

hochschule mannheim  
**Grundidee der Objektorientierung**

- Eine Grundidee der objektorientierten Programmierung (OOP) ist es, Daten und Methoden, die auf diese Daten angewandt werden können, möglichst eng in einem sogenannten Objekt zusammenzufassen und nach außen hin zu kapseln, so dass Methoden fremder Objekte diese Daten nicht versehentlich manipulieren können.
- Ein Programm ist realisiert als eine Menge interagierender Objekte.
- Im Gegensatz zu älteren Ansätzen werden durch die Objektorientierung die menschlichen Organisationsmethoden zum Verstehen der realen Welt besser unterstützt.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 3

hochschule mannheim  
**OOP Begriffe**

- Objekt, Exemplar, Instanz
  - Ein zur Laufzeit des Programms existierende Repräsentation eines Modells, zumeist aus der realen Welt.
  - Beispiel: Student „Uli Müller“, Rechnung „R342“
  - Beim Starten eines Java-Programms werden in der **main**-Methode die ersten Objekte erzeugt, die dann kommunizieren können.
- Klasse, Bauplan, Typ
  - Eine Beschreibung der Datenstruktur & der Fähigkeiten von Objekten.
  - Beispiel: „Student“, „Rechnung“, „Bruch“, „Bestellung“
  - In Java programmieren Sie Klassen, die dann zur Laufzeit konkrete Objekte erzeugen, welche dann interagieren.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 4

hochschule mannheim  
**OOP Begriffe**

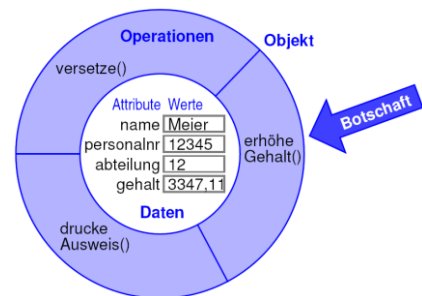
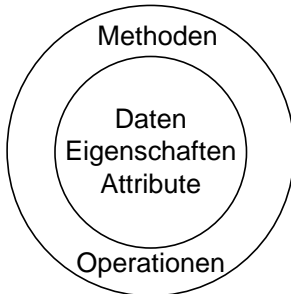
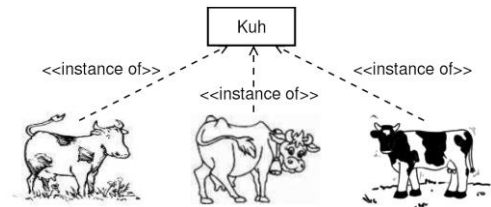
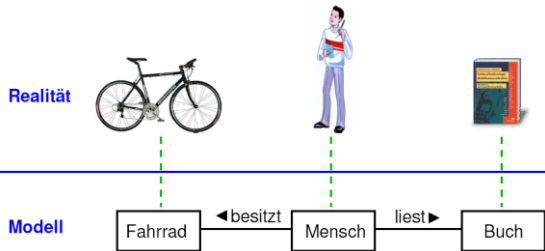
- Eigenschaft, Attribut, Variable, Datenwert
  - Die Daten, aus denen jedes Objekt einer Klasse besteht
  - Was „hat“ jedes Objekt dieser Klasse?
  - Beispiele:
    - Jeder Student hat einen Namen, einen Vornamen, eine Matrikel-Nummer,...
    - Jede Rechnung hat Rechnungspositionen, ein Datum, einen Kunden,...
    - Jede Rechnungsposition hat eine Nummer, einen Artikel, eine Menge, einen Einzelpreis,...

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 5

hochschule mannheim  
**OOP Begriffe**

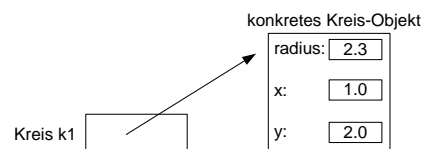
- Methode, Operation, Funktion, Prozedur, Dienst
  - Die Dienste, die jedes Objekt einer Klasse anbietet.
  - Was „kann“ jedes Objekt dieser Klasse?
  - Beispiel:
    - Jeder Student kann angelegt werden, seinen Namen und seine Matrikel-Nummer nennen, ...

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 6



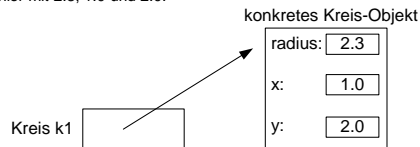
- In einer Variablen (hier: **k1**) wird eine Referenz auf einen Speicherbereich abgelegt, in dem sich das Objekt befindet.
- Der spezielle Wert null verweist nirgendwohin.
- Der Speicherbereich für die Daten wird dynamisch durch den Operator **new** angelegt.

## Wie erzeuge ich ein Objekt?

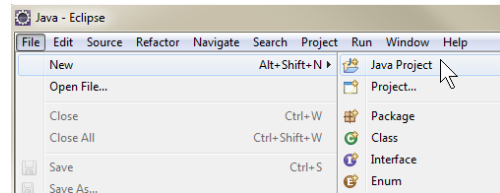


## hochschule mannheim Anlegen von Objekten: Begriffe

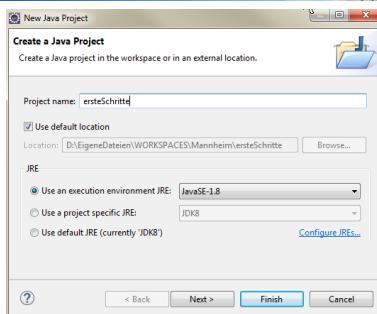
- **k1** ist eine Referenz auf ein konkretes Objekt der Klasse Kreis.
- Der Bauplan (also die Klasse) sieht vor, dass jeder Kreis einen Radius, ein x und ein y besitzt.
- Der Radius, X und Y sind Eigenschaften jedes Kreises
- Jedes konkrete Objekt der Klasse Kreis hat diese Eigenschaften befüllt, hier mit 2.3, 1.0 und 2.0:



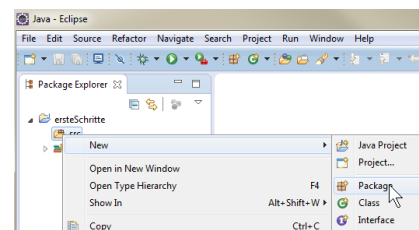
## hochschule mannheim Das erste Java Projekt



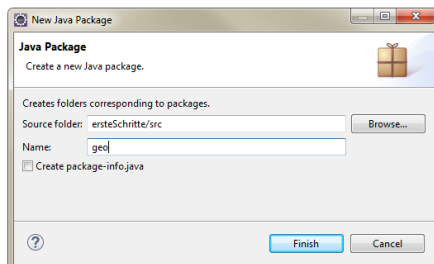
## hochschule mannheim Das erste Java Projekt



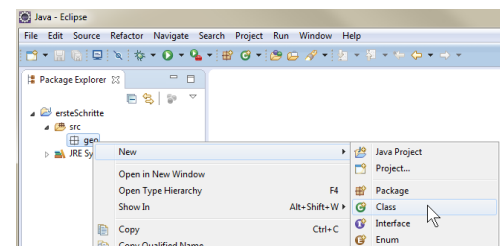
## hochschule mannheim Das erste Package

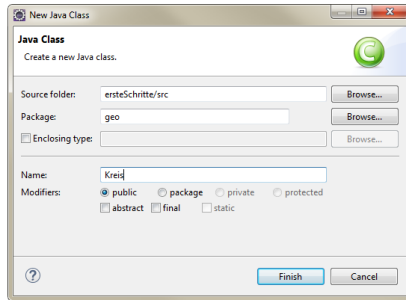


## hochschule mannheim Das erste Package

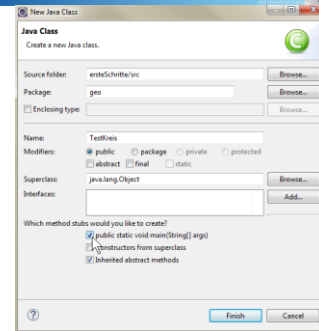
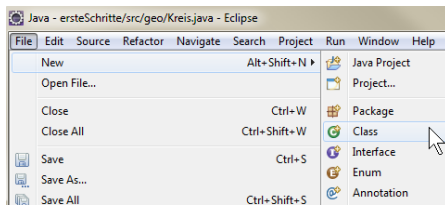


## hochschule mannheim Die erste Klasse





```
1 package geo;
2
3 public class Kreis {
4
5 }
```



```
package geo;

public class TestKreis {
    public static void main(String[] args) {
        // 3 Anweisungen zur Erstellung von
        // 3 Kreis-Objekten über ihren
        // Standard-Konstruktor:
        Kreis k1=new Kreis();
        Kreis k2=new Kreis();
        Kreis k3=new Kreis();
        // Anweisung zur Ausgabe eines Kreis-Objektes:
        System.out.println(k2);
    }
}
```

Problems @ Javadoc Declarati  
<terminated> TestKreis [Java Application]  
geo.Kreis@139a55

Wie erhält ein Objekt  
seine Eigenschaften?

In Java werden **3 Arten von Variablen** unterscheiden:

- Lokale Variablen in Methoden oder generell in Blöcken dienen der temporären Speicherung von Daten, während diese Methode ausgeführt wird.
- Formale Parameter in Methoden speichern die Werte der aktuellen Parameter, die beim Aufruf einer Methode übergeben werden.
- Attribute/Eigenschaften in Objekten speichern die Eigenschaften von Objekten.

```
package geo;

public class Kreis {
    private double radius;
    private double x;
    private double y;
}
```

## Wie kann man auf die Eigenschaften von außen zugreifen?

- Auf die Eigenschaften eines Objektes darf man nicht direkt von außen zugreifen, sonst könnte man u.a. den Radius auf -1 setzen!
- Kein mathematisch gültiger Kreis mehr!
- Klasseninvariante!
- Jedes Objekt muss stets Kontrolle über die Belegung seiner eigenen Eigenschaften besitzen, also über seinen inneren Zustand!
- Dafür ist der Programmierer der Klasse verantwortlich!
- Eigenschaften sind `private` zu deklarieren!
- Die Methoden, mit denen man auf die Eigenschaften lesend bzw. schreibend zugreift, nennt man Getter bzw. Setter.
- Diese Methoden können dann zur Formatierung und Prüfung der Eigenschaften verwendet werden.

- Die Sichtbarkeit von Eigenschaften & Methoden kann durch die Angabe von **public**, **private** und **protected** beeinflusst werden:

Sichtbarkeit	Innerhalb des Package	Abgeleitete Klassen	Außerhalb des Package
<code>private</code>	unsichtbar	unsichtbar	unsichtbar
<code>default</code>	sichtbar	unsichtbar	unsichtbar
<code>protected</code>	sichtbar	sichtbar	unsichtbar
<code>public</code>	sichtbar	sichtbar	sichtbar

- Java-Klassen können entweder mit dem Attribut **public** versehen werden, dann sind sie von überall erreichbar, oder attributfrei sein.
- In diesem Fall haben sie default-Sichtbarkeit und können nur von Klassen aus demselben Package erreicht werden.

```
package geo;

public class Kreis {
    private double radius;
    private double x;
    private double y;

    public void setRadius(double radius){
        // Wann ist das nur erlaubt?
        this.radius=radius;
    }
    public double getRadius(){
        return this.radius;
    }
    public void setX(double x){
        this.x=x;
    }
}
```

hochschule mannheim  
Getter & Setter generieren  
in Eclipse

Java - ersteSchritte/src/geo/Kreis.java - Eclipse

File Edit Source Refactor Navigate Search Project

Package Explorer: Package: geo, Class: Kreis

Generate Getters and Setters dialog:

- Select getters and setters to create:
  - ☒ radius
  - ☐ x
  - ☐ y
  - ☐ getX()
  - ☐ setY(double)
- Allow setters for final fields (remove 'final' modifier): ☐
- Insertion point: after y
- Sort by: Fields in getter/setter pairs
- Access modifier:
  - ☒ public
  - ☐ protected
  - ☐ package
  - ☐ final
  - ☐ synchronized
- Generate method comments: ☒

The format of the getters/setters may be configured on the [Code Templates](#) preference page.

Buttons: OK, Cancel

hochschule mannheim  
Die Erzeugung der ersten Kreis-Objekte  
und Zugriff auf die Eigenschaften

```
package geo;

public class TestKreis {
    public static void main(String[] args) {
        Kreis k1=new Kreis();
        Kreis k2=new Kreis();
        Kreis k3=new Kreis();
        k2.setRadius(2.3);
        k2.setX(1.0);
        k2.setY(2.0);
        System.out.println(k2.getRadius());
        System.out.println(k2.getX());
        System.out.println(k2.getY());
    }
}
```

hochschule mannheim  
Was ist this?

- Wenn der Programmierer in einem Kreis auf eine Eigenschaft dieses Kreises zugreifen will, sollte er **this** benutzen.
- this** ist eine Referenz auf „sich selbst“
- this.x=x** setzt „meine x-Koordinate“ (ich bin ein Kreis) auf den x-Wert, der von aussen übergeben wurde.

hochschule mannheim  
Klassen-Attribut: static

- Jeder Kreis besitzt einen Mittelpunkt (x/y) und einen Radius.
- Die Anzahl der erzeugten Kreise kennt aber ein Kreis-Objekt nicht!
- Die Klasse erzeugt Kreise!
  - Man muss die Klasse fragen, wie viele Kreise schon erzeugt wurden!
  - Diese Anzahl existiert einmalig für die ganze Klasse!
- Eigenschaften, die einmalig pro Klasse existieren, werden mit **static** gekennzeichnet: **private static int anzahl=0;**
- Die Zahl PI muss auch nicht in jedem Kreis gespeichert werden, denn sie ist für alle Kreise gleich.
- PI ändert sich auch nicht und ist damit eine Konstante, auf die jeder zugreifen kann:
 

```
public static final PI=3.141592654;
```

hochschule mannheim  
Zugriff auf die Eigenschaften des Kreises:  
Get und Set für eine statische Eigenschaft

```
public class Kreis {
    public static final double PI=3.141592654;
    private static int anzahl=0;

    public static int getAnzahl(){
        return Kreis.anzahl;
    }

    private static void incAnzahl(){ // wieso private?
        Kreis.anzahl++;
    }
    // wann muss incAnzahl aufgerufen werden?
}
```

hochschule mannheim  
Klassen-Methode:  
incAnzahl ist static

- Man kann auch nach der Anzahl fragen, wenn noch gar kein Kreis erzeugt wurde:
- Man muss die Klasse selbst fragen können.
- Daher wird die entsprechende Methode (hier: **incAnzahl**) ebenfalls **static** deklariert wie seine abgefragte Eigenschaft.
- Die **main**-Methode beim Programmstart ist ebenfalls **static**, da zum Beginn des Programms ja noch keine Objekte existieren.

## Wie kann man Regeln zur Erzeugung von Objekten definieren?

```
public class TestKreis {
    public static void main(String[] args) {
        Kreis k1=new Kreis();
        Kreis k2=new Kreis();
        Kreis k3=new Kreis();
        // Sind das gültige Kreise?
    }
}
```

- Eine Klasse kann spezielle Methoden definieren, die bei der Erzeugung eines Objekts ausgeführt werden: Konstruktoren
- Aufgaben eines Konstruktors sind
  - die Initialisierung der Attributwerte des neuen Objekts und
  - ggf. die Erzeugung existenzabhängiger Teil-Objekte.
- Ein Konstruktor hat immer denselben Namen wie die Klasse.
- Ein Konstruktor kann Parameter besitzen, hat aber keinen Ergebnistyp:
  - Er gibt in Kombination mit dem Schlüsselwort **new** automatisch eine Referenz auf das gerade erstellte, neue Objekt zurück.
- Definiert eine Klasse keinen Konstruktor, so wird automatisch ein parameterloser Standard-Konstruktor erzeugt.
- Dessen Eigenschaften werden dann mit Standardwerten (0 bzw. null) initialisiert.

- Innerhalb eines Konstruktors kann mit **this(<Parameter>)** ein anderer Konstruktor dieser Klasse aufgerufen werden.
- Dies vermeidet Code-Dopplung und schlechte Wartbarkeit.
- **this(...)** muss immer erste Anweisung im Konstruktor sein.
- Welcher andere Konstruktor aufgerufen wird, entscheiden die Anzahl und die Typen von **<Parameter>**.

```
public Kreis(){
    Kreis.incAnzahl(); this.setRadius(1);
}
public Kreis(double radius){
    this(); this.setRadius(radius);
}
public Kreis(double x,double y){
    this(); this.setX(x); this.setY(y);
}
public Kreis(double radius,double x,double y){
    this(x,y); this.setRadius(radius);
}

public class TestKreis {
    public static void main(String[] args) {
        Kreis k=new Kreis(2.3,1.0,2.0);
    }
}
```

Diagramm zur Aufruf-Hierarchie von Konstruktoren:

- 1: Aufruf von `Kreis(2.3,1.0,2.0)` in `main`
- 2: Aufruf von `this(x,y)` in `Kreis(2.3,1.0,2.0)`
- 3: Aufruf von `this()` in `Kreis(double radius)`
- 4: Aufruf von `this.setRadius(1)` in `Kreis()`
- 5: Aufruf von `this.setRadius(radius)` in `Kreis(double radius)`
- 6: Aufruf von `this.setX(x); this.setY(y);` in `Kreis(double x, double y)`
- 7: Aufruf von `Kreis.incAnzahl();` in `Kreis()`

## Wie erhält ein Objekt seine Funktionalität?

- Genauso wie die Getter und Setter, dies sind ja spezielle Methoden.
- Jede Methode kann bei ihrer Definition keinen, einen oder mehrere Input-Parameter bekommen, siehe `setRadius(double radius)` ;
- Jede Methode - außer ein Konstruktor - kann bei ihrer Definition einen Rückgabewert als Output-Parameter bekommen, siehe `double getRadius()` ;
- Hat eine Methode einen Rückgabewert, so muss eine Rückgabe durch Verwendung des Schlüsselwortes `return` erfolgen.
- Hat eine Methode keinen Rückgabewert, so wird sie void deklariert, also `void setRadius(double radius)` ;

<<Name der Klasse>>
<<Sichtbarkeit und Name Eigenschaft 1>>: <<Datentyp>> <<Sichtbarkeit und Name Eigenschaft 2>>: <<Datentyp>> <<Sichtbarkeit und Name Eigenschaft 3>>: <<Datentyp>>
<<Sichtbarkeit und Name Methode 1>>(<<Input-Parameterliste>>): <<Rückgabotyp>> <<Sichtbarkeit und Name Methode 2>>(<<Input-Parameterliste>>): <<Rückgabotyp>> <<Sichtbarkeit und Name Methode 3>>(<<Input-Parameterliste>>): <<Rückgabotyp>>

+ public  
- private  
# protected  
~ package

unterstrichen: static

{readOnly}: kein oder nur ein privater Setter vorhanden

Kreis
+ x: double + y: double + r: double <u>+ anzahl:int {readOnly}</u>
+zeichnen() +verschieben(dx:int, dy:int) +skalieren(faktor:double) +berechneFläche():double

```
public class Kreis {
    // ...
    public void zeichnen(){
        // Code der Methode
    }
    public void verschieben(int dx,int dy){
        // Code der Methode
    }
    public void skalieren(double faktor){
        // Code der Methode
    }
    public double berechneFläche(){
        // Code der Methode
    }
}
```

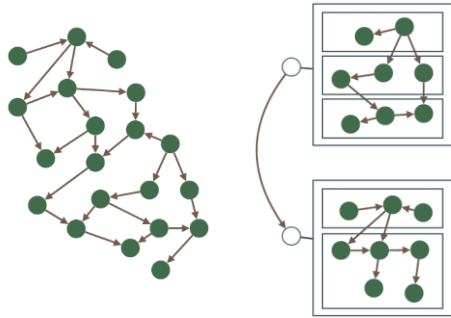
Die Eigenschaften sind + deklariert, da sie über Getter / Setter erreichbar sein sollen!

public class <Name der Klasse, 1. Buchstabe groß>{
statische Konstanten
statische Attribute
Attribute jedes Objektes
Konstruktoren, beginnend bei Default-Konstr.
statische Methoden der Klasse
Getter und Setter
@Overrides
Dienste der Klasse (öffentliche Methoden)
Hilfsmethoden (privat)
}

## Strukturierung größerer Anwendungen

- Softwaresysteme gehören zu den komplexesten Gebilden, die Menschen erzeugen
- Boeing 747  
6 Millionen Teile, davon 50% Nieten
- Windows 2003 Server  
50 Millionen Zeilen Code
- SAP Business Suite  
250 Millionen Zeilen Code
- Menschliches Genom  
3 Milliarden Basen-Paare
- Wir müssen trotzdem die Kontrolle über die Software behalten!





- Die Software wird in eigenständige Teile/Module zerlegt.
- Die Schnittstelle zwischen den Modulen wird spezifiziert.
- Die Verwender dürfen nur über die Schnittstelle zugreifen.
- Das Innenleben, die Implementierung, des Moduls geht den Verwender nichts an und wird vor ihm versteckt.
- Dieser Ansatz führt zu
  - Kapselung und
  - Information Hiding.

```
public class Datum {
    int tag;
    int monat;
    int jahr;
}

public class Verwender {
    public void m() {
        Datum d = new Datum();
        d.tag = 32;
        d.tag = 30;
        d.monat = 2;
        d.tag = 31;
        d.tag++;
    }
}
```

```
public class Datum {
    private int tag;
    private int monat;
    private int jahr;

    public void setTag(int tag) {
        if (tag > 31) {
            // kann irgendwie nicht sein
        }
        this.tag = tag; // nur hier zuweisen!
    }
}
```

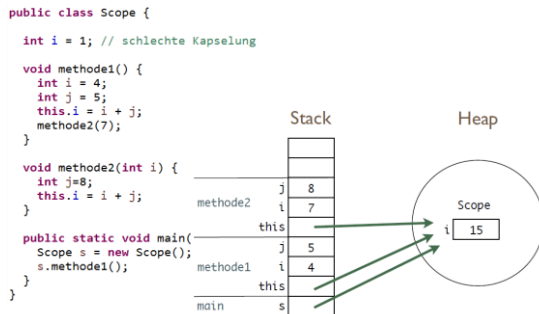
- Anweisungen (statements)
- Blöcke (blocks)
- Methoden (methods)
- Klassen (classes)
- Pakete (packages)
- Java Archive (JAR)

- Klassen sind Baupläne für Objekte, sie beschreiben
  - die Daten, die ein Objekt tragen kann: Eigenschaften, Attribute, Variablen
  - das Verhalten, das ein Objekt zeigen kann: Methoden, Funktionen
- Java-Klassen unterstützen objektorientierte Techniken
  - Abstraktion
  - Kapselung
  - Vererbung
  - Polymorphismus

- Lebensdauer von...
  - Klassenvariablen  
Während der gesamten Laufzeit des Programms.
  - Instanzvariablen  
Während der Lebensdauer des Objekts.
  - Lokale Variablen  
Von der Deklaration bis zum Ende des Blocks oder der Methode, in der sie deklariert wurden.
- Sichtbarkeit von...
  - Klassenvariablen und Instanzvariablen  
In der gesamten Klasse
  - Lokalen Variablen  
Von der Deklaration bis zum Ende des Blocks oder der Methode, in der sie deklariert wurden.

```
public class Scope {
    int i = 1; // schlechte Kapselung
    void methode1() {
        int i = 4;
        int j = 5;
        this.i = i + j;
        methode2(7);
    }
    void methode2(int i) {
        int j=8;
        this.i = i + j;
    }

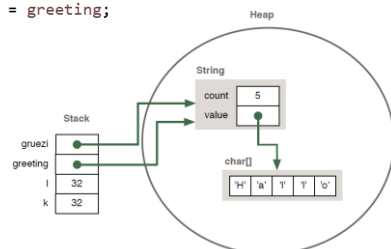
    public static void main(String[] args) {
        Scope s = new Scope();
        s.methode1();
    }
}
```



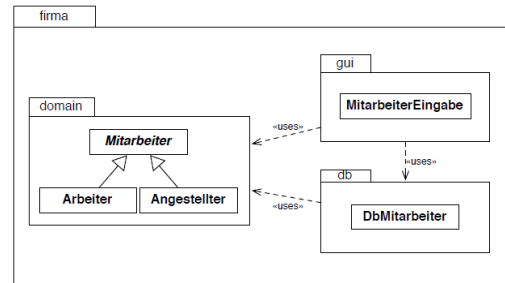
- Der Stack
  - wird auch Kellerspeicher genannt.
  - nimmt nur lokale Variablen auf.
- Der Heap
  - wird vom Garbage Collector aufgeräumt;
  - nimmt nur Objekte auf.

- Ein Objekt
  - liegt immer auf dem Heap.
  - trägt Daten: primitive Datentypen oder Referenzen.
- Eine Referenz
  - ist kein Objekt.
  - zeigt auf ein Objekt.
  - kann auf dem Heap liegen als Variable in einem Objekt.
  - kann auf dem Stack liegen als lokale Variable.
  - kann zu verschiedenen Zeiten auf verschiedene Objekte zeigen.

```
int k = 32;
int l = k;
String greeting = new String("Hallo");
String gruezi = greeting;
```



- Große Software-Projekte führen zu einer hohen Zahl von Klassen und Schnittstellen.
- Pakete verhindern Namenskonflikte, wobei gleiche Namen in unterschiedlichen Paketen möglich werden.
- Pakete erlauben die Strukturierung von UML-Diagrammen und Java-Code und damit eine weitere Strukturierung der Anwendung.
  - Logisch zusammengehörige Klassen und Schnittstellen werden in einem Paket gruppiert.
- Pakete können weitere Pakete enthalten; dies fördert
  - eine hierarchische Strukturierung der Software.
  - den Divide-and-Conquer Ansatz.
- Pakete erlauben verfeinerte Spezifikation von Sichtbarkeiten durch die **package**-Sichtbarkeit.



- Die Klasse ordnet sich selbst durch das **package** Statement einem Paket zu.
- Eine Klasse kann nur zu genau einem Paket gehören.
- **package** muss als erstes Statement in der Klasse stehen.
- In einem Paket kann es jeden Klassennamen nur einmal geben.
- Fehlt das **package**, so gehört die Klasse zum namenlosen default-Paket.
- Pakete werden daher über hierarchische Namen angesprochen:
  - in Java: **Paket.Unterpaket1.Unterpaket2.Klasse**
  - im Dateisystem: **Paket\Unterpaket1\Unterpaket2\Klasse.java**

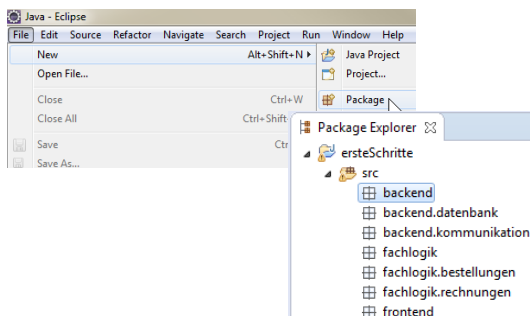
```
package firma.domain;

public class Mitarbeiter {
}
```

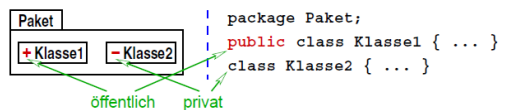
- Um Klassen bzw. Schnittstellen in Java zu benutzen, gibt es drei verschiedene Möglichkeiten:
  - Angabe des vollen Namens, z.B. **java.util.Date datum;**
  - Importieren der Klasse am Anfang der Programmdatei:
 

```
import java.util.Date;
...
Date datum;
```
  - Importieren aller Klassen eines Pakets am Anfang der Programmdatei:
 

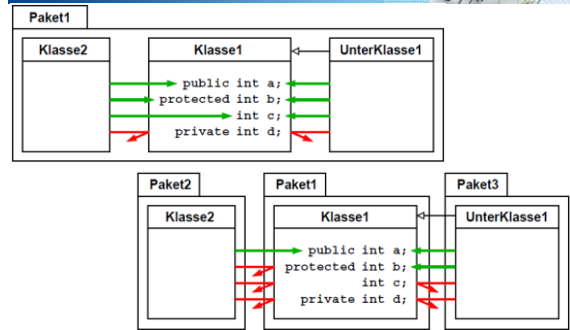
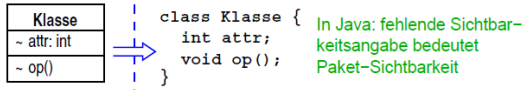
```
import java.util.*;
```
- Bei Klassen aus dem eigenen Paket reicht immer der einfache Klassenname aus.
- Die **import** Statements müssen nach dem **package** Statement, aber vor der Klassendeklaration im Quelltext stehen.



- Für die in einem Paket enthaltenen Klassen können Sichtbarkeiten definiert werden:
  - **public**: Die Klasse ist für alle Pakete sichtbar.
  - **„private“**: Die Klasse ist nur innerhalb ihres Pakets sichtbar
- Darstellung im UML-Klassendiagramm:



- Für Eigenschaften & Methoden von Klassen und Schnittstellen kann eine weitere Sichtbarkeit definiert werden:
- package: Sichtbar nur in allen Klassen desselben Pakets.
- Darstellung im UML-Klassendiagramm:



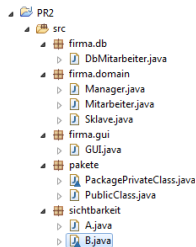
- Klassen können zwei mögliche Sichtbarkeiten haben:
- default: Nur Klassen innerhalb desselben Pakets dürfen zugreifen
- public: Jeder darf zugreifen
- Nur wenn die Klasse selbst öffentlich ist, sind die öffentlichen Methoden oder Variablen außen sichtbar.

package sichtbarkeit;	package sichtbarkeit;
public class A {	class B {
private int a;	private int a;
int b;	int b;
public int c;	public int c;
private void m1() {}	private void m1() {}
void m2() {}	void m2() {}
public void m3() {}	public void m3() {}
}	}

- Jede öffentliche Klasse `public class <Name>` muss in einer eigenen Datei mit dem Namen `<Name>.java` abgelegt werden.
- Mehrere paketprivate Klassen können jedoch zusammen in einer Datei abgelegt werden.

<p>package\PublicClass.java</p> <p>package pakete;</p> <p>public class PublicClass {</p> <p>}</p>	<p>package\PackagePrivateClass.java</p> <p>package pakete;</p> <p>class PackagePrivateClass {</p> <p>}</p> <p>class NochEineKlasse {</p> <p>}</p> <p>class UndNochEine {</p> <p>}</p>
---	---

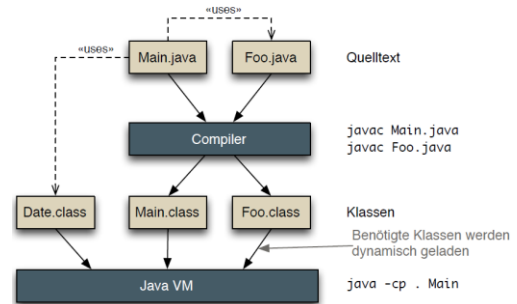
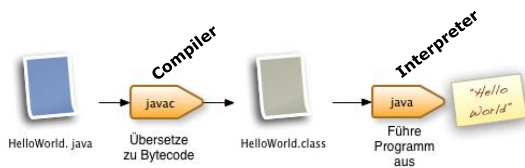
- Die kompilierten Java-Klassen (.class-Dateien) werden entsprechend der Paketzugehörigkeit in Verzeichnissen abgelegt.
- Die Java-Quelldateien (.java-Dateien) sollten ebenfalls nach Paketen organisiert werden.
- Eclipse übernimmt diese Aufgabe im package- und im project-Explorer automatisch:



- Neben der reinen Strukturierung werden Pakete für das Information Hiding eingesetzt.
- Klassen mit Sichtbarkeit auf Paket-Ebene können von außen nicht verwendet werden, und daher kann man sie vor externen Verwendern geheim halten.

- Die Java VM bringt bereits eine ganze Reihe von Paketen und Klassen mit. Hierbei gilt:
  - **java.lang** wird immer automatisch importiert, alle anderen Pakete muss man explizit importieren.
  - **java.\*** kann man frei benutzen, da Oracle die Pakete zukünftig kompatibel hält.
  - Deprecation dient zum Abkündigen von Funktionen aus den **java.\*** Paketen.
- Auf keinen Fall sollte man Klassen aus **sun.\*** und **com.sun.\*** verwenden, da Sun Microsystems von Oracle übernommen wurde und nicht mehr separat gepflegt wird.

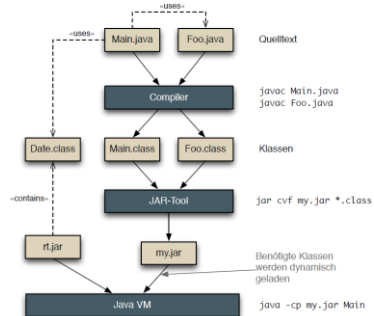
- Jede Java-Klasse ist selbstbeschreibend.
- Der Verwender kann alle Meta-Informationen aus der .class-Datei beziehen.
- Der Befehl **javap** dient dazu, diese Informationen auszugeben.
- Java kennt nur einen Compiler, die Klassen
  - werden von der Java VM dynamisch bei Bedarf geladen.
  - werden erst geladen, wenn sie das erste Mal benötigt werden.
- Java VM sucht die Klassen auf dem Klassenpfad, der über die VM-Option **-cp** oder **-classpath** modifizierbar ist.



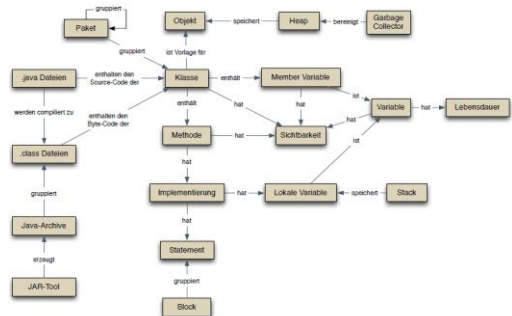
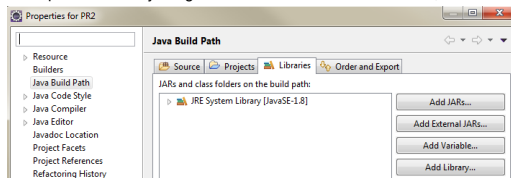
- Ein Java Archiv (JAR) ist eine Sammlung von Java-Klassen und Hilfsdateien, die in einer ZIP-Datei gepackt wurden.
- Vorteile:
  - komprimiert, weniger Speicherplatzverbrauch
  - eine einzelne Datei, mehr Ordnung
  - zusammengehörige Klassen können gruppiert werden
  - kann direkt ausgeführt werden: **java -jar foo.jar**
  - JARs können genauso auf den Klassenpfad gelegt werden, wie Verzeichnisse mit Klassen

- Sicherheit
  - Man kann den Inhalt digital signieren und so vor Veränderungen schützen.
- Sealing
  - Man kann festlegen, dass alle Klassen eines Paketes in einer JAR enthalten sein müssen.
- Versionierung
  - In den Metadaten der JAR-Datei kann man Versionsinformationen für die enthaltenen Klassen ablegen.

- JAR-Dateien werden mit dem Programm `jar` erzeugt:  
`jar -cvf archivname.jar dateien`
- Die Syntax ist ähnlich zum Unix-Kommando `tar`.
- Zusätzlich zu den eingepackten Dateien enthält eine JAR Metadaten im Verzeichnis **META-INF**.



- Eigene und fremde JARs können in die eigene Anwendung eingebunden werden.
- Deren Klassen werden dann genauso verwendet wie eigene Klassen.
- Das Einbinden erfolgt auf der Kommandozeile bei Compiler und Java-VM über die Option `-classpath`.
- in Eclipse in den Projekteigenschaften unter „Java Build Path“, Libraries:



## Alles ist ein Object!

- Die Klasse `Object` ist die Wurzel aller Vererbungshierarchien.
- Jede Java-Klasse erbt direkt oder indirekt von `Object`.
- Wenn eine Klasse von keiner anderen erbt, erzeugt der Compiler automatisch ein `extends Object`, d. h. aus  
`public class Klasse {}` wird  
`public class Klasse extends Object {}`
- `Object` enthält einige Methoden, die es an alle anderen Klassen vererbt:
  - `toString()`
  - `equals()`
  - `hashCode()`
  - `clone()`

- Die **toString()**-Methode
  - liefert eine Darstellung des Objekts als String zurück.
  - wird automatisch bei String-Verknüpfungen aufgerufen.
  - soll primär beim Debuggen helfen.
  - dient nicht der Serialisierung von Objekten.
- Die Standardimplementierung gibt den Typ des Objektes gefolgt von @ und danach den Hash-Code aus.

- Der **==** Operator bestimmt, ob zwei Objekt-Referenzen identisch sind, also auf dasselbe Objekt zeigen.
- **equals()** stellt fest, ob zwei Objekte den gleichen Inhalt haben.
- Wenn man **equals()** nicht überschreibt, verwendet die Implementierung aus Object den **==** Operator.
- Wenn man **equals()** überschreibt, so muss man auch **hashCode()** überschreiben und umgekehrt.

- **hashCode()** dient dazu, einen möglichst eindeutigen Hash-Wert als Identifier für das Objekt zu erzeugen.
- Die Implementierung von Object verwendet einen einmaligen Schlüssel für den Hash-Wert, unabhängig von den Daten im Objekt.
- Man überschreibt **hashCode()**, um einen Hash-Wert aus den Daten des Objekts zu berechnen.
- Wenn man **equals()** überschreibt, so muss man auch **hashCode()** überschreiben und umgekehrt.
- Tipp:  
Auch in anderen Kontexten (z.B. beim Debugging oder beim Speichern in Datenbanken) ist es sinnvoll, einen eindeutigen Identifier für jedes Objekt zu definieren.  
Sie sollten daher eh einen solchen eindeutigen Identifier bei jeder Klassendefinition als Attribut anlegen!

- **clone()** dient dem Kopieren von Objekten.
- Es erzeugt eine sogenannte „flache Kopie“ des Objekts.
- Dazu muss die **protected clone()**-Methode überschrieben werden.
- Die Klasse muss **Cloneable** implementieren.
- **Cloneable** ist ein Interface, dazu später mehr...
- In der ersten Zeile muss **super.clone()** aufgerufen werden.
- Sogenannte „tiefe Kopien“ kann man durch rekursiven Aufruf von **clone()** auf den Instanzvariablen erzeugen; dies kann aber sehr speicherlastig werden!

- Flache bzw. seichte Klone beinhalten nur die primitiven Datentypen und Strings des Originals.
- ```

public class Student implements Cloneable{
    private String vorname;
    private String nachname;
    private int matrikelnummer;

    public Student(String vorname, String nachname, int matrikelnummer) {
        setVorname(vorname);
        setNachname(nachname);
        setMatrikelnummer(matrikelnummer);
    }

    @Override
    public Student clone(){
        Student s = new Student(
            this.getVorname(), this.getNachname(), this.getMatrikelnummer());
        // s.setAugenfarbe(this.getAugenfarbe());
        // usw.
        return s;
    }
    
```

- Beim tiefen klonen enthält der Klon nicht nur die primitiven Datentypen und Strings des Originals, sondern auch die Referenzen (und deren Datentypen und Referenzen und deren Datentypen...).
- ```

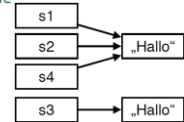
@Override
public Student clone(){
    Student s = new Student(
        this.getVorname(), this.getNachname(), this.getMatrikelnummer());
    // s.setAugenfarbe(this.getAugenfarbe());
    // usw.
    for(Veranstaltung v:meineVeranstaltungen){
        s.addVeranstaltung(v.clone());
    }
    return s;
}
    
```

```
public static void main(String[] args) {
    Student s = new Student("Uli", "Maier", 423634);
    Student t = s.clone();
    System.out.println(s.equals(t));
    System.out.println(s==t);
}
```

true  
false

- Die Java VM optimiert String-Literale und sorgt dafür, dass sie nur ein einziges Mal im Speicher liegen.
- Hierzu verwendet sie die `intern()`-Methode der Klasse `String`:

```
String s1 = "Hallo";
String s2 = "Hallo";
String s3 = new String("Hallo");
String s4 = s3.intern();
System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
System.out.println(s1 == s4); // true
```



- Bei Vergleichen von Strings mit String-Literalen kann man sich den Test auf null sparen, wenn man die Bedingung umdreht und statt
- ```
if ((s != null) && (s.equals("text"))) { ... }
```
- schreibt.
- ```
if ("text".equals(s)) { ... }
```

```
public class Mitarbeiter implements Cloneable{
    // alle Eigenschaften gehören private!
    private int personalnummer; // die eindeutige ID
    private String name;
    private static int anzahl=0;

    // hier muss der Default-Konstruktor private sein: warum?
    private Mitarbeiter(){
        incAnzahl();
    }

    public Mitarbeiter(int personalnummer,String name){
        this();
        setPersonalnummer(personalnummer);
        setName(name);
    }
}
```

```
private static void incAnzahl(){ // warum static?
    Mitarbeiter.anzahl++;
}
public static int getAnzahl(){
    return anzahl;
}
public int getPersonalnummer(){
    return personalnummer;
}
public void setPersonalnummer(int personalnummer) {
    if (personalnummer<0) // ändern von Eigenschaften NUR in den settern!
        throw new RuntimeException("Personalnummer ist ungültig!");
    this.personalnummer = personalnummer;
}
public String getName() {
    return name;
}
public void setName(String name) {
    if ((name==null)||((name.length())<2))
        throw new RuntimeException("Name ist ungültig!");
    this.name = name;
}
}
```

```
@Override
public String toString(){
    return "Mitarbeiter "+getName()+" mit Pers.Nr. "+getPersonalnummer();
}
@Override
public int hashCode(){
    return personalnummer;
}
@Override
public boolean equals(Object o){
    if (o == null) return false;
    if (o == this) return true;
    if (o.getClass() != Mitarbeiter.class) return false;
    Mitarbeiter m=(Mitarbeiter)o;
    return (this.getPersonalnummer()==m.getPersonalnummer());
}
@Override
public Mitarbeiter clone(){
    return new Mitarbeiter(getPersonalnummer(),getName());
}
}
```



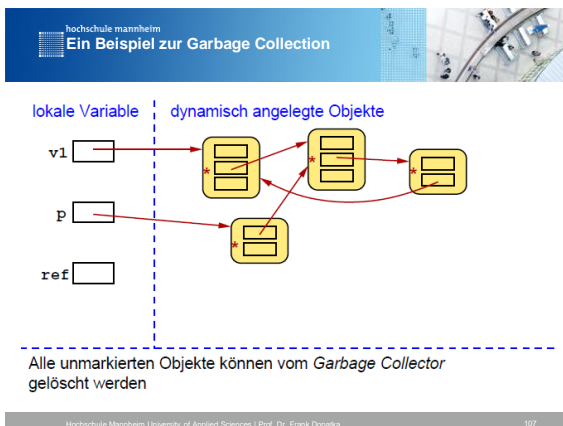
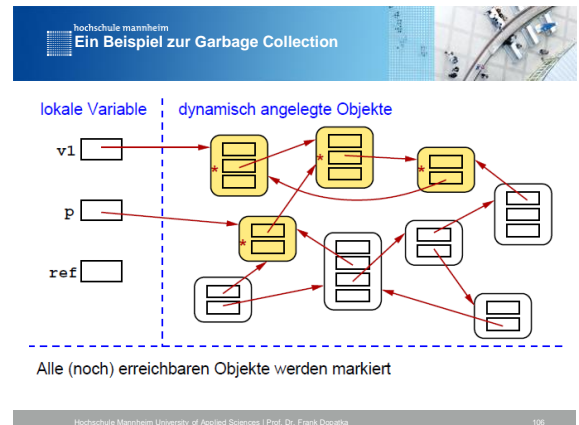
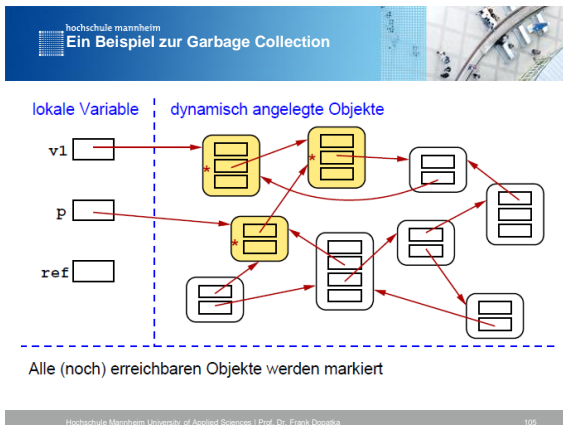
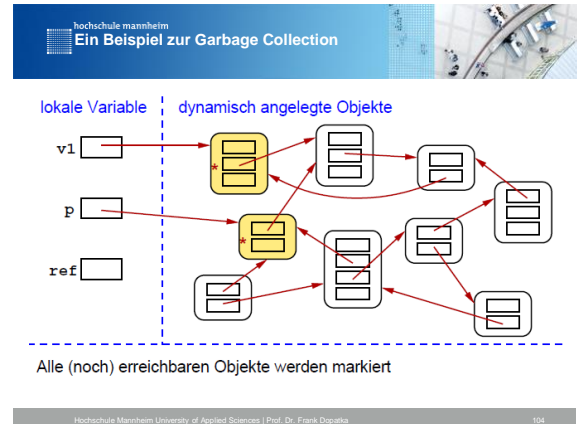
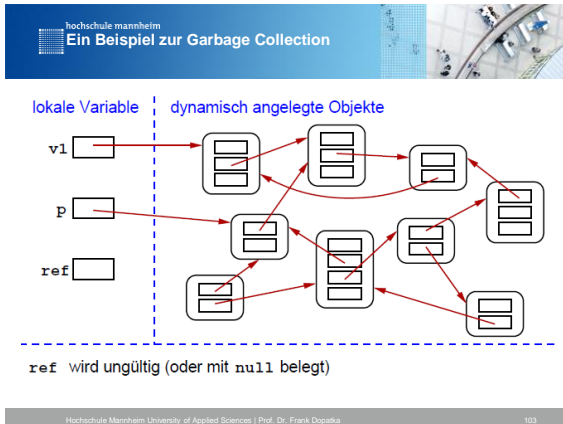
```
System.out.println("Fertig!");
}
```

**Variables** Breakpoints

Name	Value
args	String[0] (id=16)
m1	Mitarbeiter (id=17)
m2	Mitarbeiter (id=22)
name	"Frank Dopatka" (id=26)
personalnummer	1
m3	Mitarbeiter (id=23)

Mitarbeiter Frank Dopatka mit Pers.Nr. 1

## Wie funktioniert das Aufräumen?



hochschule mannheim  
Lebensdauer von Objekten

```

{
  Student s;
  {
    Student thomas = new Student("Thomas");
    Student tom = new Student("Tom");
    // 'Tom' u. 'Thomas' sind verschiedene Studenten
    tom = thomas;
    // Hier wurde nur die Referenz kopiert!
    // tom u. thomas verweisen jetzt auf dasselbe Objekt!
    // Das Objekt 'Tom' ist nicht mehr zugreifbar!
    s = thomas;
  }
  // Das Objekt 'Thomas' existiert noch!
  // (s ist eine Referenz auf 'Thomas')
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatska 108

```
public static void main(String[] args) {
    Student s = new Student("Uli", "Maier", 423634);
    System.out.println("Ich bin der Student " +
        s.getVorname() + " " + s.getNachname() + ".");
    s = null;
    System.gc();
}
```

Ich bin Uli Maier  
Ich sterbe ...

## Annotationen

- Annotationen sind ein Mittel zur Strukturierung von Programmquelltexten, bei der die Erzeugung von Programmtexten und mit der Programmierung verbundener Hilfsdateien teilweise automatisiert wird.
- Als Informationsgrundlage für die automatische Erzeugung der zusätzlichen Dateien werden Informationen aus dem Quelltext herangezogen, die vom Compiler bei der Übersetzung ausgeklammert werden.
- Diese Informationen nennt man Metainformationen, Metadaten, Anmerkungen oder eben Annotationen.
- Die erste Annotation, die man kennenlernt, ist typischerweise **@Override**:
- Damit wird signalisiert, dass Sie eine Methode einer Oberklasse überschreiben wollen.

- Programmiersprachen, die diese Form der Einbindung von Metainformationen ermöglichen, sind u.a.
  - Java und
  - C#.
- Mit Hilfe von Zusatz-Werkzeugen lassen sich Metainformationen auch in Sprachen einbetten, wenn deren Syntax diese nicht explizit unterstützt.
- Diese werden insbesondere im J2EE-Umfeld eingesetzt, um verschiedene Dateien automatisiert zu erzeugen.
- Dazu zählen beispielsweise SQL-Dateien, Deployment-Deskriptoren und die mit Enterprise Java Beans verbundenen Schnittstellen wie Home- und Remote-Interfaces.

- Mit der Sprachversion Java 5 wurden Annotationen als ein eigenes Sprachelement geschaffen.
- Annotationen werden im Quelltext durch ein @-Zeichen gefolgt vom Namen der Annotation gekennzeichnet.
- Zusätzlich ist es auch möglich, Annotationen Parameter zu übergeben.
- In Java können Sie Annotationen auch selbst definieren.
  - Dies ist ähnlich zu der Definition einer eigenen Klasse.
- Selbst erstellte Annotationen können mittels des Annotation Processing Toolkits (APT) zur Laufzeit eines Programms ausgewertet werden.
- Seit Java7 ist APT in **javac** integriert.
- Die in der JDK definierten Annotationen wertet der Java-Compiler selbst aus.

@Override

Die annotierte Methode überschreibt eine Methode aus der Oberklasse oder implementiert eine Methode einer Schnittstelle.

@Deprecated

Das markierte Element ist veraltet und sollte nicht mehr verwendet werden.

@SuppressWarnings

Unterdrückt bestimmte Compiler-Warnungen.

- Die Annotationen `@Override` und `@Deprecated` gehören zur Klasse der Marker-Annotationen, weil keine zusätzlichen Angaben nötig und erlaubt sind.
- Zusätzlich gibt es die single-value annotation, die genau eine zusätzliche Information bekommt sowie eine volle Annotation mit beliebigen Schlüssel/Werte-Paaren.

<code>@Annotationstyp</code>	Marker-Annotation
<code>@Annotationstyp( Wert )</code>	Annotation mit genau einem Wert
<code>@Annotationstyp( Schlüssel1=Wert1, Schlüssel2=Wert2, ... )</code>	Volle Annotation mit Schlüssel/Werte-Paaren

- Die Annotation `@SuppressWarnings` steuert Compiler-Warnungen. Unterschiedliche Werte bestimmen genauer, welche Hinweise unterdrückt werden.
- Beliebt ist die Annotation bei der Umstellung von älterem Quellcode mit nicht-generischen Datenstrukturen, um die Anzahl der Warnungen zu minimieren.
- Da sich mit Java 5 das Klassenformat änderte, gibt der Compiler beim Übersetzen älterer Klassen in Verbindung mit Collections schnell eine „unchecked“-Meldung aus...

- Der Compiler soll für die folgende, ungenerisch verwendete Liste keine Warnmeldung ausgeben:

```
@SuppressWarnings({ "rawtypes", "unchecked" })
public static void main(String[] args){
    ArrayList liste = new ArrayList();
    liste.add("SuppressWarnings");
    liste.add(new Student("Uli", "Maier", 434342));
}
```

- Die Warnungen würden lauten:
  - `ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized.`
  - `Type safety: The method add(Object) belongs to the raw type ArrayList. References to generic type ArrayList<E> should be parameterized.`

## Wrapper-Klassen

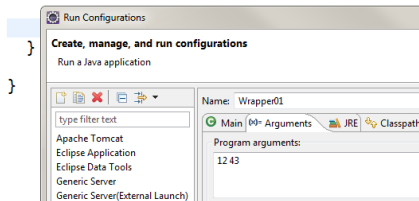
Über die Kommandozeile bzw. über rechten Mausklick -> Run As -> Run Configuration erhält Ihre main-Methode 2 Zahlen als Programm-Argumente:

```
public class Wrapper01 {

    public static void main(String[] args) {

    }

}
```



Beim Ausführen können Sie diese Argumente ausgeben:

```
public static void main(String[] args) {
    System.out.println(args.length);
    System.out.println(args[0]);
    System.out.println(args[1]);
}
```

<terminated> Wrapper01 [Java Application]  
2  
12  
43

Aufgabe: Addieren Sie die beiden Zahlen und geben Sie das Ergebnis auf der Konsole aus!

Zum Konvertieren von Strings in primitive Datentypen verwendet man Wrapper-Klassen!

- Primitive Typen sind keine Objekte!
- An manchen Stellen muss man aber Objekte anstatt primitiver Typen verwenden.
- Wrapper-Klassen (wrapper classes) stellen für jeden primitiven Typ einen passenden Klassen-Typ zur Verfügung....

Primitiver Typ	Wrapper
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

```
int i = 5;
Integer a = new Integer(i);
int j = a.intValue();
Integer b = Integer.valueOf("42");
int k = Integer.parseInt("42");
```

Seit Java 5 kann der Compiler automatisch zwischen primitiven Typen und Klassen-Typen konvertieren, man spricht hier von Autoboxing...

```
Integer k = 5;
int j = k;

Object o = true;
boolean b = new Boolean(true);
```

- Durch das Autoboxing werden primitiven Typen zu Objekten.
- Damit gelten die Regeln für equals() und ==
- Ein Vergleich von Wrapper-Typen mit == ist daher immer falsch.

```
Long a = 150L;
Long b = 200L;

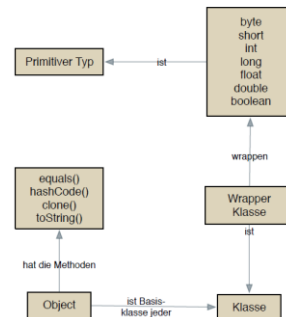
b = a - 50;
System.out.println(a == b);
```

- Durch das Autoboxing werden primitiven Typen zu Objekten.
- Damit gelten die Regeln für equals() und ==
- Ein Vergleich von Wrapper-Typen mit == ist daher immer falsch.

```
int a = 100;
int b = 100;

int c = 1000;
int d = 1000;

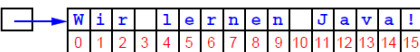
System.out.println(a == b);
System.out.println(c == d);
```



## Arbeiten mit Zeichenketten

- Anhand der Klasse „Kreis“ ist zu erkennen, wie Sie selbst Klassen programmieren können, davon Objekte anlegen und deren Dienste nutzen können.
- Java selbst bietet eine Vielzahl von vorgefertigten Klassen über die Java-API, die in JavaDoc englisch dokumentiert sind.
- Diese Klassen kann und soll man verwenden:
  - Bevor Sie eine Klasse selbst schreiben, schauen Sie nach, ob diese nicht schon existiert.
  - Vermeiden Sie, ein „quadratisches Rad neu zu erfinden“, <http://de.wikipedia.org/wiki/Anti-pattern>

- In Java ist eine Zeichenkette ein Objekt der Klasse String.
- Beispiel:
  - `String motto="Wir lernen Java!"; // automat. new!`
  - `motto` ist eine Referenzvariable.
  - Sie speichert nicht den String, sondern nur die Referenz darauf.
- Ein String ist eine Folge von (Unicode-)Zeichen, wobei jedes Zeichen eine Position hat, die ab 0 gezählt wird:

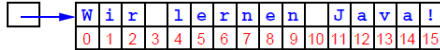
motto 

- Nach einer neuen Zuweisung an die Referenzvariable, z.B. `motto = "Carpe Diem";` ist der String "Wir lernen Java!" nicht mehr zugreifbar.
- Die folgenden Zuweisungen sind unterschiedlich!
  - `motto = null;` // zeigt auf keinen String mehr
  - `motto = "";` // zeigt auf den leeren String
- Die Länge einer Zeichenkette ist wie folgt abrufbar:
  - `int zeichenzahl = motto.length();`
- Ist jedoch `motto==null`, so wird eine `NullPointerException` geworfen.

- Zusammenfügen (Konkatenation) von Zeichenketten
  - `String s = "Zahl 1" + "2"; // "Zahl12"`
  - `s += " ist gleich 3"; // "Zahl12 ist gleich 3"`
  - `s = "elf ist " + 1 + 1; // "elf ist 11"`
  - `s = 1 + 1 + " ist zwei"; // "2 ist zwei"`
  - `s = 0.5 + 0.5 + " ist " + 1; // "1.0 ist 1"`
- Der Operator `+` ist überladen:
  - `int + int`: Addition
  - `String + String`: Konkatenation
  - `String + int` und `int + String`: Umwandlung der Zahl in einen String und anschließende Konkatenation.
- Dies funktioniert analog für andere Datentypen als `int`.

- Vergleichsoperatoren (nur `==` und `!=`):
  - Der Operator `==` liefert `true`, wenn beide Operanden auf denselben String verweisen (Objektidentität!).
- Der Vergleich:
  - `s1.equals(s2)` liefert `true`, wenn `s1` und `s2` zeichenweise übereinstimmen.
  - `s1.compareTo(s2)`
    - < 0, wenn `s1` alphabetisch vor `s2`
    - = 0, wenn `s1` und `s2` zeichenweise übereinstimmen
    - > 0, wenn `s1` alphabetisch nach `s2`
- Vergleiche benötigen immer eine boolsche Bedingung:
 

```
int a=5; String s="Hallo";
if ((a==5) && !(s==null) && (s.equals("nein"))){
    System.out.println("OK");
}
```



- Vergleich mit Anfang und Ende:
  - `boolean a = m.startsWith("Wir");// true`
  - `boolean b = m.endsWith(".");// false`
- Zugriff auf einzelne Zeichen:
  - `char c = m.charAt(5); // 'e'`
- Suche nach Zeichen:
  - `int i = m.indexOf('e',0); // 5`
  - `int j = m.lastIndexOf('e',15); // 8`
- Ausschneiden / Ersetzen:
  - `String s = m.substring(11,15); // "Java"`
  - `String t = m.replace('a','A');// "Wir lernen JAvA!"`

- Strings können nicht verändert werden, sie sind „immutable“.
- Die Operation + oder z.B. die Methode `replace` erzeugen jeweils einen neuen String.
- Dadurch verhalten sich Strings ähnlich wie primitive Datentypen.
- Es gibt auch eine Klasse `StringBuffer`, deren Objekte auch verändert werden können.
- Dies ergibt einen Geschwindigkeitsvorteil, wenn sehr viele Manipulationen an Strings vorgenommen werden.

- Ein regulärer Ausdruck (engl. regular expression, Abk. RegExp oder Regex) ist eine Zeichenkette, die der Beschreibung von Mengen beziehungsweise Untermengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient.
- Reguläre Ausdrücke finden vor allem in der Softwareentwicklung Verwendung; für fast alle Programmiersprachen existieren Implementierungen.
- Reguläre Ausdrücke stellen u.a. ein Filterkriterium für Texte dar, indem der jeweilige reguläre Ausdruck in Form eines Musters mit dem Text abgeglichen wird.
- So ist es beispielsweise möglich, alle Wörter, die mit S beginnen und auf D enden, zu „matchen“, ohne die zwischenliegenden Buchstaben explizit vorgeben zu müssen.

<http://www.regexe.de/hilfe.jsp>

- Für die folgenden Szenarien bietet Java mit den regulären Ausdrücken Lösungen an:
  - Frage nach dem kompletten Übereinstimmen (matching):  
Passt eine Zeichenfolge komplett auf ein Muster?  
Ist die Zeichenfolge z.B. eine korrekte e-Mail?
  - Finde Teilstrings:  
Das Pattern beschreibt nur einen Teil-String; gesucht sind alle Vorkommen dieses Musters in einem Such-String.
  - Ersetze Teilfolgen:  
Das Pattern beschreibt Wörter, die durch andere Wörter ersetzt werden.
  - Zerlegen einer Zeichenfolge:  
Das Muster steht für Trennzeichen, sodass nach dem Zerlegen eine Sammlung von Zeichenfolgen entsteht.  
Beispiel: Zerlegen einer CSV-Datei.

- Ein Pattern-Matcher verarbeitet reguläre Ausdrücke.
- Zugriff darauf bietet die Klasse `Matcher`.
- Dazu kommt die Klasse `Pattern`, die die regulären Ausdrücke in einem vorkompilierten Format repräsentiert.
- Beide Klassen befinden sich im Paket `java.util.regex`.
- Die statische Funktion `Pattern.matches()` und die Objektmethode `matches()` der Klasse `String` testen, ob ein regulärer Ausdruck eine Zeichenfolge komplett beschreibt.

- Der Punkt im regulären Ausdruck steht für ein beliebiges Zeichen, und der folgende Stern ist ein Quantifizierer, der wahllos viele beliebige Zeichen erlaubt:

```
System.out.println(
    Pattern.matches("'.'*", "Hallo Welt")); // true
```

```
System.out.println(
    "Hallo Welt".matches("'.'*")); // true
```

```
System.out.println(
    Pattern.matches("'.'*", "Hallo Welt")); // false
```

- Während die meisten Zeichen aus dem Alphabet erlaubt sind, besitzen Zeichen wie der Punkt, die Klammer, ein Sternchen und einige weitere Sonderfunktionen.
- So maskiert auch ein vorgestelltes »\« das folgende Sonderzeichen aus, was bei besonderen Zeichen wie », « oder »\« wichtig ist.
- Zunächst gilt es, die Anzahl Wiederholungen zu bestimmen.
- Dazu dient ein Quantifizierer (auch Wiederholungsfaktor genannt).
- Für eine Zeichenkette X gilt:

X?	X kommt einmal oder keimnal vor.
X*	X kommt keimnal oder beliebig oft vor.
X+	X kommt einmal oder beliebig oft vor.

- Da in regulären Ausdrücken oftmals ein Bereich von Zeichen, etwa alle Buchstaben, abgedeckt werden muss, gibt es die Möglichkeit, Zeichenmengen zu definieren:

[aeiou]	Zeichen a, e, i, o oder u
[^aeiou]	nicht die Zeichen a, e, i, o, u
[0-9a-zA-Z]	Zeichen 0, 1, 2, ..., 9 oder Groß-/Klein-Buchstaben a, b, c, d, e, f

- Das »^« definiert negative Zeichenmengen, also Zeichen, die nicht vorkommen dürfen.
- Mit dem »-« lässt sich ein Bereich von Zeichen angeben.

- Daneben gibt es vordefinierte Zeichenklassen, die in erster Linie Schreibarbeit ersparen. Die wichtigsten sind:

.	jedes Zeichen
\d	Ziffer: [0-9]
\D	keine Ziffer: [^0-9] bzw. [^d]
\s	Weißraum: [ \t\n\r\f\t]
\S	kein Weißraum: [^\s]
\w	Wortzeichen: [a-zA-Z0-9_]
\W	kein Wortzeichen: [^\w]
\p{Blank}	Leerzeichen oder Tab: [ \t]
\p{Lower}, \p{Upper}	Klein-/Großbuchstabe: [a-z] bzw. [A-Z]
\p{Alpha}	Buchstabe: (\p{Lower}\p{Upper})
\p{Alnum}	alphanumerisches Zeichen: (\p{Alpha}\p{Digit})
\p{Punct}	Punkt-Zeichen: [!\"#\$%&'()*+,-./:;<=>?@[\]^_`{ }~]
\p{Graph}	sichtbares Zeichen: (\p{Alnum}\p{Punct})
\p{Print}	druckbares Zeichen: (\p{Graph})

- Der einfache Aufruf `matches()` auf einem String-Objekt beziehungsweise `Pattern.matches()` ist nur eine Abkürzung für die Übersetzung eines Patterns und Anwendung von `matches()`:

String#matches()	Pattern.matches()
<pre>public boolean matches( String regex ) {     return Pattern.matches( regex, this ); }</pre>	<pre>public static boolean matches( String regex,         CharSequence input ) {     Pattern p = Pattern.compile( regex );     Matcher m = p.matcher( input );     return m.matches(); }</pre>

- Für das erste Beispiel

```
Pattern.matches( ".*", "Hallo Welt" );
```

hätten wir also äquivalent schreiben können:

```
Pattern p = Pattern.compile( ".*" );
Matcher m = p.matcher( "Hallo Welt" );
boolean b = m.matches();
```

- Neben den Quantifizierern
  - ? (einmal oder keimnal),
  - \* (keimnal oder beliebig oft) und
  - + (einmal oder beliebig oft)
- gibt es drei weitere Quantifizierer, die es erlauben, die Anzahl eines Vorkommens genauer zu beschreiben:
  - X{n}
    - X muss genau n-mal vorkommen.
  - X{n,}
    - X kommt mindestens n-mal vor.
  - X{n,m}
    - X kommt mindestens n-, aber maximal m-mal vor.



```
Pattern p =  
Pattern.compile("(\\w|-)+@\\w{\\w|-}*\\. [a-z]{2,3}");
```

- Welche Zeichenketten passen auf p?
- Welche Zeichenketten passen nicht auf p?
- Ist dies eine vollständige Prüfung der Gültigkeit von e-Mails?
- Treffen auch ungültige Mail-Adressen auf das Pattern zu?
- Treffen evtl. gültige Mail-Adressen nicht auf das Pattern zu?

- Suchen Sie reguläre Ausdrücke für
    - E-Mail Adressen,
    - Telefon-Nummern,
    - ISBN-Nummern,
    - Fahrgestell-Nummern von Autos und
    - Steuer-Identifikationsnummern
- mit dem Ziel, Eingegebene Zeichenketten auf Gültigkeit zu prüfen.

- Bisher wurde mit regulären Ausdrücken lediglich festgestellt, ob eine Zeichenfolge vollständig auf ein Muster passt.
- Die Matcher-Klasse kann jedoch auch feststellen, ob sich eine durch ein Muster beschriebene Teilfolge im String befindet.
- Dazu dient die Methode `find()`. Sie hat zwei Aufgaben:
  1. Zunächst sucht sie nach einer Fundstelle und gibt bei Erfolg `true` zurück.
  2. Das Nächste ist, dass jedes Matcher-Objekt einen Zustand mit Fundstellen besitzt, den `find()` aktualisiert.
- Die Methode `group()` gibt von einem Matcher-Objekt den erkannten Substring zurück und `start()` bzw. `end()` die entsprechenden Positionen.
- Wiederholte Aufrufe von `find()` setzen die Positionen weiter. Beispiel...

```
String s = "Demnach, welcher verheiratet, "+  
"der tut wohl; welcher aber nicht verheiratet, der tut besser." +  
" 1. Korinther 7, 38";  
Matcher m=Pattern.compile("\\d+").matcher(s);  
while (m.find()){  
    System.out.printf( "%s an Position [%d,%d] %n",  
        m.group(), m.start(), m.end());  
}
```

```
1 an Postion [94,95]  
7 an Postion [107,108]  
38 an Postion [110,112]
```

- Die drei Operatoren `?`, `*` und `+` haben die Eigenschaft, die längste mögliche Zeichenfolge abzudecken; das nennt sich gierig (engl. greedy).
- Deutlich wird diese Eigenschaft bei dem Versuch, in einem HTML-String alle fett gesetzten Teile zu finden.
- Gesucht ist also ein Ausdruck, der im String  

```
String s = "Echt <b>fett</b>. <b>Cool</b>!"
```

 die Teilfolgen `<b>fett</b>` und `<b>Cool</b>` erkennt.  
 Der erste Versuch für ein Programm lautet...

```
Pattern p=Pattern.compile("<b>.*</b>");  
Matcher m=p.matcher(s);
```

```
while (m.find()){  
    System.out.println(m.group());  
}
```

- Die Ausgabe lautet leider `<b>fett</b>. <b>Cool</b>`.
- Das verwundert nicht, denn mit dem Wissen, dass `*` gierig ist, passt `<b>.*</b>` auf die Zeichenkette vom ersten `<b>` bis zum letzten `</b>`!

- Die Lösung ist der Einsatz eines nicht gierigen Operators, den man auch »genügsam«, »zurückhaltend«, »non-greedy« oder, »reluctant« nennt.
- In diesem Fall wird hinter dem Qualifizierer einfach ein Fragezeichen gestellt:

Gieriger Operator	Nicht gieriger Operator
X?	X??
X*	X*?
X+	X+?
X{n}	X{n}?
X{n, }	X{n, }?
X{n,m}	X{n,m}?

- So ergibt der Code...

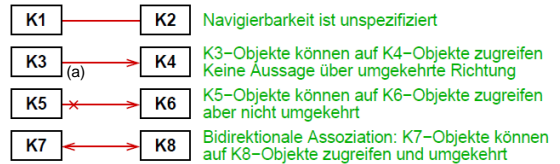
```
Pattern p=Pattern.compile("<b>.*?</b>");
Matcher m=p.matcher(s);
while (m.find()) {
    System.out.println(m.group());
}
```

die gewünschte Ausgabe

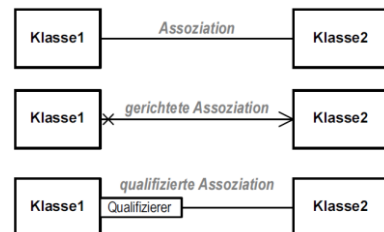
```
<b>fett</b>
<b>Cool</b>
```

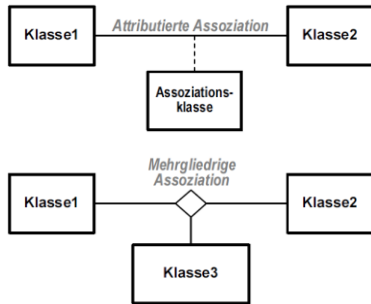
## Assoziation in UML & Java

- „Die Assoziation ist von A nach B navigierbar.“
- Objekte von A können auf Objekte von B zugreifen, aber nicht notwendigerweise umgekehrt.
- Objekte von A können auf Objekte von B „kennen“.
- Jeder Zugriff erfolgt über einen Methodenaufruf; also muss ein A-Objekt eine Referenz auf ein B-Objekt haben.

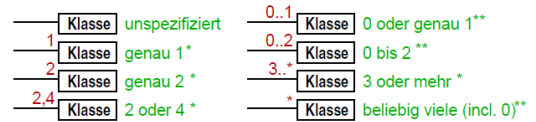


- Beispiel (a) bedeutet, daß das K3-Objekt das K4-Objekt bzw. K4-Objekte kennen können muss.
- Dazu muss ein K3-Objekt eine Referenz auf K4-Objekte haben können.
- Die Referenz muss nicht stets ausgeprägt sein, sie kann auch **null** sein.
- Die Kenntnis eines anderen Objektes ist damit bereits realisiert:
- „Jedes Tier hat einen Namen“ bedeutet, dass jedes Tier-Objekt bereits bei seiner Erstellung eine Referenz **name** auf ein Objekt der Klasse String besitzt.

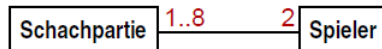




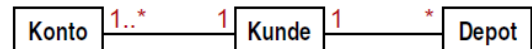
- Eine Assoziation sagt zunächst nur, daß ein Objekt andere Objekte kennen kann.
- Die Angabe einer Multiplizität legt fest, wieviele Objekte ein Objekt kennen kann oder muss.
  - Sind es mehr als 1: Speicherung in einem Array oder einer Collection
- Die Multiplizität wird in der UML am Ende der Assoziations-Linie notiert:



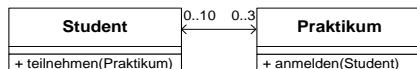
- \* **Muß-Assoziation:** Objekt muß in Beziehung zu anderen stehen
- \*\* **Kann-Assoziation:** Objekt kann, muß aber nicht in Beziehung stehen



- Jede Schachpartie wird von zwei Spielern gespielt.
- Ein Spieler spielt 1 bis 8 Partien gleichzeitig.



- Ein Kunde muß mindestens ein Konto besitzen.
- Ein Konto gehört zu genau einem Kunden.
  - Wenn der Kunde gelöscht wird, so muss auch das Konto gelöscht werden!
- Ein Kunde kann kein oder beliebig viele Depots besitzen.
- Ein Depot gehört zu genau einem Kunden.
  - Wenn der Kunde gelöscht wird, dann müssen auch alle Depots gelöscht werden!



- Bitte coden Sie das!
- Nach Möglichkeit ohne eine Endlos-Rekursion!

## Aggregation & Komposition in UML & Java

## hochschule mannheim Was bedeutet Aggregation & Komposition?

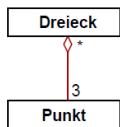
- Es bedeutet, dass etwas zusammengesetzt ist. Etwas „besteht aus“ etwas Anderem bzw. etwas „ist Teil von“ etwas Anderem.
- Beispiel: „Ein Auto besteht aus einer Karosserie, 4 Rädern, ...“
- Aggregation:
  - Die Teile existieren selbständig & können (gleichzeitig) zu mehreren Aggregat-Objekten gehören.

## hochschule mannheim Was bedeutet Aggregation & Komposition?

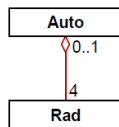
- Komposition als starke Form der Aggregation:
  - Das Teil-Objekt gehört zu genau einem Komposit-Objekt und kann nicht ohne sein Komposit-Objekt existieren.
  - Es kann nicht zu einem Zeitpunkt Teil verschiedener Komposit-Objekte sein.
  - Beim Erzeugen / Löschen des Komposit-Objekts werden auch seine Teil-Objekte erzeugt / gelöscht.
  - Bei einer Komposition dürfen die Objekt-Referenzen nicht nach aussen weiter gegeben werden, da sie sonst allein weiter existieren können; weil noch jemand eine Referenz darauf besitzt!

## hochschule mannheim Aggregation & Komposition in der UML

### Aggregation

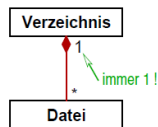


Ein Dreieck besteht immer aus 3 Punkten.  
Ein Punkt ist Teil von beliebig vielen (incl. 0) Dreiecken.



Ein Auto besteht (u.a.) aus 4 Rädern.  
Ein Rad ist Teil von höchstens einem Auto (es gibt auch Räder ohne Auto).

### Komposition



Ein Verzeichnis besteht aus beliebig vielen Dateien.  
Dateien stehen immer in einem Verzeichnis. Wird dieses gelöscht, so auch alle enthaltenen Dateien.

## hochschule mannheim Aggregation & Komposition in der UML



## hochschule mannheim Realisierung einer Aggregation: Dreiecke & Punkte - Testklasse

```
public class TestGeo {
    public static void main(String[] args) {
        Punkt p1=new Punkt(2,5,3);
        Punkt p2=new Punkt(8,6,5);
        Punkt p3=new Punkt(23,4,7);
        Dreieck d1=new Dreieck(p1,p2,p3);
        System.out.println(d1);
    }
}
```

## hochschule mannheim Realisierung einer Aggregation: Dreiecke & Punkte - Klasse Punkt

```
public class Punkt {
    private int x,y,z;

    public Punkt(int x,int y,int z){
        setX(x); setY(y); setZ(z);
    }
    @Override
    public String toString(){
        return "Punkt (" +x+" "+y+" "+z+") ";
    }

    public void setX(int x) {
        this.x = x;
    }
    public int getX() {
        return x;
    }
    public void setY(int y) {
        this.y = y;
    }
    public int getY() {
        return y;
    }
    public void setZ(int z) {
        this.z = z;
    }
    public int getZ() {
        return z;
    }
}
```

hochschule mannheim  
Realisierung einer Aggregation:  
Dreiecke & Punkte - Klasse Dreieck

```

public class Dreieck {
    private Punkt p1,p2,p3;

    public Dreieck(Punkt p1, Punkt p2, Punkt p3){
        // immer OK?
        setP1(p1);
        setP2(p2);
        setP3(p3);
    }

    @Override
    public String toString(){
        return "Dreieck (" +p1+ "/" +p2+ "/" +p3+ ")";
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 169

hochschule mannheim  
Realisierung einer Aggregation:  
Dreiecke & Punkte - Klasse Dreieck

```

public void setP1(Punkt p1) {
    // immer OK?
    this.p1 = p1;
}

public Punkt getP1() {
    return p1;
}

public void setP2(Punkt p2) {
    // immer OK?
    this.p2 = p2;
}

public Punkt getP2() {
    return p2;
}

public void setP3(Punkt p3) {
    // immer OK?
    this.p3 = p3;
}

public Punkt getP3() {
    return p3;
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 170

hochschule mannheim  
Realisierung einer Aggregation:  
Dreiecke & Punkte - Klasse Dreieck

```

public Punkt[] getPunkte() {
    Punkt[] punkte=new Punkt[3];
    punkte[0]=getP1();
    punkte[1]=getP2();
    punkte[2]=getP3();
    return punkte;
}
}

```

- Mögliche Realisierungen einer Aggregation sind...
- über den Konstruktor des Ganzen bei der „Geburt“ des Ganzen.
- nachträglich über setter des Ganzen.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 171

hochschule mannheim  
Realisierung einer Komposition:  
Verzeichnisse & Dateien - Testklasse

```

public class TestVerzeichnisse {
    public static void main(String[] args) throws Exception {
        Verzeichnis v1=new Verzeichnis("benutzer");
        v1.addDatei("dopatka.txt", "Ich bin Frank in der Datei");
        v1.addDatei("meier.txt", "Ich bin der Uli Meier...");
        System.out.println(v1.getDatei("meier.txt"));
        System.out.println(v1.getDateien());
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 172

hochschule mannheim  
Realisierung einer Komposition:  
Verzeichnisse & Dateien - Testklasse

kein public, also package-Sichtbarkeit, ~ in UML

```

class Datei {
    private String daten;
    private String name;

    public Datei(String name){
        setName(name);
    }

    @Override
    public String toString(){
        return getName();
    }

    public void setDaten(String daten) {
        this.daten = daten;
    }

    public String getDaten() {
        return daten;
    }

    public void setName(String name) {
        if ((name==null) || (name.length()<2))
            throw new RuntimeException("Name ungültig!");
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 173

hochschule mannheim  
Realisierung einer Komposition:  
Verzeichnisse & Dateien - Verzeichnis

```

public class Verzeichnis {
    private String name;
    private static final int dateienMax=3;
    private Datei[] dateien=new Datei[dateienMax];
    private int dateienAnz=0;

    public Verzeichnis(String name){
        this.setName(name);
    }

    public void setName(String name) {
        if ((name==null) || (name.length()<2))
            throw new RuntimeException("Name ungültig!");
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 174

## Realisierung einer Komposition: Verzeichnisse & Dateien - Verzeichnis

```
public int getDateIndex(String name) {
    for(int i=0;i<dateienAnz;i++){
        String dateiName=dateien[i].getName();
        if ((dateiName!=null) && dateiName.equals(name)) return i+1;
    }
    return -1; besser:
    throw new RuntimeException("gibts nisch");
}

public String getDateien(){
    String ausgabe="<<Verzeichnis "+getName()+">>\n";
    for(int i=0;i<dateienAnz;i++){
        ausgabe+=" "+dateien[i]+"\\n";
    }
    return ausgabe;
}
```

## Realisierung einer Komposition: Verzeichnisse & Dateien - Verzeichnis

```
public void addDatei(String name, String daten){
    if (dateienAnz==dateienMax)
        throw new RuntimeException("Maximale Anzahl Dateien bereits erreicht!");
    Datei d=new Datei(name);
    d.setDaten(daten);
    dateien[dateienAnz++]=d;
}

public String getDatei(String name){
    int index=getDateiIndex(name);
    if (index>0) return dateien[index-1].getDaten();
    return "";
}
```

## Realisierung einer Komposition: Verzeichnisse & Dateien – Der Test

```
public class TestVerzeichnisse {
    public static void main(String[] args) throws Exception {
        Verzeichnis vl=new Verzeichnis("benutzer");
        vl.addDatei("dopatka.txt", "Ich bin Frank in der Datei");
        vl.addDatei("meier.txt", "Ich bin der Uli Meier...");
        System.out.println(vl.getDatei("meier.txt"));
        System.out.println(vl.getDateien());
    }
}
```

```
<terminated> TestVerzeichnisse [Java Application]
Ich bin der Uli Meier...
<<Verzeichnis benutzer>>
  dopatka.txt
  meier.txt
```

## Verschiedene Möglichkeiten zur Realisierung einer Komposition

### 1. Über die Package-Sichtbarkeit...

```
package überPackage;

import java.util.ArrayList;

public class Ganzes {
    private ArrayList<Teil> teile=
        new ArrayList<Teil>();

    public Ganzes(){
        for(int i=1;i<=5;i++){
            teile.add(new Teil());
        }
    }

    package überPackage;

    class Teil {
        public Teil(){
        }
    }
}
```

## Verschiedene Möglichkeiten zur Realisierung einer Komposition

### 1. Realisierung über die Package-Sichtbarkeit...

```
import überPackage.*;

public class TestPackage {
    public static void main(String[] args) {
        Ganzes g=new Ganzes();
        Teil t=new Teil();
    }
}
```

## Verschiedene Möglichkeiten zur Realisierung einer Komposition

### 1. Realisierung über die Package-Sichtbarkeit...

```
package überPackage;

import java.util.ArrayList;

public class Ganzes {
    private ArrayList<Teil> teile=
        new ArrayList<Teil>();

    public Ganzes(){
        for(int i=1;i<=5;i++){
            teile.add(new Teil());
        }
    }

    package überPackage;

    class Teil {
        public Teil(){
        }
    }
}
```

## 2. Realisierung über den Konstruktor des Teils...

```
package überKonstruktor;

import java.util.ArrayList;

public class Ganzes {
    private ArrayList<Teil> teile=
        new ArrayList<Teil>();

    public Ganzes(){
        for(int i=1;i<=5;i++){
            teile.add(new Teil(this));
        }
    }
}
```

## 2. Realisierung über den Konstruktor des Teils...

```
package überKonstruktor;

public class Teil {
    private Ganzes g;
    public Teil(Ganzes g){
        if (g==null)
            throw new RuntimeException("'" +
                "Das Teil kann nicht ohne das Ganze existieren!");
        this.g=g;
    }

    public Ganzes getGanzes(){
        return this.g;
    }
}
```

## 3. Realisierung über innere Member-Class...

```
package überInnereKlasse;

import java.util.ArrayList;

public class Ganzes {
    private ArrayList<Teil> teile=new ArrayList<Teil>();

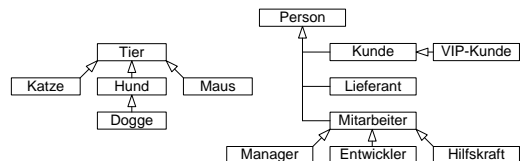
    public Ganzes(){
        for(int i=1;i<=5;i++){
            teile.add(new Teil());
        }
    }

    private class Teil {
        public Teil(){
        }
    }
}
```

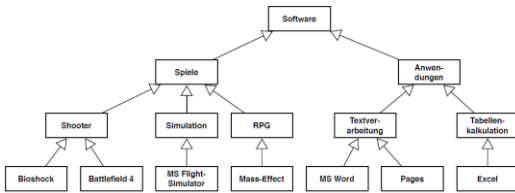
# Vererbung

- Eine Unterklasse übernimmt (erbt) von ihren Oberklassen
  - alle Eigenschaften, auch die Klassen-Eigenschaften, die **public** oder **protected** sind, sowie ggf. auch deren Anfangswert.
  - alle Methoden (auch Klassen-Methoden), die **public** oder **protected** sind:
  - Somit können alle Methoden einer Oberklasse können auch auf ein Objekt der Unterklasse angewendet werden.
- alle Assoziationen, also die Kenntnisse zu anderen Klassen.
- Die Unterklasse kann zusätzliche Eigenschaften, Methoden und Assoziationen hinzufügen, aber ererbte nicht löschen!
- Die Unterklasse kann das Verhalten neu definieren, indem sie nicht-statische Methoden der Oberklasse überschreibt.

- Die Oberklasse generalisiert von den Unterklassen.
- Die Unterklasse ist spezieller als die Oberklasse.
- Ein Objekt der Unterklasse „ist ein“ Objekt der Oberklasse.
- Beispiele:



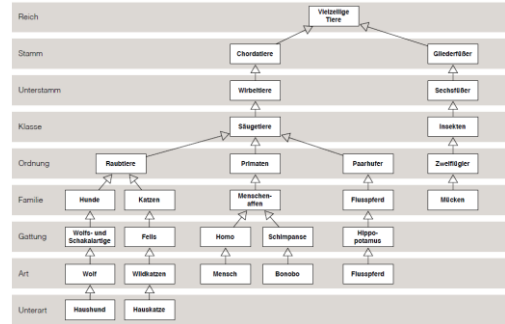
## hochschule mannheim Ein UML Vererbungsbeispiel: Computersoftware



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

187

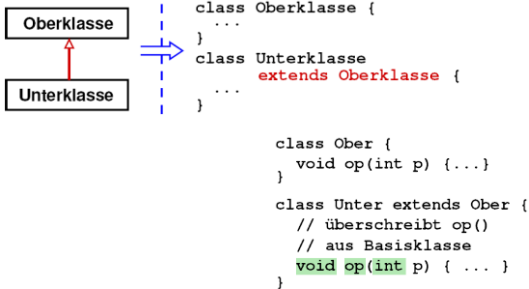
## hochschule mannheim Ein UML Vererbungsbeispiel: Biologische Systematik



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

188

## hochschule mannheim Vererbung in Java und UML



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

189

## hochschule mannheim Überschreiben vs. Überladen

- Häufig will eine Subklasse nicht exakt das Verhalten der Superklasse erben.
- Subklassen können daher Methoden der Superklasse durch eigenen Implementierungen ersetzen durch das **Überschreiben**.
- Die überschriebene Methode hat
  - denselben Namen
  - denselben Rückgabebetyp
  - dieselbe Parameterliste
 aber ein anderes Verhalten als die Methode aus der Superklasse.
- Alle Hunde können **bellen()**.
- Aber eine Dogge kann beispielsweise anders **bellen()** als ein gewöhnlicher Hund.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

190

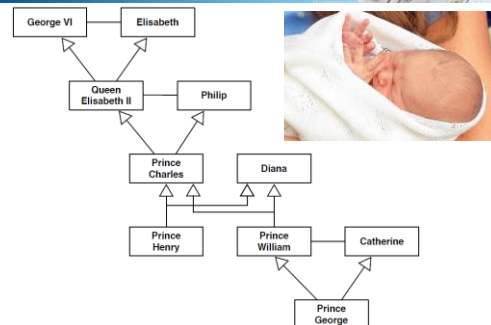
## hochschule mannheim Überschreiben vs. Überladen

- In Java dürfen mehrere Methoden einer Klasse denselben Namen haben, solange sich die Parameterlisten unterscheiden.
- Dabei unterscheiden sich die Anzahl der Parameter und/oder die Typen der Parameter.
- Der Rückgabebetyp der Methode wird nicht beachtet.
- Der Compiler wählt die Methode aus, die von der Parameterliste her am besten passt.
- So hatte der Kreis verschiedene Konstruktoren und hat dadurch verschiedene Möglichkeiten angeboten, einen gültigen Kreis zu erzeugen.
- In diesem Fall spricht man von **überladenen** Konstruktoren.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

191

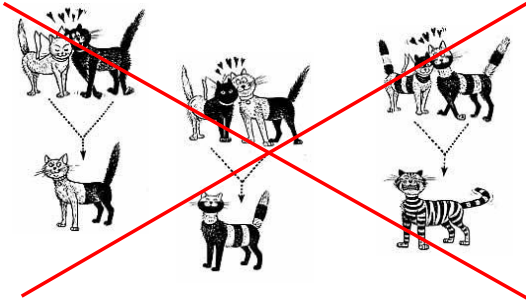
## hochschule mannheim Ein UML Vererbungsbeispiel: Prinz George of Cambridge (\*22.07.2013)



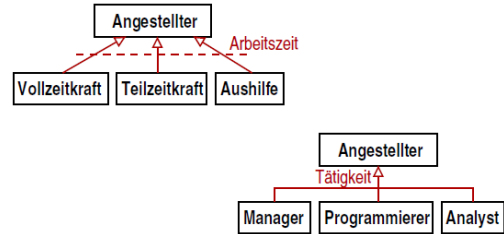
Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

192





- Ein zusätzlicher Diskriminator kann das Kriterium angeben, nach dem klassifiziert wird:



- In Instanzmethoden abgeleiteter Klassen gibt es eine spezielle „Referenzvariable“ **super**.
- Wie **this** muss auch **super** nicht deklariert werden.
- Im Gegensatz zu **this** ist **super** aber keine Referenz auf ein reales Objekt!
- super** erlaubt u.a. den Zugriff auf überschriebene Methoden der Basisklasse:

```
class Ober {
    int op(int i) { ... }
}
class Unter extends Ober {
    int op(int i) { return super.op(i) / 2; }
}
```

*Ruft op(i) in Klasse Ober*

- Beispiel: Auto & Sparauto
- Sparauto** **extends** **Auto**, Sparauto „ist ein“ Auto
- Methode: **getBenzinverbrauch(int gefahreneKM)**

- Die Konstruktoren einer Klasse werden nicht an die Unterklassen vererbt!
- In einem Konstruktor kann mittels **super( [<Parameterliste> ] )** ein Konstruktor der Oberklasse aufgerufen werden:

```
class Shape {
    Shape(int color) { ... }
    ...
}
class Circle extends Shape {
    Circle(double[] center, double radius, int color) {
        super(color); // ruft Konstruktor Shape(int color),
        // muß erste Anweisung sein!
    }
    ...
}
```

- Vor der Ausführung eines Unterklassen-Konstruktors wird immer ein Konstruktor der Oberklasse ausgeführt!
- Falls kein expliziter Aufruf erfolgt, so wird der Default-Konstruktor der Oberklasse ausgeführt:

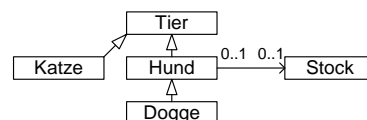
```
class A {
    A() { ... (A) }
}
class B extends A {
    B(int i) { ... (B) }
}
class C extends B {
    C() { super(i); ... (C) }
}
```

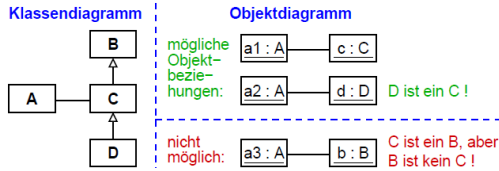
*implizites super()*

Reihenfolge der Konstruktor-Aufrufe bei **new C()**:  
C() → B(1) → A()

Reihenfolge der Abarbeitung der Konstruktor-Rümpfe:  
(A) → (B) → (C)

- Wenn der Hund einen Stock kennen kann, kann auch eine Dogge einen Stock kennen, da die Dogge ja ein Hund ist.
- Nicht jedes Tier kennt einen Stock!
- Das ist nur möglich, wenn das Tier zufällig ein Hund ist.
- Katzen kennen keine Stöcke.





```

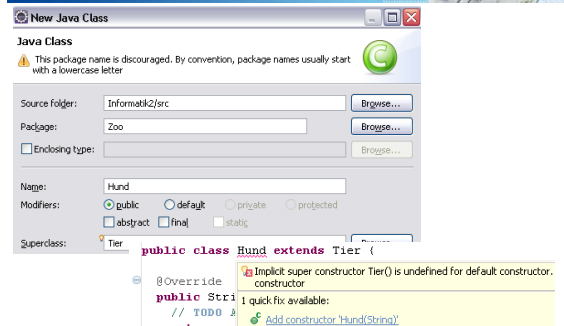
public abstract class Tier{
    private String name;

    private Tier(){ // kein Default-Konstruktor erlaubt
    }
    public Tier(String name){
        this();
        this.setName(name);
    }

    public void setName(String name){
        // ist jeder Name erlaubt?
        this.name=name;
    }
    public String getName(){
        return this.name;
    }

    public abstract String gibLaut();
}
    
```

- Die Klasse Tier ist abstrakt :  
Von dieser Klasse können keine Objekte angelegt werden!
- Tier ist nur eine gemeinsame Schnittmenge von konkreten Unterklassen bezüglich der Eigenschaften und der angebotenen Dienste.
- Tier definiert die Funktionalität `gibLaut()`, die jedoch nicht mit einem Default-Dienst besetzt werden soll:
  - Ein Standard-Laut für ein Tier ist sinnlos!
- Wenn man aber eine konkrete Klasse erstellt, von der man Objekte bilden will, so wird man gezwungen, `gibLaut()` zu implementieren:
  - Jedes Tier muss einen Laut von sich geben!
- Ein ähnliches Beispiel:  
„Person“ als Oberklasse für „Student“ und „Professor“



```

public class Hund extends Tier {
    private String marke;

    public Hund(String name, String marke) {
        super(name);
        setMarke(marke);
    }

    public void setMarke(String marke) {
        this.marke = marke;
    }

    public String getMarke() {
        return marke;
    }

    public String bellen() {
        return getName()+": wow";
    }

    @Override
    public String gibLaut() {
        return bellen();
    }
}
    
```

`@Override`  
`implements Zoo.Tier.gibLaut() {`  
`return bellen();`

```

public class Katze extends Tier {
    public Katze(String name) {
        super(name);
    }

    public String miauen() {
        return getName()+": miau";
    }

    @Override
    public String gibLaut() {
        return miauen();
    }
}
    
```

hochschule mannheim

Abstrakte Klassen:  
Aufbau einer Vererbungs-Hierarchie

```

public class TestTier {
    public static void main(String[] args){

        // Wieso geht das? Tier ist abstract!
        Tier t1=new Hund("Hasso", "XK45");
        Tier t2=new Katze("Nicki");
        Tier t3=new Hund("Bello", "FE75");

        t1.gibLaut();
        t2.gibLaut();
        t3.gibLaut();
    }
}

```

Hasso: wow  
Nicki: miau  
Bello: wow

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

206

hochschule mannheim

Abstrakte Klassen:  
Aufbau einer Vererbungs-Hierarchie

Klasse A hat zwei abstrakte und eine konkrete Methode. Da sie abstrakte Methoden hat, ist sie selber wieder abstrakt

Klasse B implementiert die von A geerbte Methode abstrakt1() und fügt der Vererbungshierarchie eine weitere abstrakte (abstrakt2()) und eine konkrete Methode (konkret2()) hinzu. Da abstrakte Methoden vorhanden sind (eine geerbt, eine eigene) ist die Klasse selber wieder abstrakt.

Klasse C implementiert alle verbliebenen (und geerbten) abstrakten Methoden und ist daher eine konkrete Klasse.

Klasse D ersetzt die von B stammende Implementierung der Methode abstrakt1() durch eine eigene (überschreibt sie).

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

208

hochschule mannheim

Was ist Polymorphie & dynamisches Binden?

- Eine Methode ist polymorph, wenn sie in verschiedenen Klassen die gleiche Signatur hat, jedoch erneut implementiert ist.
- Gibt es in einem Vererbungsbaum einer Klassenhierarchie mehrere Methoden auf unterschiedlicher Hierarchieebene, jedoch mit gleicher Signatur, so wird erst zur Laufzeit bestimmt, welche der Methoden verwendet wird.
- Bei einer mehrstufigen Vererbung wird jene Methode verwendet, die im Vererbungsbaum am weitesten "unten" liegt.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

207

hochschule mannheim

Was ist Polymorphie & dynamisches Binden?

- t ist eine Referenz (auch: Attribut, Variable, Parameter) auf den statischen Typ **Tier**. Hund ist ein dynamischer **Typ**.
- Der Aufruf der Methode **gibLaut()** bindet das Tier-Objekt zur Laufzeit dynamisch an die Klasse **Hund** und führt dort den Methoden-Aufruf aus.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

209

hochschule mannheim

Was ist Polymorphie & dynamisches Binden?

```

public class TestTier {
    public static void main(String[] args){

        // Wieso geht das? Tier ist abstract!
        Tier[] tiere=new Tier[2];
        tiere[0]=new Hund("Hasso", "XK45");
        tiere[1]=new Katze("Nicki");
        tiere[2]=new Hund("Bello", "FE75");

        for(Tier t:tiere){
            System.out.println(t.gibLaut());
        }
    }
}

```

Hasso: wow  
Nicki: miau  
Bello: wow

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

209

hochschule mannheim

Wieso geht das nicht?

```

public class TestTier {
    public static void main(String[] args){

        // Wieso geht das? Tier ist abstract!
        Tier[] tiere=new Tier[2];
        tiere[0]=new Hund("Hasso", "XK45");
        tiere[1]=new Katze("Nicki");
        tiere[2]=new Hund("Bello", "FE75");

        for(Tier t:tiere){
            System.out.println(t.bellen());
        }
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

210



- Für Ausnahmen gilt die handle or declare Regel:
- Die Ausnahme wird durch einen `try/catch` behandelt oder
- die Ausnahme muss bei der Methode mit dem `throws` Schlüsselwort angegeben werden.
- Dies gilt nicht für Runtime Exceptions und Error:
- Programmierfehler sollen nicht behandelt werden.
- Ein Error kann (eigentlich) nicht behandelt werden.

```
public void userInterface() {

    String dateiName = askUser();

    try {
        dateiAnlegen(dateiName);
    }
    catch (FileNotFoundException ex) {
        // Benutzer erneut nach Dateinamen Fragen
    }
    catch (IOException ex) {
        // Benutzer auf Problem hinweisen
    }
}
```

- RuntimeExceptions werden meist durch Fehler im Programmcode verursacht.
- Dann müssen sie nicht behandelt werden; statt dessen sollte der Programmcode verbessert werden.
- Beispiele, vgl. Dokumentation zum Paket `java.lang`:
- `ArithmeticException`: z.B. `1/0` (ganzzahlig!)
- `IndexOutOfBoundsException`: z.B. `array[-1]`
- `NegativeArraySizeException`: z.B. `new double[-5]`
- `NullPointerException`:  
z.B. `Hund meinHund = null; meinHund.gibLaut();`
- `ClassCastException`:  
z.B. `Tier t = new Hund("Bello"); Katze k = (Katze)t;`

- 1. Im einfachsten Fall wird die Exception gar nicht behandelt!
- Die Methode bricht beim Auftreten der Exception sofort ab.
- Die Exception wird an Aufrufer der Methode weitergegeben.
- Wenn die Exception nicht spätestens in der main-Methode gefangen wird, dann bricht das Programm ab.
- Jede Methode muß deklarieren, welche Exceptions sie werfen kann:  
`void anmelden(...) throws AnmeldungException`  
oder  
`public static void main(...) throws AnmeldungException, OtherException`
- Ausnahmen davon sind Error und RuntimeException.

- 2. Behandlung von Exceptions mit dem `try-catch` Konstrukt:

```
try {
    int[] zahlen=new int[10];
    int z = zahlen[index];
    double kehrwert = 1.0 / z;
    System.out.println(kehrwert);
    // ...
}
catch (IndexOutOfBoundsException e) {
    System.out.println("Unzulässiger Index");
}
catch (ArithmeticException e) {
    System.out.println("Fehler: " + e.getMessage());
}
catch (Exception e) {
    System.out.println(e);
}
```

- Wenn eine Exception im `try`-Block auftritt, dann
- wird der `try`-Block verlassen.
- wird der erste „passender“ `catch`-Block ausgeführt und die Ausführung nach dem letzten `catch`-Block fortgesetzt.
  - Falls kein passender `catch`-Block vorhanden ist, dann wird die Methode unmittelbar abgebrochen und die Exception an den Aufrufer weitergegeben.
- Wann ist ein `catch`-Block „passend“?
- Wenn das erzeugte Exception-Objekt an den Parameter des `catch`-Blocks zugewiesen werden kann.
- Dies ist dann möglich, wenn die erzeugte Exception identisch mit der spezifizierten Exception-Klasse ist oder eine Unterklasse davon ist.

- Nach dem letzten **catch**-Block kann noch ein **finally**-Block angefügt werden.
- Auch **try-finally** ohne **catch** ist erlaubt.
- Die Anweisungen dieses Blocks werden immer nach Verlassen des **try**-Blocks ausgeführt, egal ob
  - der **try**-Block normal beendet wird,
  - der Block (und die Methode) durch **return** verlassen wird,
  - eine Exception auftritt und durch **catch** gefangen wird; dann wird **finally** nach dem **catch** ausgeführt oder ob
  - eine Exception auftritt und an den Aufrufer weitergegeben wird.
- Anwendung: „Aufräumarbeiten“, z.B. Löschen temporärer Dateien, Schließen von Fenstern, ...

```
private static int testFunktion() {
    try{
        int y=2; int z=0;
        int x=y/z;
        System.out.println("x ist:"+x);
        return 0;
    }
    catch (Exception e){
        System.out.println("Fehler!");
        return 1;
    }
    finally{
        return 2;
    }
}

public static void main(String[] args) {
    System.out.println("Ausgabe: "+testFunktion());
}
```

- Exceptions können im Programm explizit durch die Anweisung **throw** ausgelöst werden:
 

```
public static double invertiere(double x){
    if (x==0.0){ // wenn x=0 ist, dann werfe...
        throw new ArithmeticException("Div. durch 0");
    }
    return 1.0 / x;
}
```
- **throw** kann auch in einem **catch**-Block verwendet werden, um einen Fehler teilweise zu behandeln und dann weiterzuwerfen:
 

```
catch (Exception e) {
    ... // Lokale Fehlerbehandlung ...
    throw e; // Exception an Aufrufer weitergeben
}
```

- **throw** kann auch in einem **catch**-Block verwendet werden, um einen Fehler nur teilweise zu behandeln und dann weiterzuwerfen:
 

```
try{
    //...
}
catch (Exception e) {
    // Lokale Fehlerbehandlung ...
    throw e; // Exception an Aufrufer weitergeben
}
```

- Folgende Konstruktionen sollten vermieden werden:
  - Sehr lange **try**-Blöcke  
Der **try**-Block sollte nur den Code umfassen, der den Fehler erzeugen kann.
  - **catch** von **Exception**  
Da **Exception** die Mutter aller Ausnahmen ist, fängt man damit auch **Runtime-Exceptions**, die man aber nicht fangen sollte.
  - Leere **catch**-Blöcke  
Eine Ausnahme signalisiert einen Fehlerfall; sie einfach zu ignorieren ist fast nie richtig.
  - **printStackTrace()**  
**catch**-Blöcke, die die Ausnahme nur ausdrucken, ignorieren ebenfalls den Fehler, statt ihn zu behandeln.

```
try {
    FileInputStream fis = new FileInputStream("/tmp/gibtsnicht");
    String s = null;
    s = s.toUpperCase();
} catch (Exception e) { // SCHLECHT!!
    // Fehlerbehandlung
}

FileInputStream fis = null;

try {
    fis = new FileInputStream("/tmp/gibtsnicht");
} catch (IOException e) {
    e.printStackTrace(); // SCHLECHT!!
}

fis.read();
```

## hochschule mannheim Eigene Fehler-Klassen schreiben

- Sie selbst können Klassen von Exception ableiten und somit eine eigene Klassen-Hierarchie für ein Fehler-Management entwerfen.
- Beim Erstellen eines neuen Fehler-Objektes kann nun automatisch
  - eine E-Mail oder
  - eine SMS versendet werden oder
  - eine Protokoll-Datei geschrieben werden.

## hochschule mannheim Die erste eigene Fehler-Klasse

```
public class AnwendungsFehler extends Exception {
    private int id;

    public AnwendungsFehler(int id, String meldung) {
        super(meldung);
        setId(id);
    }

    private void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    @Override
    public String toString() {
        return "Fehler ID "+getId()+" aufgetreten: "+getMessage();
    }
}
```

## hochschule mannheim Test der eigenen Fehler-Klasse

```
public class TestFehler2 {
    public static void main(String[] args) {
        try {
            testMethode();
        } catch (AnwendungsFehler e) {
            System.err.println(e);
        }
    }

    private static void testMethode() throws AnwendungsFehler {
        if (1!=2) throw new AnwendungsFehler(23, "Das ist nicht gleich!");
    }
}
```

```
<terminated> TestFehler2 [Java Application] C:\Programme\jre\bin\javaw.exe
Fehler ID 23 aufgetreten: Das ist nicht gleich!
```

## hochschule mannheim Test der eigenen Fehler-Klasse

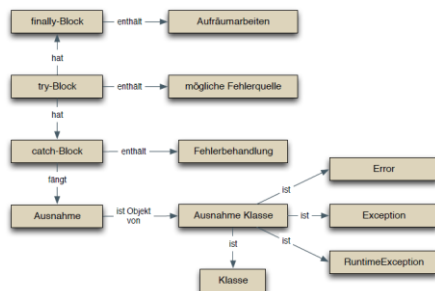
```
} catch (AnwendungsFehler e) {
    e.printStackTrace();
}

<terminated> TestFehler2 [Java Application] C:\Programme\jre\bin\javaw.exe (27.12.2009 17:05:40)
Fehler ID 23 aufgetreten: Das ist nicht gleich!
    at Fehler.TestFehler2.testMethode(TestFehler2.java:13)
    at Fehler.TestFehler2.main(TestFehler2.java:6)

} catch (AnwendungsFehler e) {
    System.out.println(e.getMessage());
}

<terminated> TestFehler2 [Java Application]
Das ist nicht gleich!
```

## hochschule mannheim Begriffsübersicht



## Enums

- Wie kann ich ...
- einen festen Satz von Werten vorgeben?
- dafür sorgen, dass nur diese Werte und kein anderer Wert übergeben werden kann?
- die Werte typsicher machen?
- mir dabei die Überprüfung vom Compiler abnehmen lassen?
- die Werte in gängigen Kontrollstrukturen verwenden?

- In der Schreibweise für Aufzählungen ersetzt das Schlüsselwort **enum** das Schlüsselwort **class** und ist auch ähnlich zu nutzen.
- Eine **enum**-Deklaration erlaubt auch die Deklaration von Methoden und Variablen.
- Somit verhält sie sich wie eine bekannte Klassendeklaration, nur dass leider keine Vererbung erlaubt ist.

```
«enumeration»
Weekday
+ MONDAY
+ SATURDAY
+ TUESDAY
+ SUNDAY
+ FRIDAY
+ WEDNESDAY
+ THURSDAY

public class Weekday {
    public static final Weekday MONDAY;
    public static final Weekday TUESDAY;
    ...
}

public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
```

• Achtung:  
Enums werden per Referenz übergeben, daher ist immer noch **null** als separate Ausprägung zu berücksichtigen!

```
public class TestWeekday {
    public static void main(String[] args) {
        Weekday day = Weekday.FRIDAY;
    }
}
```

```
public class TestWeekday {
    public static void main(String[] args) {
        Weekday day = Weekday.SATURDAY;
        System.out.println(day);
        System.out.println(day.compareTo(Weekday.WEDNESDAY));
        System.out.println(day.ordinal());
    }
}
```

```
<terminated> TestWeekday [Java Application]
SATURDAY
3
5
```

- JavaDoc:  
„Compares this enum with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. Enum constants are only comparable to other enum constants of the same enum type. The natural order implemented by this method is the order in which the constants are declared.“

```
public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}

Weekday day = Weekday.SATURDAY;
System.out.println(day);
System.out.println(day.compareTo(Weekday.WEDNESDAY));
```

+3



- JavaDoc:  
„Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero). Most programmers will have no use for this method.“

```
System.out.println(day.ordinal());
```

```
public enum Weekday {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;  
}
```

```
public class TestWeekday {  
    public static void main(String[] args) {  
        Weekday day = Weekday.SATURDAY;  
        switch (day) { // breaks absichtlich weggelassen!  
            case MONDAY: // nicht Weekday.MONDAY!  
            case TUESDAY:  
            case WEDNESDAY:  
            case THURSDAY:  
            case FRIDAY:  
                System.out.println("arbeiten... lernen... schlafen...");  
                break;  
            case SATURDAY:  
            case SUNDAY:  
                System.out.println("Wochenende: Party!");  
                break;  
        }  
    }  
}
```

```
public enum RechtsformenEnum {  
    GmbH, AG, KG, OHG, GbR, VVaG, Person, Ltd;  
  
    public static RechtsformenEnum vonZahl(int zahl) {  
        if ((zahl < 0) || (zahl > RechtsformenEnum.values().length))  
            throw new IndexOutOfBoundsException("Ungültige Zahl");  
        return RechtsformenEnum.values()[zahl];  
    }  
}
```

```
public abstract class GeschaeftsPartner {  
    private String name;  
    private String anschrift;  
    private String anrede; // besser: enum  
    private RechtsformenEnum rechtsform;  
  
    public GeschaeftsPartner(String name, String anschrift,  
        String anrede, RechtsformenEnum rechtsform) {  
        setName(name);  
        setAnschrift(anschrift);  
        setAnrede(anrede);  
        setRechtsform(rechtsform);  
    }  
  
    public RechtsformenEnum getRechtsform() {  
        return rechtsform;  
    }  
  
    public void setRechtsform(RechtsformenEnum rechtsform) {  
        this.rechtsform = rechtsform;  
    }  
}
```

```
public class TestFachlogik {  
    public static void main(String[] args) throws Exception {  
        VerwaltungAkte VA=VerwaltungAkte.getInstance();  
        Glaebiger g1=new Glaebiger(1,"Dopatka","Schmiedeweg 37",  
            "Herr Dr.",RechtsformenEnum.Person);  
        VA.addAkte(g1,78,"Müller","Pleiteweg 3",7,"23.05.2009",2000);  
        VA.addAkte(g1,565,"Meier","Hauptstrasse 234",8,"23.10.2009",100);  
    }  
}
```

