Objektorientierte Programmiertechnik

Mark Keinhörster

FOM
Hochschule für Ökonomie und Management

5. Juli 2013

Inhaltsverzeichnis

Vorkenntnisse aus dem vorherigen Semester

- Sprechen Sie Java?
  - Was sind Datentypen und Variablen?
  - Kennen Sie Ausdrücke?
  - Wie werden Operatoren genutzt? Welche gibt es?
  - Wann und wie werden Kontrollstrukturen eingesetzt?
  - Was sind Blöcke?

Vorkenntnisse aus dem vorherigen Semester

- Sprechen Sie Java?
  - Was sind Datentypen und Variablen?
  - Kennen Sie Ausdrücke?
  - Wie werden Operatoren genutzt? Welche gibt es?
  - Wann und wie werden Kontrollstrukturen eingesetzt?
  - Was sind Blöcke?

Vorkenntnisse aus dem vorherigen Semester

- Sprechen Sie Java?
  - Was sind Datentypen und Variablen?
  - Kennen Sie Ausdrücke?
  - Wie werden Operatoren genutzt? Welche gibt es?
  - Wann und wie werden Kontrollstrukturen eingesetzt?
  - Was sind Blöcke?

Vorkenntnisse aus dem vorherigen Semester

- Sprechen Sie Java?
  - Was sind Datentypen und Variablen?
  - Kennen Sie Ausdrücke?
  - Wie werden Operatoren genutzt? Welche gibt es?
  - Wann und wie werden Kontrollstrukturen eingesetzt?
  - Was sind Blöcke?

Vorkenntnisse aus dem vorherigen Semester

- Sprechen Sie Java?
  - Was sind Datentypen und Variablen?
  - Kennen Sie Ausdrücke?
  - Wie werden Operatoren genutzt? Welche gibt es?
  - Wann und wie werden Kontrollstrukturen eingesetzt?
  - Was sind Blöcke?

Ihre Erwartungen an die Veranstaltung

Was möchten Sie gerne behandeln?

-
-
-
-
-

Erwartungen an die Veranstaltung

Was sollten Sie am Ende können?

- OOP verstehen und darstellen können
- Konzepte der OOP erläutern und anhand von Code darstellen
- Die OOP bei der Softwareentwicklung richtig einsetzen
- Weiterführende Konzepte und Lösungen der OOP verstehen
- Die Grundlagen von Java als Werkzeug der OOP beherrschen

In der Veranstaltung verwendete Werkzeuge

Programmiersprache

Entwicklungsumgebung

Modellierung

Java ist. . .

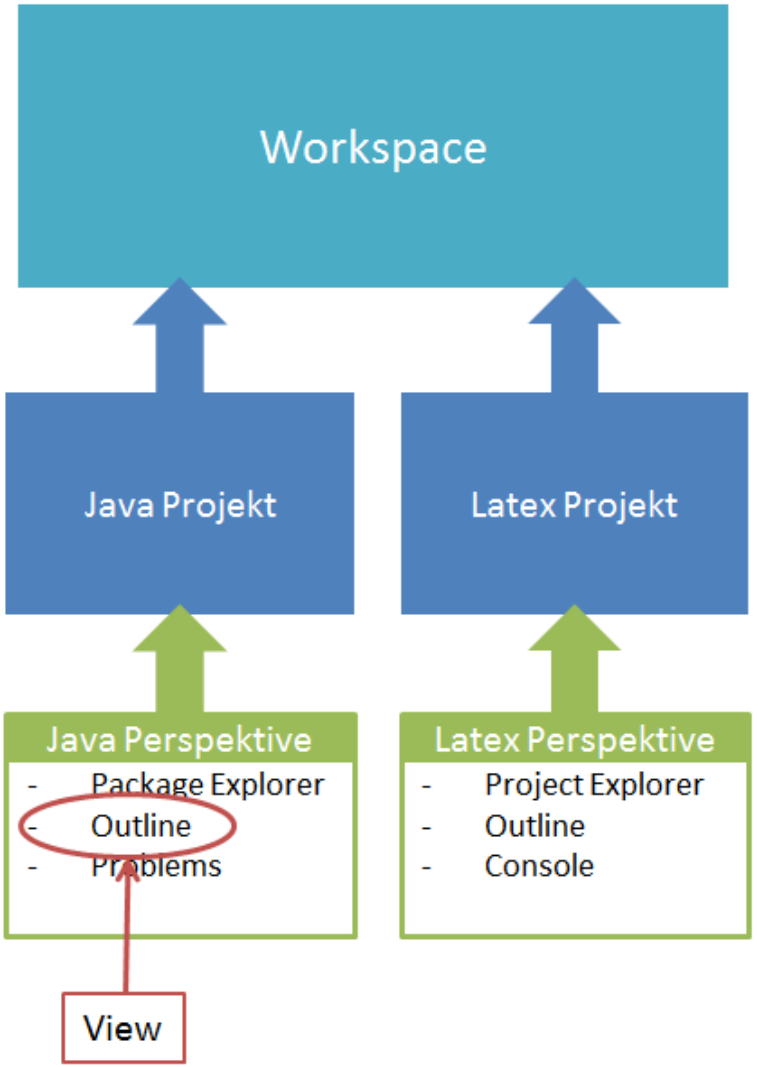
- Weit verbreitet.
- Verhältnismäßig leicht zu erlernen.
- Plattformunabhängig.
- Objektorientiert.

Dokumentation

`http://docs.oracle.com/javase/7/docs/api/`

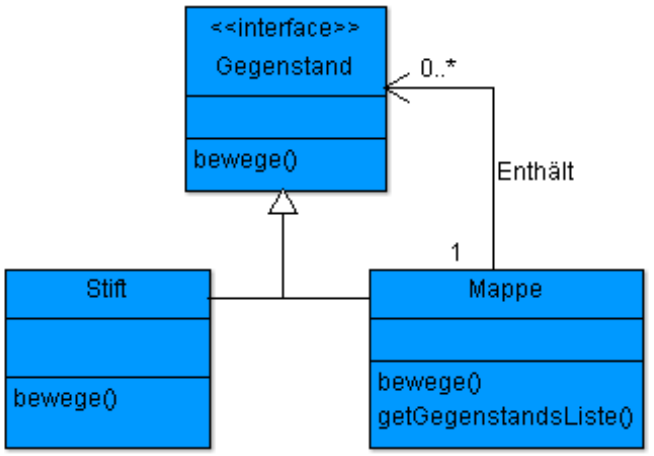
Eclipse erleichtert uns die Entwicklung

- Integrated Development Environment
- Verwaltet Dateien in Projekten
- Es existieren 4 Hauptkomponenten:
  1. Workspaces
  2. Projekte (Projects)
  3. Perspektiven (Perspectives)
  4. Sichten (Views)



UML unterstützt bei der Darstellung der OOP

- Unified Modeling Language
- Sprache zur objektorientierten Modellierung
- Standardisiert von der OMG
- Auch zur Implementierung genutzt



OOP Grundlagen

Objektorientierte Programmierung

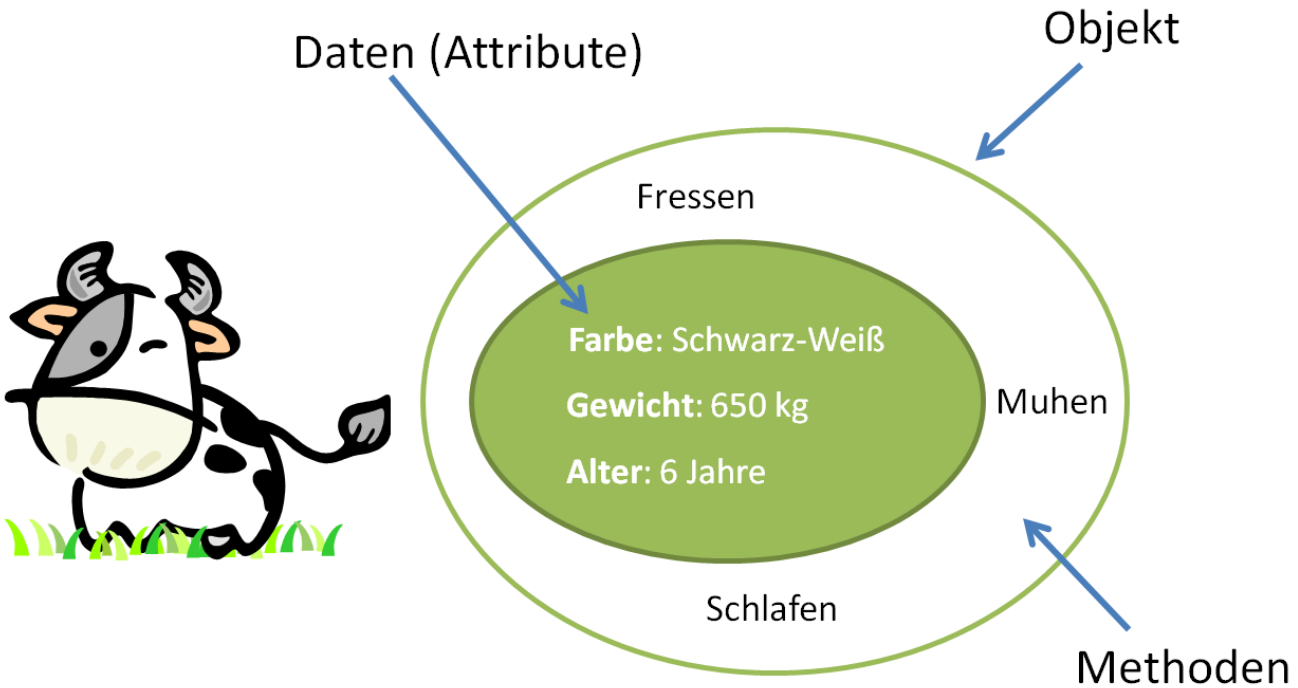
- Modularer Programmaufbau
- Die einzelnen Module heißen Objekte
- Objekte haben Daten und Methoden
- Objekte erlauben den Aufbau komplexer Programme

Erhöhte Abstraktion

- OOP ermöglicht die Abbildung von Problemen auf Modelle der Realität
- Fokus auf die fachliche Aufgabenstellung

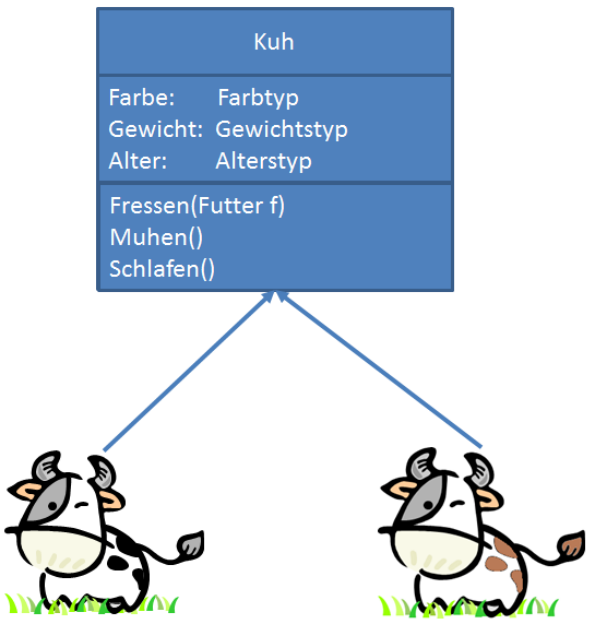
Grundbegriffe

Objekte, Methoden und Attribute in der OOP



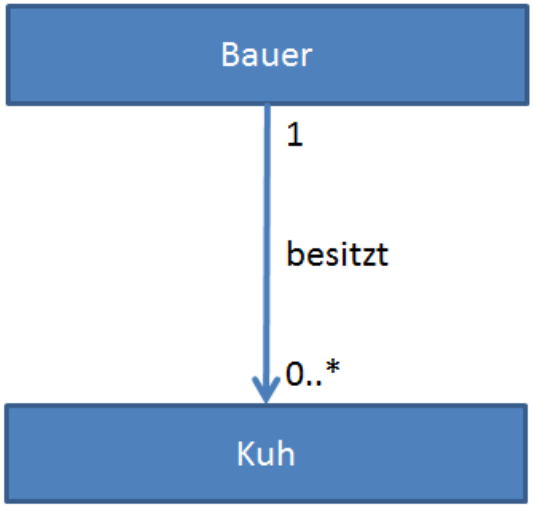
Klassen als Objektbauplan

- Klasse fungiert als Bauplan für Objekte
- Aus diesem Bauplan lassen sich beliebig viele Objekte erzeugen
- Objekte aus einer Klasse besitzen die gleichen Methoden
- Objekte aus einer Klasse besitzen die gleichen Attribute
  (Werte können jedoch variieren)
- Klassen können durch Vererbung weiter spezialisiert werden



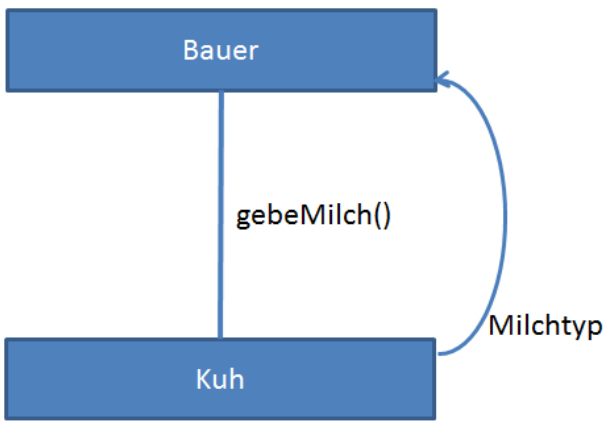
Beziehungen zwischen Objekten

- Zwischen Objekten können Beziehungen existieren
- Beziehung zwischen Bauer und Kuh:
  1. Der Bauer kann 0 bis n Kühe besitzen
  2. Eine Kuh gehört genau einem Bauer
  3. Keine Kuh kann einen Bauer besitzen (gerichtete Beziehung)



Methodenaufrufe

- Objekte können sich untereinander Botschaften senden
- Damit wird der Empfänger aufgefordert etwas auszuführen
- Erweitern wir die Klasse "Kuh"
  1. Kuh bekommt die Methode: "gebeMilch"
  2. "gebeMilch" gibt ein Objekt vom Typ "Milchtyp" zurück
- Der Bauer löst durch senden einer Botschaft die Methode "gebeMilch" aus
- Die Kuh führt die Methode aus und gibt Milchtyp-Objekt zurück

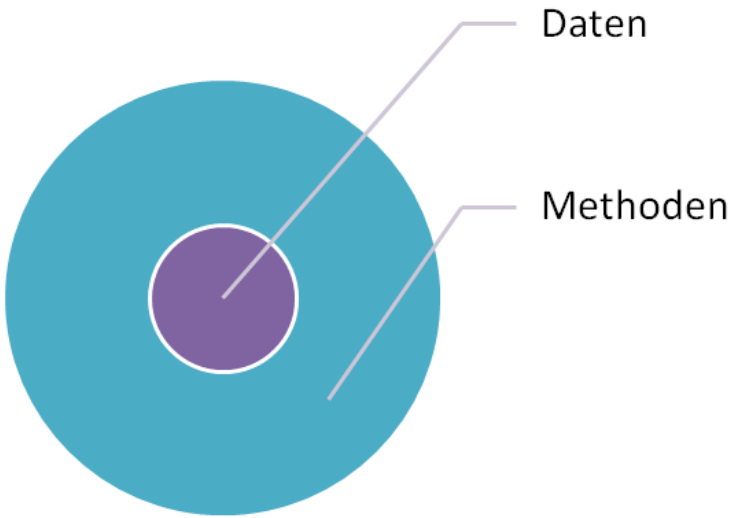


Die Grundprinzipien der OOP

- Kapselung
- Vererbung
- Polymorphismus

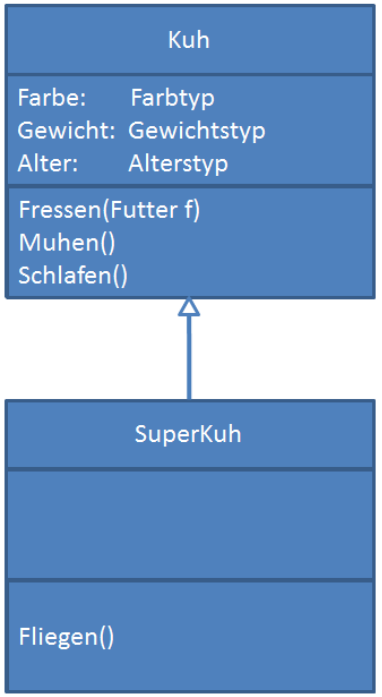
Das Prinzip der Kapselung (Geheimnisprinzip)

- Attribute sind von Methoden gekapselt
- Damit sind die Methoden die Schnittstellen zur Außenwelt
- Dient dem kontrollierten Zugriff auf Attribute und Logik
- Reduziert die Komplexität da interne Strukturen verborgen bleiben



Das Prinzip der Vererbung

- Klassen lassen sich durch Vererbung spezialisieren
- Alle Attribute und Methoden der Vaterklasse sind Attribute und Methoden der Subklasse (Die Kind-Klasse "erbt")
- Funktionalität der Vaterklasse bleibt vorhanden, nur die Unterschiede werden definiert

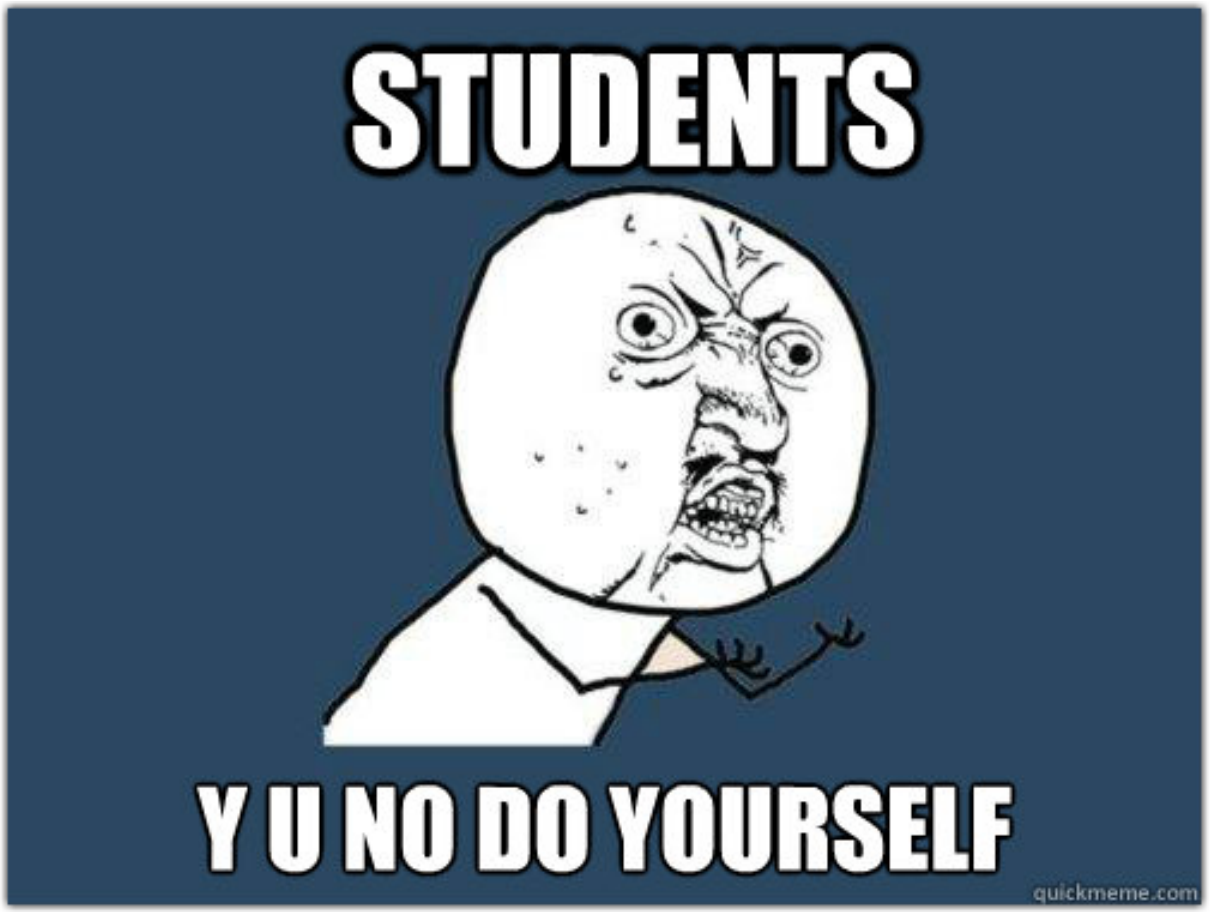


Das Prinzip des Polymorphismus

- Die Subklasse kann überall dort genutzt werden, wo auch die Superklasse genutzt wird
- Damit wird zur Laufzeit entschieden aus welcher Klasse die Methode kommt
- Dieses Prinzip wird als "späte Bindung" bezeichnet

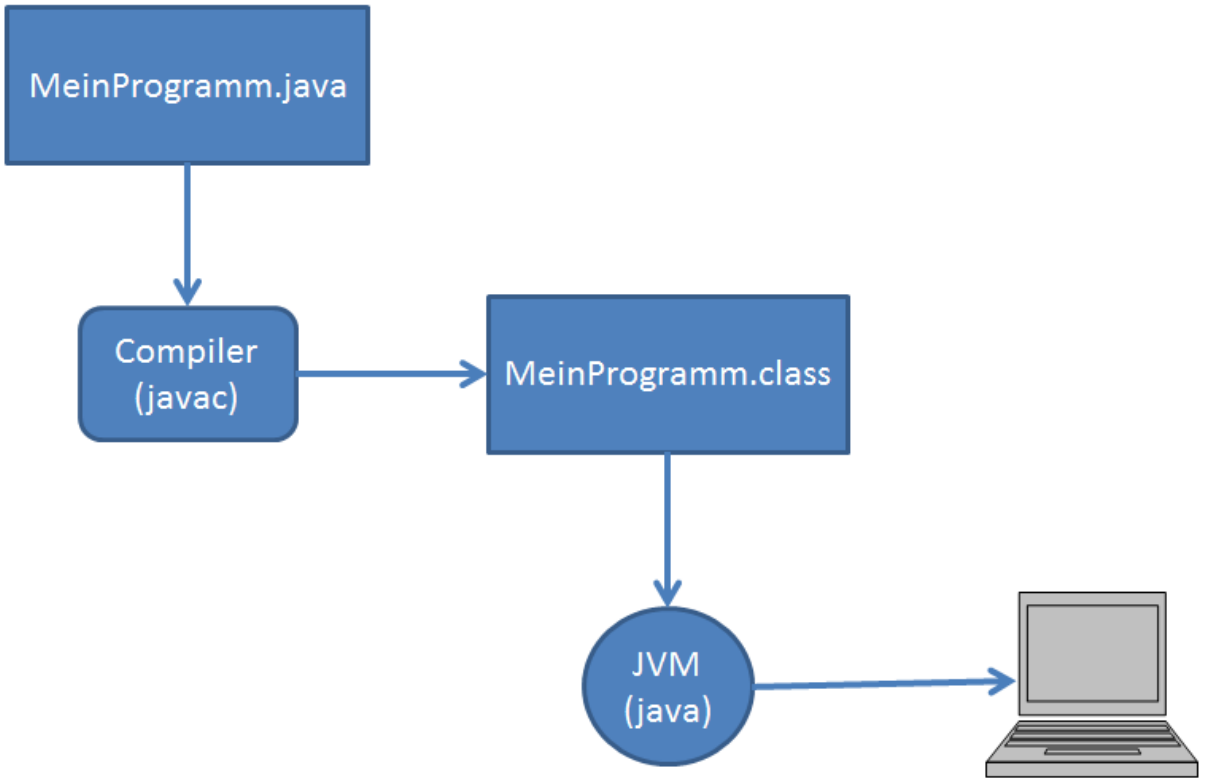
```
// Instanz einer normalen Kuh
// Kuh-Referenz zeig auf Kuh
Kuh kuh = new Kuh();

// Instanz einer SuperKuh
// Kuh-Referenz zeigt auf
// SuperKuh(Polymorphie)
Kuh sKuh = new SuperKuh();

// Eine normale Kuh schlaeft
// 4 Stunden am Tag
kuh.schlafen();

// Superkuehe schlafen nur
// 1 Stunde am Tag
sKuh.schlafen();
```

Zeit für eine Übung



Exkurs

Java - Von der Übersetzung zur Ausführung



Aufbau von Java-Dateien

- Jede .java Datei enthält eine Klassendefinition
- Wichtig: Dateiname = Klassenname
- Klassen enthalten Methoden
- Methoden enthalten Anweisungen

Test.java

```
//Klassendefinition
class Test {
// Der Methodenkopf
void test(){
//Eine Anweisung
System.out.println("Dies ist ein
     Test");
}
}
```

Aufbau von Java-Dateien

- Programm besteht (meistens) aus mehreren Klassen
- Eine Klasse beeinhaltet eine Main-Methode
- JVM startet Main-Methode
- Programm endet nach Ablauf der Main-Methode

TestMitMain.java

```
class TestMitMain {
public static void main(String[]
      args){
System.out.println("Dies ist ein
      Test");
}
}
```

Methoden

Methoden besitzen:

1. Einen Methodennamen
2. Anweisungen
3. Lokale Variablen
4. 0..* Parameter
5. 0..1 Rückgabewerte (ohne Rückgabe: void)
6. Wenn Rückgabewert dann "return wert;"
7. Zugriff auf Klassenattribute
8. Können überladen werden

```
Rueckgabetyp Methodenname (Typ
     Parameter, ...){
Anweisung1;
...
AnweisungN;

return wert;
}
```

Datentypen

- Java enthält 8 primitive Datentypen
- Primitive Datentypen sind keine Objekte

Name	Wertebereich	Standard
byte	-128 bis 127	0
short	-32768 bis 32767	0
int	-2147483648 bis 2147483647	0
long	-2^{63} bis $2^{63}-1$	0
float	+/-3.40282347*1038	0.0
double	+/-1.79769313486231570*10308	0.0
char	Unicode-Zeichen	u0000
boolean	false, true	false

Variablen

- Es gibt in Java drei Arten von Variablen
  1. Klassenvariablen
  2. Lokale Variablen
  3. Instanzvariablen
- Variablen haben immer einen Datentyp

```
class Variablen {

//Klassenvariable
static boolean bool = true;

//Instanzvariable
int i = 5;

public static void main(String[]
      args){

//lokale Variable
int i = 1;

}
}
```

Variablen

Lebensdauer

- Klassenvariablen: Gesamte Programmlaufzeit
- Lokale Variablen: Bis zum Ende des Methodenaufrufs
- Instanzvariablen: Existenz des Objekts

Sichtbarkeit

- Klassenvariablen: Innerhalb der Klasse
- Lokale Variablen: Innerhalb eines Blocks
- Instanzvariablen: Innerhalb des Objekts

Casting

- Java kann Typen implizit casten
- Entwickler kann Typen explizit casten
- Casten in größeren Datentyp geht implizit
- Casten in kleineren Datentyp explizit da Informationsverlust!

```
int i = 10;

// Cast in groesseren Typ
// kein Problem
long l = i;

// Cast in kleineren Typ explizit
short s = (short) i;
```

Operatoren für numerische Datentypen:

Name	Erläuterung
+	Addition, pos. Vorzeichen
-	Subtraktion, neg. Vorzeichen
*	Multiplikation
/	Division
%	Modulo
++	Prä -/ Postinkrement
--	Prä -/ Postdekrement

Vergleichsoperatoren

Name	Erläuterung
==	Gleich
!=	Ungleich
<	Kleiner
>	Größer
<=	Kleiner gleich
>=	Größer gleich

Logische Operatoren

Name	Erläuterung
&&	UND (Shortcircuit)
||	ODER (Shortcircuit)
!	NICHT
&	UND
|	ODER
^	Exklusiv ODER

Operatoren und Ausdrücke

- Verknüpfung von Variablen und anderen Ausdrücken mit Operatoren
- Klammern beeinflussen die Reihenfolge der Auswertung
- Auswerten von Teilausdrücken von links nach rechts

Kontrollstrukturen - Block

- Fasst mehrere Anweisungen Zusammen
- Kann stehen wo auch einzelne Anweisungen stehen
- Kann geschachtelt werden

```
{
System.out.println("Ausgabe");
System.out.println("Ausgabe");
}
```

- Bedingte Anweisung
- Ausdruck "Bedingung" muss boolescher Ausdruck sein
  d.h. *true* oder *false*

```
if(Bedingung){
System.out.println("Die Bedingung
      ist True");
}
```

Kontrollstrukturen - Fallunterscheidung

- Mehrfachauswahl
- Auch hier:
  Ausdrücke "Bedingung1" und "Bedingung2" müssen boolesche Ausdrücke sein
- else-Zweig wird durchlaufen wenn kein Ausdruck wahr ist

```
if(Bedingung1){
System.out.println("Bedingung1 ist
      True");
} else if(Bedingung2){
System.out.println("Bedingung2 ist
      True");
} else {
System.out.println("Keine Bedingung
      ist True");
}

// If-Then-Else als
// ternaerer Operator
Ausdruck = ( BedingungX ) ? Wahr :
      Falsch;
```

Kontrollstrukturen - Fallunterscheidung

- Mehrfachauswahl
- Ausdruck vom Typ: byte, short, char, int
- Nach case darf genau 1 Konstante stehen
- Wenn keine passende Konstante dann "default" Anweisung (wenn vorhanden)
- Ohne break: ausführen aller Anweisungen ab Übereinstimmung
- "break" ist syntaktisch nicht erforderlich

```
switch(Ausdruck){

case konst1:
Anweisung1;
break;

case konst2:
Anweisung2;
break;

default:
Anweisung3;
break;

}
```

- Kopfgesteuerte Schleife
- Prüft Ausdruck zu Beginn

```
while (Bedingung){
Anweisung1;
...
Anweisungn;
}
```

- Fußgesteuerte Schleife
- Prüft Ausdruck am Ende der Schleife

```
do{
Anweisung1;
...
Anweisungn;
}while (Bedingung);
```

- Zählschleife
- Auch Kopfgesteuert
- "Initial" wird vor 1. Durchlauf ausgewertet
- "Bedingung" wird vor jedem Durchlauf ausgewertet
- "Inc/Dec" wird nach jedem Durchlauf ausgewertet

```
for(Initial; Bedingung; Inc/Dec)
{
Anweisung1;
...
AnweisungN;
}
```

Arrays

- Arrays fassen Variablen mit gleichem Typ zusammen
- Zugriff via Index
- Array ist ein Objekt
- Kann primitiven Datentyp oder Objektreferenzen enthalten (nicht veränderbar)
- Elementanzahl wird in public Attribut "length" gespeichert
- Kann mehrdimensional sein

```
// Array deklarieren
int[] noten;

// Int-Array erzeugen
// Mit Platz fuer 25
// Elemente
noten = new int[25];

// Werte zuweisen
// Erste Element an
// Stelle 0
noten[0] = 5;
noten[1] = 4;
noten[2] = 6;

// Auch moeglich
// Array mit 3 Elementen
int[] ar = {1,4,5};

// Mehrdimensionales Array
int[][] tabelle = new int[4][4];
```

Strings

- Strings sind Objekte
- Können ohne "new" angelegt werden
- Konstruktoren existieren auch
- Strings sind immutable (nicht veränderbar)
- Ändern eines Zeichens erzeugt neuen String
- Vergleich mit equals-Methode

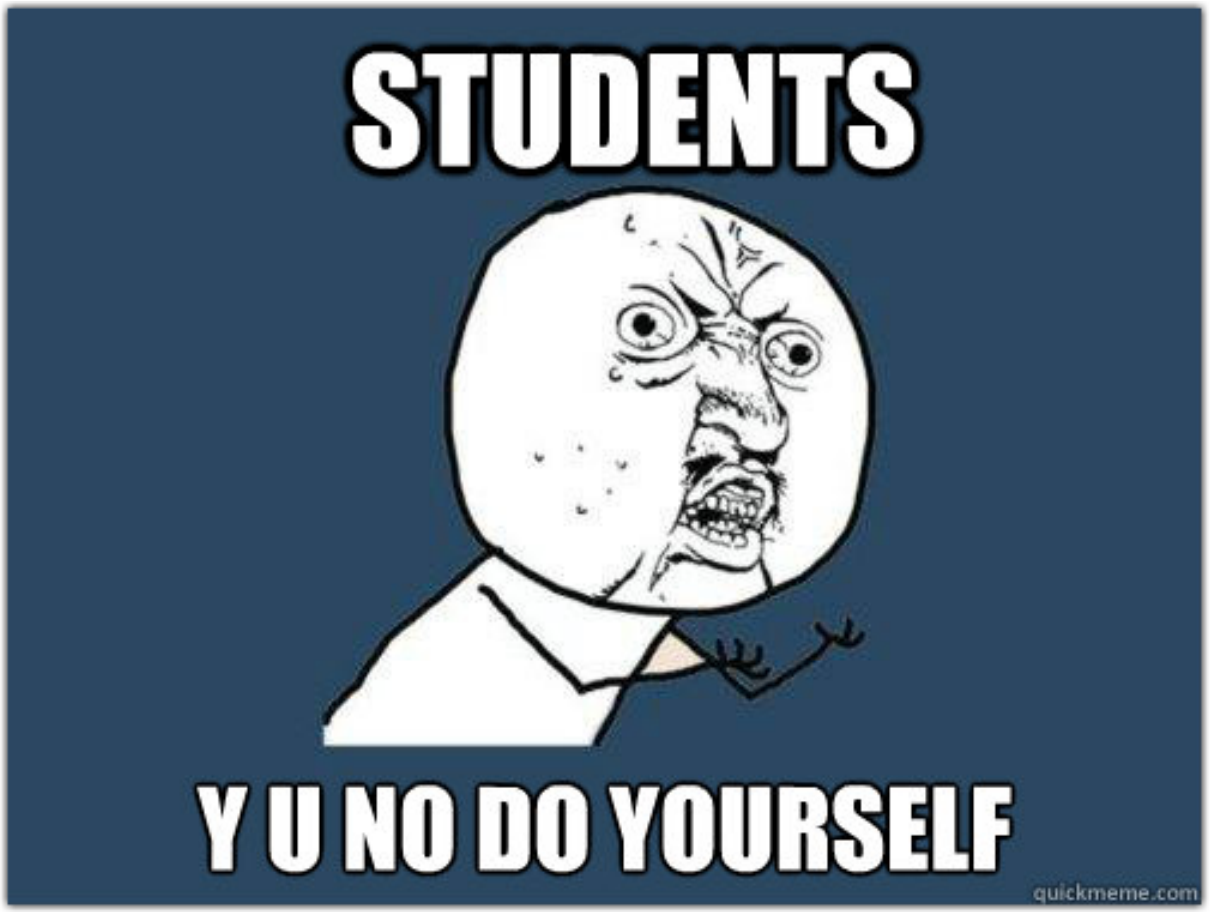
```
// Erstellen ohne ''new''
String farbe = "rot";
String farbe2 = "blau";

   // Klasse hat viele Methoden
   // Hier 3 Beispiele,
   // mehr sind in der API-Doku

// Gibt die Laenge zurueck
int laenge = farbe.length();

// Gibt das Zeichen
// an Position 2
// zurueck
// Wichtig: 1. Zeichen
// steht an Position 0
char c = farbe.charAt(2);

// Ein Vergleich
// Ergebnis: false
farbe.equals(farbe2);

// Noch ein Vergleich
// Ergebnis: true
"blau".equals(farbe2);
```

Strings

- "+"-Operator fügt Strings zusammen
- Andere Datentypen werden beim Zusammenfügen automatisch in Strings umgewandelt

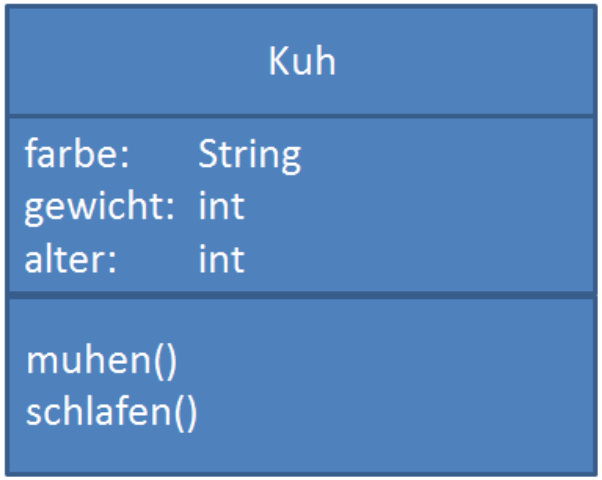
```
// String concat
String a = "rot";
String b = "blau";
String c = "gruen";
System.out.println(a+b+c);

// Umwandlung in String
int i = 10;
String prefix = "Geld: ";
String postfix = "Euro";
String message = prefix + i +
     postfix;
System.out.println(message);
```

Zeit für eine Übung



Klassen und Objekte

Vereinfachung der Kuh-Klasse:




```
public class Kuh
{
private int alter;
private int gewicht;
private String farbe;

public void muhen(){
System.out.println("Muh");
}

public void schlafen(){
...
}
}
```

Klassen in Java

- Eine Java-Klasse besteht aus:
  1. Klassennamen
  2. Konstruktoren
  3. Methoden
  4. Attributen (Instanz- / Klassenvariablen)

```
Modifier class Klassenname{
//Haben Typ und Variablennamen
Attribute

// Spezielle Methoden die nach
// der Instanziierung aufgerufen
// werden
Konstruktoren

// Besitzen ggf. Parameter und
// Rueckgabewert
Methoden
}
```

Modifier

- Kann Klassen, Methoden, Attributen vorangestellt werden
- Hat Einfluss auf:
  1. Sichtbarkeit
  2. Lebensdauer
  3. Modifikation
- Attribute sind private (Geheimnisprinzip)
- Beispiele:
  1. public
  2. private
  3. final
  4. ...

Sichtbarkeit

- 3 Modifier beeinflussen die Sichtbarkeit:
  1. private
  2. protected
  3. public
- Default ist "package"

```
// Nur innerhalb der
// eigenen Klasse sichtbar
private int alter;

// Sichtbar fuer abgeleitete
// Klassen und Klasesn im
// selben Package
protected int gewicht;

// Ueberall sichtbar
// Nur Public-Klassen koennen
// ausserhalb des eigenen Pakets
// genutzt werden
public String farbe;
```

Modifier "final"

- Anwendung auf:
  1. Methoden
  2. Variablen
  3. Klassen
- Verhindert die nachträgliche Veränderung
- Konstanten = static + final Attribute

```
// final auf Klassenebene
     verhindert
// das Erben von Kuh
public final class Kuh{
...
// Kann nur 1 mal beim
// direkten Initialisieren
// oder im Konstruktor
// gesetzt werden
private final String farbe;
...
// Kann nicht durch Vererbung
// veraendert werden
public final void muhen(){
...
}
```

Packages

- Klassen werden in Paketen verwaltet
  1. Zur besseren Darstellung der Struktur
  2. Zur Vermeidung von Namenskonflikten
  3. Zur Zugriffsbeschränkung
- Jede Klasse liegt im thematisch passenden Package

Packages

- Imports
  1. Eingabe des FQN ist umständlich
  2. Lösung: Import-Anweisung
  3. Imports am Anfang jeder Java-Datei
  4. macht Angabe des FQN überflüssig
- Paket java.lang wird automatisch importiert

```
// Wenn mehr aus Util
// verwendet wird:
// import java.util.*;
import java.util.Calendar;

public class HatKalender{
  //statt: java.util.Calendar c =
        ...;
Calendar c = Calendar.getInstance
     ();
}
```

Mit Java mitgelieferte Pakete:
(Auszug, es gibt weit mehr!)

Name	Wertebereich
java.lang	Basiskonstrukte
java.util	Hilfsklassen (Datenstrukturen etc. . .)
java.io	Input-/Output
javax.swing	GUI

Packages

- Eigene Klasse wird mit "package" in ein Paket gepackt
- Steht als als erstes (!) in jeder Java-Datei
- Namen immer klein geschrieben
- Java-Dateien liegen in entsprechendem Verzeichnis
- Ohne Angabe landet Klasse in Default-Paket (aktuelles Verzeichnis)

```
// Ab jetzt liegt unsere
// Klasse im Package
// Wichtig: Im Filesystem
// liegt die Klasse nun auch
// unter:
// ./com/fom/oop/bauernhof
package com.fom.oop.bauernhof;

// Alle Imports die
// wir benoetigen
import ...;
...

public class Kuh{
...
}
```

Objekte in Java

- Objekte werden mit dem Operator "new" erzeugt
- Objekte werden dynamisch auf Heap erzeugt
- Variablen können Referenzen auf Objekte aufnehmen
  Nicht die Objekte selbst!

```
// Variable vom Typ Kuh
// Kann eine Referenz auf
// ein Kuh-Objekt aufnehmen
Kuh milkaKuh;

// Neues Objekt erzeugen und
// Referenz der Variable zuweisen
milkaKuh = new Kuh();
```

Ein Objekt der Klasse Kuh:

- Attribute sind private
- Nur Zugriff auf Objektmethoden möglich
- Attribute der Klasse sind nicht initialisiert Standardwerte: alter = 0, gewicht = 0, farbe = null
- Attribute setzen durch: Methoden oder Konstruktor

Unser Kuh-Bauplan:

```
public class Kuh
{
private int alter;
private int gewicht;
private String farbe;

public void muhen(){
System.out.println("Muh");
}

public void schlafen(){
...
}
}
```

Konstruktoren

- Spezielle Methode
- Wird direkt nach Objekterzeugung aufgerufen
- Gleicher Name wie Klasse
- 0..* Parameter
- Kein Rückgabewert
- Klassen kann mehrere Konstruktoren besitzen
- Besitzt Klasse keinen Konstruktor wird default-Konstruktor aufgerufen

```
public class Kuh
{
private int alter;
private int gewicht;
private String farbe;

//Konstruktor
public Kuh(int a, int g, String f){
alter = a;
gewicht = g;
farbe = f;
}

//Konstruktor
public Kuh(int a, int g){
alter = a;
gewicht = g;
farbe = "Schwarz-Weiss";
}

public void muhen(){
System.out.println("Muh");
}

public void schlafen(){
...
}
}
```

Initialisierung mit Konstruktoren

- Konstruktor wirkt sich auf Art der Instanziierung auf
- Ohne parameterlosen Konstruktur müssen bei "new" Parameter übergeben werden (gemäß Konstruktoren)

```
// Deklaration der Variable
Kuh elsa;

// Intanziieren des Objekts
elsa = new Kuh(10, 350, "lila");

// Intanziieren des Objekts
elsa = new Kuh(10, 350);
```

Methodenaufrufe

- Durch Methodenaufruf wird Objekt eine Nachricht geschickt
- Aufruf:
  Referenzvariable.
  Methodenname(Parameter. . .);

```
// Deklaration der Variable
Kuh elsa;

// Intanziieren des Objekts
elsa = new Kuh(10, 350, "
     lila");

// Aufruf von Objektmethoden
elsa.schlafen();
elsa.muhen();
```

Objekterzeugung

- Von einer Klasse können mehrere Objekte erzeugt werden
- Jedes Objekt ist individuell auch wenn die Attribute die selbern Werte haben
- Achtung:
  Beim Vergleich werden die Referenzen verglichen

```
// Unsere Elsa
Kuh elsa = new Kuh(10,350, "lila");

// Unsere Bertha
Kuh bertha = new Kuh(10, 350, "lila"
     );

// Vergleich liefert
// false zurueck
if(elsa == bertha){
...
}
```

Getter und Setter

- Unsere Kuh bekommt das Feld "anzBesuchteHoefe"
- "anzBesuchteHoefe" ist private
- Für Zugriff werden get-/set-Methoden implementiert
- Setter ändert den Wert des betroffenen Attributs
- Getter gibt aktuellen Wert zurück
- Namen der Methoden beginnen mit "get" oder "set"

```
public class Kuh
{
private int alter;
private int gewicht;
private String farbe;
private int anzBesuchteHoefe;

//Konstruktor
public Kuh(int a, int g, String f){
alter = a;
gewicht = g;
farbe = f;
}

//Konstruktor
public Kuh(int a, int g){
alter = a;
gewicht = g;
farbe = "Schwarz-Weiss";
}

public int getAnzBesuchteHoefe(){
return anzBesuchteHoefe;
}

public void setAnzBesuchteHoefe(int
     anz){
this.anzBesuchteHoefe = anz;
}
...
}
```

Getter und Setter

- Setter ändert Objektzustand
- "this" ist Referenz auf das Objekt selbst
- Durch "this" können Objektvariablen verwendet werden, die durch lokale überdeckt sind

```
// Unsere Kuh Elsa
Kuh elsa = new Kuh(10, 350);

// Elsa hat schon 5 Hoefe besucht
elsa.setAnzBesuchteHoefe(5);

// Ausgabe der Anzahl
System.out.println(elsa.
     getAnzBesuchteHoefe());
```

Modifier "static"

Statische Methoden:

- Implementiert Verhalten unabhängig von Objektzustand
- Z.B. für mathematische Funktionen
- Objekt in dem Fall nicht sinnvoll
- Steht ab Laden der Klasse bis Ende des Programms zur Verfügung
- Nicht an Objektexistenz gebunden
- Beispiel: Main-Methode
- Aufruf über Klassenname

```
public class MatheLib {
// Privater Konstruktor
// Instanziierung nicht
// mglich
private MatheLib(){}

// Static Methode
// kann auch ohne Objekt
// aufgerufen werden
public static int sum(int z1, int
     z2){
return z1+z2;
}
}
```

```
// Aufruf
// summe = 2
int summe = MatheLib.sum(1,1);
```

Modifier "static"

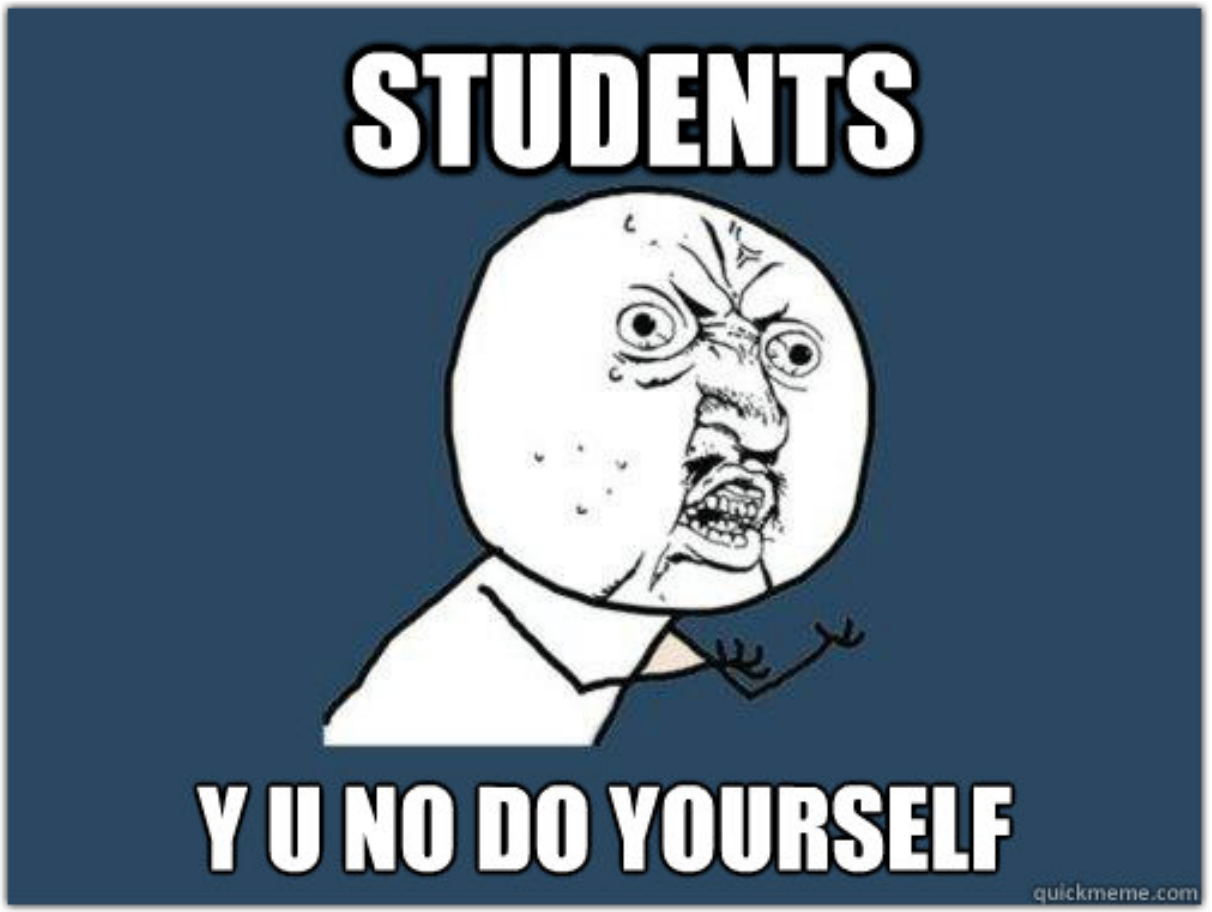
Statische Attribute:

- Klassenvariablen
- Nicht an Objekte gebunden
- Klassenvariablen werden von allen Objektinstanzen geteilt
- Verwendung: z.B. als Objektzähler
- Initialisierung vor Objekterzeugung
- Initialisierung vor Aufruf statischer Methoden

Zu Beachten

- Keine Aufrufe von non-static Methoden aus static Methoden heraus
- Keine Verwendung von non-static Objektvariablen innerhalb von static Methoden

Zeit für eine Übung



Vererbung

Vererbung mit Java

- Abgeleitete Klasse erbt alle non-private Eigenschaften
- Geerbte Methoden können überschrieben werden
- Abgeleitete Klasse kann zusätzliche Methoden enthalten
- eine Klasse erbt mit dem Keyword "extends"
- Konstruktor der Basisklasse muss aufgerufen werden

Default-Konstruktor:

```
public class Kuh{
private int alter;
private int gewicht;

public void muhen(){
System.out.println("Muh!");
}
}

public class SuperKuh extends Kuh{
// Ueberschreiben der
// Methode von Kuh
public void muhen(){
System.out.println("Supermuh!!!!");
}

// Neue Methode
public void fliegen()
// fliegen
}
}
```

Spezifischer Konstruktor:

```
public class Kuh{
private int alter;
private int gewicht;

public Kuh(int alter, int gewicht){
this.alter = alter;
this.gewicht = gewicht;
}
...
}

public class SuperKuh extends Kuh{
// Parameter duerfen sich
// unterscheiden, jedoch
// muss Konstruktor der Vaterklasse
// aufgerufen werden
public SuperKuh(){
super(5,350);
}
...
}
```

Vererbung und Eigenschaften

- Automatischer Aufruf des Default-Konstruktors (wenn kein spezieller)
- Superkuh kann nicht direkt auf Variablen von Kuh zugreifen
- Zugriff nur über public Methoden
- Die Methode "muhen" wird überschrieben
- Die Methode fliegen() gibts nur bei Superkühen

Überschreiben vs Überladen

Überschreiben

- Methodensignatur ist in Vater- und Kindklasse gleich
- Abgeleitete Klasse des ursprünglichen return-Werts möglich
- Sichtbarkeit darf nicht eingeschränkter werden
- Nutzung bei Vererbung

Überladen

- Methoden mit gleichen Namen in Klasse
- Methoden unterscheiden sich hinsichtlich Parameter
- Rückgabewerte können unterschiedlich sein
- Sichtbarkeit kann sich ändern

Vererbungsstufen

- Vererbungsstufen sind nicht begrenzt
- Von SuperKuh kann wieder geerbt werden
- Jede Kind-Klasse erbt alle Eigenschaften aus ihrer Hierarchie (Gesetz den Vorgaben von Java)
- Von Klassen kann auch mehrfach geerbt werden (MilkaKuh erbt von Kuh)
- Vererbung beschreibt eine "ist eine" - Beziehung
- Eine Superkuh ist eine Kuh
  (Aber nicht jede Kuh ist eine Superkuh)

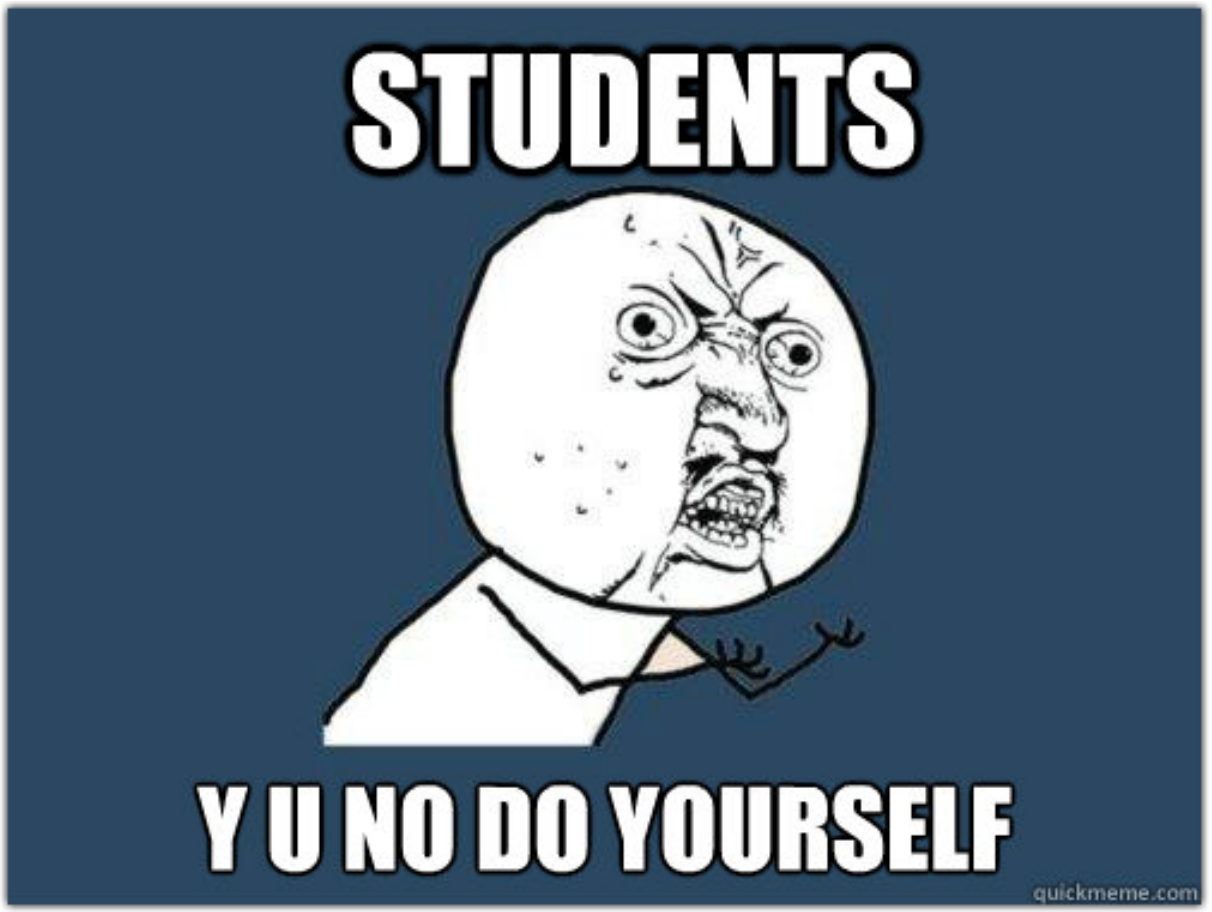
Die Klasse Object

- Alle Klassen erben automatisch von Object
- Erbt eine Klasse von niemanden wird "extends Object" implizit angefügt
- Von Object geerbte Methoden:
  1. boolean equals(Object obj)
  2. protected Object clone()
  3. String toString()
  4. int hashCode()

Vorteile von Vererbung

- Bessere Wartbarkeit
- Fördert Wiederverwendung
- Bessere Erweiterbarkeit

Zeit für eine Übung



Polymorphismus

- Die Kindklasse kann überall dort benutzt werden wo auch die Vaterklasse genutzt wird
- Während des Kompilierens steht nicht fest welche Methode ausgeführt wird
- Es kann Methode der Basisklasse sein, oder eine einer Kindklasse

```
// Kuh-Referenz zeigt auf
// SuperKuh-Objekt
Kuh hilde = new SuperKuh();

// Es wird das ''muhen''
// einer SuperKuh ausgefuehrt
// Ausgabe: Supermuh!!!
hilde.muhen();
```

Möglichkeiten des Polymorphismus

- Kein Wissen über konkrete Klasse oder Implementierung von Methoden nötig
- Unterschiedliche Methoden für gleiche Referenztypen
- Wichtig für Erweiterbarkeit

- Compiler checkt den Referenztyp nicht das Objekt
- Daher können nur Methoden aufgerufen werden, die der Referenz-Klasse angehören
- Lösung: Casting

```
// Es koennen nur Methoden
// der Kuh-Klasse aufgerufen
// werden
Kuh hilde = new SuperKuh();

// Compiler wirft einen Fehler
hilde.fliegen();

// Um den Fehler zu vermeiden
// Casten wir unsere Hilde
SuperKuh superHilde = null;
if(hilde instanceof SuperKuh){
superHilde = (SuperKuh) hilde;
}
superHilde.fliegen();
```

Achtung

- Referenz der Kindklasse auf die abgeleitete Klasse funktioniert nicht
- Fehler:
  SuperKuh heldin = new Kuh();
- Der Objekttyp kann die Methoden des Referenztyps nicht bedienen

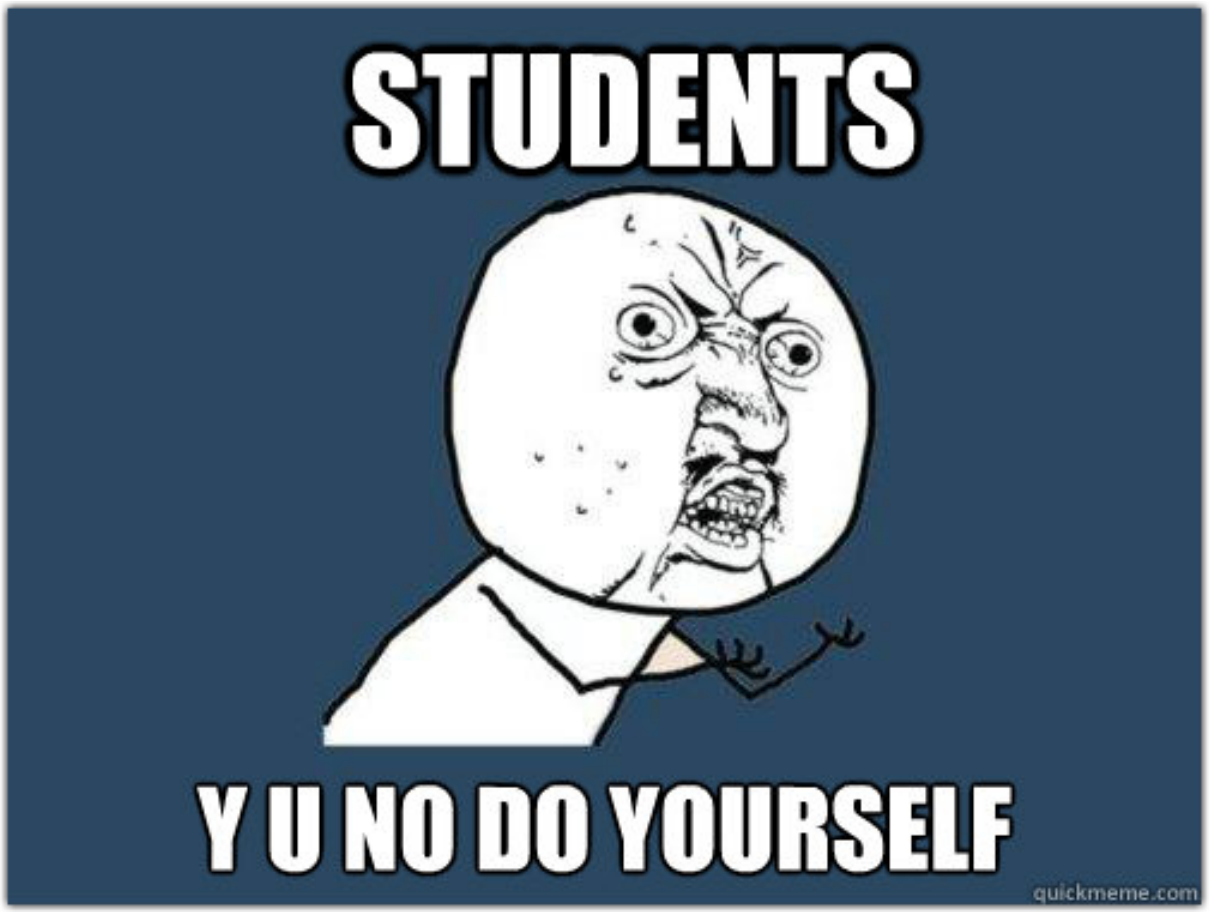
Dynamisches Binding

- Da Methode nicht bei Kompilierung bestimmt werden kann muss JVM sie zur Laufzeit suchen
- Start bei der speziellsten Klasse in Richtung allgemeinere

Statisches Binding

- Compiler kann zur Compilezeit die Methode zuordnen
- Dazu muss die Methode
  - private oder
  - static oder
  - final sein

Zeit für eine Übung



Abstrakte Klassen

Abstrakte Klassen

- Abstrakte Klassen beschreiben Klassen die zu allgemein sind um eigenes Objekt zu sein
- Beispiel ist die Klasse Tier
- Es kann nicht einfach nur ein Tier-Objekt existieren
- Mit dem Schlüsselwort "abstract" wird die Instanziierung einer Klasse verhindert
- Abstrakte Klassen können abstrakte Methoden enthalten
  1. "abstract" muss vor Rückgabetyp stehen
  2. Semikolon am Ende der Signatur
  3. Keinen Rumpf, nur Signatur
  4. Macht enthaltende Klasse automatisch abstrakt, also muss die Klasse auch abstrakt sein

Abstrakte Klassen

- Abstrakte Klassen werden durch konkrete Klassen genutzt
- Abstrakte Klasse kann als Referenztyp genutzt werden
- Abstrakte Klasse kann auch lediglich aus konkreten Methoden bestehen
- Implementiert Kind-Klasse eine abstrakte Methode nicht, muss Kind auch abstrakt sein
- Es kann nur von einer Klasse gleichzeitig geerbt werden

```
// Unsere abstrakte Klasse
abstract class Tier{
// unsere abstrakte Methode
abstract void gattung();
}

public class Kuh extends Tier{
public Kuh(){}
...
// Diese Methode muss ueberschrieben
// werden, ansonsten muss Kuh auch
// ''abstract'' sein
void gattung(){
System.out.println("Ich bin eine Kuh
     ");
}
...
}

public class KuhTest{
 public static void main(String[]
      args){
  // Das geht schief!
  Tier tierchen = new Tier();
  // So geht es richtig:
  // Abstrakte Klasse kann
  // auch Referenztyp sein
  Tier kuh = new Kuh();
  kuh.gattung();
 }
}
```

Interfaces

Interfaces

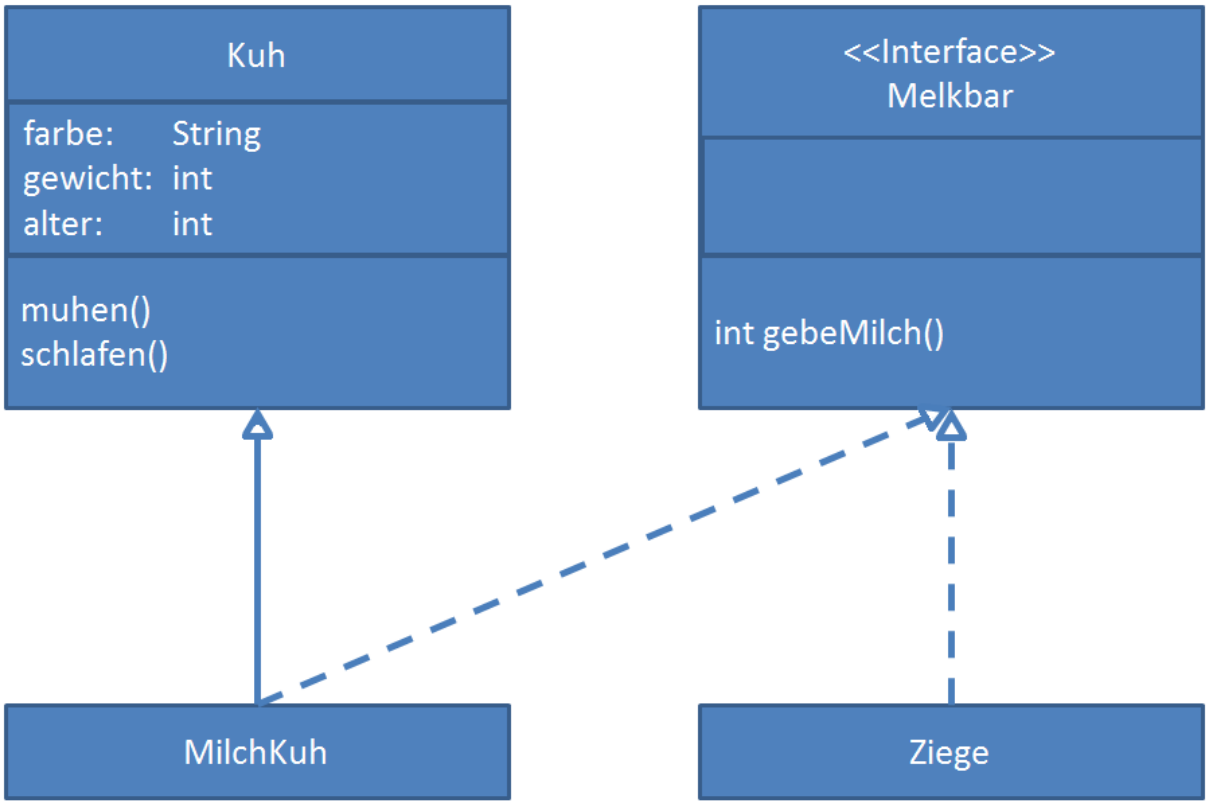
- Schnittstellen sind spezielle Klassen
- Sie enthalten nur abstrakte Methoden sowie Konstanten
- Klassen können mehrere Interfaces gleichzeitig implementieren
- Kann von unterschiedlichen Klassen implementiert werden
- Methoden sind immer public und abstract (default)
- Wird auch als Vertrag bezeichnet

Interfaces

- Interfaces verwenden nicht "class" sondern "interface"
- Werden durch "implements" implementiert
- Mehrere Interfaces in einer Klasse werden hintereinander durch Kommata getrennt
- Schnittstellen werden durch Kinder weiter vererbt
- Interfaces erben mit "extends" von anderen Interfaces

```
public interface Melkbar{
// Gibt eine Anzahl in Litern wieder
// (vereinfacht)
int gebeMilch();
}

public class MilchKuh extends Kuh
     implements Melkbar{
...
// 5 Liter Milch
private int milch = 5;

public int gebeMilch(){
int anzahl = 5;
milch -= anzahl;
return anzahl;
}
}

public class Ziege extends Tier
     implements Melkbar{
...
// 3 Liter Milch
private int milch = 3;

public int gebeMilch(){
int anzahl = 3;
milch -= anzahl;
return anzahl;
}
}
```

Auszug der Vererbungsstruktur

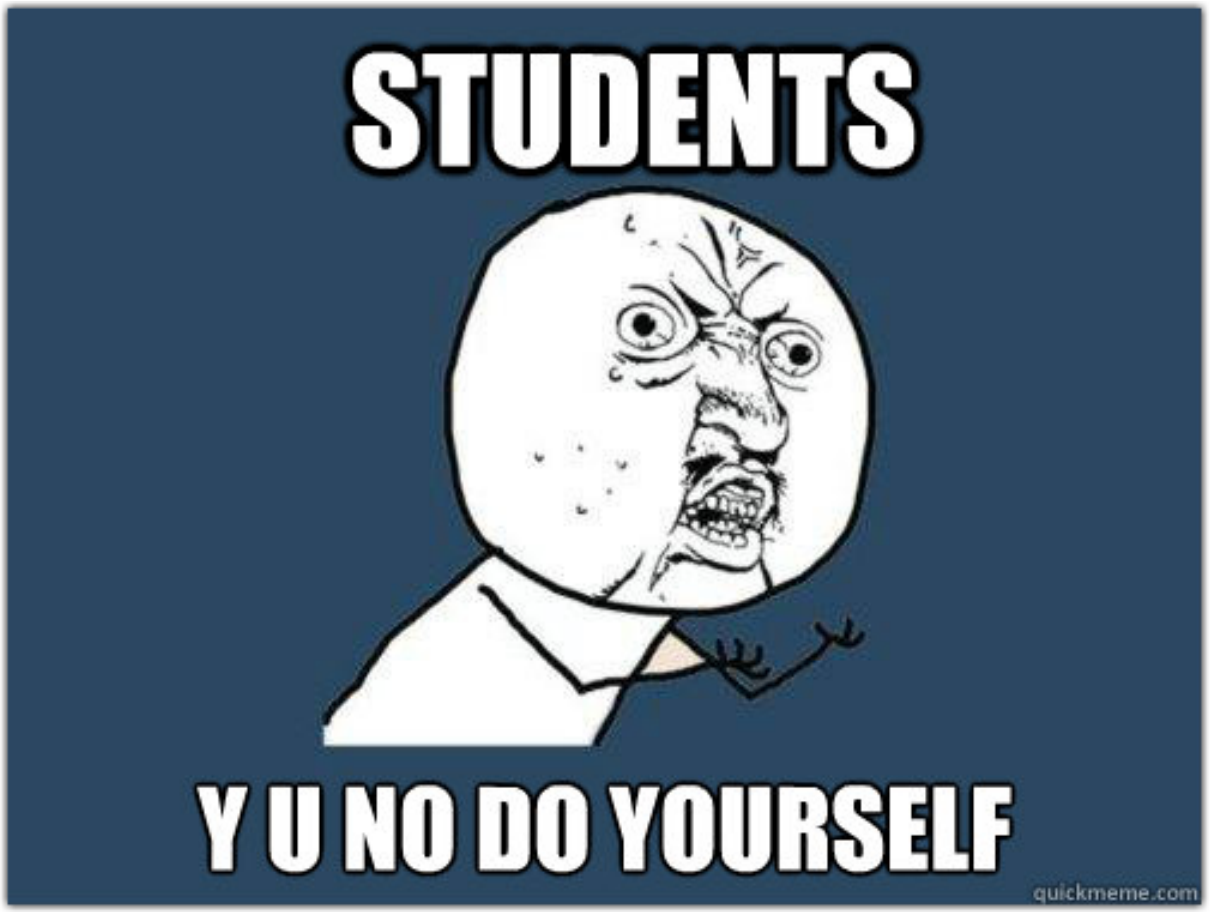


Interfaces in Aktion

Ziegen und Kühe können jetzt gemeinsam als melkbare Objekte behandelt werden

```
Melkbar[] zuMelken = new Melkbar[2];
int gesamtMilch = 0;

MilchKuh frida = new MilchKuh();
Ziege susi = new Ziege();

zuMelken[0] = frida;
zuMelken[1] = susi;

for(int i = 0; i < zuMelken.length;
      i++){
gesamtMilch += zuMelken[i].gebeMilch
      ();
}
```

Zeit für eine Übung



Prinzipien für den objektorientierten Entwurf

- Geheimnisprinzip
- Kopplung
- Gesetz von Demeter
- Kohäsion
- Separation of Concerns
- SOLID

Geheimnisprinzip

- Abkapselung von Attributen und innerer Logik von der Außenwelt
- Methoden werden zu Schnittstellen
- Realisierung durch Modifier:
  1. public
  2. private
  3. protected
  4. package

Kopplung

- Abhängigkeit zwischen Klassen
- Kopplung definiert Grad der Abhängigkeit
- Zeigt Einfluss von Änderungen einer Klassen auf andere Klassen auf
- Ziel: Lose/Geringe Kopplung zwischen Klassen
- Lässt sich über Metriken messen

Kohäsion

- Grad des Zusammenhangs aller Verantwortlichkeiten, Daten und Methoden einer Klasse
- Ziel ist eine hohe Kohäsion innerhalb einer Klasse/Methode...
- Lässt sich über Metriken messen

Gesetz von Demeter

- Objekte sollen nur mit Objekten in ihrer direkten Umgebung kommunizieren
- Eine Methode sollte nur andere Methoden verwenden
  - Methoden der eigenen Klasse
  - Methoden der übergebenen Parameter
  - Methoden von assoziierten Klassen
  - Methoden von selbst erzeugten Objekten

Gesetz von Demeter

Negatives Beispiel:

```
class Motor {
void starten(){}
}

class Auto {
private Motor motor;

public Auto() {
motor = new Motor();
}

public Motor getMotor(){
return motor;
}
}

class Fahrer(){
void fahren(){
Auto a = new Auto();
Motor m = a.getMotor();
m.starten();
}



}
```

Positives Beispiel:

```
class Motor {
void starten(){}
}

class Auto {
private Motor motor;

public Auto() {
motor = new Motor();
}

public Motor getMotor(){
return motor;
}

public void starten(){
motor.starten();
}
}

class Fahrer(){
void fahren(){
Auto a = new Auto();
a.starten();
}
}
```

Separation of Concerncs

- Aufteilung komplexer Programme/Systeme/Funktionalität nach Verantwortlichkeiten
- Beispielsweise durch Subsysteme oder Schichten
- Vorteile
  1. Erhöhte Qualität durch Nachverfolgbarkeit von Änderung und Wirkung
  2. Leichtere Austauschbarkeit
  3. Erhöhte Wiederverwendung

- *S* ingle Responsibility Principle
- *O* pen Closed Principle
- *L* iskov Substitution Principle
- *I* nterface Segregation Principle
- *D* ependency Inversion Principle

Single Responsibility Principle (SRP)

- Jede Klasse sollte nur genau EINE fest definierte Aufgabe erfüllen
- responsibility = 'reason for change'
- „Es sollte nie mehr als einen Grund geben eine Klasse zu ändern.“
- Einhalten dieses Konzepts impliziert einen sehr hohen Grad der Kohäsion

Open Closed Principle (OCP)

- Klassen sollen offen für Erweiterungen aber geschlossen für Modifikation sein
- Erweiterung: Existierendes Verhalten wird nicht geändert
- Modifikation: Änderung von bestehendem Verhalten

Negatives Beispiel:

```
int melke(Tier t){
if(t instanceof MilchKuh){
t.gebeMilch();
} else if(t instanceof Ziege){
t.gebeMilch();
}
}
```

Positives Beispiel:

```
int melke(Melkbar m){
m.gebeMilch();
}
```

Liskov Substitution Principle (LSP)

- Klassen sollen durch dessen Subklassen ersetzbar sein
- "Unterklassen sollen nicht mehr erwarten und nicht weniger liefern als ihre Oberklassen"

LSP Verstoß:

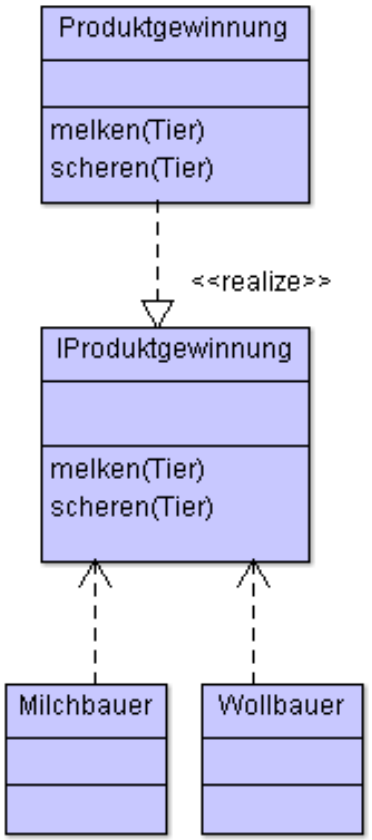
- Kuh spezial = new SpezialKuh();
- spezial.setFarbe("blau", "rot");
- Annahme: spezial.getFarbe2() = rot

Beispiel für Verstoß:

```
public class Kuh{
String farbe1;
String farbe2;

void setFarbe(String f1, String f2){
farbe1 = f1;
farbe2 = f2;
}

String getFarbe1(){
return farbe1;
}

String getFarbe2(){
return farbe2;
}
}

public class SpezialKuh extends Kuh{
void setFarbe(String f1, String f2){
farbe1 = f1;
farbe2 = f1;
}
}
```

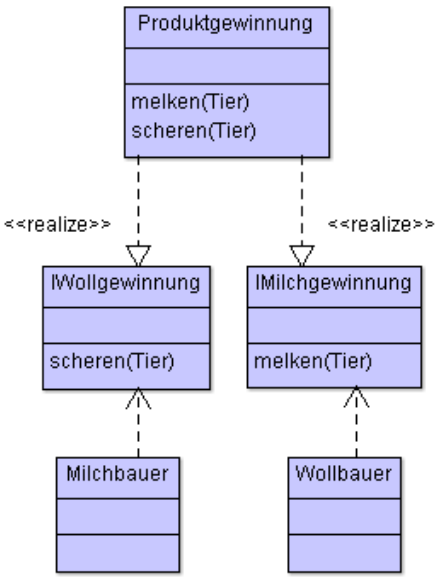
Interface Segregation Principle (ISP)

- Kopplung erhöht durch Verwendung einer globalen Schnittstelle
- Änderungen wirken auf beide Nutzer



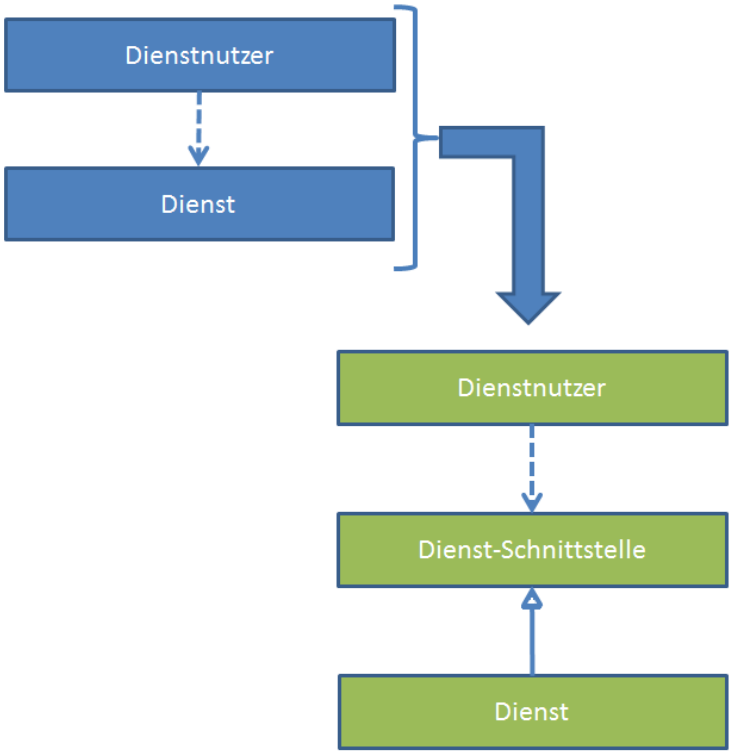
Interface Segregation Principle (ISP)

- Schnittstellen sollten speziell auf ihre Aufgaben zugeschnitten sein
- "Clients sollten nicht von Methoden abhängen die sie nicht benutzen"



Dependency Inversion Principle (DIP)

- Module höherer Ebenen sollten nicht von niedrigeren Modulen abhängen
- Abstraktionen hängen nicht von Details ab, sondern Details von Abstraktionen
- Abhängigkeit immer von konkreten zu abstrakten Programmteilen
- Reduziert Kopplung
- 2 Grundkonzepte:
  1. Inversion of Control
  2. Dependency Injection

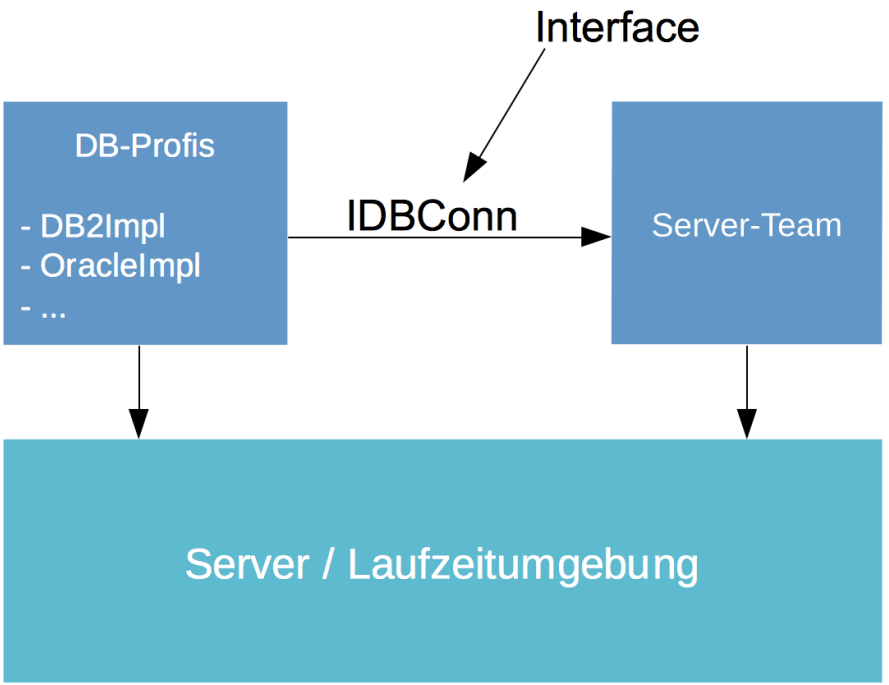


Inversion of Control (DIP)

- "Don't call us, we call you"
- Kontrolle der Ausführung liegt bei Framework, nicht bei Klassen die es nutzen
- Umsetzung durch Callback-Methoden

Dependency Injection (DIP)

- Problem: Wie kann zur Laufzeit konkrete Implementierung einer Schnittstelle erzeugt werden?
- Lösung: Dependency Injector
  Er entscheided und "injiziert" die benötigte Implementierung der Schnittstelle
- Umsetzung durch Callback-Methoden



Diskussion!

Wrapperklassen

Wrapperklassen

- Verpacken primitive Datentypen in Objekte
- Für Situationen in denen Objekte benötigt werden Beispielsweise Collections
- Wrapper im Package "java.lang"

Wrapper	Datentyp
Byte	byte
Short	short
Integer	int
Long	long
Float	float
Double	double
Boolean	boolean
Character	char
Void	void

Wrapperklassen

- Wrapper enthalten Attribut vom Datentyp
- Es existieren verschiedene Konstruktoren (Parameter: Datentyp, String...)
- Können prim. Datentyp via Methoden zurückliefern
- Es existieren static-Methoden zur Konvertierung

```
// Konstruktor mit prim. Typ
Short myShort = new Short((short) 5)
    ;

// Konstruktor mit String
Short myShort2 = new Short("5");

//Prim. Typ zurueckholen
short s = myShort.shortValue();

//Oder als String
String shortString = myShort.
    toString();

// Konvertierung von String in short
short ps = Short.parseShort("5");

// Konvertierung von String
// in Short-Objekt
Short vs = Short.valueOf("5");
```

Autoboxing bei Wrapperklassen

- Umwandlung zwischen prim. Datentyp und Wrapper nimmt Compiler automatisch vor
- Name dieser Funktion: Autoboxing

```
// Array mit Integer-Objekten
Integer[] meineInts = new Integer
      [5];

// Primitiver Typ ins Objekt-Array
meineInts[0] = 5;

// Objekt in prim. Typ gepackt
int erster = meineInts[0];
```

Innere Klassen

Innere Klassen

- Innere Klassen sind in Klassen definierte Klassen
- Inner-Class hat Zugriff auf alle Attribute der Outer-Class
- Outer-Class hat Zugriff auf alle Attribute der Inner-Class
- Outer-Class kann Objekt der Inneren instanziieren

```
// Aeussere Klasse
public class Outer {
private final int out = 5;
private final Inner innen = new
     Inner();

// Innere Klasse
public class Inner {
int out;
private int whatsOut() {
// Outer.this da
// das innere ''out''
// das aeussere ueberdeckt
// ansonsten: return out;
return Outer.this.out;
}
}

public static void main(String[]
     args) {
// Instanz von Outer-Class
Outer o = new Outer();

// Zugriff auf Private-Konstrukte
// Was passiert wenn folgende
// Zeile nicht in Outer steht?
// z.B. in einer anderen Klasse?
o.innen.whatsOut();
}
}
```

Zugriff auf Inner-Classes von außen

- Ist innere Klasse nicht private, kann auch außerhalb von Outer darauf zugegriffen werden
- Wichtig: Objekt von Outer-Class
- Inner-Objekt ist an Outer-Objekt gebunden

```
// Outer Objekt
Outer o = new Outer();

// Inner kann nun wie normales
// Objekt genutzt werden
Inner inner = o.new Inner();
```

Weitere Eigenschaften von inneren Klassen

- Innere Klassen können auch in Methoden definiert werden
  Dann hat innere Klasse Zugriff auf "final" Variablen der Äußeren
- Oftmals Einsatz in GUIs zur Ereignisbehandlung
- Zur mehrfachen Implementierung von Interfaces in einer Klasse

Anonyme Klassen

Anonyme Klassen

- Haben keinen Klassennamen
- Wird dort definiert wo "new" aufgerufen wird
- Z.B. zur Implementierung von Interfaces (Listener etc. . .)
- Kann wie normales Objekt benutzt werden

```
Melkbar m =
// In den Klammern koennen Parameter
// fuer Konstruktoren stehen
new Melkbar(){
public int gebeMilch(){
return 5;
}
};
```

Ausnahmen

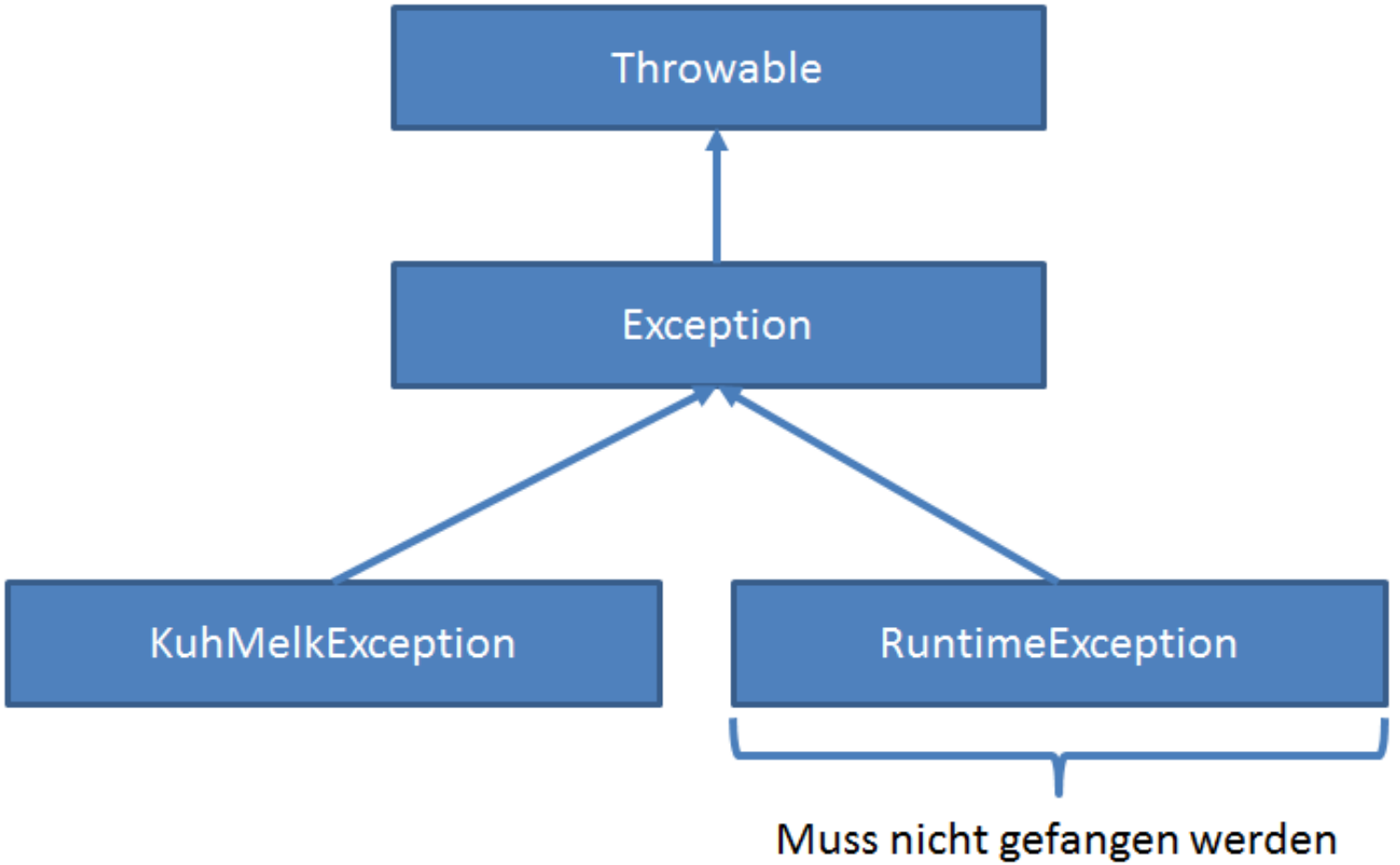
Exceptions

- Zur kontrollierten Fehlerbehandlung
- Damit nicht direkt das Programm abbricht
- Dazu werden Exceptions erzeugt und im Fehlerfall geworfen
- Exceptions = Ausnahme = Fehlerobjekt
- Exception ist vom Typ "throwable"

Exceptions

- try-catch kann Exceptions fangen
- Exception ist ein "Throwable"
- Es wird zwischen "Unchecked Exceptions" und "Checked Exceptions" Exceptions unterschieden
- "Unchecked Exceptions" erben von RuntimeException
- Durch erben von "Exception" können eigene Ausnahmen gebaut werden

Exception-Hierarchie



Exception-Hierarchie

- Methoden muss mit "throws" zeigen dass sie eine Exception werfen kann
- Mit "throw" wird Exception geworfen
- Bei RuntimeExceptions wird kein "throws" im Methodenkopf benötigt

```
public int gebeMilch()
// Methode kann Exception werfen
throws KuhMelkException{
if(milch == 0){
// Exceptoin erzeugen
throw new KuhMelkException("Milch
     leer");
}
return milch;
}
```

Exceptions behandeln

- 2 Möglichkeiten im Umgang mit Exceptions:
  1. Try-Catch
  2. Wiederrum ein Throws in aufrufende Methode
- In Catch kann auch geworfen werden
- Mit mehreren Catch-Blöcken können mehrere Exception-Typen gefangen werden
- Methode kann mehrere Exception-Typen werfen

```
// try-catch
try {
meinMelkbar.gebeMilch();
} catch (KuhMelkException k){
// Code zur Fehlerbehandlung
System.out.println(k.getMessage());
} finally {
// Wird immer ausgefuehrt
// z.B. Kuh Herausbringen
}
```

Swing

Swing

- Im Gegensatz zu AWT (Abstract Windowing Toolkit) sorgt Swing für einheitliches Look-and-Feel
- Swing verwendet JFrames zur Fensterdarstellung
- JFrame hat "ContentPane", ein Container für Komponenten

Ein erstes Fenster

- Für ein neues Fenster leiten wir von JFrame ab
- Konstruktor unserer Klasse:

1. super("Erstes Fenster");
2. setSize(100,100);
3. setVisible(true);
4. ...

```
// MeinFenster erbt von JFrame
public class MeinFenster extends
      JFrame {
MeinFenster() {
// Konstruktor von JFrame
// Fuer den Titel
super("Erstes Fenster");

// Groesse des Fenster
setSize(100, 100);

// Fenster sichtbar machen
setVisible(true);

// Bestimmen was beim Schliessen
// des Fensters passiert
setDefaultCloseOperation(
      EXIT_ON_CLOSE);
}

public static void main(String[]
      args){
MeinFenster fenster = new
      MeinFenster();
}
}
```

Grafik

Zeichnen mit Graphics

- Zum zeichnen überschreiben wir die Paint-Methode aus JFrame
- Paint-Methode:
  1. Kümmert sich um Ausgabe
  2. Automatischer Aufruf bei Neuzeichnung des Inhalts
  3. Wird nicht vom Entwickler aufgerufen, dafür ist Repaint
  4. Parameter: Graphics-Objekt

```
public class MeinFenster extends
      JFrame {
MeinFenster() {
super("Erstes Fenster");
setSize(100, 100);
setVisible(true);
setDefaultCloseOperation(
      EXIT_ON_CLOSE);
}

// Die Paint-Methode
// Aufruf vom Entwickler nur!
// durch Repaint
public void paint(Graphics g){
// Mit Graphics-Objekt kann man
      zeichnen
g.fillRect(20,20,50,50);
}

public static void main(String[]
      args){
MeinFenster fenster = new
      MeinFenster();
}
}
```

Zeichnen mit Graphics

- Graphics dient der grafischen Ausgabe
- Verwaltet Farbe und Schriftart
- Farbdarstellung mit der Klasse "Color"
- Schriftart wird durch die Klasse "Font" definiert (Schriftart für drawString(...))

```
public class MeinFenster extends JFrame {
MeinFenster() {
super("Erstes Fenster");
...
}

// Die Paint-Methode
// Niemals selbst aufrufen NUR
// durch Repaint
public void paint(Graphics g){
// Mit Graphics-Objekt kann man zeichnen
g.fillRect(20,20,50,50);

// Farbe setzen
g.setColor(Color.GREEN)

// Wieder zeichnen
g.drawLine(5,5,100,100);

// Font erstellen und setzen
Font f = new Font("Serif", Font.BOLD, 12);
g.setFont(f);

// Schreiben
g.drawString("Hallo", 20,20);
}
...
}
```

Fonts

- Konstruktor:
  new Font(String name, int style, int size)
- 5 Schriftarten = Name
- 3 Stile = style

Schriftarten
SansSerif
Serif
Monospaced
Dialg
DialogInput

Style	Bedeutung
Font.PLAIN	Normal
Font.BOLD	Fett
Font.ITALIC	Kursiv

Event-Handling

Events

- Bei Programmierung (insbesondere GUI) existiert eine Fülle wichtiger Ereignisse
- Ereignisse haben eine Quelle
- Um Ereignis mitzubekommen kann Objekt sich bei Quelle registrieren
- Quelle informiert dann bei Änderung
- Welches Entwurfsprinzip entspricht diesem Muster?

Listener

- Bei Quelle registrierte Objekte heißen Listener
- Dazu implementiert Empfänger eine Schnittstelle ein "Listener-Interface"

Listener	Registrierung	Quelle
ActionListener	addActionListener	Button. . .
MouseListener	addMouseListener	Component
MouseMotionListener	addMouseMotionListener	Component
. . .	. . .	. . .

Ereignisobjekte

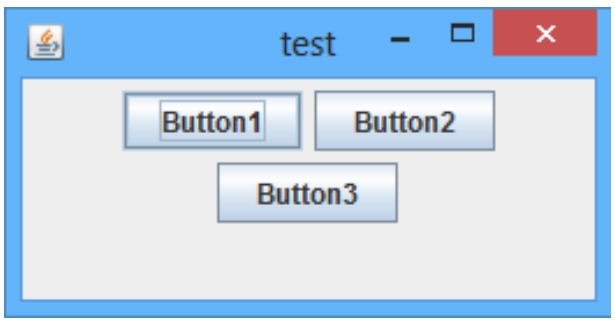
- Ereignis wird durch Objekt repräsentiert
- Ereignisobjekte beziehen sich auf die Art des Ereignisses (ActionEvent, MouseEvent ...)
- Bei Event wird Event-Objekt erzeugt und an Listener übergeben

```
public class Fenster extends JFrame {
Fenster() {
super("test");
setDefaultCloseOperation(EXIT_ON_CLOSE);
setSize(400, 400);
setLocation(100, 100);
addMouseListener(new MeinMouseListener());
}

// Listener als Inner-Class
private class MeinMouseListener implements
      MouseListener {
...
@Override
public void mousePressed(MouseEvent e) {
Graphics g = Fenster.this.getGraphics();
// Event hat Attribute wie x und y
g.fillRect(e.getX(), e.getY(), 2, 2);
}
...
}

public static void main(String[] args) {
Fenster f = new Fenster();
f.setVisible(true);
}
}
```

GUI

- Anordung von Interaktionselementen in einem Fenster
  1. Buttons
  2. Radio-Buttons
  3. Check-Boxen
  4. Textfields
  5. Label
- Diese Elemente sind Komponenten
- Sie erben alle von der Klasse Component

- Anordnung der Komponenten über Layoutmanager
  1. Border-Layout-Manager
  2. Flow-Layout-Manager
  3. Grid-Layout-Manager
- Lassen sich mit JPanels auch verschachteln




```
// 1. Borderlayout fuer das Fenster
BorderLayout border = new
      BorderLayout();
this.getContentPane().setLayout(
      border);

// Panel
JPanel p = new JPanel();

// 2. Layout fuer das JPanel
p.setLayout(new FlowLayout());

// Button fuer das Panel
JButton button = new JButton("
      Button1");
JButton button2 = new JButton("
      Button2");
JButton button3 = new JButton("
      Button3");
p.add(button);
p.add(button2);
p.add(button3);

// Panel der Contentpane hinzufuegen
this.getContentPane().add(p,
      BorderLayout.CENTER);
```