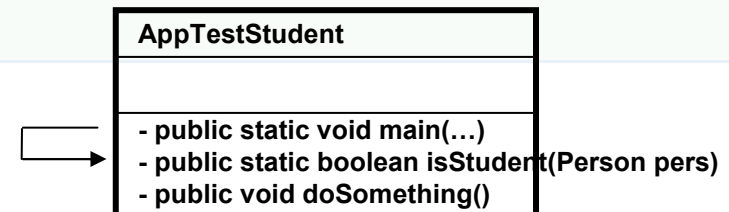
- In `AppTestStudent`:

```
public void doSomething(){  
    System.out.println("doSomething() wurde gestartet");  
}
```

# Zusammenspiel

- Lösung, Schritt 3: Aufruf von `isStudent(...)`

```
public static void main(String[] args) {  
    //...  
    Student stud1 = new Student("Karola Fleißig", "654321");  
    Person pers2 = stud1;  
    if (AppTestStudent.isStudent(pers2)){  
        Student stud3 = (Student)pers2;  
        System.out.println("Matrikelnummer: " + stud3.getMatrNr());  
    }  
    //...
```



- Anmerkung:
  - Jede Botschaft (jeder Methodenaufruf) braucht einen Empfänger. In diesem Fall ist das die Klasse `AppTestStudent`.
  - Da `isStudent()` in derselben Klasse definiert ist, könnte auch einfach geschrieben werden:

```
if (isStudent(pers2)) {  
    //...
```

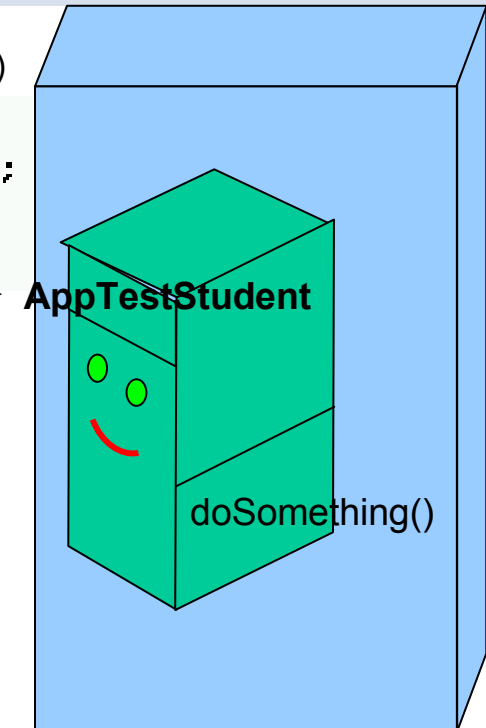
# Zusammenspiel

- Lösung, Schritt 4: Aufruf von `doSomething()`

```

public static void main(String[] args) {
    AppTestStudent ats = new AppTestStudent();
    ats.doSomething();
    //...
  
```

**AppTestStudent ats** →



- Anmerkung:

- Der Empfänger der Botschaft `doSomething()` muss ein Objekt vom Typ `AppTestStudent` sein.
- Da noch kein solches existiert müssen wir eines erzeugen mit `new ...`
- Wir könnten auch kürzer schreiben:

```

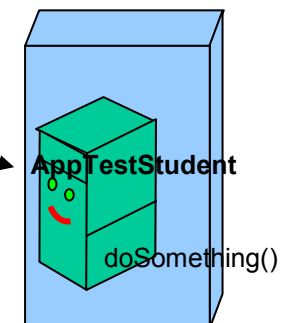
public static void main(String[] args) {
    (new AppTestStudent()).doSomething();
    //...
  
```

- oder sogar:

```

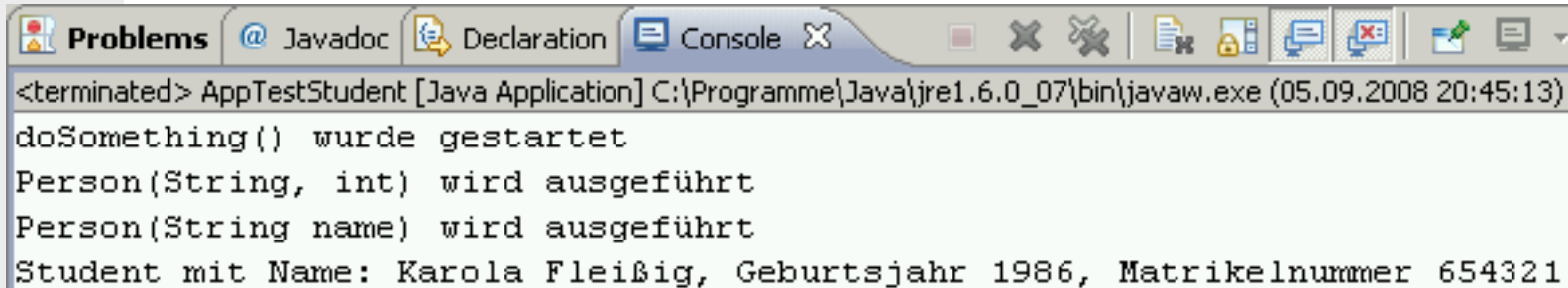
public static void main(String[] args) {
    new AppTestStudent().doSomething();
    //...
  
```

**<no name>** →

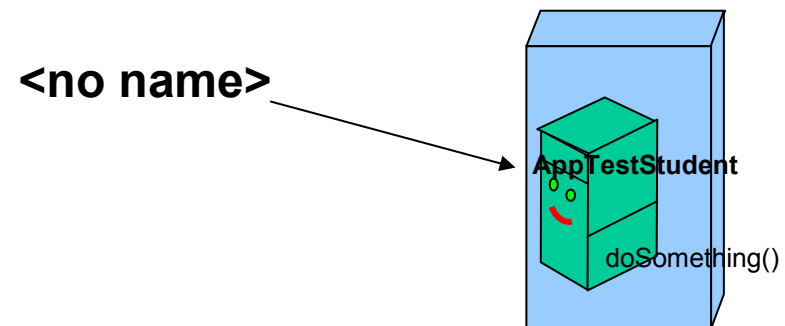


# Zusammenspiel

- Lösung, Schritt 5: Ausführung von `AppTestStudent`
  - Bis auf die Ausgabe von "doSomething() wurde gestartet" keine Änderung der Konsolenausgabe



```
<terminated> AppTestStudent [Java Application] C:\Programme\Java\jre1.6.0_07\bin\javaw.exe (05.09.2008 20:45:13)
doSomething() wurde gestartet
Person(String, int) wird ausgeführt
Person(String name) wird ausgeführt
Student mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321
```



# Schlüsselwort `final`

- Aufgabe 4: Implementieren Sie in Ihrer Klasse `Person` eine Methode `toPersonString()`,
  - die nur Name und Geburtsjahr anzeigt
  - und auch in Unterklassen genau diesen Effekt hat und nicht überschrieben werden kann.
- Lösungsansatz:
  - Selber Code wie in der `toString()`-Methode der Klasse `Person`.
  - Das Schlüsselwort `final` verhindert das Überschreiben in Unterklassen.

# Schlüsselwort `final`

- Lösung, Schritt 1: Implementierung
  - Erinnerung:

```
public String toString(){// In Person
    return this.getClass().getSimpleName() +
        " mit Name: " + this.getName() +
        ", Geburtsjahr " + this.getGeburtsjahr();
}
```

- neu:

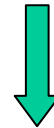
```
public final String toPersonString(){
    return this.getClass().getSimpleName() +
        " mit Name: " + this.getName() +
        ", Geburtsjahr " + this.getGeburtsjahr();
}
```



# Schlüsselwort `final`

- Lösung, Schritt 2: Test:
  - Ausgabemethoden in `AppTestStudent` einfügen
  - und zwar sowohl für ein `Person`-Objekt als auch für ein `Student`-Objekt:

```
public static void main(String[] args) {  
    Person pers1 = new Person("Herbert Hochschulrat", 1950);  
    Student stud1 = new Student("Karola Fleißig", "654321");  
    stud1.setGeburtsjahr(1986);  
    System.out.println(pers1.toPersonString());  
    System.out.println(stud1.toPersonString());  
}
```



- Ergebnis (Konsole):

```
Person mit Name: Herbert Hochschulrat, Geburtsjahr 1950  
Student mit Name: Karola Fleißig, Geburtsjahr 1986
```

- In Student:

```
public String toPersonString(){  
    Cannot override the final method from Person  
    Press 'F2' for focus.  
}
```

- Beobachtung:
  - In der Klasse `Person` kommt der Code der `toString()`-Methode jetzt doppelt vor.
- Aufgabe 5:
  - Passen Sie die Methoden `toString()` und `toPersonString()` so an, dass der Code nicht doppelt vorkommt.
- Lösungsansatz:
  - Eine der beiden Methoden soll die andere aufrufen anstatt den Code selbst auszuführen.

# Schlüsselwort `final`

- Lösung, 1. Versuch:

- `toPersonString()` ruft `toString()` auf:

```
public String toString(){// In Person
    return this.getClass().getSimpleName() +
           " mit Name: " + this.getName() +
           ", Geburtsjahr " + this.getGeburtsjahr();
}
public final String toPersonString(){
    return this.toString();
}
```

- Test:

```
System.out.println(pers1.toPersonString());
System.out.println(stud1.toPersonString());
```



- Ergebnis:

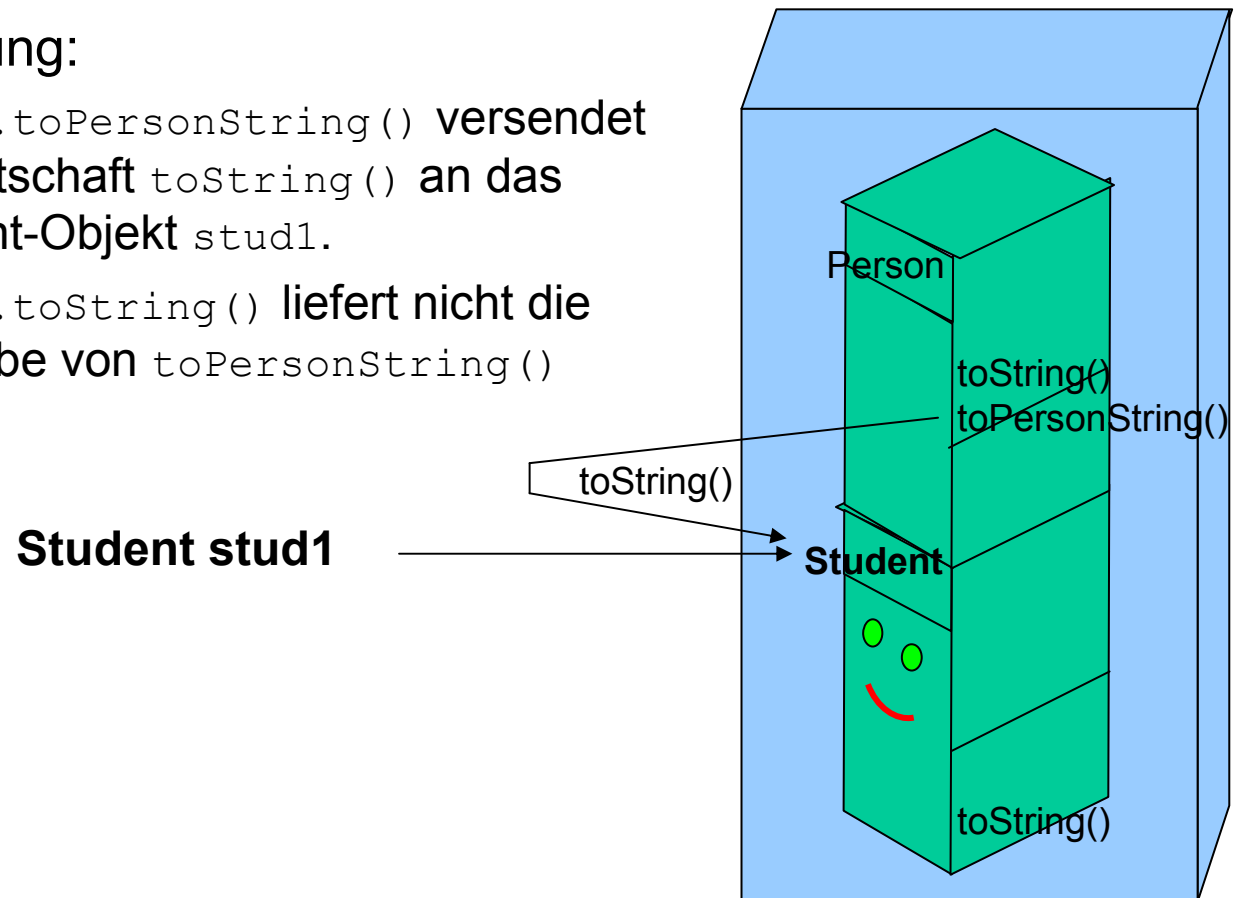
```
Person mit Name: Herbert Hochschulrat, Geburtsjahr 1950
Student mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321
```

- Beobachtung:

- Bei dem `Student`-Objekt wird zuviel angezeigt.

# Schlüsselwort `final`

- **Beobachtung:**
  - Bei `stud1.toPersonString()` wird zuviel angezeigt.
- **Begründung:**
  - `stud1.toPersonString()` versendet die Botschaft `toString()` an das Student-Objekt `stud1`.
  - `stud1.toString()` liefert nicht die Ausgabe von `toPersonString()`



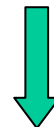
# Schlüsselwort `final`

- Lösung, 2. Versuch:
  - In `Person` soll `toString()` die `toPersonString()` aufrufen:

```
public String toString(){  
    return this.toPersonString();  
}  
public final String toPersonString(){// In Person  
    return this.getClass().getSimpleName() +  
        " mit Name: " + this.getName() +  
        ", Geburtsjahr " + this.getGeburtsjahr();  
}
```

- Test:

```
System.out.println(pers1.toPersonString());  
System.out.println(stud1.toPersonString());
```



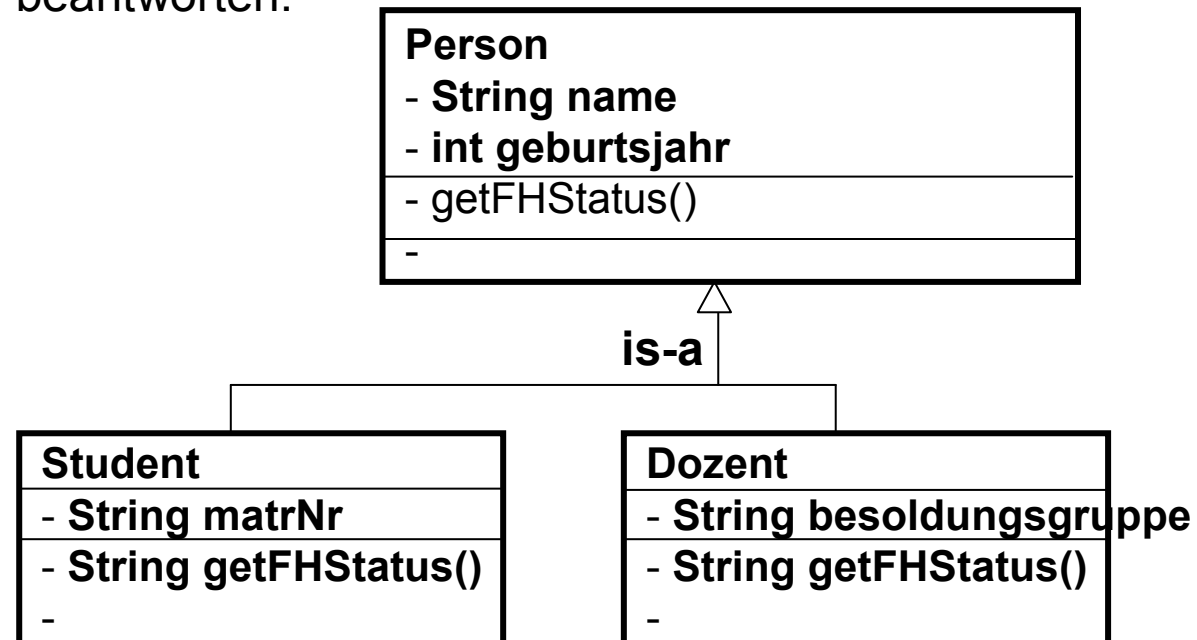
- Ergebnis:

```
Person mit Name: Herbert Hochschulrat, Geburtsjahr 1950  
Student mit Name: Karola Fleißig, Geburtsjahr 1986
```

- Beobachtung:
  - Es klappt.

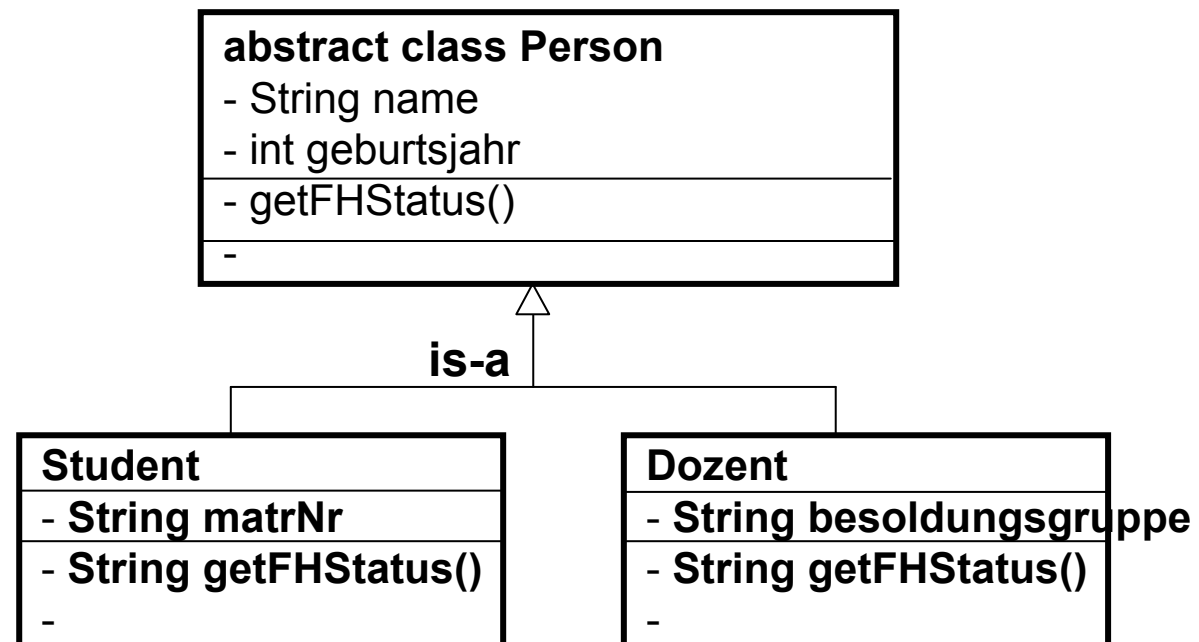
# FH-Status

- In einer Anwendung zur Verwaltung von Hochschulangehörigen könnte folgendes sinnvoll sein:
- Aufgabe 6:
  - a) Personen, die nicht entweder Student oder Dozent sind, sollen gar nicht vorkommen
  - b) Jede Person, die tatsächlich vorkommt, soll eine Methode `getFHStatus()` akzeptieren und mit einem entsprechenden String beantworten.



# FH-Status

- Es gibt verschiedene Lösungsmöglichkeiten für dieses Szenario, wir verwenden folgenden
- Lösungsansatz:
  - `Person` wird in eine abstrakte Klasse umgewandelt,
  - `Person` erhält eine abstrakte Methode `getFHStatus()`,
  - `Student` und `Dozent` erhalten konkrete Methoden `getFHStatus()`.



- Lösung, Schritt 1: `Person` wird als abstrakte Klasse definiert:

```
public abstract class Person {
```

- Zwischenstand:

- Objekte vom Typ `Person` können nun nicht mehr instanziiert bzw. (mit `new ...`) erzeugt werden:

```
public class AppTestStudent {  
    public static void main(String[] args) {  
        Person pers1 = new Person("Herbert Hochschulrat", 1950);  
        Student stud1 = new Student("Karola Fleißig", "654321");  
    }  
}
```

Cannot instantiate the type Person  
Press 'F2' for focus.



- Erklärung: Abstrakte Klassen
  - bleiben in der Klassenhierarchie,
  - vererben Attribute und Methoden,
  - können aber nicht selbst instanziiert werden.
- Ihre Unterklassen
  - können ganz normal instanziiert werden.
  - Dabei werden auch die Konstruktoren der abstrakten Klasse durchlaufen:

```
public class AppTestStudent {  
    public static void main(String[] args) {  
        //      Person  pers1 = new Person("Herbert Hochschulrat", 1950);  
        Student stud1 = new Student("Karola Fleißig", "654321");  
    }  
}
```

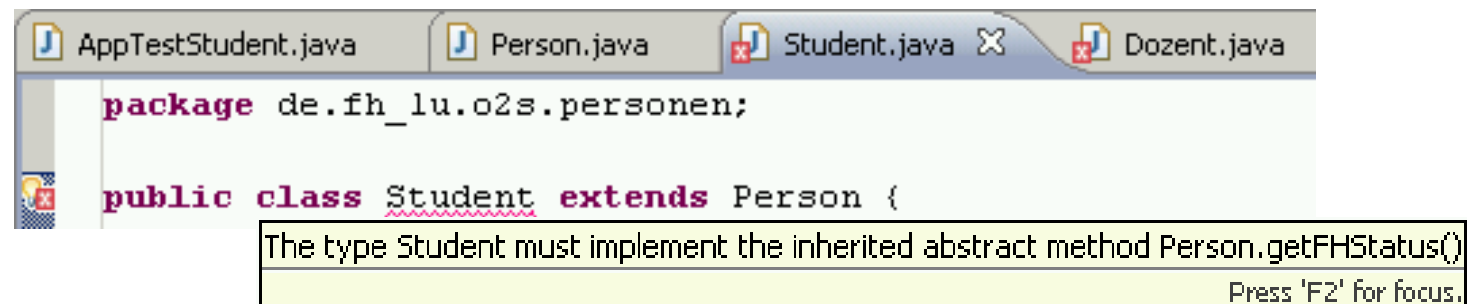


```
Person(String name) wird ausgeführt  
Student(String, String) wird ausgeführt
```

- Lösung, Schritt 2:
  - In `Person` wird die abstrakte Methode `getFHStatus()` definiert.

```
public abstract class Person {  
    ...  
    public abstract String getFHStatus();  
}
```

- Dann muss diese Methode auch in allen Unterklassen definiert werden, andernfalls gibt es einen Fehler:



- Lösung, Schritt 3:
  - Die Methode `getFHStatus()` wird in `Student` und `Dozent` definiert:



```
public String getFHStatus(){  
    return "Student";  
}
```

```
public String getFHStatus(){  
    return "Dozent";  
}
```

- Ausprobieren in `AppTestStudent`:

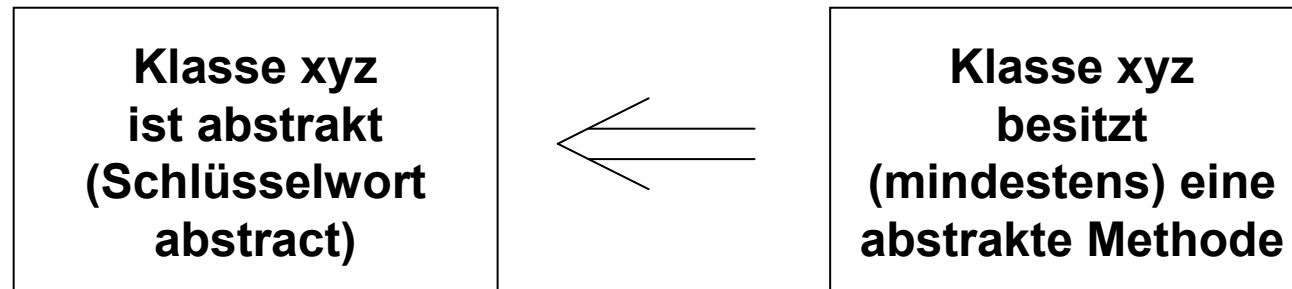
```
public class AppTestStudent {  
    public static void main(String[] args) {  
        //      Person pers1 = new Person("Herbert Hochschulrat", 1950);  
        Student stud1 = new Student("Karola Fleißig", "654321");  
        System.out.println("stud1 ist " + stud1.getFHStatus());  
    }  
}
```



```
Person(String name) wird ausgeführt  
Student(String, String) wird ausgeführt  
stud1 ist Student
```

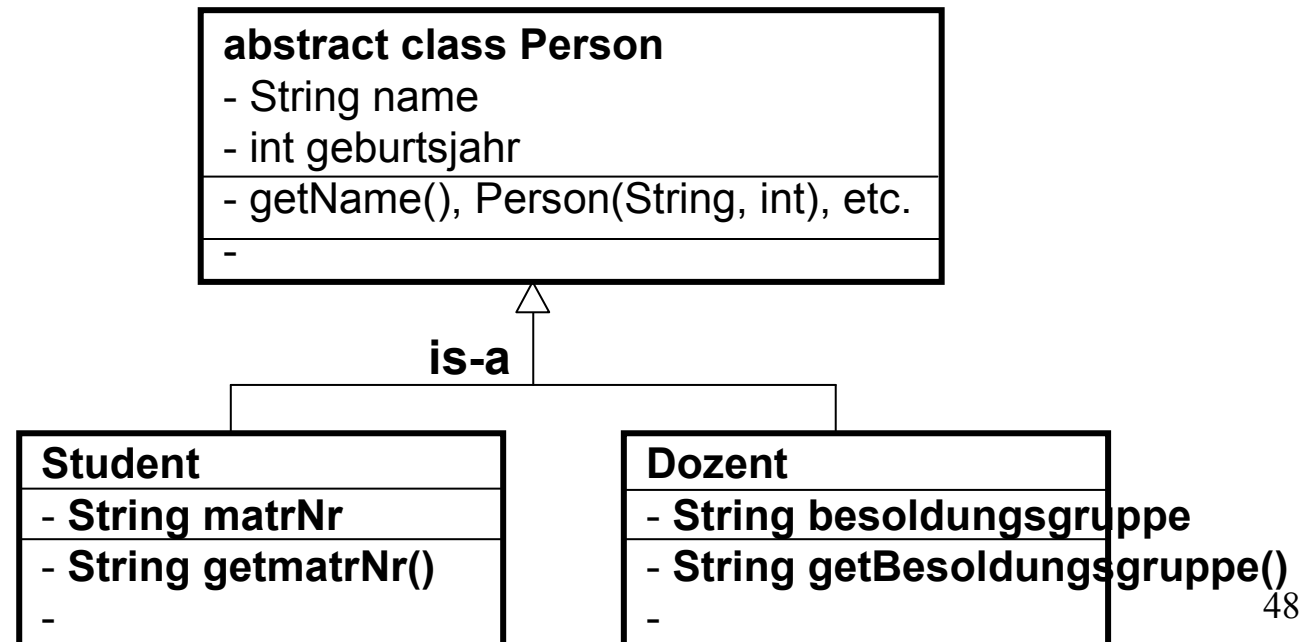
# Abstrakte Methoden

- Anmerkungen:
  - Eine Klasse kann abstrakt sein, auch ohne eine abstrakte Methode zu haben.
  - Wenn Sie aber eine abstrakte Methode hat, dann muss sie auch eine abstrakte Klasse sein.



# Abstrakte Methoden

- Anmerkungen:
  - Wenn zwei (konkrete, nicht-abstrakte) Klassen gemeinsame Eigenschaften oder Fähigkeiten haben,
  - dann kann durch eine abstrakte Klasse
    - diese Gemeinsamkeit betont werden und
    - Verdopplung von gemeinsamem Code vermieden werden, z.B. name, geburtsjahr, Konstruktoren, get-/set-Methoden,



# Abstrakte Methoden

- Anmerkung:
  - Wenn zwei (konkrete, nicht-abstrakte) Klassen eine gemeinsame Oberklasse besitzen,
  - dann können Ihre Objekte in einem Array dieses Oberklassentyps zusammengefasst werden,
  - auch wenn die gemeinsame Oberklasse abstrakt ist.
  - Es funktioniert z.B. unverändert:

```
Student stud1 = new Student("Karola Fleißig", "654321");
Dozent doz1 = new Dozent("Uwe Klug", "C2");
Person[] persA = new Person[3];
persA[0] = pers1;
persA[1] = stud1;
persA[2] = doz1;
for (Person myPers: persA){
    if (myPers instanceof Dozent){
        System.out.println("Achtung, der Dozent kommt");
    }
    System.out.println(myPers);
}
```



```
null
Student mit Name: Karola Fleißig, Geburtsjahr 1986, Matrikelnummer 654321
Achtung, der Dozent kommt
Dozent mit Name: Uwe Klug, Geburtsjahr 0
```

- Programmierung nach Vertrag:
  - Wenn ein System oder Framework Anforderungen an einen Programmierer stellt und dafür einen Nutzen verspricht,
- Bsp. 1: Ich darf das Java-Framework nutzen
  - und z.B. von `Object` erben
  - aber nur so wie das Framework es vorschreibt, z.B. muss `toString()` immer einen String zurückgeben.
- Bsp. 2: Ich darf eine Unterklasse von der abstrakten Klasse `Person` bilden und
  - alle Funktionen von `Person` erben und
  - meine Objekte in ein `Person`-Array eintragen,
  - aber nur, wenn ich die Methode `public String getFHStatus()` implementiere.
- Abstrakte Klassen erlauben damit die Implementierung von Verträgen.

- Aufgabe 7:
  - Welche Alternative zur Implementierung der Methode `getFHStatus()` würde Ihnen einfallen?
  - Was wären die Vor- oder Nachteile?
- Lösungsansatz:
  - Die Methode könnte in `Person` konkret (nicht abstrakt) implementiert werden.
  - Soweit die Subklassen bisher bekannt sind, können diese mit `instanceof` ermittelt werden.



- Lösung: Implementierung in Person:

```
public String getFHStatus(){  
    if (this instanceof Student) return "Student";  
    else if (this instanceof Dozent) return "Dozent";  
    else return "FH-Status unbekannt";  
}
```

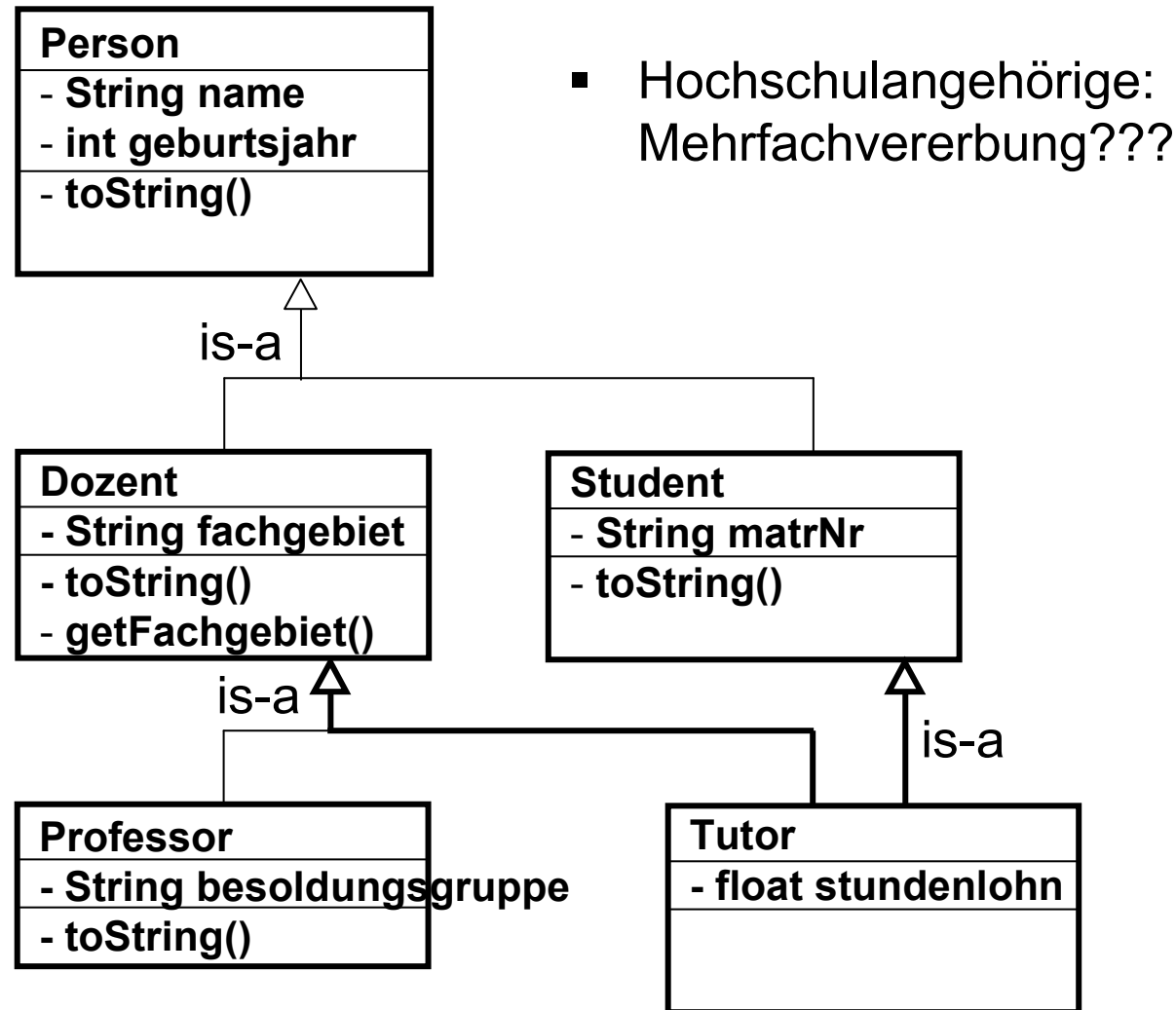
- Vorteil:
  - Implementierung ist nur an einer Stelle notwendig.
- Nachteil:
  - Zukünftige Unterklassen sind unbekannt und werden nicht gezwungen, selbst etwas zu tun.
  - Unterklassen wissen selbst eigentlich viel besser, wie sie ihren Status ausdrücken sollten. Dies kann zwar durch Überschreiben geregelt werden, dann geht aber der oben genannte Vorteil verloren.

# **Programmierung II**

## **Thema 6: Java-Interfaces**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

# Beispiel

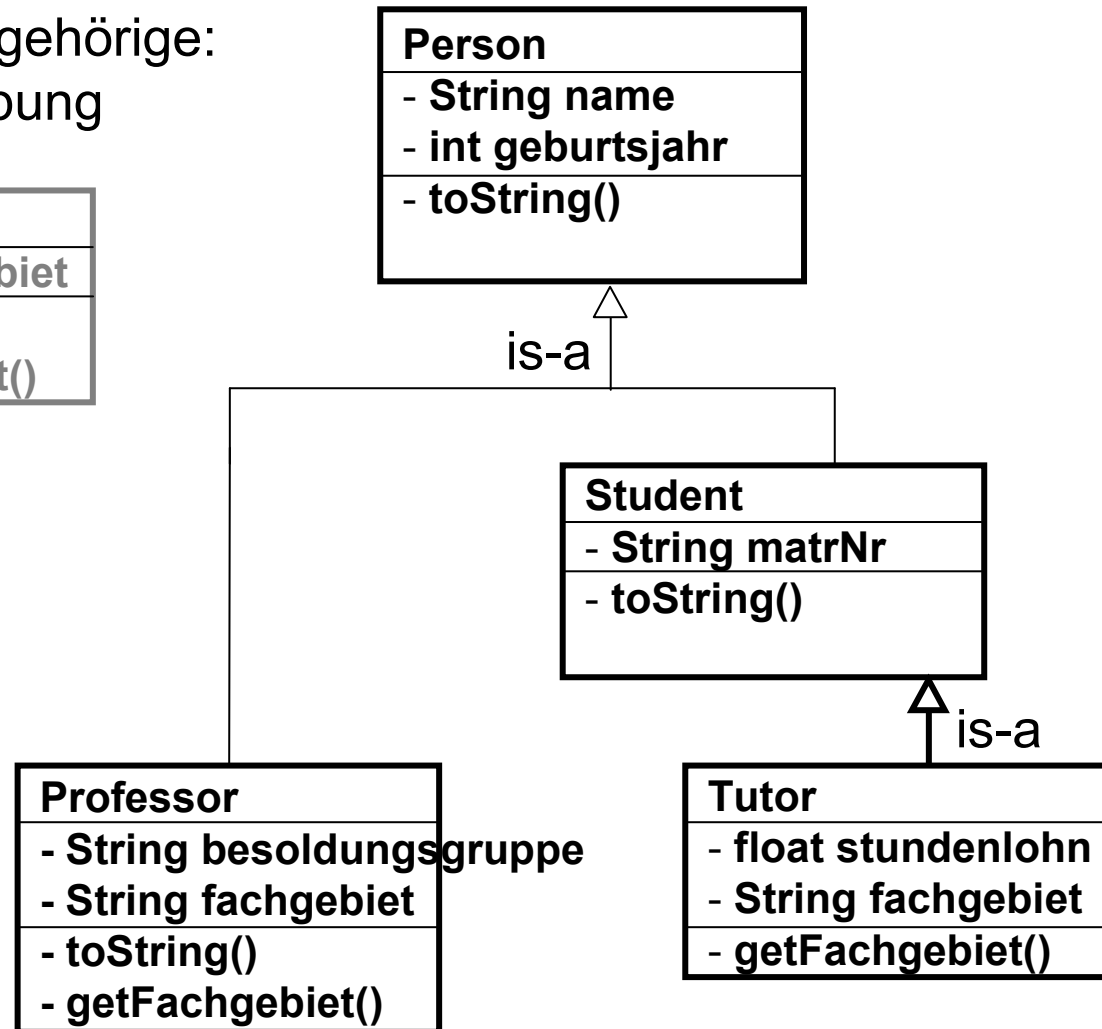
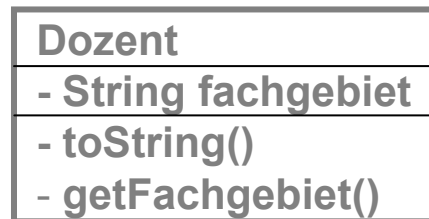


# Mehrfachvererbung

- Mehrfachvererbung wie im obigen Beispiel kann zu Konflikten führen, wenn
  - zwei Oberklassen dieselbe Methode implementiert haben aber mit unterschiedlichem Code.
  - zwei Oberklassen dieselbe Konstante implementiert haben aber mit unterschiedlichem Wert.
- Deshalb gilt in Java: Mehrfachvererbung ist nicht möglich!  
~~`public class Tutor extends Student, Dozent {`~~
- Anmerkung:
  - Ohne Mehrfachvererbung ist die Klassenhierarchie ein einfacher Baum, z.B. in Java oder Smalltalk.
  - Um Mehrfachvererbung zu ermöglichen, müssen die genannten Konflikte aufgelöst werden, z.B. in C++ und Eiffel.

# Beispiel

- Hochschulangehörige:  
Einfachvererbung



# Einfachvererbung

- Ohne Dozent in der Vererbungskette müssen Prof und Tutor Ihr Fachgebiet selbst implementieren → Mehraufwand

Dozent
- <code>String fachgebiet</code>
- <code>toString()</code>
- <code>getFachgebiet()</code>

- Weitere Nachteile:
  - Wir kennen in Java dann keinen Typ „Dozent“.
  - Die Gemeinsamkeit von `Professor` und `Tutor` ist im System nicht mehr erkennbar.
  - Eventuell könnten `Professor` und `Tutor` die Funktionalität unterschiedlich implementieren, z.B.

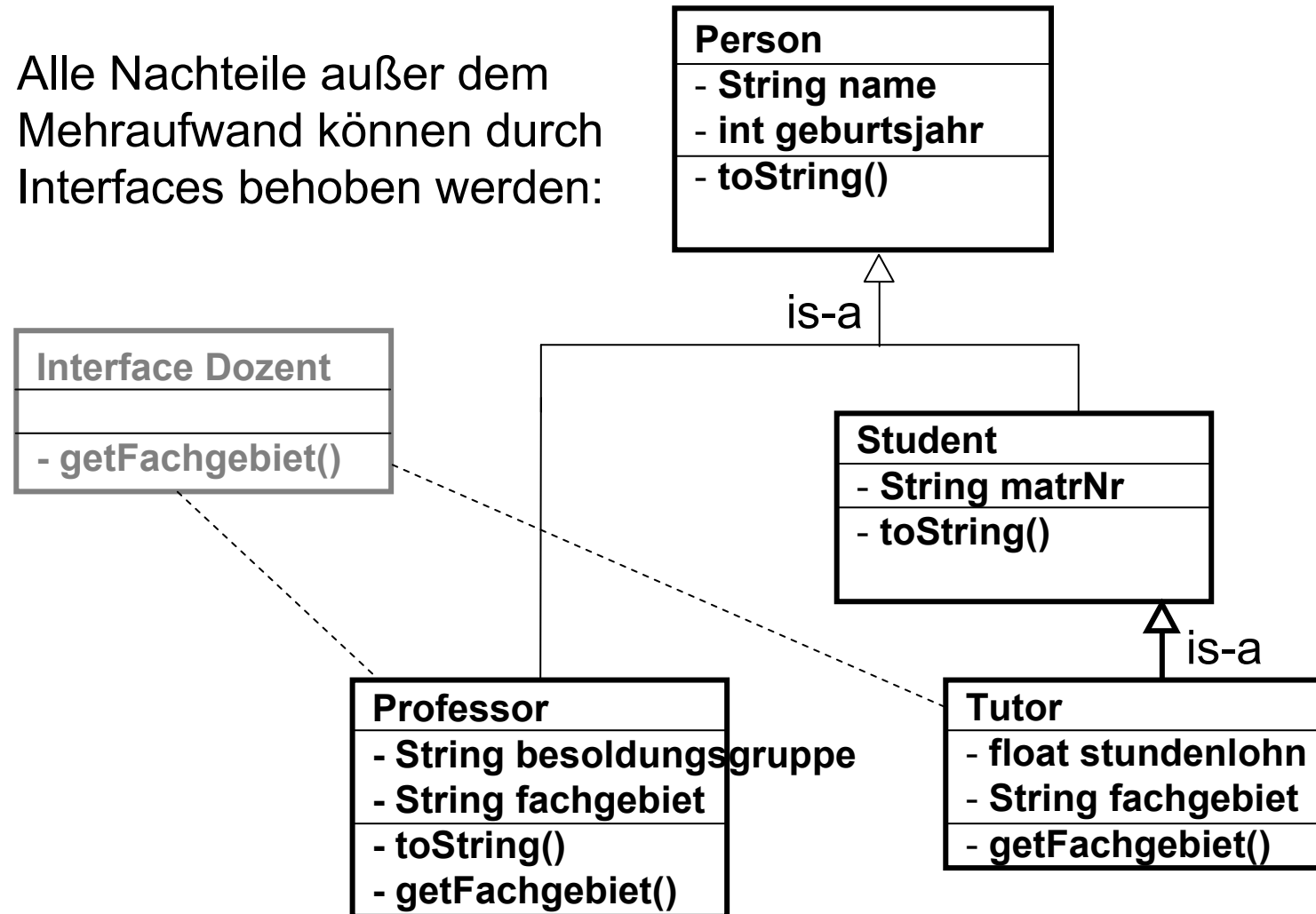
Professor
-
- <code>getFachgebiet()</code>

Tutor
-
- <code>fachgebiet()</code>

- Dann müssen diese auch von aufrufenden Programmen unterschiedlich behandelt werden
- Eine einheitliche Behandlung aller `Dozent`-Objekte ist dann nicht mehr möglich.

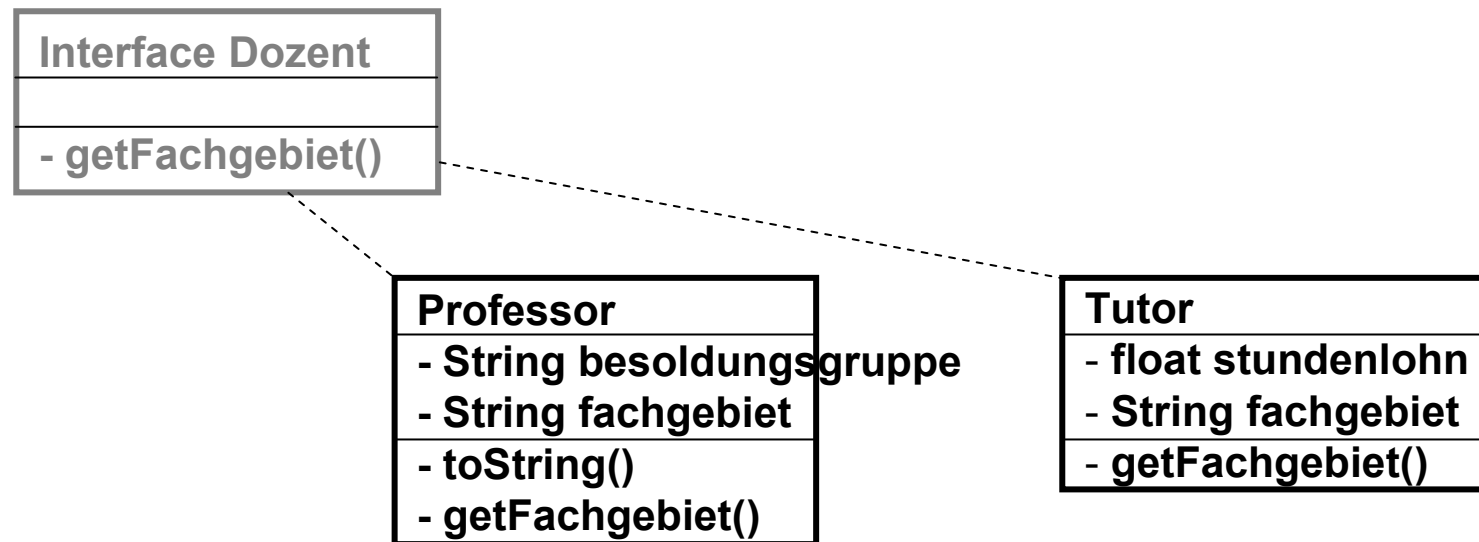
# Interfaces

- Alle Nachteile außer dem Mehraufwand können durch Interfaces behoben werden:



# Interfaces

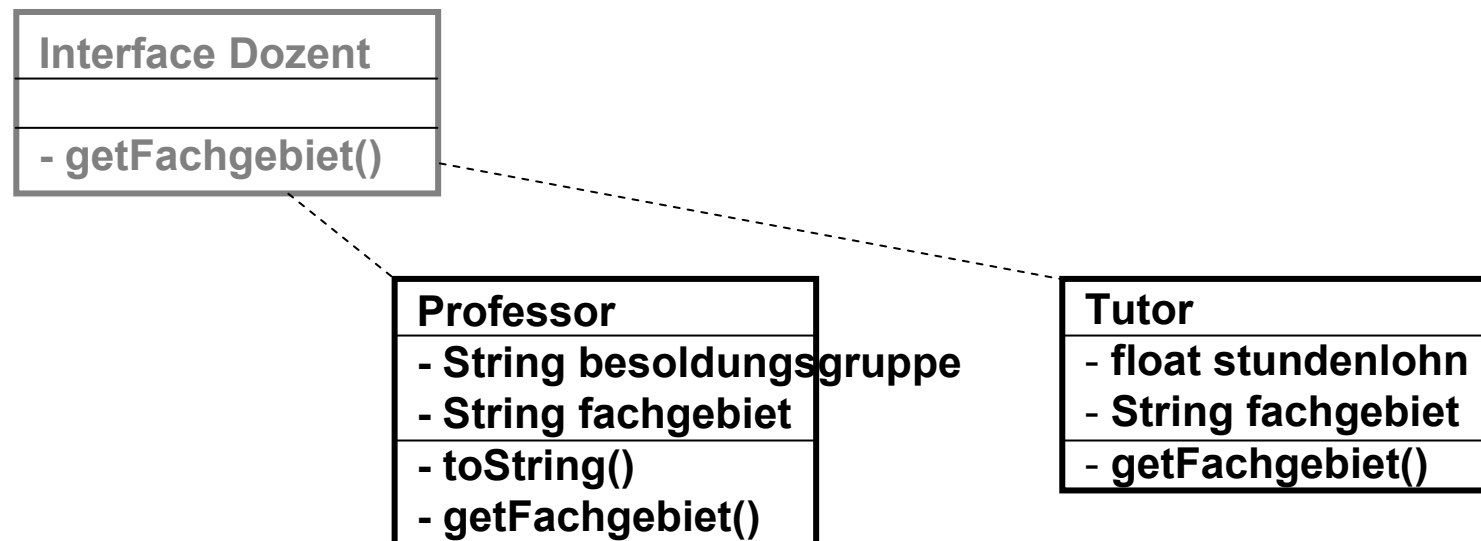
- Ein Interface ist
  - eine Sammlung **abstrakter** Methoden
  - evtl. statische Konstanten
  - Keine Funktionalität
- Man sagt, „eine Klasse implementiert ein Interface“, wenn Sie alle (abstrakten) Methoden des Interfaces implementiert.



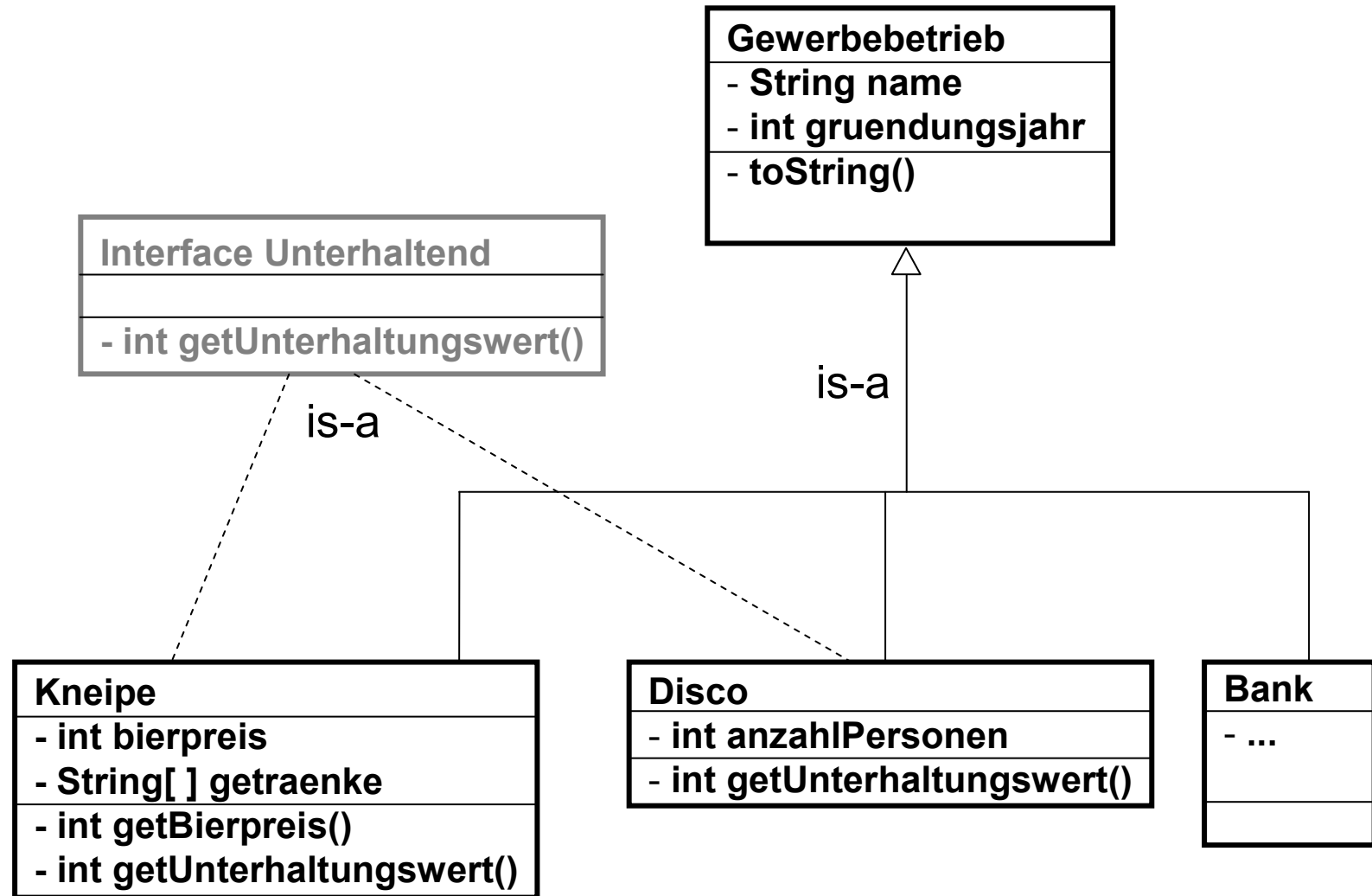


# Interfaces

- Vorteile
  - Wir kennen dann in Java einen Typ „Dozent“.
  - Die Gemeinsamkeit von `Professor` und `Tutor` ist im System erkennbar.
  - `Professor` und `Tutor` implementieren die Funktionalität identisch.
  - Aufrufende Programmen können diese gleich behandeln
  - Eine einheitliche Behandlung aller `Dozent`-Objekte ist möglich.

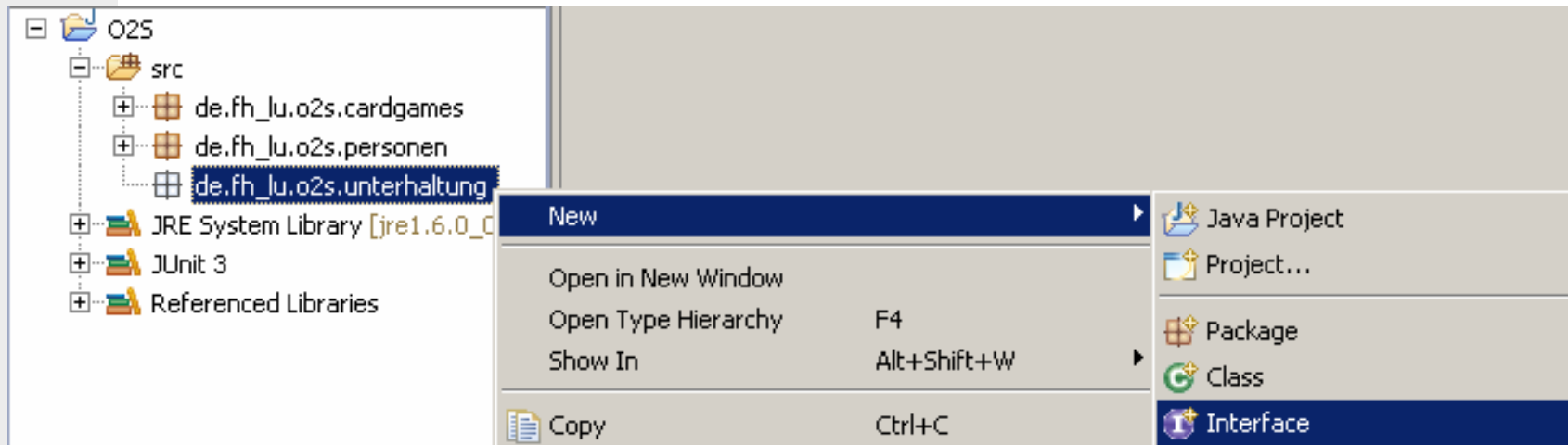


# Anderes Beispiel



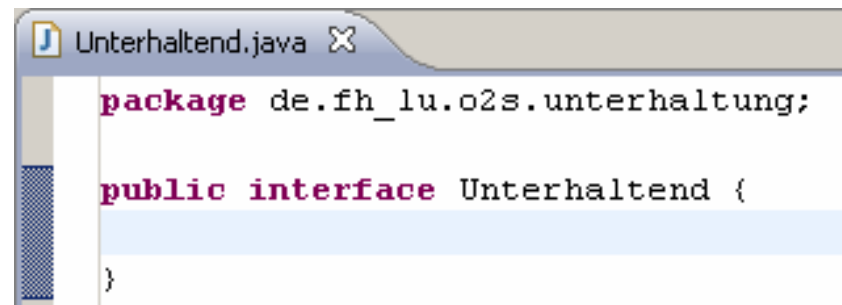
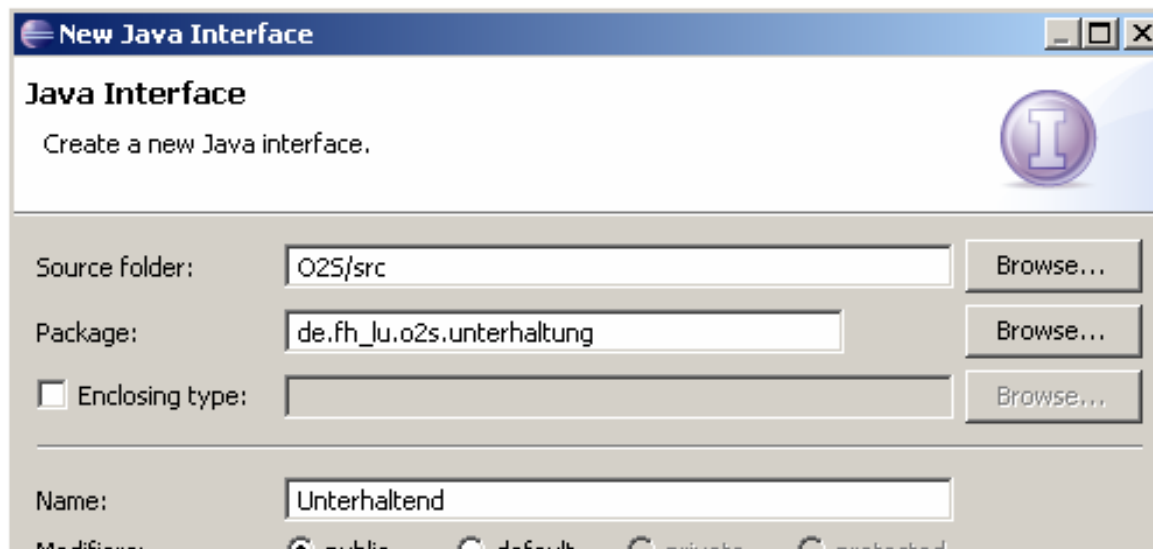
# Los geht's

- Aktion 1: Entwickeln Sie ein Interface „Unterhaltend“
  - mit einer (abstrakten) Methode „getUnterhaltungswert()“
  - in einem (neuen) package „de.fh\_lu.o2s.unterhaltung“:
- Lösungsansatz:
  - Analog zu abstrakten Klassen
  - Schlüsselwort „interface“ anstatt „abstract class“.



# Interface Unterhaltend

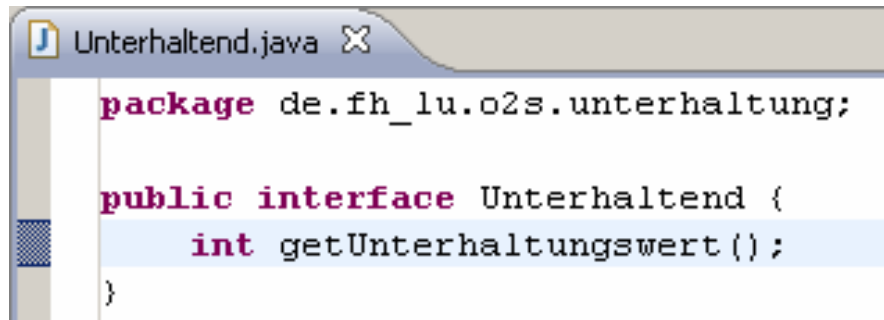
- Lösung, Schritt 1: Interface „Unterhaltend“ anlegen



```
package de.fh_lu.o2s.unterhaltung;  
  
public interface Unterhaltend {  
  
}
```

# Interface Unterhaltend

- Lösung, Schritt 2:
  - (Abstrakte) Methode „getUnterhaltungswert()“ anlegen



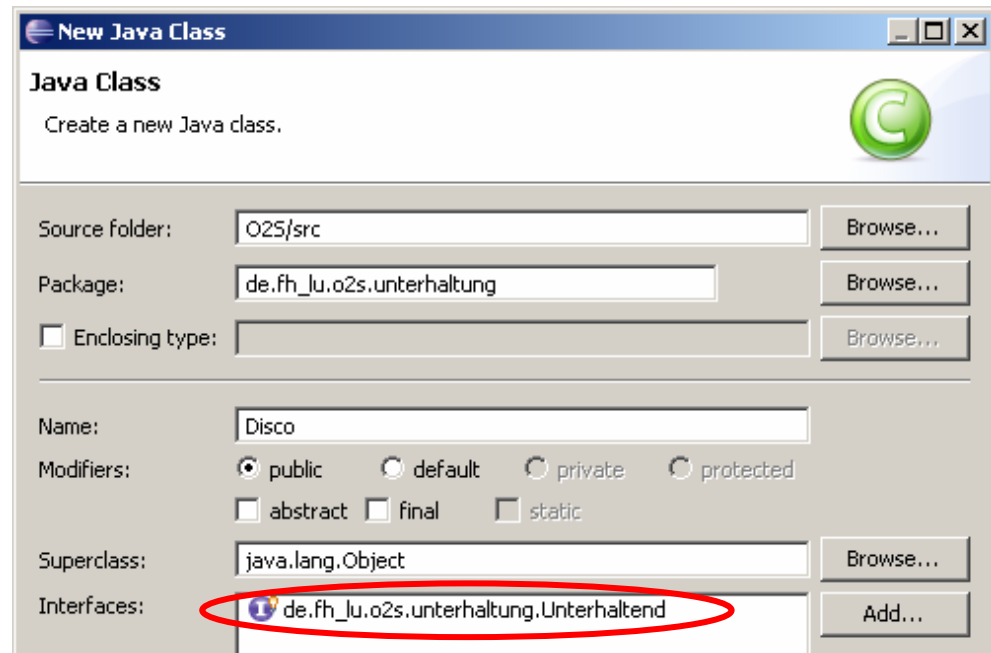
```
Unterhaltend.java X
package de.fh_lu.o2s.unterhaltung;

public interface Unterhaltend {
    int getUnterhaltungswert();
}
```

- Anmerkung:
  - Alle Methoden in Interfaces sind automatisch `public` und `abstract`, deshalb braucht man das nicht extra hinzuschreiben.
  - Oft beschreiben Interfaces eine Eigenschaft, z.B. `Comparable`, `Serializable`, `Unterhaltend`, **etc.**

# Disco

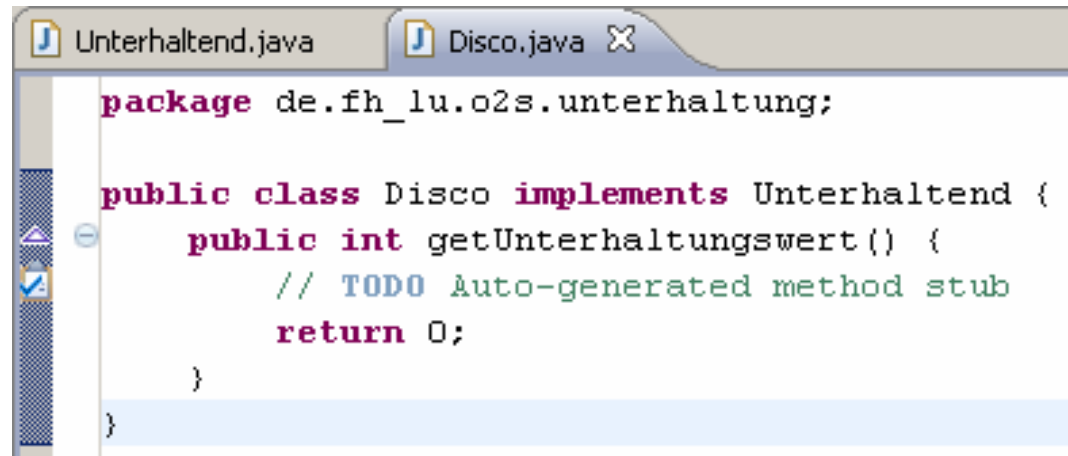
- Aktion 2: Definieren Sie die Klassen „Disco“ und „Kneipe“ wie oben angegeben
  - so dass diese das Interface „Unterhaltend“ implementieren
  - Anmerkung: Die Klasse „Gewerbebetrieb“ lassen wir erstmal weg.
- Lösung, Schritt 1: Zunächst die Disco.
  - Wir können das Interface bereits bei der Klassendefinition angeben.



The screenshot shows the 'New Java Class' dialog box. The 'Name' field is set to 'Disco'. The 'Package' field is set to 'de.fh\_lu.o2s.unterhaltung'. The 'Superclass' field is set to 'java.lang.Object'. The 'Interfaces' field is set to 'de.fh\_lu.o2s.unterhaltung.Unterhaltend', which is circled in red. The 'Modifiers' section shows 'public' selected. The 'Enclosing type' checkbox is unchecked.

# Disco

- Beobachtung: Dann erzeugt Eclipse für uns automatisch
  - eine „implements“ Klausel in der Klassendeklaration und
  - die nötigen Methoden, die wir implementieren müssen

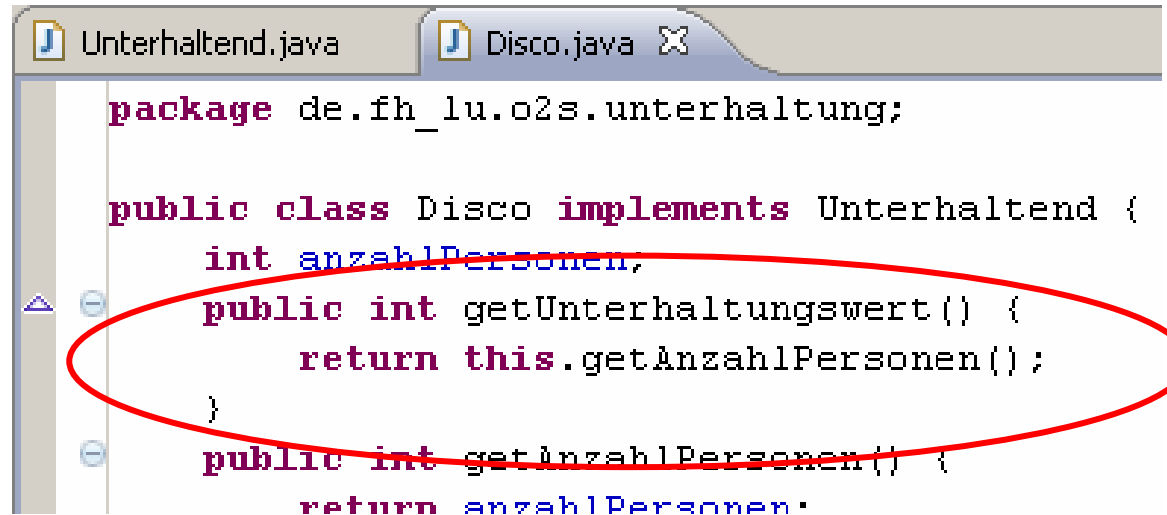


```
package de.fh_lu.o2s.unterhaltung;

public class Disco implements Unterhaltend {
    public int getUnterhaltungswert() {
        // TODO Auto-generated method stub
        return 0;
    }
}
```

- Anmerkung:
  - Wir hätten beides auch von Hand in den Klassentext schreiben können.

- Lösung, Schritt 2a:
  - Einfügen des Attributs „anzahlPersonen“ und
  - der zugehörigen `get-` und `set-`Methoden
- Lösung, Schritt 2b: Ausprogrammieren der Interface-Methode
  - Ein geeignetes Maß für den Unterhaltungswert könnte z.B. die Größe der Disco sein. Grob vereinfacht: Je größer, desto besser.

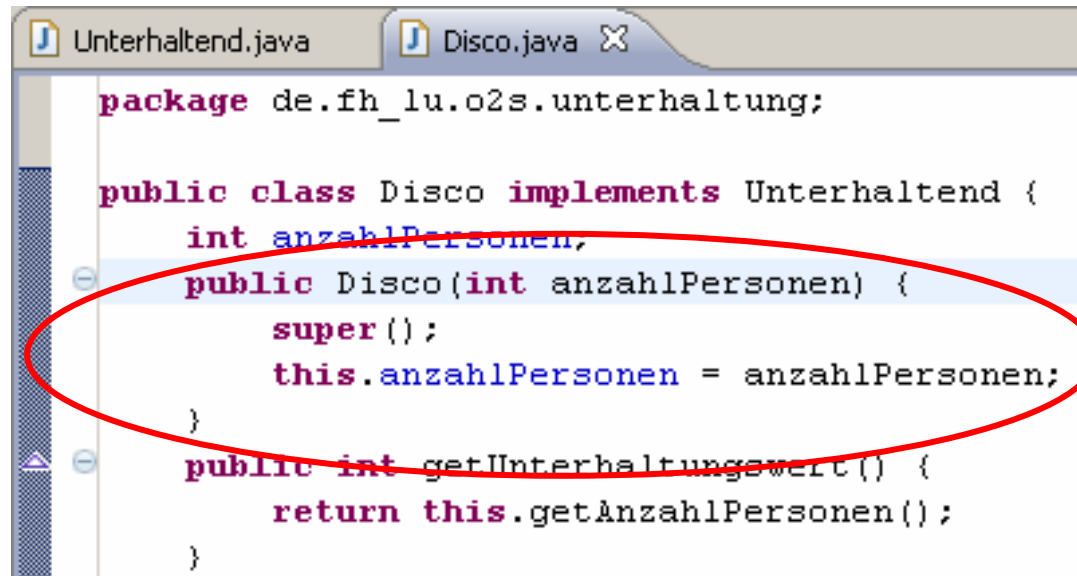


```
package de.fh_lu.o2s.unterhaltung;

public class Disco implements Unterhaltend {
    int anzahlPersonen;
    public int getUnterhaltungswert() {
        return this.getAnzahlPersonen();
    }
    public int getAnzahlPersonen() {
        return anzahlPersonen;
    }
}
```



- Lösung, Schritt 3:
  - Der Vollständigkeit halber soll die Klasse `Disco` noch einen Konstruktor bekommen.



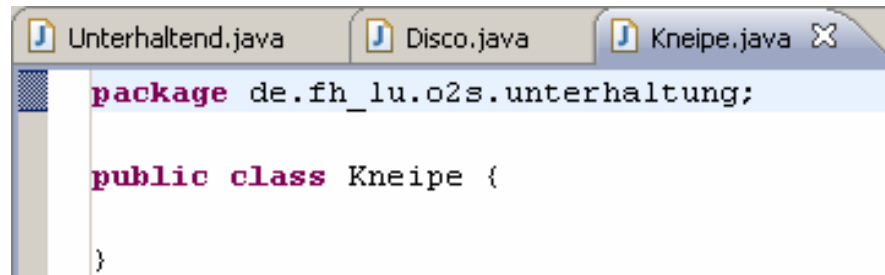
```
package de.fh_lu.o2s.unterhaltung;

public class Disco implements Unterhaltend {
    int anzahlPersonen;
    public Disco(int anzahlPersonen) {
        super();
        this.anzahlPersonen = anzahlPersonen;
    }
    public int getUnterhaltungswert() {
        return this.getAnzahlPersonen();
    }
}
```

- Anmerkung:
  - Die Methoden, die vom Interface vorgeschrieben werden, müssen `public` sein.

# Kneipe

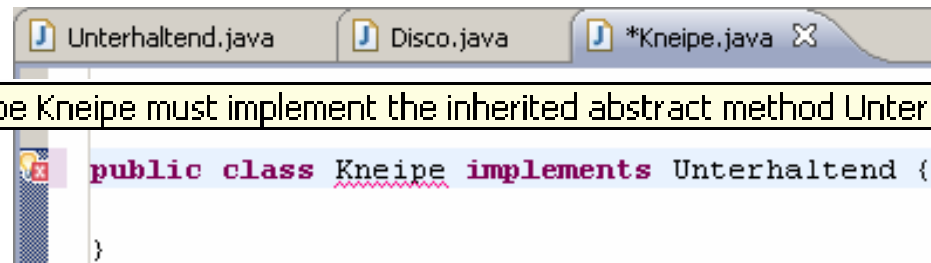
- Lösung, Schritt 4:
  - Zu Lernzwecken legen wir die `Kneipe` zunächst an, ohne das Interface `Unterhaltend` anzugeben.



```
package de.fh_lu.o2s.unterhaltung;

public class Kneipe {
}
```

- Lösung, Schritt 5:
  - Wir fügen die Klausel „implements `Unterhaltend`“ hinzu



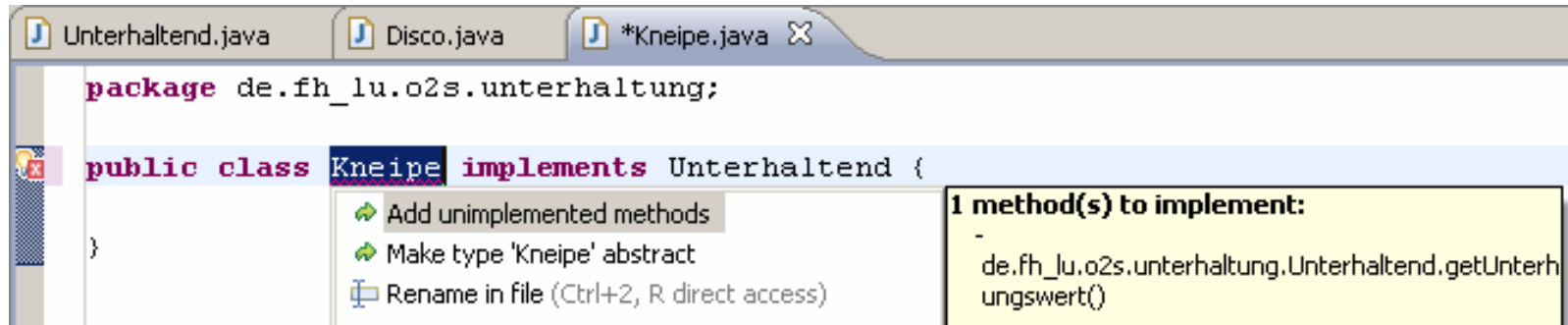
```
public class Kneipe implements Unterhaltend {
}
```

The type Kneipe must implement the inherited abstract method Unterhaltend.getUnterhaltungswert()

- und bekommen einen Fehler, weil die Methode `getUnterhaltungswert()` noch nicht implementiert ist

# Kneipe

- Lösung, Schritt 6:
  - Um den Fehler zu beseitigen, implementieren wir `getUnterhaltungswert()` zunächst als leere Methode
  - Eclipse hilft uns dabei mit „QuickFix“:



The screenshot shows the Eclipse IDE with three tabs: `Unterhaltend.java`, `Disco.java`, and `*Kneipe.java`. The `Kneipe` class is selected, and a QuickFix menu is open. The menu options are:

- Add unimplemented methods
- Make type 'Kneipe' abstract
- 🔗 Rename in file (Ctrl+2, R direct access)

On the right, a yellow box indicates "1 method(s) to implement:" with the following details:

- `de.fh_lu.o2s.unterhaltung.Unterhaltend.getUnterhaltungswert()`

- Vorläufiges Ergebnis:

```
public class Kneipe implements Unterhaltend {  
  
    @Override  
    public int getUnterhaltungswert() {  
        // TODO Auto-generated method stub  
        return 0;  
    }  
}
```

- Auf die Annotation „`@Override`“ gehen wir hier nicht ein.

# Kneipe

- Lösung, Schritt 7:
  - Attribute `bierpreis` und `getraenke`, get- und set-Methoden und Konstruktor



```
Unterhaltend.java  Disco.java  Kneipe.java X
package de.fh_lu.o2s.unterhaltung;

public class Kneipe implements Unterhaltend {
    float bierpreis = (float) 2.40;
    String[] getraenke = {"Bier", "mehr Bier", "noch mehr Bier"};
    public Kneipe(float bierpreis, String[] getraenke) {
        super();
        this.bierpreis = bierpreis;
        this.getraenke = getraenke;
    }
    public float getBierpreis() {
        return bierpreis;
    }
}
```

# Kneipe

- Lösung, Schritt 8: Berechnen des Unterhaltungswerts
  - Je mehr Getränkesorten es gibt, umso besser
  - Je billiger das Bier ist, umso besser
  - Skalierung, damit es mit einer `Disco` vergleichbar wird.

```
public int getUnterhaltungswert() {  
    double wert = gettraenke.length / getBierpreis();  
    return 10 * (int) Math.floor(wert); //Rückgabewert muss int sein  
}
```

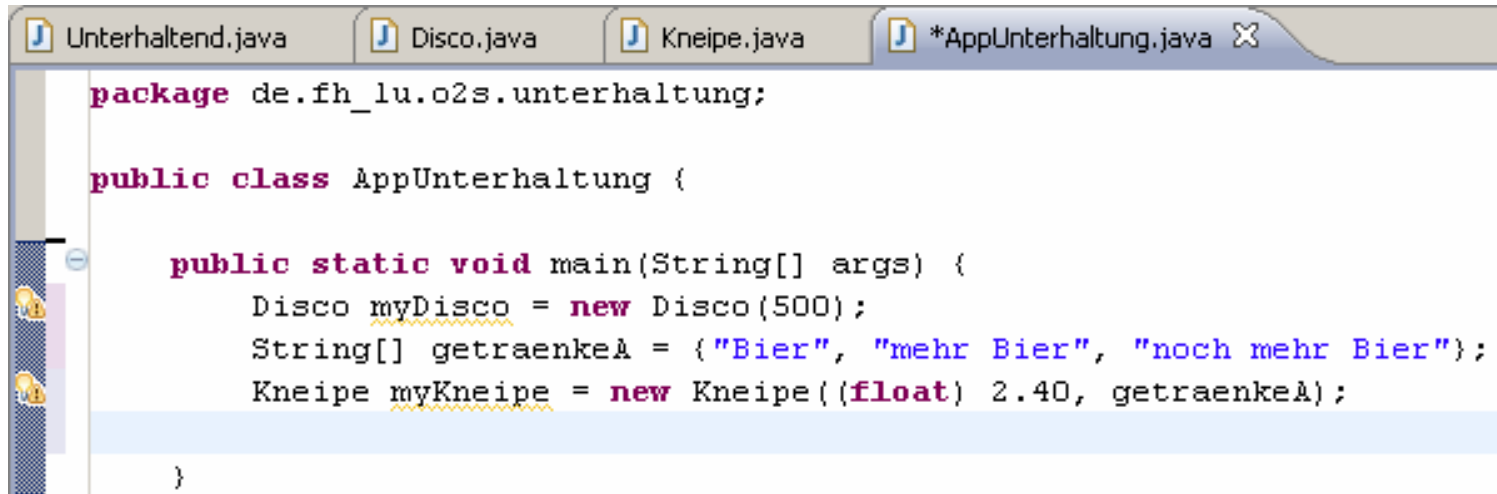
- Anmerkungen:
  - Der Unterhaltungswert könnte natürlich auch anders berechnet werden.
- Lösung, Schritt 9:
  - `toString()` Methoden für `Disco` und `Kneipe`
  - Fingerübung

# Testapplikation

- Aktion 3: Entwickeln Sie eine Anwendung `AppUnterhaltung`, die die Klassen `Disco` und `Kneipe` testet:
  - Legen Sie eine `Disco` an, die Platz für 500 Gäste bietet.
  - Legen Sie eine `Kneipe` mit einem Bierpreis von 2.50 und mindestens drei Getränken an.
  - Legen Sie ein Array mit zwei Speicherplätzen vom Datentyp `Unterhaltend` an und fügen Sie Ihre `Disco` und Ihre `Kneipe` ein.
  - Durchlaufen Sie Ihr Array mit einer `foreach`-Schleife und
    - geben Sie die enthaltenen Objekte aus,
    - ermitteln Sie den Gesamt-Unterhaltungswert aller enthaltenen Objekte zusammen.

# Testapplikation

- Lösung, Schritt 1: Disco und Kneipe anlegen:
  - Konstruktoren nutzen:



```
Unterhaltend.java  Disco.java  Kneipe.java  *AppUnterhaltung.java X

package de.fh_lu.o2s.unterhaltung;

public class AppUnterhaltung {

    public static void main(String[] args) {
        Disco myDisco = new Disco(500);
        String[] getraenkeA = {"Bier", "mehr Bier", "noch mehr Bier"};
        Kneipe myKneipe = new Kneipe((float) 2.40, getraenkeA);
    }
}
```

- Anmerkungen:
  - Die Array-Formulierung {...} kann nicht innerhalb des Konstruktor-Aufrufs geschrieben werden.
  - 2.40 ist vom Datentyp double und muss deshalb auf float gecastet werden, damit der Konstruktor es akzeptiert.

# Testapplikation

- Lösung, Schritt 2: `Unterhaltend`-Array anlegen:

```
Unterhaltend[] betriebeA = new Unterhaltend[2];  
betriebeA[0] = myDisco;  
betriebeA[1] = myKneipe;
```

- Anmerkungen:
  - Nach dem oben dargestellten Datenmodell gilt „Disco is-a `Unterhaltend`“ und „Kneipe is-a `Unterhaltend`“
  - `Disco` und `Kneipe` „passen“ also auf den (Interface-)Datentyp `Unterhaltend`.
  - Deshalb kann `Unterhaltend` als Referenztyp verwendet werden
    - für Objekte vom Typ `Disco` oder `Kneipe`
    - allgemeiner: Für Objekte, die das Interface `Unterhaltend` implementieren.



# Testapplikation

- Beobachtung:

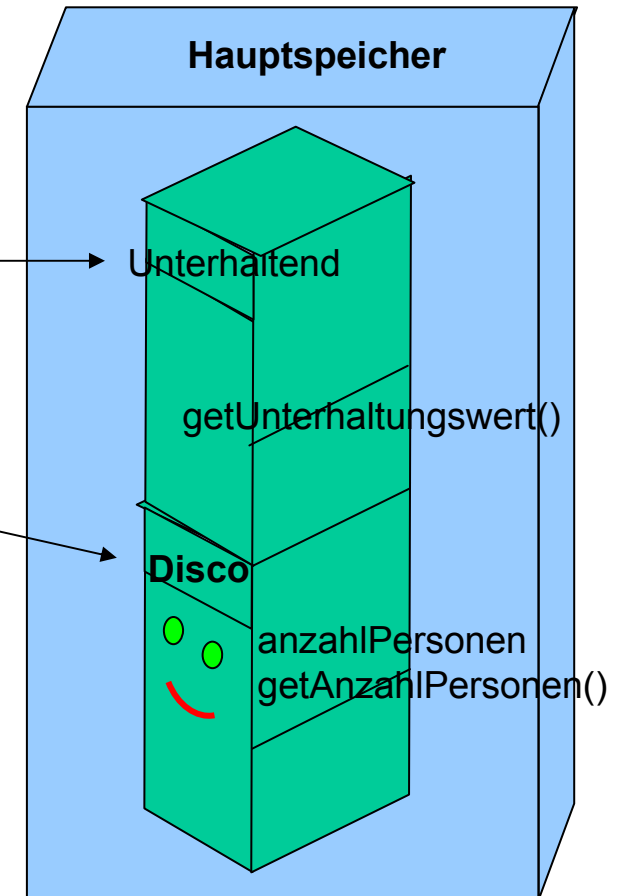
`betriebeA[0] = myDisco;`

**Unterhaltend betriebeA[0]**

**Disco myDisco**

- Anmerkung:

- Bei AppUnterhaltung sind die Plätze im Array, also `betriebeA[0]` und `betriebeA[1]`, jeweils vom Datentyp `Unterhaltend`



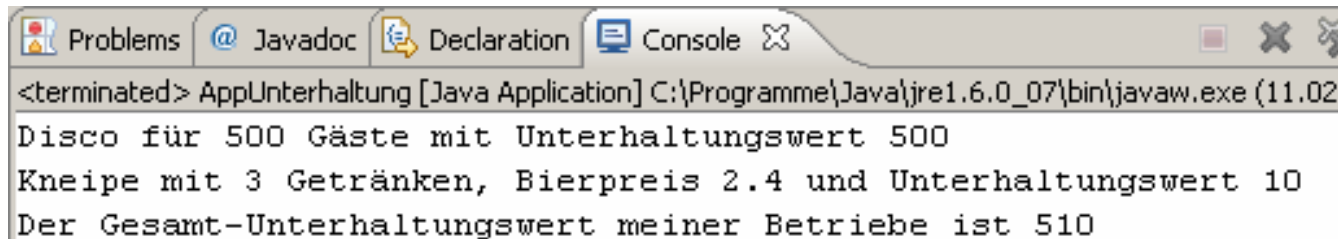
```
Unterhaltend[] betriebeA = new Unterhaltend[2];  
betriebeA[0] = myDisco;  
betriebeA[1] = myKneipe;
```

# Testapplikation

- Lösung, Schritt 3: Schleife programmieren:

```
int gesamtwert = 0;
for (Unterhaltend myBetrieb : betriebeA){
    System.out.println(myBetrieb.toString());
    gesamtwert += myBetrieb.getUnterhaltungswert();
}
System.out.println("Der Gesamt-Unterhaltungswert " +
    "meiner Betriebe ist " + gesamtwert);
```

- Anmerkungen:
  - An eine Variable vom Typ `Unterhaltend` können Methoden geschickt werden, die entweder in `Unterhaltend` oder in `Object` definiert sind.
  - Wird eine Methode aus `Object` aufgerufen, die im Objekt überschrieben wurde, dann wird die überschreibende Methode ausgeführt, z.B. `toString()`, weil `myBetrieb` zur Laufzeit eine `Disco` bzw. eine `Kneipe` ist. → **Dynamisches Binden!**



Problems Javadoc Declaration Console

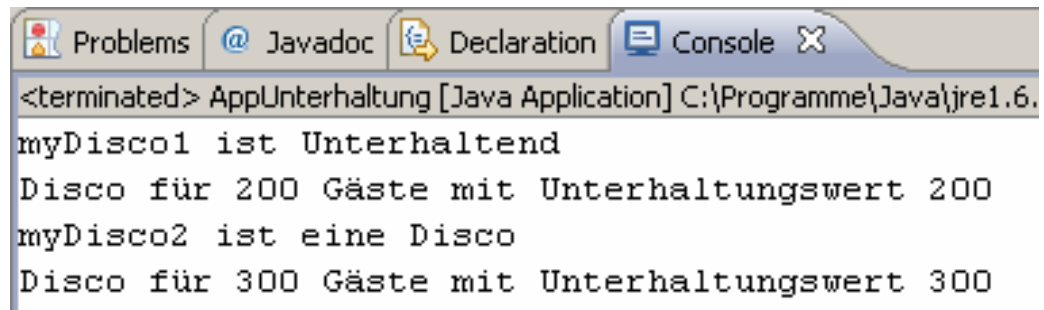
<terminated> AppUnterhaltung [Java Application] C:\Programme\Java\jre1.6.0\_07\bin\javaw.exe (11.02)

Disco für 500 Gäste mit Unterhaltungswert 500  
Kneipe mit 3 Getränken, Bierpreis 2.4 und Unterhaltungswert 10  
Der Gesamt-Unterhaltungswert meiner Betriebe ist 510

# Testapplikation

- Anmerkung:
  - Folgendes Casting funktioniert, alle `System.out`-Methoden werden ausgeführt:

```
Disco      myDisco1 = new Disco(200);
Unterhaltend myDisco2 = new Disco(300);
if (myDisco1 instanceof Unterhaltend){
    System.out.println("myDisco1 ist Unterhaltend");
    Unterhaltend myUnterhaltend = (Unterhaltend) myDisco1;
    System.out.println(myUnterhaltend);
}
if (myDisco2 instanceof Disco){
    System.out.println("myDisco2 ist eine Disco");
    Disco myNewDisco = (Disco) myDisco2;
    System.out.println(myNewDisco);
}
```



The screenshot shows a Java IDE window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of the application. The output is as follows:

```
<terminated> AppUnterhaltung [Java Application] C:\Programme\Java\jre1.6.
myDisco1 ist Unterhaltend
Disco für 200 Gäste mit Unterhaltungswert 200
myDisco2 ist eine Disco
Disco für 300 Gäste mit Unterhaltungswert 300
```

# Interface Comparable

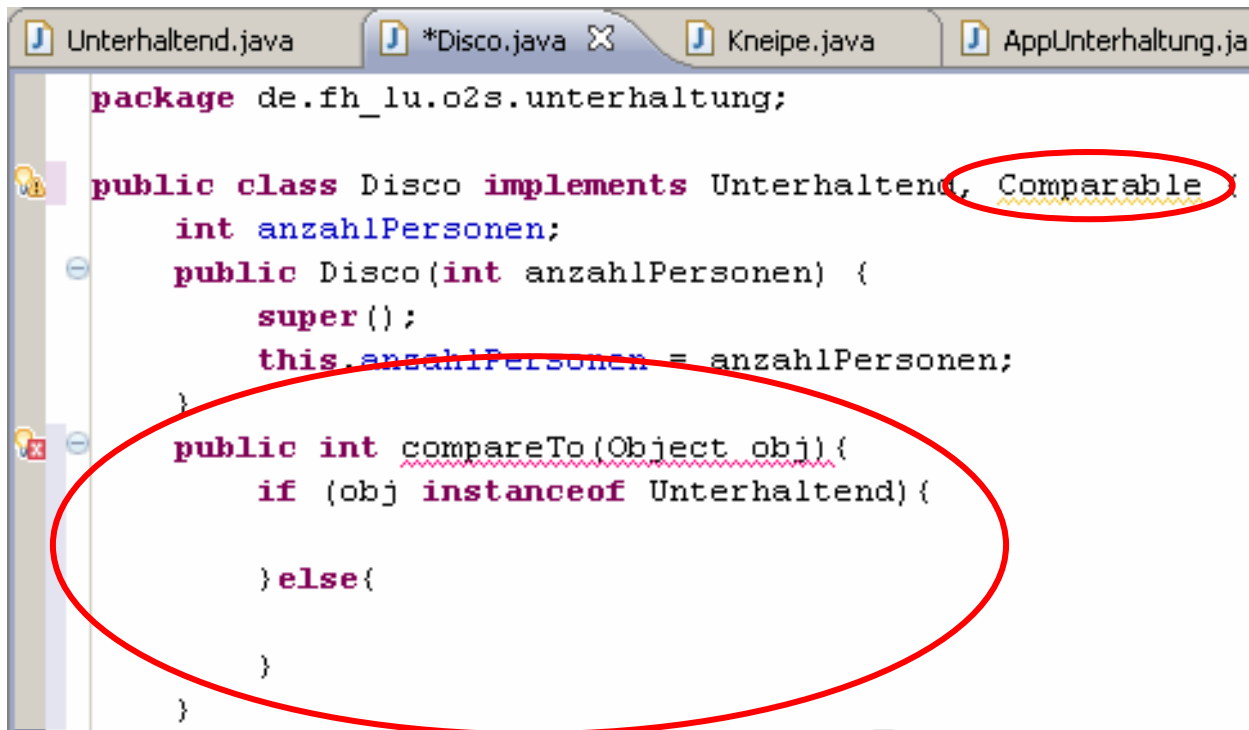
- Das Interface `java.lang.Comparable`
  - enthält die Methode `int compareTo(Object obj)`
  - Das ausführende Objekt (`this`) wird mit dem Parameter-Objekt (`obj`) verglichen.
- Aufrufbeispiel:
  - `myDisco1.compareTo(myDisco2)`
- Ergebnis des Vergleichs soll ein Integer-Wert sein:
  - < 0, wenn `this < obj` (z.B. `myDisco1 < myDisco2`)
  - = 0, wenn `this` gleich groß ist wie `obj` (z.B. `myDisco1 = myDisco2`)
  - > 0, wenn `this > obj` (z.B. `myDisco1 > myDisco2`)
- Frage:
  - Wie vergleichen wir Discos?
  - Nach Größe? / Nach Unterhaltungswert? / Nach dem Namen? / Nach der Adresse?

# Interface Comparable

- Aktion 4: Sorgen Sie dafür,
  - dass `Disco` das Interface `Comparable` implementiert.
  - Die Methode `compareTo(Object obj)` soll Unterhaltend-e Objekte nach dem Unterhaltungswert vergleichen
- Erinnerung: Ergebnis soll ein Integer-Wert sein:
  - `this.compareTo(obj) < 0`, wenn `this < obj`
  - `this.compareTo(obj) == 0`, wenn `this` gleich groß ist wie `obj`
  - `this.compareTo(obj) > 0`, wenn `this > obj`
- Lösungsansatz:
  - Prüfen ob `obj` ein Unterhaltend-es Objekt ist und ggfs.
  - Vergleich von `this.getUnterhaltungswert()` mit `obj.getUnterhaltungswert()`

# Interface Comparable

- Lösung, Schritt 1:
  - implements-Klausel in der Klassendeklaration von Disco einfügen,
  - Methode `compareTo(Object obj)` anlegen,
  - In der Methode zunächst prüfen, ob `obj` Unterhaltend ist



```
package de.fh_lu.o2s.unterhaltung;

public class Disco implements Unterhaltend, Comparable {
    int anzahlPersonen;

    public Disco(int anzahlPersonen) {
        super();
        this.anzahlPersonen = anzahlPersonen;
    }

    public int compareTo(Object obj) {
        if (obj instanceof Unterhaltend) {

        } else {

        }
    }
}
```

# Interface Comparable

- Lösung, Schritt 2:

- Vergleich der Unterhaltungswerte und Rückgabe eines Wertes: positiv, negativ oder 0

```
public int compareTo(Object obj){  
    if (obj instanceof Unterhaltend){  
        int myWert = this.getUnterhaltungswert();  
        int objWert = ((Unterhaltend) obj).getUnterhaltungswert();  
        return myWert - objWert;  
    }else{  
  
    }  
}
```

- Beobachtungen:

- obj muss auf einen Datentyp gecastet werden, der die Methode `getUnterhaltungswert()` kennt.
- Wenn der Unterhaltungswert von `this` kleiner ist als der von `obj`, dann ist das Ergebnis  $< 0$ , das bedeutet `this < obj`.
- Auch für gleich / größer stimmt das Ergebnis.
- Ein Fehler tritt (noch) auf, weil im `else`-Fall kein Wert zurückgegeben wird.

# Interface Comparable

- Lösung, Schritt 3:

- Wenn obj nicht Unterhaltend ist, sind this und obj nicht vergleichbar. Wir geben dann einfach +1 zurück.

```
public int compareTo(Object obj){  
    if (obj instanceof Unterhaltend){  
        int myWert = this.getUnterhaltungswert();  
        int objWert = ((Unterhaltend) obj).getUnterhaltungswert();  
        return myWert - objWert;  
    }else{  
        return 1;  
    }  
}
```

- Anmerkungen:

- Wenn obj nicht Unterhaltend ist, ist es damit „kleiner“ als jedes Unterhaltend-e Objekt.
- Wir hätten alternativ eine Exception werfen können, vgl. nächste Vorlesung.
- Wenn this und obj Unterhaltend sind, kann jede int-Zahl als Ergebnis auftreten. Wenn wir nur -1, 0 +1 als Ergebnis haben wollen, können wir die Funktion `Math.signum(...)` verwenden.



# Beobachtungen

- Beobachtung 1:
  - Eine Klasse kann mehrere Interfaces implementieren (aber nur von einer Klasse erben)
- Beobachtung 2:
  - Eine `Disco` kann mit jedem „Unterhaltend“-en Objekt verglichen werden, z.B. mit einer `Kneipe`.
  - Eine `Disco` kann **nur** mit „Unterhaltend“-en Objekten verglichen werden.
  - Dadurch dass in der Implementierung nicht der Klassen-Typ `Disco` sondern der Interface-Typ `Unterhaltend` verwendet wurde, kann der Code 1-zu-1 auch für `Kneipe` verwendet werden (muss dafür aber kopiert werden).
  - Bitte tun Sie das.
  - Damit können alle `Disco`-s und `Kneipe`-n mit der Methode `compareTo()` verglichen werden.

# Vergleichen

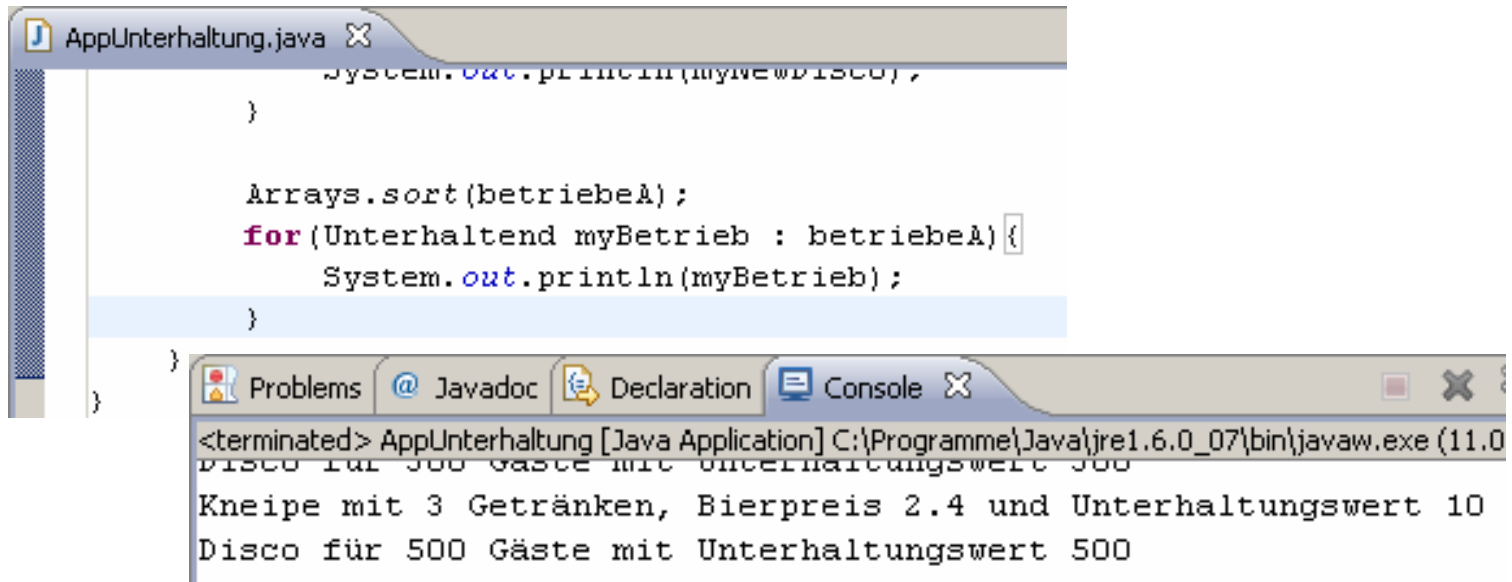
- Beobachtung 3:
  - Für Unterhaltend-Objekte  $u_1$ ,  $u_2$ ,  $u_3$  sind folgende mathematischen Zusammenhänge notwendig:
    - Wenn  $u_1 < u_2$ , und  $u_2 < u_1$ , dann ist  $u_1$  gleichgroß wie  $u_2$
    - Wenn  $u_1 < u_2$ , und  $u_2 < u_3$ , dann ist  $u_1 < u_3$
  - In Methoden ausgedrückt heißt das:
    - `u1.compareTo(u2) < 0` und `u2.compareTo(u1) < 0`, dann ist `u2.compareTo(u1) == 0`
    - `u1.compareTo(u2) < 0` und `u2.compareTo(u3) < 0`, dann ist `u1.compareTo(u3) < 0`

# Programmierung nach Vertrag

- Beobachtung 4:
  - Discos und Kneipen sind jetzt „Comparable“
  - Das System weiß das, weil wir es ihm mit der „implements“-Klausel gesagt haben
  - Für Objekte, die `Comparable` sind, steht weitere Funktionalität zur Verfügung, z.B. Sortierung mit `Arrays.sort()`
  - Es wird ein Vertrag geschlossen
    - zwischen dem Java-Framework und uns
    - Wenn wir `Comparable` implementieren, dann erhalten wir die Sortierung geschenkt.

# Interface Comparable

- Lösung, Schritt 4: Testen
  - Erweitern Sie Ihre Anwendung AppUnterhaltung, indem Sie das Array `betriebeA` mit `Arrays.sort(...)` sortieren und
  - mit einer `foreach`-Schleife nochmal auf der Konsole ausgeben.



```
AppUnterhaltung.java X
    System.out.println(myNewDisco);
}

Arrays.sort(betriebeA);
for (Unterhaltend myBetrieb : betriebeA) {
    System.out.println(myBetrieb);
}
}
```

Problems @ Javadoc Declaration Console X

<terminated> AppUnterhaltung [Java Application] C:\Programme\Java\jre1.6.0\_07\bin\javaw.exe (11.0  
Disco für 500 Gäste mit Unterhaltungswert 500  
Kneipe mit 3 Getränken, Bierpreis 2.4 und Unterhaltungswert 10  
Disco für 500 Gäste mit Unterhaltungswert 500

# Gleich vs. Gleichgroß

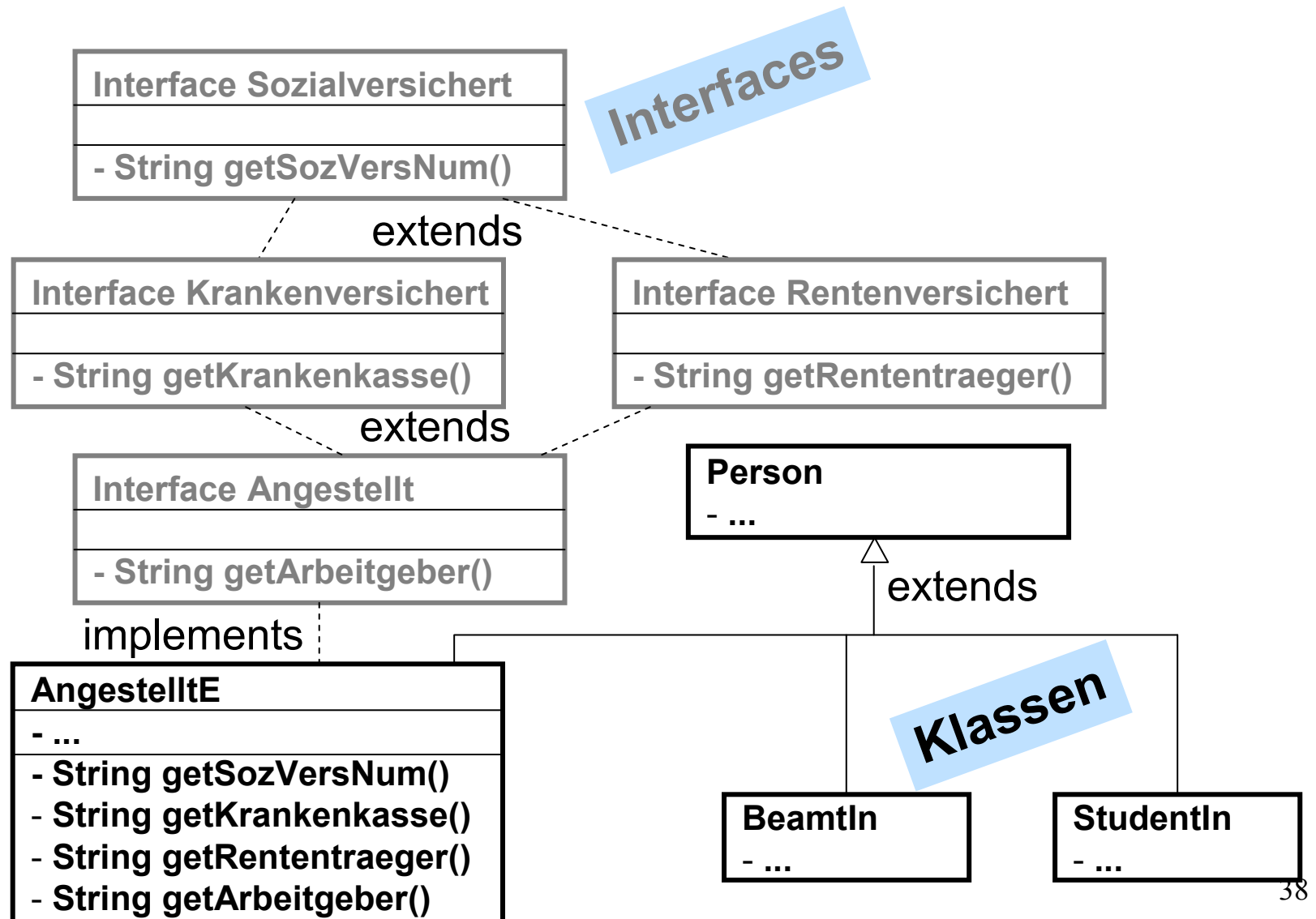
- Gleichheit wird implementiert mit `equals()`
- Discos bzw. Kneipen können „gleich groß“ sein, ohne „gleich“ zu sein:
  - `u1.compareTo(u2) == 0`, **aber** `u1.equals(u2) == false`.
  - Beim Sortieren ist unter „gleich großen“ Objekten keine vernünftige Reihenfolge zu ermitteln.
- Schlimmer ist es, wenn zwei Objekte „gleich“ sind aber nicht „gleich groß“:
  - `u1.equals(u2) == true`, **aber** `u1.compareTo(u2) != 0`.
  - Man sagt dann: `compareTo()` ist inkonsistent zu `equals()`

# Markierungsinterfaces

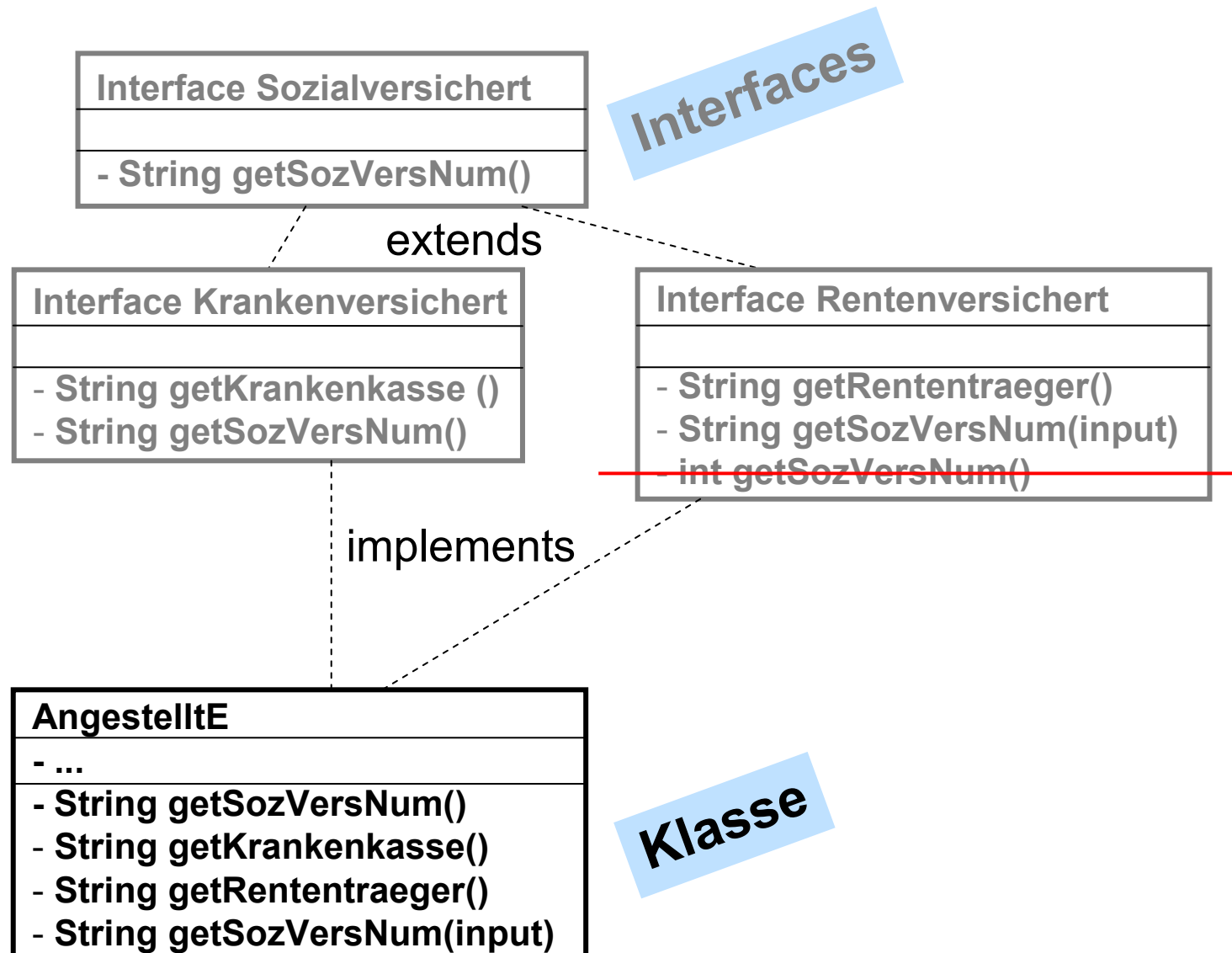
- Eine weitere Eigenschaft ist „Serializable“
  - Dafür müsste die Klasse `Disco` auch noch das Interface `Serializable` implementieren:
  - ```
public class Disco implements Unterhaltend,  
                                     Comparable, Serializable {...
```
- Das Interface „Serializable“
  - hat aber gar keine Methoden...
  - wofür soll das gut sein?
- nur um dem System zu zeigen,
  - dass die betreffende Klasse sich für „serializable“ hält (was immer das bedeuten mag)
  - ```
(myDisco1 instanceof Serializable)
```

 wäre dann `true`
- Man spricht dann von einem „Markierungsinterface“.

# Hierarchien von Interfaces



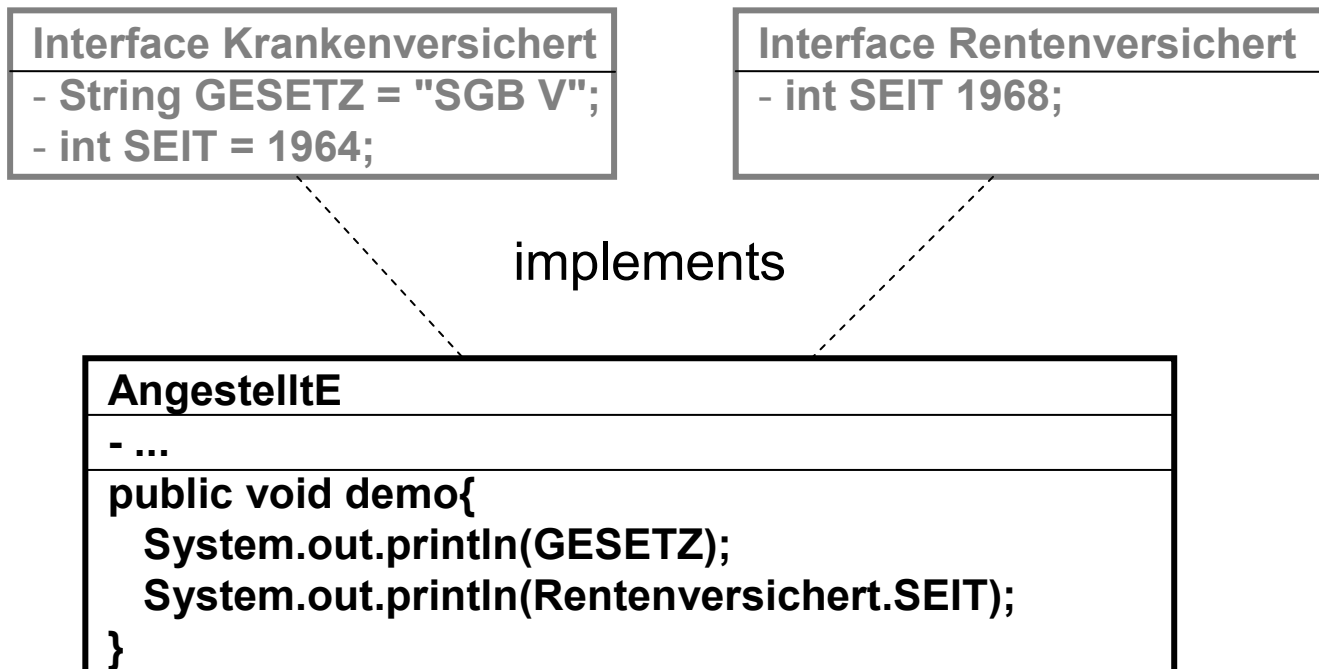
# Konflikte?





# Konstanten

- Interfaces dürfen auch Konstanten enthalten.
  - Diese sind immer `static` und `final` und
  - werden an die implementierenden Klassen „vererbt“.
  - Konflikte können auftreten und gelöst werden
- Bsp.:



# Interface CharSequence

- Die Klasse `String` kennt u.a. die folgenden Methoden:
  - `char charAt(index int)`
  - `int length()`
  - `CharSequence subSequence(int start, int end)`
  - `String toString()`
- Funktionalität: Vgl. <http://java.sun.com/j2se/1.5.0/docs/api/>  
→ Ausprobieren.
- Die Klasse `StringBuffer` kennt diese Methoden auch.  
→ Ausprobieren.
- Das ist kein Zufall, denn beide implementieren das Interface `java.lang.CharSequence`, das aus genau den genannten Methoden besteht.
- Anmerkung:
  - `toString()` muss nicht unbedingt implementiert werden, weil eine Version davon von `Object` geerbt wird.

# Ausprobieren

```
package de.fh_lu.o2s.charseq;
public class AppTestCharSeq {
    public static void main(String[] args) {
//4 Möglichkeiten der Initialisierung
        //String      str = "Unterhaltung";
        //StringBuffer str = new StringBuffer("Unterhaltung");
        //CharSequence str = "Unterhaltung";
        CharSequence str = new StringBuffer("Unterhaltung");
        //CharArray    str = "Unterhaltung";
//ein bisschen Funktionalität
        System.out.println("Unsere CharSequence ist " + str);
        System.out.println("Erstes Zeichen ist " + str.charAt(0));
        System.out.println("Länge ist " + str.length());
        System.out.println("Subsequenz von 2 bis 8 ist " +
                               str.subSequence(2, 8));
        System.out.println("toString() ergibt " + str.toString());
    }
}
```

- Egal wie initialisiert wird, das Ergebnis ist immer das Gleiche.  
→ Vgl. Übungsaufgaben.

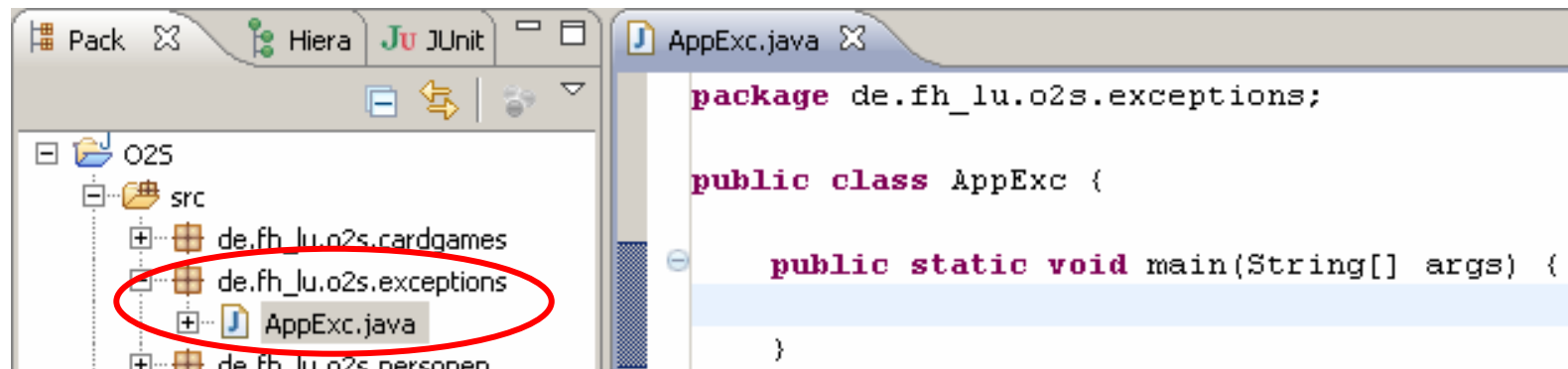
# **Programmierung II**

## **Thema 7: Exceptions**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

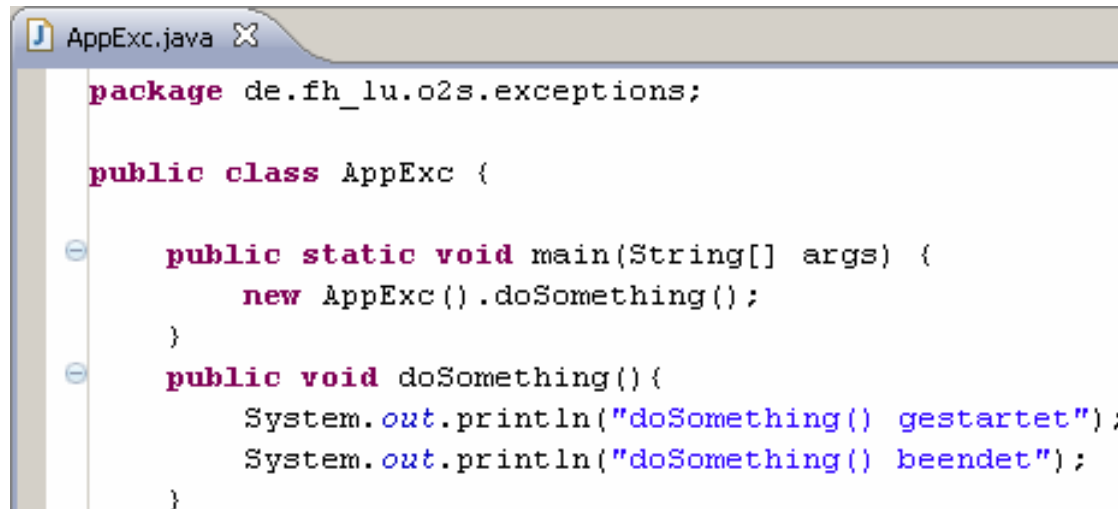
# Einführendes Beispiel

- In vielen modernen Programmiersprachen werden Exceptions verwendet, um Fehler abzufangen.
- Aktion 1: Erzeugen Sie
  - in einem Package `de.fh_lu.o2s.exceptions`
  - eine Java-Anwendung `AppExc`,
  - deren `main()`-Methode eine nicht-statische Methode `doSomething()` aufruft,
  - die eine weitere Methode `divideByZeroNoCatch()` aufruft, in der ohne weitere Prüfung durch Null geteilt wird.
- Lösung, Schritt 1: Package und Anwendungsklasse anlegen



# Einführendes Beispiel

- Lösung, Schritt 2: Methode `doSomething()` entwickeln und aufrufen



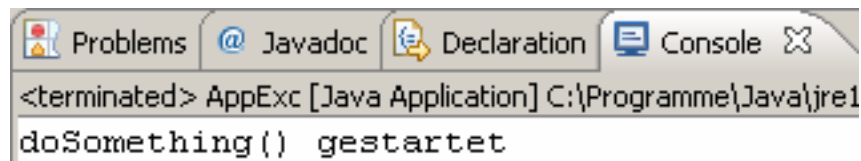
```
AppExc.java X
package de.fh_lu.o2s.exceptions;

public class AppExc {

    public static void main(String[] args) {
        new AppExc().doSomething();
    }

    public void doSomething(){
        System.out.println("doSomething() gestartet");
        System.out.println("doSomething() beendet");
    }
}
```

- Anmerkung:
  - Um vom statischen Kontext der `main()`-Methode in den nicht-statischen Kontext der Methode `doSomething()` zu kommen, müssen wir mit `new AppExc()` ein Anwendungsobjekt erzeugen, vgl. V5: „Vererbung, Teil 2“.



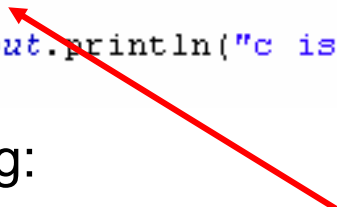
Problems Javadoc Declaration Console X

```
<terminated> AppExc [Java Application] C:\Programme\Java\jre1
doSomething() gestartet
```

# Einführendes Beispiel

- Lösung, Schritt 3: Methode `divideByZeroNoCatch()` entwickeln und aufrufen

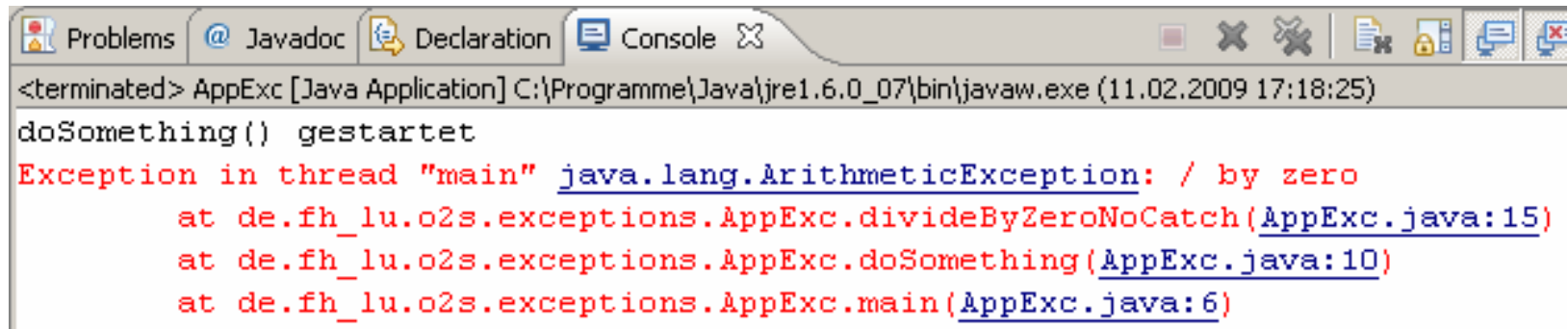
```
public void doSomething() {  
    System.out.println("doSomething() gestartet");  
    this.divideByZeroNoCatch();  
    System.out.println("doSomething() beendet");  
}  
  
public void divideByZeroNoCatch() {  
    int a=1, b=0, c;  
    c = a/b;  
    System.out.println("c ist " + c);  
}
```



- Anmerkung:
  - Zur Laufzeit muss bei `c = a/b;` ein Fehler auftreten.

# Einführendes Beispiel

- Lösung, Schritt 4: AppExc ausführen

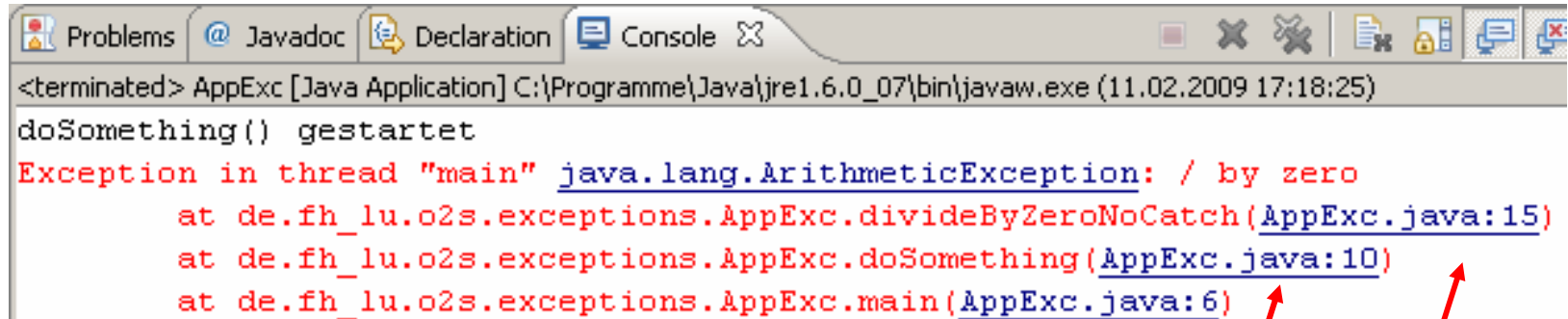


```
<terminated> AppExc [Java Application] C:\Programme\Java\jre1.6.0_07\bin\javaw.exe (11.02.2009 17:18:25)
doSomething() gestartet
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at de.fh_lu.o2s.exceptions.AppExc.divideByZeroNoCatch(AppExc.java:15)
    at de.fh_lu.o2s.exceptions.AppExc.doSomething(AppExc.java:10)
    at de.fh_lu.o2s.exceptions.AppExc.main(AppExc.java:6)
```

- Beobachtungen:
  - Der Fehler wird in einem Exception-Objekt gekapselt.
  - Man könnte sagen, der Fehler ist eine Exception (Ausnahme).
  - In diesem Fall ist es ein Exception-Objekt vom Typ `java.lang.ArithmeticException`
  - Das Objekt kennt die Fehlermeldung (Message): „/ by zero“.
  - Eclipse gibt zunächst „main()“ als Fehlerquelle an,
  - gibt dann aber die **Aufrufkette** aus, die zu dem Fehler geführt hat. Diese ist von unten nach oben zu lesen.



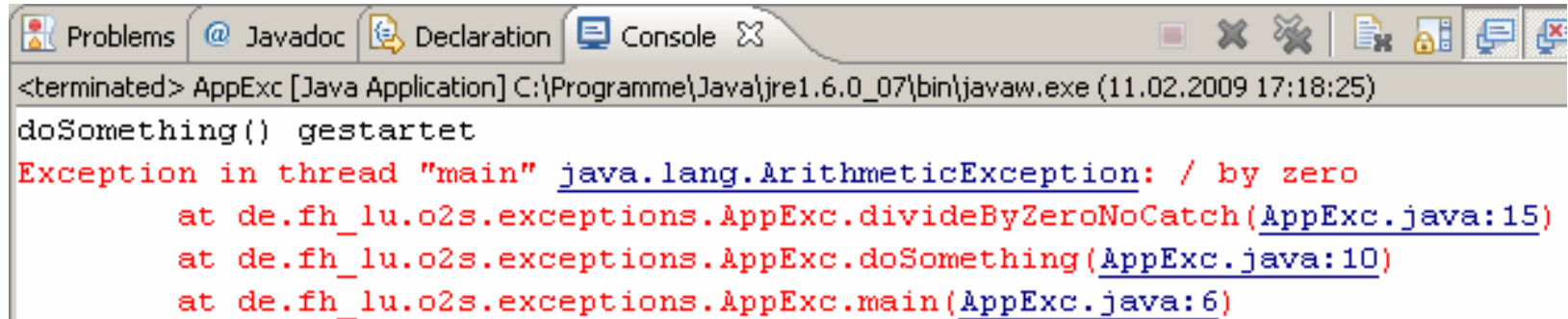
# Aufrufkette



```
<terminated> AppExc [Java Application] C:\Programme\Java\jre1.6.0_07\bin\javaw.exe (11.02.2009 17:18:25)
doSomething() gestartet
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at de.fh_lu.o2s.exceptions.AppExc.divideByZeroNoCatch(AppExc.java:15)
    at de.fh_lu.o2s.exceptions.AppExc.doSomething(AppExc.java:10)
    at de.fh_lu.o2s.exceptions.AppExc.main(AppExc.java:6)
```

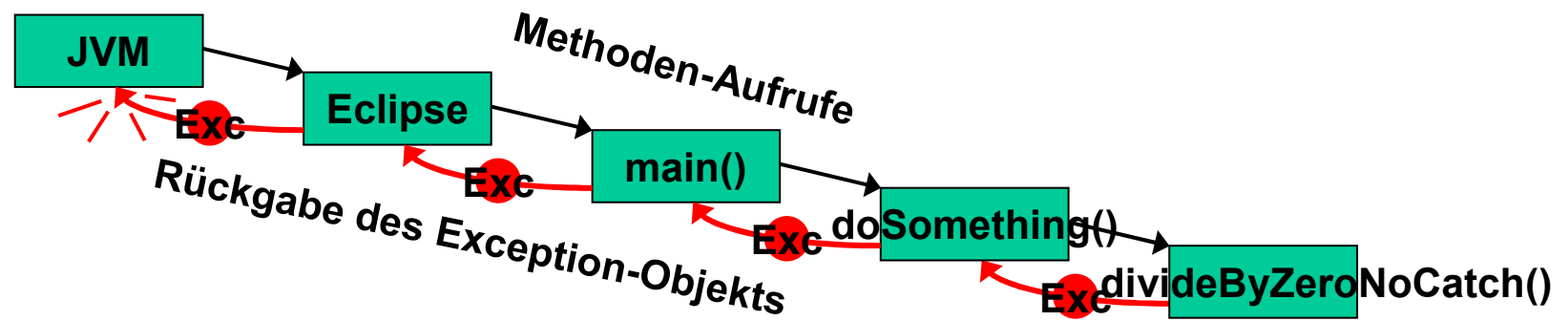
- Die Verarbeitung des Programms
  - beginnt bei der Java Virtual Machine.
  - Diese führt das Eclipse aus.
  - Eclipse startet unser Programm durch Aufruf der `main()`-Methode.
  - `main()` ruft `doSomething()` auf (Zeile 6 in der Datei `AppExc.java`)
  - `doSomething()` ruft `divideByZeroNoCatch()` auf (Zeile 10 in `AppExc.java`)
  - Innerhalb von `divideByZeroNoCatch()` tritt der Fehler auf (Zeile 15 der Datei `AppExc.java`).
  - Die Methoden werden in der Aufrufkette gestapelt (stacked).
  - Das Ende von `doSomething()` wird nicht erreicht.

# Stack Trace



```
<terminated> AppExc [Java Application] C:\Programme\Java\jre1.6.0_07\bin\javaw.exe (11.02.2009 17:18:25)
doSomething() gestartet
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at de.fh_lu.o2s.exceptions.AppExc.divideByZeroNoCatch(AppExc.java:15)
    at de.fh_lu.o2s.exceptions.AppExc.doSomething(AppExc.java:10)
    at de.fh_lu.o2s.exceptions.AppExc.main(AppExc.java:6)
```

- Der Fehler / das Exception-Objekt wird
  - entweder behandelt oder
  - entlang der Aufrufkette an die jeweils vorhergehende Methode weitergegeben, solange bis er behandelt wird.
  - Wenn er in unserem Programm **überhaupt nicht behandelt** wird, geht er weiter bis zum Eclipse oder bis zur Java Virtual Machine.
  - Dies führt zu einem **unkontrollierten Abbruch** des Programms.



# Vorbeugen hilft (auch)

- Unkontrollierte Abbrüche
  - Sollen durch guten Programmierstil verhindert werden.
  - Eine klassische Methode dagegen ist „vorbeugen“.
- Aktion 2:
  - Entwickeln Sie in `AppExc` eine Methode `divideByZeroMitVorbeugen()`,
  - in der Sie vor der Teilung prüfen, dass der Nenner nicht Null ist.
  - Rufen Sie diese Methode aus `doSomething()` auf und nicht mehr `divideByZeroNoCatch()`
- Lösung, Schritt 1: Methode `divideByZeroMitVorbeugen()`

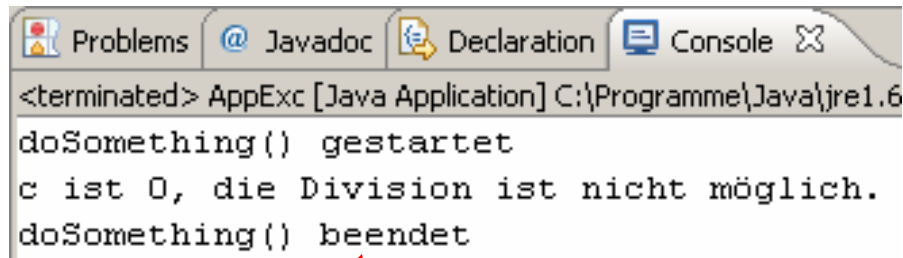
```
public void divideByZeroMitVorbeugen(){
    int a=1, b=0, c;
    if (b != 0){
        c = a/b;
        System.out.println("c ist " + c);
    }
    else{
        System.out.println("c ist 0, die Division ist nicht möglich.");
    }
}
```

# Vorbeugen hilft (auch)

- Lösung, Schritt 2: Aufruf aus `doSomething()`

```
public void doSomething() {  
    System.out.println("doSomething() gestartet");  
    // this.divideByZeroNoCatch();  
    this.divideByZeroMitVorbeugen();  
    System.out.println("doSomething() beendet");  
}
```

- Lösung, Schritt 3: Test durch Ausführung von `AppExc`



```
<terminated> AppExc [Java Application] C:\Programme\Java\jre1.6  
doSomething() gestartet  
c ist 0, die Division ist nicht möglich.  
doSomething() beendet
```

- Anmerkungen:
  - „Vorbeugen“ ist nicht ganz einfach, denn
    - viele Abfragen machen den Code schwieriger lesbar.
    - Man muss vorher ganz genau wissen, was evtl. schiefgehen könnte.
  - Das Ende von `doSomething()` wird erreicht.

# Fehlerbehandlung mit Exceptions

- Man muss
  - nicht mehr genau wissen, **was** passieren kann,
  - es reicht zu wissen, **wo** etwas passieren kann.
- Diese Stelle wird
  - in einen `try`-Block eingeschlossen.
  - Wer „`try`“ sagt, muss auch „`catch`“ sagen.
- Syntax

```
public void demoMethode() {  
    try{  
        <Befehle, wo etwas schiefgehen könnte>  
    } catch (Exception e) {  
        <Fehlerbehandlung>  
    }  
}
```

# divideByZero()

- Aktion 3: Entwickeln Sie
  - eine Methode `divideByZero()`,
  - die von `doSomething()` aufgerufen wird und
  - in der der Code, der durch 0 teilt in einem `try`-Block steht.
- Lösung, Schritt 1: `divideByZero()`
  - erzeugen, z.B. als Kopie von `divideByZeroNoCatch()`,
  - umbenennen und
  - aus `doSomething()` heraus aufrufen:

```
//      this.divideByZeroNoCatch(),  
//      this.divideByZeroMitVorbeugen();  
    this.divideByZero();  
    System.out.println("doSomething() beendet");  
}  
  
public void divideByZero(){  
    int a=1, b=0, c;  
    c = a/b;  
    System.out.println("c ist " + c);  
}
```

# divideByZero()

- Lösung, Schritt 2:

- Den kritischen Code „a/b“ in einen `try`-Block einschließen,
- `catch`-Block hinzufügen.

```
public void divideByZero(){  
    int a=1, b=0, c;  
    try{  
        c = a/b;  
        System.out.println("c ist " + c);  
    } catch (Exception e){  
  
    }  
}
```

- Beobachtungen:

- `try`- und `catch`- leiten Code-Blöcke ein, die – wie in Java üblich – in geschweiften Klammern „{...}“ stehen.
- Der `catch`-Block hat als Parameter – in runden Klammern – ein `Exception`-Objekt mit einem beliebigen Namen, hier: „e“
- Im `catch`-Block ist beliebiger Java-Code zur Fehlerbehandlung möglich (zurzeit noch gar keiner).

# divideByZero()

- Lösung, Schritt 3:
  - Fehlerbehandlung hinzufügen, zunächst nur als einfache Konsol-Ausgabe.

```
public void divideByZero(){  
    int a=1, b=0, c;  
    try{  
        c = a/b;  
        System.out.println("c ist " + c);  
    } catch (Exception e){  
        System.out.println("Exception gefangen.");  
    }  
}
```

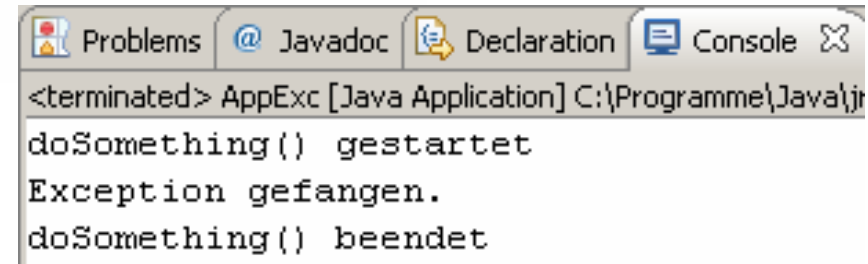
- Lösung, Schritt 4:
  - Testen durch Ausführung von AppExc
  - Ergebnis auf der nächsten Folie



# divideByZero()

```
public void divideByZero() {  
    int a=1, b=0, c;  
    try{  
        c = a/b;  
        System.out.println("c ist " + c);  
    } catch (Exception e) {  
        System.out.println("Exception gefangen.");  
    }  
}
```

- Beobachtungen:
  - doSomething() wird beendet, es gibt also keinen unkontrollierten Abbruch.
  - „c ist ...“ wird nicht ausgegeben, dieser Programmteil wird also nicht erreicht.
- Erklärung: Sobald im try-Block ein Fehler auftritt, wird
  - der try-Block abgebrochen und ein Exception-Objekt erzeugt.
  - Dieses wird als Parameter an den catch-Block übergeben.
  - Der catch-Block wird als ganz normaler Java-Code ausgeführt.
  - Danach geht es hinter dem catch-Block weiter.

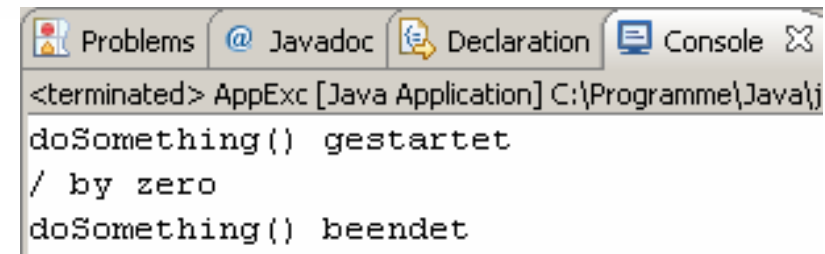


# divideByZero()

- Lösung, Schritt 5: Verbesserung der Fehlerbehandlung
  - durch Ausgabe der Fehlermeldung.
  - Anschließend Test durch Ausführen von AppExc

```
        System.out.println("Exception gefangen");  
    } catch (Exception e) {  
        // System.out.println("Exception gefangen");  
        System.out.println(e.getMessage());  
    }
```

- Beobachtungen:
  - Das Exception-Objekt `e` versteht die Methode `getMessage()` und antwortet darauf mit einer Fehlermeldung.
  - Im catch-Block ist das Exception-Objekt `e` bekannt und kann verwendet werden wie jedes andere Java-Objekt.



# Anmerkungen

- Anmerkungen: Im Fehlerfall
  - wird der try-Block sofort abgebrochen.
  - Befehle, die im try-Block nach dem Fehler kommen, werden nicht mehr ausgeführt.
  - Man sagt, ein Fehler „wirft“ eine Exception.
  - Mit dieser Exception bzw. diesem Exception-Objekt wird der catch-Block ausgeführt.
  - Der Variablenname für das Exception-Objekt ist frei wählbar.
  - Für Aufräumarbeiten ist auch noch ein finally-Block möglich, der auf jeden Fall ausgeführt wird:

```
public void demoMethode() {  
    try{  
        <Befehle, wo etwas schiefgehen könnte>  
    } catch (Exception excep) {  
        <Fehlerbehandlung, z.B. excep.getMessage()>  
    } finally{  
        <Befehle, die auf jeden Fall ausgeführt werden>  
    }  
    <danach gehts ganz normal weiter>  
}
```

# divideByZero()

- Aktion 4: Passen Sie die Fehlerbehandlung in `divideByZero()` so an, dass
  - wie in Aktion 1 der ganze Stack Trace angezeigt wird,
  - das Programm trotzdem nicht unkontrolliert abbricht.

- Lösung, Schritt 1:

- Verwenden der Exception-Methode `printStackTrace()`

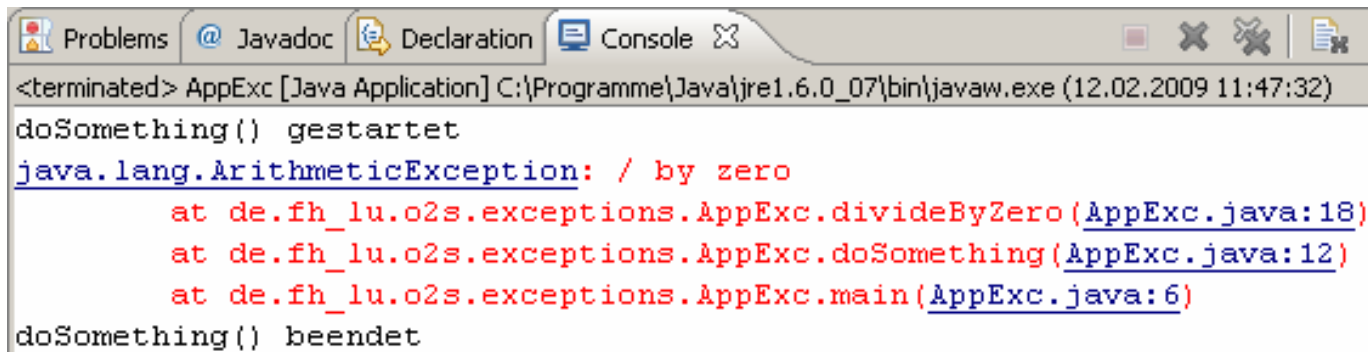
```
        System.out.println("0 180 " + c);  
    } catch (Exception e) {  
        //          System.out.println("Exception gefangen");  
        //          System.out.println(e.getMessage());  
        e.printStackTrace();  
    }
```

- Beobachtung:

- Die Ausgabe des Stack Trace ist eine Fähigkeit des Exception-Objekts.
- `System.out (...)` muss hier nicht hingeschrieben werden.

# divideByZero()

- Lösung, Schritt 2:
  - Test durch Ausführung von AppExc



```
<terminated> AppExc [Java Application] C:\Programme\Java\jre1.6.0_07\bin\javaw.exe (12.02.2009 11:47:32)
doSomething() gestartet
java.lang.ArithmeticException: / by zero
    at de.fh_lu.o2s.exceptions.AppExc.divideByZero(AppExc.java:18)
    at de.fh_lu.o2s.exceptions.AppExc.doSomething(AppExc.java:12)
    at de.fh_lu.o2s.exceptions.AppExc.main(AppExc.java:6)
doSomething() beendet
```

- Beobachtung:
  - Das Programm wurde kontrolliert beendet, obwohl es genauso aussieht wie in Aktion 1.
  - Der Stack Trace enthält u.a. auch den Exception-Typ und die Fehlermeldung (Error-Message).
- Anmerkung: Die Ausgabe des Stack Trace dient vor allem zu Testzwecken.

# Exception-Objekte

Methoden
getMessage()
getCause()
toString()
printStackTrace()
getStackTrace()

`java.lang`

## Class Exception

`java.lang.Object`

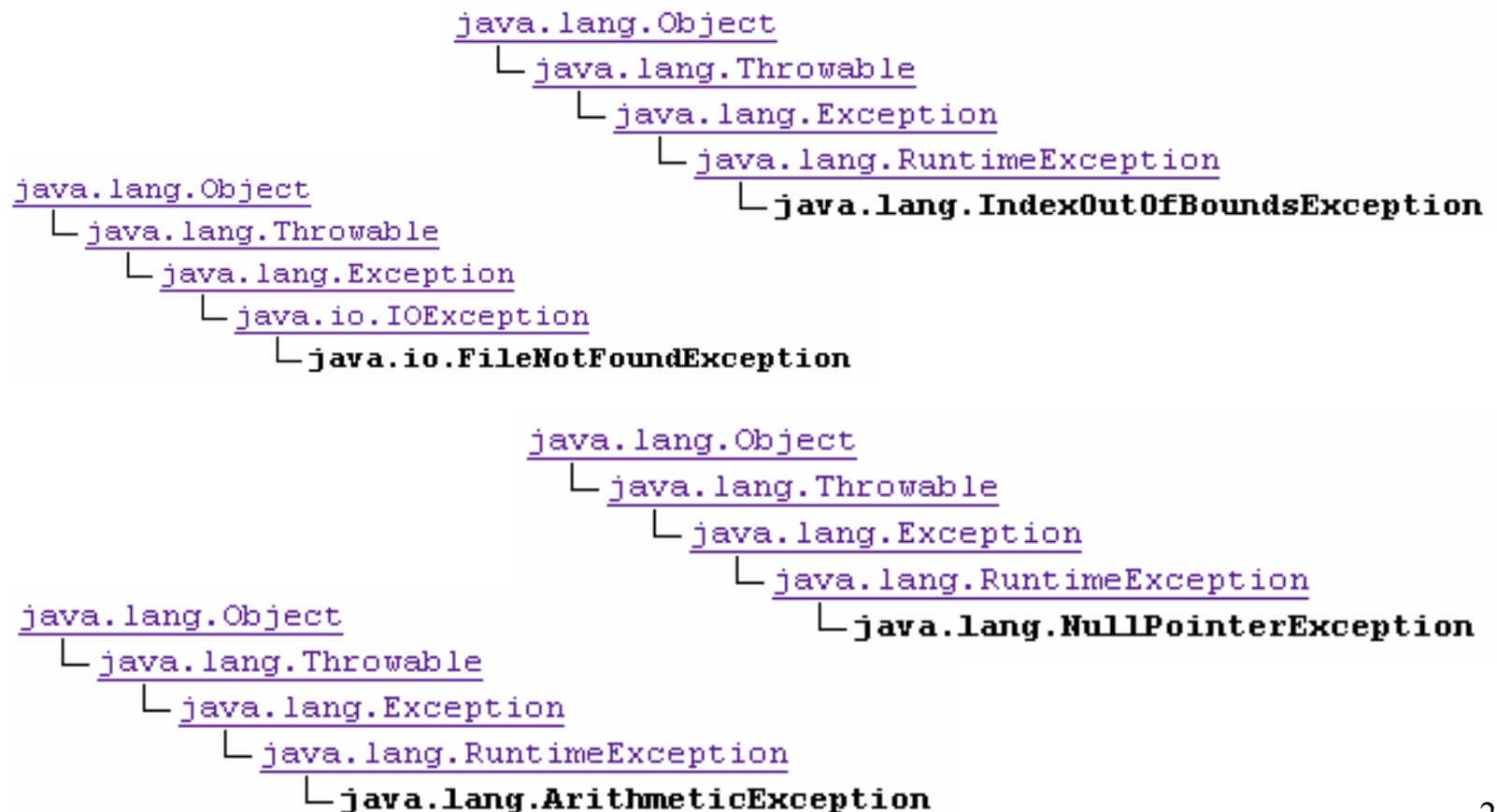
└ `java.lang.Throwable`

└ `java.lang.Exception`

- Jedes Exception-Objekt kann haben:
  - eine Message,
  - eine Beschreibung,
  - ein für uns weniger wichtiges technisches Objekt „cause“.
- Jedes Exception-Objekt kann sagen, wie es dazu gekommen ist (Aufrufkette).

# Typen von Exceptions

- Es gibt viele Exception-Klassen, um unterschiedliche Fehler auseinanderhalten zu können, z.B.



# Typen von Exceptions

- Das oberste Unterscheidungsmerkmal von Exceptions ist
  - die Klassenzugehörigkeit,
  - erst danach kommt die Message.
- Neue Fehlerarten werden
  - durch neue Exceptionklassen implementiert.
  - Dazu wird von bestehenden Exceptions geerbt.
  - Spezialisierung / Erweiterung
- Eine besondere Unterscheidung besteht zwischen
  - `RuntimeException` und den Unterklassen davon, für die also gilt „`SpezielleException` is-a `RuntimeException`“
  - Exceptions, die keine Unterklassen von `RuntimeException` sind.
  - Vgl. vorherige Folie



# Hierarchien von Exceptions

- Anmerkungen:
  - Die Art eines Fehlers wird am Typ der Exception erkannt
  - Neue Exception-Klassen können
    - bestehende Methoden überschreiben oder
    - zusätzliche Methoden oder Konstruktoren mitbringen
  - Und dadurch z.B.
    - zusätzliche Informationen liefern (vgl. Beispiel) oder
    - Fehlermeldungen anpassen
  - Beispiel

```
class SQLException extends Exception{  
    public String getSQLState{  
        return <SQL-Status-Kennzeichen>;  
    }  
}
```

# Unterschiedliche Exceptions

- Vorbereitung von Aktion 5: Kopieren Sie
  - die Datei `sevExcFrag.txt` aus dem Materialfile in Ihr Package.
  - Legen Sie die nachstehende – zurzeit fehlerhafte – Methode `severalExceptions()` in `AppExc` an, z.B. indem Sie sie aus der Datei `sevExcFrag.txt` kopieren und
  - rufen Sie `severalExceptions()` aus Ihrer `doSomething()` auf.

```
// this.divideByZero();  
    this.severalExceptions();  
    System.out.println("doSomething() beendet");  
}  
  
public void severalExceptions(){  
    int a=1, b=1, c;  
    //ArithmeticException möglich  
    c=a/b;  
    System.out.println("c ist " + c);  
    java.io.RandomAccessFile f;  
    //FileNotFoundException möglich  
    f = new java.io.RandomAccessFile(  
        "c:\\windows\\desktop.ini", "r");  
    String line;  
    //IOException möglich  
    while ( (line = f.readLine()) != null ){  
        System.out.println( line );  
    }  
    f.close();  
}
```

# Unterschiedliche Exceptions

```
//ArithmeticException möglich
c=a/b;
System.out.println("c ist " + c);
java.io.RandomAccessFile f;
//FileNotFoundException möglich
Unhandled exception type FileNotFoundException
        accessFile(
            "c:\\windows\\desktop.ini", "r" );
String line;
```

- Beobachtung: Eclipse merkt,
  - dass bei dem Befehl `new RandomAccessFile(...)` eine `FileNotFoundException` auftreten kann und zeigt deshalb einen Compiler-Fehler an.
  - Eclipse merkt aber nicht, dass bei „a/b“ ein Fehler auftreten kann.
- Unterschied:
  - `FileNotFoundException` ist keine `RuntimeException`.
  - Bei „a/b“ handelt es sich um eine `RuntimeException`.
  - Mögliche `RuntimeException`s werden nicht vom Compiler angemerkt.

# Fehler vermeiden

- Jede Fehlermöglichkeit, wo eine Nicht-`RuntimeException` auftreten kann,
  - **muss** im Programm behandelt werden,
  - z.B. mit `try` und `catch`,
  - Andernfalls beschwert sich der Compiler.
- Jede Möglichkeit, wo eine `RuntimeException` auftreten kann,
  - **sollte** entweder vorab bereits ausgeschlossen werden oder
  - analog zu den Nicht-`RuntimeExceptions` behandelt werden.
- Beispiel:
  - `IndexOutOfBoundsException` ist eine `RuntimeException`, die durch saubere Programmierung von Array-Zugriffen vermieden werden muss.
  - `ClassCastException` ist eine `RuntimeException`, die z.B. durch eine vorherige `instanceof`-Abfrage vermieden werden kann.

# Unterschiedliche Exceptions

- Aktion 5: Ermitteln Sie,
  - welche Exception-Typen in der Methode `severalExceptions()` auftreten können,
  - packen Sie den gesamten Code in einen einzigen `try`-Block und
  - fügen Sie zunächst nur einen einzigen `catch`-Block hinzu, in dem Sie den Stack Trace ausgeben lassen.
- Lösung, Schritt 1:
  - Eclipse zeigt uns die Compiler-Fehler an.
  - Die mögliche `ArithmeticException` (is-a `RuntimeException`) in „a/b“ müssen wir selbst wissen.

```
f = new java.io.RandomAccessFile(  
    "c:\\windows\\desktop.ini", "r" );  
String line;  
while ( (line = f.readLine()) != null ) {  
    System.out.println( line );  
}  
f.close();
```

Unhandled exception type FileNotFoundException

Unhandled exception type IOException

Unhandled exception type IOException

# Unterschiedliche Exceptions

- Lösung, Schritt 2:

```
public void severalExceptions(){  
    try{  
        int a=1, b=1, c;  
        //ArithmeticException möglich  
        c=a/b;  
        System.out.println("c ist " + c);  
        java.io.RandomAccessFile f;  
        //FileNotFoundException möglich  
        f = new java.io.RandomAccessFile(  
            "c:\\windows\\desktop.ini", "r" );  
        String line;  
        //IOException möglich  
        while ( (line = f.readLine()) != null ){  
            System.out.println( line );  
        }  
        f.close();  
    } catch (Exception e){  
        e.printStackTrace();  
    }  
}
```

- Beobachtung: Alle Fehler werden kontrolliert.

# Unterschiedliche Exceptions

- Aktion 6: Erweitern Sie,
  - Ihre Methode `severalExceptions()`,
  - Indem Sie für jeden möglichen Exception Typ einen eigenen `catch`-Block anlegen.
- Lösung, Schritt 1:
  - Anstatt des allgemeinen `catch`-Blocks `catch(Exception e){...}` schreiben wir zunächst `catch(ArithmeticException ae){...}`

```
        System.out.println( line );
    }
    f.close();
} catch (ArithmeticException ae){
    System.out.println("ArithmeticException gefangen");
    ae.printStackTrace();
}
```

Unhandled exception type IOException

- Beobachtung:
  - Die `IOException` wird noch nicht behandelt,
  - weil `IOExceptions` keine `ArithmeticExceptions` sind.

# Unterschiedliche Exceptions

- Lösung, Schritt 2:
  - Wir legen einen zusätzlichen catch-Block

catch(IOException ioe) {...} an und ...

```
f.close();
} catch (IOException ioe) {
    System.out.println("IOException gefangen");
    ioe.printStackTrace();
} catch (ArithmeticException ae) {
    System.out.println("ArithmeticException gefangen");
    ae.printStackTrace();
}
```

IOException cannot be resolved to a type

... importieren die Klasse java.io.IOException

- Beobachtung:

- Dann treten  
zunächst  
keine Fehler  
mehr auf

```
//FileNotFoundException möglich
f = new java.io.RandomAccessFile(
    "c:\\windows\\desktop.ini", "r" );
String line;
//IOException möglich
while ( (line = f.readLine()) != null ) {
    System.out.println( line );
}
f.close();
} catch (IOException ioe) {
    System.out.println("IOException gefangen");
    ioe.printStackTrace();
} catch (ArithmeticException ae) {
```



# Unterschiedliche Exceptions

- Warum beschwert sich der Compiler nicht,
  - obwohl die `FileNotFoundException` nicht ausdrücklich behandelt wurde?
- Antwort:
  - Wegen „`FileNotFoundException` is-a `IOException`“ wurde die `FileNotFoundException` bereits vom `catch-Block` der `IOException` behandelt.

# Unterschiedliche Exceptions

- Lösung, Schritt 3: Wir wollen trotzdem einen zusätzlichen catch-Block für die FileNotFoundException anlegen.

```
l.close();  
} catch (IOException ioe) {  
    System.out.println("IOException gefangen");  
} catch (FileNotFoundException ioe) {  
    System.out.println("FileNotFoundException gefangen");  
    ioe.printStackTrace();  
} catch (ArithmeticException ae) {
```

Unreachable catch block for FileNotFoundException. It is already handled by the catch block for IOException

- Beobachtung:
  - Obwohl wir FileNotFoundException importiert haben, gibt es noch einen Fehler, weil
  - die FileNotFoundException bereits im früheren catch-Block der IOException behandelt wurde.

- Lösung, Schritt 4:
  - Der `catch`-Block für die `FileNotFoundException` muss vor dem `catch`-Block für die `IOException` stehen.

```
l.close();  
} catch (FileNotFoundException fnfe) {  
    System.out.println("FileNotFoundException gefangen");  
    fnfe.printStackTrace();  
} catch (IOException ioe) {  
    System.out.println("IOException gefangen");  
    ioe.printStackTrace();  
} catch (ArithmeticException ae) {  
    System.out.println("ArithmeticException gefangen");  
    ae.printStackTrace();  
}
```

# Unterschiedliche Exceptions

- Lösung, Schritt 5:
  - Wir fügen noch einen `catch`-Block für allgemeine `Exceptions` hinzu, für den Fall, dass eine unerwartete `Exception` auftritt.
  - Dieser Block muss natürlich am Ende der `catch`-Blöcke stehen.

```
try {  
    // ...  
} catch (FileNotFoundException fnfe) {  
    System.out.println("FileNotFoundException gefangen");  
    fnfe.printStackTrace();  
} catch (IOException ioe) {  
    System.out.println("IOException gefangen");  
    ioe.printStackTrace();  
} catch (ArithmeticException ae) {  
    System.out.println("ArithmeticException gefangen");  
    ae.printStackTrace();  
} catch (Exception e) {  
    System.out.println("Unerwartete Exception gefangen");  
    e.printStackTrace();  
}
```

# Unterschiedliche Exceptions

- Zusammenfassung:
  - Wenn ein Fehler mit Exception  $e$  auftritt,
  - wird der **erste** catch-Block (von oben) aktiviert, dessen angegebener Exception-Typ  $T$  „passt“.
  - „passt“ bedeutet: Die aufgetretene Exception  $e$  „is a“  $T$ , d.h.  $(e \text{ instanceof } T)$  ist wahr.
  - Im Beispiel: Eine `FileNotFoundException` passt zu jedem der folgenden Blöcke:

```
catch(java.io.FileNotFoundException fnfe){...}  
catch(java.io.IOException ioe){...}  
catch(Exception e){...}
```

```
java.lang.Object  
└─ java.lang.Throwable  
    └─ java.lang.Exception  
        └─ java.io.IOException  
            └─ java.io.FileNotFoundException
```

# Unterschiedliche Exceptions

- Zusammenfassung:
  - Die `catch`-Blöcke müssen immer in der Reihenfolge geschrieben werden, dass die speziellere Exception vor der allgemeineren Exception gefangen wird.

```
try {  
    // ...  
} catch (FileNotFoundException fnfe) {  
    System.out.println("FileNotFoundException gefangen");  
    fnfe.printStackTrace();  
} catch (IOException ioe) {  
    System.out.println("IOException gefangen");  
    ioe.printStackTrace();  
} catch (ArithmeticException ae) {  
    System.out.println("ArithmeticException gefangen");  
    ae.printStackTrace();  
} catch (Exception e) {  
    System.out.println("Unerwartete Exception gefangen");  
    e.printStackTrace();  
}
```

java.lang.Object

└ java.lang.Throwable

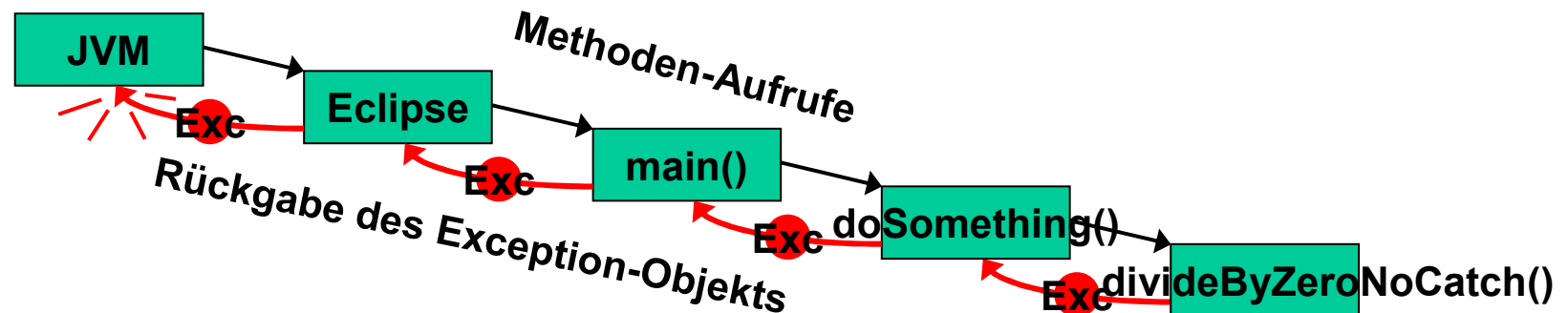
└ java.lang.Exception

└ java.io.IOException

└ **java.io.FileNotFoundException**

# Exceptions werfen

- In Aktion 1 wurde eine `ArithmeticException` geworfen
  - Nicht bis zu einem `catch`-Block derselben Methode,
  - sondern bis zum Eclipse.



- Beobachtung:
  - Das ist passiert, weil es sich um eine `RuntimeException` handelt,
  - die nicht gefangen wurde.
- Anmerkung:
  - Das geht auch mit Nicht-`Runtime`-Exceptions, allerdings muss dies dem Compiler dann ausdrücklich mitgeteilt werden,
  - weil ja andernfalls ein Compiler-Fehler auftritt.

# Exceptions werfen

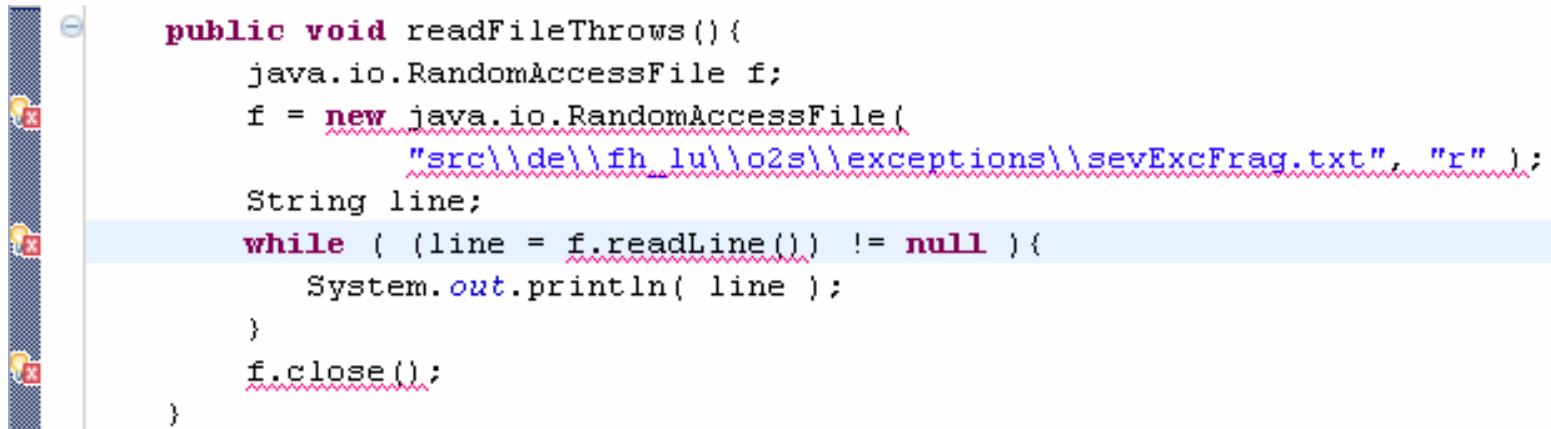
- Aktion 7: Entwickeln Sie
  - eine Methode `readFileThrows()`, die
  - die Datei „sevExcFrag.txt“ auf der Konsole ausgibt.
  - Werfen Sie eventuelle Exceptions an die aufrufende Methode.
  - Rufen Sie `readFileThrows()` von `doSomething()` aus auf und fangen Sie dort die eventuell geworfenen Exceptions.
- Lösung, Schritt 1:
  - Erzeugen Sie die Methode `readFileThrows()` und
  - Rufen Sie diese aus `doSomething()` auf.

```
        this.sevExcFrag.readFileThrows();  
        this.readFileThrows();  
        System.out.println("doSomething() beendet");  
    }  
    public void readFileThrows(){  
  
    }
```



# Exceptions werfen

- Lösung, Schritt 2:
  - Kopieren Sie den Code aus `severalExceptions()`,
  - entfernen Sie sämtliche `try`- und `catch`-Elemente und
  - passen Sie den Code auf die folgende Datei an:  
"src\\de\\fh\_lu\\o2s\\exceptions\\sevExcFrag.txt,"



```
public void readFileThrows(){
    java.io.RandomAccessFile f;
    f = new java.io.RandomAccessFile(
        "src\\de\\fh_lu\\o2s\\exceptions\\sevExcFrag.txt", "r" );
    String line;
    while ( (line = f.readLine()) != null ){
        System.out.println( line );
    }
    f.close();
}
```

- Beobachtung:
  - Der Compiler beschwert sich über die möglichen Nicht-  
Runtime-Exceptions.

# Exceptions werfen

- Lösung, Schritt 3:
  - Fügen Sie im Kopf der Methoden `readFileThrows()` die Klausel „throws Exception“ ein.

```
// this.severalExceptions();
this.readFileThrows();
System.out.println("doSomething() beendet");
}

public void readFileThrows() throws Exception {
    java.io.RandomAccessFile f;
    f = new java.io.RandomAccessFile(
        "src\\de\\fh_lu\\o2s\\exceptions\\sevExcFrag.txt", "r" );
    String line;
    while ( (line = f.readLine()) != null ){
        System.out.println( line );
    }
    f.close();
}
```

- Beobachtung:
  - Jetzt wird die Exception in die aufrufende Methode `doSomething()` geworfen,
  - wo ein Fehler angemerkt wird, weil die Exception bisher noch nicht behandelt wird.

# Exceptions werfen

- Anmerkung:

- Wir hätten anstatt `public void readFileThrows() throws Exception {`  
auch `public void readFileThrows() throws IOException {`  
oder

`public void readFileThrows() throws FileNotFoundException, IOException {`  
schreiben können.

- Wichtig ist nur, dass alle eventuell auftretenden Typen von Nicht-Runtime-Exceptions aufgeführt sind.

- Anmerkung:

- Die aufrufende Methode `doSomething()` steht jetzt in der Pflicht,
  - diese Exceptions zu behandeln oder
  - Sie ebenfalls mit `throws` weiterzuwerfen (an `main()`)
- Eclipse bietet QuickFix für beide Varianten eine Hilfestellung.

```
this.readFileThrows();
```

- ! Add throws declaration
- ! Surround with try/catch

# Exceptions werfen

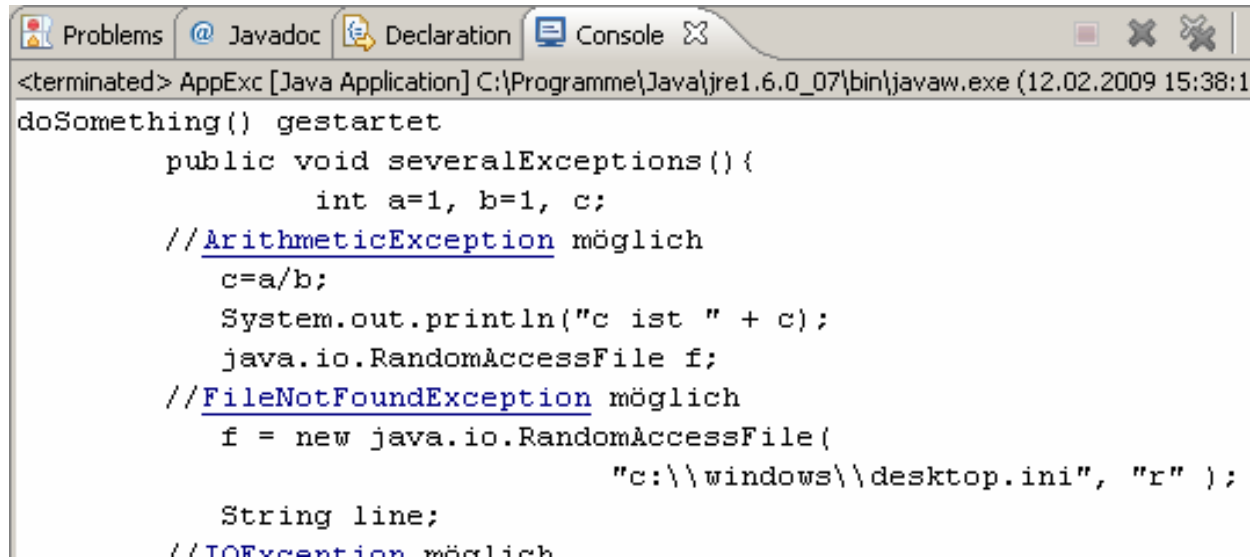
- Lösung, Schritt 4:
  - Behandeln Sie die Exception in `doSomething()`, indem Sie den Methodenaufruf `this.readFileThrows()` in try-/catch-verpacken.

```
this.readFileThrows();  
try {  
    this.readFileThrows();  
} catch (Exception e) {  
    e.printStackTrace();  
}  
System.out.println("doSomething() beendet");  
}  
  
public void readFileThrows() throws Exception {  
    // ...  
}
```

- Anmerkung:
  - In der aufrufenden Methode muss jeder Exception-Typ behandelt werden, der in der `throws`-Klausel angegeben ist.

# Exceptions werfen

- Lösung, Schritt 5: Test durch Ausführung von AppExc



```
<terminated> AppExc [Java Application] C:\Programme\Java\jre1.6.0_07\bin\javaw.exe (12.02.2009 15:38:1
doSomething() gestartet
    public void severalExceptions(){
        int a=1, b=1, c;
        //ArithmeticException möglich
        c=a/b;
        System.out.println("c ist " + c);
        java.io.RandomAccessFile f;
        //FileNotFoundException möglich
        f = new java.io.RandomAccessFile(
                                "c:\\windows\\desktop.ini", "r" );

        String line;
        //IOException möglich
```

# Zusammenfassung

- Exceptions kann man behandeln, entweder indem man sie
  1. in `try-/catch-` verpackt und selbst eine Fehlerbehandlung implementiert oder indem man sie
  2. mit „`throws ...Exception`“ an die aufrufende Methode weiterwirft.
    - Wenn man sie an die aufrufende Methode wirft, muss **diese** sie behandeln (nach 1. oder 2.)
    - Wenn man bei `main()` angekommen ist, führt ein weiterwerfen zu einem unkontrollierten Abbruch des Programms.
- Nicht-Runtime-Exceptions
  - **müssen** zwingend behandelt werden (nach 1. oder 2.), sonst gibt es einen Compiler-Fehler.
- RuntimeExceptions (und ihre Untertypen)
  - **können** wie alle Exceptions mit 1. oder 2. explizit behandelt werden, müssen aber nicht.
  - Wenn Sie nicht explizit behandelt werden und trotzdem auftreten, dann werden sie automatisch geworfen (nach 2.), obwohl es nicht im Programmcode steht.

# Erzeugen von Exceptions

- Aktion 8: Entwickeln Sie eine Methode `createAndThrow()`,
  - die Sie aus `doSomething()` heraus aufrufen und in der Sie
  - ein neues `Exception`-Objekt mit der Fehlermeldung „allgemeiner Fehler“ erzeugen und einer Variablen zuweisen,
  - ein neues `ArithmeticException`-Objekt mit der Fehlermeldung „falsch gerechnet“ erzeugen und einer Variablen zuweisen.
  - Wenn `Math.random() < 0.5` werfen Sie Ihr `Exception`-Objekt, andernfalls Ihr `ArithmeticException`-Objekt.
  - Nutzen Sie die `try-/catch`-Technik, um Ihr `ArithmeticException`-Objekt ggfs. bereits in der Methode `createAndThrow()` wieder zu fangen und die Fehlermeldung auszugeben.
  - Nutzen Sie die `throws`-Klausel, um Ihr `Exception`-Objekt ggfs. an die aufrufende Methode weiterzuwerfen und behandeln Sie diese dort, indem Sie einen Stack Trace ausgeben.

# Erzeugen von Exceptions

- Lösung, Schritt 1:

- `createAndThrow()` anlegen und aus `doSomething()` heraus aufrufen.
- `Exception`- und `ArithmeticException`-Objekt erzeugen

```
//  
    }  
    this.createAndThrow();  
    System.out.println("doSomething() beendet");  
}  
public void createAndThrow(){  
    Exception exc = new Exception("allgemeiner Fehler");  
    ArithmeticException arExc = new ArithmeticException("falsch gerechnet");  
}
```

- Beobachtung:

- `Exception`-Objekte kann man, analog zu allen anderen Objekten, mit `new Exception()` erzeugen.
- Beim erzeugenden Konstruktoraufruf kann man eine Fehlermeldung angeben.



# Erzeugen von Exceptions

- Lösung, Schritt 2:
  - Je nach Zufall werfen des einen oder des anderen Objekts.

```
public void createAndThrow() {  
    Exception exc = new Exception("allgemeiner Fehler");  
    ArithmeticException arExc = new ArithmeticException("falsch gerechnet");  
    if (Math.random() < 0.5) throw exc;  
    else throw arExc;  
}
```

Unhandled exception type Exception

- Beobachtung:
  - Seither wurde das Werfen von Exception-Objekten immer von Java-Standard-Methoden übernommen.
  - Dieses Mal werfen wir sie selbst und zwar mit „throw ...“.
  - Der Compiler beschwert sich, dass eine `Exception` geworfen aber nicht behandelt wird.
  - Über die `ArithmeticException` beschwert er sich nicht, weil es eine `RuntimeException` ist.

# Erzeugen von Exceptions

- Lösung, Schritt 3:
  - Fangen und behandeln der `ArithmeticException`.
  - Weiterwerfen der allgemeinen `Exception`

```
this.createAndThrow();  
System.out.println("doSomething() beendet");  
}  
  
public void createAndThrow() throws Exception{  
    try{  
        Exception exc = new Exception("allgemeiner Fehler");  
        ArithmeticException arExc = new ArithmeticException("falsch gerechnet");  
        if (Math.random() < 0.5) throw exc;  
        else throw arExc;  
    } catch (ArithmeticException ae){  
        System.out.println(ae.getMessage());  
    }  
}
```

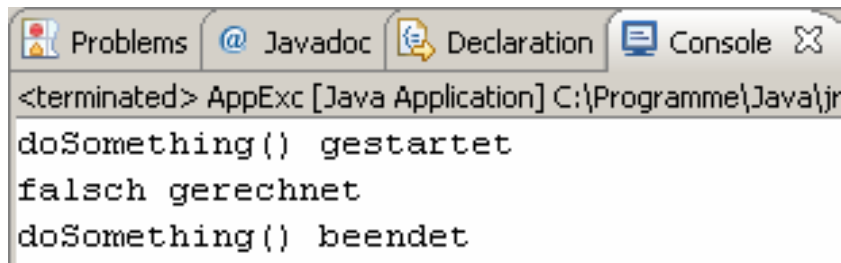
- Beobachtung:
  - Der Compiler beschwert sich jetzt in `doSomething()`, dass dort eine allgemeine `Exception` ankommt aber nicht behandelt wird.

# Erzeugen von Exceptions

- Lösung, Schritt 4:
  - Behandeln der allgemeinen Exception in `doSomething()` und Test durch Ausführung von `AppExc`

```
try {  
    this.createAndThrow();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

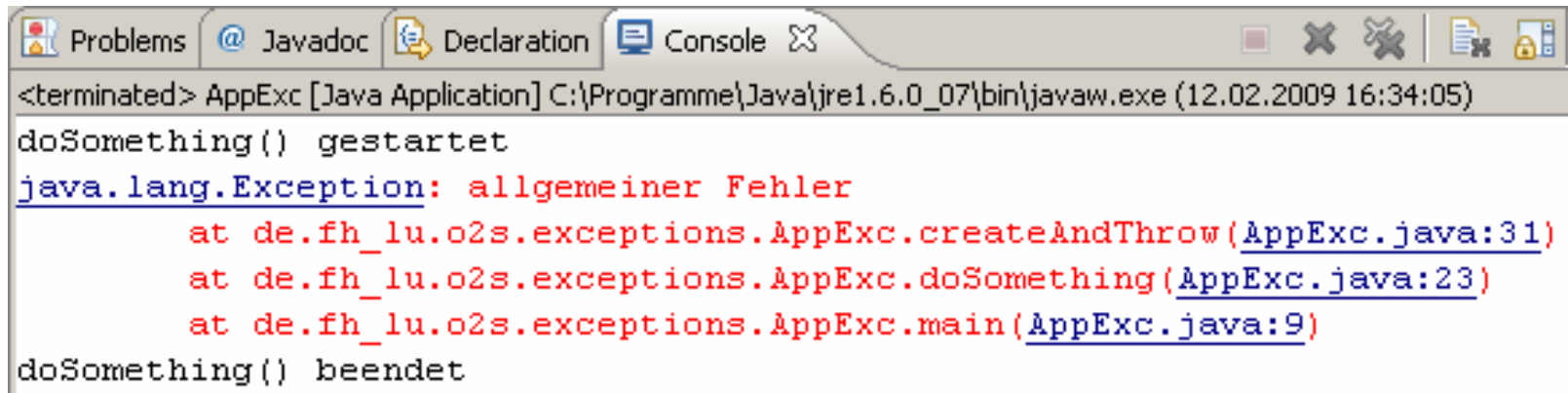
**Math.random() >= 0.5**



Problems Javadoc Declaration Console

<terminated> AppExc [Java Application] C:\Programme\Java\jr  
doSomething() gestartet  
falsch gerechnet  
doSomething() beendet

**Math.random() < 0.5**



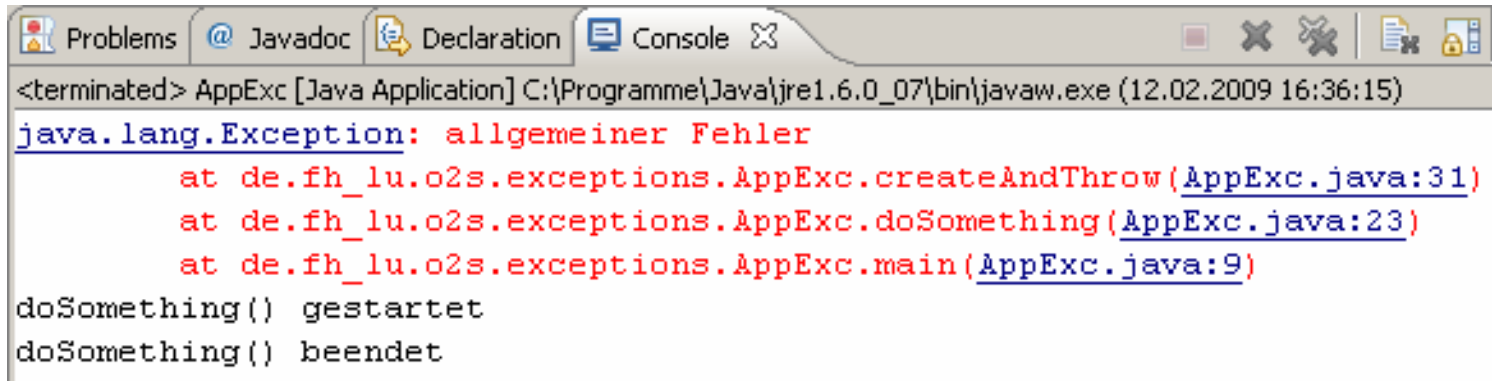
Problems Javadoc Declaration Console

<terminated> AppExc [Java Application] C:\Programme\Java\jre1.6.0\_07\bin\javaw.exe (12.02.2009 16:34:05)

doSomething() gestartet  
**java.lang.Exception: allgemeiner Fehler**  
    at de.fh\_lu.o2s.exceptions.AppExc.createAndThrow(AppExc.java:31)  
    at de.fh\_lu.o2s.exceptions.AppExc.doSomething(AppExc.java:23)  
    at de.fh\_lu.o2s.exceptions.AppExc.main(AppExc.java:9)  
doSomething() beendet

# Ausgabe

- Beobachtung:
  - Bei der Ausführung von AppExc nach Fertigstellung von Aktion 8 tritt manchmal eine Ausgabe auf wie folgt:



```
<terminated> AppExc [Java Application] C:\Programme\Java\jre1.6.0_07\bin\javaw.exe (12.02.2009 16:36:15)
java.lang.Exception: allgemeiner Fehler
    at de.fh_lu.o2s.exceptions.AppExc.createAndThrow(AppExc.java:31)
    at de.fh_lu.o2s.exceptions.AppExc.doSomething(AppExc.java:23)
    at de.fh_lu.o2s.exceptions.AppExc.main(AppExc.java:9)
doSomething() gestartet
doSomething() beendet
```

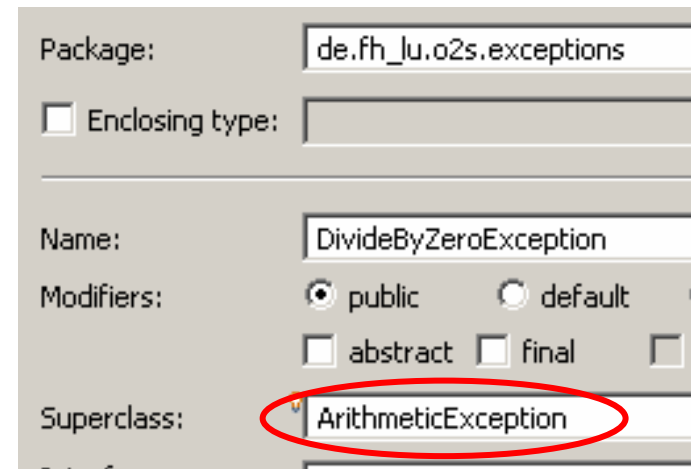
- Folgerung:
  - Im Zusammenhang mit `printStackTrace()` kann man sich nicht auf die Reihenfolge der Ausgabe verlassen, denn
  - Die Ausgabe des Programms und die Ausgabe der Exception erfolgen in verschiedenen Threads.
  - Damit kann es passieren, dass diese sich gegenseitig überholen.

# Eigene Exceptions

- Hinweis:
  - Java-Entwickler dürfen eigene Exceptions anlegen, das heißt konkret: Unterklassen vorhandener Exception-Klassen erzeugen.
- Gründe / Vorteile:
  - Es können sprechendere Exception-Namen gewählt werden, die den Code lesbarer machen.
  - Spezielle Fehlersituationen können durch eigene Exception-Typen individuell behandelt werden.
  - Es können eigene Fehlermeldungen definiert werden,
    - Z.B. im Konstruktor des neuen Exception-Typs oder
    - Durch überschreiben von getMessage()
  - Dem Exception-Typ können weitere Methoden hinzugefügt werden, die ihn ggfs.noch aussagekräftiger machen können.

# Eigene Exceptions

- Aktion 9: Legen Sie
  - im Package `de.fh_lu.o2s.exceptions` eine eigene Exception-Klasse `DivideByZeroException` an.
  - Machen Sie diese zu einer Unterklasse von `ArithmeticException`.
  - Definieren Sie als Standard-Fehlermeldung: „Da hat doch echt wieder so ein Hanswurst versucht, durch Null zu teilen“.
- Lösung, Schritt 1: Klasse im richtigen Package mit der richtigen Oberklasse anlegen



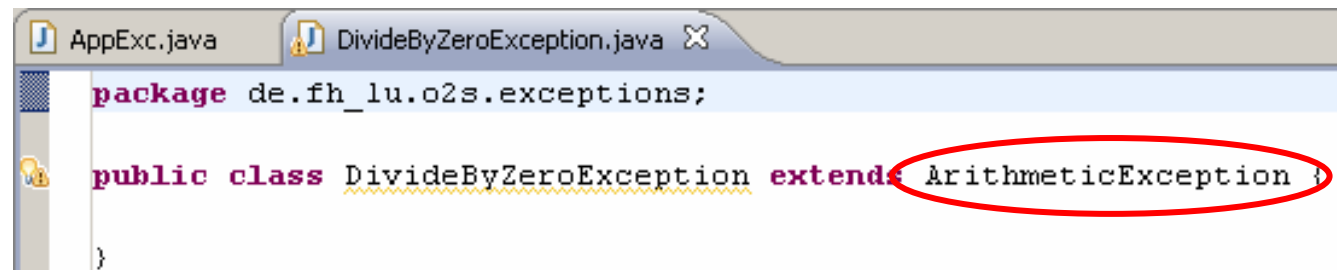
Package: `de.fh_lu.o2s.exceptions`

☐ Enclosing type:

Name: `DivideByZeroException`

Modifiers: ☒ public ☐ default ☐ abstract ☐ final

Superclass: `ArithmeticException`



```
package de.fh_lu.o2s.exceptions;

public class DivideByZeroException extends ArithmeticException {
}
```

# Eigene Exceptions

- Lösung, Schritt 2: Vergabe der richtigen Fehlermeldung
  - im Konstruktor der neuen Exception-Klasse

Standard-  
lösung

```
public class DivideByZeroException extends ArithmeticException {  
    public DivideByZeroException(){  
        super("Da hat doch echt wieder so ein Hanswurst versucht, durch Null zu teilen");  
    }  
}
```

- Anmerkung:
  - Wenn wir (wie die Standard-Exception-Klassen) auch einen Konstruktor liefern wollen, mit dem man die Fehlermeldung bei der Erzeugung festlegen kann, dann mit folgendem zusätzlichen Konstruktor:

zusätzlich  
sinnvoll

```
public DivideByZeroException(String msg){  
    super(msg);  
}
```

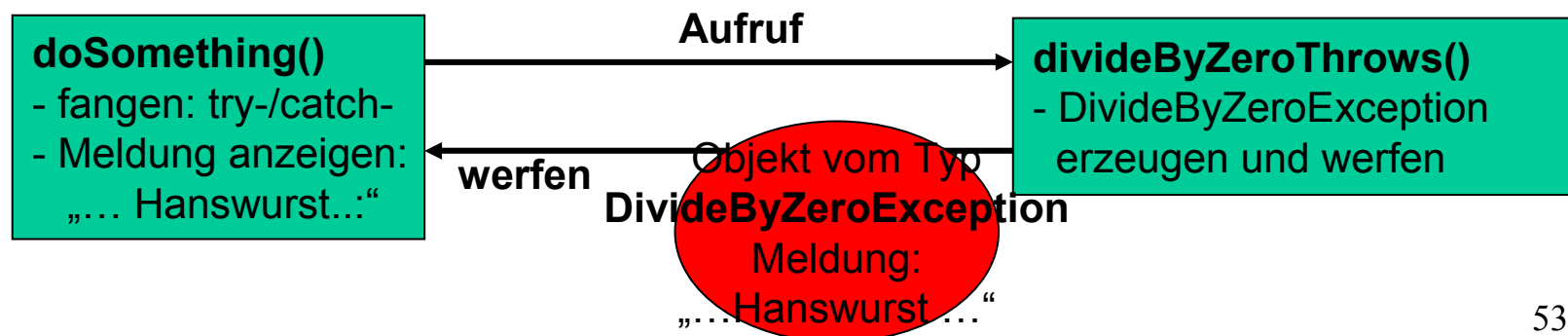
- Alternativ (aber nicht so schön) hätten wir auch `getMessage()` überschreiben können:

```
public String getMessage(){  
    return "Da hat doch echt wieder so ein " +  
        "Hanswurst versucht, durch Null zu teilen!";  
}
```

möglich  
aber weniger  
sinnvoll

# Eigene Exceptions

- **Aktion 10: Entwickeln Sie**
  - eine Methode `divideByZeroThrows()`,
  - die von `doSomething()` aufgerufen wird.
  - Kopieren Sie dazu den Inhalt der Methode `divideByZeroMitVorbeugen()`.
  - Wenn Sie nicht teilen können (weil `b == 0` ist), dann (im `else`-Block) erzeugen Sie ein `DivideByZeroException`-Objekt und werfen Sie dieses an die aufrufende Methode.
  - Passen Sie den Aufruf innerhalb `doSomething()` so an, dass die Fehlermeldung von `DivideByZeroException` ausgegeben wird.





# Eigene Exceptions

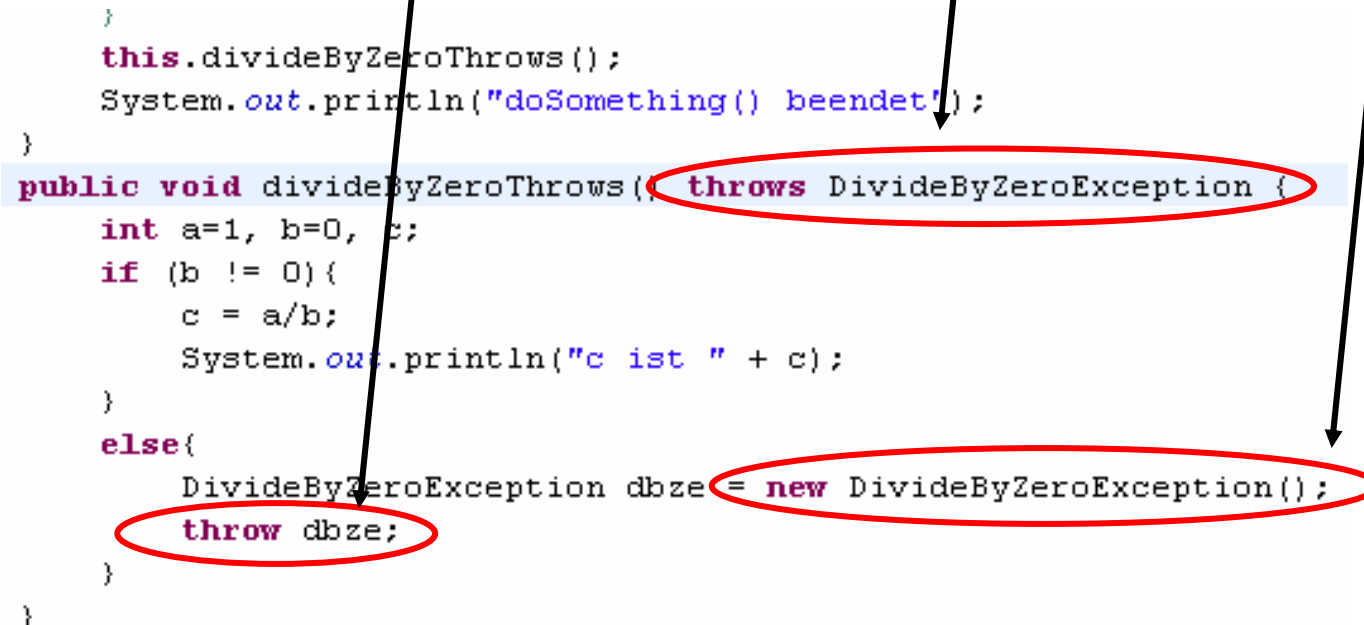
- Lösung, Schritt 1:
  - Methode anlegen und aufrufen,
  - Inhalt aus `divideByZeroMitVorbeugen()` kopieren,
  - `else`-Block putzen

```
//  
    }  
    this.divideByZeroThrows();  
    System.out.println("doSomething() beendet");  
}  
public void divideByZeroThrows() {  
    int a=1, b=0, c;  
    if (b != 0) {  
        c = a/b;  
        System.out.println("c ist " + c);  
    }  
    else {  
  
    }  
}
```

# Eigene Exceptions

- Lösung, Schritt 2:
  - Im else-Block ein `DivideByZeroException`-Objekt erzeugen
  - Objekt werfen
  - `throws`-Klausel im Methodenkopf einfügen

```
}  
this.divideByZeroThrows();  
System.out.println("doSomething() beendet");  
}  
public void divideByZeroThrows(throws DivideByZeroException {  
    int a=1, b=0, c;  
    if (b != 0){  
        c = a/b;  
        System.out.println("c ist " + c);  
    }  
    else{  
        DivideByZeroException dbze = new DivideByZeroException();  
        throw dbze;  
    }  
}
```



- Beobachtung:
  - Da `DivideByZeroException` eine `RuntimeException` ist, gibt es keinen Compiler-Fehler in `doSomething()`

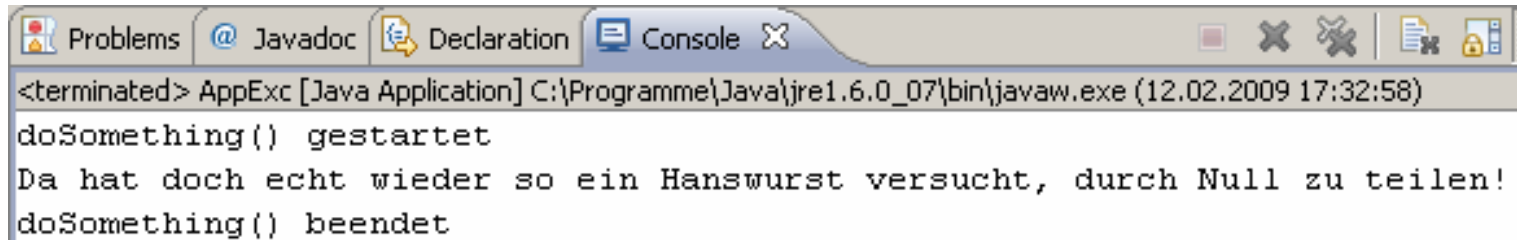
# Eigene Exceptions

- Alternativen:
  - Wir hätten unsere `DivideByZeroException` mit `throw new DivideByZeroException();` auch in einem Rutsch erzeugen und werfen können, ohne einen Variablennamen zu vergeben
  - In der `throws`-Klausel hätten wir auch `throws Exception` schreiben können.
- Lösung, Schritt 3:
  - Obwohl es in `doSomething()` keinen Compiler-Fehler gibt, wollen wir die Exception behandeln.
  - Wir tun dies mit try-/catch- und geben im catch-Block die Fehlermeldung aus.

```
try {  
    this.divideByZeroThrows();  
} catch (Exception e) {  
    System.out.println(e.getMessage());  
}  
System.out.println("doSomething() beendet");
```

# Eigene Exceptions

- Lösung, Schritt 4: Test durch Ausführung von AppExc



```
<terminated> AppExc [Java Application] C:\Programme\Java\jre1.6.0_07\bin\javaw.exe (12.02.2009 17:32:58)
doSomething() gestartet
Da hat doch echt wieder so ein Hanswurst versucht, durch Null zu teilen!
doSomething() beendet
```

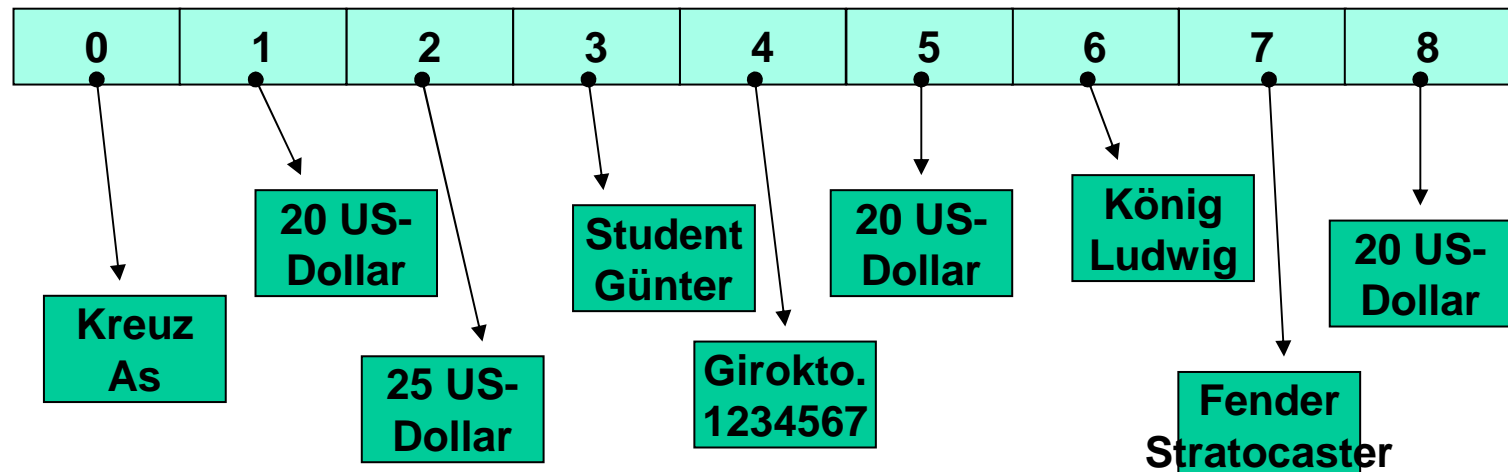
# **Programmierung II**

## **Thema 8: Collections, Teil 1**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

# Objekte zusammenfassen

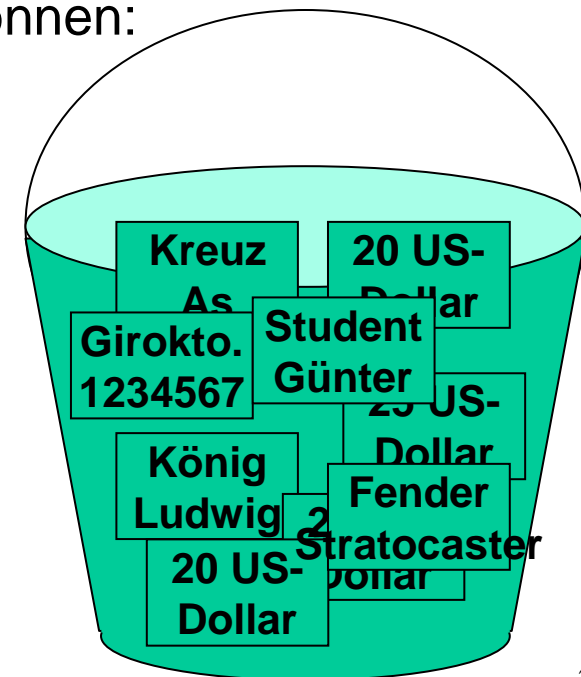
- Arrays sind nicht objektorientiert:
  - Die Arbeit mit Arrays geht nicht mit Methoden sondern nur über Indizes,
  - in Java ist „Array“ ein Sprachmittel und kein Objekt, obwohl eigentlich „alles“ ein Objekt sein sollte.
  - Vererbung wird für Arrays nicht eingesetzt.
- Arrays sind unflexibel:
  - Die Größe muss schon beim Anlegen bekannt sein und
  - Lässt sich nachträglich nicht ändern (nur wenn ein neues Array erzeugt wird).



# Objekte zusammenfassen

- „Schöner“ als ein Array wäre ein „Sammlungs“-Objekt, das
  - Objekte zusammenfassen kann und
  - mit Methoden arbeitet (Fähigkeiten)
- In der „richtigen Welt“ könnte man zum Beispiel einen Eimer („Bucket“) nehmen
- So ein Eimer müsste z.B. folgendes können:

Fähigkeit	Methode
Objekte aufnehmen	add(obj)
Objekte entfernen	remove(obj)
Prüfen, ob ein Objekt enthalten ist	contains(obj)
Objekte lesen	read(obj) / get(obj)

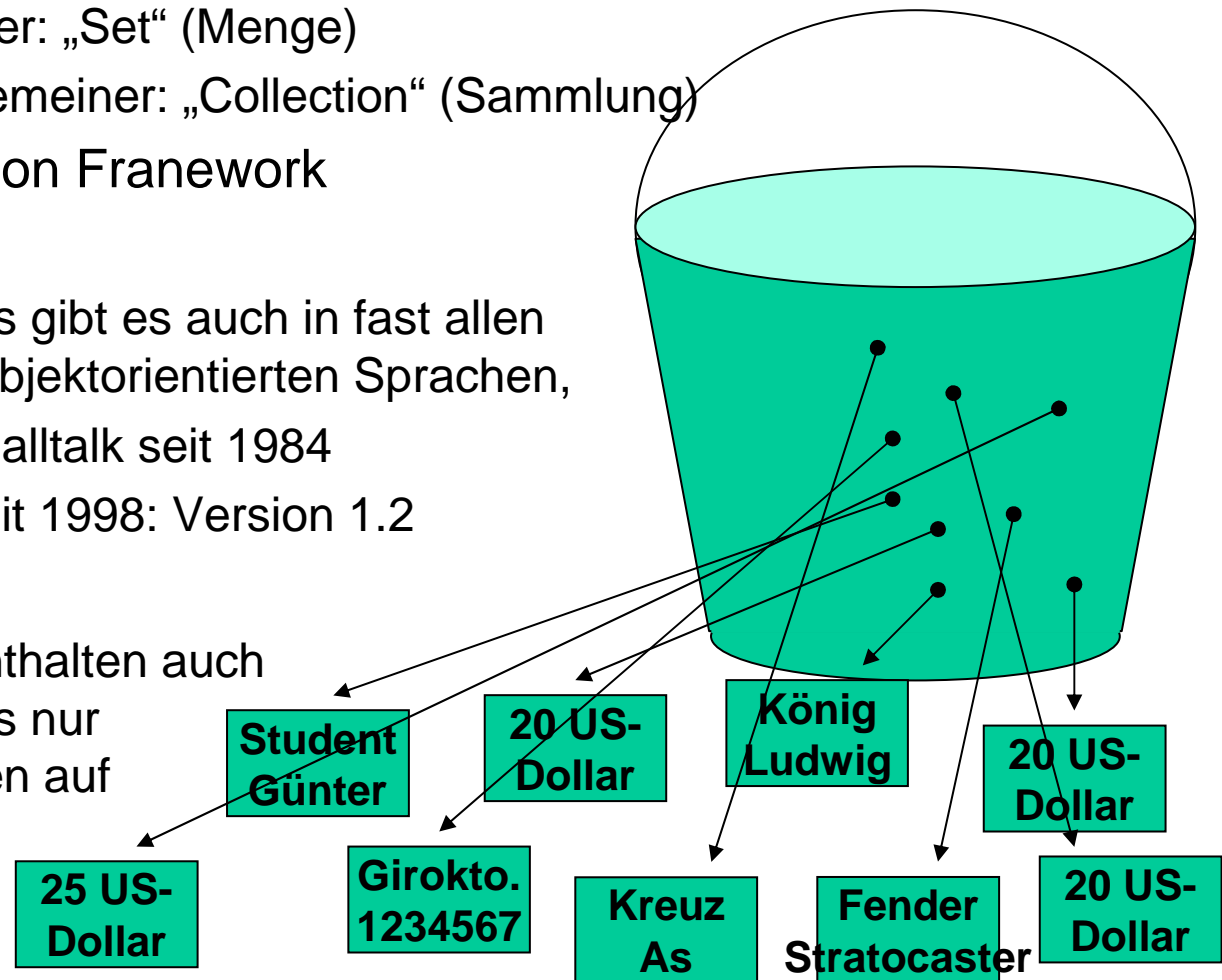


# Java Collection Framework

- Wir haben Glück: Solche Sammlungsobjekte gibt's schon!
- Das einfachste davon heißt in Java „HashSet“.
  - Allgemeiner: „Set“ (Menge)
  - Noch allgemeiner: „Collection“ (Sammlung)

## → Java Collection Framework

- Anmerkung:
  - Collections gibt es auch in fast allen anderen objektorientierten Sprachen,
  - z.B. in Smalltalk seit 1984
  - In Java seit 1998: Version 1.2
- Anmerkung:
  - In Java enthalten auch Collections nur Referenzen auf Objekte

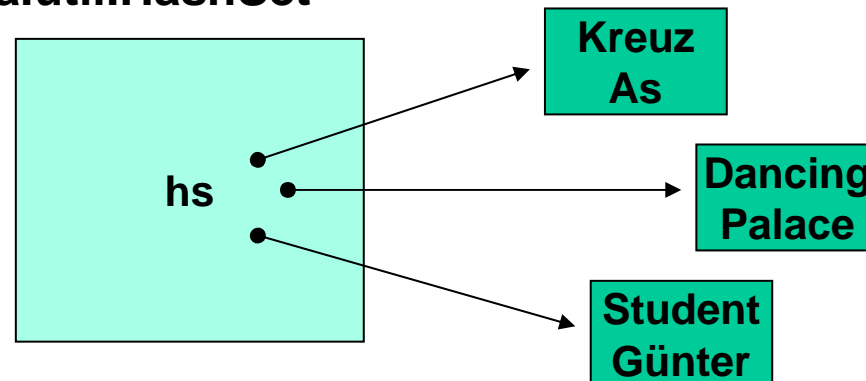




# Anwendung

- Aktion 1: Erzeugen Sie
  - in einem Package `de.fh_lu.o2s.collections`
  - eine Java-Anwendung `AppColl`.
- In deren `main()`-Methode soll
  - ein `HashSet` erzeugt werden,
  - eine Spielkarte, eine Disco und ein Student erzeugt und in das `HashSet` eingefügt werden,
  - das `HashSet` angezeigt werden

**Java.util.HashSet**

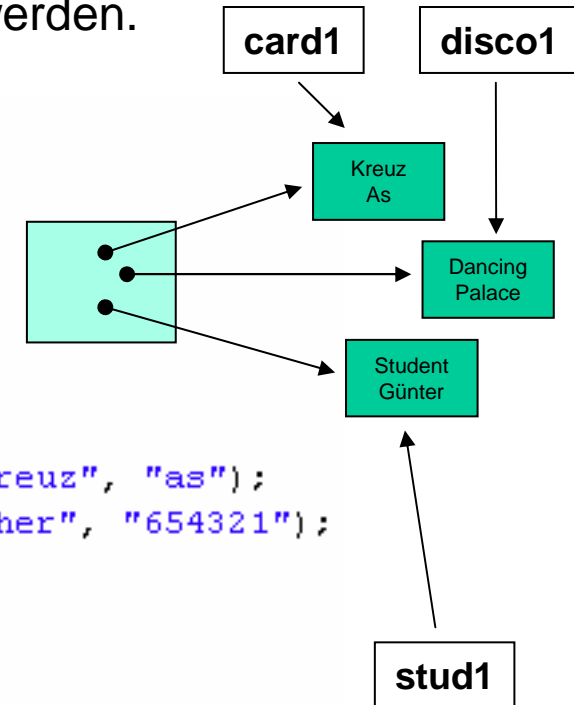


# Anwendung

- Lösung:
  - erzeugen der HashSet mit `new HashSet()`;
  - einfügen von Objekten mit `add(obj)`;
  - Anzeigen mit `System.out.println(...)`;
  - Warnungen können zunächst ignoriert werden.

```
import java.util.HashSet;
import de.fh_lu.o2s.cardgames.Spielkarte;
import de.fh_lu.o2s.personen.Student;
import de.fh_lu.o2s.unterhaltung.Disco;

public class AppColl {
    public static void main(String[] args) {
        HashSet hs = new HashSet();
        Spielkarte card1 = new Spielkarte("kreuz", "as");
        Student stud1 = new Student("Günther", "654321");
        Disco disco1 = new Disco(500);
        hs.add(card1);
        hs.add(stud1);
        hs.add(disco1);
        System.out.println(hs);
    }
}
```



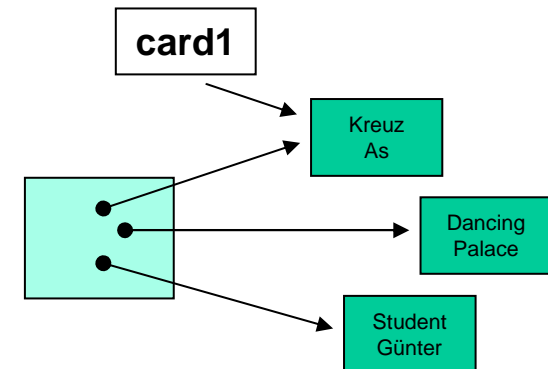
# Anwendung

- AppColl ausführen:



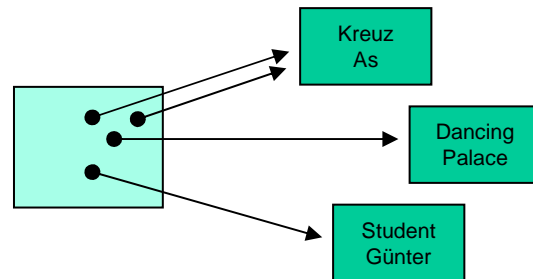
```
<terminated> AppColl [Java Application] C:\Programme\Java\jre6\bin\javaw.exe (23.07.2009 10:29:05)
Person(String) wird ausgeführt
Student(String, String) wird ausgeführt
[Disco für 500 Gäste mit Unterhaltungswert 500, Student mit Name: Günther und Geburtsja
```

- Beobachtung:
  - `System.out.println(...);` liefert den Inhalt des HashSet. Nicht schön aber vollständig.

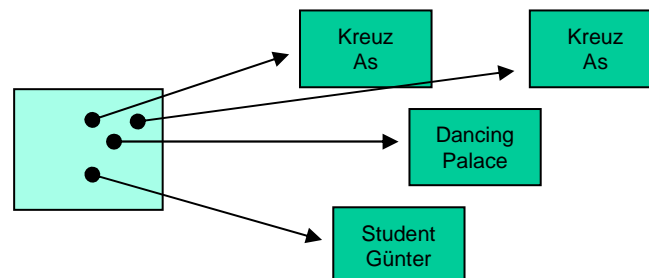


# Anwendung

- Aktion 2: Erweitern Sie `AppColl`, indem Sie
  - die Spielkarte (dasselbe Objekt) dem `HashSet` ein zweites Mal hinzuzufügen.
  - Führen Sie die Anwendung wieder aus.



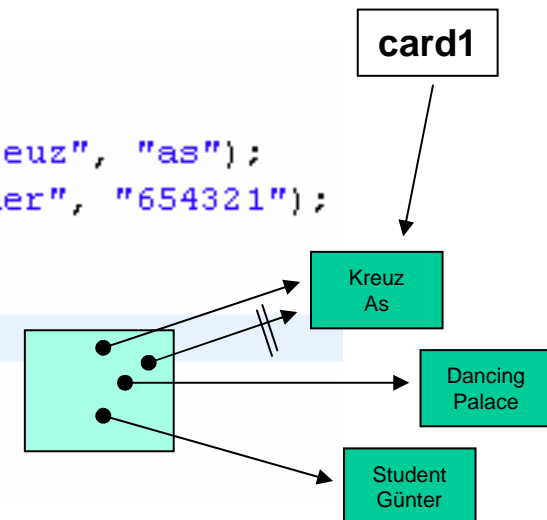
- Fügen Sie eine weitere Spielkarte (neues Objekt) mit derselben Farbe und Wert hinzu.
- Führen Sie die Anwendung wieder aus.



# Anwendung

- Lösung, Schritt 1:
  - Dieselbe Karte nochmal hinzufügen

```
public static void main(String[] args) {  
    HashSet hs = new HashSet();  
    Spielkarte card1 = new Spielkarte("kreuz", "as");  
    Student stud1 = new Student("Günther", "654321");  
    Disco disco1 = new Disco(500);  
    hs.add(card1);  
    hs.add(card1);  
    hs.add(stud1);  
    hs.add(disco1);  
}
```



- Ausführen:



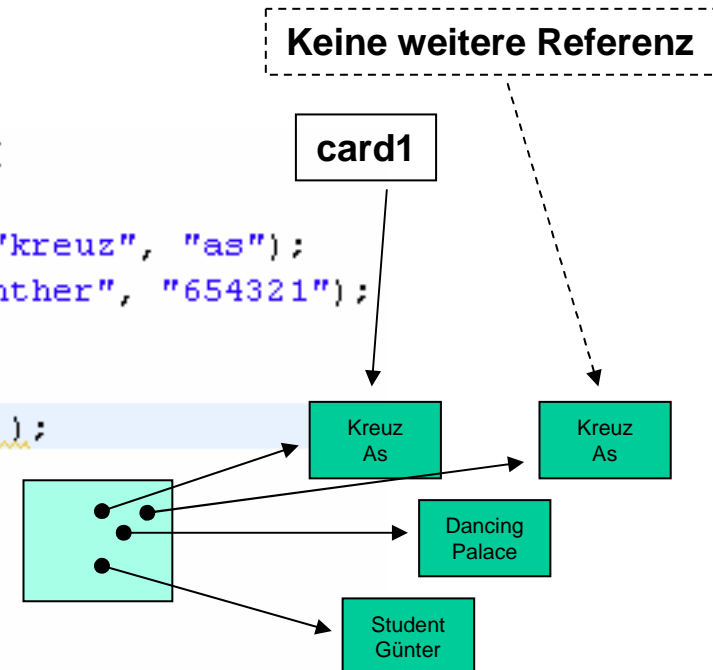
```
<terminated> AppColl [Java Application] C:\Programme\Java\jre6\bin\javaw.exe (23.07.2009 10:39:41)  
Person(String) wird ausgeführt  
Student(String, String) wird ausgeführt  
[Spielkarte mit Farbe kreuz und Wert as, Disco für 500 Gäste mit Unterhaltungswert 500, Stu
```

- Beobachtung:
  - Die Karte ist nur einmal vorhanden. Ein Objekt kann nur einmal in einem HashSet vorhanden sein.

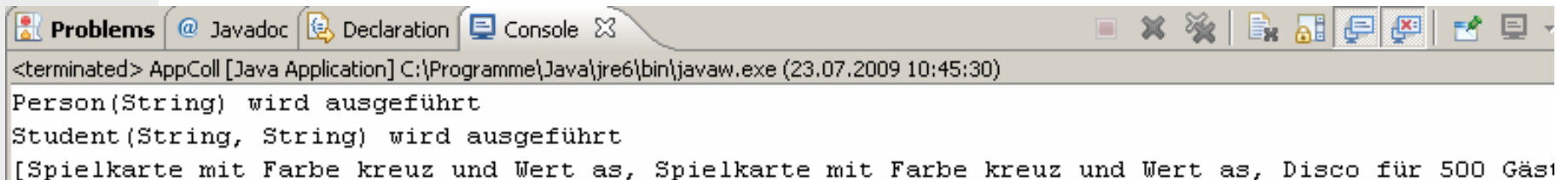
# Anwendung

- Lösung, Schritt 2:
  - Eine weitere Karte hinzufügen

```
public static void main(String[] args) {
    HashSet hs = new HashSet();
    Spielkarte card1 = new Spielkarte("kreuz", "as");
    Student stud1 = new Student("Günther", "654321");
    Disco disco1 = new Disco(500);
    hs.add(card1);
    hs.add(new Spielkarte("kreuz", "as"));
    //hs.add(card1);
    hs.add(stud1);
    hs.add(disco1);
    System.out.println(hs);
}
```



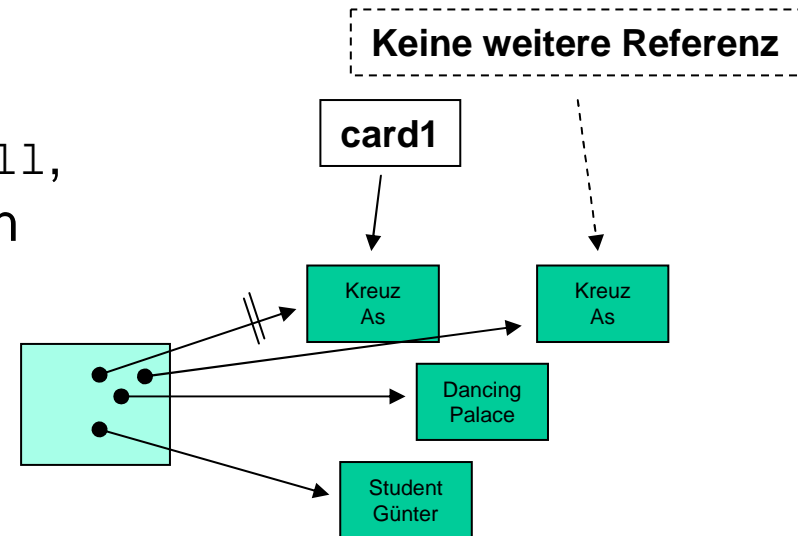
- Ausführen:



- Beobachtung:
  - Es sind jetzt zwei Karten vorhanden. Der Objektvergleich geht mit "==" und nicht mit equals()

# Entfernen

- Aktion 3: Erweitern Sie `AppColl`, indem Sie eine der Spielkarten wieder entfernen.



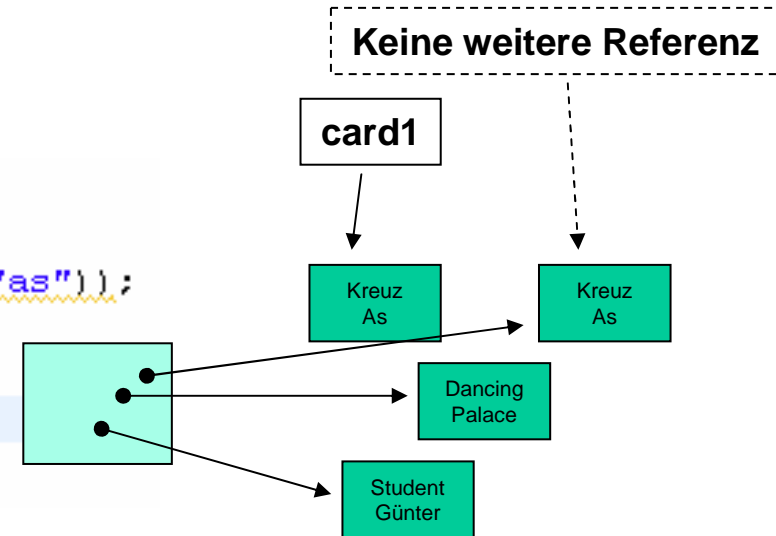
- Lösungsansatz:
  - Wir schicken unserem `HashSet hs` die Methode `remove(...)`
  - Als Parameter brauchen wir eine Referenz auf das Objekt, das wir löschen wollen (die Spielkarte).
  - Im oben dargestellten Beispiel haben wir keine Referenz auf die zweite Karte  

```
hs.add(card1);  
hs.add(new Spielkarte("kreuz", "as"));
```
  - Wir können diese deshalb nicht löschen und löschen stattdessen die erste Karte über die Referenz `card1`

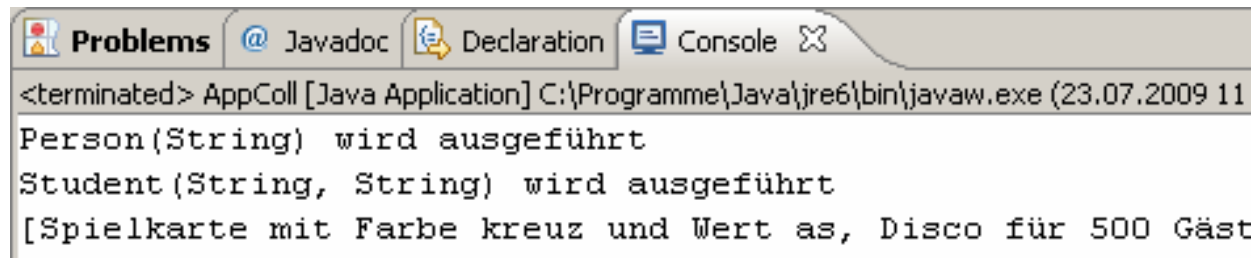
# Entfernen

- Lösung:

```
hs.add(card1);  
//hs.add(card1);  
hs.add(new Spielkarte("kreuz", "as"));  
hs.add(stud1);  
hs.add(disco1);  
hs.remove(card1);  
System.out.println(hs);
```



- Ausführen



```
Problems @ Javadoc Declaration Console X  
<terminated> AppColl [Java Application] C:\Programme\Java\jre6\bin\javaw.exe (23.07.2009 11  
Person(String) wird ausgeführt  
Student(String, String) wird ausgeführt  
[Spielkarte mit Farbe kreuz und Wert as, Disco für 500 Gäst
```

- Beobachtung:

- Es ist nur noch eine Karte im `HashSet` `hs` vorhanden.

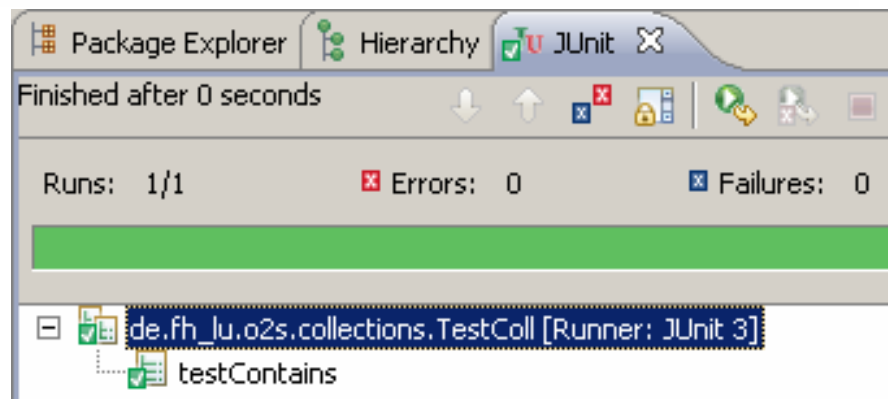


- Aktion 4a: Entwickeln Sie einen JUnit Test, um zu prüfen, ob `card1` im `HashSet hs` enthalten ist, und zwar
  - Vor dem Löschvorgang aus Aktion 3
  - Nach dem Löschvorgang aus Aktion 3
- Lösungsansatz:
  - Wir legen einen JUnit Test `TestColl` an mit einer Testmethode `testContains()`
  - Wir übernehmen den Code aus Aktion 3 nach `testContains()` (Student und Disco können wir weglassen)
  - Wir schicken unserem `HashSet hs` zweimal die Methode `boolean contains(card1)` und prüfen das Ergebnis mit `assertTrue()` bzw. `assertFalse()`

- Lösung: TestCase TestColl

```
public class TestColl extends TestCase {  
  
    public void testContains(){  
        HashSet hs = new HashSet();  
        Spielkarte card1 = new Spielkarte("kreuz", "as");  
        hs.add(card1);  
        hs.add(new Spielkarte("kreuz", "as"));  
        assertTrue(hs.contains(card1));  
        hs.remove(card1);  
        assertFalse(hs.contains(card1));  
    }  
}
```

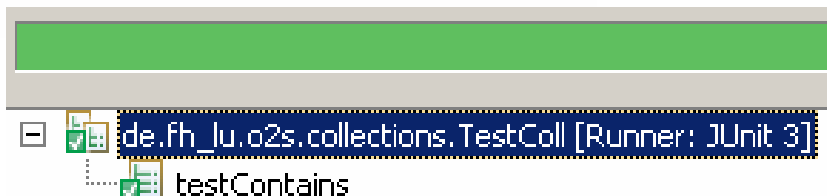
- Ausführung



# Testen

- Aktion 4b: Nutzen Sie außerdem die Methode `size()`, um die Anzahl der im `HashSet` `hs` enthaltenen Objekte zu prüfen
  - Vor dem Löschvorgang: 2 Objekte
  - Nach dem Löschvorgang: 1 Objekt
- Lösung:

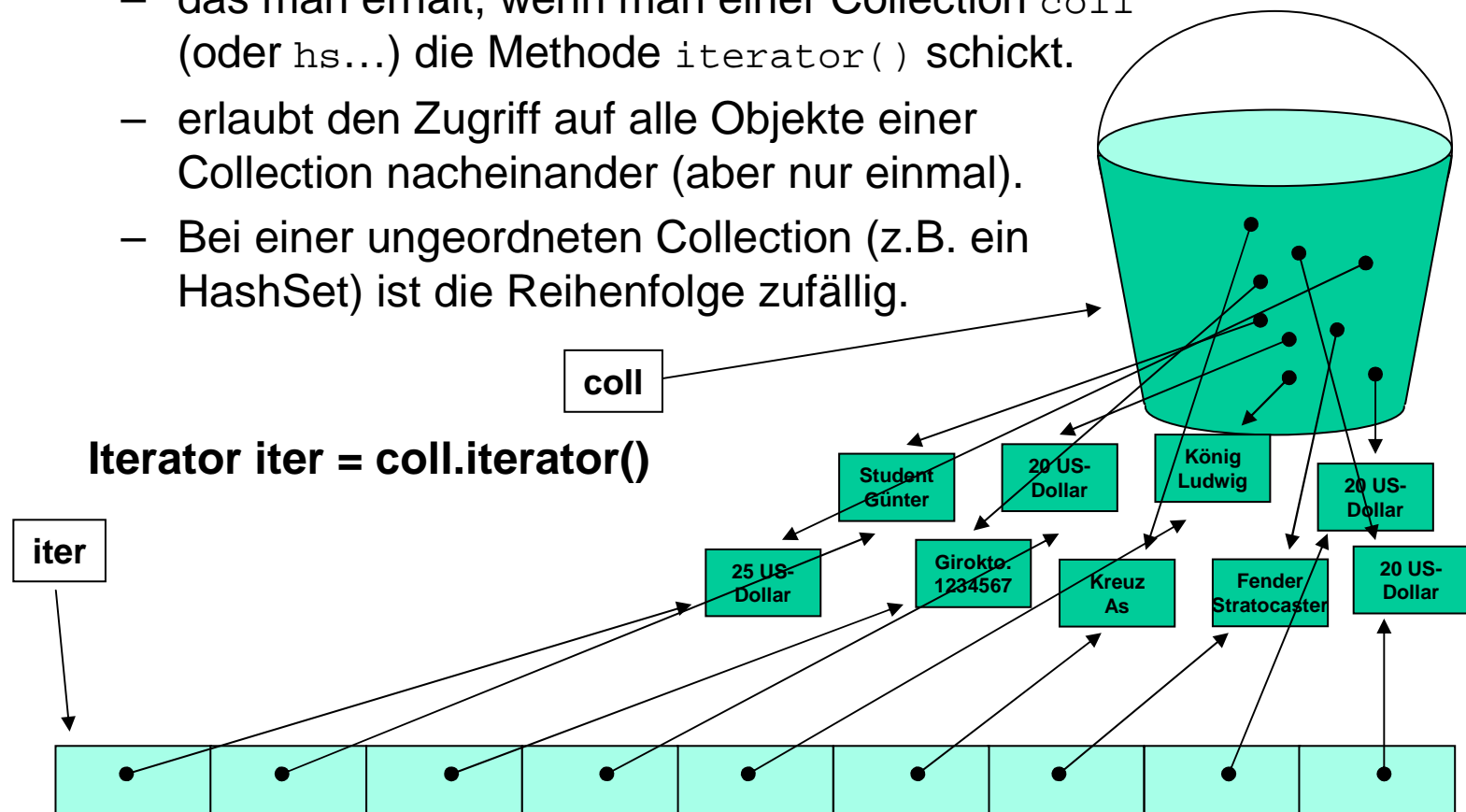
```
public void testContains(){  
    HashSet hs = new HashSet();  
    Spielkarte card1 = new Spielkarte("kreuz", "as");  
    hs.add(card1);  
    hs.add(new Spielkarte("kreuz", "as"));  
    assertTrue(hs.contains(card1));  
    assertEquals(hs.size(), 2);  
    hs.remove(card1);  
    assertFalse(hs.contains(card1));  
    assertEquals(hs.size(), 1);  
}
```



- Grundproblem
  - Ein HashSet ist nicht geordnet (im Gegensatz zu einem Array).
  - Deshalb ist eine Methode `getErstesElement()` oder `get(0)` für ein HashSet nicht sinnvoll.
- Übrigens: Es gibt auch geordnete Collections...
- Grundfrage:
  - Wie kann auf ein Element eines HashSet zugegriffen werden?
  - Eine Methode `get(obj)` ist nicht sinnvoll, weil wenn ich das Objekt `obj` bereits habe, brauche ich es nicht mehr `get`-ten.
- Was wäre denn jetzt eigentlich sinnvoll?
  - ➔ Alle Objekte hintereinander durchlaufen, z.B. um sie anzuzeigen oder auszuwerten
- Dazu gibt es ein Verfahren, das ein bisschen kompliziert wirkt, dafür funktioniert es für alle Arten von Collections gleich.

# Iterator

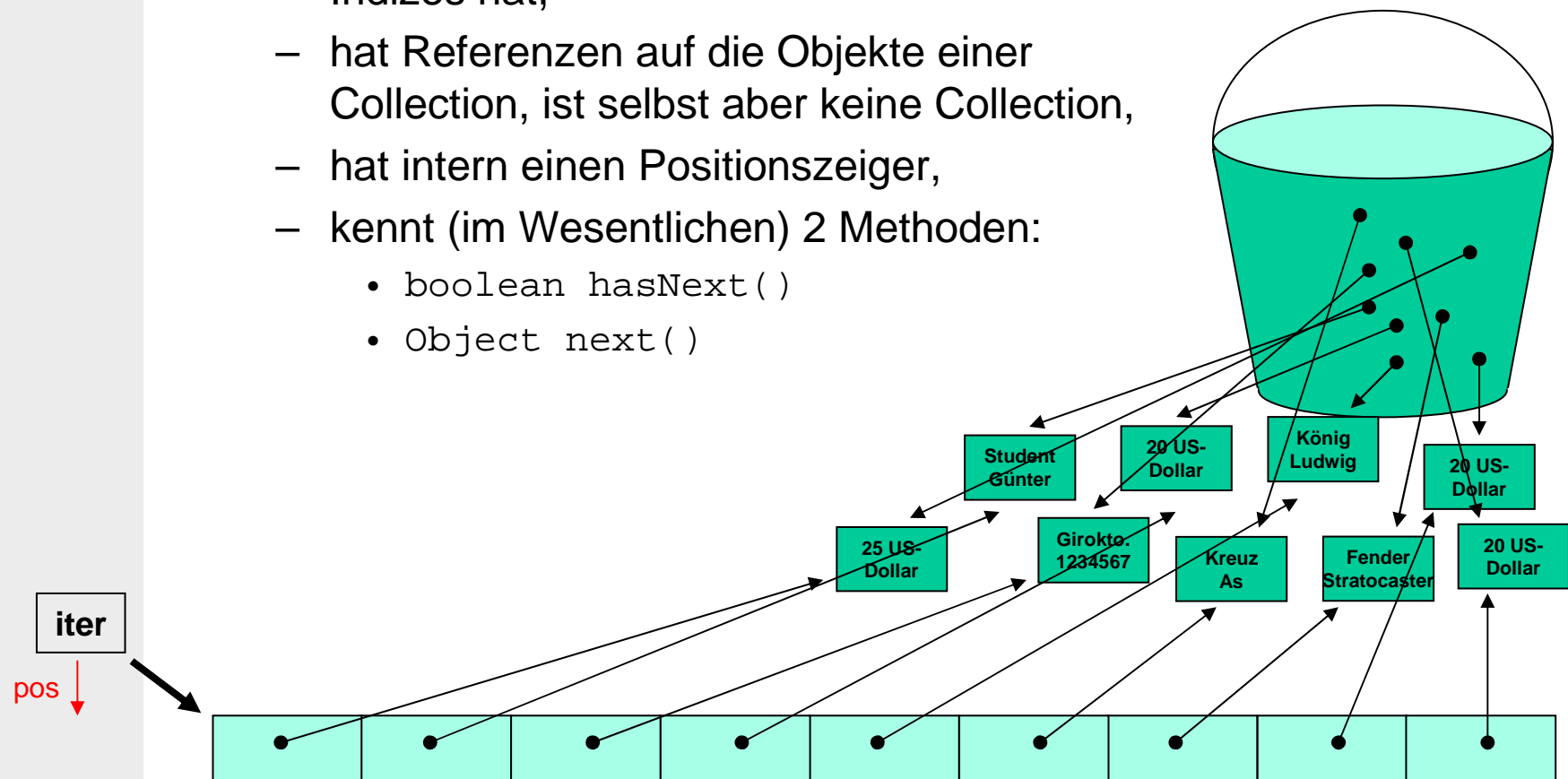
- Ein Iterator
  - ist ein zusätzliches Objekt vom Typ `java.util.Iterator`,
  - das man erhält, wenn man einer Collection `coll` (oder `hs...`) die Methode `iterator()` schickt.
  - erlaubt den Zugriff auf alle Objekte einer Collection nacheinander (aber nur einmal).
  - Bei einer ungeordneten Collection (z.B. ein `HashSet`) ist die Reihenfolge zufällig.



# Iterator

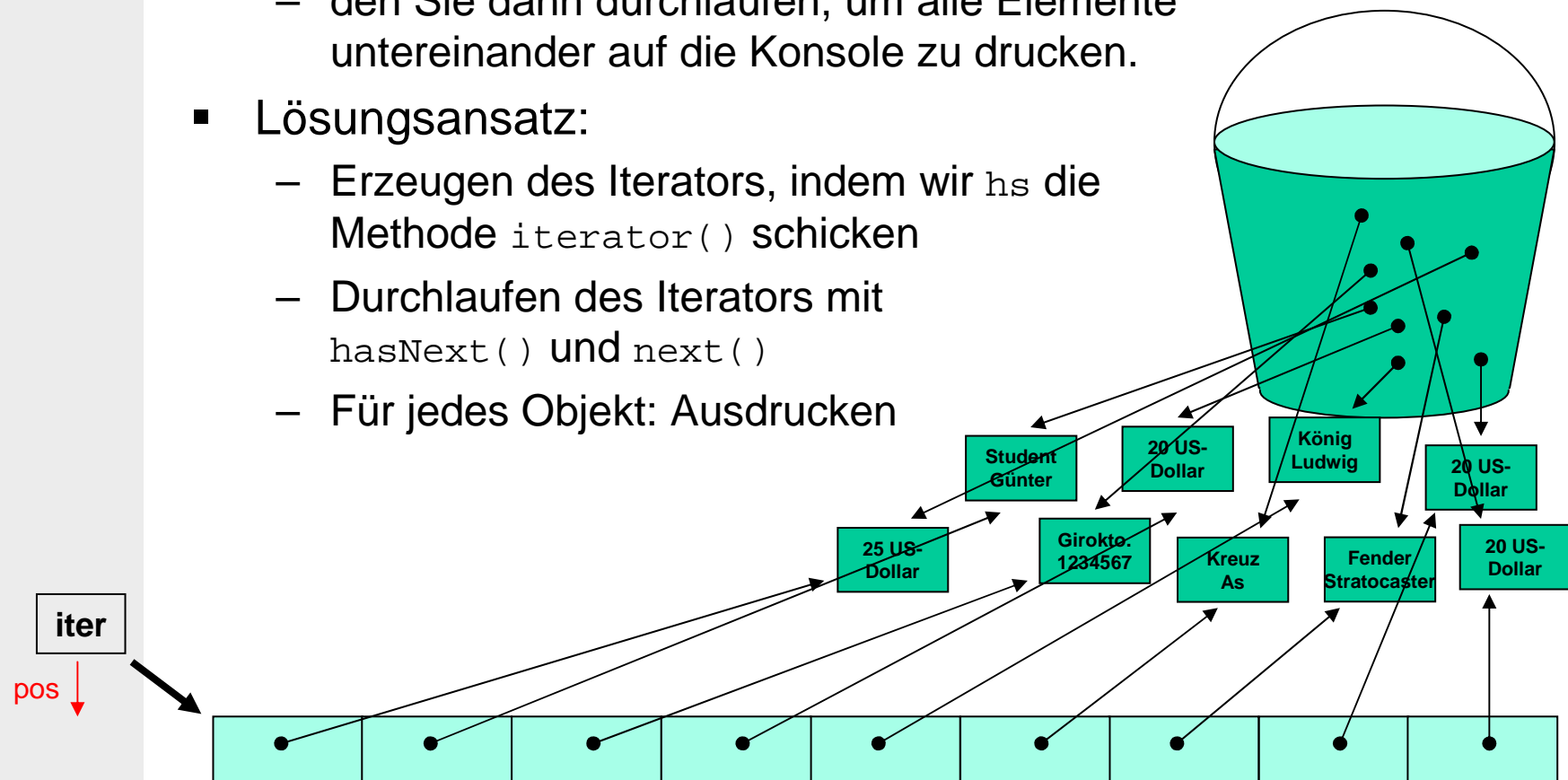
## ■ Ein Iterator

- sieht so ähnlich aus wie ein Array, ist aber keines, weil er keine Indizes hat,
- hat Referenzen auf die Objekte einer Collection, ist selbst aber keine Collection,
- hat intern einen Positionszeiger,
- kennt (im Wesentlichen) 2 Methoden:
  - `boolean hasNext()`
  - `Object next()`



# Iterator

- Aktion 5: Erweitern Sie Ihre Anwendung AppColl, indem Sie
  - Ihr HashSet `hs` einen Iterator erzeugen lassen,
  - den Sie dann durchlaufen, um alle Elemente untereinander auf die Konsole zu drucken.
- Lösungsansatz:
  - Erzeugen des Iterators, indem wir `hs` die Methode `iterator()` schicken
  - Durchlaufen des Iterators mit `hasNext()` und `next()`
  - Für jedes Objekt: Ausdrucken



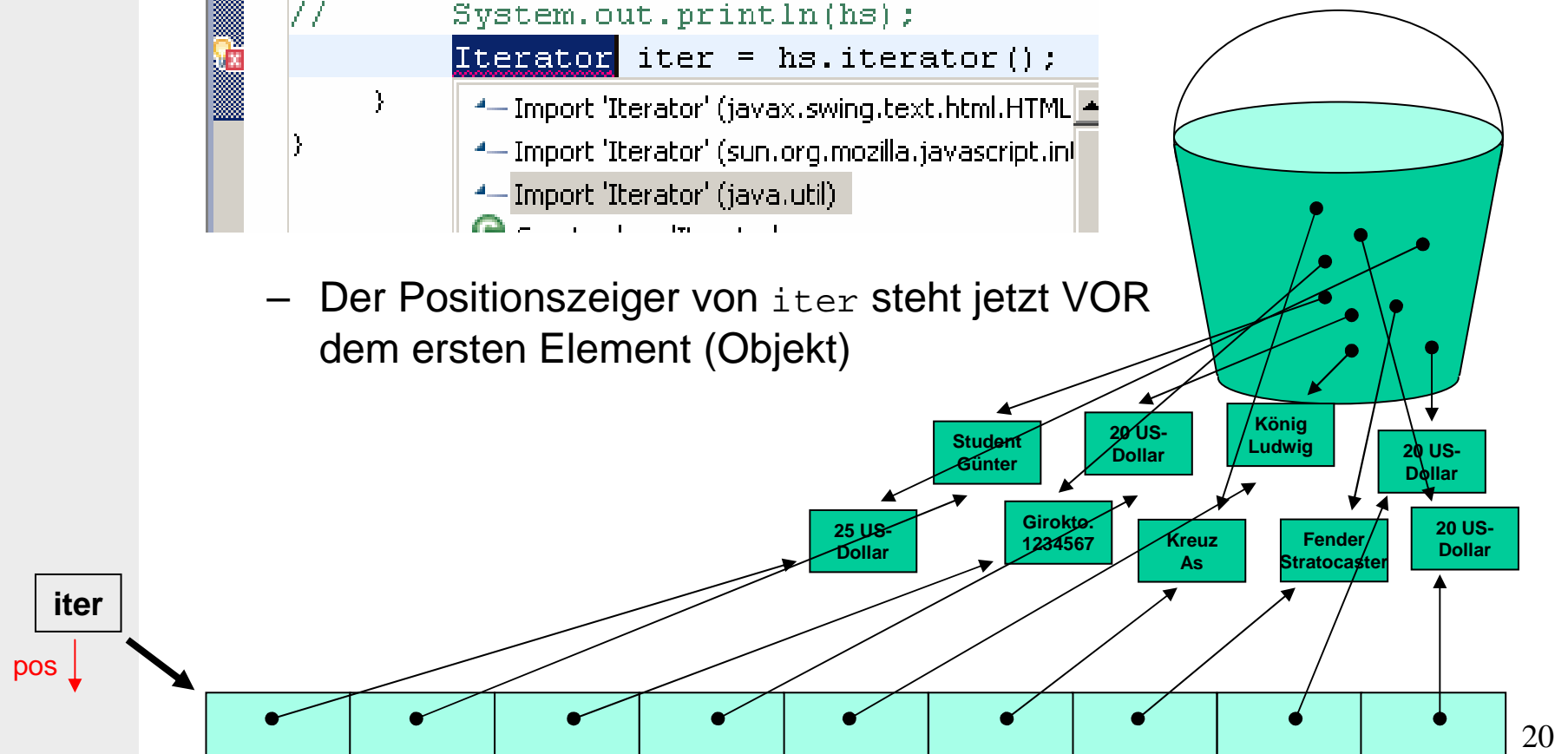
# Iterator

- Lösung, Schritt 1:
  - Datentyp `java.util.iterator` importieren und `Iterator` erzeugen.

```
//      System.out.println(hs);  
Iterator iter = hs.iterator();  
}  
}
```

Import 'Iterator' (javax.swing.text.html.HTML  
Import 'Iterator' (sun.org.mozilla.javascript.int  
Import 'Iterator' (java.util)

- Der Positionszeiger von `iter` steht jetzt VOR dem ersten Element (Objekt)



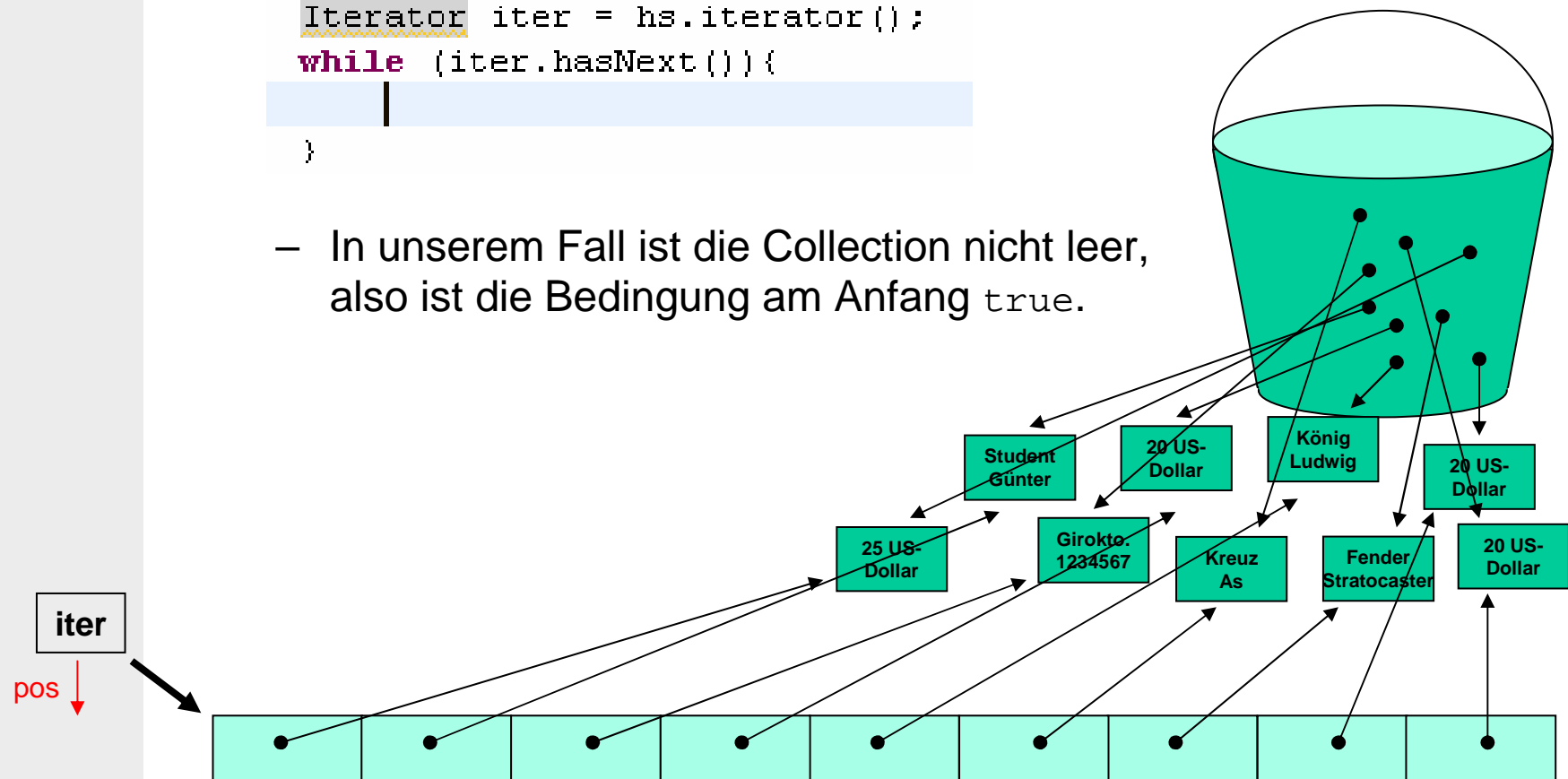


# Iterator

- Lösung, Schritt 2:
  - While-Schleife anlegen, in der gefragt wird, ob der Iterator noch nicht abgearbeitete Elemente besitzt.

```
Iterator iter = hs.iterator();  
while (iter.hasNext()) {  
    |  
}
```

- In unserem Fall ist die Collection nicht leer, also ist die Bedingung am Anfang `true`.

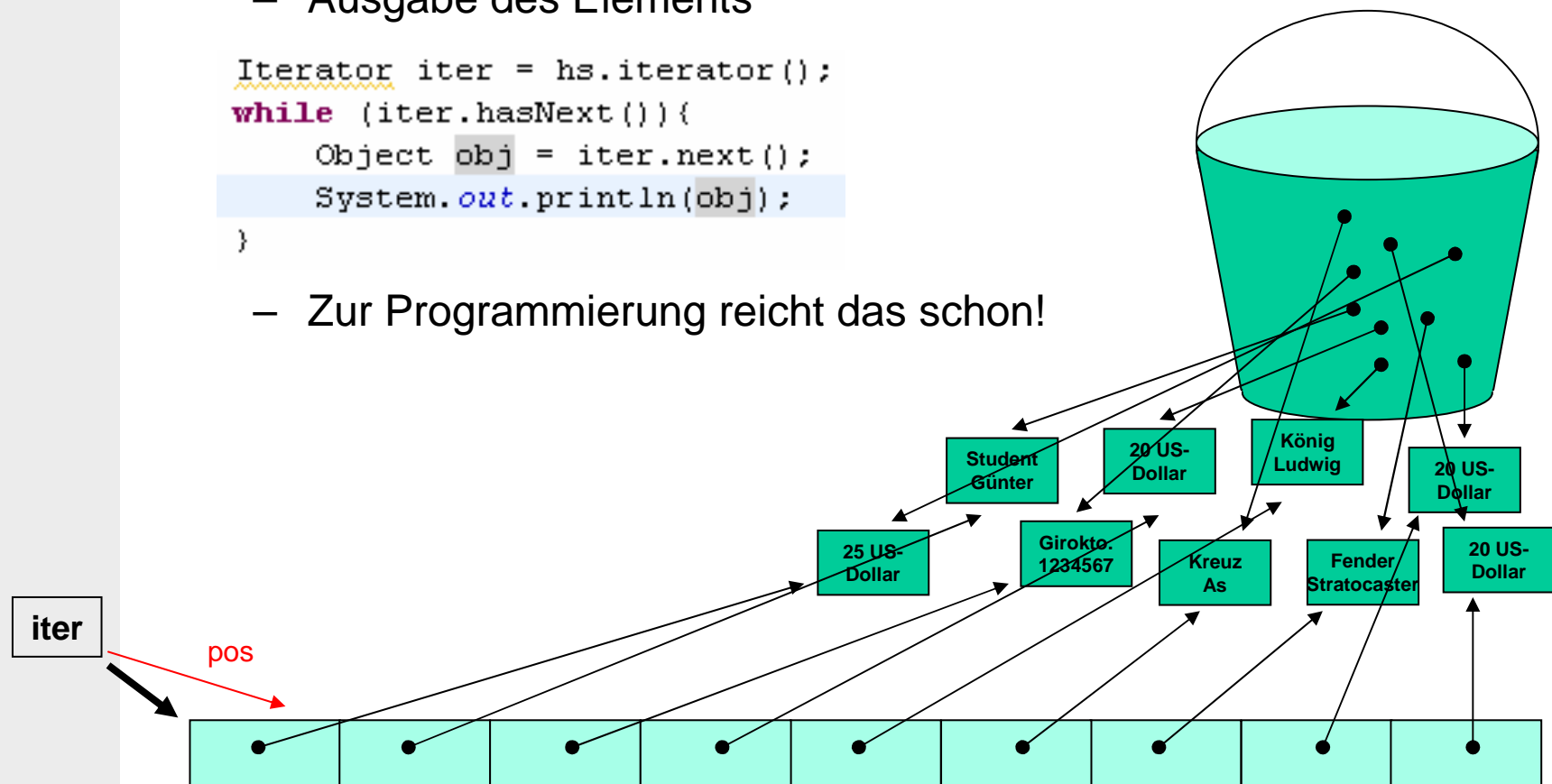


# Iterator

- Lösung, Schritt 3: Innerhalb der While-Schleife:
  - Nächstes Element holen mit `next()`. Dadurch wird auch der Positionszeiger um eine Position weiter gesetzt.
  - Ausgabe des Elements

```
Iterator iter = hs.iterator();  
while (iter.hasNext()) {  
    Object obj = iter.next();  
    System.out.println(obj);  
}
```

- Zur Programmierung reicht das schon!

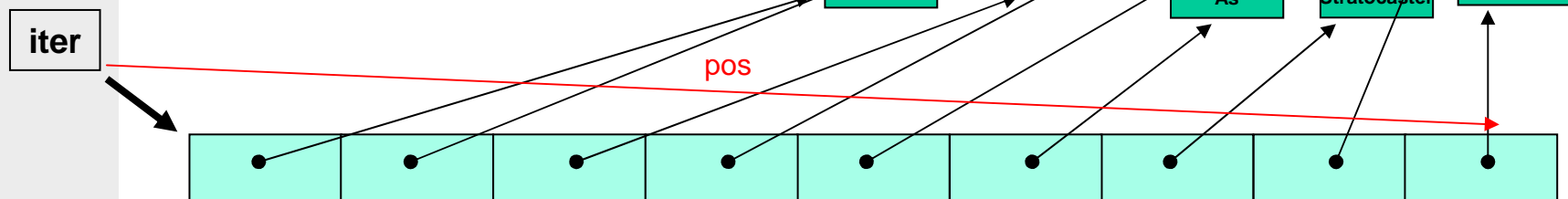


# Iterator

- Lösung, Schritt 4: Ausführen.
  - Mit jedem `next()` wandert der Positionszeiger weiter, bis er am Ende steht, weil er alle Elemente des Iterators durchlaufen hat.
  - Dann ist `hasNext() == false`, also bricht die Schleife ab.

Problems Javadoc Declaration Console  
<terminated> AppColl [Java Application] C:\Programme\Java\jre6\bin\javaw

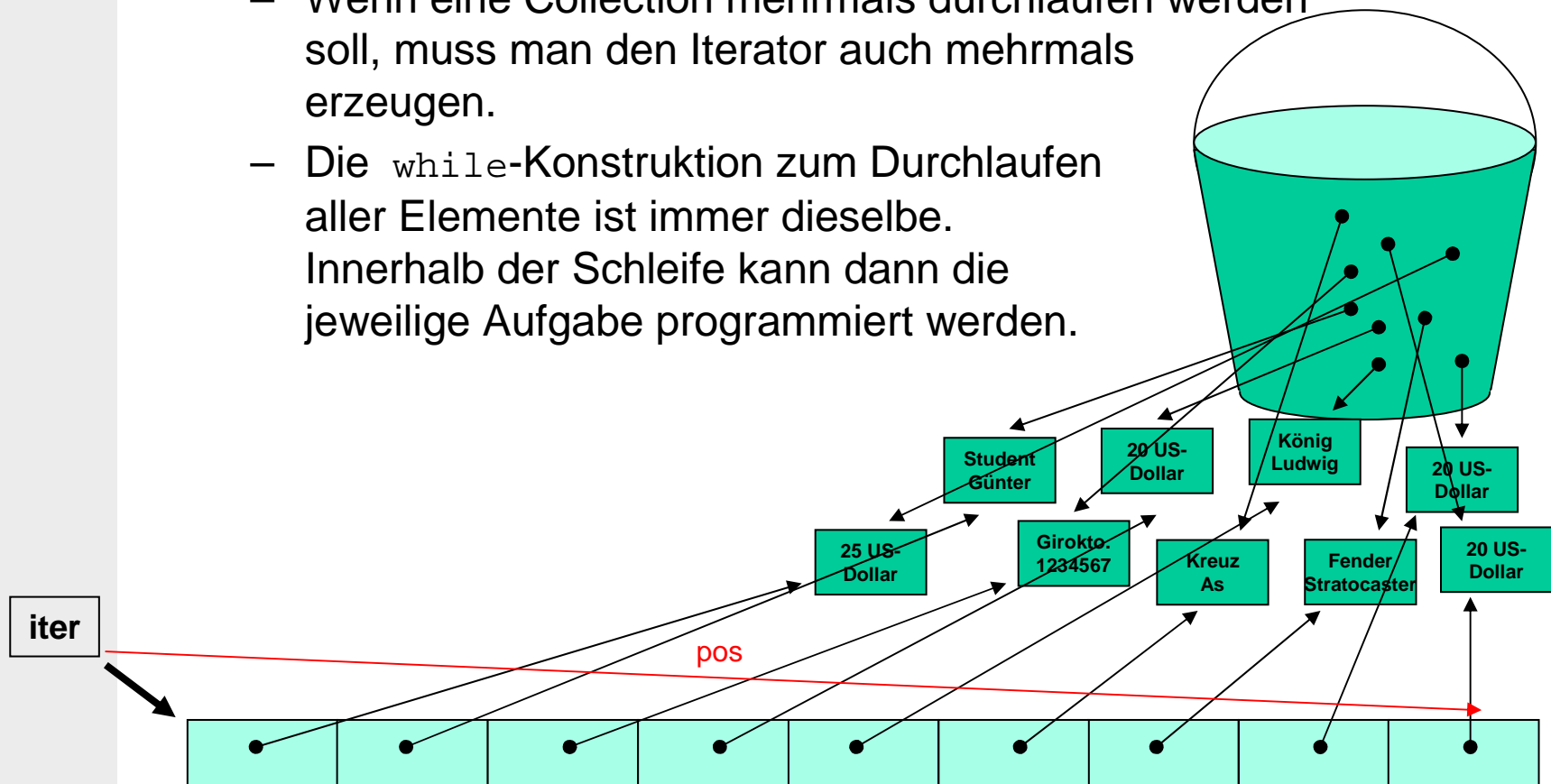
```
Person(String) wird ausgeführt  
Student(String, String) wird ausgeführt  
Disco für 500 Gäste mit Unterhaltungswert 500  
Student mit Name: Günther und Geburtsjahr: 0  
Spielkarte mit Farbe kreuz und Wert as
```



# Iterator

## ■ Anmerkungen:

- Wenn `next()` aufgerufen wird, obwohl `hasNext() == false`, dann gibt es eine `NoSuchElementException`
- Wenn eine Collection mehrmals durchlaufen werden soll, muss man den Iterator auch mehrmals erzeugen.
- Die `while`-Konstruktion zum Durchlaufen aller Elemente ist immer dieselbe. Innerhalb der Schleife kann dann die jeweilige Aufgabe programmiert werden.



# Interface Iterator

- **Anmerkung:**

- `java.util.Iterator` ist keine Klasse, sondern ein Interface. Dieses besitzt (im Wesentlichen) nur die Methoden `hasNext()` und `next()`.
- Natürlich entsteht durch `hs.iterator()` trotzdem ein Objekt einer bestimmten Klasse (Test mit `hs.iterator().getClass().getName()`)
- Jede Collection kann also ihre eigenen Iterator-Typen haben, die das Iterator-Interface implementieren müssen, damit alle einheitlich durchlaufen werden können.

**Interface Iterator**  
`boolean hasNext()`  
`Object next()`  
`void remove()`

**java.util.HashMap\$KeyIterator**  
`boolean hasNext()`  
`Object next()`  
`void remove()`



# foreach-Schleife

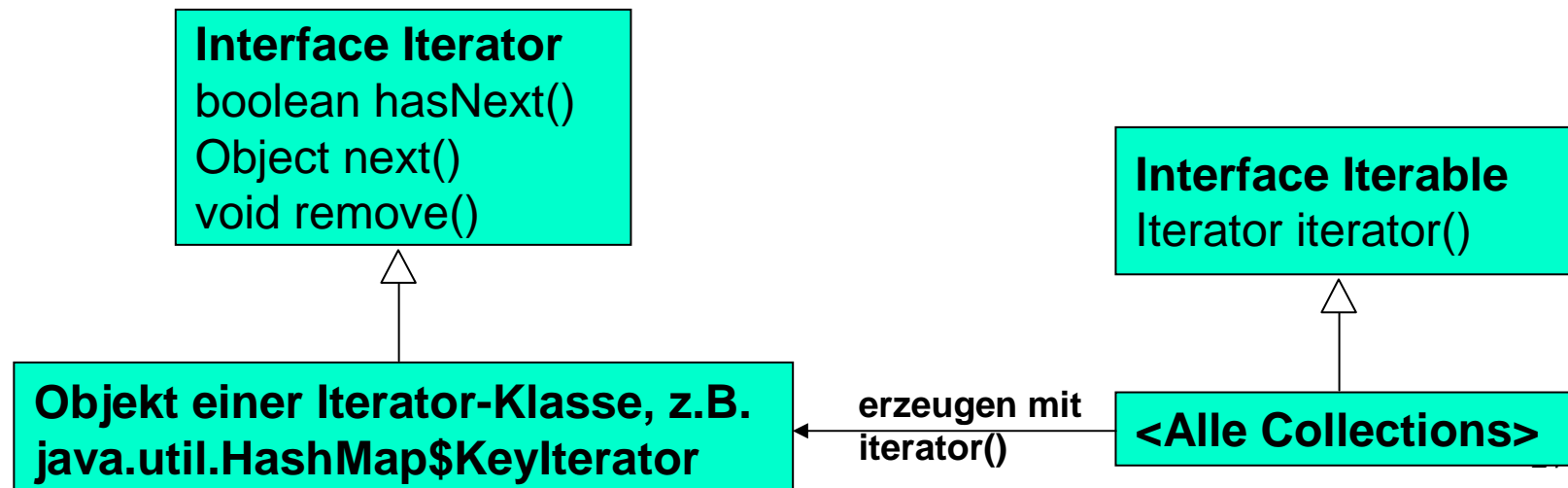
- Anmerkung:
  - Die Elemente einer Collection können auch mit einer objektorientierten `for`-Schleife (`foreach`-Schleife) durchlaufen werden.
- Syntax:

```
for (<Datentyp> <var> : <Collection>){  
    <Aktionen, wobei <var> benutzt werden darf>  
}
```
- Aktion 6:
  - Nutzen Sie eine `foreach`-Schleife, um die Elemente aus Ihrem `HashSet hs` zeilenweise auszugeben
- Lösung

```
for(Object obj : hs){  
    System.out.println(obj);  
}
```
- Ausführung
  - Analog zu Aktion 5 und auch selbes Ergebnis

# Interface Iterable

- Anmerkung:
  - Eine `foreach`-Schleife wird intern mit einem Iterator implementiert.
  - Sie kann deshalb nur genutzt werden für Collections, die die Methode `iterator()` besitzen.
  - Für alle Collections aus `java.util` ist dies der Fall.
  - Um dem System mitteilen zu können, dass man die Methode `iterator()` implementiert hat, gibt es das Interface `Iterable`, das aus genau dieser Methode besteht.



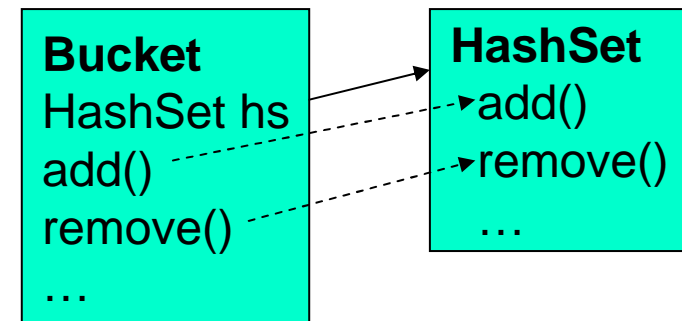
Bis jetzt haben wir kennengelernt:

- Die Collection-Klasse
  - HashSet
- Die Collection-Methoden
  - `boolean add(obj)`
  - `boolean remove(obj)`
  - `boolean contains(obj)`
  - `int size()`
- Das Iterable-Interface und die Collection-Methode
  - `java.util.Iterator iterator()`
- Das Iterator-Interface und die Iterator-Methoden
  - `hasNext()`
  - `next()`
- Anmerkung: Der Rückgabewert der Collection-Methoden `add()` und `remove()` gibt an, ob die Aktion geklappt hat (`true/false`).



# Fall 1: Klasse besitzt Collection

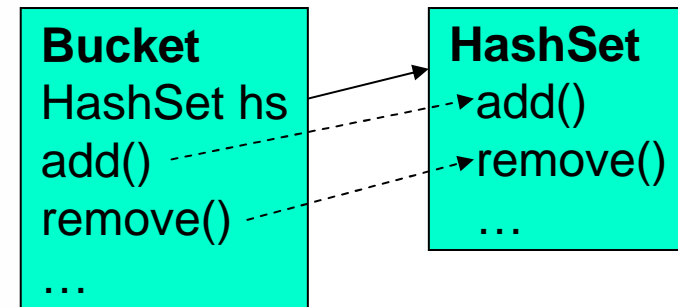
- Selbst entwickelte Klassen können Collections enthalten.
- Aktion 7a: Entwickeln Sie eine Klasse `Bucket` (Eimer), die
  - viele Objekte enthalten kann,
  - diese intern in einem `HashSet` speichert und
  - Auch die Methoden `add()`, `remove()`, `contains()` und `size()` kennt.
- Lösungsansatz:
  - `Bucket` reicht die Methodenaufrufe einfach weiter an das enthaltene `HashSet`
  - Der Konstruktor erzeugt zunächst ein leeres `HashSet`.
- Anmerkung:
  - Wie im Fall von Arrays ist zu unterscheiden zwischen einer leeren Variable (`hs == null`)
  - und einem leeren `HashSet` (`hs.size() == 0`)



# Fall 1: Klasse besitzt Collection

## ■ Lösung

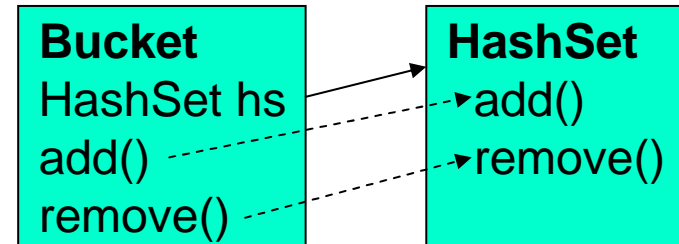
```
public class Bucket {  
    HashSet hs;  
    public Bucket() {  
        this.hs = new HashSet();  
    }  
    public boolean add(Object obj) {  
        return hs.add(obj);  
    }  
    public boolean remove(Object obj) {  
        return hs.remove(obj);  
    }  
    public boolean contains(Object obj) {  
        return hs.contains(obj);  
    }  
    public int size() {  
        return hs.size();  
    }  
}
```



# Fall 1: Klasse besitzt Collection

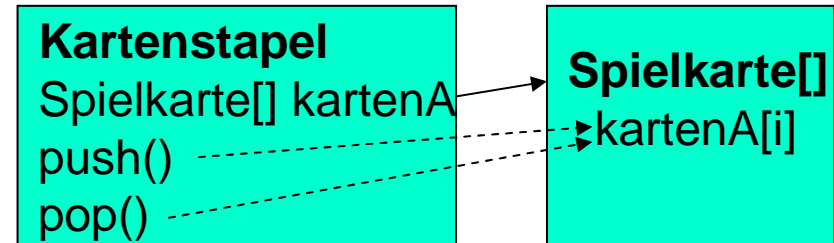
- Anmerkung:

- `Bucket` funktioniert im Prinzip analog zur Klasse `Kartenstapel`, nur einfacher.



- Erinnerung:

- `Kartenstapel` speichert ihre Karten in einem Array und
- stellt eigene Methoden für die Verwaltung bereit, speziell `push()`, `pop()`.
- Diese Methoden legen dann Karten im Array ab oder lesen diese aus dem Array.



- Anmerkung:

- Wenn `Bucket` nicht mehr Fähigkeiten besitzt als `HashSet`, dann brauchen wir eigentlich keinen `Bucket`,
- aber `Bucket` kann jetzt erweitert werden, vgl. nächste Folie

# Fall 1: Klasse besitzt Collection

- Aktion 7b: erweitern Sie Ihre Klasse `Bucket`
  - um eine `toString()`-Methode,
  - die den Inhalt des `Bucket` – im Gegensatz zum `HashSet` – zeilenweise ausgibt.
- Lösungsansatz:
  - Iterator über das `HashSet` nutzen, um alle Elemente zu durchlaufen und daraus den String bauen.
- Lösung:

```
public String toString(){  
    String s = new String();  
    Iterator iter = hs.iterator();  
    while (iter.hasNext()){  
        s += iter.next().toString() + "\n";  
    }  
    return s;  
}
```

# Fall 1: Klasse besitzt Collection

- Ausführen: Passen Sie Ihre AppColl an, so dass
  - anstatt eines HashSet ein Bucket genutzt wird.
  - Alle anderen Zeilen bleiben unverändert.

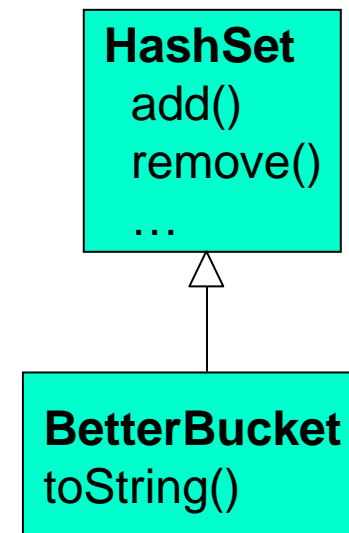
```
public static void main(String[] args) {  
    //HashSet hs = new HashSet();  
    Bucket      hs      = new Bucket();  
    Spielkarte card1 = new Spielkarte("kreuz", "as");  
    Student     stud1 = new Student("Günther", "654321");  
    Disco       disco1 = new Disco(500);  
    hs.add(card1);  
    hs.add(new Spielkarte("kreuz", "as"));  
    hs.add(stud1);  
    hs.add(disco1);  
    hs.remove(card1);  
    System.out.println(hs);  
}
```

<terminated> AppColl [Java Application] C:\Programme\Java\jre6\bin\javaw.exe (23.07.2009 17:28:20)

```
Person(String) wird ausgeführt  
Student(String, String) wird ausgeführt  
Disco für 500 Gäste mit Unterhaltungswert 500  
Student mit Name: Günther und Geburtsjahr: 0 und Matrikelnummer: 654321  
Spielkarte mit Farbe kreuz und Wert as
```

## Fall 2: Klasse ist Collection

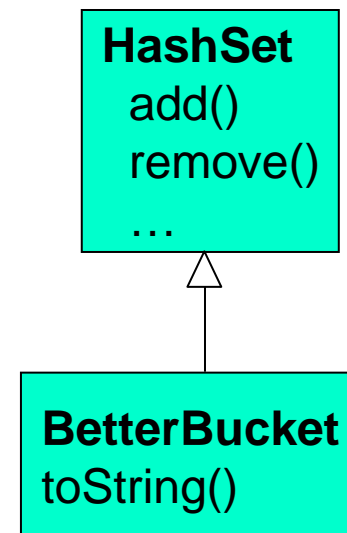
- Selbst entwickelte Klassen können auch Collections sein.
- Aktion 8: Entwickeln Sie eine Klasse `BetterBucket`, die
  - alles kann, was `Bucket` auch kann (vgl. Aktionen 7a, 7b), aber
  - intern von `HashSet` erbt.
- Lösungsansatz:
  - Die Methoden funktionieren automatisch,
  - ein expliziter Konstruktor ist nicht nötig, eine leere Variable kann nicht vorkommen
  - die `toString()`-Methode bezieht sich auf `this` anstatt `hs`.



## Fall 2: Klasse ist Collection

- Lösung:

```
public class BetterBucket extends HashSet {  
    public String toString(){  
        String s = new String();  
        Iterator iter = this.iterator();  
        while (iter.hasNext()){  
            s += iter.next().toString() + "\n";  
        }  
        return s;  
    }  
}
```



- Anmerkung:

- Hier dargestellt ist die gesamte Klasse und nicht nur die `toString()`-Methode.

- Anmerkung:

- Für dieses Vorgehen gibt es keine Analogie bei Arrays. Hier sind Collections wesentlich mächtiger!

## Fall 2: Klasse ist Collection

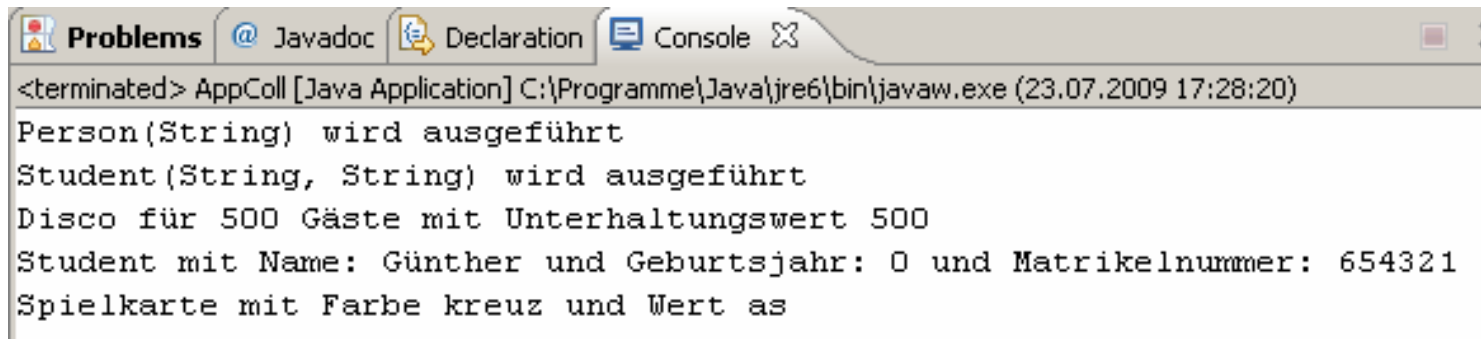
- Ausführung: Passen Sie Ihre AppColl an, so dass
  - anstatt eines Bucket ein BetterBucket genutzt wird.
  - Alle anderen Zeilen bleiben unverändert.

```
public static void main(String[] args) {  
    //HashSet hs = new HashSet();  
    //Bucket hs = new Bucket();  
    BetterBucket hs = new BetterBucket();  
    Spielkarte card1 = new Spielkarte("kreuz")
```

**HashSet**  
add()  
remove()  
...

**BetterBucket**  
toString()

- Ergebnis (identisch zu Aktion 7b):



Problems Javadoc Declaration Console

<terminated> AppColl [Java Application] C:\Programme\Java\jre6\bin\javaw.exe (23.07.2009 17:28:20)

Person(String) wird ausgeführt  
Student(String, String) wird ausgeführt  
Disco für 500 Gäste mit Unterhaltungswert 500  
Student mit Name: Günther und Geburtsjahr: 0 und Matrikelnummer: 654321  
Spielkarte mit Farbe kreuz und Wert as



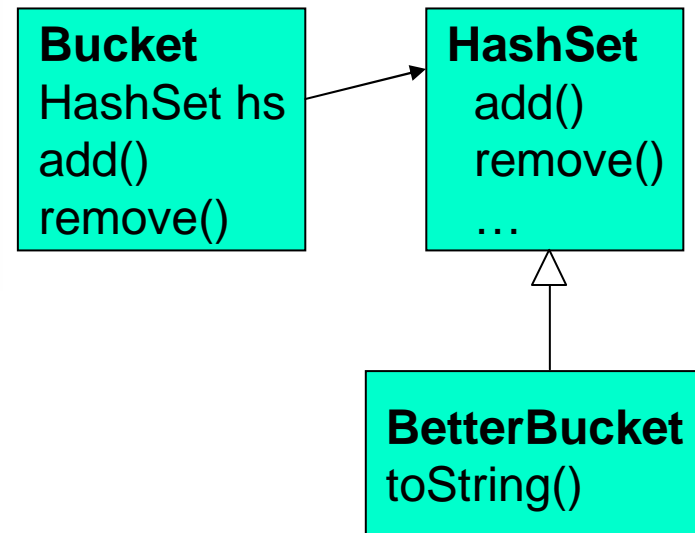
# Alternative: foreach-Schleifen

- Die toString()-Methoden aus den Aktionen 7b und 8
  - Hätten alternativ auch mit foreach-Schleifen programmiert werden können.
- Lösung in Bucket (Zugriff auf hs):

```
public String toString(){  
    String s = new String();  
    for(Object obj : hs){  
        s += obj.toString() + "\n";  
    }  
    return s;  
}
```

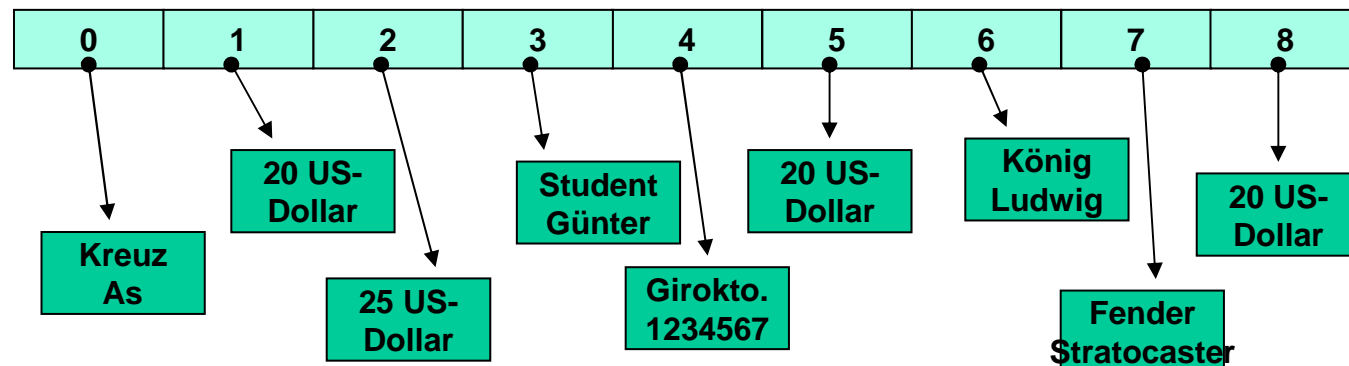
- Lösung in BetterBucket (Zugriff auf this):

```
public String toString(){  
    String s = new String();  
    for(Object obj : this){  
        s += obj.toString() + "\n";  
    }  
    return s;  
}
```

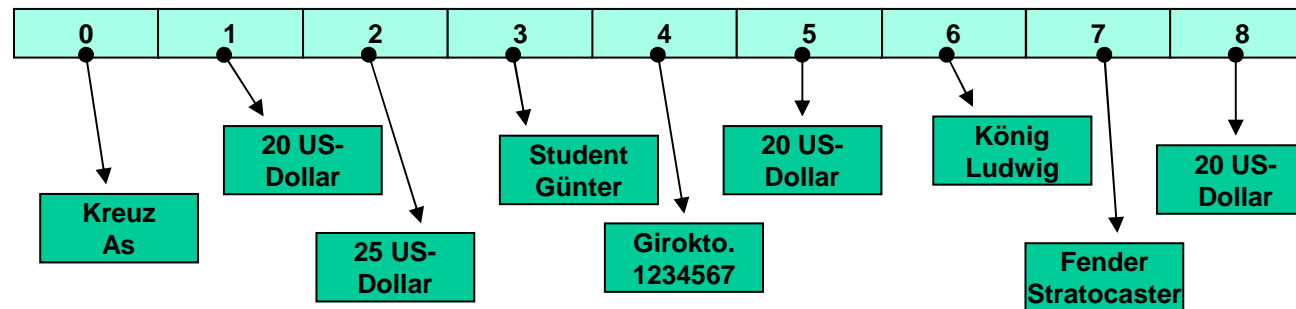


# Geordnete Collections - List

- In geordneten Collections
  - besitzen die enthaltenen Objekte eine bestimmte Reihenfolge.
  - Es muss nicht unbedingt eine Sortierreihenfolge sein.
  - Dies ist analog zu Arrays.
  - In Java nennt man dies „Liste“ – englisch: „List“.
- Damit gibt es weitere sinnvolle Methoden,
  - z.B. könnte es sinnvoll sein, auf das i-te Element einer Liste list zuzugreifen: `list.get(i)`
  - oder auf das erste oder letzte Element der Liste zuzugreifen: `list.get(0)` bzw. `list.get(list.size() - 1)`



- Damit sehen Listen ähnlich aus wie Arrays.



- Außerdem enthalten Listen
  - alle Methoden von HashSet (`add()`, `remove()`, etc.)
  - Dadurch kann – im Gegensatz zu Arrays – die Größe (Länge) einer Liste dynamisch verändert werden.
- Es gibt verschiedene Arten von Listen, z.B.:
  - ArrayList
  - LinkedList
  - Vector

- Aktion 9: Benutzen Sie in Ihrer AppColl
  - anstatt `HashSet` / `Bucket` / `BetterBucket`
  - zunächst eine `ArrayList` und danach
  - einen `Vector`, um dasselbe Ergebnis zu erzielen.
- Lösung, Schritt 1: `ArrayList`

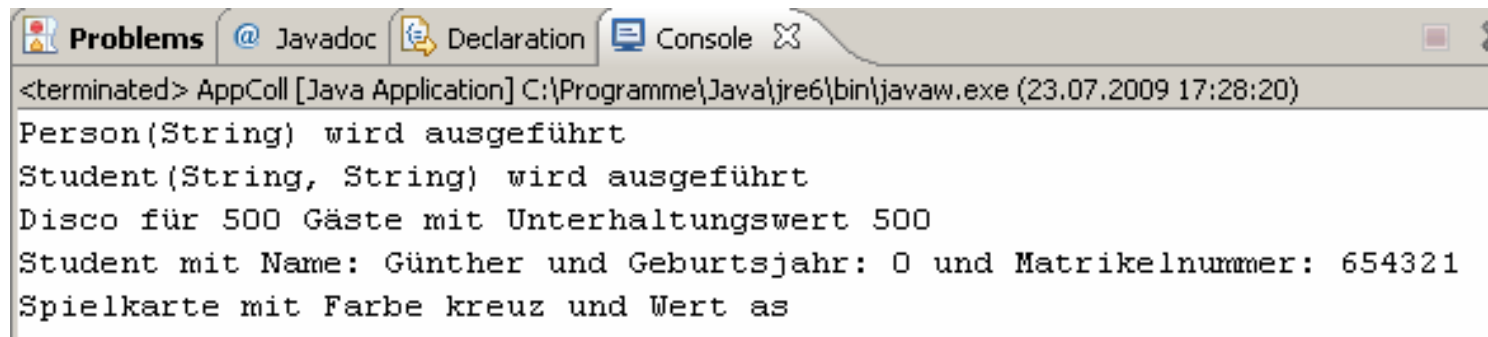
```
public static void main(String[] args) {  
    //    HashSet hs = new HashSet();  
    //    Bucket  hs = new Bucket();  
    //    BetterBucket hs  = new BetterBucket();  
    ArrayList coll = new ArrayList();  
    Spielkarte card1 = new Spielkarte("kreuz", "as");  
    Student stud1 = new Student("Günther", "654321")  
    Disco disco1 = new Disco(500);  
    coll.add(card1);  
    coll.add(new Spielkarte("kreuz", "as"));  
    coll.add(stud1);  
}
```

- Anmerkung:
  - Hier wurde zur Vereinheitlichung nebenbei auch der Name der Variable von `hs` auf `coll` geändert.

- Lösung, Schritt 2: Vector

```
public static void main(String[] args) {  
    //      HashSet hs = new HashSet();  
    //      Bucket  hs = new Bucket();  
    //      BetterBucket hs  = new BetterBucket();  
    //      ArrayList coll = new ArrayList();  
    Vector coll = new Vector();  
    Spielkarte card1 = new Spielkarte("Kreuz"  "as");
```

- Ergebnis (wie üblich):



```
Problems  @ Javadoc  Declaration  Console  X  
<terminated> AppColl [Java Application] C:\Programme\Java\jre6\bin\javaw.exe (23.07.2009 17:28:20)  
Person(String) wird ausgeführt  
Student(String, String) wird ausgeführt  
Disco für 500 Gäste mit Unterhaltungswert 500  
Student mit Name: Günther und Geburtsjahr: 0 und Matrikelnummer: 654321  
Spielkarte mit Farbe kreuz und Wert as
```

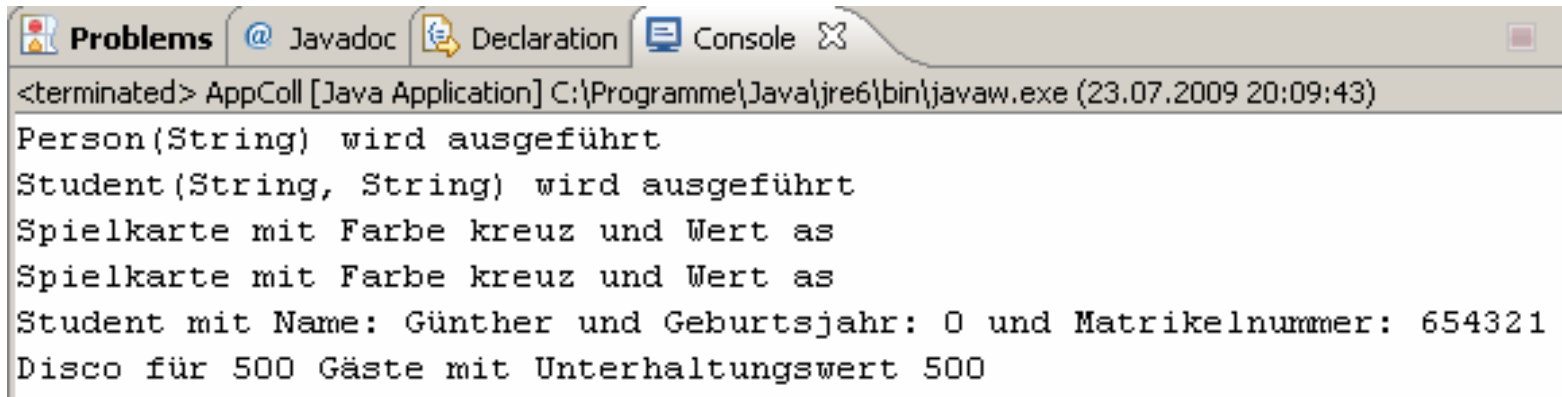
- Aktion 10: Fügen Sie der Collection (dem Vector) in Ihrer AppColl dieselbe Spielkarte (dasselbe Objekt) mehrfach hinzu

- Lösung:

```
Vector coll = new Vector();  
Spielkarte card1 = new Spielkarte("kreuz", "as");  
Student stud1 = new Student("Günther", "654321");  
Disco disco1 = new Disco(500);  
coll.add(card1);  
coll.add(card1);  
coll.add(stud1);  
coll.add(disco1);  
coll.add(new Spielkarte("kreuz", "as"));  
coll.remove(card1);  
System.out.println(coll);  
for(Object obj : coll){  
    System.out.println(obj);  
}
```

- Anmerkung:
  - Eine Liste (z.B. ein Vector) könnte auch mit einer for-Schleife mit Index durchlaufen werden.

- Ergebnis:
  - Die Karte ist mehrfach vorhanden.

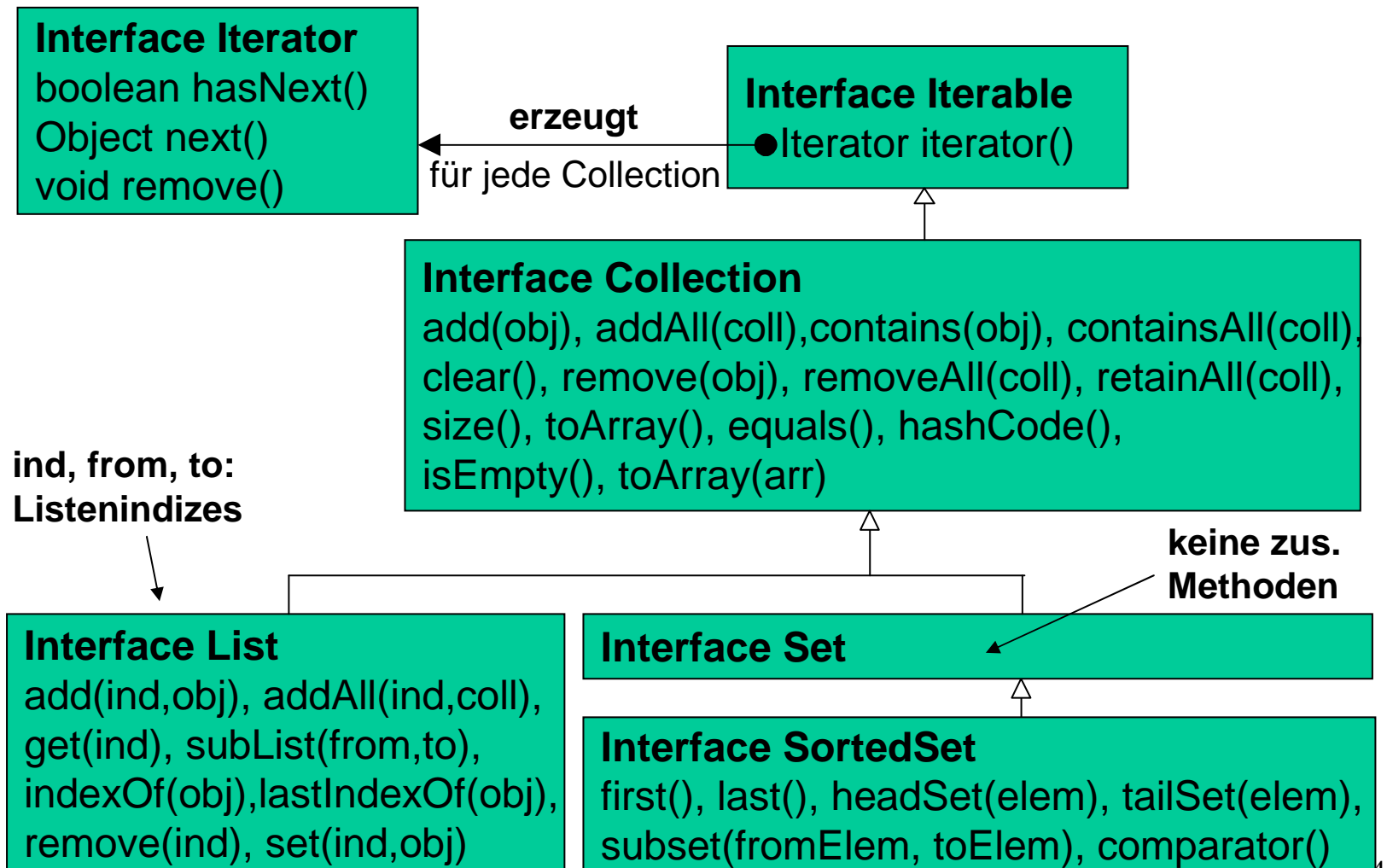


```
<terminated> AppColl [Java Application] C:\Programme\Java\jre6\bin\javaw.exe (23.07.2009 20:09:43)
Person(String) wird ausgeführt
Student(String, String) wird ausgeführt
Spielkarte mit Farbe kreuz und Wert as
Spielkarte mit Farbe kreuz und Wert as
Student mit Name: Günther und Geburtsjahr: 0 und Matrikelnummer: 654321
Disco für 500 Gäste mit Unterhaltungswert 500
```

- Zusammenfassung:
  - Mengen (Set) können Elemente nur einmal enthalten
  - Listen (List) sind geordnet und können Elemente dadurch auch mehrfach enthalten.
  - Die nächsten Folien stellen die Standard-Interfaces und –klassen dar.

# Java Collection Framework

## ■ Interfaces im Java Collection Framework

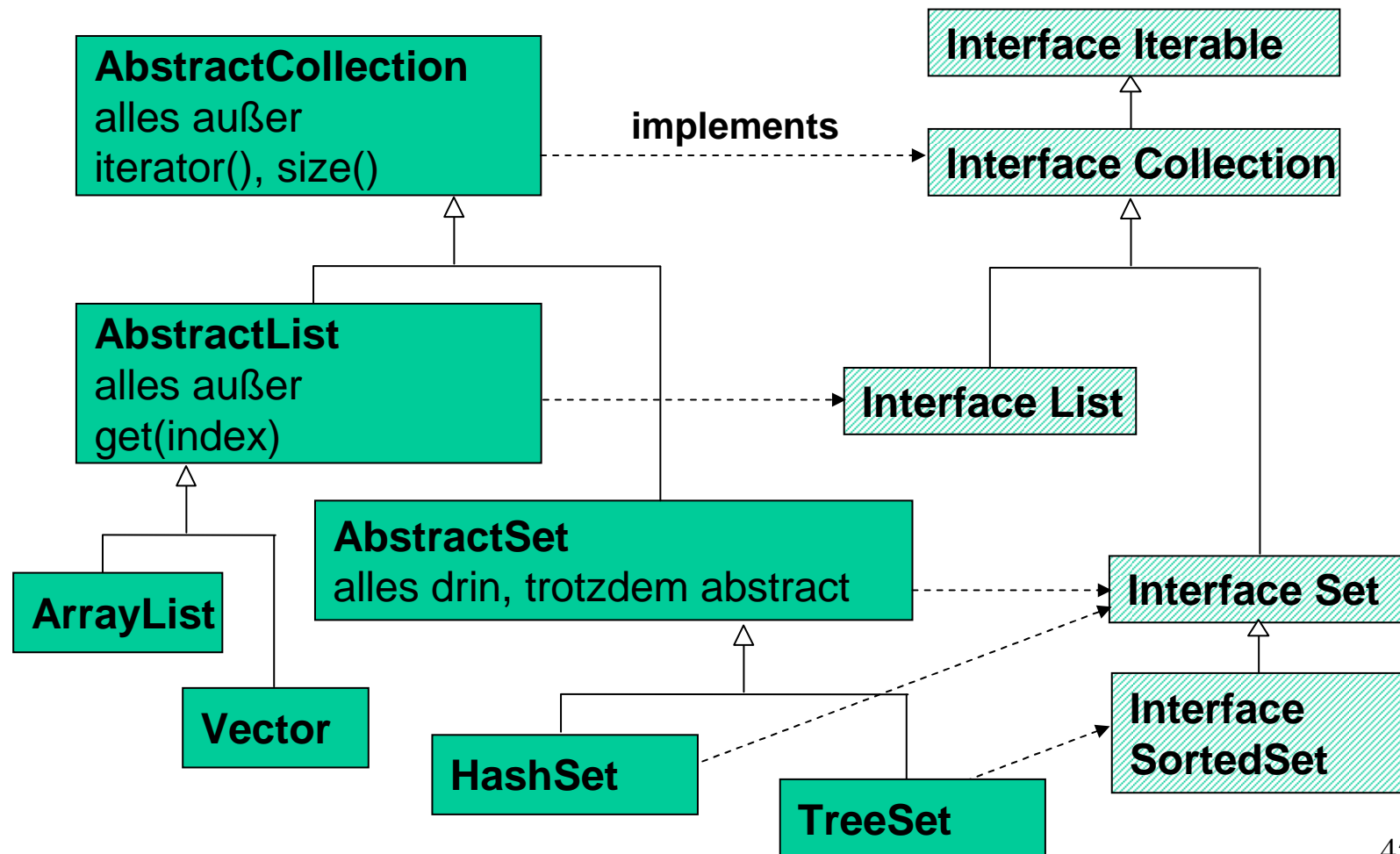


Anm.: Das Interface Queue wurde hier weggelassen



# Java Collection Framework

- Klassen im Java Collection Framework



- Beobachtung:
  - Damit eine Klasse `MyClass` von sich sagen kann “`MyClass` ‘is-a’ `Collection`“, muss diese alle 14 `Collection`-Methoden und die Methode `iterator()` implementieren.
  - Klassen des Java Collection Frameworks tun dies.
- Erinnerung:
  - Für die Benutzung der `foreach`-Schleife reicht es aus, das Interface `Iterable` (nur die Methode `iterator()`) zu implementieren.
- Anmerkung:
  - Zur Benutzung in eigenen Programmen sind die bestehenden `Collection`-Typen oft ausreichend, z.B. `Vector`
  - Oft empfiehlt es sich aber auch, eigene `Collection`-Klassen zu entwickeln.

- Anmerkung: Wenn man selber eine Collection bauen will, kann man
  - Sich überlegen, wie man die Daten intern speichern will (z.B. in einem Array) und dann
  - alle 15 Methoden selbst schreiben.
- Alternativ kann man
  - von `AbstractCollection`, `AbstractSet` oder `AbstractList` erben und
  - braucht dann nur noch wenige Methoden selbst zu schreiben (s.o.).
- Noch besser: Man kann (wie in `BetterBucket`)
  - Von einer fertigen Collection-Klasse erben (z.B. `HashSet` oder `Vector`) und
  - braucht dann nur noch genau das zu programmieren, was man ändern will (z.B. die `toString()`-Methode)

# Übersicht

Klasse	ArrayList	Vector	LinkedList
Interface(s)	List	List	List, Queue
Collection-Methoden	add(obj), addAll(coll), contains(obj), containsAll(coll), clear(), remove(obj), removeAll(coll), retainAll(coll), size(), toArray(), toArray(arr), equals(), hashCode(), isEmpty()		
List-Methoden	add(ind,obj), addAll(ind,coll), set(ind,obj) get(ind), subList(from,to), indexOf(obj), lastIndexOf(obj), remove(ind)		
Queue-Methoden			element(), offer(obj), peek(), poll(), remove()
Eigene Methoden	clone(), zus.: s.u.	clone(), zus.: s.u.	clone(), getFirst(), getLast(), removeFirst(), removeLast()

# Übersicht

Klasse	HashSet	TreeSet
Interface(s)	Set	SortedSet
Collection-Methoden	wie oben	
List/Queue		
Set-Methoden	keine	
SortedSet-Methoden		comparator(), first(), last(), headSet(elem), tailSet(elem), subset(fromElem,toElem),
Eigene Methoden	clone()	clone()

# Übersicht

Klasse	ArrayList	Vector
Interface(s)	List	List
Interface-Methoden	S.O.	
Eigene Methoden	clone(), ensureCapacity(), removeRange( from,to), trimToSize()	clone(), capacity(), copyInto(Object[]), addElement(e), setElementAt(obj,ind), insertElementAt(obj,ind), elementAt(ind), elements(), ensureCapacity(), setSize(n), trimToSize(), firstElement(), lastElement(), indexOf(obj,ind), lastIndexOf(obj,ind), removeAllElements(), removeElement(obj), removeRange(from, to), removeElementAt(ind), toString()

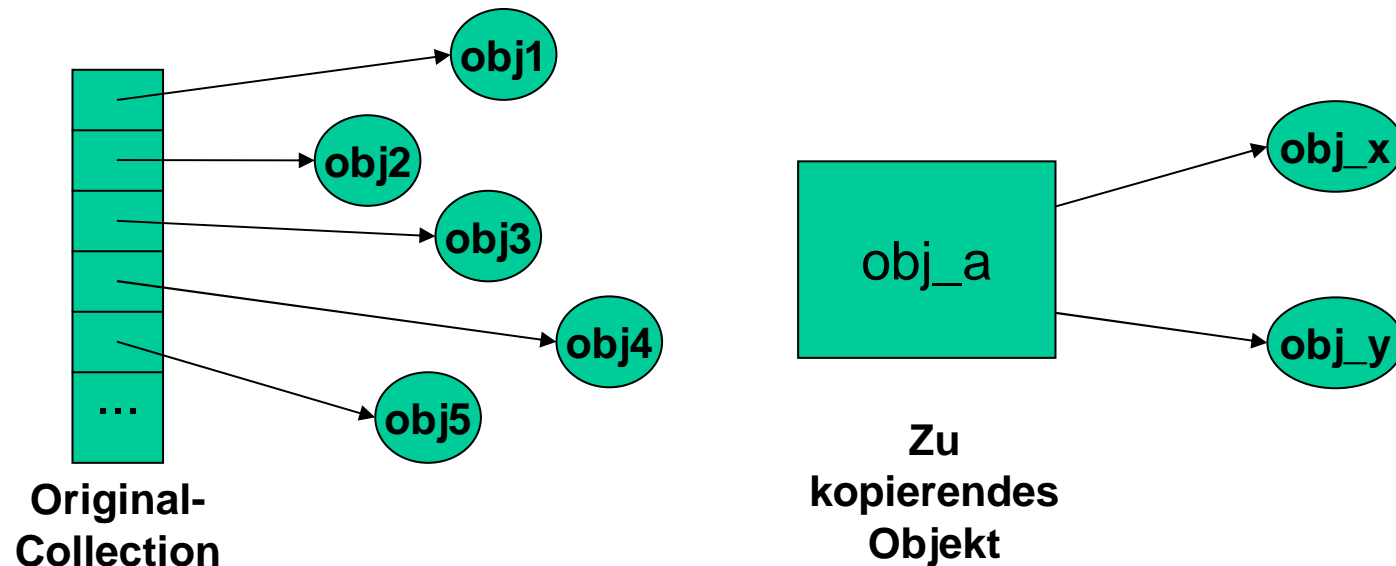
# **Programmierung II**

## **Thema 9: Collections, Teil 2**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

# Kopieren von Objekten

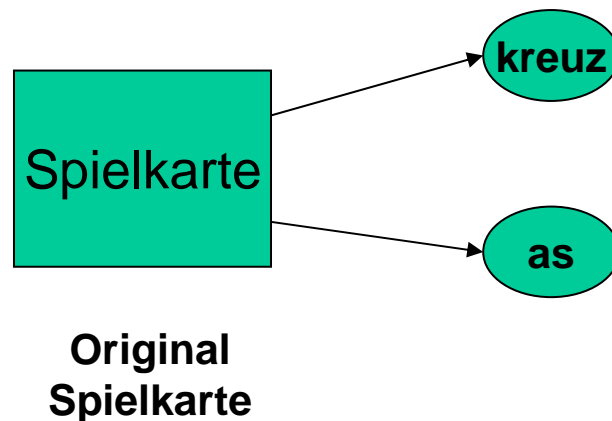
- In der Programmierpraxis müssen gelegentlich Objekte oder Collections kopiert werden.
- Fragen & Antworten:
  - Wie macht man das? → mit einer clone() - Methode
  - Wie wird dabei mit referenzierten Objekten umgegangen?  
→ „shallow copy“ oder „deep copy“





# Kopieren von Objekten

- **Aktion 1:**
  - Entwickeln Sie in der Klasse Spielkarte eine clone()-Methode, die eine Spielkarte kopiert (kloniert).
- **Lösungsansatz**
  - Wenn die clone()-Methode eines Spielkartenobjektes aufgerufen wird, wird eine neue Spielkarte erzeugt – new Spielkarte().
  - Der neuen Spielkarte werden die Attribute der alten Spielkarte zugewiesen – dies erfolgt bereits im Konstruktor.
  - Die neue Spielkarte wird zurückgegeben.

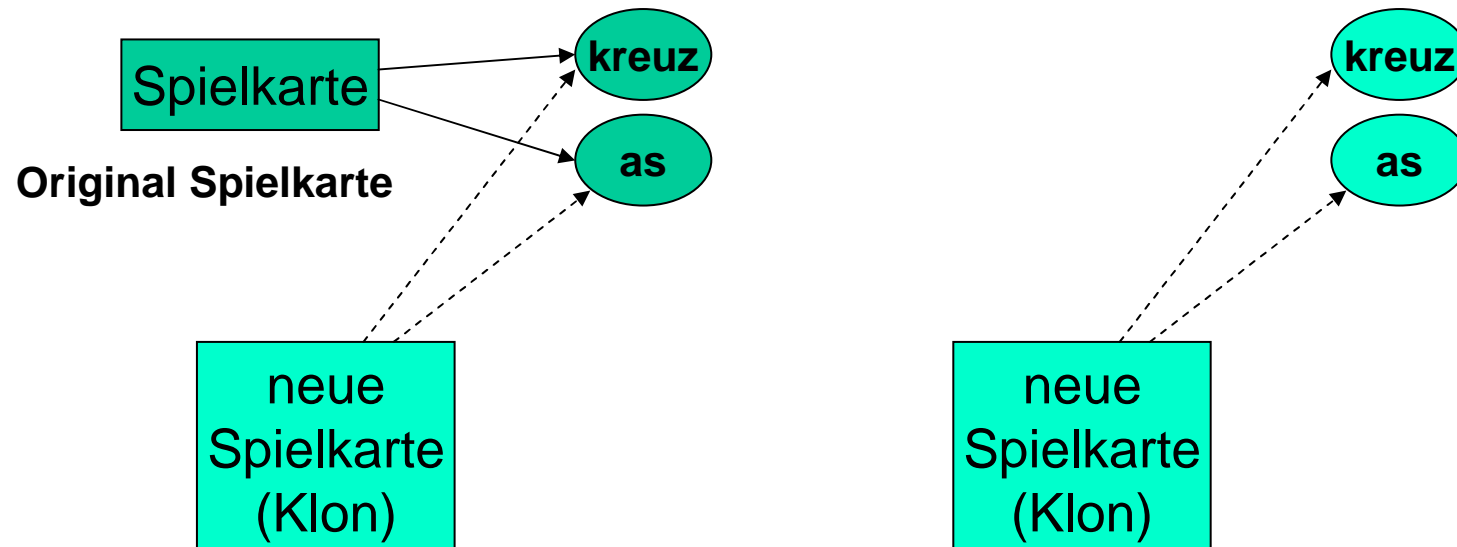


# Kopieren von Objekten

- Lösung, Schritt 1, die clone()-Methode:

```
public Spielkarte clone(){  
    Spielkarte newCard = new Spielkarte(this.getFarbe(), this.getWert());  
    return newCard;  
}
```

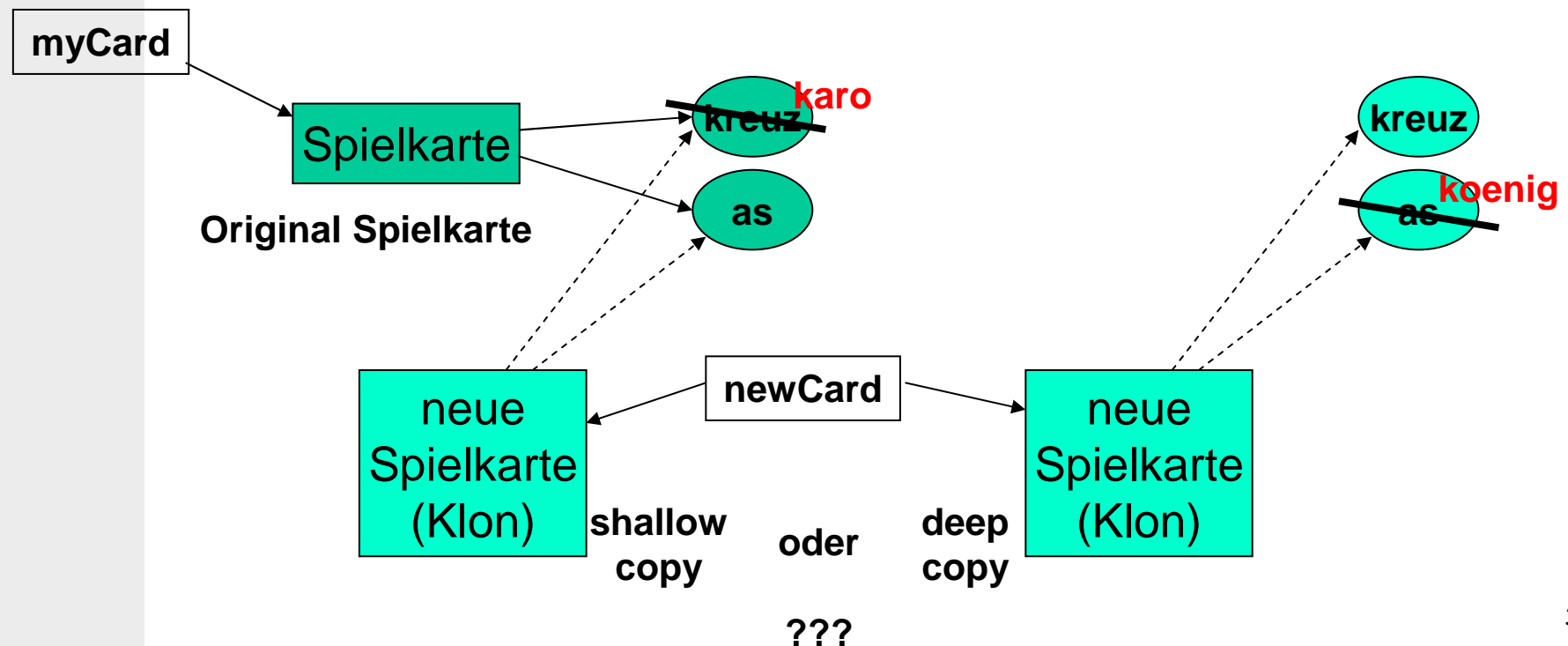
- Was passiert mit den Attributen?  
→ Es gibt zwei Möglichkeiten!



1. Möglichkeit (shallow copy / flache Kopie
2. Möglichkeit (deep copy / tiefe Kopie) <sup>4</sup>

# Kopieren von Objekten

- Lösung, Schritt 2, Test, Überlegung:
  - Wenn nach der Klonierung ein Attribut der Original Spielkarte geändert wird, müsste sich das
    - im Fall der shallow copy auch auf die neue Spielkarte auswirken
    - im Fall der deep copy nicht auf die neue Spielkarte auswirken.
  - Analog: Wenn ein Attribut der neuen Karte geändert wird...



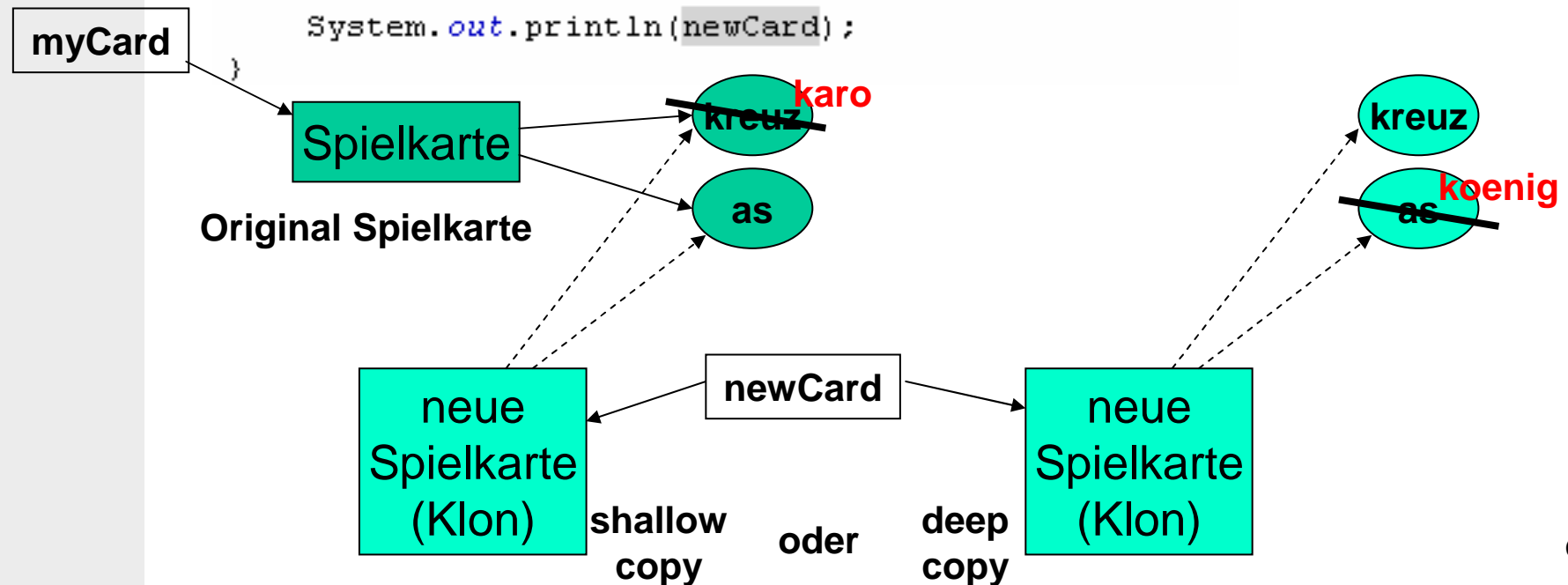
# Kopieren von Objekten

- Lösung, Schritt 2, Erweiterung von TestSpielkarte
  - Anlegen einer Test-Methode testClone()

```

public void testClone() {
    Spielkarte newCard = myCard.clone();
    assertTrue(newCard.getFarbe().equals("kreuz"));
    assertTrue(newCard.getWert().equals("as"));
    myCard.setFarbe("karo");
    newCard.setWert("koenig");
    System.out.println(myCard);
    System.out.println(newCard);
}

```



# Kopieren von Objekten

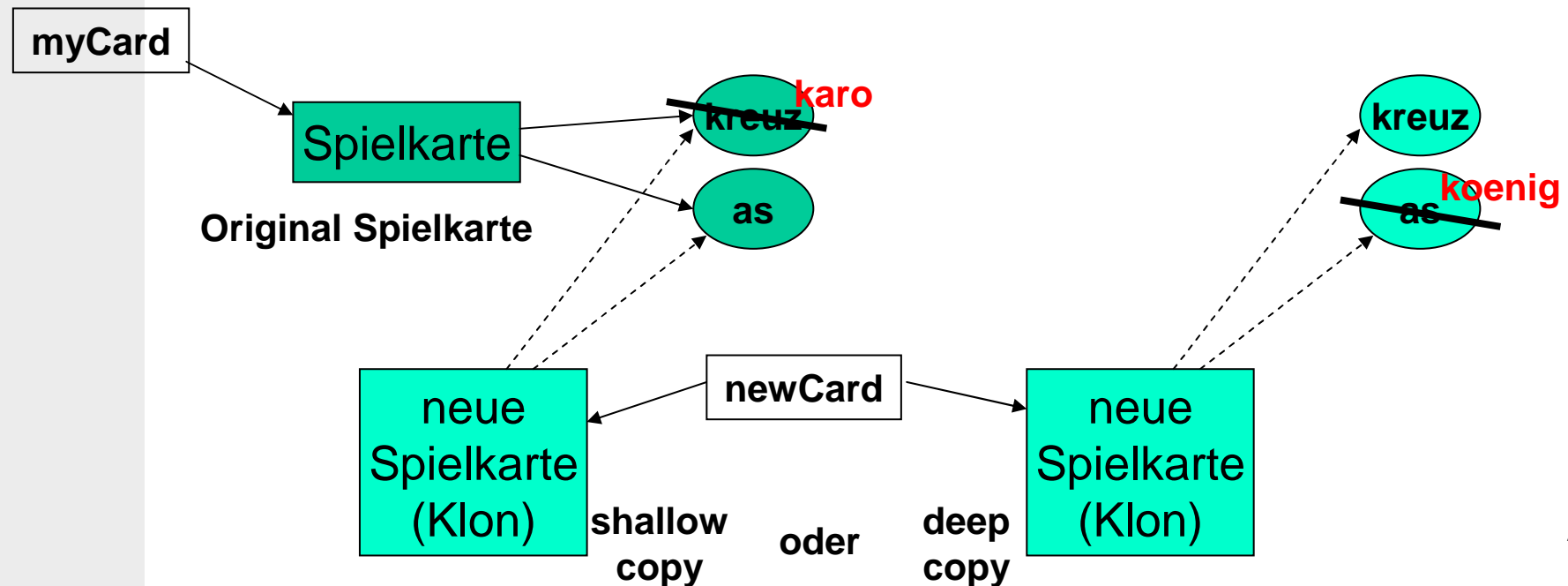
- Lösung, Schritt 2, Testergebnis

```
Runs: 5/5      Errors: 0      Failures: 0

Problems  Javadoc  Declaration  de.fh_lu.o2s.cardgames.TestSpielkarte [Runner: JUnit 3]

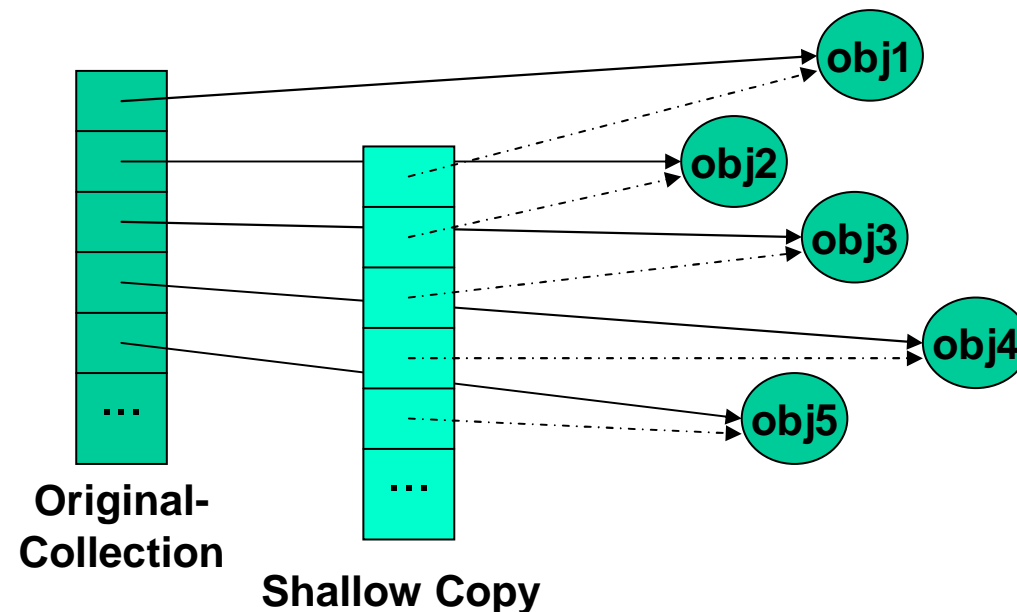
<terminated> TestSpielkarte [JUnit] C:\Programme
Spielkarte mit Farbe karo und Wert as
Spielkarte mit Farbe kreuz und Wert koenig
```

- Beobachtung: Die Attribute sind unabhängig → Deep copy



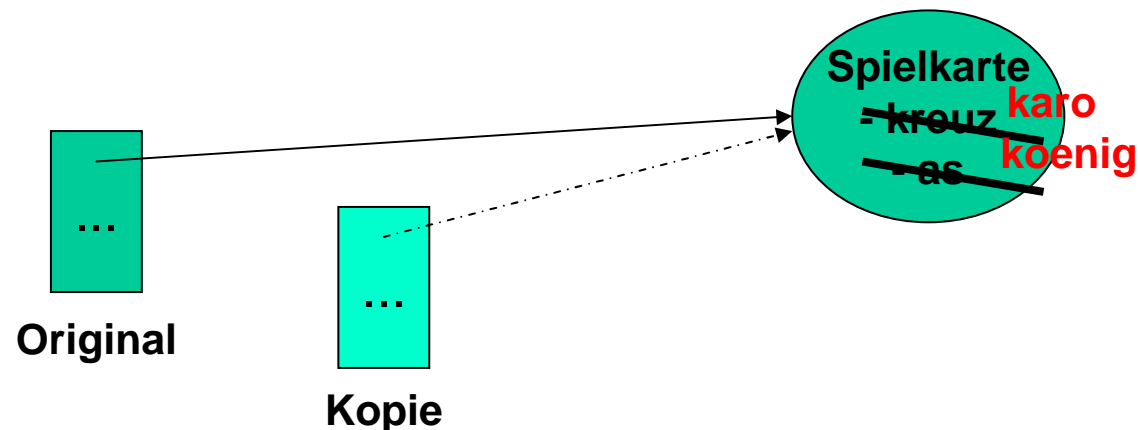
# Kopieren von Collections

- Anmerkung:
  - Bei der Kopie von Strings handelt es sich um einen Sonderfall
- Viele Collection-Klassen implementieren die Methode `clone()`.
  - Diese erzeugt normalerweise eine „shallow copy“ (flache Kopie)
    - des Collection-Objekts und
    - der Zeiger auf die gesammelten Objekte
    - aber nicht der Objekte selbst.



# Kopieren von Collections

- Aktion 2: Erweitern Sie Ihre AppColl, indem Sie
  - die dort verwendete Collection klonieren und
  - testen, ob es sich um eine shallow copy oder eine deep copy handelt.
- Lösungsansatz:
  - In AppColl wird zurzeit ein Vektor verwendet, der u.a. eine Spielkarte enthält.
  - Wir ändern die Attribute der Spielkarte einmal über den Original-Vektor, einmal über die Kopie und geben beide aus.



# Kopieren von Collections

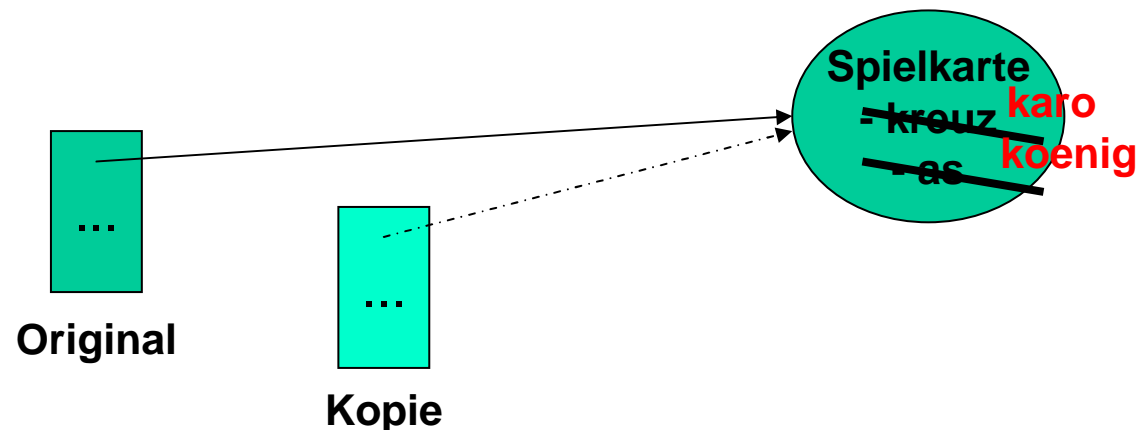
- Lösung:

```
Vector coll = new Vector();  
Spielkarte card1 = new Spielkarte("kreuz", "as");  
coll.add(card1);
```

...

```
Vector newColl = (Vector) coll.clone();  
(Spielkarte) coll.get(0).setFarbe("karo");  
(Spielkarte) newColl.get(0).setWert("koenig");
```

- Anmerkung: Die cast-Vorgänge sind nötig, weil
  - die clone()-Methode und die get()-Methoden von Vector alle den Rückgabetyt Object haben





# Kopieren von Collections

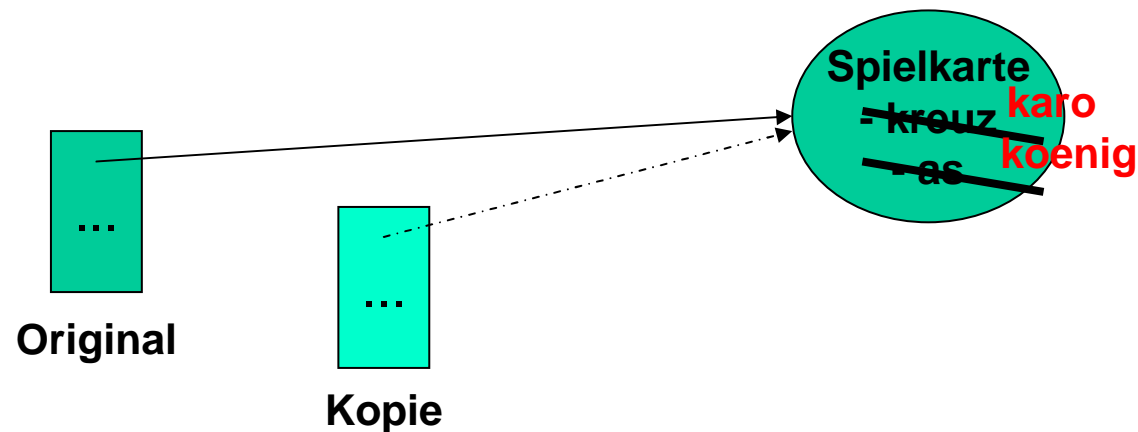
- Ergebnis:



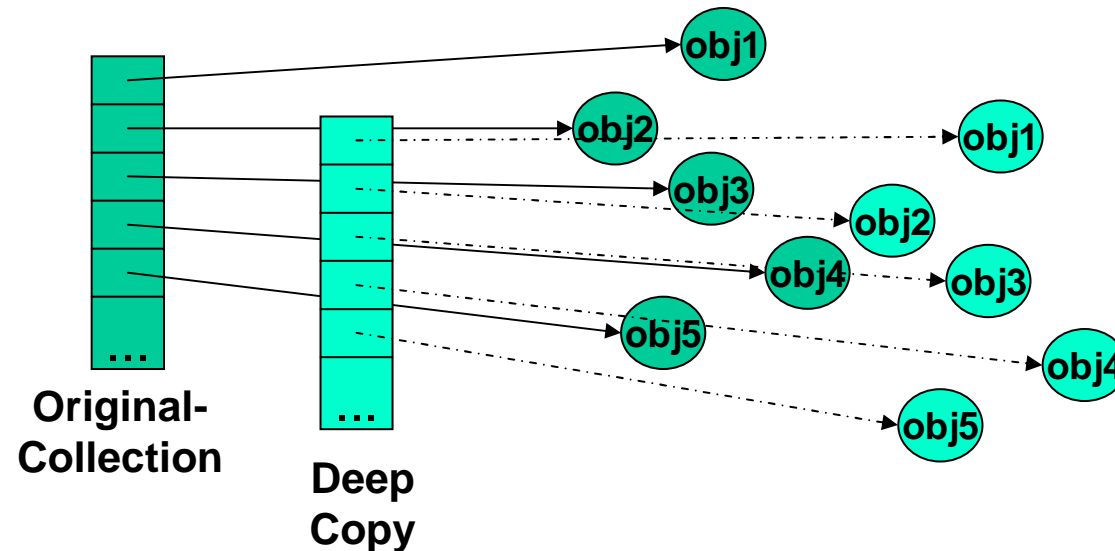
```
<terminated> AppColl [Java Application] C:\Programme\Java\jre6\bin\
Spielkarte mit Farbe karo und Wert koenig
```

- Beobachtung:

- Beide Änderungen beziehen sich auf dieselbe Spielkarte, d.h. es wurde tatsächlich eine shallow copy erzeugt.



# Deep Copy



- Von einer „deep copy“ (tiefen Kopie) spricht man, wenn auch die referenzierten Objekte kopiert werden.
    - Dies würde erfordern, dass auch alle enthaltenen Objekte klonierbar sind.
    - Da dies im Allgemeinen nicht vorausgesetzt werden kann, ist shallow copy das Standardverhalten.
    - Ausnahme: String-Attribute.
- ➔ Übungen: DeepCopyVector

# Cloneable

- Standard-Methode clone():
  - Ist in der Klasse Object vorhanden, dort aber protected,
  - kann also von eigenen Klassen nicht genutzt werden.
- Eigene Methode clone():
  - Eine Klasse, deren Objekte kopierbar sein sollen, muss die Methode clone() selbst implementieren.
  - Dann darf sie von sich sagen, sie sei **Cloneable**
- Interface „Cloneable“:
  - Das Interface „Cloneable“ liefert keine abstrakte Methode,
  - clone() ist ja bereits von Object spezifiziert.
  - Es ist ein reines Anzeigeinterface (Indikator-Interface)
  - Eine Klasse soll von sich sagen, sie implementiert Cloneable, wenn sie clone() implementiert.
  - Wenn die Klasse lügt, können wir aber auch nichts machen.

# Enumeration

- Außer dem Iterator kennt `Vector` auch
  - die Methode `elements()`, die eine `Enumeration` (Aufzählung) zurückgibt.
- Mit einer `(java.util.)Enumeration` kann
  - vollkommen analog zu einem Iterator
  - die `Collection` (der `Vector`) einmal durchlaufen werden.
- Eine `Enumeration` ist ein Interface analog zu `Iterator`, nur älter. Es kennt die Methoden
  - `hasMoreElements()`
  - `nextElement()`
- Diese werden gleich verwendet wie `hasNext()` / `next()`:

```
Enumeration enu = coll.elements();  
while (enu.hasMoreElements()) {  
    System.out.println(enu.nextElement());  
}
```

# Assoziative Speicher

- Speicherung von Key / Value-Paaren
  - Beispiel aus PHP

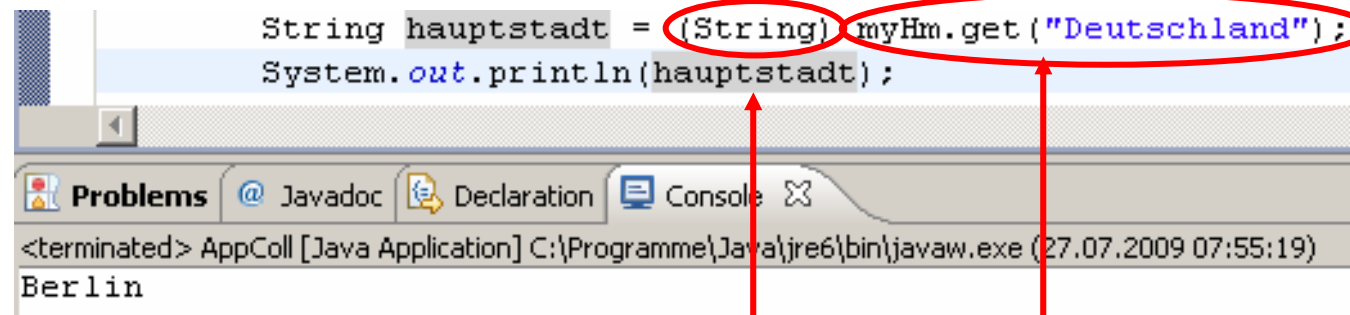
```
$hauptstadt["Deutschland"] = "Berlin";  
$hauptstadt["Belgien"] = "Brüssel";  
$hauptstadt["Niederlande"] = "Den Haag";
```
  - Es handelt sich um Zuordnungen Objekt1 → Objekt2
  - Man kann dies als Erweiterung von Arrays betrachten.
- Aktion 3: Verwenden Sie eine (`java.util.`)`HashMap`, um
  - solche key/value-Paare auch mit einem Java-Programm zu speichern.
  - Speichern Sie die folgenden (Objekt-)Zuordnungen:
    - „Deutschland“ → „Berlin“
    - „Stapel“ (String) → neue Spielkarte „kreuz as“
    - „FH LU“ (String) → neuer Student „Gerd Müller“
  - Lesen Sie anschließend den Wert der zu „Deutschland“ gehört und geben Sie diesen auf der Konsole aus.

# Assoziative Speicher

- Lösung, Schritt 1:
  - HashMap importieren (aus java.util) und anlegen,
  - Zuordnungen (key/value-Paare) in der HashMap speichern.

```
HashMap myHm = new HashMap();  
myHm.put("Deutschland", "Berlin");  
myHm.put("Stapel", new Spielkarte("kreuz", "as"));  
myHm.put("FH LU", new Student("Gerd Müller", "654321"));
```

- Lösung, Schritt 2: Wert auslesen und ausgeben



- Beobachtung:
  - Das Auslesen ist ganz einfach, wenn man den „key“ kennt.
  - Die HashMap kann beliebige Objekte speichern, deshalb muss beim Auslesen gecastet werden.

# Assoziative Speicher

- Aktion 4:
  - Lesen Sie die HashMap aus, auch wenn Sie die „key“s nicht schon vorher kennen.
- Lösungsansatz:
  - Die Methode `keySet()` liefert ein Set mit den vorhandenen keys,
  - das mit einer Schleife (z.B. mit einem Iterator) durchlaufen werden kann.
- Lösung (ausführliche Version):

```
Set kSet = myHm.keySet();  
Iterator iter = kSet.iterator();  
while (iter.hasNext()){  
    Object key = iter.next();  
    Object value = myHm.get(key);  
    System.out.println(key.toString() + " --> " + value.toString());  
}
```

- Anmerkung:
  - Der Rückgabewert von `keySet()` ist vom Interface-Typ `Set`. Dieser Typ muss also noch importiert werden.

# Assoziative Speicher

- Lösung (kompakte Version):

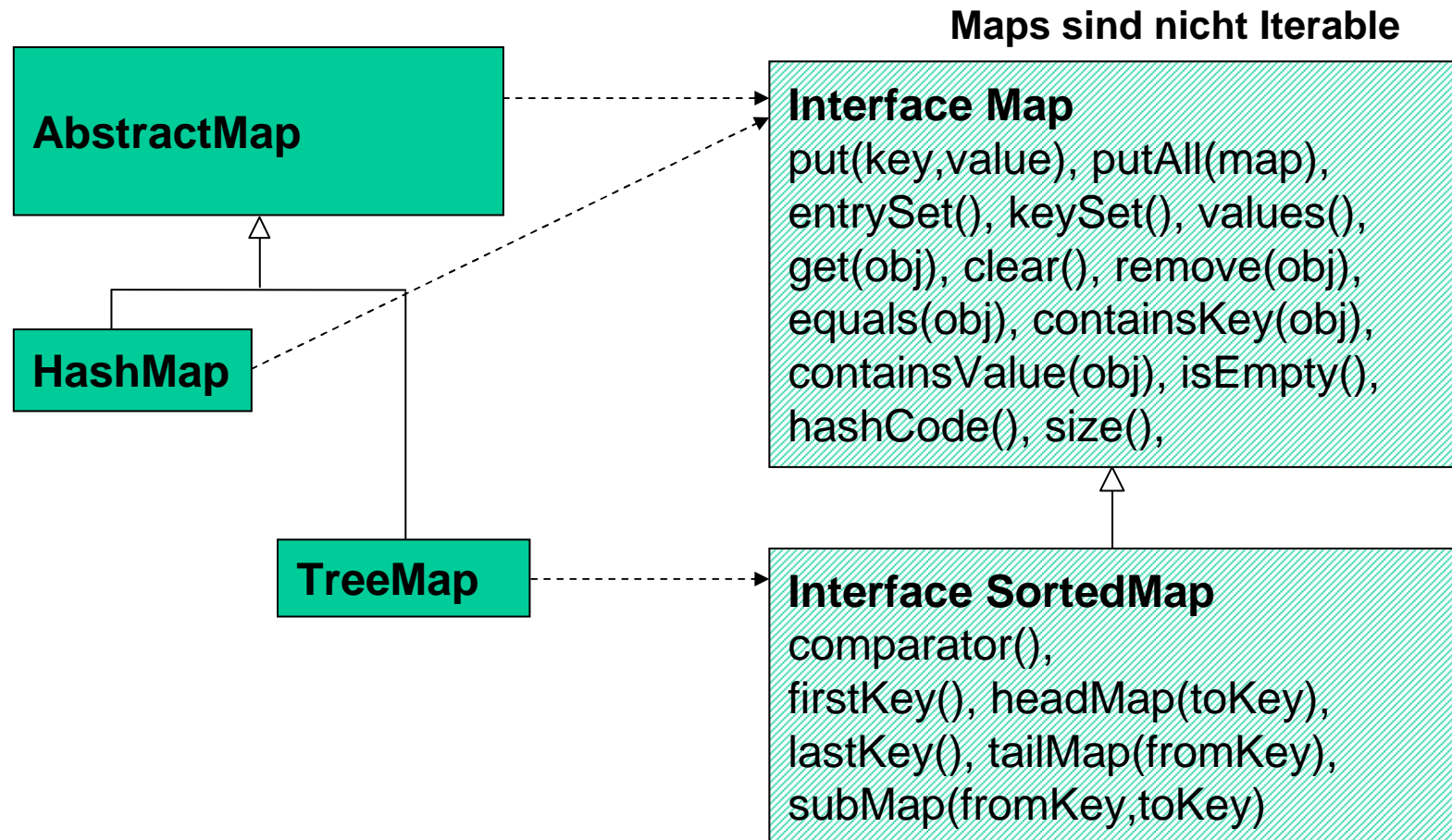
```
Iterator iter = myHm.keySet().iterator();  
while (iter.hasNext()){  
    Object key = iter.next();  
    System.out.println(key.toString() + " --> " + myHm.get(key).toString());  
}
```

- Anmerkung:

- Bei dieser Lösung muss der Datentyp `Set` nicht importiert werden, weil er nur implizit auftritt.



- Klassen und Interfaces



# Was ist drin?

- Wenn man nichts dazu sagt, kann man in einer Collection immer beliebige Objekte speichern:

```
HashSet hs = new HashSet();
```

```
Vector vec = new Vector();
```

- In einer HashMap kann man dann sogar beliebige key- und value-Objekte benutzen:

```
HashMap hm = new HashMap
```

```
hm.put(new Spielkarte("kreuz", "as"), 11)
```

```
Hm.put("naechste", new Spielkarte("kreuz", "as"))
```

- In der Praxis braucht man aber meistens Mengen oder Listen aus bestimmten Objekten, z.B.

```
Spielkarte[] kartenA = new Spielkarte[52];
```

- So eine Einschränkung funktioniert bei Collections auch, man spricht von „Generics“.

# Menge von Spielkarten

- Aktion 5: Erzeugen Sie in `AppColl` ein `HashSet`,
  - das nur Spielkarten aufnehmen kann.
  - Speichern Sie darin drei neue Spielkarten.
  - Versuchen Sie, einen Studenten zu speichern.

- Lösung, Schritt 1: `HashSet` erzeugen

```
HashSet<Spielkarte> cardSet = new HashSet<Spielkarte>();
```

- Beobachtung:
  - Beim Erzeugen der Collection wird in Spitzklammern angegeben, was für Daten gespeichert werden sollen, hier `<Spielkarte>`.
  - Diese Angabe „gehört“ noch zur Klasse, deshalb steht dies noch vor den Klammern.
  - Auch auf der linken Seite der Deklaration muss dies angegeben werden.

# Menge von Spielkarten

- Lösung, Schritt 2:

```
HashSet<Spielkarte> cardSet = new HashSet<Spielkarte>();  
cardSet.add(new Spielkarte("karo", "koenig"));  
cardSet.add(new Spielkarte("herz", "dame"));  
cardSet.add(new Spielkarte("pik", "sieben"));  
cardSet.add(new Student("Gerd Müller", "654321"));
```

The method add(Spielkarte) in the type HashSet<Spielkarte> is not applicable for the arguments (Student)

- Beobachtung:

- Passende Objekte werden gespeichert wie immer.
- Objekte, die nicht zum angegebenen Datentyp (Spielkarte) passen, können nicht gespeichert werden.
- Die Zeile mit dem Studenten muss deshalb auskommentiert werden.

```
//      cardSet.add(new Student("Gerd Müller", "654321"));
```

# Liste von Spielkarten

- Aktion 6: Erzeugen Sie
  - eine Liste und
  - kopieren Sie alle Karten aus Ihrem `HashSet` in diese Liste.
  - Geben Sie dabei außerdem jede Karte und deren Punktwert auf der Konsole aus.
- Lösungsansatz:
  - Wir erzeugen eine `ArrayList` und
  - kopieren die Karten mit einer `foreach`-Schleife aus dem `HashSet` in die `ArrayList`.

# Liste von Spielkarten

- Lösung, Schritt 1, nicht-eingeschränkt:

```
ArrayList cardList = new ArrayList();  
for (Object sk : cardSet) {  
    cardList.add(sk);  
    System.out.println(sk + ", Punktwert: " +  
                        ((Spielkarte) sk).getPunktWert());  
}
```

- Beobachtung:
  - Einige Compiler-Warnungen wegen unklarer Typsituation
  - Wenn wir Object als Datentyp in der foreach-Schleife verwenden, dann
  - müssen wir casten, bevor wir getPunktWert() aufrufen können.

Multiple markers at this line

- ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized
- ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized

# Liste von Spielkarten

- Lösung, Schritt 2, eingeschränkt:

```
ArrayList<Spielkarte> cardList = new ArrayList<Spielkarte>();  
for (Spielkarte sk : cardSet){  
    cardList.add(sk);  
    System.out.println(sk + ", Punktwert: " +  
                        sk.getPunktwert());  
}
```

- Beobachtung:

- Keine Compiler-Warnungen mehr. Alle Datentypen sind klar.
- Wir können `Spielkarte` als Datentyp in der `foreach`-Schleife verwenden und
- sparen uns das casten.

# Liste von Spielkarten

- Aktion 7: Kopieren Sie
  - noch einmal alle Karten aus Ihrem `HashSet` in Ihre Liste.
  - Benutzen Sie dafür dieses Mal einen Iterator.

- Lösung, 1. Versuch, nicht-eingeschränkter Iterator:

```
Iterator iter2 = cardSet.iterator();  
while (iter2.hasNext()) {  
    Spielkarte sk2 = iter2.next();  
    cardList.add(sk2);  
}
```

- Beobachtung:
  - Fehler, weil `iter.next()` ein `Object` liefert und keine `Spielkarte`.

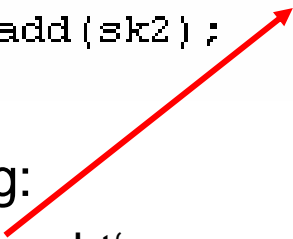
|Type mismatch: cannot convert from Object to Spielkarte



# Liste von Spielkarten

- Lösung, 2. Versuch, nicht-eingeschränkter Iterator:

```
Iterator iter2 = cardSet.iterator();  
while (iter2.hasNext()) {  
    Spielkarte sk2 = (Spielkarte) iter2.next();  
    cardList.add(sk2);  
}
```



- Beobachtung:
  - Mit casten geht's.
  - Einige Compiler-Warnungen bleiben noch.

# Liste von Spielkarten

- Lösung, 3. Versuch, eingeschränkter Iterator:

```
Iterator<Spielkarte> iter2 = cardSet.iterator();  
while (iter2.hasNext()){  
    Spielkarte sk2 = iter2.next();  
    cardList.add(sk2);  
}
```

- Beobachtung:

- Wenn wir zu der eingeschränkten Collection auch einen eingeschränkten Iterator verwenden, dann wird's am einfachsten.
- Die `foreach`-Schleife nimmt uns die Arbeit ab (vgl. Aktion 6).

# Zusammenfassung

- Eine Collection ist „generisch“, wenn
  - man sie für alle Arten von Datentypen brauchen und
  - jeweils entsprechend einschränken kann.
- Der Compiler hilft uns mit Warnungen.
- Zu generischen Collections gehören auch generische Iteratoren.
- Alle Collections des Java Collection Framework können generisch verwendet werden.

# Eigene generische Collections

- Wenn man eine eigene generische Collection entwickeln möchte, muss man bei der Klassendefinition
  - eine Typ-Variable angeben, z.B.

```
public class BetterBucket<T> extends HashSet<T>{  
    ...  
}
```
- Innerhalb der Klassendefinition
  - wird diese Typ-Variable verwendet wie ein normaler Datentyp, z.B.

```
public boolean add(T obj){  
    super.add(obj);  
}
```
- Bei der Erzeugung der Collection
  - wird die Typ-Variable durch einen konkreten Datentyp ersetzt, z.B.

```
BetterBucket<Spielkarte> cardBB =  
    new BetterBucket<Spielkarte>()
```

# Eigene generische Collections

- Aktion 8:
  - Machen Sie Ihren `BetterBucket` generisch und
  - benutzen Sie ihn in `AppColl` eingeschränkt auf Spielkarten, indem Sie die drei Spielkarten Ihres `HashSet` auch in Ihren `BetterBucket` kopieren.
- Lösungsansatz:
  - `BetterBucket` generisch machen wie oben angegeben.
  - Methoden von `BetterBucket` prüfen, ob diese angepasst werden müssen (zurzeit nur `toString()`).
  - `BetterBucket` in `AppColl` verwenden genau wie die `ArrayList`.

# Eigene generische Collections

- Lösung, Schritt 1, BetterBucket:

```
public class BetterBucket<T> extends HashSet<T> {  
    public String toString() {  
        String s = new String();  
        for(T obj : this){  
            s += obj.toString() + "\n";  
        }  
        return s;  
    }  
}
```

- Beobachtung:

- In der foreach-Schleife von `toString()` werden alle Elemente des `BetterBucket` durchlaufen.
- Diese sind jetzt nicht mehr allgemein vom Typ `Object`,
- sondern vom angegebenen Datentyp `T`.
- Deshalb sollte in der foreach-Schleife „Object“ durch „T“ ersetzt werden.

# Eigene generische Collections

- Lösung, Schritt 2, AppColl:

```
BetterBucket<Spielkarte> cardBB = new BetterBucket<Spielkarte>();  
for (Spielkarte sk : cardSet){  
    cardBB.add(sk);  
    System.out.println(cardBB);  
}
```

- Beobachtung:

- BetterBucket kann jetzt auch generisch bzw. eingeschränkt exakt genauso verwendet werden wie HashSet, ArrayList oder Vector.

# Von generischen Collections erben

- Wenn man eine eigene Collection entwickeln will, die
  - ➔ nicht generisch ist, aber
  - ➔ trotzdem nur Daten eines bestimmten Typs akzeptieren soll,
- dann kann
  - von einer generischen Collection geerbt werden,
  - indem der Datentyp fest angegeben wird.
- Bsp.:

```
public class StudentenStapel extends Vector<Student>{  
    ...  
}
```
- Anmerkung:
  - Weitere Beispiele finden sich in den Übungen.





# Von generischen Collections erben

- Aktion 9: Entwickeln Sie einen „StudentenStapel“, d.h.
  - eine Collection, die nur Studenten enthalten darf und
  - die Stapelmethoden `push(...)`, `pop()`, `top()` und `isEmpty()` besitzt.
- Lösungsansatz:
  - Wir erzeugen den StudentenStapel nicht-generisch,
  - erben von einer Collection und programmieren die genannten Methoden noch dazu.
  - Als Oberklasse können wir `Vector` wählen, dann bekommen wir jede Menge mächtige Methoden, die wir nutzen können.

# Von generischen Collections erben

- Lösung, Schritt 1, Rahmen der Klasse:

```
import java.util.Vector;
import de.fh_lu.o2s.personen.Student;

public class StudentenStapel extends Vector<Student> {
    public void push(Student newStud) {

    }
    public Student pop(){

    }
    public Student top(){

    }
    public boolean isEmpty(){

    }
}
```

# Von generischen Collections erben

- Lösung, Schritt 2, die Methoden:
  - Erst mal sehn was der Vector so kann → [java.sun.com/apis](http://java.sun.com/apis)
  - Wir nutzen...

boolean	<code><u>add</u>(<u>E</u> e)</code>
---------	-------------------------------------

Appends the specified element to the end of this Vector.

## **remove**

```
public E remove(int index)
```

Removes the element at the specified position in this Vector.

Shifts any subsequent elements to the left (subtracts one from

their indices). Returns the element that was removed from the

Vector.

# Von generischen Collections erben

- Lösung, Schritt 2, die Methoden:

- ... und machen draus:

```
public void push(Student newStud){  
    this.add(newStud);  
}  
public Student pop(){  
    return this.remove(this.size() - 1);  
}
```

- Außerdem:

```
public Student top(){  
    return this.get(this.size() - 1);  
}
```



# Von generischen Collections erben

- Lösung, Schritt 3, Rest der Methoden:
  - Vector hat bereits eine `isEmpty()`-Methode, diese könnten wir verwenden:

```
public boolean isEmpty(){  
    return super.isEmpty();  
}
```

- Andererseits können wir diese dann auch direkt erben, müssen sie also gar nicht selbst implementieren.



```
// public boolean isEmpty(){  
//     return super.isEmpty();  
// }
```



# Von generischen Collections erben

- Lösung, Schritt 4, Test:
  - zunächst `setUp()` und `testPush()`

```
import de.fh_lu.o2s.personen.Student;
import junit.framework.TestCase;

public class TestStudentenStapel extends TestCase {
    StudentenStapel testStapel;
    Student testStudent;
    public void setUp(){
        testStapel = new StudentenStapel();
        testStudent = new Student("Gerd Müller", "654321");
    }
    public void testPush(){
        int stackSize = testStapel.size();
        testStapel.push(testStudent);
        assertFalse(testStapel.isEmpty());
        assertTrue(testStapel.size() == stackSize + 1);
    }
}
```

Runs: 1/1       Errors: 0       Failures: 0

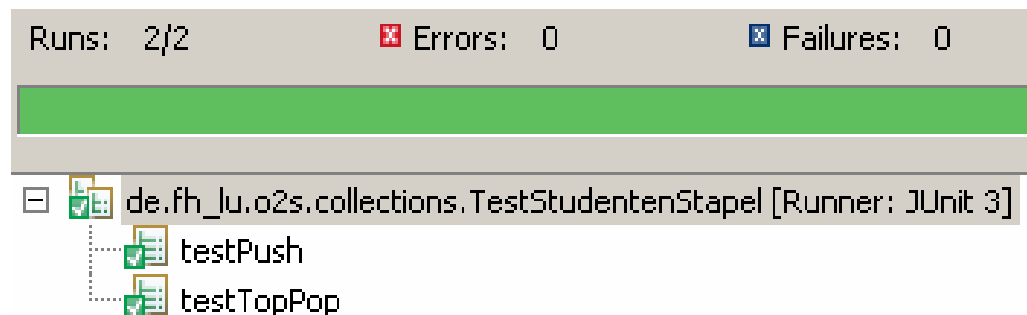
  de.fh\_lu.o2s.collections.TestStudentenStapel [Runner: JUnit 3]  
.....  
 testPush



# Von generischen Collections erben

- Lösung, Schritt 5, Test für pop() und top():

```
public void testTopPop() {  
    int stackSize = testStapel.size();  
    Student newStud1 = new Student("Carola Fingerle", "654322");  
    testStapel.push(newStud1);  
    Student newStud2 = testStapel.top();  
    Student newStud3 = testStapel.pop();  
    assertTrue(newStud2.getName().equals(newStud1.getName()));  
    assertTrue(newStud3.getName().equals(newStud1.getName()));  
    assertTrue(testStapel.size() == stackSize);  
}  
}
```



# Anmerkungen

- Grundsätzlich sollten alle generischen Collections typisiert werden, z.B.

```
HashSet<Spielkarte> hs;
```

- Dies gilt auch, wenn verschiedene Objekttypen zugelassen werden sollen, z.B.

```
Vector<Object> vec;
```

- Gelegentlich kann es auch sinnvoll sein, anzugeben, dass der Typ derzeit nicht bekannt ist, z.B.

```
ArrayList<?> list;
```



# Zusammenfassung

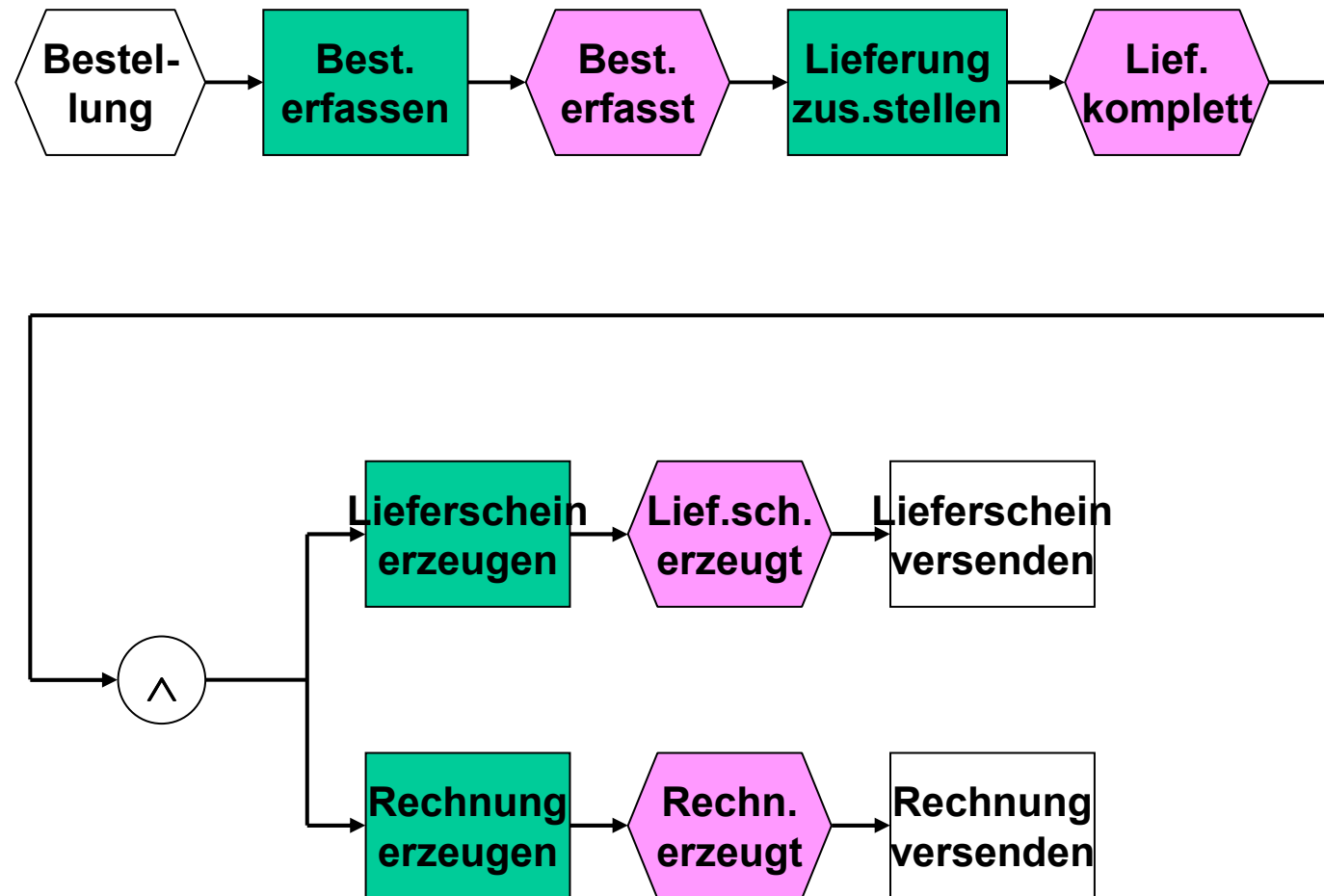
- Alle Collections des Java Collection Frameworks sind generisch, können und sollten typisiert werden.
- Wenn man einen Iterator im Zusammenhang mit einer typisierten Collection verwendet, sollte auch der Iterator typisiert werden.
- Wenn eine eigene generische Klassen entwickelt werden soll, muss bei der Klassendefinition eine Typ-Variable angegeben werden.
- Von einer generischen Klasse kann auf zweierlei Art geerbt werden:
  - So dass die Unterklasse ebenfalls generisch ist, dann muss für die Unterklasse eine Typ-Variable angegeben werden oder
  - so dass die Unterklasse nicht generisch ist, dann muss für die Oberklasse ein konkreter Typ angegeben werden.

# **Programmierung II**

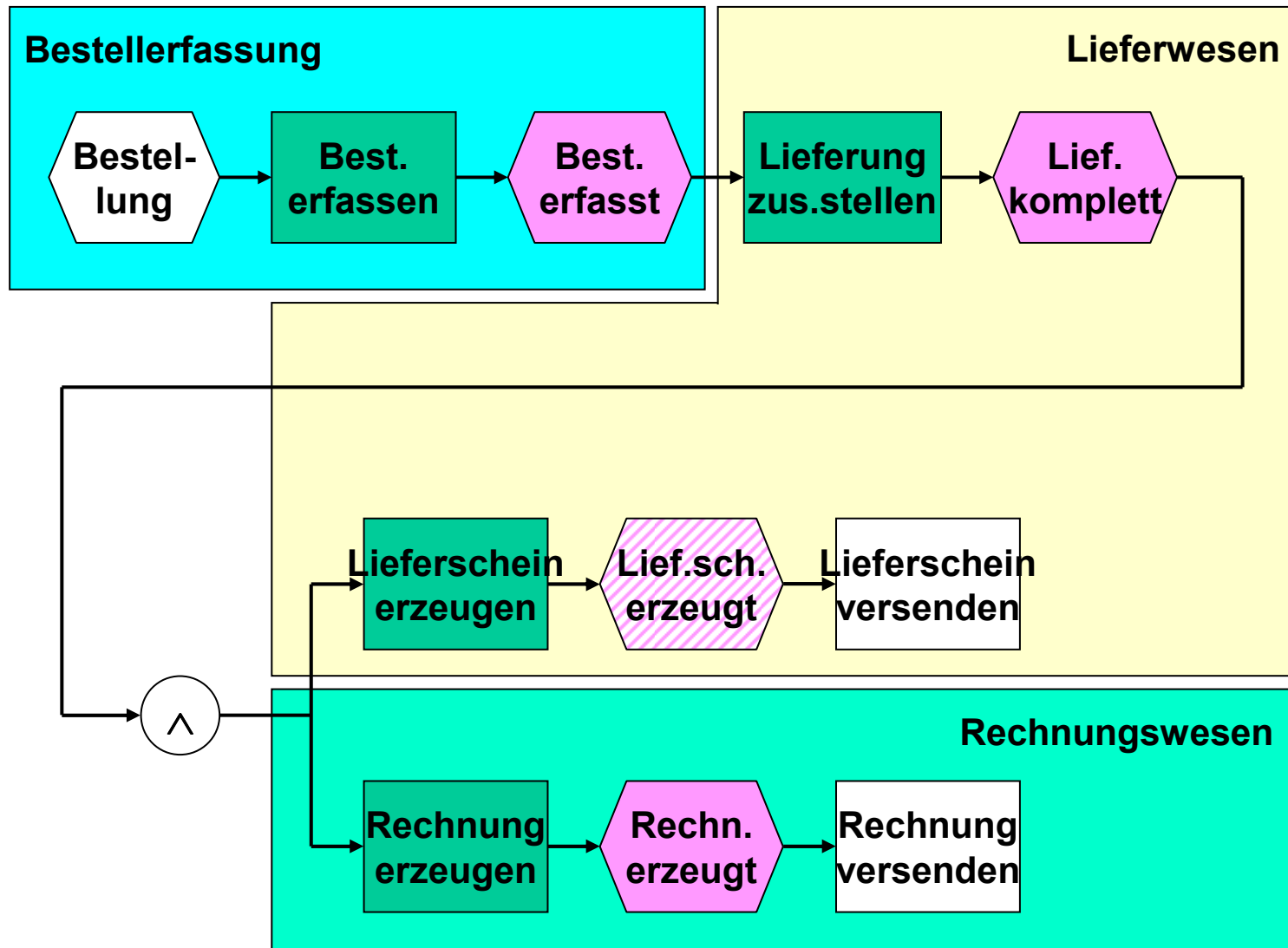
## **Thema 10: Fallstudie Bestellung**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

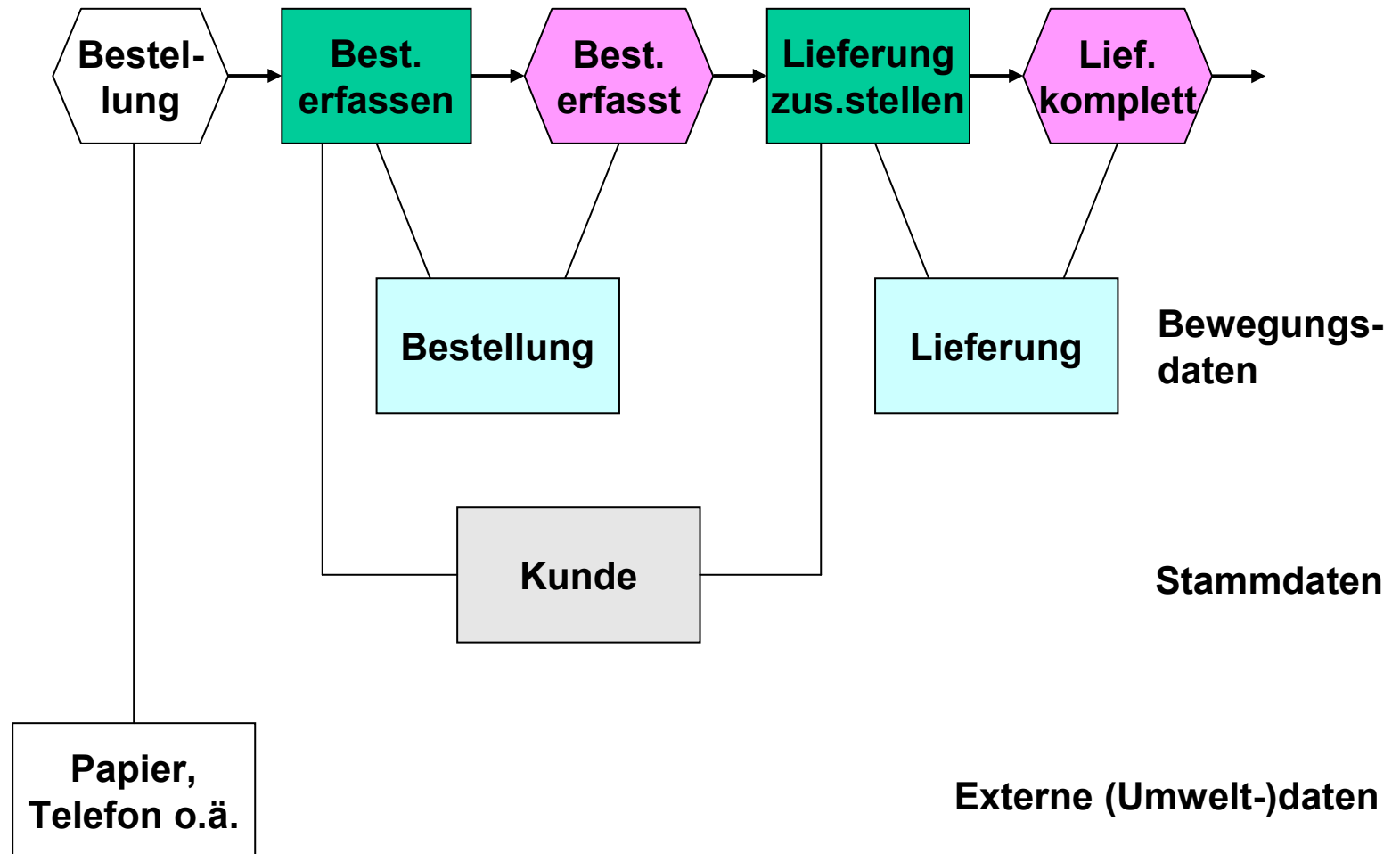
# Prozesssicht



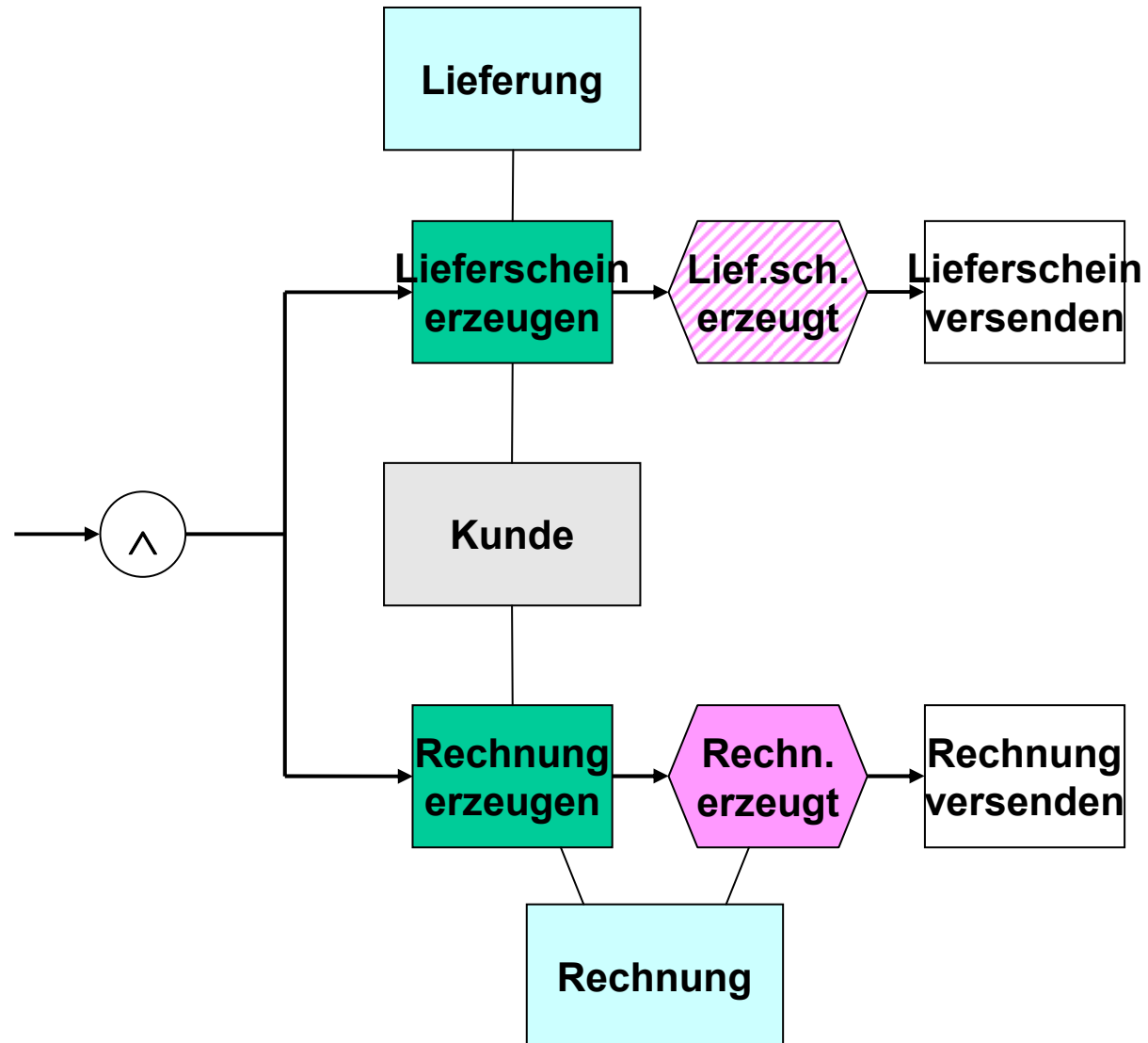
# Prozesssicht



# Daten

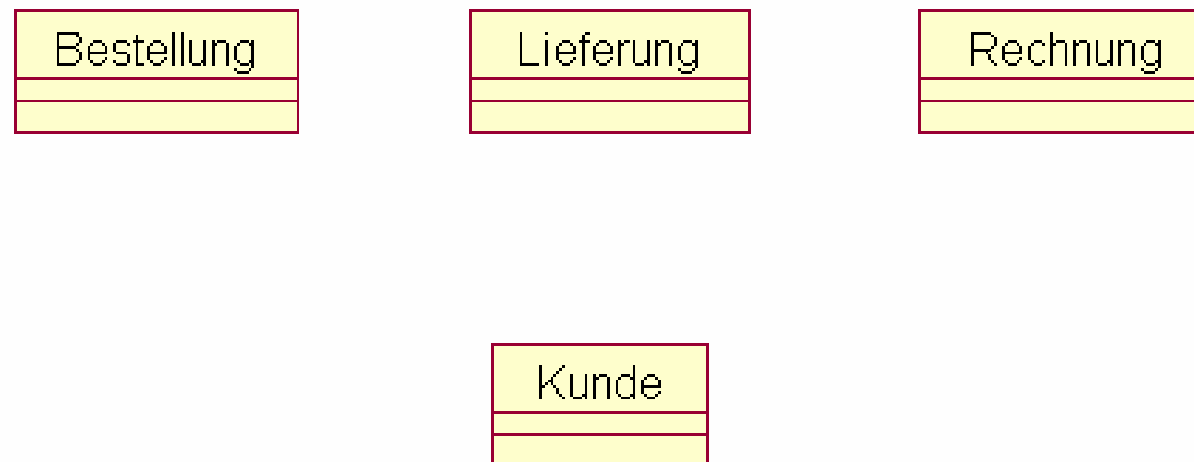


# Daten



# Klassendiagramm

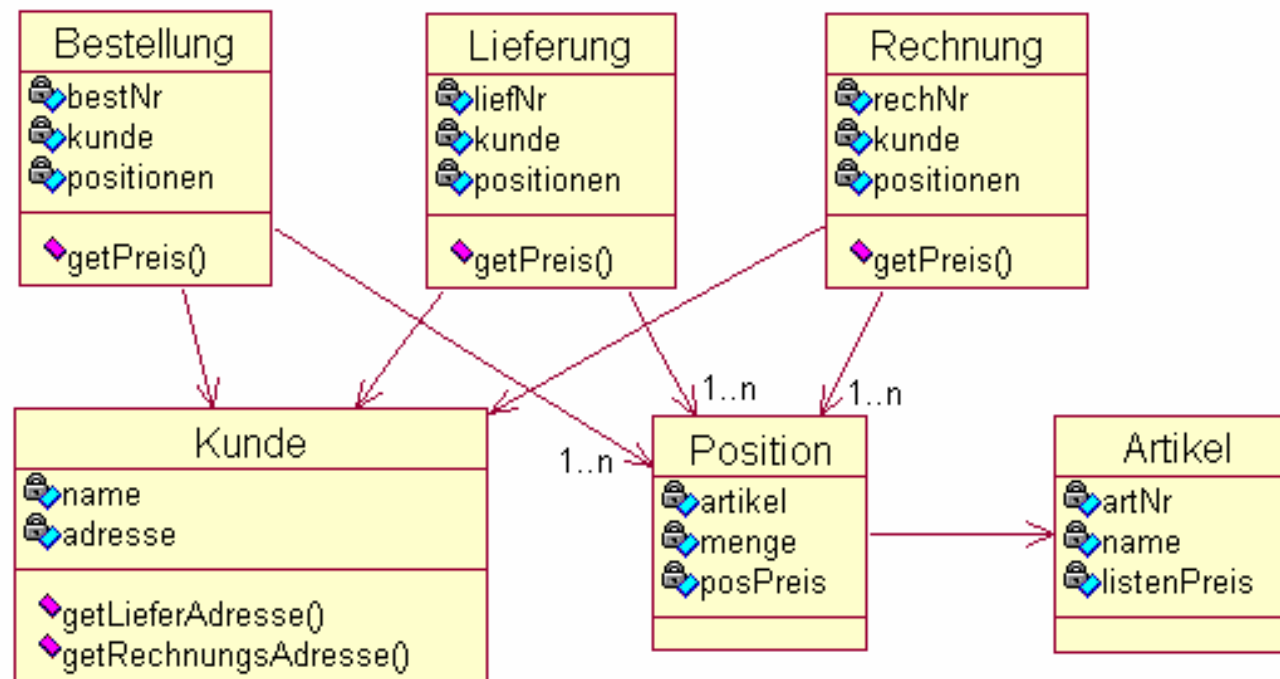
- Erste Näherung



➔ Attribute fehlen noch

# Klassendiagramm

- Zweite Näherung

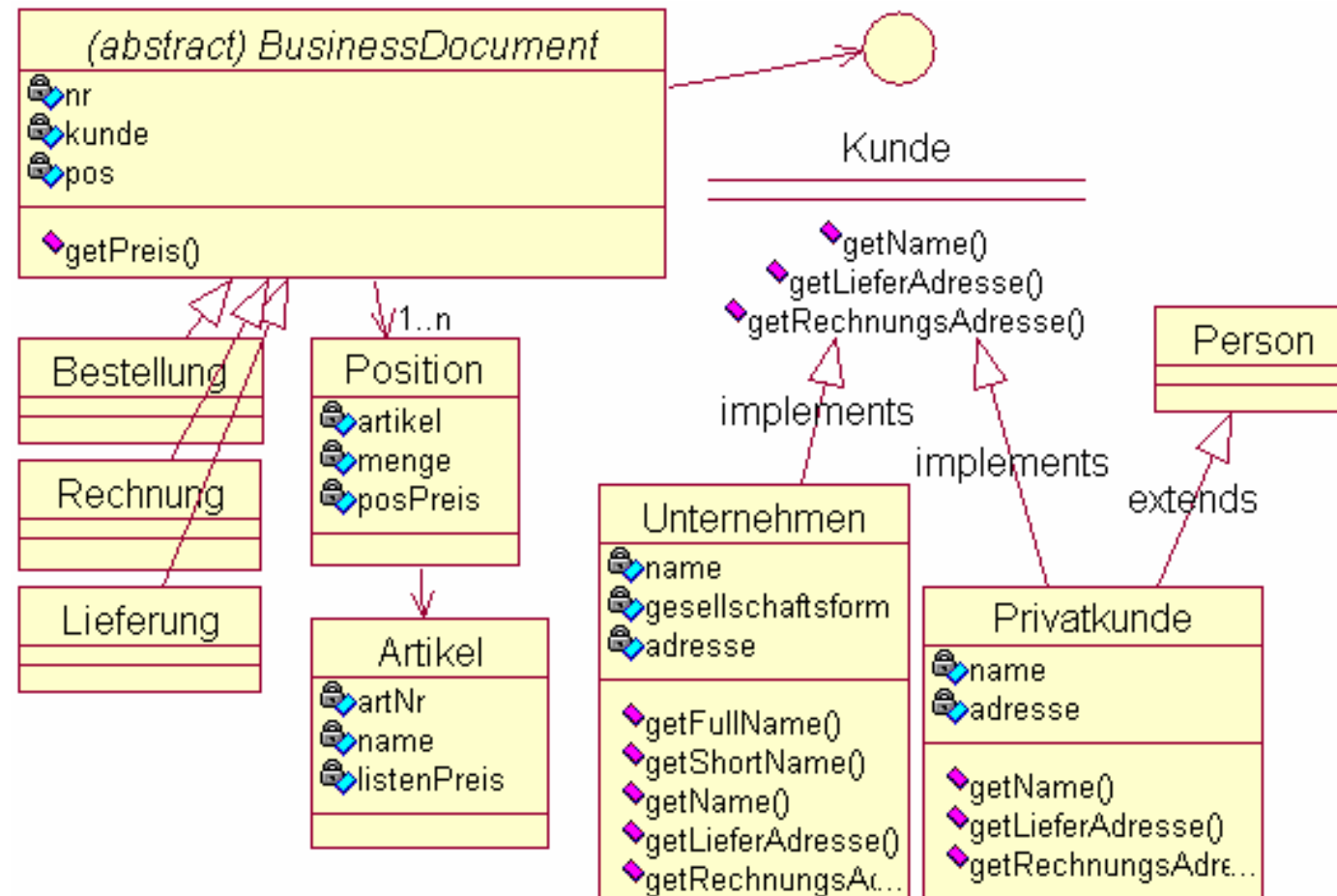


- Beobachtung:
  - Bestellung, Lieferung, Rechnung sehen ähnlich aus
  - Kunde könnte Privatkunde oder Geschäftskunde sein

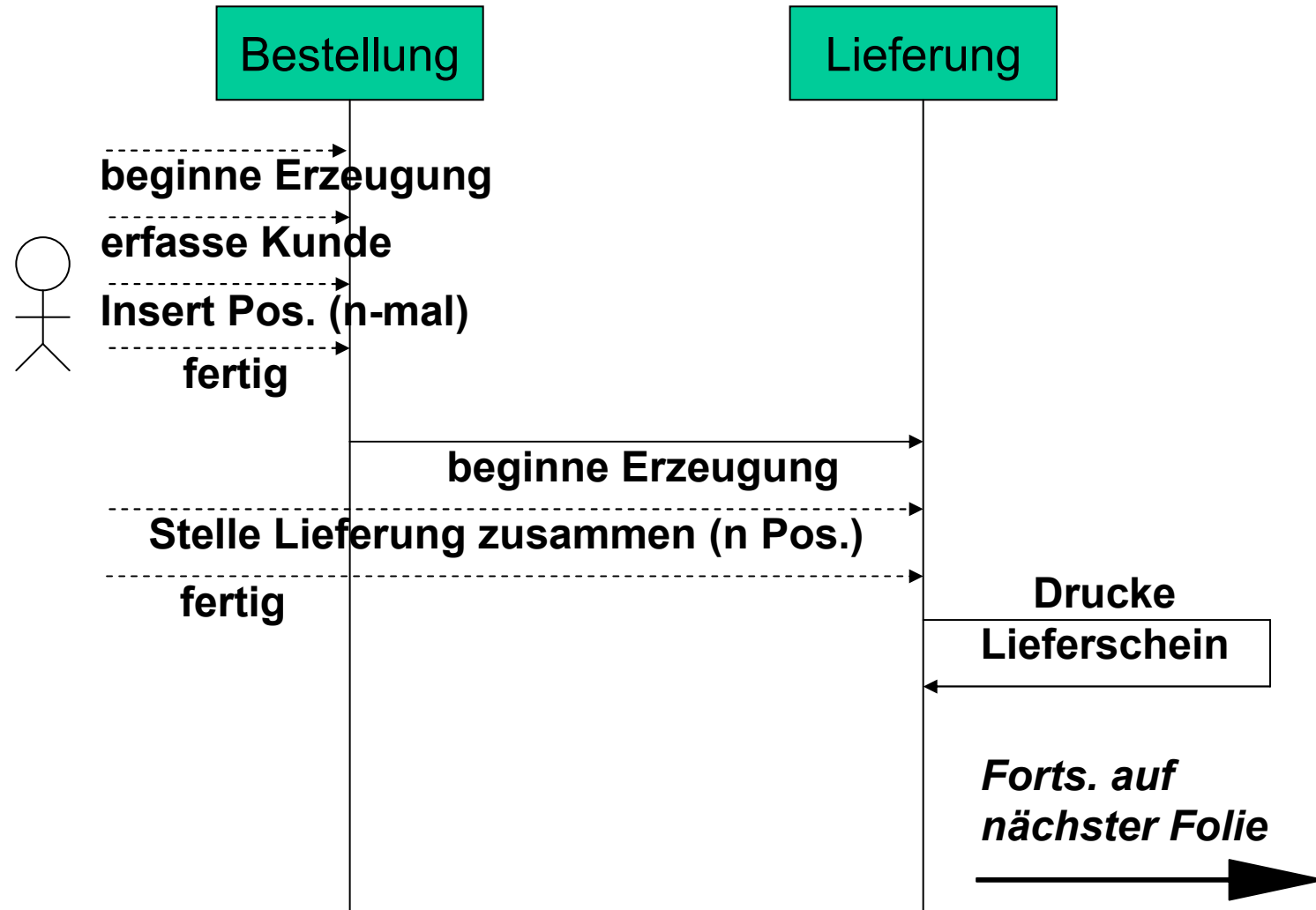


# Klassendiagramm

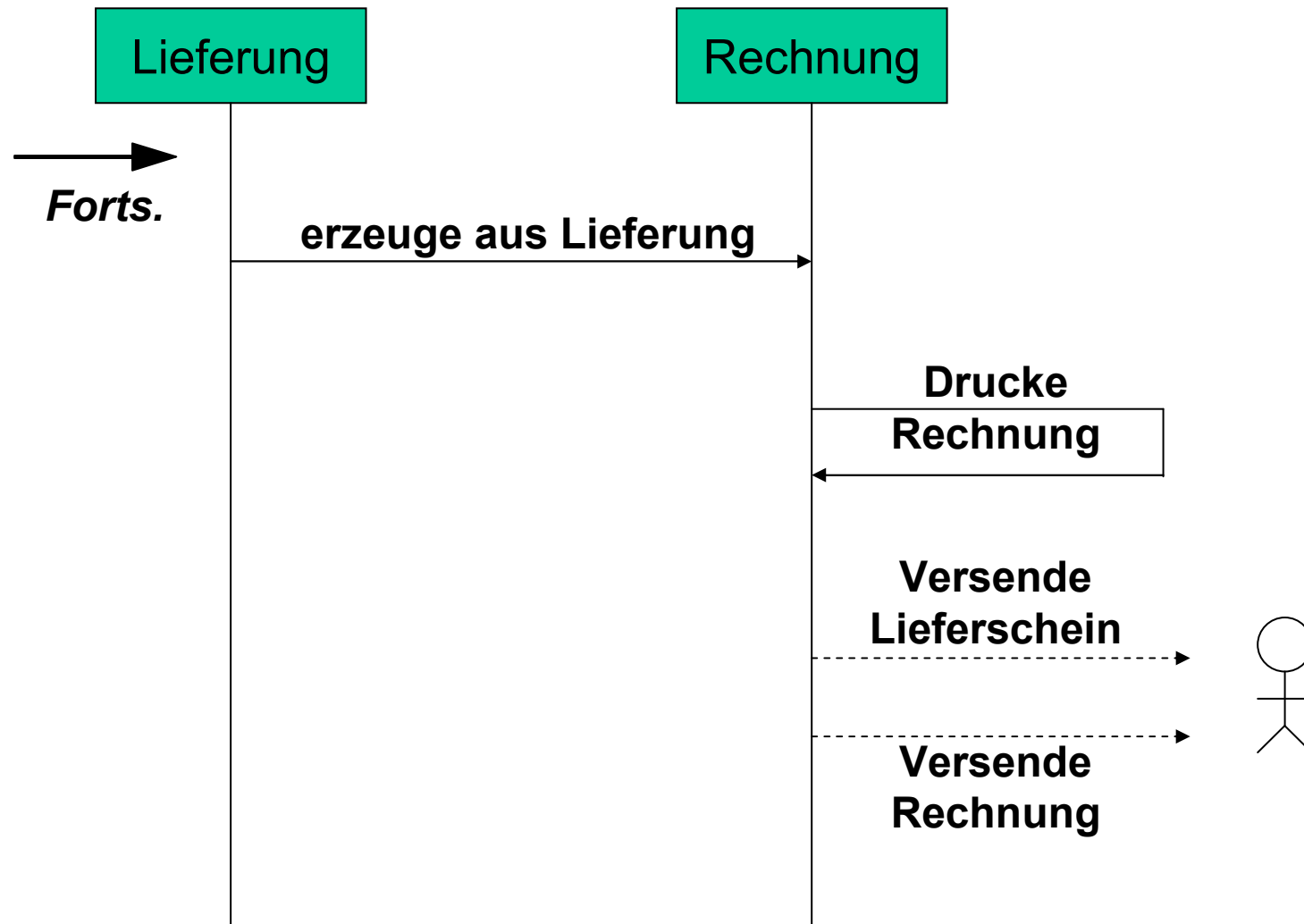
## ■ Verschönerung



# Workflow, Teil I



# Workflow, Teil II



# Nötige Methoden

- Bestellung
  - beginne Erzeugung
  - insert Position
  - erfasse Kunde
  - fertig: Löst Zusammenstellung der Lieferung aus
- Lieferung
  - erzeuge mit unfertigen Positionen
  - setze die einzelnen Positionen auf „fertig“
  - fertig (weitere Positionen können ggfs. nicht geliefert werden)
  - drucke Lieferschein
- Rechnung
  - erzeuge aus Lieferung
  - drucke Rechnung
- Anmerkung:
  - versenden von Lieferschein und Rechnung geht dann von Hand. <sup>11</sup>

# Umsetzung

- Hier wird nur beschrieben, was nicht bereits mit Standardfunktionalität (get-, set-, etc.) geht:
- Bestellung
  - beginne Erzeugung: Konstruktor „Bestellung()“ mit automatisch hochgezählter Nr. Dafür Klassenvariable neueNr, die jeweils eine neue nr bereit halten soll.
  - insert Position: Methode „add()“ – kann in BusinessDocument gepackt werden.
  - fertig:
    - Zusätzliches Attribut „fertig“.
    - Ist am Anfang false und kann nur einmal auf true gesetzt werden, andernfalls Exception!
    - Wenn „fertig“ auf true gesetzt wird, wird automatisch die Zusammenstellung der Lieferung ausgelöst.
    - Dazu neue Methode „triggerLieferung()“

- Lieferung
  - erzeuge mit unfertigen Positionen: Konstruktor mit Argument Bestellung.
  - Attribut „bestellung“ als Hinweis auf den Auslöser.
  - setze die einzelnen Positionen auf „fertig“: Klasse LieferPosition als Unterklasse von Position mit zusätzlichem Attribut „fertig“ und get-/set-Methoden
  - fertig:
    - Attribut „fertig“ in der Lieferung mit get-/set-Methoden.
    - Ist am Anfang false und kann nur einmal auf true gesetzt werden, andernfalls Exception!
    - Wenn „fertig“ auf true gesetzt wird, wird automatisch der Ausdruck des Lieferscheins und die Erzeugung der Rechnung ausgelöst.
    - Dazu neue Methode „triggerRechnung()“ und Methode „druckeLieferschein()“

# Umsetzung

- Rechnung
  - erzeuge aus Lieferung: Konstruktor mit Argument Lieferung
  - Attribut „lieferung“ als Hinweis auf den Auslöser.
  - drucke Rechnung: Methode „druckeRechnung()“
- Außerdem
  - toString() Methode für Artikel und Kunden

- Klasse AppBestellung
- Schritt 1: Bestellung erfassen:
  - Bestellungsobjekt anlegen,
  - Kunde erfassen,
  - beliebig oft Positionen erfassen,
  - Bestellung auf fertig setzen.
- Schritt 2: Lieferung zusammenstellen:
  - Lieferungsobjekt wird automatisch erzeugt, die Positionen müssen aber manuell auf „fertig“ gesetzt werden.
  - Lieferung auf fertig setzen. Der Rest geht automatisch
- Schritt 3: Manuelle Nacharbeiten
  - Lieferschein und Rechnung versenden



# AppBestellung

```
public void doSomething() {  
    try{  
        Bestellung myBest = this.erfasseBestellung();  
        Lieferung myLief = myBest.setFertig(true);  
        this.stelleLieferungZusammen(myLief);  
@SuppressWarnings("unused")  
        Rechnung myRech = myLief.setFertig(true);  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
    System.out.println("doSomething() beendet");  
}
```

# AppBestellung

```
public Bestellung erfasseBestellung(){
    Bestellung myBest = new Bestellung();
    Adresse myKundenAdresse = new Adresse(
        "Hans-Dumm-Str. 12", "13", "14151",
        "Darmstadt", "Deutschland");
    Kunde myKunde = new Privatkunde("Günter", "Semmler",
                                     myKundenAdresse);

    myBest.setKunde(myKunde);
    myBest.add(new Position(new Artikel("1", "Hammer", 8.50), 1));
    myBest.add(new Position(new Artikel("2", "Meißel", 6.50), 1));
    myBest.add(new Position(new Artikel("3", "Sichel", 5.80), 1));
    myBest.add(new Position(new Artikel("4", "Amboss", 18.00), 1));
    myBest.add(new Position(new Artikel("5", "Zange", 7.50), 1));
    myBest.add(new Position(new Artikel("6", "Rohr", 2.50), 10));
    myBest.add(new Position(new Artikel("7", "Pömpel", 1.80), 1));
    return myBest;
}
```

# AppBestellung

```
public void stelleLieferungZusammen(Lieferung myLief) {  
    for (LieferPosition myPos :  
        (LieferPosition[]) myLief.getPosA()) {  
        myPos.setFertig(true);  
    }  
}
```

# **Programmierung II**

## **Thema 11: Threads**

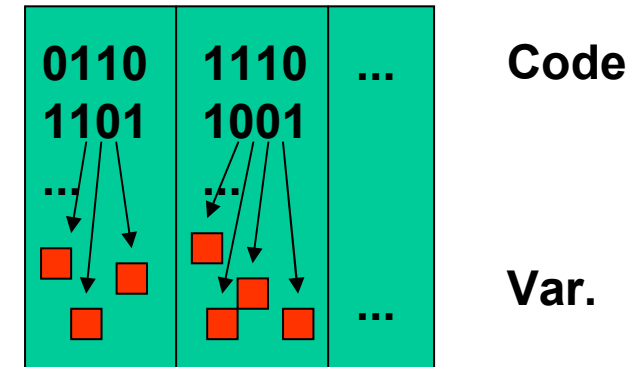
**Fachhochschule Ludwigshafen  
University of Applied Sciences**

# Prozesse und Threads

➔ Sind bekannt aus der Vorlesung Betriebssysteme:

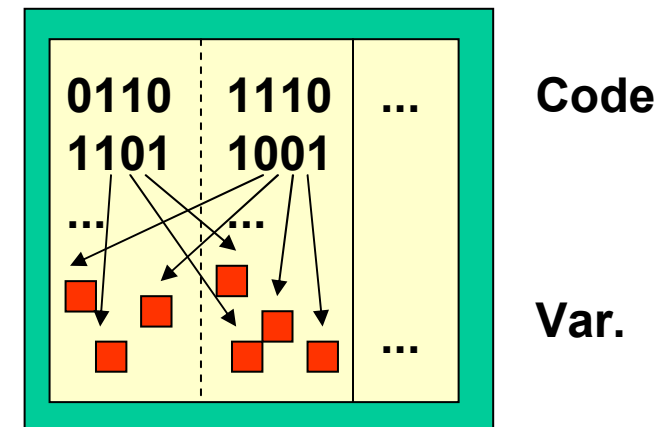
■ Prozesse:

- Getrennt laufende Programme, die vom Betriebssystem gesteuert werden (Systemressourcen erhalten),
- Haben keine gemeinsamen Variablen.



■ Threads (Fäden):

- Parallel laufende Programme oder Programmteile, die entweder vom Betriebssystem oder einem Hauptprogramm gesteuert werden,
- Können gemeinsame Variablen haben.



# Warum?

- Es gibt Programmteile, die logisch unabhängig von anderen und / oder kontinuierlich laufen sollen, z.B.

```
while (true){  
    myText.blink() ;  
}
```

- Parallelisierung für mehr Performance auf Mehrprozessor-/ Mehrkernsystemen:
  - Evtl. können unterschiedliche Threads auf unterschiedlichen Prozessoren / Kernen laufen.
  - Allerdings kann es sein, dass ein Thread „auf einen anderen warten“ muss, also pausieren muss, bis ein anderer Thread einen bestimmten Zustand erreicht hat.

# Threads in Java

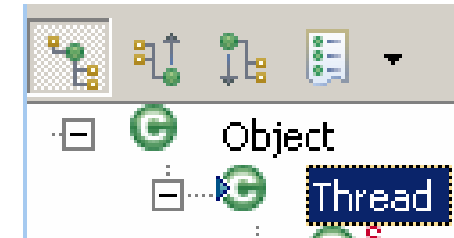
- Die Klasse Thread ist Subklasse von Object.

- Eigene Thread-Klassen können von Thread **erben**: ZaehlThread **is-a** Thread

```
public class ZaehlThread extends Thread {  
    int start, inkrement, sleeptime;  
    public ZaehlThread(int start, int inkrement,  
                        int sleeptime) {...}
```

- Ein Thread ist prinzipiell ein Programm.
- Der Programmcode steht in einer Methode run().

```
public void run() {  
    int counter = start;  
    while(counter < 1000) {  
        System.out.println(counter);  
        counter += inkrement;  
        try{sleep(sleeptime);} catch (Exception e) {}  
    }  
}
```



# Threads in Java

- Start des Threads:
  - Durch Aufruf der Methode `start()` wird implizit `run()` aufgerufen und als eigenständiger Thread ausgeführt.
  - Die Methode `run()` kann auch direkt aufgerufen werden, wird dann aber nicht als Thread ausgeführt.

```
public class AppZaehlung {  
    public static void main(String[] args) {  
        //Konstruktor:  
        //ZaehlThread(int start, int inkrement, int sleeptime)  
        ZaehlThread ungerade = new ZaehlThread(1, 2, 250);  
        ZaehlThread gerade   = new ZaehlThread(2, 2, 250);  
        ungerade.start();  
        gerade.start();  
    }  
}
```



# Experimente

- Beobachtung:
  - Die Threads `ungerade` und `gerade` laufen paralell.
  - Sie sind nicht streng abwechselnd dran, sondern eher zufällig.
  - ➔ Das Betriebssystem verwaltet, welcher Thread wann dran ist. In unserem Fall die Java VM.
- In `AppZaehlung` kann experimentiert werden, z.B. mit unterschiedlichen `sleeptimes`...
- Einem Thread kann gesagt werden, wann er die Kontrolle erstmal abgeben soll:

```
public void run() {  
    int counter = start;  
    while(counter < 1000) {  
        System.out.println(counter);  
        counter += inkrement;  
        yield();  
        try{sleep(sleeptime);} catch (Exception e) {}  
    }  
}
```

- Dadurch wird etwas Kontrolle über den Ablauf gewonnen.

# Klasse Thread

void run()	Programm
void start()	Start des Programms
static void yield()	Abgabe der Kontrolle
static void sleep(long) throws InterruptedException	Pause ohne unbedingt die Kontrolle abzugeben
void interrupt()	unterbricht einen Thread
boolean isInterrupted()	
boolean alive()	
void wait() throws InterruptedException	
void notify()	
InterruptedException	Tritt auf, wenn ein pausierender Thread unterbrochen wird.

# Interface Runnable

- Falls eine Anwendung bereits in einer Vererbungshierarchie steht, kann sie nicht zusätzlich von Thread erben.
- wenn sie trotzdem eine `run()`-Methode anbietet, kann sie von sich sagen, dass sie das Interface `Runnable` implementiert.

```
public class ZaehlRunnable implements Runnable {  
    int start, inkrement, sleeptime;  
    public ZaehlRunnable(int start, int inkrement,  
                          int sleeptime){...}  
  
    public void run() {  
        int counter = start;  
        while(counter < 1000) {  
            System.out.println(counter);  
            counter += inkrement;  
            Thread.yield();  
            try { Thread.sleep(sleeptime); } catch (Exception e) {}  
        }  
    }  
}
```

- Da `yield()` und `sleep()` static-Methoden von Thread sind, können sie mit Bezug auf die Klasse Thread aufgerufen werden.

# Interface Runnable

- Eine Anwendung, die `Runnable` implementiert, kann mit Hilfe eines `Standard-Threads` gestartet werden:

```
public class AppZaehlung1 {  
    public static void main(String[] args) {  
        //Konstruktor:  
        //ZaehlRunnable(int start, int inkrement, int sleeptime)  
        ZaehlThread ungerade = new ZaehlThread(1, 2, 25);  
        ZaehlRunnable gerade = new ZaehlRunnable(2, 2, 25);  
        ungerade.start();  
        new Thread(gerade).start();  
    }  
}
```

- Man sagt: Die (Runnable-)Anwendung ist „target“ des Threads.

# Weitere Möglichkeiten

- Ein Runnable kann auch seinen „eigenen“ Thread besitzen.
- und diesen bei Bedarf sogar sofort ausführen („Autostart“).

```
public class ZaehlRunnable1 implements Runnable {  
    Thread myThread;  
    public ZaehlRunnable1(...) { ...  
        Thread myThread = new Thread(this);  
        myThread.start();  
    }  
}
```

- Ein Thread kann sich bei Bedarf ebenfalls sofort ausführen:

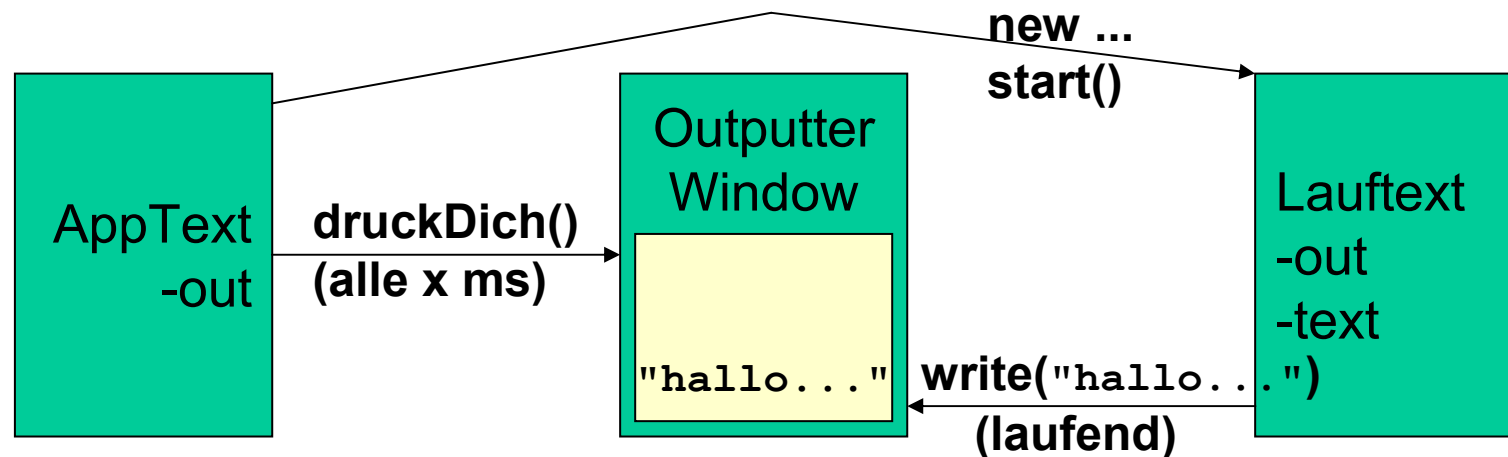
```
public class ZaehlThread1 extends Thread {  
    public ZaehlThread1(...) { ...  
        this.start();  
    }  
}
```

- Die Anwendung muss dann nur noch konstruieren:

```
public static void main(String[] args) {  
    ZaehlThread1 ungerade = new ZaehlThread1(1, 2, 25);  
    ZaehlRunnable1 gerade = new ZaehlRunnable1(2, 2, 25);  
    ...  
}
```

# Beispiel

- Ein Lauftext soll unabhängig vom Rest des Programms über den Bildschirm laufen.
- Konzept:
  - Es wird eine Anzeigeklasse `OutputterWindow` verwendet
  - Eine Instanz der Klasse `Lauftext` wird als Thread gestartet und schreibt die Ausgabe regelmäßig in das `OutputterWindow`.
  - Die Anwendung selbst druckt regelmäßig das `OutputterWindow` auf den Bildschirm.



# Runnable Lauftext, Teil I

```
public class Lauftext implements Runnable {  
    String text;  
    int pos = 0;  
    int inkrement = 2;  
    OutputterWindow out;  
    public Lauftext(OutputterWindow out, String str){  
        text = str;  
        this.out = out;  
    }  
    public void run() {  
        ...  
    }  
}
```

- OutputterWindow dient als Hilfsklasse:
  - overWrite(line,pos,text) schreibt den String text an Position pos in Zeile line des Textpuffers out.

## OutputterWindow

- final int HORIZ
- final int VERT
- String[ ] out
- OutputterWindow(h, v)
- getMaxLine()
- getMaxCol()
- clearLine(line)
- overWrite(line,pos,text)

# Runnable Lauftext, Teil II

- Die Methode `run()` des Lauftext schreibt den Text an die jeweils nächste Position im Textpuffer des `OutputterWindow`

```
public class Lauftext implements Runnable {  
    ...  
    public void run() {  
        try {  
            int textLine = out.getMaxLine();  
            while (true) {  
                out.clearLine(textLine);  
                out.overWrite(textLine, pos, text);  
                pos += inkrement;  
                if (pos > out.getMaxCol()) {  
                    pos = 0;  
                }  
                Thread.yield();  
                try { Thread.sleep(300); } catch (Exception e) {}  
            }  
        } catch (Exception e) { e.printStackTrace(); }  
    }  
}
```



# AppText, Teil I

- Die Anwendung selbst besitzt eine `run()`-Methode, die regelmäßig das `OutputterWindow` ausgibt.
- Bis jetzt sind die Threads aber noch nicht gestartet, das passiert im „Hauptprogramm“ (nächste Folie).

```
public class AppText implements Runnable{
    OutputterWindow myOut = new OutputterWindow(80,10);
    Lauftext myText = new Lauftext(myOut, "Hallo Du da!");
    public static void main(String[] args) {
        new AppText().doSomething();
    }
    public void run(){
        while (true){
            myOut.paintWindow();
            Thread.yield();
            try{Thread.sleep(200);}catch(Exception e){}
        }
    }
    ...
}
```

# AppText, Teil II

```
public class AppText implements Runnable{  
    ...  
    public void doSomething() {  
        Thread textThread = new Thread(myText);  
        Thread outputThread = new Thread(this);  
        textThread.start();  
        outputThread.start();  
        MoreHelpers.waitMillis(120000);  
        textThread.stop();  
        outputThread.stop();  
    }  
}
```

- Lauftext und Ausgabe werden als Threads gestartet.
- Nach spätestens 120000 Millisekunden werden beide Threads wieder gestoppt und das Programm beendet.

# Anmerkung

- Die Thread-Methode `stop()` ist veraltet („deprecated“)
  - sie soll nicht mehr verwendet werden,
  - weil sie den Thread sofort (und damit in undefiniertem Zustand) abwürgt.
- Übrigens:
  - Ein Thread, der einmal gestoppt wurde, kann nicht mehr neu gestartet werden.
- Statt `stop()` soll die Methode `interrupt()` verwendet werden:
  - Wenn ein Thread die Methode `interrupt()` erhält, wird intern ein „interrupted“-Flag gesetzt. Sonst nichts.
  - Der Thread weiß jetzt, dass er sich beenden soll und
  - kann sich vernünftig, also zu einem geeigneten Zeitpunkt, beenden, z.B. indem er die Methode `run()` verlässt.

# Zustände von Threads

Zustände	Prüfung durch
Als Objekt erzeugt aber noch nicht gestartet	isAlive() = false
gestartet und am Laufen	isAlive() = true
gestartet aber am schlafen (sleep)	getState() = TIMED_WAITING
gestartet aber am warten (wait())	getState() = WAITING
gestartet aber blockiert (wegen synchronized)	getState() = BLOCKED
gestartet und interrupt-Flag gesetzt	isInterrupted() = true
beendet aber als Objekt noch existent	getState() = TERMINATED

# Konkurrierende Threads

- Folgender Thread schreibt ein Zeichen 20 mal in eine Zeile der Konsole und macht dann in der nächsten Zeile weiter.
- Solange bis er einen `interrupt()` geschickt bekommt.

```
public class CharWriter extends Thread {  
    char c = 'A';  
    public CharWriter(char c){  
        this.c = c;  
    }  
    public void run() {  
        while (!isInterrupted()){  
            for (int index = 0; index < 19; index++){  
                System.out.print(c);  
            }  
            System.out.println(c);  
            MoreHelpers.waitMillis(10);  
        }  
    }  
}
```

# Konkurrierende Threads

- Folgende Anwendung lässt zwei solcher Threads 10 Sekunden lang gegeneinander laufen.
- Einen mit dem Zeichen 'X' und einen mit dem Zeichen 'U'.

```
public class AppCharWriter {  
    public static void main(String[] args) {  
        Thread myCW1 = new CharWriter('X');  
        Thread myCW2 = new CharWriter('U');  
        myCW1.start();  
        myCW2.start();  
        MoreHelpers.waitMillis(10000);  
        myCW1.interrupt();  
        myCW2.interrupt();  
    }  
}
```

- Die Threads konkurrieren um die Ressource „Ausgabe“.

# Beobachtung

- Die Ausdrücke der X und U erfolgen nicht unbedingt zeilenweise.
- Wir haben (bisher) keinen Einfluss darauf, wann das Betriebssystem den einen Thread laufen lässt und wann den anderen.

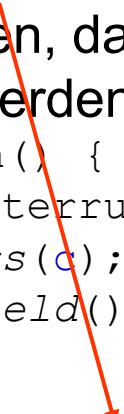
```
public class CharWriter extends Thread {  
    ...  
    public void run() {  
        while (!isInterrupted()) {  
            for (int index = 0; index < 19; index++) {  
                System.out.print(c);  
            }  
            System.out.println(c);  
            Thread.yield();  
        }  
    }  
}
```

- Mit `yield()` kann man immerhin sagen, dass nach jeder Zeile gewechselt werden soll.
- Evtl. wechselt das System aber schon früher...

# Synchronisation

- Mit „synchronize“ kann einem Block oder einer Methode gesagt werden, dass sie zu jeder Zeit nur von einem Thread ausgeführt werden darf:

```
public void run() {  
    while (!isInterrupted()) {  
        writeChars(c);  
        Thread.yield();  
    }  
}  
  
public static synchronized void writeChars(char sc) {  
    for (int index = 0; index < 19; index++) {  
        System.out.print(sc);  
    }  
    System.out.println(sc);  
    MoreHelpers.waitMillis(1);  
}
```





# Synchronisation

```
public void run() {... writeChars(c);...}  
public static synchronized void writeChars(char sc){  
    for (int index = 0; index < 19; index++){  
        System.out.print(sc);  
    }  
    System.out.println(sc);  
    MoreHelpers.waitMillis(1);  
}
```

- Erklärung:

- Jeder Thread muss die Methode `writeChars(char sc)` aufrufen, damit er seine Zeichen ausgeben kann.
- Da diese Methode aber `synchronized` ist, muss jeder Thread warten, bis der andere die Methode ganz ausgeführt hat.

# Synchronisation

```
public void run() {... writeChars(c);...}  
public static synchronized void writeChars(char sc){  
    for (int index = 0; index < 19; index++){  
        System.out.print(sc);  
    }  
    System.out.println(sc);  
    MoreHelpers.waitMillis(1);  
}
```

- Beobachtung
  - Die Ausgabe erfolgt jetzt zeilenweise.
  - Es wird aber nicht unbedingt nach jeder Zeile gewechselt.
- Anmerkung
  - Das funktioniert hier nur, wenn man `writeChars(char sc)` static macht, also als Klassenmethode definiert.
  - Andernfalls hat jeder Thread seine eigene (Objekt-)Methode, die sich gegenseitig nicht beeinflussen bzw. sperren.

# Synchronisation

- Wie kriegen wir die Threads dazu, die Zeilen abwechselnd auszugeben?
- Lösungsansatz:
  - Nachdem ein Thread seine Zeile ausgegeben hat, wird der andere benachrichtigt und
  - er selbst macht dann erstmal Pause,
  - solange bis er selbst wieder benachrichtigt wird.
- Die Befehle dazu
  - heißen `wait()` und `notify()` bzw. `notifyAll()`,
  - sind in `Object` definiert und
  - funktionieren noch wesentlich allgemeiner als wir es hier darstellen.

# Synchronisation

- Konkreter:
  - `wait()` und `notify()` bzw. `notifyAll()` können nur in einem `synchronized` Block stehen.
  - `wait()` geht an den Thread, der gerade in dem `synchronized` Block drin ist (das kann ja nur einer sein!)
  - `notifyAll()` geht an alle Threads, die auf den `synchronized` Block warten.
  - `notify()` geht an einen Thread (zufällig ausgewählt), der auf den `synchronized` Block wartet.
- Folgerung:
  - Die Threads, die abgewechselt werden sollen, müssen also auf einen gemeinsamen `synchronized`-Block zugreifen.
  - Sonst gehts nicht, weil der `notify()`-Befehl dann nicht den richtigen Thread trifft.

# Synchronisation

- Implementierung, 1. Ansatz:

```
public static synchronized void writeChars(char sc){  
    for (int index = 0; index < 19; index++){  
        System.out.print(sc);  
    }  
    System.out.println(sc);  
    MoreHelpers.waitMillis(1);  
    notifyAll();  
    wait();  
}
```

- Beobachtung:

- geht nicht, weil die Methode static ist, `notifyAll()` und `wait()` aber Objekt-Methoden sind.
- Wir brauchen eine Objekt-Methode.

- Ansatz:

- Wir bauen uns ein Ausgabeobjekt, das wir beiden Threads zuweisen.
- Dieses bekommt eine `synchronized`-Ausgabemethode.

# Synchronisation

- Implementierung Ausgabeklasse:

```
public class CharWriteOut{  
    public synchronized void writeChars(char c, Thread thr){  
        for (int index = 0; index < 19; index++){  
            System.out.print(c);  
        }  
        System.out.println(c);  
        MoreHelpers.waitMillis(1);  
        notify();  
        try{wait();}catch (InterruptedException e)  
            {thr.interrupt();}  
    }  
}
```

- Beide Threads müssen dieselbe Methode benutzen,
- also die Methode desselben Objekts.
- Anmerkung zur `InterruptedException` kommt noch...

# Synchronisation

- Implementierung Anwendung:

```
public class AppCharWriterWN {  
    public static void main(String[] args) {  
        CharWriteOut cwo = new CharWriteOut();  
        Thread myCW1 = new CharWriterWN('X', cwo);  
        Thread myCW2 = new CharWriterWN('U', cwo);  
        myCW1.start();  
        myCW2.start();  
        MoreHelpers.waitMillis(5000);  
        myCW1.interrupt();  
        myCW2.interrupt();  
    }  
}
```

- Nur ein gemeinsames Ausgabeobjekt

- Implementierung Threads

```
public class CharWriterWN extends Thread {  
    char c = 'A';  
    CharWriteOut cwo;  
    public CharWriterWN() {}  
    public CharWriterWN(char c, CharWriteOut cwo) {  
        this.c = c;  
        this.cwo = cwo;  
    }  
    public void run() {  
        while (!isInterrupted()) {  
            cwo.writeChars(c, this);  
        }  
    }  
}
```




# Anmerkung

- InterruptedException


- Tritt auf, wenn ein Thread interrupted wird, während er wartet.
- Durch die `InterruptedException` wird der Interrupt aber wieder vergessen.
- Deshalb muss der Interrupt im `catch`-Block erneut gesetzt werden.

```
public synchronized void writeChars(char c, Thread thr){  
    ...  
    try{wait();} catch (InterruptedException e)  
        {thr.interrupt();}  
}
```



- Deshalb muss der Thread selbst an die Methode übergeben werden.

```
public void run() {  
    while (!isInterrupted()){  
        cwo.writeChars(c, this);  
    }  
}
```



# **Programmierung II**

## **Thema 12: Diverses**

**Fachhochschule Ludwigshafen  
University of Applied Sciences**

- Software muss dokumentiert werden:
  - Bei gemeinsamer Entwicklung
  - Für spätere Änderung, Korrektur oder Weiterentwicklung
  - Beschreibung der Schnittstellen
- Externe Dokumentation: Separate Beschreibung
  - Kann „einfach so“ gelesen und weitergegeben werden
  - Enthält viele Redundanzen
  - Muss bei jeder Softwareänderung aktualisiert werden.  
➔ Gefahr von Inkonsistenzen
- Interne Dokumentation
  - Das Programm ist seine eigene Dokumentation: Zu knapp
  - Zusätzliche Kommentare im Programm ➔ Gefahr der Inkonsistenz besteht immer noch (allerdings reduziert)
  - Möglichkeit, daraus eine aktuelle externe Dokumentation automatisch zu erzeugen

- Kommentare in Java:

```
// Diese ganze Zeile ist ein Kommentar  
/* Von hier ab ... geht dieser Kommentar bis  
...hier */
```

- Dokumentationskommentar

```
/** Dokumentationskommentare haben einen Stern mehr am  
 * Anfang. Der Stern am Anfang jeder Zeile ist nur  
 * Konvention.  
 * ... das Ende ist wie bei einem normalen Kommentar  
 */
```

- Dokumentationskommentare kann man setzen

- vor Klassen, Schnittstellen (`Interface`), Aufzählungen (`Enum`)
- vor Attribute und Methoden,
- die dann durch den Kommentar beschrieben werden sollen.

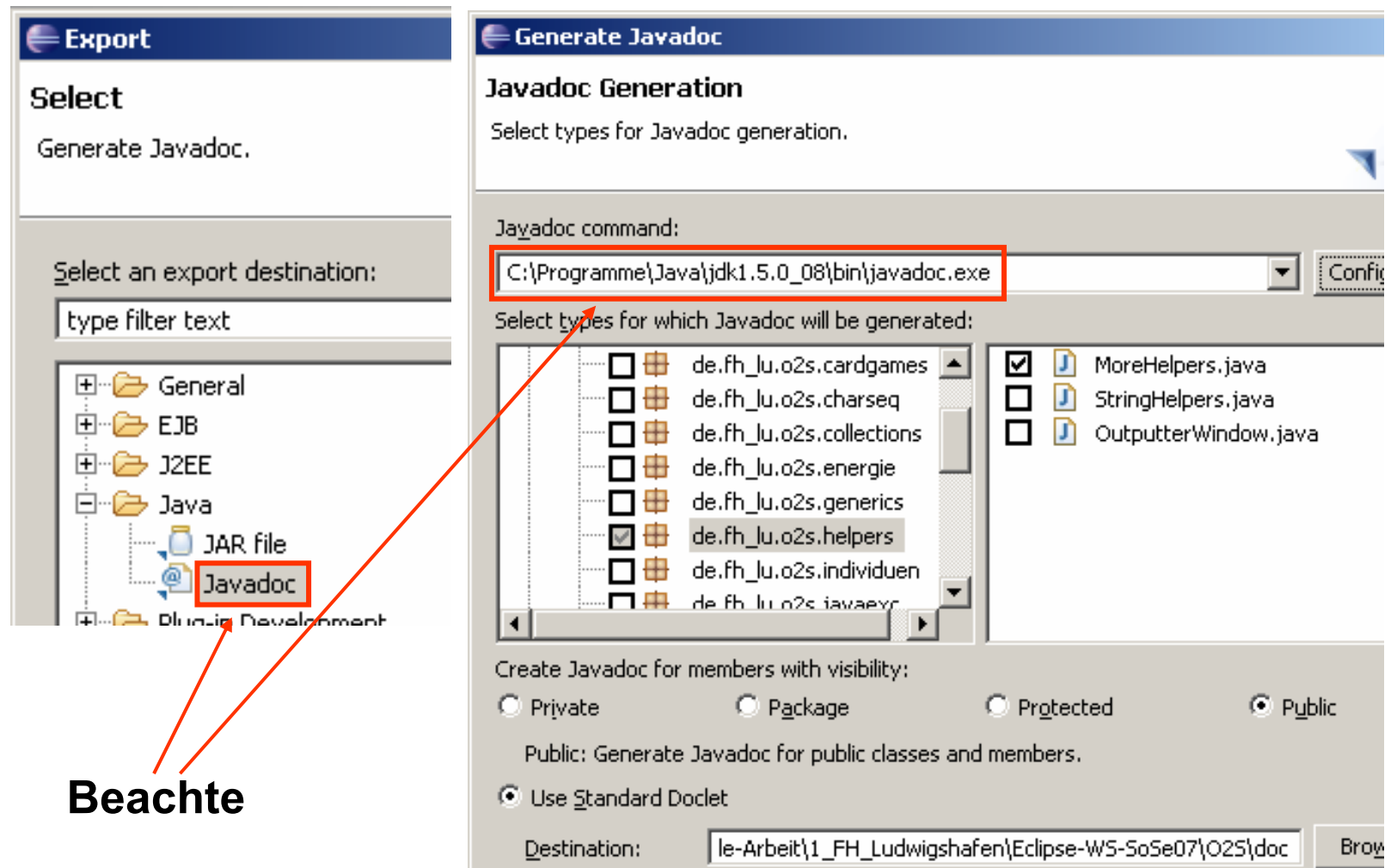
# Beispiel

- Dokumentationskommentare setzen:

```
package de.fh_lu.o2s.helpers;
/**
 * Diese Klasse stellt verschiedene Hilfsfunktionen zur
 * Verfügung, die vom Java-Standard nicht oder nicht so
 * einfach geboten werden.
 * @author Haio Röckle
 */
public class MoreHelpers {
    /**
     * liest ein Zeichen von der Tastatur.
     * @return Gibt das eingegebene Zeichen als char zurück.
     */
    public static char liesZeichen() {
        ...
    }
}
```

# Beispiel

- Externe Dokumentation erzeugen, in Eclipse: File → Export...



# Beispiel

`de.fh_lu.o2s.helpers`

## Class MoreHelpers

```
java.lang.Object
└─ de.fh_lu.o2s.helpers.MoreHelpers
```

---

```
public class MoreHelpers
    extends java.lang.Object
```

Diese Klasse stellt verschiedene Hilfsfunktionen zur Verfügung, die vom Java-Standard nicht oder die der Autor benötigt aber nicht gefunden hat.

**Author:**

Haio Röckle

---

### Constructor Summary

[MoreHelpers](#) ()

### Method Summary

static char	<a href="#"><u>liesZeichen</u></a> ()
-------------	---------------------------------------

Unterbricht das Programm, um ein Zeichen von der Tastatur zu lesen.

- Javadoc
  - Java-Hilfsprogramm
  - findet sich als `<JDK>\bin\javadoc.exe`
  - kann auch ohne Eclipse ausgeführt werden.
- erzeugte Dokumentation
  - sieht „professionell“ aus
  - kann trotzdem Blödsinn oder veraltetes Zeug enthalten
  - falls keine Dokumentationskommentare angelegt wurden, werden die Deklarationen unkommentiert aufgelistet.
- Andere Formate
  - können mit Zusatzprogrammen, so genannten JavaDoclets, erzeugt werden.



# Veraltete Methoden

- Manche Java-Methoden sind nicht mehr aktuell, z.B.
  - weil sie einen Rechtschreibfehler enthielten
  - weil es jetzt bessere Methoden gibt
- Sie werden aber von älteren Programmen noch benutzt,
  - deshalb darf man sie nicht einfach löschen.
  - stattdessen werden sie als „deprecated“ gekennzeichnet.

```
/**
 * Unterbricht das Programm, um ein Zeichen von der T
 * @return Gibt als char das eingegebene Zeichen (die
 */
@Deprecated
public static char liesZeichen() {
```

- Die Kennzeichnung erfolgt durch das „Annotation“-Tag  
`@Deprecated`

# Annotations

- Manchmal bringt der Compiler eine Warnung, z.B.
  - wenn eine `deprecated`-Methode verwendet wird,
  - wenn eine Variable deklariert aber nicht verwendet wird,
  - ...
- Meistens ist dies ein Hinweis auf einen logischen Fehler.
- Manchmal ist es aber auch Absicht, z.B.
  - wenn nur eine `deprecated`-Methode verwendet werden kann,
  - wenn ein Thread direkt bei der Erzeugung gestartet und nie wieder angefasst wird.
- Dann kann man vor der Methode oder dem Code, der die Warnung erzeugt, ebenfalls eine Annotation einfügen, z.B.
  - `@SuppressWarnings("deprecation")`  
`public void doSomething(){...`
  - `@SuppressWarnings("unused")`  
`ZaehlRunnable1 gerade = new ZaehlRunnable1(2, 2, 25);`
- Eclipse hilft dabei über „QuickFix“.

# Pakete, Klassen und Dateien

- Java-Software besteht aus einer Menge von Klassen:
  - Java-Standardklassen aus `java.lang`
  - Java-Standardklassen aus weiteren packages, z.B. `java.util`, `java.io`, etc.
  - Klassen von weiteren Anbietern, z.B. zur Datenbankbindung
- Diese Klassen stehen in Archiven
  - darin werden viele verschiedenen Dateien zusammengefasst
  - incl. Pfadangabe, z.B. `java\util\Vector.class`
- Damit die Java-Software die Klassen findet, muss sie
  - wissen, wo die Archive stehen → **Class-Path**
  - in den Archiven die richtigen Pfadangaben kennen → **JAR-Files**

# Pakete, Klassen und Dateien

- Für die installierte Java-JRE oder das JDK gibt es eine Umgebungsvariable `CLASSPATH`:

```
CLASSPATH=.;C:\PROGRA~1\IBM\SQLLIB\java\db2java.zip;C:\PROGRA~1\IBM\SQLLIB\java\
db2jcc.jar;C:\PROGRA~1\IBM\SQLLIB\java\db2jcc_license_cu.jar;C:\PROGRA~1\IBM\SQL
LIB\bin;C:\PROGRA~1\IBM\SQLLIB\java\common.jar;C:\PROGRA~1\IBM\SQLLIB\java\sqlj.
zip;C:\PROGRA~1\IBM\SQLLIB\tools\db2XTrigger.jar
```

- In Eclipse hat jedes Projekt seinen eigenen Classpath, der in einer XML-Datei `.classpath` steht (der '.' muss da hin!):

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
  <classpathentry kind="src" path=""/>
  <classpathentry kind="con"
                  path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
  <classpathentry kind="output" path=""/>
</classpath>
```

- In umfangreicher Software mit vielen externen Paketen, kann der Classpath sehr lang werden.

# JAR-Files und andere Archive

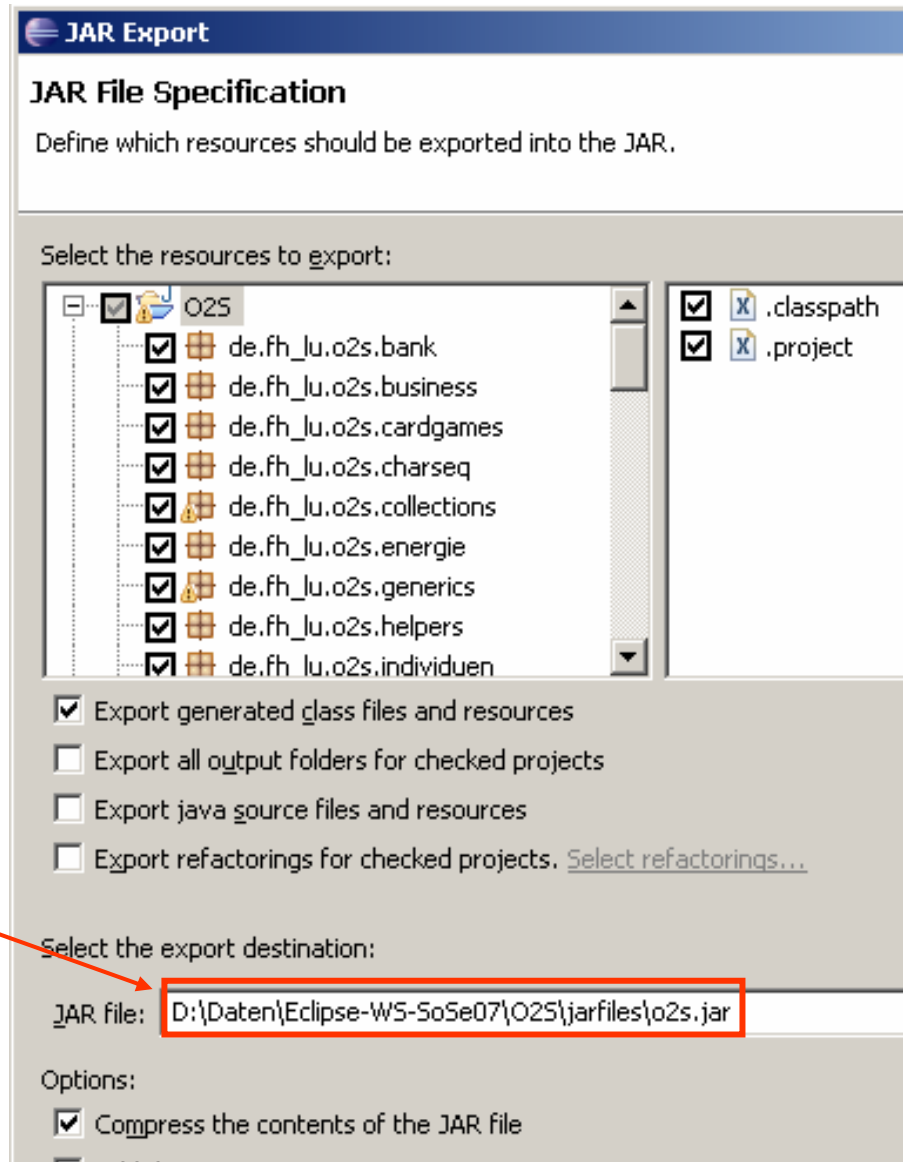
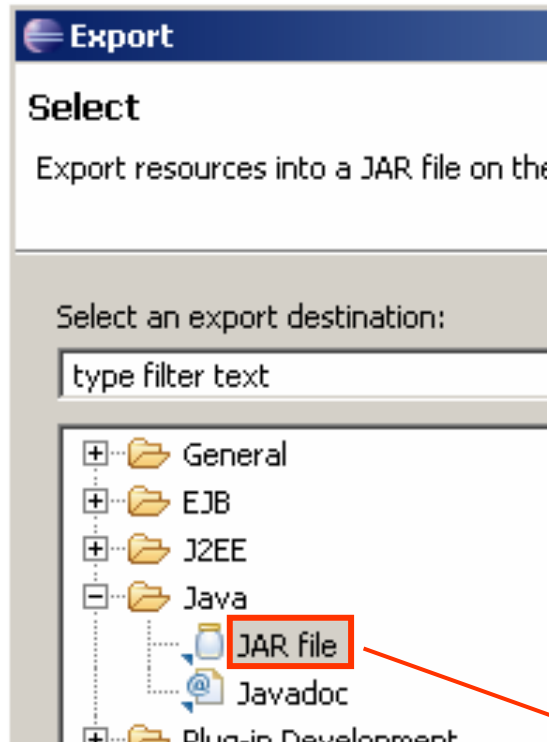
- Java-Dateien werden in Archiven zusammengepackt
  - JAR = Java Archive
  - WAR = Web Archive für Web Anwendungen
  - EAR = Enterprise Archive (aus J2EE)
- Darin enthalten sind
  - Java-Klassen (`....class`)
  - Konfigurationsfiles, z.B. (`....xml`)
  - ein Manifest-File `Manifest.mf`, das z.B. den Autor, die Version oder bei einer Anwendung die Haupt-/Anwendungsklasse angibt.
  - in einer Dateistruktur mit Pfadangaben (Unterverzeichnissen)
- Archive werden im Classpath eingetragen

# JAR-Files und andere Archive

- Wenn Java eine Datei sucht,
  - dann muss sie einen Pfad zu der Datei angeben,
  - das System sucht dann mit dem angegebenen Pfad innerhalb aller Archive im Classpath.
- Es müssen also die Anwendung, der Classpath und die Pfadangaben in den Archiven zusammenpassen.
- Übrigens: Ein Archiv ist
  - strukturell ein ZIP-File
  - nur mit anderer Namensendung
- Zur Erzeugung von Archiven dient
  - das Programm JAR
  - Es findet sich als `<JDK>\bin\jar.exe`

# Beispiel

- JAR-File erzeugen, in Eclipse: File → Export...



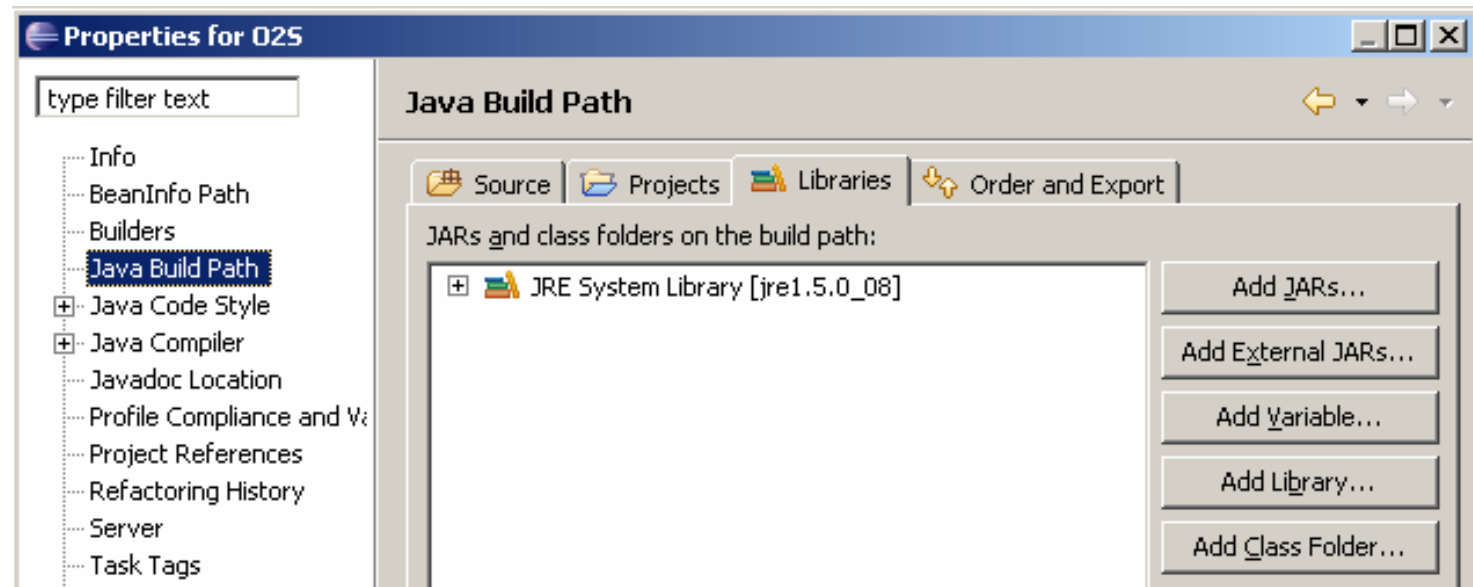
# JAR-Files erzeugen

- JAR-Files erzeugen
  - ist in Eclipse ganz einfach
  - kann aber auch ohne Eclipse ausgeführt werden.
- Mit Eclipse
  - ist es bequemer,
  - man hat aber keine hundertprozentige Kontrolle über die Pfade
- Ohne Eclipse
  - hat man viele Parameter für die Anwendung `jar.exe` zur Verfügung
  - ➔ sollte man in einem Skript zusammenfassen und speichern
- Verpackungsstrategien
  - viele Pakete in einem JAR-File → ein Classpath-Eintrag reicht
  - mehrere JAR-Files → evtl. detailliert ausliefern und installieren
  - ➔ Vorgehen nach inneren Zusammenhängen der Pakete wählen.



# Anmerkungen

- Archive einbinden: Immer im Classpath
  - ohne Eclipse (für JRE / JDK): auf Betriebssystemebene
  - mit Eclipse über Projekteigenschaften (Properties) → Java Build Path → Libraries



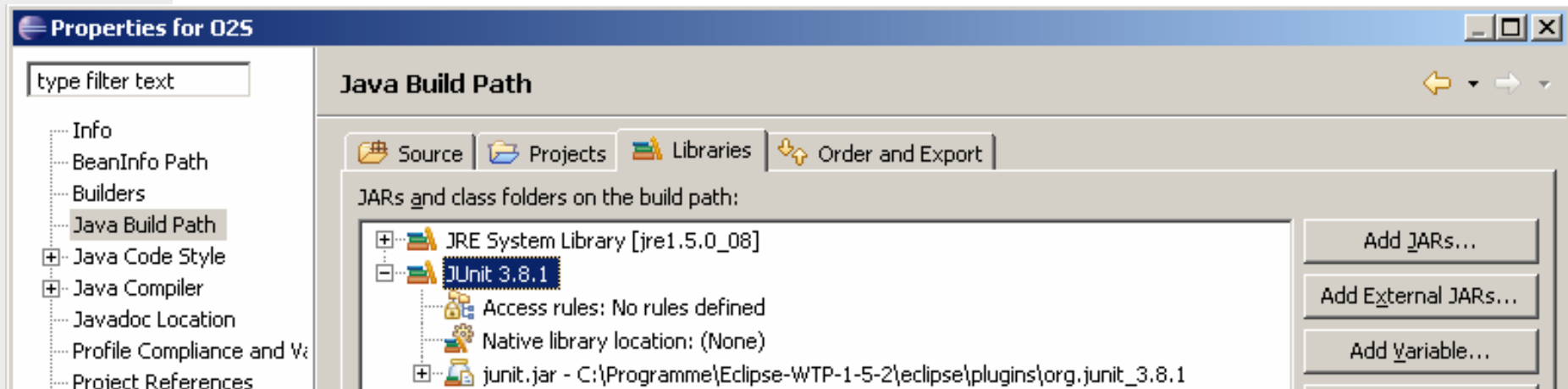
- Ursprünglich wurde Java-Code aus dem Internet geladen
  - So genannte Applets
  - auf Webseiten eingebettet
  - Autor unklar → prinzipiell unsicher
- Deshalb durften Applets nicht
  - auf Dateien zugreifen, sie „lebten in einer Sandbox“
  - keine seriösen Programme möglich.
- Lösungsansatz: Java-Programme werden signiert
  - Damit kann der Autor zweifelsfrei nachgewiesen werden
  - Wenn der Autor bekannt ist, dann darf das Programm auch auf Dateien, Datenbanken, Drucker, etc. zugreifen

# Digitale Signatur

- Der Signierer hat einen privaten Schlüssel,
  - den sonst keiner kennt.
  - Damit kann er Software signieren.
- Jeder Interessent erhält einen öffentlichen Schlüssel,
  - mit dem er die Echtheit des Signierers und
  - der Software nachweisen kann.
- Anmerkung: Die Hintergründe davon sind ziemlich kompliziert...
- Code-Signatur
  - bedeutet digitale Signatur der Archiv-Dateien.
  - Wird gemacht mit einem Programm JARSIGNER.
  - Das findet sich als `<JDK>\bin\jarsigner.exe`

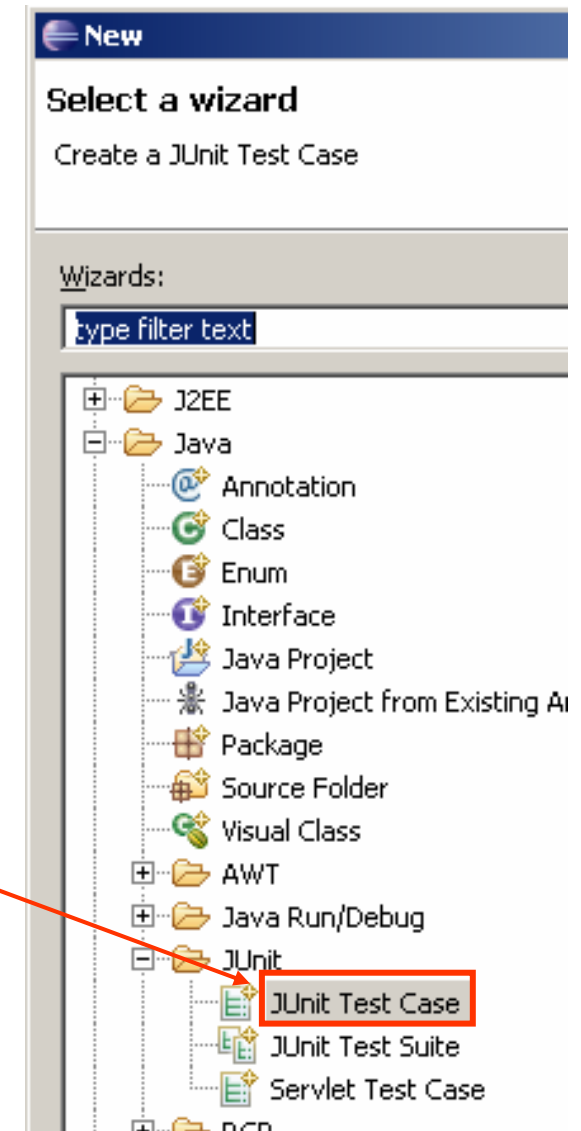
# Testen mit JUnit

- Zum Test von Java-Anwendungen
  - wird automatisiertes testen empfohlen
  - Dafür hat sich das Test-Framework JUnit durchgesetzt (steht für Java-Unit-Test).
- In Eclipse WTP 1.5.2
  - wird JUnit mit ausgeliefert,
  - muss aber noch ins jeweilige Projekt eingebunden werden mit Projekt → Properties → Java Build Path → Libraries → Add External JARs... und auswählen von ...\\junit.jar



# JUnit

- Ein Unit-(Einheiten-)Test mit JUnit besteht aus einem oder mehreren Testfällen.
- Ein Testfall ist eine Subklasse von TestCase aus dem Package junit.framework.
- Ein Testfall (eine solche Subklasse) kann mit Eclipse angelegt werden über New → Other... → Java → JUnit → TestCase



# Beispiel

- Im Beispiel wollen wir unsere Klasse `StringStapel` aus Package `de.fh_lu.o2s.testpackage` testen. Wir verwenden
  - Ein separates Testpaket (selbst angelegt, nicht unbedingt nötig)
  - Einen Testfall (Subklasse von `TestCase`) namens `TestStringStapel`.
  - Die Klasse `StringStapel`, die wir testen wollen.

```
package de.fh_lu.o2s.test;  
  
import de.fh_lu.o2s.testpackage.StringStapel;  
import junit.framework.TestCase;  
  
public class TestStringStapel extends TestCase {  
    ...  
}
```

# Testmethoden

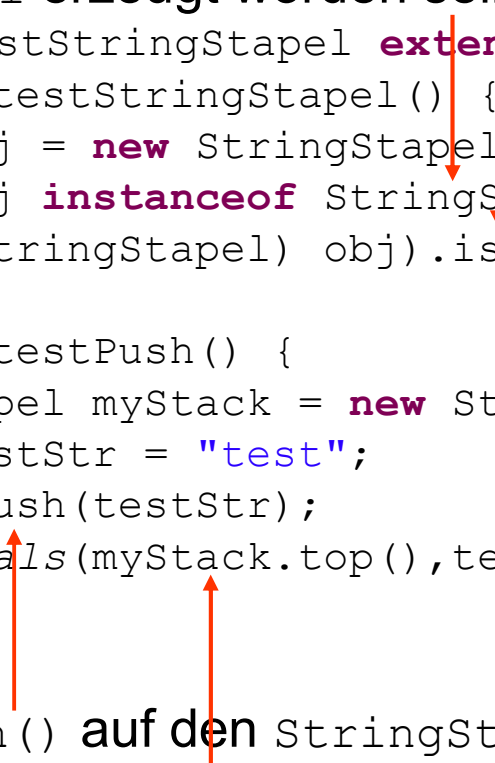
- Im TestStringStapel entwickeln wir Testmethoden:
  - Methodenname muss mit „test“ beginnen
  - keine Eingabeparameter
  - kein Ausgabeparameter
  - enthält Prüfungen
- Prüfungen sind implementiert als Assertions (Zusicherungen):
  - `assert (Ausdruck)` : Ausdruck muss true sein,
  - `assertFalse (Ausdruck)` : Ausdruck muss false sein,
  - `assertEquals (Objekt1, Objekt2)` : Beide Objekte müssen gleich sein,
  - **Weitere:** `assertNotNull()`, `assertSame()`,
  - `fail (message)` : Geht immer schief und gibt dabei die Message aus.

# TestStringStapel

- Beispiel:

1.) Nach dem Aufruf von `new StringStapel()` muss ein leerer `StringStapel` erzeugt worden sein.

```
public class TestStringStapel extends TestCase {  
    public void testStringStapel() {  
        Object obj = new StringStapel();  
        assert(obj instanceof StringStapel);  
        assert((StringStapel) obj).isEmpty();  
    }  
    public void testPush() {  
        StringStapel myStack = new StringStapel();  
        String testStr = "test";  
        myStack.push(testStr);  
        assertEquals(myStack.top(), testStr);  
    }  
    ...  
}
```



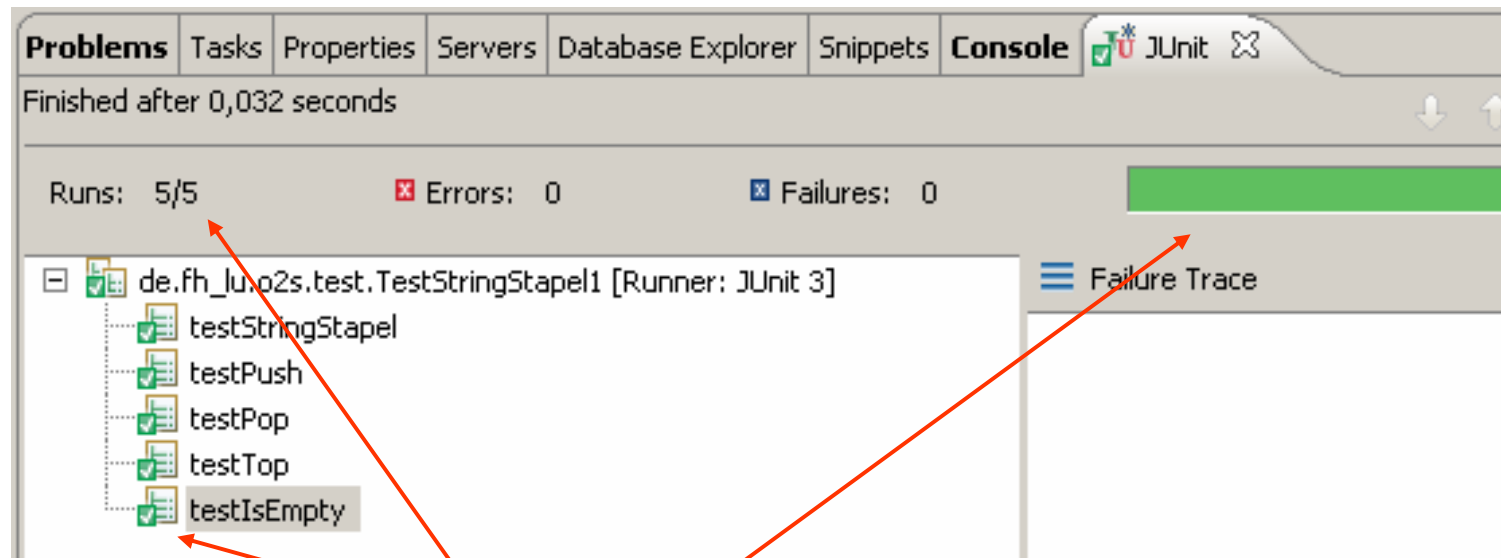
2.) Was mit `push()` auf den `StringStapel` gelegt wurde, muss anschließend drauf liegen.

3.-5.) `testPop()`, `testTop()`, `testIsEmpty()` entsprechend...



# Test durchführen

- Testausführung mit Eclipse:
  - Run As... → JUnit Test
- Ergebnis:



- Methoden einzeln erfolgreich
- ganzer Test erfolgreich

# Anmerkungen

- Vorteile
  - Der Test wird einmal angelegt und kann immer wieder durchgeführt werden: „Regressionstest“
  - Dadurch lohnt es sich, den Test gründlich zu entwickeln
  - Der Test kann bereits vor der Entwicklung der Software angelegt werden: „test-driven Softwareentwicklung“
- Nachteile
  - Aufwand zur Testfallentwicklung,
  - Unvollständige Testfälle können zu falscher Sicherheit führen.

# Debugging

- Weitere Möglichkeit zur Fehlersuche: Debugging
  - Ein Programm kann mittendrin angehalten werden (Breakpoints)
  - An dieser Stelle können die Variablenwerte geprüft werden (inspect)
  - Mit Einzelschrittverarbeitung (single step) kann die Korrektheit des Programms quasi in Zeitlupe (super slo mo) geprüft werden.
- In Eclipse:
  - Anstatt Run As → Java Application
  - Verwende Debug As → Java Application
- Es erscheint die Debug-Perspective
  - Klick links von einer Programmzeile: Breakpoint setzen (geht auch in der Java-Perspective)

# Beispiel

**Debug - AppRennen.java - Eclipse SDK**

File Edit Source Refactor Navigate Search Project Run Window Help

Debug Servers

AppRennen (1) [Java Application]  
 de.fh\_lu.o2s.rennen.AppRennen at localhost:1320  
 Thread [main] (Running)  
 C:\Programme\Java\jre1.5.0\_08\bin\javaw.exe (12.05.2007 14:11:48)

AppRennen (1) [Java Application]  
 de.fh\_lu.o2s.rennen.AppRennen at localhost:1323  
 Thread [main] (Suspended (breakpoint at line 27 in AppRennen))  
 AppRennen.doSomething() line: 27  
 AppRennen.main(String[]) line: 20  
 C:\Programme\Java\jre1.5.0\_08\bin\javaw.exe (12.05.2007 14:12:14)

**Variables**

Name	Value
this	AppRennen (id=10)
DISTANZ	80
ESCAPECHAR	x
HINTERRADZEICHEN	>
painter	RennPainter (id=12)
DISTANZ	80

**AppRennen.java**

```

public void doSomething() {
    stand = new Rennstand();
    painter = new RennPainter(stand);
    painter.drawRace();
    MoreHelpers.waitMillis(STEPMILLIS);

    while(stand.getLeaderPos() < DISTANZ - teilnlaenge){
        int[] zweiWerte = moveRacers();
        if (zweiWerte[0] > 0){

```

**Outline**

- de.fh\_lu.o2s.rennen
  - import declarations
  - AppRennen
    - TEILNEHMERZAHL : int
    - DISTANZ : int
    - VORDERRADZEICHEN : ch
    - HINTERRADZEICHEN : ch
    - ESCAPECHAR : char
    - STEPMILLIS : long

Breakpoint

Singlestep

Variablen

# Was jetzt doch nicht dran kam

- GUI
- Netzwerkprogrammierung
- Files und Streams
- ➔ Schade eigentlich!