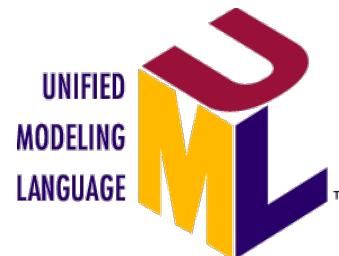
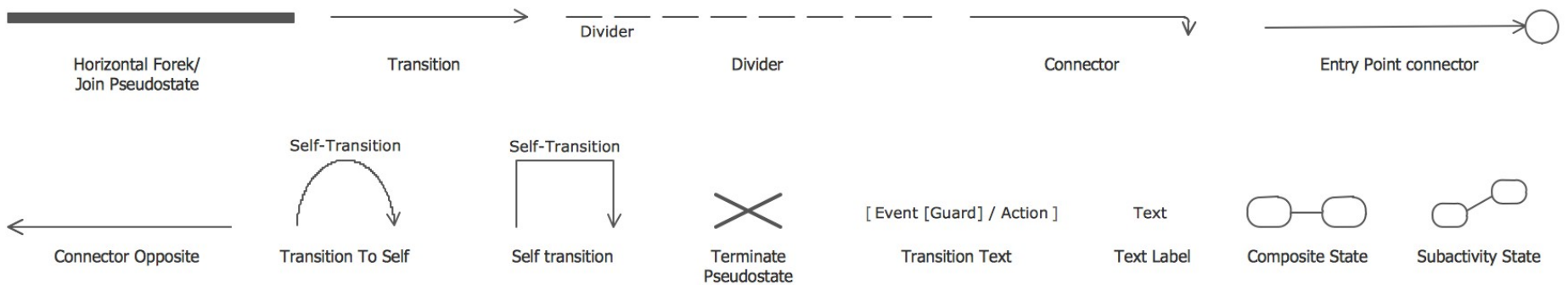
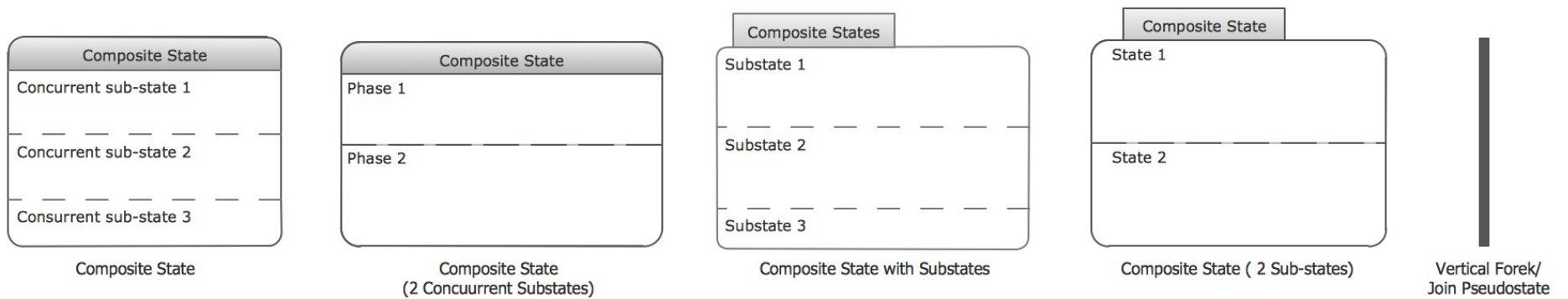
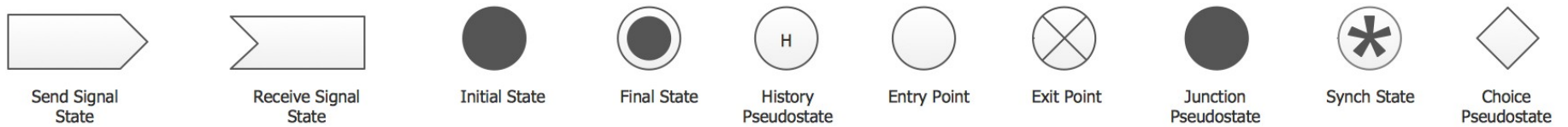
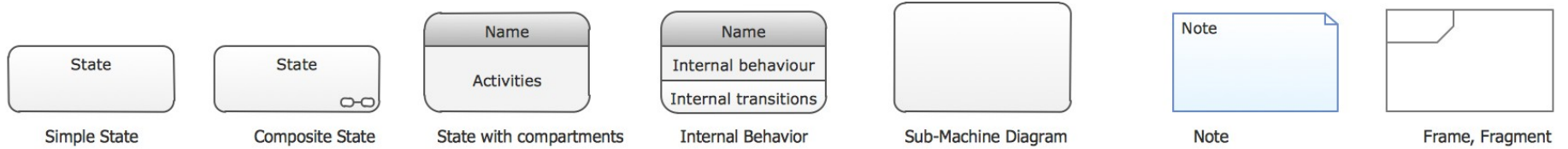


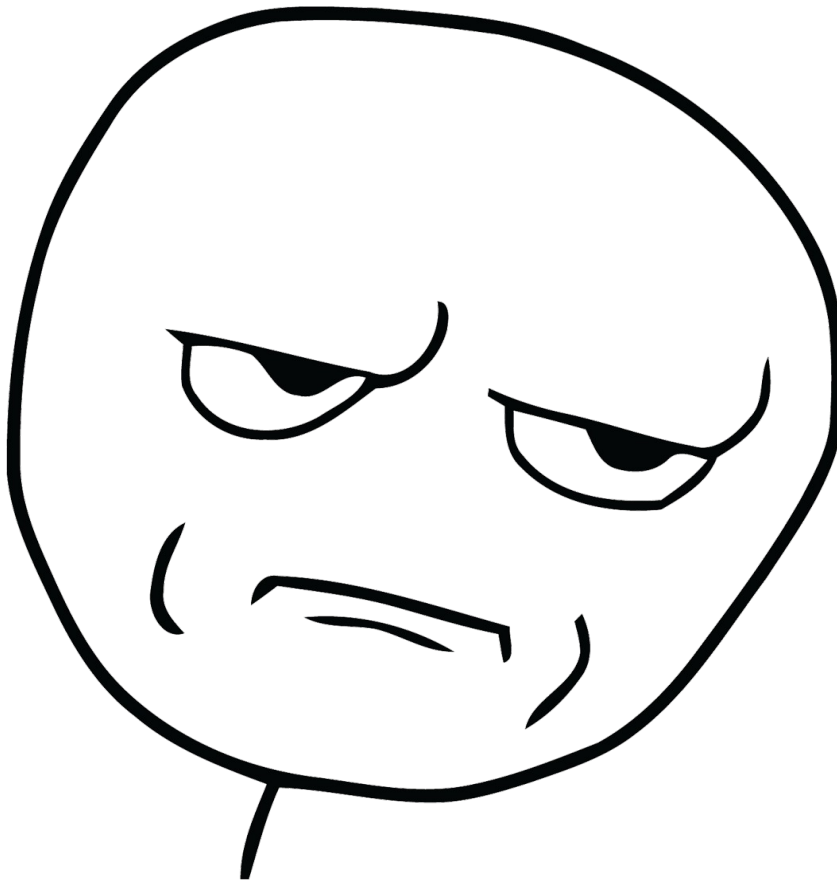
Einführung in die Programmiersprache Java II







???????????? UML



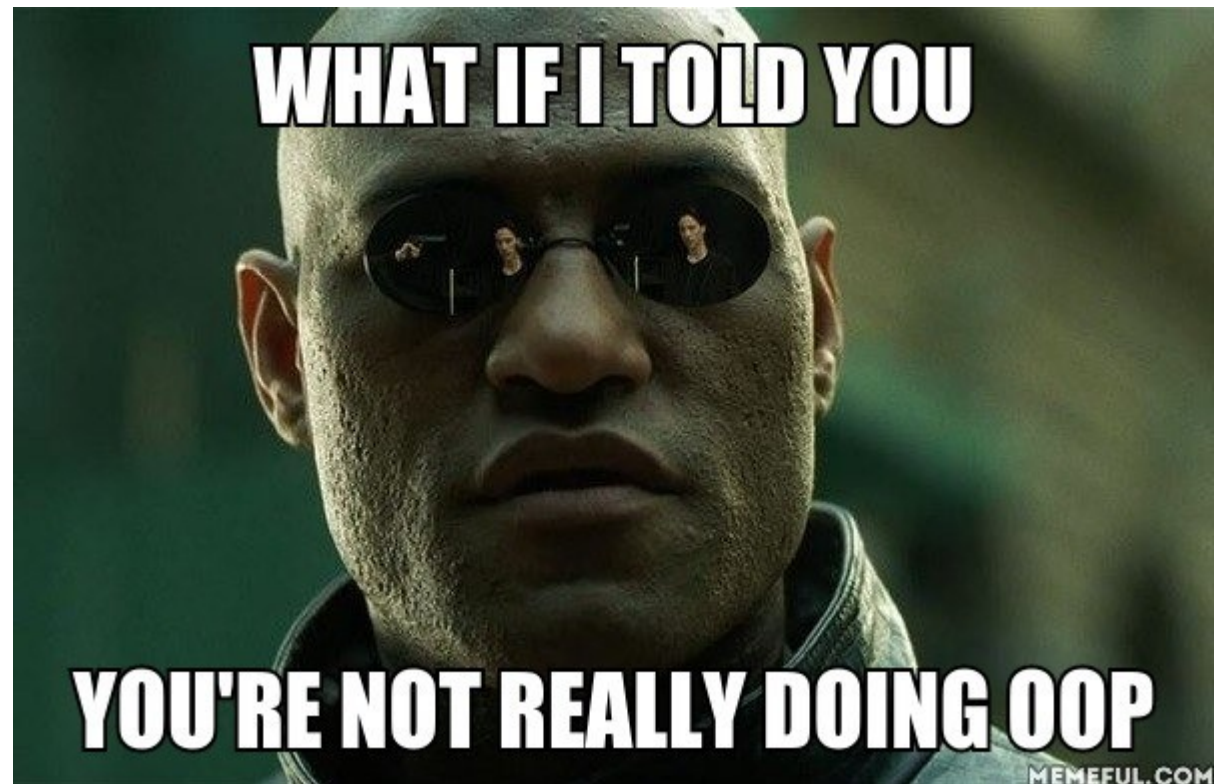
REALLY..????



OOP

"Object oriented programming is bad"

- professional retard





90s...



UML Entwicklungsziele

- verschiedenen existierenden objektorientierten Modellierungsmethoden zu vereinheitlichen
- Ermöglichung der Modellierung großer Systeme
- Schaffung einer Modellierungssprache, die sowohl von Menschen als auch Maschinen verstanden wird

Was kann UML?

- Spezifikation
- Visualisierung
- Entwurf der Architektur
- Simulation und Test
- Dokumentation

Was ist UML nicht?

- kein Vorgehensmodell
- kein Modellierungswerkzeug
- keine Modellierungsrichtlinien
- keine Programmiersprache

Klassendiagramm

- zeigt Klassen und ihre Beziehungen untereinander
- Visualisierung und Modellierung der statischen Struktur eines Systems



Klassen und Objekte (Instanzen)

- Jedes Objekt gehört einer Klasse an
- Die Attribute und Methoden der Objekte werden in der zugehörigen Klasse definiert
- Alle Objekte einer Klasse besitzen dasselbe Verhalten, da sie dieselbe Implementierung der Methoden besitzen
- Objekte einer Klasse werden auch als Instanzen (instance) oder Exemplare bezeichnet

Erzeugung von Objekten

- Es wird Speicherplatz für das Objekt auf dem Heap allokiert (abhängig von den Datentypen der Attribute)
- Bei der Erzeugung wird ein Konstruktor (spezielle Methode) ausgeführt, der die Initialisierung des Objekts vornimmt
- Es werden den Attributen Werte zugeordnet, wodurch der Zustand des Objekts definiert wird
 - entweder definiert durch den Konstruktor
 - oder Default-Werte (0 für Zahlen, null für Referenzen, etc.)

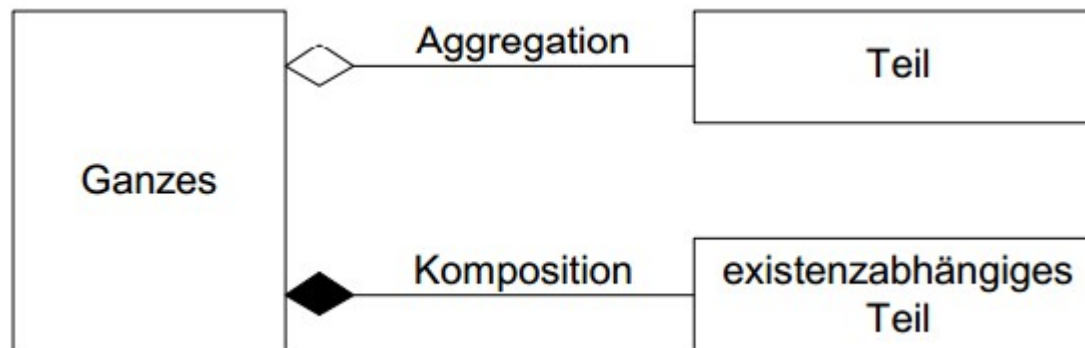
Messages//Objects

- " 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."



Komposition

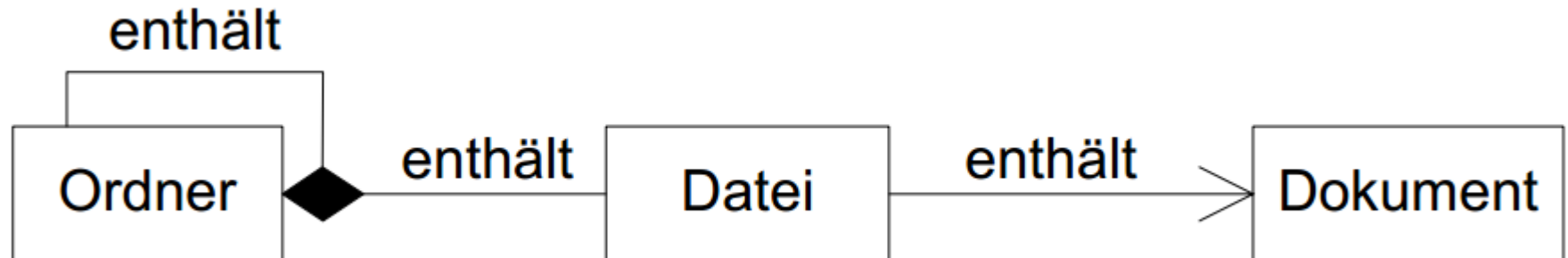
- Eine spezielle Art der Aggregation ist die Komposition, bei der die Teile vom Ganzen existenzabhängig sind
- Ein Teil kann also nur zu einem einzigen Ganzen angehören und die Lebensdauer dieses Teils wird durch die des Ganzen bestimmt



Komposition

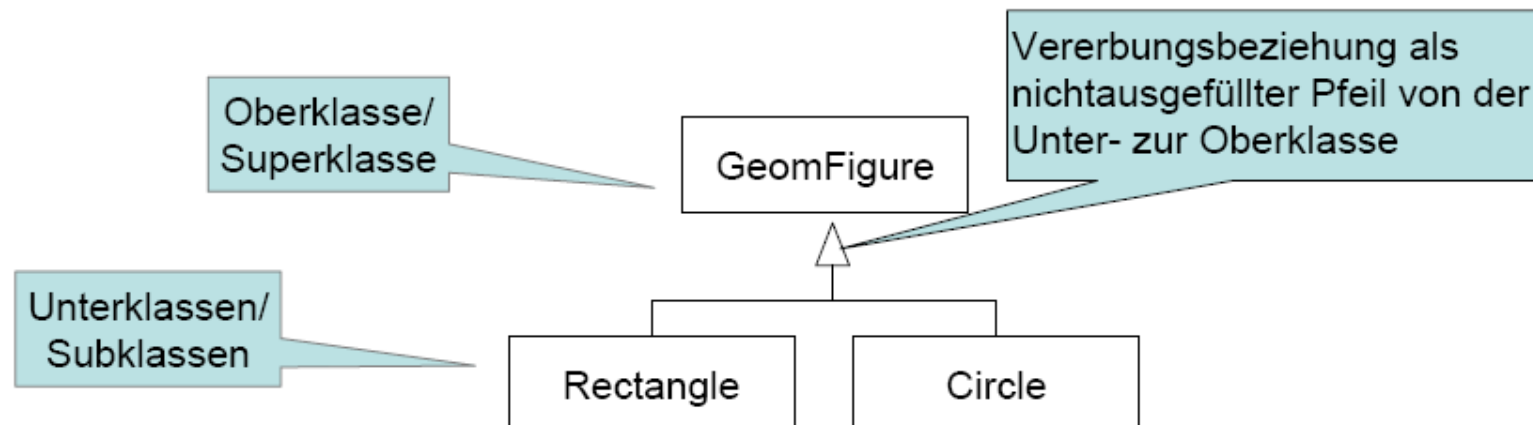
Es ist nicht immer einfach zu entscheiden, ob eine Aggregation oder Komposition vorliegt

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

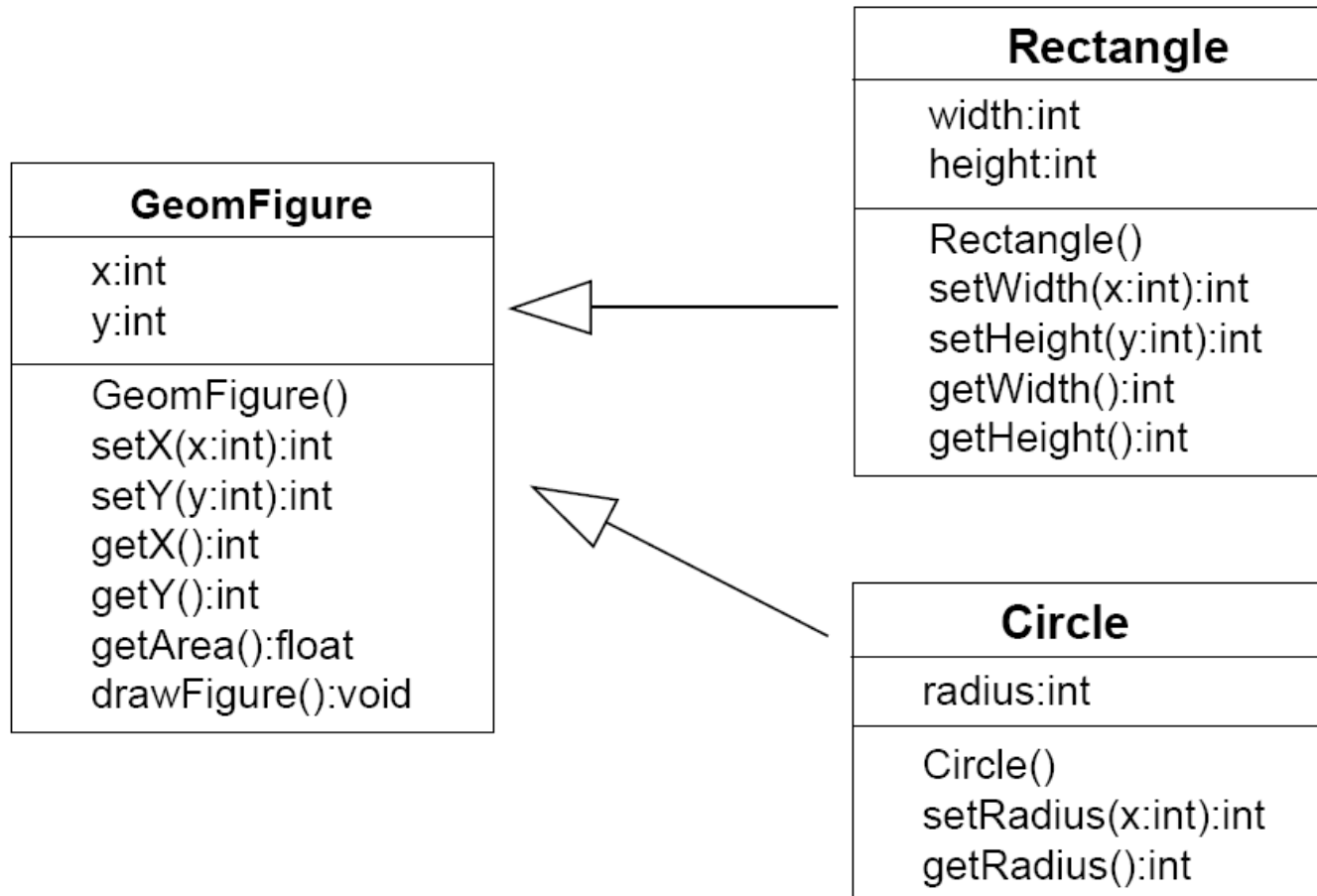
- Ist ein Konzept, wodurch Attribute und Methoden der Oberklasse auch den Unterklassen zugänglich gemacht werden
- Vererbungsbeziehungen zwischen Klassen stellen Generalisierungen



Vererbung

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

Vererbung



Überschreiben von Methoden

- Hat eine Methode einer Subklasse **die gleiche Signatur** einer Methode der Superklasse, so wird die Superklassenmethode überschrieben (**overriding**)
- **GeomFigure** definiert eine Methode `toString()`:

```
String tmp="[" + myX + ", " + myY + "]" ;  
return tmp;
```
- **Square** überschreibt die Methode `toString()`:

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

Überladen von Methoden

- Gibt es mehrere Methoden mit demselben Namen aber mit **unterschiedlichen Signaturen**, so spricht man von Überladen (**overloading**).
- Beispiel:
void println() ...
void println (String s) ...
void println (int i) ...

Subtyping

- Sei A Basisklasse und B von A abgeleitet
- Typkonvertierung von B nach A ist eine erweiternde Typkonvertierung ("up – cast")
- Konsequenz: Variablen von Typ A können Objekte von Typ B enthalten. Methoden mit Parametertyp A können Argumente von 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 Methoden - Implementierungen von Klasse B verwendet

Subtyping

```
1 class Kreis {
2     int x, y, r;
3
4     int flaecheninhalt() {
5         return Math.PI * r * r;
6     }
7 }
8
9 class BunterKreis extends Kreis {
10     int farbe;
11 }
12
13 class KreisTest {
14     public static void main(String[] args) {
15         Kreis k = new BunterKreis();
16         k.r = 3; // das ist ok
17         k.farbe = 6; // Compiler - Fehler
18
19         System.out.println(k.flaechinhalt());
20         System.out.println("Radius = " + k.r);
21     }
22 }
```

Super

- Die Anweisung `super` mit den Parametern `name` und `alter` ruft den Konstruktor der Klasse auf, von der geerbt wird

```
1 public class Spieler extends Person{
2     // Zusätzliche Eigenschaften eines Spielers:
3     private int staerke; // von 1 (schlecht) bis 10 (super)
4     private int torschuss; // von 1 (schlecht) bis 10 (super)
5     private int motivation; // von 1 (schlecht) bis 10 (super)
6     private int tore;
7
8     // Konstruktoren
9     public Spieler(String n, int a, int s, int t, int m){
10         super(n, a);
11         staerke = s;
12         torschuss = t;
13         motivation = m;
14         tore = 0;
15     }
16     //...
17 }
```

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

Sichtbarkeit

- In einem Programm kann es Felder und Variablen mit gleichem Namen geben
- Wie entscheidet der Compiler, welche Definition verwendet wird?
- An jeder Stelle darf nur eine Definition sichtbar sein; die anderen sind verschattet.

Zugriffskontrolle

- Kapselung : Benutzer der Klasse sollen Implementierungsdetails nicht sehen können
- Zugriffskontrolle : Verstecke Members so, dass Zugriff aus anderen Klassen ein syntaktischer Fehler ist
- Schon Compiler verhindert also, dass fremde Programme auf Implementierung zugreifen

Public und Private

- Ein als `public` deklariertes Member ist aus allen Klassen auf der Welt sichtbar
- Ein als `private` deklariertes Member ist nur aus Methoden der Klasse selbst sichtbar
- Ignoriere fürs Erste `protected` und `package` - weite Zugänglichkeit
- Gewöhnt euch an, zwischen `public` und `private` zu unterscheiden!
- Normalerweise sind Felder `private`, Methoden `public` oder `private`

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

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

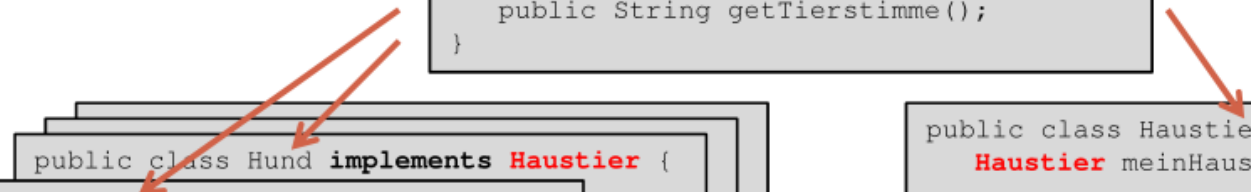
```
1  class Zahlengenerator {  
2  
3      private static int next = 0 ;  
4  
5      public static int getNext() {  
6          return next++;  
7      }  
8  }  
9  
10 class Test {  
11     public static void main(String[] args) {  
12         System.out.println(Zahlengenerator.getNext());  
13         System.out.println(Zahlengenerator.getNext());  
14         System.out.println(Zahlengenerator.getNext());  
15     }  
16 }
```

Interfaces

- Interface bieten in Java ist die Möglichkeit, einheitliche Schnittstelle für Klassen zu definieren, die
 - später oder/und
 - durch andere Programmierer
- implementiert werden
- Interfaces können definiert und implementiert werden

Interfaces

```
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

- Soll genauso wie Interface/implements ein einheitliches Interface für alle abgeleiteten Klassen definieren. Hier ist es aber möglich einzelne Methoden bereits in der abstrakten Klasse zu implementieren und Instanzvariablen zu deklarieren
- Wird in einer Klasse für mindestens eine Methode nur die Schnittstelle definiert (abstrakte Methode), muss die gesamte Klasse als abstrakt definiert werden.

Abstrakte Klassen

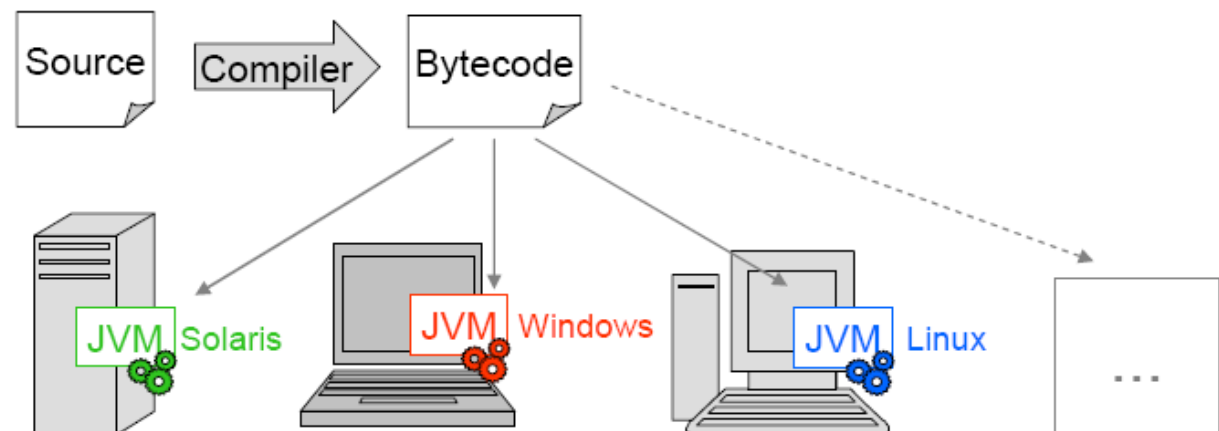
- Die Kennzeichnung einer Klasse als abstrakt verhindert, dass Instanzen dieser Klasse erzeugt werden können.
- Alle abstrakten Methoden müssen in den nicht abstrakten Subklassen implementiert werden.

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

Eigenschaften von Java (1)

Java ist plattformunabhängig

- Java-Code wird mit einem Java-Compiler (javac) in plattformunabhängigen Bytecode überführt
- Bytecode wird von einer **Java Virtual Machine (JVM)** teils interpretiert teils zur Laufzeit in Maschinencode kompiliert (Just-In-Time-Compiler in JVM integriert)
- Für jede unterstützte Plattform muss es eine JVM geben



Eigenschaften von Java (2)

Java wurde auf Sicherheit und Robustheit hin entwickelt:

- Fehleranfällige Sprachkonstrukte fehlen:
 - keine Pointer und Pointerarithmetik
 - keine Destruktoren sondern automatische Garbage Collection
 - keine Mehrfachvererbung*
 - kein Operator-Overloading
- Bytecode Verification
- Exception Handling
- Thread-Sicherheit in die Sprache integriert