Einführung in die Programmiersprache Java II

Inhalt

- Beziehungen
- Method Dispatching und Parameterübergabe
- Interfaces
- Abstrakte Klassen

Klassen und Objekte

- Jedes Objekt gehört zu einer Klasse
- Die Attribute und Methoden der Objekte werden in der zugehörigen Klasse definiert
- Alle Objekte einer Klasse besitzen dasselbe Verhalten
  - sie besitzen dieselbe Implementierung der Methoden
  - teoretisch!
- Objekte einer Klasse werden auch als Instanzen bezeichnet

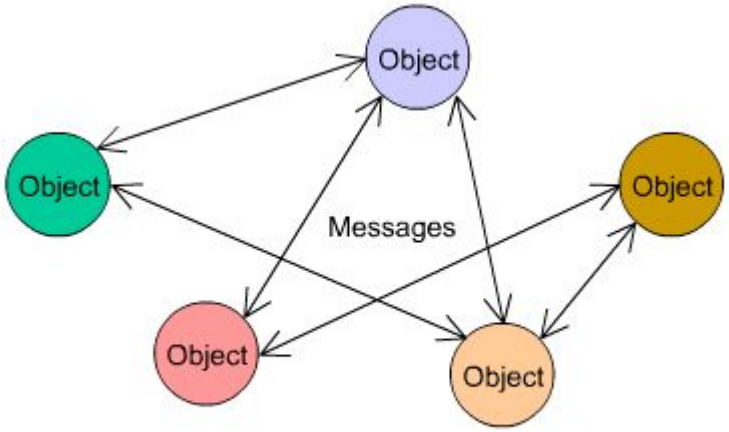
Erzeugung von Objekten

- Speicherplatz wird für das Objekt auf dem Heap allokiert
  - abhängig von den Datentypen der Attribute
- Attribute und statische Variablen sind initialisiert
  - lokale Variablen muss man explizit initialisieren
- Bei der Erzeugung wird ein Konstruktor aufgerufen, der die Initialisierung des Objekts vornimmt
- Werte werden den Attributen zugeordnet, wodurch der Zustand des Objekts definiert wird
  - entweder definiert durch den Konstruktor
  - oder Default-Werte
    - `0/0.0` für Zahlen, `false` für bool
    - `null` für Arrays, Referenzen

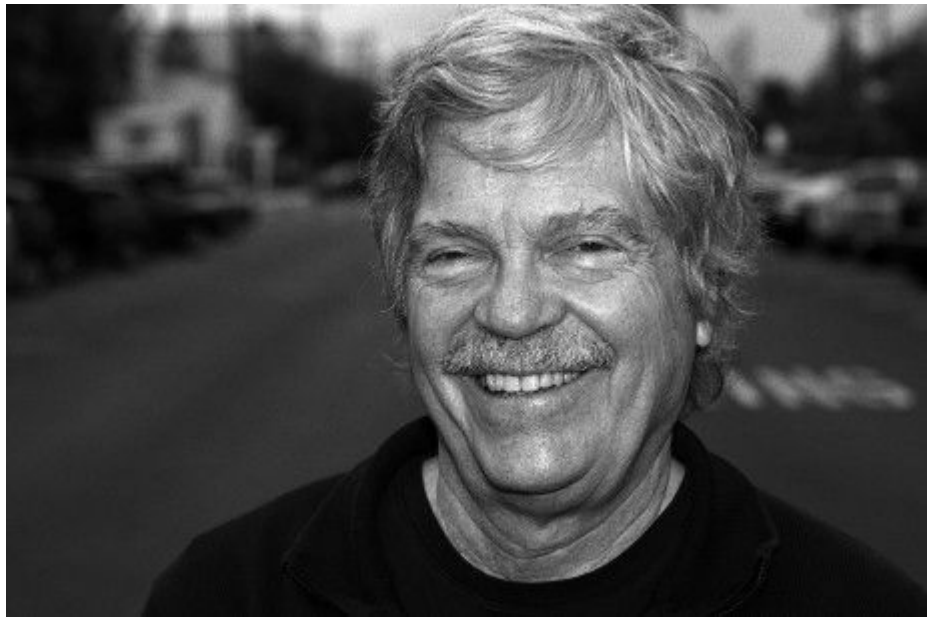
Messages//Objects

- Die Kommunikation beruht auf dem Versenden von Nachrichten zu Empfängern
- Nachrichtenformen sind u. a. der Funktionsaufruf

- " I invented the term Object-Oriented, and I can tell you I did not have C++ in mind."
- "OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."



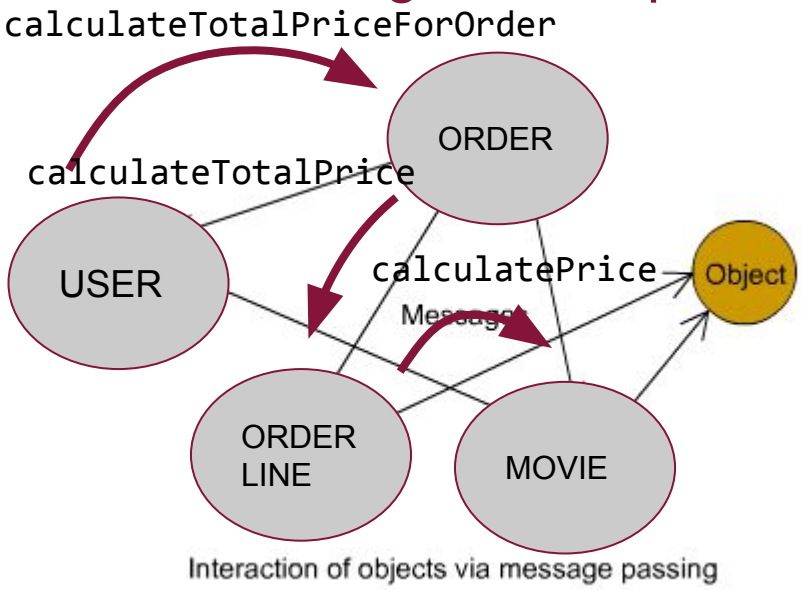
Interaction of objects via message passing



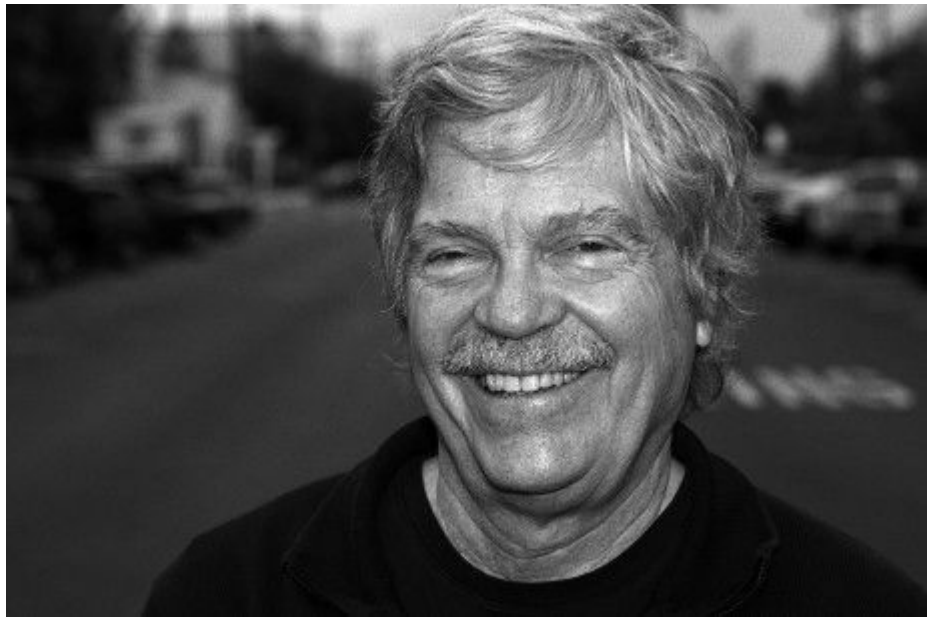
Messages//Objects//Seminar 2

- Die Kommunikation beruht auf dem Versenden von Nachrichten zu Empfängern
- Nachrichtenformen sind u. a. der Funktionsaufruf

- " I invented the term Object-Oriented, and I can tell you I did not have C++ in mind."
- "OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."

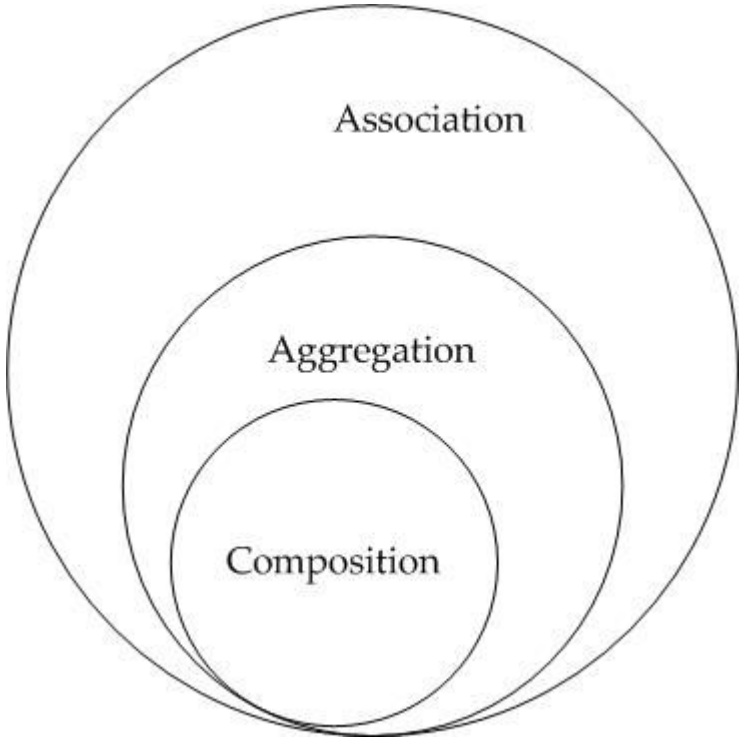


Interaction of objects via message passing



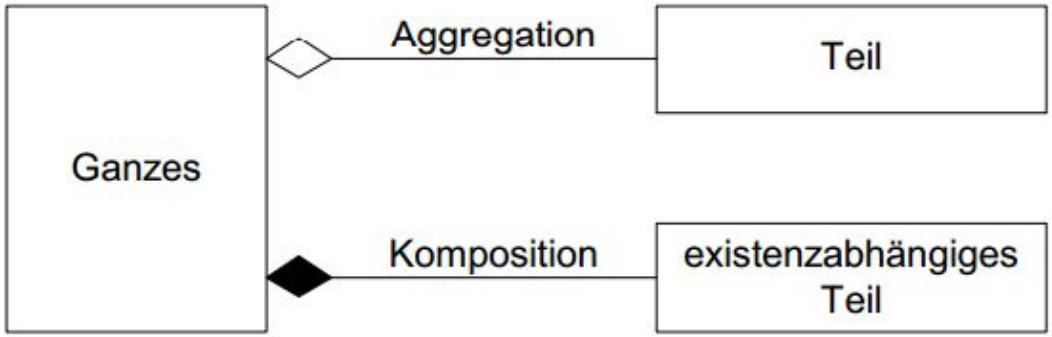
Beziehungen zwischen Objekten

- zwei verschiedene Ansätze
- Vererbung
  - starke Abhängigkeit
- Assoziation
  - beschreibt eine Abhängigkeit
  - Aggregation
  - Komposition



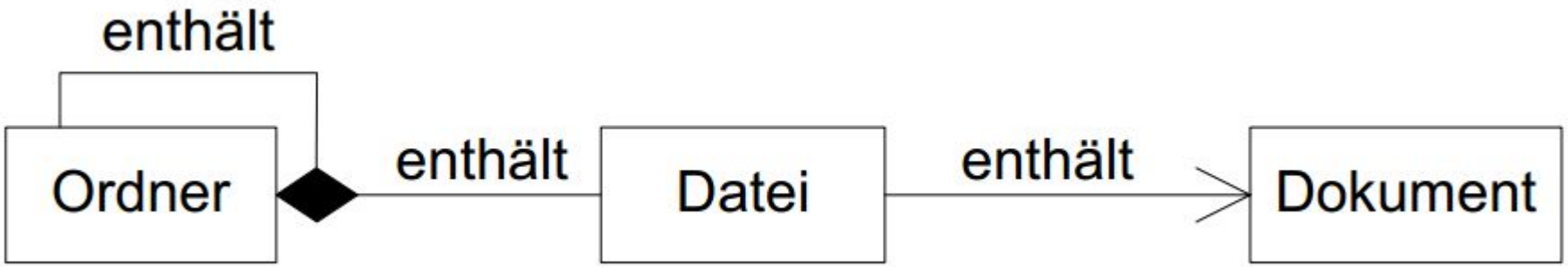
Aggregation und Komposition

- Modelliert "has - a" – Relationen
- ein Teil gehört zum Ganzen
- eine spezielle Art der Aggregation ist Komposition, bei der die Teile vom Ganzen existenzabhängig sind
- ein Teil kann also nur zu einem einzigen Ganzen angehören
  - die Lebensdauer dieses Teils wird durch die des Ganzen bestimmt



Komposition

- Es ist nicht immer einfach zu entscheiden, ob eine Aggregation oder Komposition vorliegt
- Eine Datei ist existenzabhängig vom Ordner, in dem diese Datei gespeichert ist.
- Wird der Ordner gelöscht, so werden auch alle darin befindlichen Dateien gelöscht



Vererbung

- Modelliert "is - a" – Relationen
- die Basisklasse definiert eine **Schnittstelle**
  - abgeleitete Klassen implementieren die Methoden der Schnittstelle spezifisch
- Besonders nützlich für GUI - Programmierung: Erbe von GUI - Klasse, füge eigene Methoden dazu
- Wiederverwendbarkeit: Muss Code der Basisklasse nicht neu programmieren

Exkurs: Implementierung in Java

Komposition und Vererbung: Wiederverwendung

Vererbung

- Hierarchische Beziehung, bei der eine abgeleitete Klasse eine spezialisierte Version der Superklasse darstellt. Sie wird oft verwendet, um allgemeine Eigenschaften und Verhaltensweisen wiederzuverwenden und zu erweitern.

- Kann zu einer stärkeren Kopplung zwischen Klassen führen, da Änderungen an der Superklasse Auswirkungen auf alle abgeleiteten Klassen haben können. Dies kann zu Problemen führen, wenn sich die Anforderungen ändern.

- Ermöglicht die Wiederverwendung von Code, da abgeleitete Klassen Eigenschaften und Methoden der Superklasse erben.

I just wanted a banana!

You wanted a banana but what you got was a gorilla holding the banana and the entire jungle

Komposition und Vererbung: Wiederverwendung

Komposition

- Aggregierte Beziehung, bei der eine Klasse andere Klassen als Teile enthält, um ihre Funktionalität zu erzielen. Dies wird verwendet, um modularere und lose gekoppelte Systeme zu erstellen.

- Bietet in der Regel mehr Flexibilität, da Klassen unabhängiger voneinander sind. Änderungen in einer Klasse beeinflussen normalerweise nicht direkt andere Klassen.

- Ermöglicht ebenfalls die Wiederverwendung von Code, aber auf eine flexiblere Weise, indem Sie Klassen so zusammensetzen, dass sie bereits vorhandene Funktionalität nutzen.

Exkurs: Implementierung in Java

die folgende Klasse existiert

Fax-Klasse

Methoden:
- scanDocument()
- sendDocument()
- receiveDocument()

ich brauche eine neue Klasse

MultiFunctionalPrinter-Klasse

Methoden:
- scanDocument()
- printDocument()
- EmailDocument()

Überschreiben von Methoden

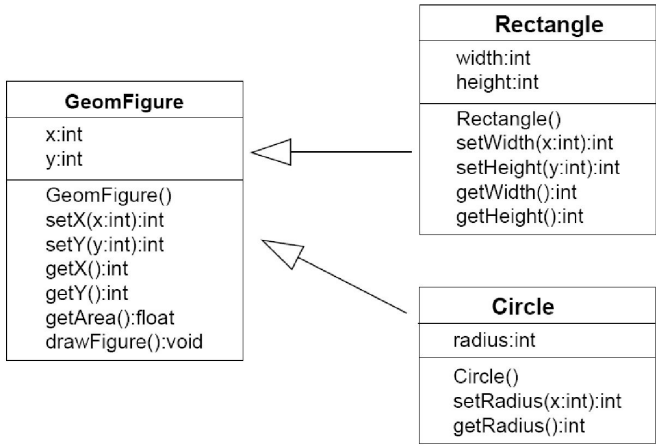
- Hätte eine Methode der Subklasse **die gleiche Signatur** einer Methode der Superklasse
  - wird die Superklassenmethode überschrieben (**overriding**)

- `GeomFigure` definiert eine Methode `toString()`:

```
String tmp="[“ + myX + ",“ + myY + "]";
return tmp;
```

- `Circle` überschreibt die Methode `toString()`:

```
String tmp=super.toString();
tmp=tmp+" "+radius;
return tmp;
```



- man muss getters und super verwenden, da die Attribut der Basisklasse privat sind

Überschreiben von Methoden


```
class Tier {
    void speak() {

System.out.println("___");
    }
}
class Hund extends Tier {
    @Override
    void speak() {

System.out.println("ham");
    }
}
class Katze extends Tier {
    @Override
    void speak() {

System.out.println("miau");
    }
}
```

```
public class Überschreiben {
    public static void
main(String[] args) {
        Tier tier = new Tier();
        Tier bob = new Hund();
        Tier boba = new Katze();

        tier.speak();
           // Ausgabe: ___
       bob.speak();
           // Ausgabe: ham
        boba.speak();
           // Ausgabe: miau
    }
}
```

Polymorphismus

- Beim Polymorphismus werden die Methodenaufrufe zur Laufzeit (Dynamisch) aufgelöst.
- Dies erfolgt durch das Überschreiben von Methoden in abgeleiteten Klassen (Subklassen)
  - wobei die Subklasse eine spezifische bzw. andere Implementierung der Methode bereitstellt.
- Die Auswahl der Methode erfolgt zur Laufzeit basierend auf dem tatsächlichen (dynamischen) Typ des Objekts.
- die Basisklasse muss nicht unbedingt abstrakt sein
- alle Methoden sind automatisch virtual

Überladen von Methoden

- Falls mehrere Methoden mit demselben Namen aber mit **unterschiedlichen Signaturen** definiert sind
  - so spricht man von Überladen (**overloading**)

- Beispiel:

```
void println () ...
void println (String s) ...
void println (int i) ...
```

Überladen von Methoden

```
public class Überladen {

  public int quadrat(int num) {
        return num * num;
  }

  public double quadrat(double num){
        return num * num;
  }

  public static void main(String[] args) {
    Überladen beispiel = new Überladen ();

    int r1 = beispiel.quadrat(5);

    double r2 = beispiel.quadrat(3.5);

  }
}
```

Ein “komplettes” Beispiel

```
class Animal {
    public void speak() {System.out.println("___");};
}

class Dog extends Animal {
    public void speak() {System.out.println("ham");}
}

class Cat extends Animal {
    public void speak() {System.out.println("miau");}
}

class Talk{
    public void speak(Dog d) {
        d.speak();
    }

    public void speak(Cat c) {
        c.speak();
    }
}

public class Main {
    public static void main(String args[]) {
        Animal animal = new Dog();
        new Talk().speak(animal);
    }
}
```

Ein “komplettes” Beispiel

```
class Animal {
    public void speak() {System.out.println("___");};
}

class Dog extends Animal {
    public void speak() {System.out.println("ham");}
}

class Cat extends Animal {
    public void speak() {System.out.println("miau");}
}

class Talk{
    public void speak(Dog d) {
        d.speak();
    }

    public void speak(Cat c) {
        c.speak();
    }
}

public class Main {
    public static void main(String args[]) {
        Animal animal = new Dog();
        new Talk().speak(animal); //ERROR!! (argument mismatch; Animal cannot be converted to Dog)
    }
}
```

Ein “komplettes” Beispiel

```
class Animal {
    public void speak() {System.out.println("___");};
}

class Dog extends Animal {
    public void speak() {System.out.println("ham");}
}

class Cat extends Animal {
    public void speak() {System.out.println("miau");}
}

class Talk{
    public void speak(Dog d) {
        d.speak();
    }

    public void speak(Cat c) {
        c.speak();
    }
}

public class Main {
    public static void main(String args[]) {
        new Talk().speak(new Dog());
    }
}
```

Methode Auswahl

- Java ist eine sogenannte Single-Dispatch-Sprache
  - Dispatching bezieht sich auf die Art und Weise, wie die Auswahl der Methode zur Laufzeit erfolgt.

- Single-Dispatch-Sprache: die Auswahl einer Methode zur Laufzeit basierend auf dem dynamischen Typ (tatsächlichen Typ) eines einzelnen Objekts durchgeführt.
  - das bedeutet, dass die Wahl der Methode zur Laufzeit nur von einem Parameter (dem Objekt selbst) abhängt
  - dem Objekt, auf dem die Methode aufgerufen wird.

- wenn eine Methode aufrufen wird, die in einer Basisklasse definiert ist, aber in einer abgeleiteten Klasse überschrieben wurde, wird die Methode des tatsächlichen Objekttyps zur Laufzeit aufgerufen.

Subtyping

- Sei A die Basisklasse und B von A abgeleitet
- Typkonvertierung von B nach A ist eine "up – cast" Konvertierung
- Konsequenz: Variablen von Typ A können Objekte von Typ B enthalten.
  - A a = new B();
  - Methoden mit Parametertyp A können Argumente vom Typ B nehmen
- Variable hat immer noch Typ A, d.h. zur Compilezeit nur Methoden und Felder für Typ A zulässig
- Zur Laufzeit werden aber die Methoden bzw. Implementierungen von Klasse B verwendet

Subtyping

```
class Kreis {
    int x, y, r;
    int area() { return Math.PI*r*r); }
}
class BunterKreis extends Kreis {int farbe;}
class KreisTest {
    public static void main (String[] arr) {
        Kreis k = new BunterKreis();
        k.r = 3; //OK
        k.farbe = 6; //Fehler
        System.out.println(k.area());
        System.out.println("Radius= " + k.r);
    }
}
```

Super

- Die Anweisung `super` mit den Parametern `x, y` und `radius` ruft den Konstruktor der Basisklasse auf
- mit `super` kann man auch Methoden der Basisklasse aufrufen
  - `super.area();`

```
class BunterKreis extends Kreis {
   int farbe;

   public BunterKreis (int x, int y, int r, int f) {
      super (x,y,r);
      farbe = f;
   }
}
```

Vererbung und Konstruktoren

- Konstruktoren werden nicht vererbt
- Konstruktoren der Basisklasse können mit `super(...)` aufgerufen werden
- Falls Basisklasse mehrere Konstruktoren hat, ruft `super(...)` den mit den richtigen Parametertypen auf
- Ein solcher Aufruf von `super(...)` muss die erste Zeile der Konstruktordefinition sein

Zugriffskontrolle

- Kapselung: Benutzer einer Klasse sollen Implementierungsdetails nicht sehen können
- Zugriffskontrolle: Verstecke Members so, dass Zugriff aus anderen Klassen ein syntaktischer Fehler ist
- der Compiler verhindert also, dass fremde Programme auf Implementierung zugreifen
- public, private, protected...
  - details später
- Normalerweise sind Felder private, Methoden public oder private

Exkurs: Kapselung in Java

Lebensdauer von Variablen

- Lebensdauer eines Feldes ist die Lebensdauer des Objekts, zu dem es gehört
  - wird erzeugt, wenn das Objekt erzeugt wird
  - ein Exemplar pro Objekt der Klasse
- Lebensdauer einer lokalen Variable ist ein einziger Aufruf der Methode
  - wird erzeugt, wenn die Methode aufgerufen wird
  - ein Exemplar pro Methodenaufruf
- Parameterübergabe in Java
  - in C++: `&`, `*`
  - pass-by-value?
  - pass-by-reference?

Parameterübergabe

```
class Parameter{
    static void exchange(int x, int y){
        int tmp = x;
        x = y;
        y = tmp;
    }

    public static void main(String[] args) {
        int x=2, y=4;
        exchange(x,y);
        System.out.println("x=" + x +" y=" + y);
    }
}
```

Parameterübergabe

```
class B{
    int val;
    public B(int x){
        this.val=x;
    }
    public String toString(){
        return ""+val;
    }
    static void exchange(B x, B y){
        B tmp = x;
        x = y;
        y = tmp;
        System.out.println("[exchange
        B] x="+x+" y="+y);
    }
```

```
public static void main(String[]
args) {
    B bx=new B(2);
    B by=new B(4);
    System.out.println("bx="+bx+"
    by="+by);
    exchangeData(bx,by);
    System.out.println("bx="+bx+"
    by="+by);
}
```

Parameterübergabe

```
class B{
    int val;
    public B(int x){
        this.val=x;
    }
    public String toString(){
        return ""+val;
    }
    static void exchangeData(B x, B
    y){
        int tmp=x.val;
        x.val=y.val;
        y.val=tmp;
        System.out.println("[exchangeD
        ata] x="+x+" y="+y);
    }
}
```

```
public static void main(String[]
args) {
    B bx=new B(2);
    B by=new B(4);
    System.out.println("bx="+bx+"
    by="+by);
    exchangeData(bx,by);
    System.out.println("bx="+bx+"
    by="+by);
}
```

Parameterübergabe

- primitive Typen (boolean, int, long, double)
  - pass-by-value
  - eine Kopie wird auf dem Stack erstellt

- reference Typen
  - pass-by-value
  - eine Kopie auf dem Stack, aber der Inhalt (Attribute für Objekte, Positionen für Arrays) kann verändert werden, falls die Methode Zugriff hat

- Dieses Verhalten kann nicht verändert sein

Statische Members

- Statische Felder und Methoden gehören nicht zu Objekten, sondern zur Klasse als Ganzes
- Statisches Feld wird erzeugt, sobald die Klasse geladen wird
- Statische Methoden dürfen nur statische Members (und lokale Variablen) verwenden

Statische Members

- Deklaration von statischen Members: Schlüsselwort `static`

- Verwendung von statischen Members:
  - `KlassenName.membername`
  - `objekt.membername`
    - Verwendung des Members `membername` in Compilezeit - Klasse von objekt
  - `membername`
    - Verwendung des (evtl. ererbten) Members `membername` in aktueller Klasse.

Statische Members

```
class Zahlengenerator {
   private static int next = 0;
   public static int getNext() { return next++;}
}

class GeneratorTest {
   public static void main (String[] arr) {
      System.out.println(Zahlengenerator.getNext());
      System.out.println(Zahlengenerator.getNext());
      System.out.println(Zahlengenerator.getNext());
   }
}
```

Statische Members

```
public static long MAX=2000;

public class Natural {
   public static long MAX;
   static {
      MAX=2000;
  }
}

private static long counter; //0
```

Statische Klassen

- klassen können auch `static` sein
  - aber nur interne Klasse
- In Java kann man statische Blöcke definieren
  - Der Block wird nur einmal ausgeführt, wenn die Klasse in den Speicher geladen wird

```
class Test {
    static int i;
    int j;

    // start of static block
    static {
        i = 10;
        System.out.println("static block called");
    }
    // end of static block
}
```

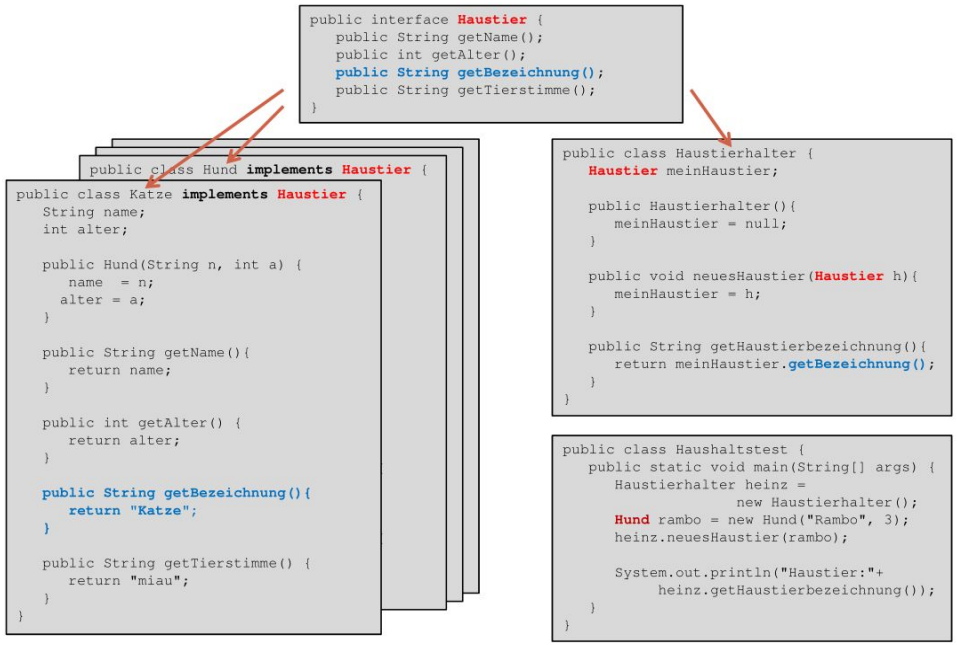
Interfaces

- Interfaces definieren ein Verhalten, das in jeder abgeleiteten Klasse implementiert werden muss
- eine solche Unterklasse implementiert die Interface
- bieten die Möglichkeit, einheitliche Schnittstelle für Klassen zu definieren, die
  - später oder/und
  - durch andere Programmierer implementiert werden
- pur abstrakte Klassen in C++

Abstrakte Klassen

- definieren genauso wie Interfaces ein einheitliches Interface für alle abgeleiteten Klassen

- möglich einzelne Methoden bereits in der abstrakten Klasse zu implementieren
  - und Instanzvariablen zu deklarieren



```
public interface Haustier {
   public String getName();
   public int getAlter();
   public String getBezeichnung();
   public String getTierstimme();
}
```

```
public class Hund implements Haustier {
```

```
public class Katze implements Haustier {
   String name;
   int alter;

   public Hund(String n, int a) {
      name  = n;
     alter = a;
   }

   public String getName(){
      return name;
   }

   public int getAlter() {
      return alter;
   }

   public String getBezeichnung(){
      return "Katze";
   }

   public String getTierstimme() {
      return "miau";
   }
}
```

```
public class Haustierhalter {
   Haustier meinHaustier;

   public Haustierhalter(){
      meinHaustier = null;
   }

   public void neuesHaustier(Haustier h){
      meinHaustier = h;
   }

   public String getHaustierbezeichnung(){
      return meinHaustier.getBezeichnung();
   }
}
```

```
public class Haushaltstest {
   public static void main(String[] args) {
      Haustierhalter heinz =
                  new Haustierhalter();
      Hund rambo = new Hund("Rambo", 3);
      heinz.neuesHaustier(rambo);

      System.out.println("Haustier:"+
           heinz.getHaustierbezeichnung());
   }
}
```

Abstrakte Klassen

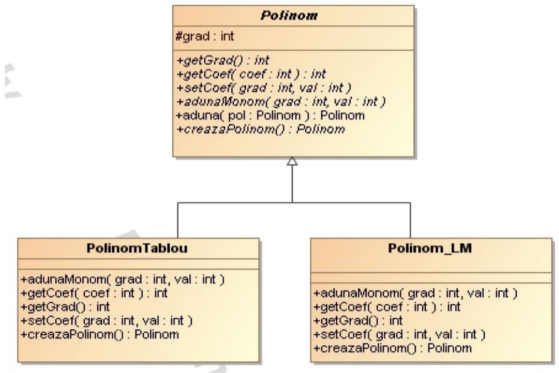
- eine Klasse darf mehrere Interfaces implementieren
  - aber nur eine Klasse vererben
- definiert mit `abstract`
- Instanzen der Klasse können nicht erzeugt werden
- muss nicht unbedingt eine abstrakte Methode bereitstellen
- Alle abstrakten Methoden müssen in den nicht abstrakten Subklassen implementiert werden

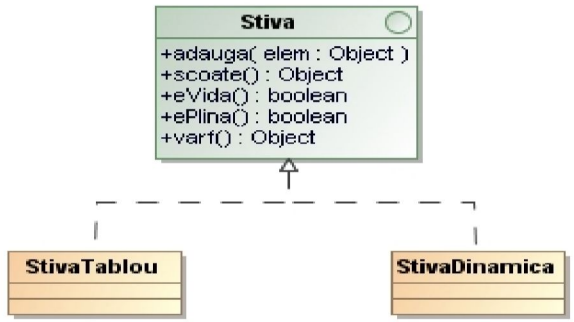
```
abstract class KlassenName {
    void aNormalMethod(int a) {...}
    abstract void aAbstractMethod(int b);
}
```

Abstrakte Klasse vs Interfaces

Public, protected, private Methoden	nur public Methoden
Hat Attribute	Darf nur statische bzw. finale Attribute enthalten
Hat Konstruktoren	Keine Konstruktoren
Darf ohne abstrakte Methoden definiert sein	Darf ohne Methoden definiert sein

Darf nicht instanziert sein





Packages

```
System.out.println(“bla”);
```

- Stellt Klassen und Interfaces zusammen
- Namespace Verwaltung
- package `java.lang` enthält
  - `System, Integer, String` usw.
- Definiert mit der Verwendung von `package`

```
//structuri/Stiva.java
package structuri;
public interface Stiva{
//...
}
```

Packages

- `package` muss die erste Anweisung in einer Java Datei sein
- die Lista Datei wird in einem Folder structuri gespeichert

```
//structuri/liste/Lista.java
package structuri.liste;
public interface Lista{
//...
}
```

Packages

- `import`
  - Zugriff auf Klassen in einem Package
- `import pac1.[pac2.[...]]Klasse;`
  - eine Klasse
- `import pac1.[pac2.[...]]*;`
  - alle Klassen aber nicht alle subpackages
- eine Java Datei darf mehrere import Anweisungen enthalten
- sollen bevor der Klassendefinitionen stehen

Naming Collisions

```
// unu/A.java
package unu;
public class A{
//…
}
```

```
// doi/A.java
package doi;
public class A{
//...
}
```

```
//Test.java
import unu.*;
import doi.*;

public class Test{
    public static void main(String[] args){
        A a=new A();
        unu.A a1=new unu.A();
        doi.A a2=new doi.A();
    }
}
```

Static import

- ab Java 1.5
- `import static pac1.[pac2.[. ...]]Class.static_mem;`
- `import static pac1.[pac2.[...]]Class.*;`
- erlaubt die Verwendung der statischen Attribute einer Klasse `Class` ohne den Klassenname

Static import

```
package utile;
public class EncodeUtils {
    public static String encode(String txt){...}
    public static String decode(String txt){...}
}

//Test.java
import static utile.EncodeUtils.*;
public class Test {
    public static void main(String[] args) {
        String txt="aaa";
        String enct=encode(txt);
        String dect=decode(enct);
        //...
    }
}
```

Default Package

- Jede Klasse gehört zu einem Package
- Falls eine `.java` Datei die `package` Anweisung nicht enthält, gehören alle Klassen zu einem Anonymus/Default Package

```
//Persoana.java
public class Persoana{...}

//Complex.java
public class Complex{...}
```

```
//Test.java
public class Test{
    public static void main(){
        Persoana p=new Persoana();
        Complex c=new Complex();
        //...
    }
}
```

Zugriffsmodifikatoren II

- private, public, protected, default

	Default	private	protected	public
Zugang in der Klasse	Yes	Yes	Yes	Yes
Zugang in Package durch eine Subklasse	Yes	No	Yes	Yes
Zugang in Package außer der Klasse und nicht in einer Subklasse	Yes	No	Yes	Yes
Zugang außer Package durch eine Subklasse	No	No	Yes	Yes
Zugang außer Package außer der Klasse und nicht in einer Subklasse	No	No	No	Yes